

# NetXPTO - LinkPlanner

March 19, 2019

---

# Contents

|   |           |
|---|-----------|
| <b>1 Preface</b>                                      | <b>10</b> |
| <b>2 Introduction</b>                                 | <b>11</b> |
| <b>3 Simulator Structure</b>                          | <b>12</b> |
| 3.1 System . . . . .                                  | 12        |
| 3.2 Blocks . . . . .                                  | 12        |
| 3.3 Signals . . . . .                                 | 12        |
| 3.3.1 Circular Buffer . . . . .                       | 13        |
| 3.4 Log File . . . . .                                | 13        |
| 3.4.1 Introduction . . . . .                          | 13        |
| 3.4.2 Parameters . . . . .                            | 14        |
| 3.4.3 Output File . . . . .                           | 14        |
| 3.4.4 Testing Log File . . . . .                      | 15        |
| Bibliography . . . . .                                | 16        |
| 3.5 Input Parameters System . . . . .                 | 16        |
| 3.5.1 Introduction . . . . .                          | 16        |
| 3.5.2 How To Include The IPS In Your System . . . . . | 18        |
| 3.6 Documentation . . . . .                           | 20        |
| <b>4 Development Cycle</b>                            | <b>21</b> |
| <b>5 Visualizer</b>                                   | <b>22</b> |
| <b>6 Case Studies</b>                                 | <b>23</b> |
| 6.1 QPSK Transmitter . . . . .                        | 23        |
| Bibliography . . . . .                                | 25        |
| 6.2 Optical Detection . . . . .                       | 26        |
| 6.2.1 Theoretical Analysis . . . . .                  | 26        |
| 6.2.2 Simulation Analysis . . . . .                   | 39        |
| 6.2.3 Experimental Analysis . . . . .                 | 45        |

|   |     |
|---|-----|
| <i>Contents</i>   | 2   |
| 6.2.4 Comparative analysis . . . . .                            | 49  |
| 6.2.5 Known problems . . . . .                                  | 50  |
| Bibliography . . . . .  | 51  |
| 6.3 BPSK Transmission System . . . . .                          | 52  |
| 6.3.1 Theoretical Analysis . . . . .                            | 52  |
| 6.3.2 Simulation Analysis . . . . .                             | 53  |
| 6.3.3 Comparative Analysis . . . . .                            | 57  |
| Bibliography . . . . .  | 59  |
| 6.4 Hamming Channel Encoder and Decoder . . . . .               | 60  |
| 6.4.1 Introduction . . . . .                                    | 60  |
| 6.4.2 Theoretical Analysis . . . . .                            | 60  |
| 6.4.3 Simulation Analysis . . . . .                             | 61  |
| Bibliography . . . . .  | 68  |
| 6.5 Huffman Source Encoder and Decoder . . . . .                | 69  |
| 6.5.1 Theoretical Analysis . . . . .                            | 69  |
| 6.5.2 Simulation Analysis . . . . .                             | 75  |
| 6.6 Arithmetic Encoding & Decoding . . . . .                    | 83  |
| 6.6.1 Encoding Algorithm . . . . .                              | 83  |
| 6.6.2 Decoding Algorithm . . . . .                              | 85  |
| 6.6.3 Encoding and Decoding Simulation Results . . . . .        | 86  |
| Bibliography . . . . .  | 87  |
| 6.7 Mutual information estimation for a binary source . . . . . | 87  |
| 6.7.1 Theoretical Analysis . . . . .                            | 87  |
| 6.7.2 Simulation Analysis . . . . .                             | 88  |
| Bibliography . . . . .  | 92  |
| 6.8 Dynamic Huffman Coder and Decoder . . . . .                 | 93  |
| 6.8.1 Code Analysis . . . . .                                   | 93  |
| 6.8.2 Practical Test . . . . .                                  | 94  |
| Bibliography . . . . .  | 96  |
| 6.9 M-QAM Transmission System . . . . .                         | 97  |
| 6.9.1 Introduction . . . . .                                    | 97  |
| 6.9.2 Theoretical Analysis . . . . .                            | 98  |
| 6.9.3 Simulation Analysis . . . . .                             | 111 |
| 6.9.4 Experimental Setup . . . . .                              | 159 |
| 6.9.5 Homodyne Detection . . . . .                              | 171 |
| 6.9.6 Digital Signal Post-Processing . . . . .                  | 180 |
| 6.9.7 Open Issues . . . . .                                     | 187 |
| 6.9.8 Future work . . . . .                                     | 187 |
| Bibliography . . . . .  | 188 |
| 6.10 Low Baud M-QAM Transmission System . . . . .               | 189 |
| 6.10.1 Experimental Results . . . . .                           | 189 |
| 6.10.2 Open Issues . . . . .                                    | 221 |

|  |     |
|--|-----|
| Bibliography . . . . .   | 222 |
| 6.11 Kramers-Kronig Transceiver . . . . .                                      | 223 |
| 6.11.1 Theoretical Analysis . . . . .  | 223 |
| 6.11.2 Numerical Validations . . . . .   | 236 |
| 6.11.3 Simulation Analysis . . . . .   | 253 |
| 6.11.4 Iterative method . . . . .  | 256 |
| Bibliography . . . . .   | 260 |
| 6.12 DSP Laser Phase Noise Compensation . . . . .                              | 261 |
| 6.12.1 Theoretical Analysis . . . . .  | 261 |
| 6.12.2 Simulation Analysis . . . . .   | 263 |
| 6.12.3 Simulation Results . . . . .  | 264 |
| 6.12.4 VHDL Implementation . . . . .   | 266 |
| 6.12.5 VHDL Simulation Results . . . . .                                       | 270 |
| 6.12.6 DSP Architectures for Coherent Receiver . . . . .                       | 272 |
| 6.12.7 Open Issues . . . . .   | 274 |
| 6.12.8 FMC + FPGA . . . . .  | 274 |
| Bibliography . . . . .   | 277 |
| 6.13 Quantum Random Number Generator . . . . .                                 | 278 |
| 6.13.1 Theoretical Analysis . . . . .  | 278 |
| 6.13.2 Simulation Analysis . . . . .   | 280 |
| 6.13.3 Experimental Analysis . . . . .   | 285 |
| 6.13.4 Open Issues . . . . .   | 287 |
| Bibliography . . . . .   | 288 |
| 6.14 BB84 with Discrete Variables . . . . .                                    | 289 |
| 6.14.1 Protocol Analysis . . . . .   | 289 |
| 6.14.2 Simulation Analysis . . . . .   | 293 |
| 6.14.3 Open Issues . . . . .   | 304 |
| Bibliography . . . . .   | 305 |
| 6.15 Quantum Oblivious Key Distribution with Discrete Variables . . . . .      | 306 |
| 6.15.1 Theoretical Description . . . . .                                       | 306 |
| 6.15.2 Simulation Analysis . . . . .   | 318 |
| 6.15.3 Experimental Setup . . . . .  | 341 |
| 6.15.4 Open Issues . . . . .   | 342 |
| Bibliography . . . . .   | 344 |
| 6.16 Discrete Variables Polarization Encoding . . . . .                        | 345 |
| 6.16.1 Theoretical Analysis . . . . .  | 345 |
| 6.16.2 Algorithm for polarization compensation . . . . .                       | 356 |
| 6.16.3 Simulation Analysis - Algorithm for polarization compensation . . . . . | 359 |
| 6.16.4 Experimental Setup . . . . .  | 359 |
| 6.16.5 Open Issues . . . . .   | 365 |
| Bibliography . . . . .   | 366 |
| 6.17 Discrete Variables Polarization Encoding Bob Processor . . . . .          | 368 |

|         |   |     |
|---------|---|-----|
| 6.17.1  | System overview . . . . .   | 368 |
| 6.17.2  | Brief Alice overview . . . . .                                      | 368 |
| 6.17.3  | Bob overview . . . . .  | 369 |
| 6.17.4  | Bob processor overview . . . . .                                    | 370 |
| 6.17.5  | Frame structure . . . . .   | 371 |
| 6.17.6  | Configuration file . . . . .  | 372 |
| 6.17.7  | VHDL Implementation of Bob processor . . . . .                      | 373 |
| 6.17.8  | Main processor . . . . .  | 373 |
| 6.17.9  | Main processor: Coincedencedetector . . . . .                       | 374 |
| 6.17.10 | Main processor: Frame alignment . . . . .                           | 375 |
| 6.17.11 | Main processor: FIFO buffer . . . . .                               | 380 |
| 6.17.12 | Main processor: Clock management and trigger . . . . .              | 381 |
| 6.17.13 | Main processor: Trigger phase adjustment . . . . .                  | 382 |
| 6.17.14 | QBER estimator . . . . .  | 383 |
| 6.17.15 | Output . . . . .  | 384 |
| 6.17.16 | Simulation of Bob processor . . . . .                               | 384 |
| 6.17.17 | VHDL code . . . . .   | 385 |
| 6.17.18 | Main entity . . . . .   | 385 |
| 6.17.19 | Open Issues . . . . .   | 388 |
|         | Bibliography . . . . .  | 389 |
| 6.18    | Quantum Noise . . . . .   | 389 |
| 6.18.1  | Theoretical Analysis . . . . .                                      | 389 |
| 6.18.2  | Numerical Analysis . . . . .  | 390 |
| 6.18.3  | Experimental Analysis . . . . .                                     | 393 |
|         | Bibliography . . . . .  | 394 |
| 6.19    | Frequency and Phase Recovery in CV-QC Systems . . . . .             | 395 |
| 6.19.1  | Classical Frequency and Phase Recovery - State of the art . . . . . | 395 |
| 6.19.2  | Quantum Frequency and Phase Recovery - State of the art . . . . .   | 398 |
| 6.19.3  | Open issues . . . . .   | 399 |
| 6.19.4  | Novelty . . . . .   | 399 |
| 6.19.5  | Novel Frequency Mismatch Compensation Technique . . . . .           | 399 |
| 6.19.6  | Implementation issues . . . . .                                     | 401 |
| 6.19.7  | Error Vector Magnitude . . . . .                                    | 406 |
| 6.19.8  | Comparative analysis of the frequency ranging techniques . . . . .  | 407 |
| 6.19.9  | Simulation Analysis . . . . .                                       | 409 |
| 6.19.10 | Simulation Results . . . . .  | 410 |
| 6.19.11 | New study . . . . .   | 411 |
| 6.19.12 | Future work . . . . .   | 415 |
|         | Bibliography . . . . .  | 421 |
| 6.20    | Quantum Key Distribution Without Basis Switching . . . . .          | 422 |
| 6.20.1  | Theoretical Analysis . . . . .                                      | 422 |
|         | Bibliography . . . . .  | 424 |

|   |            |
|---|------------|
| <i>Contents</i>   | 5          |
| 6.21 Intradyne Continuous Variables QKD Transmission System . . . . . | 425        |
| 6.21.1 Theoretical Analysis . . . . .                                 | 425        |
| 6.21.2 Simulation Analysis . . . . .                                  | 441        |
| 6.21.3 Experimental Analysis . . . . .                                | 441        |
| 6.21.4 Comparative Analysis . . . . .                                 | 441        |
| Bibliography . . . . .  | 442        |
| 6.22 Classical Multi-Party Computation . . . . .                      | 442        |
| 6.22.1 Introduction . . . . .   | 442        |
| 6.22.2 Two-Party Computation . . . . .                                | 443        |
| 6.22.3 Party Behavior Models . . . . .                                | 443        |
| 6.22.4 MPC Problems and Solutions . . . . .                           | 444        |
| 6.22.5 Garbled Circuit Protocol . . . . .                             | 445        |
| 6.22.6 Hardware Description Languages . . . . .                       | 445        |
| 6.22.7 TinyGarble . . . . .   | 445        |
| 6.22.8 ARM2GC . . . . .   | 446        |
| Bibliography . . . . .  | 447        |
| 6.23 Quantum Multi-Party Computation . . . . .                        | 448        |
| 6.23.1 Our Approach . . . . .   | 448        |
| Bibliography . . . . .  | 451        |
| 6.24 Secure Multi-Party Computation . . . . .                         | 452        |
| 6.24.1 Problem definition . . . . .                                   | 452        |
| 6.24.2 Secure Two-Party Computation . . . . .                         | 453        |
| 6.24.3 ANP - Problem definition . . . . .                             | 458        |
| 6.24.4 ANP - Secure Two-Party Computation . . . . .                   | 458        |
| 6.24.5 TinyGarble . . . . .   | 463        |
| Bibliography . . . . .  | 473        |
| 6.25 Tiny Garble . . . . .  | 474        |
| 6.25.1 Installation . . . . .   | 474        |
| 6.25.2 Usage . . . . .  | 476        |
| 6.25.3 Programming . . . . .  | 481        |
| 6.25.4 Open Issues . . . . .  | 489        |
| Bibliography . . . . .  | 490        |
| <b>7 Library</b>  | <b>491</b> |
| 7.1 ADC . . . . .   | 492        |
| 7.2 Add . . . . .   | 495        |
| 7.3 Arithmetic Encoder . . . . .                                      | 496        |
| 7.4 Arithmetic Decoder . . . . .                                      | 498        |
| 7.5 Alice QKD . . . . .   | 500        |
| 7.6 Ascii Source . . . . .  | 502        |
| 7.7 Ascii To Binary . . . . .   | 504        |
| 7.8 Balanced Beam Splitter . . . . .                                  | 506        |
| 7.9 Bit Error Rate . . . . .  | 507        |

|   |     |
|---|-----|
| Bibliography . . . . .                      | 509 |
| 7.10 Binary Source . . . . .                | 510 |
| 7.11 Binary To Ascii . . . . .              | 514 |
| 7.12 Bob QKD . . . . .                      | 516 |
| 7.13 Bit Decider . . . . .                  | 517 |
| 7.14 Clock . . . . .                        | 518 |
| 7.15 Clock_20171219 . . . . .               | 520 |
| 7.16 Complex To Real . . . . .              | 523 |
| 7.17 Coupler 2 by 2 . . . . .               | 525 |
| 7.18 Carrier Phase Compensation . . . . .   | 526 |
| 7.19 Decision Circuit . . . . .             | 530 |
| 7.20 Decoder . . . . .                      | 532 |
| 7.21 Discrete To Continuous Time . . . . .  | 534 |
| 7.22 DownSampling . . . . .                 | 536 |
| 7.23 DSP . . . . .                          | 538 |
| 7.24 EDFA . . . . .                         | 540 |
| 7.25 Electrical Signal Generator . . . . .  | 543 |
| 7.25.1 ContinuousWave . . . . .             | 543 |
| 7.26 Entropy Estimator . . . . .            | 545 |
| 7.27 Entropy Estimator . . . . .            | 546 |
| 7.28 Fork . . . . .                         | 548 |
| 7.29 Gaussian Source . . . . .              | 549 |
| 7.30 Hamming Decoder . . . . .              | 551 |
| 7.31 Hamming Encoder . . . . .              | 552 |
| 7.32 MQAM Receiver . . . . .                | 553 |
| 7.33 Huffman Decoder . . . . .              | 557 |
| 7.34 Huffman Encoder . . . . .              | 558 |
| 7.35 Ideal Amplifier . . . . .              | 559 |
| 7.36 IQ Modulator . . . . .                 | 561 |
| Bibliography . . . . .                      | 566 |
| 7.37 IIR Filter . . . . .                   | 567 |
| Bibliography . . . . .                      | 568 |
| 7.38 Local Oscillator . . . . .             | 569 |
| 7.39 Local Oscillator . . . . .             | 571 |
| 7.40 Mutual Information Estimator . . . . . | 574 |
| Bibliography . . . . .                      | 577 |
| 7.41 MQAM Mapper . . . . .                  | 578 |
| 7.42 MQAM Transmitter . . . . .             | 581 |
| 7.43 Netxpto . . . . .                      | 585 |
| 7.43.1 Version 20180118 . . . . .           | 585 |
| 7.43.2 Version 20180418 . . . . .           | 585 |
| 7.44 Alice QKD . . . . .                    | 586 |

|   |     |
|---|-----|
| 7.45 Polarizer . . . . .                          | 588 |
| 7.46 Probability Estimator . . . . .              | 589 |
| 7.47 Bob QKD . . . . .                            | 592 |
| 7.48 Eve QKD . . . . .                            | 593 |
| 7.49 Rotator Linear Polarizer . . . . .           | 594 |
| 7.50 Mutual Information Estimator . . . . .       | 596 |
| Bibliography . . . . .                            | 599 |
| 7.51 Optical Switch . . . . .                     | 600 |
| 7.52 Optical Hybrid . . . . .                     | 601 |
| 7.53 Photodiode pair . . . . .                    | 603 |
| 7.54 Photoelectron Generator . . . . .            | 606 |
| Bibliography . . . . .                            | 611 |
| 7.55 Power Spectral Density Estimator . . . . .   | 612 |
| Bibliography . . . . .                            | 618 |
| 7.56 Pulse Shaper . . . . .                       | 619 |
| 7.57 Quantizer . . . . .                          | 621 |
| 7.58 Resample . . . . .                           | 624 |
| 7.59 SNR Estimator . . . . .                      | 626 |
| Bibliography . . . . .                            | 631 |
| 7.60 Sampler . . . . .                            | 632 |
| 7.61 SNR of the Photoelectron Generator . . . . . | 634 |
| Bibliography . . . . .                            | 639 |
| 7.62 Sink . . . . .                               | 640 |
| 7.63 SNR Estimator . . . . .                      | 642 |
| Bibliography . . . . .                            | 647 |
| 7.64 Single Photon Receiver . . . . .             | 648 |
| 7.65 SOP Modulator . . . . .                      | 652 |
| Bibliography . . . . .                            | 656 |
| 7.66 Source Code Efficiency . . . . .             | 657 |
| 7.67 White Noise . . . . .                        | 658 |
| 7.68 Ideal Amplifier . . . . .                    | 661 |
| 7.69 Phase Mismatch Compensation . . . . .        | 663 |
| Bibliography . . . . .                            | 665 |
| 7.70 Frequency Mismatch Compensation . . . . .    | 666 |
| Bibliography . . . . .                            | 669 |
| 7.71 Cloner . . . . .                             | 670 |
| Bibliography . . . . .                            | 671 |
| 7.72 Error Vector Magnitude . . . . .             | 671 |
| Bibliography . . . . .                            | 673 |
| 7.73 Load Ascii . . . . .                         | 673 |
| 7.74 Load Signal . . . . .                        | 675 |

|   |            |
|---|------------|
| <i>Contents</i>   | 8          |
| <b>8 Mathlab Tools</b>                                      | <b>676</b> |
| 8.1 Generation of AWG Compatible Signals . . . . .          | 677        |
| 8.1.1 sgnToWfm.m . . . . .                                  | 677        |
| 8.1.2 sgnToWfm_20171121.m . . . . .                         | 678        |
| 8.1.3 Loading a signal to the Tektronix AWG70002A . . . . . | 680        |
| 8.2 Polarization Analysis Signals . . . . .                 | 684        |
| 8.2.1 jonesToStokes.m . . . . .                             | 684        |
| 8.2.2 ACF.m . . . . .                                       | 685        |
| 8.2.3 plotPhotonStream_20180102.m . . . . .                 | 686        |
| 8.2.4 plot_sphere.m . . . . .                               | 687        |
| <b>9 Algorithms</b>   | <b>688</b> |
| 9.1 Fast Fourier Transform . . . . .                        | 689        |
| Bibliography . . . . .                                      | 705        |
| 9.2 Overlap-Save Method . . . . .                           | 706        |
| Bibliography . . . . .                                      | 735        |
| 9.3 Filter . . . . .  | 736        |
| Bibliography . . . . .                                      | 745        |
| 9.4 Hilbert Transform . . . . .                             | 746        |
| Bibliography . . . . .                                      | 750        |
| <b>10 Code Development Guidelines</b>                       | <b>751</b> |
| 10.0.1 Integrated Development Environment . . . . .         | 751        |
| 10.0.2 Compiler Switches . . . . .                          | 751        |
| <b>11 Building C++ Projects Without Visual Studio</b>       | <b>752</b> |
| 11.1 Installing Microsoft Visual C++ Build Tools . . . . .  | 752        |
| 11.2 Adding Path To System Variables . . . . .              | 752        |
| 11.3 How To Use MSBuild To Build Your Projects . . . . .    | 753        |
| 11.4 Known Issues . . . . .                                 | 753        |
| 11.4.1 Missing ucrtbased.dll . . . . .                      | 753        |
| <b>12 Git Helper</b>  | <b>754</b> |
| 12.1 Starting with Git . . . . .                            | 754        |
| 12.2 Data Model . . . . .                                   | 754        |
| 12.2.1 Objects Folder . . . . .                             | 756        |
| 12.3 Refs . . . . .   | 756        |
| 12.3.1 Refs Folder . . . . .                                | 757        |
| 12.3.2 Branch . . . . .                                     | 757        |
| 12.3.3 Heads . . . . .                                      | 757        |
| 12.4 Git Logical Areas . . . . .                            | 757        |
| 12.5 Merge . . . . .  | 757        |
| 12.5.1 Fast-Forward Merge . . . . .                         | 757        |

|   |            |
|---|------------|
| <i>Contents</i>                                       | 9          |
| 12.5.2 Three-Way Merge . . . . .                      | 758        |
| 12.6 Remotes . . . . .                                | 758        |
| 12.6.1 GitHub . . . . .                               | 758        |
| 12.7 Commands . . . . .                               | 759        |
| 12.7.1 Porcelain Commands . . . . .                   | 759        |
| 12.7.2 Pluming Commands . . . . .                     | 763        |
| 12.8 Navigation Helpers . . . . .                     | 764        |
| 12.9 Configuration Files . . . . .                    | 764        |
| 12.10 Pack Files . . . . .                            | 764        |
| 12.11 Applications . . . . .                          | 765        |
| 12.11.1 Meld . . . . .                                | 765        |
| 12.11.2 GitKraken . . . . .                           | 765        |
| 12.12 Error Messages . . . . .                        | 765        |
| 12.12.1 Large files detected . . . . .                | 765        |
| 12.13 Git with Overleaf . . . . .                     | 765        |
| Bibliography . . . . .                                | 766        |
| <b>13 Beamer Helper</b>                               | <b>767</b> |
| 13.1 Intro to Beamer presentations . . . . .          | 767        |
| 13.2 Adding presenter notes to beamer . . . . .       | 767        |
| 13.2.1 Installing and using <i>pympress</i> . . . . . | 767        |
| 13.3 IT template . . . . .                            | 769        |
| <b>14 Simulating VHDL Programs with GHDL</b>          | <b>770</b> |
| 14.1 Adding Path To System Variables . . . . .        | 770        |
| 14.2 Using GHDL To Simulate VHDL Programs . . . . .   | 771        |
| 14.2.1 Simulation Input . . . . .                     | 771        |
| 14.2.2 Executing Testbench . . . . .                  | 771        |
| 14.2.3 Simulation Output . . . . .                    | 771        |

## **Chapter 1**

---

## **Preface**

Th

## **Chapter 2**

---

### **Introduction**

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

## Chapter 3

### Simulator Structure

---

LinkPlanner is a signals open-source simulator.

The major entity is the system.

A system comprises a set of blocks.

The blocks interact with each other through signals.

#### 3.1 System

#### 3.2 Blocks

#### 3.3 Signals

List of available signals:

- Signal

##### PhotonStreamXY

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

##### PhotonStreamXY\_MP

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

### 3.3.1 Circular Buffer

The signals use a circular buffer to store data. Because standard C++ do not have a circular buffer container (at least up to ISO C++17) one was developed. The circular buffer was developed using the same principles and style of the other STL containers, trying to make sure that the future integration of a standard circular buffer in our code will as easy as possible. In this development we use the following references [1, 2, 3]. In [1] a simple circular buffer implementation is presented, in [2] a standard like version of a circular buffer is discussed, and in [3] a comparative assessment is presented considering different implementation strategies. We try to follow [2] as possible.

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

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

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

## 3.4 Log File

### 3.4.1 Introduction

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

```

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

```

Figure 3.1: Disabling Log File

### 3.4.2 Parameters

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

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

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

### 3.4.3 Output File

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

```

log.txt
1 2018-04-08 23:27:37
2 MQamTransmitter|S1|space=20
3 MQamTransmitter|S0|space=20
4 0.224
5
6 2018-04-08 23:27:37
7 LocalOscillator|S2|space=20
8 0.001
9
10 2018-04-08 23:27:37
11 BalancedBeamSplitter|S1|ready=20
12 BalancedBeamSplitter|S2|ready=20
13 BalancedBeamSplitter|S3|space=20
14 BalancedBeamSplitter|S4|space=20
15 0
16
17 2018-04-08 23:27:37
18 I_HomodyneReceiver|S3|ready=20
19 I_HomodyneReceiver|S4|ready=20
20 I_HomodyneReceiver|S5|space=20
21 0.004

```

Figure 3.2: Output File Example

Figure 3.3 shows a portion of code that consists in the declaration and initialization of the I\_HomodyneReceiver block. In line 97, we can see that the block has 2 input signals,  $S_3$  and  $S_4$ , and is assigned 1 output signal,  $S_5$ . Going back to Figure 3.2 we can observe that  $S_3$  and  $S_4$  have 20 samples ready to be processed and the buffer of  $S_5$  is empty.

```

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

```

Figure 3.3: I-Homodyne Receiver Block Declaration

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

```

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

```

Figure 3.4: Four input parameters where loaded from a file

### 3.4.4 Testing Log File

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

## References

- [1] URL: <https://embeddedartistry.com/blog/2017/4/6/circular-buffers-in-cc>.
- [2] URL: [https://www.boost.org/doc/libs/1\\_68\\_0/doc/html/circular\\_buffer.html](https://www.boost.org/doc/libs/1_68_0/doc/html/circular_buffer.html).
- [3] URL: <https://www.codeproject.com/Articles/1185449/Performance-of-a-Circular-Buffer-vs-Vector-Deque-a>.

## 3.5 Input Parameters System

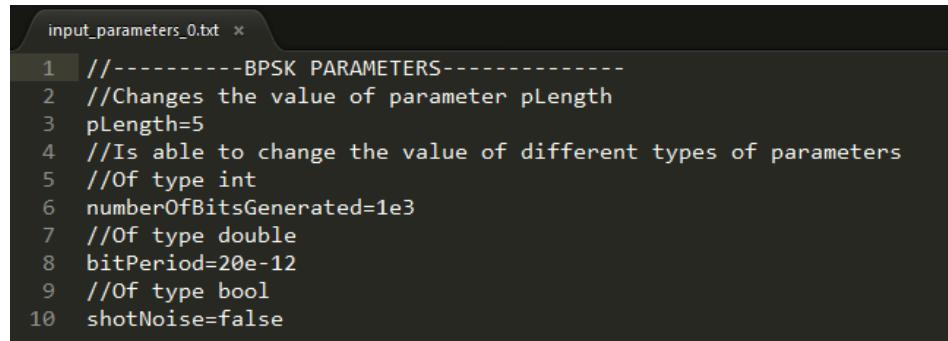
### 3.5.1 Introduction

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

#### Format of the Input File

We are going to explain the use of the IPS using as an example the PBSK system. In Figure 3.5, it is possible to observe the contents of the file **input\_parameters\_0.txt** used to load the values of some of the BPSK system's input parameters. The input file must respect the following properties:

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



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

Figure 3.5: Content of file input\_parameters\_0.txt

### Loading Input Parameters From A File

Execute the following command in the Command Line:

```
some_system.exe <input_file_path> <output_directory>
```

where **some\_system.exe** is the name of the executable generated after compiling the project, **<input\_file\_path>** is the path to the file containing the new input parameters; **<output\_directory>** is the directory where the output signals will be written into.

### 3.5.2 How To Include The IPS In Your System

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

1. Your system must include **netxpto\_20180418.h** or later.
2. A class that will contain the system input parameters must be created. This class must be a derived class of **SystemInputParameters**. In this case the created class is called **BPSKInputParameters**.
3. The created class must have 2 constructors. The implementation of these constructors is the same as **BPSKInputParameters**.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    }
};
```

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

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

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

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

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

    //System Run
    MainSystem.run();

    return 0;
}
```

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

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

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

### 3.6 Documentation

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

## **Chapter 4**

## **Development Cycle**

---

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

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

## **Chapter 5**

---

## **Visualizer**

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

## Chapter 6

## Case Studies

### 6.1 QPSK Transmitter

|                     |   |   |
|---------------------|---|---|
| <b>Student Name</b> | : | André Mourato (2018/07/08 - 2018/07/08) |
| <b>Goal</b>         | : | ...                                     |
| <b>Directory</b>    | : | sdf/qpsk_transmitter                    |

This system simulates a QPSK transmitter [1]. A schematic representation of this system is shown in figure 6.1.

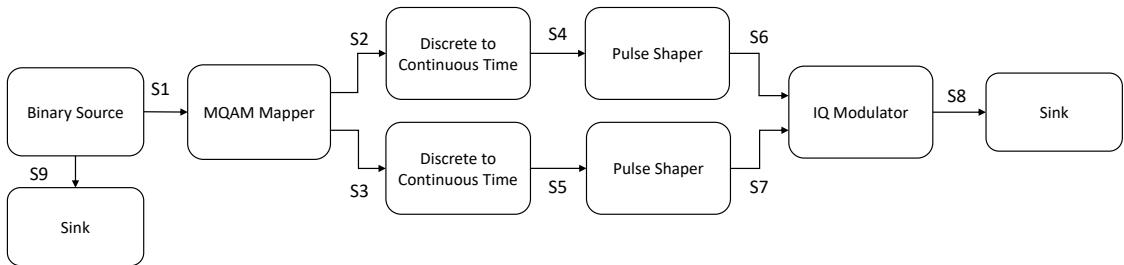


Figure 6.1: QPSK transmitter block diagram.

### Required files

| Header Files                           |          |        |
|--|----------|--------|
| File                                   | Comments | Status |
| binary_source_20180523.h               |          | ✓      |
| discrete_to_continuous_time_20180118.h |          | ✓      |
| iq_modulator_20180130.h                |          | ✓      |
| m_qam_mapper_20180118.h                |          | ✓      |
| netxpto_20180418.h                     |          | ✓      |
| pulse_shaper_20180118.h                |          | ✓      |
| sink_20180620.h                        |          | ✓      |

| Source Files                             |          |        |
|--|----------|--------|
| File                                     | Comments | Status |
| binary_source_20180523.cpp               |          | ✓      |
| discrete_to_continuous_time_20180118.cpp |          | ✓      |
| iq_modulator_20180130.cpp                |          | ✓      |
| m_qam_mapper_20180118.cpp                |          | ✓      |
| netxpto_20180418.cpp                     |          | ✓      |
| pulse_shaper_20180118.cpp                |          | ✓      |
| sink_20180620.cpp                        |          | ✓      |

### System Input Parameters

This system takes into account the following input parameters:

| System Input Parameters   |  |          |
|---------------------------|--|----------|
| Parameter                 | Default Value  | Comments |
| sourceMode                | Random   |          |
| patternLength             | 5  |          |
| bitStream                 | “0”  |          |
| bitPeriod                 | $\frac{1}{50e9}$                                     |          |
| iqAmplitude               | $\{\{\{1, 1\}, \{-1, 1\}, \{-1, -1\}, \{1, -1\}\}\}$ |          |
| numberOfBits              | 1000   |          |
| numberOfSamplesPerSymbol  | 16   |          |
| rollOffFactor             | 0.3  |          |
| impulseResponseTimeLength | 16   |          |
| bLength                   | 16   |          |

## References

- [1] Rodney Loudon. *The Quantum Theory of Light*. Oxford University Press, 2000.

## 6.2 Optical Detection

|                     |   |   |
|---------------------|---|---|
| <b>Contributors</b> | : | Nelson Muga, (2017-12-21 - ...)               |
|                     | : | Diamantino Silva, (2017-08-18 - 2018-02-05)   |
|                     | : | Armando Pinto (2017-08-15 - ...)              |
| <b>Goal</b>         | : | Analise of various optical detection schemes. |

The detection of light is a fundamental stage in every optical communication system, bridging the optical domain into the electrical domain. This section will review various theoretical, practical, and implementation aspects of the most important light detection schemes. The objective of this work is to develop numerical models for the various optical detections schemes and to validate such numerical models with experimental results.

### 6.2.1 Theoretical Analysis

|                     |   |   |
|---------------------|---|---|
| <b>Contributors</b> | : | Nelson Muga (2017-12-20 - )                                   |
|                     | : | Diamantino Silva (2017-08-18 - ...)                           |
|                     | : | Armando Pinto (2017-08-15 - ...)                              |
| <b>Goal</b>         | : | Theoretical description of various optical detection schemes. |

Receiver schemes for the detection of optical modulated signals can be roughly divided into two basic groups: Direct detection and coherent detection. In a direct detection receiver, its photodiode only responds to changes in the receiving signal optical power, and cannot extract any phase or frequency information from the optical carrier. However, using direct detection along with additional optics, e.g. an interferometer, the phase in one symbol may be compared to the phase in the previous symbol. This is often called interferometric detection. Coherent receivers use a carrier phase reference generated at the receiver, i.e. a local oscillator, and can track the phase of an optical transmitter so as to extract any phase and frequency information carried by a transmitted signal. Table 6.1 compares the three detection techniques, including maximum number of degrees of freedom and receiver sensitivities for binary and quaternary modulations.

In this subsection, we are going to calculate the signal-to-noise ratio at the input of the decision circuit for the various detection schemes under analysis. For each detection scheme a classical and a quantum description is going to be developed and a comparative analysis is going to be performed.

Table 6.1: COMPARISON OF DETECTION TECHNIQUES. SHADING DENOTES AN ADVANTAGE (from [Kahn2004] and [Kahn2006]).

| Attribute   | Noncoherent             |                  | Differential            |                  | Coherent                      |
|---|-------------------------|------------------|-------------------------|------------------|-------------------------------|
| Maximum number of degrees of freedom (per polarization)                               | 1 (magnitude)           |                  | 1 (phase)               |                  | 2 (two quadrature components) |
| Receiver sensitivity for binary   | 38 photons/bit (2-PAM)  |                  | 20 photons/bit (2-DPSK) |                  | 18 photons/bit (2-PSK)        |
| Receiver sensitivity for quaternary   | 134 photons/bit (4-PAM) |                  | 31 photons/bit (4-DPSK) |                  | 18 photons/bit (4-PSK)        |
|   | Heterodyne / Homodyne   | Direct Detection | Heterodyne / Homodyne   | Direct Detection | Heterodyne / Homodyne         |
| Electrical filtering can be used to select channel (enables frequency-agile receiver) | Yes                     | No               | Yes                     | No               | Yes                           |
| Chromatic dispersion is linear distortion (enables effective electrical compensation) | Yes                     | No               | Yes                     | No               | Yes                           |
| Local oscillator laser required at receiver   | Yes                     | No               | Yes                     | No               | Yes                           |
| Polarization control or diversity required at receiver                                | Yes                     | No               | Yes                     | No               | Yes                           |

## Classical Description

|                     |   |   |
|---------------------|---|---|
| <b>Contributors</b> | : | Nelson Muga (2017-12-20 - )   |
|                     | : | Diamantino Silva (2017-08-18 - ...)                                   |
|                     | : | Armando Pinto (2017-08-15 - ...)                                      |
| <b>Goal</b>         | : | Develop a classical description of various optical detection schemes. |

### Direct Detection

Direct detection schemes are convincingly simple as no phase, frequency or polarization control is necessary to recover the intensity of the optical field. This section will focus only the recover of the intensity of the optical field through the calculation of the current intensity at the output of a photodiode.

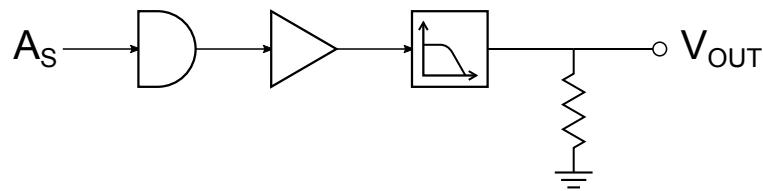


Figure 6.2: Noncoherent detection technique with direct detection implementation.

The electric field generated by an monochromatic single mode laser arriving at the input of the optical receiver can be written as

$$E_s(t) = \sqrt{P_s} \cdot A(t) \cdot e^{i(\omega_s t + \varphi_s)} \quad (6.1)$$

where  $P_s$  is the optical power,  $\omega_s$  is the carrier frequency,  $\varphi_s$  is the initial phase, and  $A = a(t)e^{i\varphi(t)}$  is the modulated envelop amplitude, with  $a(t)$  and  $\varphi(t)$  standing for the modulated amplitude and phase, respectively.

The output current of the photodiode,  $I_{DD}(t)$  can be written as

$$I_{DD}(t) = R \cdot E_s(t) \cdot E_s^*(t) + i_{sh} + i_{th} \quad (6.2)$$

$$= R \cdot a^2(t) \cdot P_s + i_{sh} + i_{th}, \quad (6.3)$$

where  $R$  represents the responsivity of the photodiode, which is equal to

$$R = \eta \frac{2\pi q}{h\omega_s}, \quad (6.4)$$

with  $q = 1.6 \cdot 10^{-19}$  C being the charge of the electron,  $h\omega_s/2\pi$  is the energy per photon with  $h = 6.63 \cdot 10^{-34}$  J·s being the Planck constant, and  $\eta$  is the fundamental quantum efficiency of the photodiode that corresponds to the averaged number of electrons generated per photon (Ref.Agrawal2013).

The two last terms on the right hand side of equation (6.3),  $i_{sh}$  and  $i_{th}$ , represent the current fluctuation related to the shot noise and thermal noise effects, respectively. From the classical interpretation, the shot noise is induced by the optical-to-electrical conversion process. Mathematically,  $i_{sh}(t)$  is a stationary random process with Poisson statistics, whose autocorrelation function is related with to the spectral density (Ref.Agrawal2012)

$$\langle i_{sh}(t)i_{sh}(t+\tau) \rangle = \int_{-\infty}^{+\infty} S_{sh}(f) e^{i2\pi f\tau} df \quad (6.5)$$

where  $\langle \cdot \rangle$  stands for the ensemble average over fluctuations. The two-sided spectral density (notice that the integral in (6.5) includes both negative and positive frequencies) of the shot noise is a constant term and can be written as

$$S_{sh}(f) = qI_p \quad (6.6)$$

where  $I_p$  is the average output current of the photodiode presented in equation (6.2), i.e.,

$$I_p(t) = I_{DD}(t) - (i_{sh} + i_{th}). \quad (6.7)$$

The shot noise variance can be calculated by setting  $\tau = 0$

$$\sigma_{sh}^2 = \langle i_{sh}(t)^2 \rangle = \int_{-\infty}^{+\infty} S_{sh}(f) df = 2qI_p B \quad (6.8)$$

where  $B$  is the photodetector bandwidth. The thermal noise term represented in (6.3) is related with the random thermal motion of electrons in the load resistor in the front end of the receiver, see Fig. 6.2. Mathematically, this noise contribution is modeled as a stationary Gaussian random process with a two-sided spectral density approximately constant (ref.Agrawal), given by

$$S_{th}(f) = 2k_B T / R_L \quad (6.9)$$

where  $k_B$  is the Boltzmann constant,  $T$  is the absolute temperature and  $R_L$  is the load resistor. Considering a autocorrelation function given by (6.5), with the subscripts  $sh$  replaced by  $th$ , and setting  $\tau = 0$ , the noise variance becomes

$$\sigma_{th}^2 = \langle i_{th}(t)^2 \rangle = \int_{-\infty}^{+\infty} S_{th}(f) df = \frac{4k_B T}{R} B. \quad (6.10)$$

It is worth noticing that: a) de photodiode bandwidth appears on the expressions of both noise terms; b) in contrast with the shot noise case, the variance of the thermal noise does not depend on the average photocurrent  $I_p$ .

Since the shot and thermal noise are independent random processes with approximately Gaussian statistics, the total variance of the current fluctuations,  $\Delta i = i_{sh} + i_{th}$ , can be written as the sum of the individual variances presented in (6.11) and (6.10)

$$\sigma_{\Delta I_{DD}}^2 = \langle \Delta I_{DD}^2 \rangle = \left( 2qI_p + \frac{4k_B T}{R} \right) B \quad (6.11)$$

Once obtained the variance of the photocurrent one can easily compute the signal-to-noise-ratio (SNR) of the optical receiver, which is a key parameter on the performance assessment of the device. The SNR of a electrical signal is defined as the ratio of the average signal power over the noise power, which leads to

$$SNR = \frac{I_p^2}{\sigma_{\Delta I_{DD}}^2} \quad (6.12)$$

Using (6.7) and (6.11) into the previous equation, one obtains the SNR as a function of the optical field impinging the photodiode

$$SNR = \frac{R^2(a^2(t) \cdot P_s)^2}{(2qI_p + 4k_B T/R) B} \quad (6.13)$$

-VV -VV -VV -VV -VV -VV -

We will consider that the detector has a bandwidth  $B$ , greater than the signal  $A(t)$ , but much smaller than  $2\omega_0$ . The calculation of power incident in the photodiode is given by the expected value of the square of the amplitude during a time interval  $\Delta t = 2\pi/\omega$

Measurable optical power, assuming that the detector bandwidth,  $B$ , is greater than the signal,  $A(t)$ , bandwidth but much small than  $2\omega_0$

$$\begin{aligned} P(t) &= \overline{E_R^2(t)} \\ &= \overline{|A(t)|^2} + \overline{|A(t)|^2 \cos(-2\omega t + 2\theta(t))} \\ &= |A(t)|^2 \quad W \end{aligned} \tag{6.14}$$

To simplify calculations, the electric field can be expressed the complex notation

$$E(t) = A(t)e^{-i\omega_0 t} \tag{6.15}$$

The physically measurable quantities are obtained by taking the real part of the complex wave. Using this notation, the beam power,  $P(t)$ , is obtained by multiplying the electric field's conjugate by itself

$$\begin{aligned} P(t) &= E^*(t)E(t) \\ &= |A(t)|^2 \end{aligned} \tag{6.16}$$

$$i(t) = \eta q \frac{P(t)}{\hbar\omega_0} \tag{6.17}$$

in which  $\eta$  is the photodiode's responsivity,  $q$  is the unit charge and  $P(t)/\hbar\omega_0$  is the number of removed electrons.

which will use to express the second moment of the photocurrent as

$$\langle i^2(t) \rangle = \eta^2 q^2 \frac{\langle |A(t)|^4 \rangle}{\hbar^2 \omega_0^2} \tag{6.18}$$

Assuming a fase modulation, in which the amplitude is constant, the signal is simplified to

$$A(t) = |A|e^{i\theta} \tag{6.19}$$

Therefore, the current becomes constant

$$i(t) = I_0 = \eta q \frac{A_s^2}{\hbar\omega_0} \tag{6.20}$$

and it's second moment becomes simply

$$\langle i^2(t) \rangle = I_0^2 \tag{6.21}$$

### Shot noise in photodiodes

$$\langle i_n^2(t) \rangle = 2qBI_0 \quad (6.22)$$

The signal to noise ratio is obtained by the relation between the second moment of the signal to the second moment of the noise

$$\begin{aligned} \frac{S}{N} &= \frac{\langle i^2(t) \rangle}{\langle i_n^2(t) \rangle} \\ &= \frac{I_0}{2qB} \\ &= \eta \frac{|A|^2}{\hbar\omega_0 B} \end{aligned} \quad (6.23)$$

### Homodyne Detection

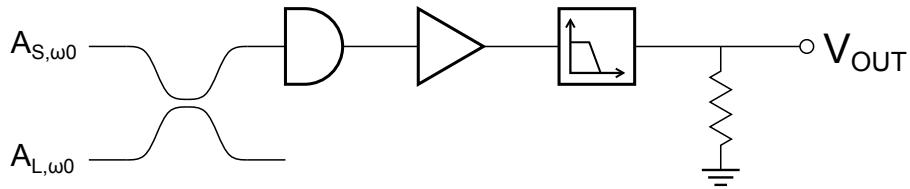


Figure 6.3: Homodyne detection.

The homodyne detection scheme uses an auxiliary local oscillator, which is combined in a beamsplitter with the signal beam. After this step, it is similar to the direct detection. As we will see this has some implications in the phase detection???

Given a splitter with intensity transmission  $\epsilon$ , the resulting field incident to the photodetector is [1]

$$E(t) = \sqrt{\epsilon}E_S(t) + \sqrt{1-\epsilon}E_{LO}(t) \quad (6.24)$$

in which  $E_{LO} = e^{i\omega_0 t}$ . Given a local oscillator with a much larger power than the signal, then, the incident power in the photodiode is

$$P(t) = \eta \left[ (1 - \epsilon)P_{LO}(t) + 2\sqrt{\epsilon(1 - \epsilon)}\text{Re}[E_S(t)E_{LO}^*(t)] \right] \quad (6.25)$$

$$= \eta \left[ (1 - \epsilon)P_{LO}(t) + 2\sqrt{\epsilon(1 - \epsilon)}|E_S(t)||E_{LO}(t)|\cos(\phi) \right] \quad (6.26)$$

$$(6.27)$$

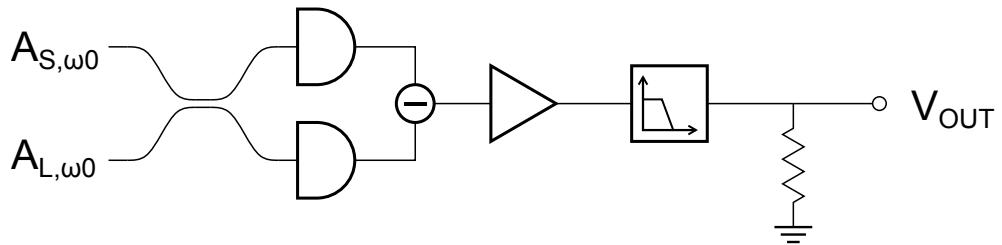
*Balanced Homodyne Detection*

Figure 6.4: Balanced homodyne detection.

"In a balanced homodyne detector (BHD), the signal to be measured is mixed with a local oscillator (LO) at a beam splitter. The interference signals from the two output ports of the beam splitter are sent to two photodiodes followed by a subtraction operation, and then, amplification may be applied. The output of a BHD can be made to be proportional to either the amplitude quadrature or the phase quadrature of the input signal depending on the relative phase between the signal and the LO".

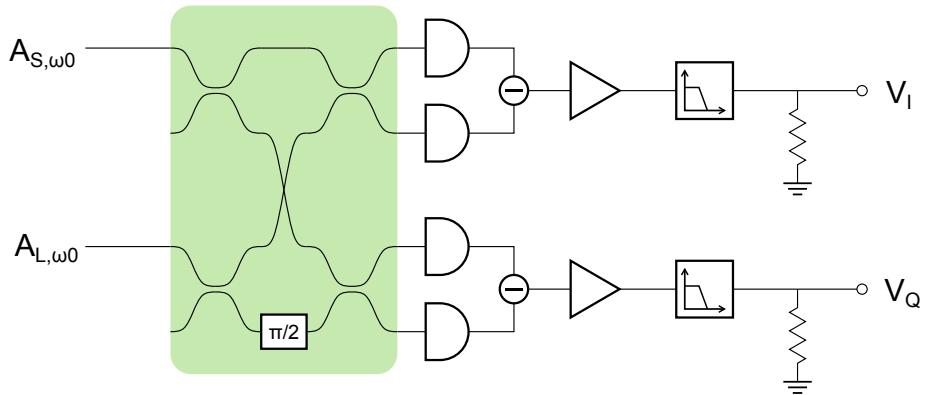
*IQ Homodyne Balanced Detection*

Figure 6.5: IQ balanced homodyne detection.

*Semiclassical model*

*Quantum model*

**Heterodyne Detection**

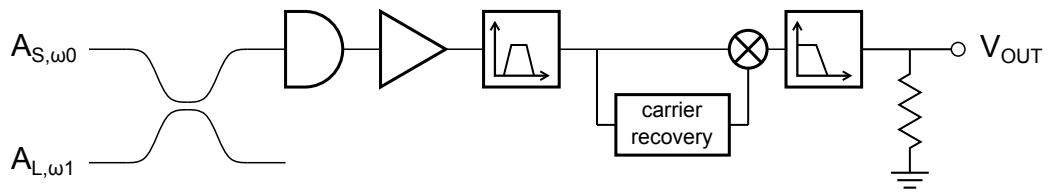


Figure 6.6: Heterodyne detection.

In contrast with the homodyne detection, in which the frequency of the signal carrier is equal to the frequency of the local oscillator, in the heterodyne detection, these frequencies are different.

Because of this, the inference will result in a new signal with an intermediate frequency at...

**Balanced Heterodyne Detection**

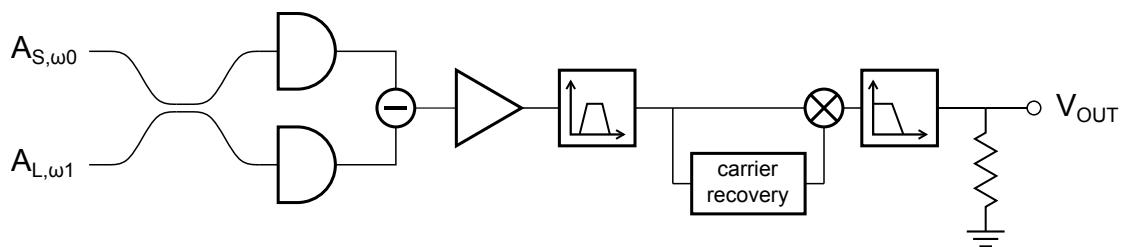


Figure 6.7: Balanced heterodyne detection.

*Semiclassical model*

### *Quantum model*

#### **IQ Heterodyne Balanced Detection**

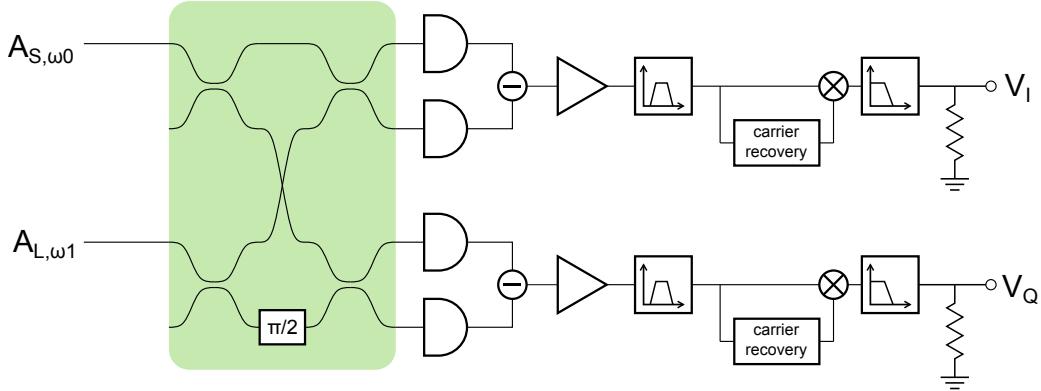


Figure 6.8: IQ balanced heterodyne detection.

#### **Thermal noise**

Thermal noise is generated by electrons in response to temperature. Its contribution to the resulting current can be described by the following equation [2]

$$\langle (\Delta i_T)^2 \rangle = 4K_B T_0 B / R_L \quad (6.28)$$

in which  $K_B$  it's Boltzmann's constant,  $T_0$  is the absolute temperature,  $B$  is the bandwidth and  $R_L$  is the receiver load impedance. The  $B$  value is imposed by default or chosen when the measurements are made, but the  $R_L$  value is dependent in the internal setup of the various components of the detection system. Nevertheless, for simulation purposes, we can just introduce an experimental value.

#### **Quantum Description**

|                     |   |   |
|---------------------|---|---|
| <b>Contributors</b> | : | Diamantino Silva (2017-08-18 - ...)   |
|                     | : | Armando Pinto (2017-08-15 - ...)  |
| <b>Goal</b>         | : | Develop a quantum description of various optical detection schemes, and compare with the classical description. |

We start by defining number states  $|n\rangle$  (or Fock states), which correspond to states with perfectly fixed number of photons [3]. Associated to those states are two operators, the creation  $\hat{a}^\dagger$  and annihilation  $\hat{a}$  operators, which in a simple way, remove or add one photon from a given number state [2]. Their action is defined as

$$\hat{a}|n\rangle = \sqrt{n}|n-1\rangle \quad (6.29), \quad \hat{a}^\dagger|n\rangle = \sqrt{n+1}|n+1\rangle \quad (6.30), \quad \hat{n}|n\rangle = n|n\rangle \quad (6.31)$$

in which  $\hat{n} = \hat{a}^\dagger\hat{a}$  is the number operator. Therefore, number states are eigenvectors of the number operator.

Coherent states have properties that closely resemble classical electromagnetic waves, and are generated by single-mode lasers well above the threshold. [3] We can define them, using number states in the following manner

$$|\alpha\rangle = e^{-\frac{|\alpha|^2}{2}} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \quad (6.32)$$

in which the complex number  $\alpha$  is the sole parameter that characterizes it. In fact, if we calculate the expected number of photons with  $\langle\alpha|\hat{n}|\alpha\rangle$  we will obtain  $|\alpha|^2$ . The coherent state is an eigenstate of the annihilation operator,  $\hat{a}|\alpha\rangle = \alpha|\alpha\rangle$ .

Using the creation and annihilation operators, we can define two quadrature operators [3]

$$\hat{X} = \frac{1}{2} (\hat{a}^\dagger + \hat{a}) \quad (6.33), \quad \hat{Y} = \frac{i}{2} (\hat{a}^\dagger - \hat{a}) \quad (6.34)$$

The expected value of these two operators, using a coherent state  $|\alpha\rangle$  are

$$\langle\hat{X}\rangle = \text{Re}(\alpha) \quad (6.35), \quad \langle\hat{Y}\rangle = \text{Im}(\alpha) \quad (6.36)$$

We see that the expected value of these operators give us the real and imaginary part of  $\alpha$ . Now, we can obtain the uncertainty of these operators, using:

$$\text{Var}(\hat{X}) = \langle\hat{X}^2\rangle - \langle\hat{X}\rangle^2 \quad (6.37)$$

For each of these quadrature operators the variance will be

$$\text{Var}(\hat{X}) = \text{Var}(\hat{Y}) = \frac{1}{4} \quad (6.38)$$

This result shows us that for both quadratures, the variance of measurement is the same and independent of the value of  $\alpha$ .

### Homodyne detection

The measurement of a quadrature of an input signal (S) is made by using the balanced homodyne detection technique, which measures the phase difference between the input signal and a local oscillator (LO). The measurement of quadrature are made relative to a reference phase of the LO, such that if the measurement is made in-phase with this reference,

the value will be proportional to the  $\hat{X}$  quadrature of the signal. If the phase of the LO is has an offset of  $\pi/2$  relative to the reference, the output will be proportional to the  $\hat{Y}$  quadrature of the signal.

Experimentally, the balanced homodyne detection requires a local oscillator with the same frequency as the input signal, but with a much larger amplitude. These two signals are combined using a 50:50 beam splitter, from were two beams emerge, which are then converted to currents using photodides. Finally, the two currents are subtracted, resulting in an output current proportional to a quadrature of the input signal [2].

A phase of the local oscillator can be defined as the reference phase. A phase offset equal to 0 or  $\pi/2$  will give an output proportional to the signal's in-phase component or to the quadrature component, respectively. Therefore, the  $\hat{X}$  operator will correspond to the in-phase component and  $\hat{Y}$  operator correspond to quadrature component

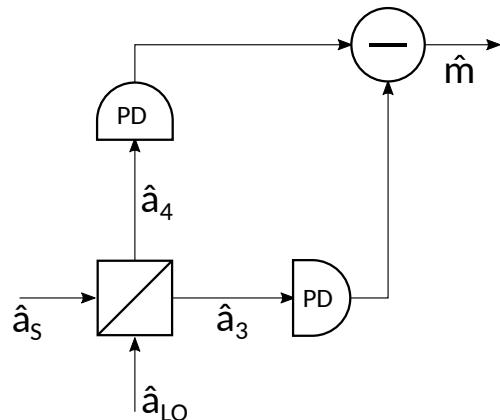


Figure 6.9: Balanced homodyne detection.

In the lab and in our simulations, a more complex system is used, the double balanced homodyne detection, which allows the simultaneous measurement of the  $\hat{X}$  and  $\hat{Y}$  components. The signal is divided in two beam with half the power of the original. One of the beams is used in a balanced homodyne detection with a local oscillator. The other beam is used in another balanced homodyne detection, but using a local oscillator with a phase difference  $\pi/2$  relative to the first one.

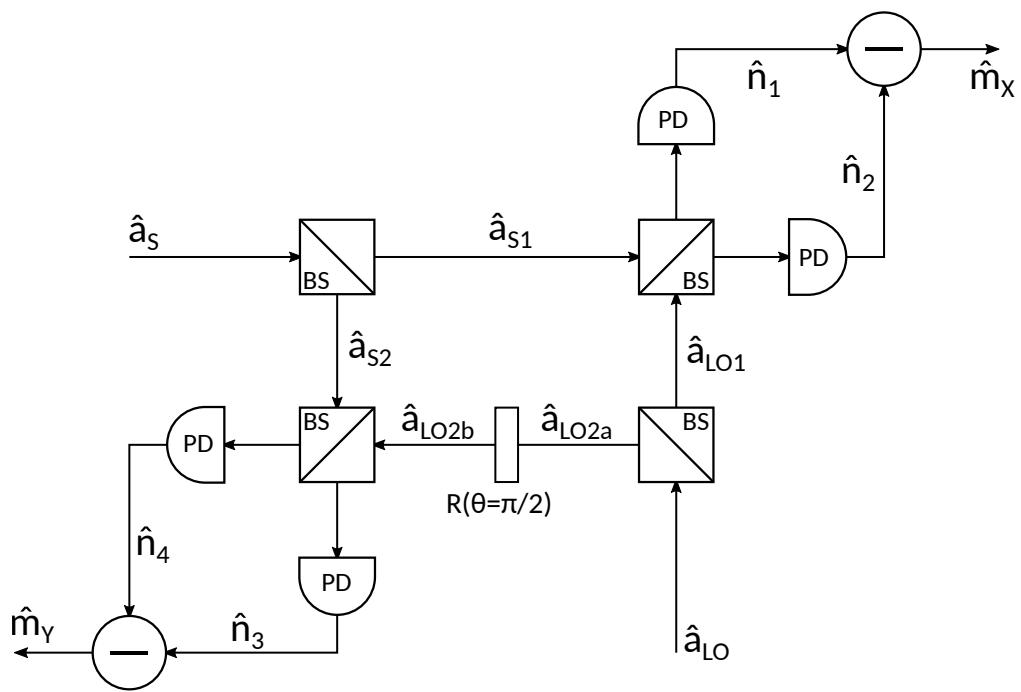


Figure 6.10: Balanced double homodyne detection.

---

## Bibliography

- [1] Joseph M. Kahn, *Modulation and Detection Techniques for Optical Communication Systems*, OSA/COTA - Coherent Optical Technologies and Applications Topical Meeting, BC, Canada, 2006.
- [2] Joseph M. Kahn, and Keang-Po Ho, Spectral Efficiency Limits and Modulation/Detection Techniques for DWDM Systems, *IEEE Journal Of Selected Topics In Quantum Electronics*, Vol. 10, No. 2, 2004.

### Noise sources in homodyne detection

The detection of light using photodiodes is subjected to various sources of noise. One of these sources is the electrical field itself. The interaction of the signal with the vacuum field adds quantum noise to the detection. Another source of noise comes from the detection system, such as photodiodes and other electrical circuits, originating various kinds of noise, such as thermal noise, dark noise and amplifier noise [4]. In the following sections, we will focus on two noise sources, quantum noise and thermal noise.

### Quantum Noise

In order to grasp this effect, the quantum mechanical description of balanced homodyne detection will be used, employing quantum operators to describe the effect of each component in the system (fig. ??). We start with the operators  $\hat{a}_S$  and  $\hat{a}_{LO}$  corresponding to the annihilation operator for the signal and local oscillator, which are the inputs in a beam divisor. The outputs will be  $\hat{a}_3$  and  $\hat{a}_4$ . Using a balanced beam splitter, we can write the output as

$$\hat{a}_3 = \frac{1}{\sqrt{2}} (\hat{a}_S + \hat{a}_{LO}) \quad (6.39), \quad \hat{a}_4 = \frac{1}{\sqrt{2}} (\hat{a}_S - \hat{a}_{LO}) \quad (6.40)$$

The final output of a homodyne measurement will be proportional to the difference between the photocurrents in arm 3 and 4. Then

$$I_{34} = I_3 - I_4 \sim \langle \hat{n}_3 - \hat{n}_4 \rangle \quad (6.41)$$

We can define an operator that describes the difference of number of photons in arm 3 and arm 4:

$$\hat{m} = \hat{a}_3^\dagger \hat{a}_3 - \hat{a}_4^\dagger \hat{a}_4 \quad (6.42)$$

If we assume that the local oscillator produces the the coherent state  $|\beta\rangle$ , then the expected value of this measurement will be

$$\langle m \rangle = 2|\alpha||\beta| \cos(\theta_\alpha - \theta_\beta) \quad (6.43), \quad \text{Var}(m) = |\alpha|^2 + |\beta|^2 \quad (6.44)$$

The local oscillator normally has a greater power than the signal , then  $|\alpha| \ll |\beta|$ . If we use as unit,  $2|\beta|$ , then these two quantities can be simplified to

$$\langle m \rangle = |\alpha| \cos(\theta_\alpha - \theta_\beta) \quad (6.45), \quad \text{Var}(m) \approx \frac{1}{4} \quad (6.46)$$

[4]

Has we have seen previously, in order to measure two quadratures simultaneously, we can use double balanced homodyne detection. For each quadrature, the input signal now has half the power, so  $|\alpha| \rightarrow |\alpha/\sqrt{2}|$ . If we use a local oscillator that produces states  $|\beta\rangle$ , then we can divide it in two beams in state  $|\beta/\sqrt{2}\rangle$  and  $|i\beta/\sqrt{2}\rangle$  which will be used in each homodyne detection. In this setting, the expected values for each quadrature,  $X$  and  $Y$ , (in normalized values of  $\sqrt{2}|\beta|$ ) are

$$\langle m_X \rangle = \left| \frac{\alpha}{\sqrt{2}} \right| \cos(\theta_\alpha - \theta_\beta) \quad (6.47), \quad \text{Var}(m_X) \approx \frac{1}{4} \quad (6.48)$$

$$\langle m_Y \rangle = \left| \frac{\alpha}{\sqrt{2}} \right| \sin(\theta_\alpha - \theta_\beta) \quad (6.49), \quad \text{Var}(m_Y) \approx \frac{1}{4} \quad (6.50)$$

Therefore the measurement of each quadrature will have half the amplitude, but the same variance.

### 6.2.2 Simulation Analysis

|                     |   |  |
|---------------------|---|--|
| <b>Contributors</b> | : | Diamantino Silva, (2017-08-18 - ...)             |
| <b>Goal</b>         | : | Simulation of various optical detection schemes. |
| <b>Directory</b>    | : | sdf/optical_detection                            |

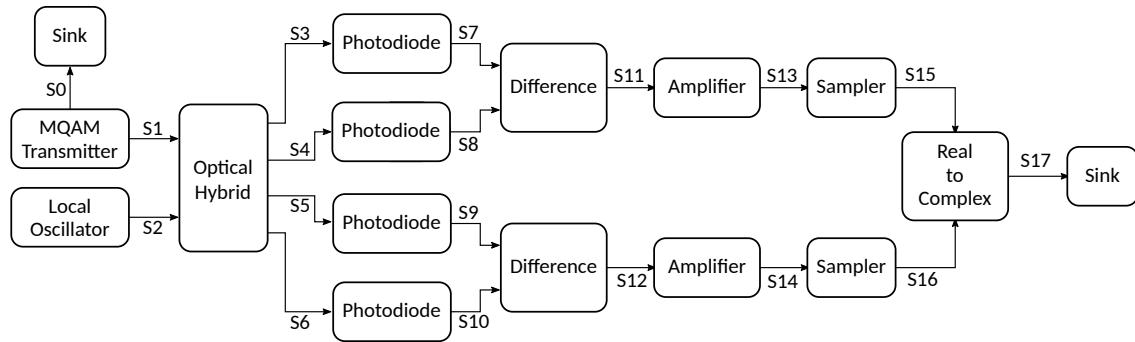


Figure 6.11: Overview of the simulated optical system.

List of signals used in the simulation:

| Signal name | Signal type                           | Status |
|-------------|---------------------------------------|--------|
| S0          | Binary                                | check  |
| S1          | OpticalSignal                         | check  |
| S2          | OpticalSignal                         | check  |
| S3          | OpticalSignal                         | check  |
| S4          | OpticalSignal                         | check  |
| S5          | OpticalSignal                         | check  |
| S6          | OpticalSignal                         | check  |
| S7          | TimeContinuousAmplitudeContinuousReal | check  |
| S8          | TimeContinuousAmplitudeContinuousReal | check  |
| S9          | TimeContinuousAmplitudeContinuousReal | check  |
| S10         | TimeContinuousAmplitudeContinuousReal | check  |
| S11         | TimeContinuousAmplitudeContinuousReal | check  |
| S12         | TimeContinuousAmplitudeContinuousReal | check  |
| S13         | TimeContinuousAmplitudeContinuousReal | check  |
| S14         | TimeContinuousAmplitudeContinuousReal | check  |
| S15         | TimeDiscreteAmplitudeContinuousReal   | check  |
| S16         | TimeDiscreteAmplitudeContinuousReal   | check  |
| S17         | OpticalSignal                         | check  |

This system takes into account the following input parameters:

| System Parameters     | Default value                                    | Description  |
|-----------------------|--|--|
| localOscillatorPower1 | $2.0505 \times 10^{-8} \text{W}$                 | Sets the optical power, in units of W, of the local oscillator inside the MQAM             |
| localOscillatorPower2 | $2.0505 \times 10^{-8} \text{W}$                 | Sets the optical power, in units of W, of the local oscillator used for Bob's measurements |
| localOscillatorPhase  | 0 rad  | Sets the initial phase of the local oscillator used in the detection                       |
| responsivity          | 1 A/W  | Sets the responsivity of the photodiodes used in the homodyne detectors                    |
| iqAmplitudeValues     | $\{\{1, 1\}, \{-1, 1\}, \{-1, -1\}, \{1, -1\}\}$ | Sets the amplitude of the states used in the MQAM  |

The simulation setup is represented in figure 6.11. The starting point is the MQAM, which generates random states from the constellation given by the variable iqAmplitudeValues. The output from the generator is received in the Optical Hybrid where it is mixed with a local oscillator, outputing two optical signal pairs. Each pair is converted to currents by two photodiodes, and the same currents are subtracted from each other, originating another current proportional to one of the quadratures of the input state. The other pair suffers the same process, but the resulting subtraction current will be proportional to another quadrature, dephased by  $\pi/2$  relative to the other quadrature.

## Required files

### Header Files

| File                | Description   | Status |
|---------------------|---|--------|
| netxpto.h           | Generic purpose simulator definitions.                | check  |
| m_qam_transmitter.h | Outputs a QPSK modulated optical signal.              | check  |
| local_oscillator.h  | Generates continuous coherent signal.                 | check  |
| optical_hybrid.h    | Mixes the two input signals into four outputs.        | check  |
| photodiode.h        | Converts an optical signal to a current.              | check  |
| difference.h        | Ouputs the difference between two input signals.      | check  |
| ideal_amplifier.h   | Performs a perfect amplification of the input sinal   | check  |
| sampler.h           | Samples the input signal.                             | check  |
| real_to_complex.h   | Combines two real input signals into a complex signal | check  |
| sink.h              | Closes any unused signals.                            | check  |

### Source Files

| <b>File</b>           | <b>Description</b>                                    | <b>Status</b> |
|-----------------------|---|---------------|
| netxpto.cpp           | Generic purpose simulator definitions.                | check         |
| m_qam_transmitter.cpp | Outputs a QPSK modulated optical signal.              | check         |
| local_oscillator.cpp  | Generates continuous coherent signal.                 | check         |
| optical_hybrid.cpp    | Mixes the two input signals into four outputs.        | check         |
| photodiode.h          | Converts an optical signal to a current.              | check         |
| difference.h          | Ouputs the difference between two input signals.      | check         |
| ideal_amplifier.h     | Performs a perfect amplification of the input sinal   | check         |
| sampler.cpp           | Samples the input signal.                             | check         |
| real_to_complex.cpp   | Combines two real input signals into a complex signal | check         |
| sink.cpp              | Empties the signal buffer.                            | check         |

## Simulation Results

To test the simulated implementation, a series of states  $\{|\phi_i\rangle\}$  were generated and detected, resulting in a series of measurements  $\{(x_i, y_i)\}$ . The simulation result is presented in figure 6.12:

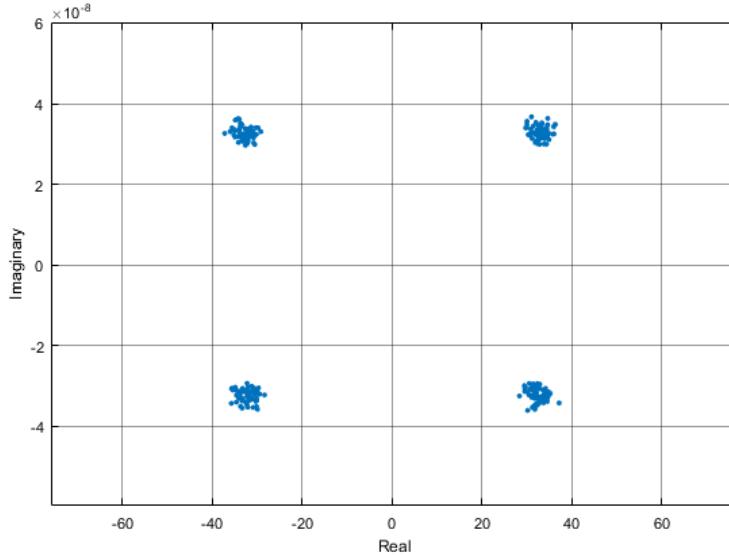


Figure 6.12: Simulation of a constelation of 4 states ( $n = 100$ )

We see that the measurements made groups in certain regions. Each of this groups is centered in the expected value  $(\langle X \rangle, \langle Y \rangle)$  of one the generated states. Also, they show some variance, which was tested for various expected number of photons,  $\langle n \rangle$ , resulting in figure 6.13:

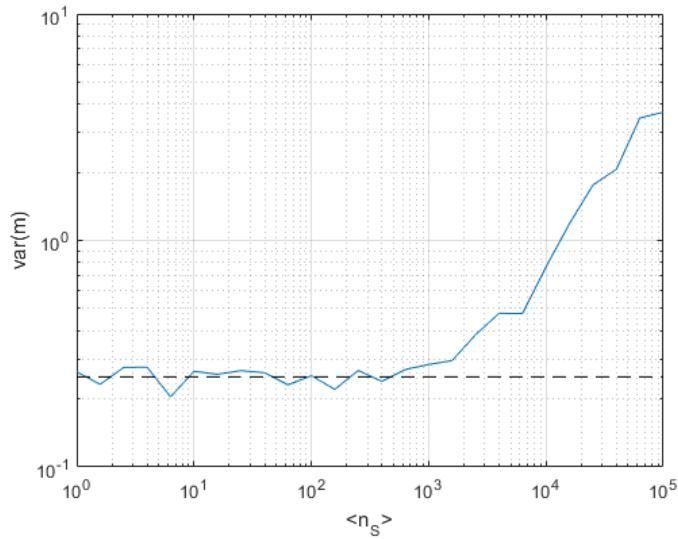


Figure 6.13: Simulation of the variance of  $m$ .  
Local oscillator expected number of photons:  $10^4$

It was expected that the variance should independent of the input's signal number of photons. Plot 6.13 shows that for low values of  $n_S$ , the simulation is in accordance with the theoretical prevision, with  $\text{Var}(X) = \text{Var}(Y) = \frac{1}{4}$ . For large values of  $n_S$ , when the number of photons is about the same has the local oscillator, the quantum noise variance starts to grow proportionally to  $n_S$ , in accordance with the non approximated calculation of quantum noise (eq. ??).

#### Noise Variance with LO power Simulation

The following plot shows the behavior of current noise variance  $\langle(\Delta i)^2\rangle$  with local oscilator power,  $P_{LO}$ :

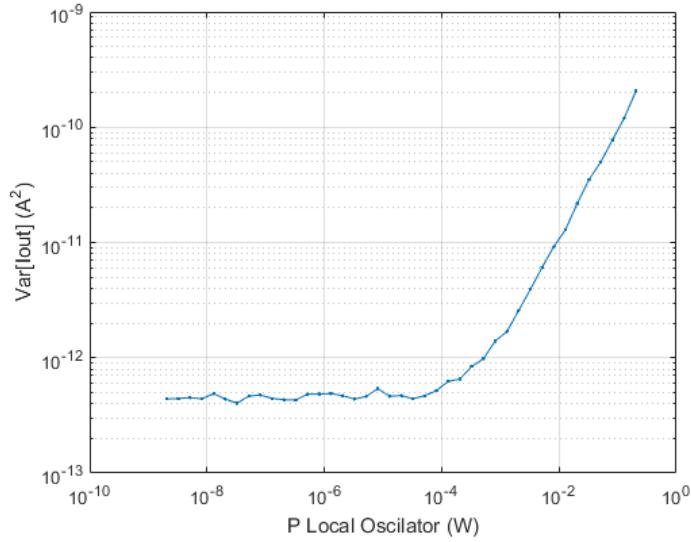


Figure 6.14: Output current variance in function of LO power.

We see that for low LO power, the dominant noise is the thermal contribution, but for higher power, quantum noise dominates, growing proportionally to  $P_{LO}$ . This in accordance with equation 6.45

### 6.2.3 Experimental Analysis

In this section, we will test experimentally the setups discussed in section 6.2.1. This comparison between the theoretical and experimental results will require a high degree of precision from the devices used in these setups. Therefore, the correct characterization of these devices must be the starting point of this experimental phase. One of the most fundamental components is the photodetector, which performs the signal's conversion from the optical domain into the electrical domain.

#### Thorlabs detector

The detector used in the laboratory is the Thorlabs PDB 450C. This detector consists of two well-matched photodiodes and a transimpedance amplifier that generates an output voltage (RF OUTPUT) proportional to the difference between the photocurrents of the photodiodes. Additionally, the unit has two monitor outputs (MONITOR+ and MONITOR-) to observe the optical input power level on each photodiode separately. [5]

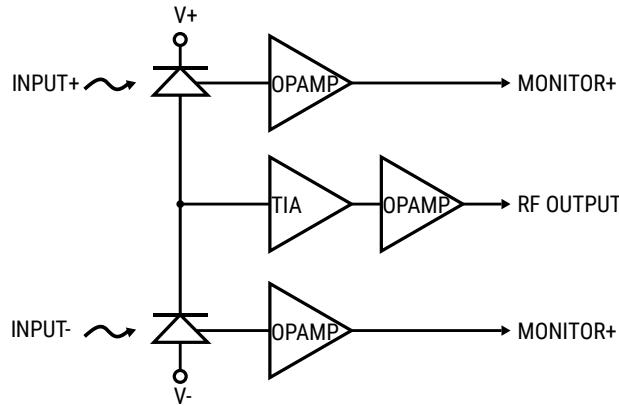


Figure 6.15: Functional block diagram of the PDB 450C Thorlabs detector [5]

Figure 6.15 shows a functional block diagram of the photodetector, in which the TIA (transimpedance amplifier) is an opamp-based current to voltage conversion amplifier. In contrast to the photodiode's non-linear voltage response to incident light, its current response is linear. To take advantage of this linear response, the TIA is used to perform the conversion of the difference of currents between the two photodiodes into a voltage proportional to that same difference. Various of those parameters can be readily extracted from the device's manual, which are presented in the following tables and plots.

| Parameter        | Value   |
|------------------|---------|
| Max Responsivity | 1.0 A/W |

Table 6.2: Thorlabs PDB450C PIN parameters

| Parameter           | Value                       |
|---------------------|-----------------------------|
| Bandwidth (-3dB)    | 1 MHz                       |
| Voltage Gain        | 10 V/mW @ peak responsivity |
| Voltage Noise (RMS) | <180 $\mu$ V (RMS)          |

Table 6.3: Thorlabs PDB450C MONITOR +/- output parameters

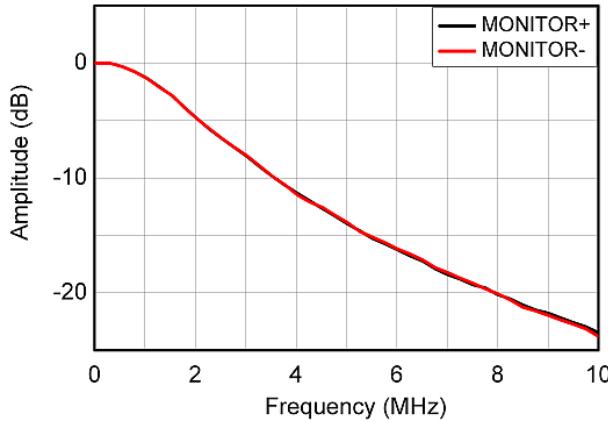


Figure 6.16: MONITOR +/- output response with frequency. [5]

The parameters of the RF OUTPUT have 5 values each, corresponding to the 5 gain settings of the TIA.

| Parameter                          | Values |        |        |        |        |     |
|------------------------------------|--------|--------|--------|--------|--------|-----|
| Bandwidth (-3dB)                   | 150    | 45     | 4      | 0.3    | 0.1    | MHz |
| Transimpedance Gain                | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | V/A |
| Conversion Gain                    | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | V/W |
| Overall Output Voltage Noise (RMS) | 0.50   | 0.80   | 1.0    | 1.1    | 2.0    | mV  |

Table 6.4: Thorlabs PDB450C RF OUTPUT parameters

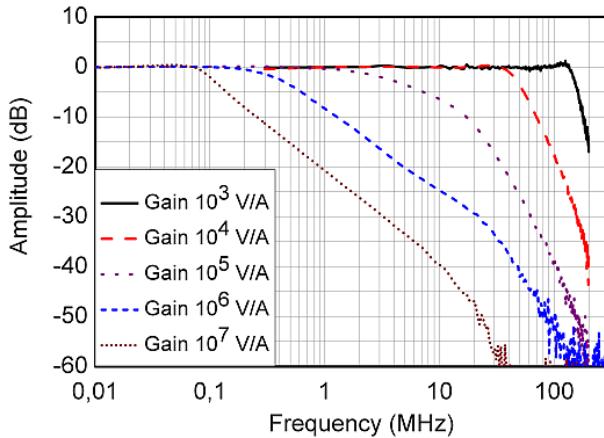


Figure 6.17: RF output response with frequency, for various gain values. [5]

### Data analysis

For each configuration of amplitude and frequency of the input signal, the photodetector output voltage is collected by the Digital Oscilloscope during a time interval and saved in a

data file. The data consists on a sequence of pulses and it's analysis will be focused on the samples with equal phase. The steps will be the following

1. For each phase value, the average and variance of the samples with the same phase is calculated;
2. The representative amplitude and variance for the present configuration will be the obtained from the middle of the maximum plateau.

To confirm the theoretical results obtained in section 6.2.1, two experimental setups will be created. In the first experiment, we will study quantum noise in the single homodyne detection. The experimental setup will be based on the paper [6]. In the second experiment, we will study quantum noise in the double homodyne detection setup, which will be basically an extension of the single homodyne detection setup.

### Single homodyne detection

To keep the experiment simple and avoid extra sources of noise, we will avoid using black boxes which have complicated inner workings, having a preference in using simple components, as shown in fig. 6.18:

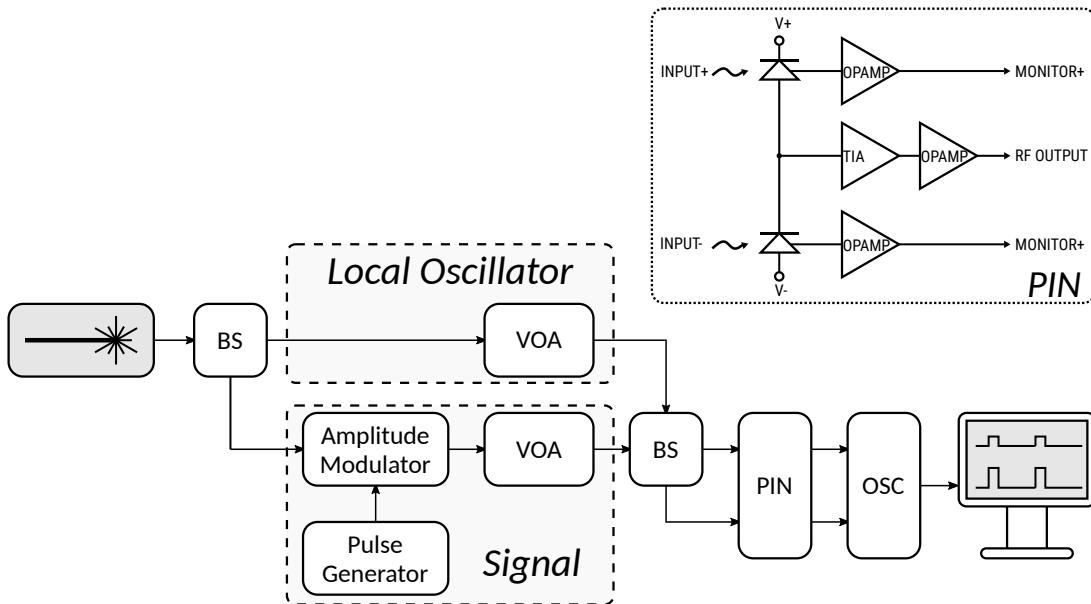


Figure 6.18: Experimental setup

### Material list

| Device              | Description                |
|---------------------|----------------------------|
| Local Oscillator    | Yenista OSICS Band C/AG    |
| BS                  | Beam Splitter              |
| Pulse Generator     | HP 8116A Pulse Generator   |
| Amplitude Modulator | Mach Zehnder SDL OC 48     |
| VOA                 | Eigenlicht Power Meter 420 |
| VOA                 | Thorlabs VOA 45-APC        |
| PIN                 | Thorlabs PDB 450C          |
| ADC                 | Picoscope 6403D            |

A single laser is splitted and used as the source for the signal (S) and the local oscillator (LO). The signal beam is pulsed and highly attenuated. The local oscillator is also attenuated, but not pulsed. The signal and local oscillator interfere in a Beam Splitter originating two beams which are then converted to voltages in the PIN. These voltages are read in the Digital Oscilator (OSC) and collect in the computer. In the post processing phase, the quantum noise is measured by applying a difference between the two beams and measuring it's variance.

The second stage of the experiment will be very similar to the first one, in which the signal and local oscillator branches will be divided. One of the new branches of the local oscillator will suffer a phase delay of  $\pi/2$ , in order to measure the quadrature component of the incoming signal.

#### 6.2.4 Comparative analysis

Given the theoretical, simulated and experimental frameworks, we will now compare the results obtained by each of them.

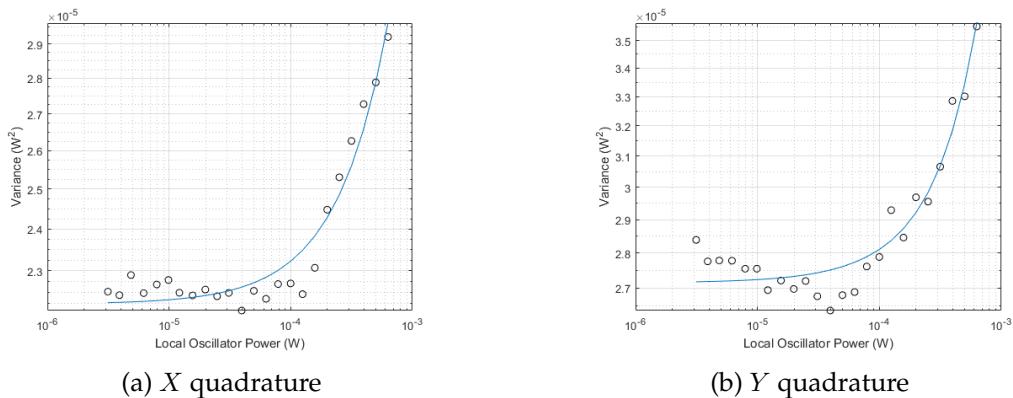


Figure 6.19: Noise variance dependency with local oscillator power for two different quadratures. Experimental vs fitted data.

Figures 6.19a and 6.19b show measurements of total noise for two different quadratures. For low power of LO, the noise variance fluctuates around a constant value. For high power of LO, ( $P_{LO} > 10^{-4}W$ ), the variance of noise shows an increasing trend roughly proportional to  $P_{LO}^2$ . The polynomial fittings confirm this trend, showing a degree 2 coefficient much larger than the degree 1 coefficient

$$\text{Var}_X = 2.22 \times 10^{-5} + 9.6 \times 10^{-3}P_{LO} + 3.40P_{LO}^2 \quad (6.51)$$

$$\text{Var}_Y = 2.71 \times 10^{-5} + 8.9 \times 10^{-3}P_{LO} + 7.25P_{LO}^2 \quad (6.52)$$

The expected growth should be proportional to  $P_{LO}$ , but the RIN noise, originated by the electric apparatus, which grows quadratically with the power, is dominating the noise amplitude for large  $P_{LO}$ .

We see that both the simulation and experimental data display a similar behaviour, but the quadratic growth of noise for large  $P_{LO}$  was not predicted in the simulations.

### 6.2.5 Known problems

## References

- [1] J Shapiro. "Quantum noise and excess noise in optical homodyne and heterodyne receivers". In: *IEEE journal of quantum electronics* 21.3 (1985), pp. 237–250.
- [2] Mark Fox. *Quantum Optics: An Introduction*. Oxford University Press, 2006.
- [3] Rodney Loudon. *The Quantum Theory of Light*. Oxford University Press, 2000.
- [4] Hans-A. Bachor and Timothy C. Ralph. *A Guide to Experiments in Quantum Optics*. Wiley-VCH, 2004.
- [5] Thorlabs. *Thorlabs Balance Amplified Photodetectors: PDB4xx Series Operation Manual*. 2014.
- [6] Yue-Meng Chi et al. "A balanced homodyne detector for high-rate Gaussian-modulated coherent-state quantum key distribution". In: *New Journal of Physics* 13.1 (2011), p. 013003.

### 6.3 BPSK Transmission System

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

Binary Phase Shift Keying (BPSK) is the simplest form of Phase Shift Keying (PSK), in which binary information is encoded into a two state constellation with the states being separated by a phase shift of  $\pi$  (see Figure 6.20).

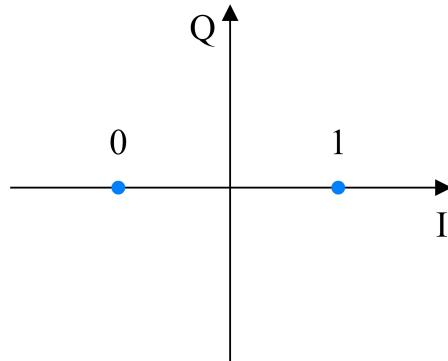


Figure 6.20: BPSK symbol constellation.

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

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

#### 6.3.1 Theoretical Analysis

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

$$m_i = 2\sqrt{P_L P_S} G_{ele} \cos(\Delta\theta_i), \quad (6.53)$$

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

$\Delta\theta_i$  is the phase difference between the local oscillator and the signal, for BPSK this takes the values  $\pi$  and 0, in which case (6.53) can be reduced to,

$$m_i = (-1)^{i+1} 2 \sqrt{P_L P_S} G_{ele}, \quad i = 0, 1. \quad (6.54)$$

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

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

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-m_0)^2}{2\sigma^2}}. \quad (6.55)$$

The BER is calculated in the following manner,

$$BER = \frac{1}{2} \int_0^{+\infty} f(x|\Delta\theta = \pi) dx + \frac{1}{2} \int_{-\infty}^0 f(x|\Delta\theta = 0) dx, \quad (6.56)$$

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

$$BER = \int_0^{+\infty} f(x|\Delta\theta = \pi) dx = \frac{1}{2} \operatorname{erfc} \left( \frac{-m_0}{\sqrt{2}\sigma} \right) \quad (6.57)$$

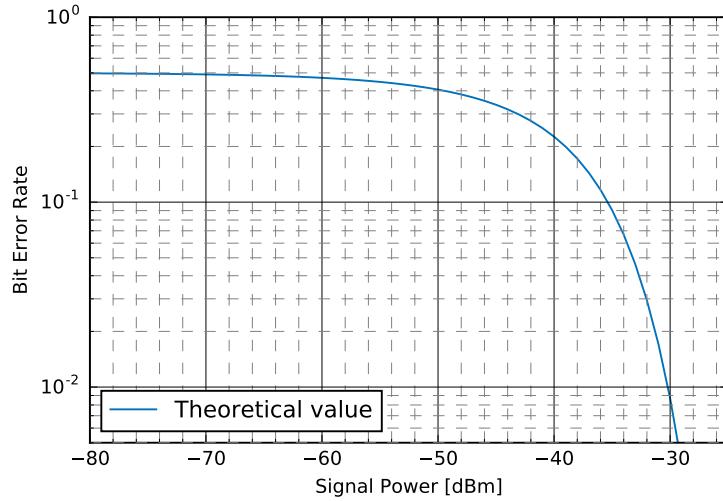


Figure 6.21: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm.

### 6.3.2 Simulation Analysis

A diagram of the system being simulated is presented in the Figure 6.22. A random binary sequence is generated and encoded in an optical signal using BPSK modulation. The decoding of the optical signal is accomplished by an homodyne receiver, which combines the

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

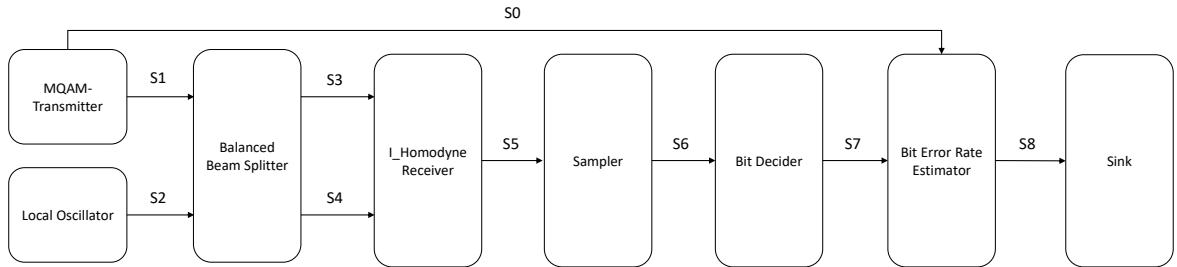


Figure 6.22: Overview of the BPSK system being simulated.

## Required files

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

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

### System Input Parameters

This system takes into account the following input parameters:

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

## Outputs

This system outputs the following objects:

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

## Bit Error Rate - Simulation Results

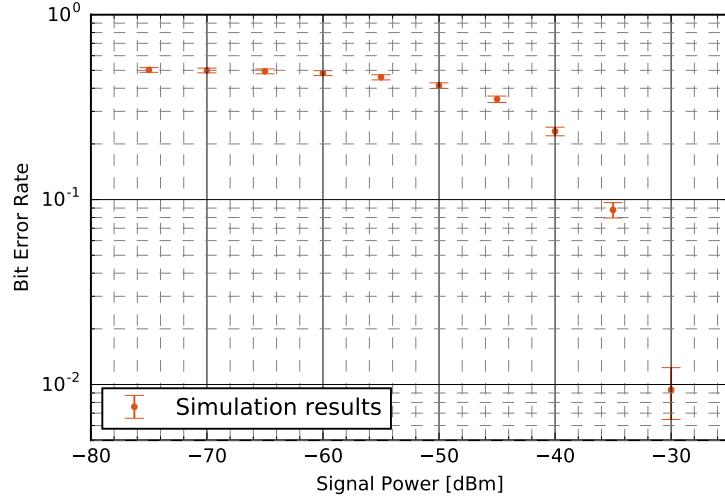


Figure 6.23: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm.

### 6.3.3 Comparative Analysis

The following results show the dependence of the error rate with the signal power assuming a constant Local Oscillator power of 0 dBm, the signal power was evaluated at levels between -70 and -25 dBm, in steps of 5 dBm between each. The simulation results are presented in orange with the computed lower and upper bounds, while the expected value, obtained from (6.57), is presented as a full blue line. A close agreement is observed between the simulation results and the expected value. The noise spectral density was set at  $5 \times 10^{-4}\sqrt{2} V^2$  [1].

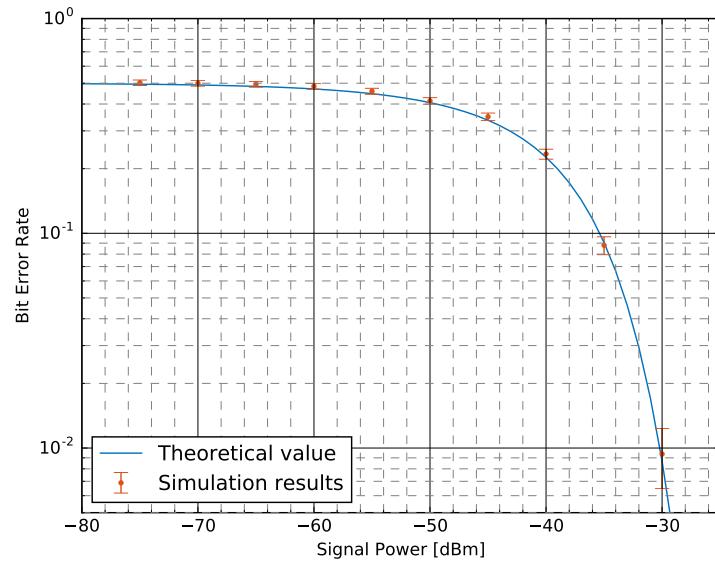


Figure 6.24: Bit Error Rate in function of the signal power in dBm at a constant local oscillator power level of 0 dBm. Theoretical values are presented as a full blue line while the simulated results are presented as a errorbar plot in orange, with the upper and lower bound computed in accordance with the method described in 7.71

## References

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

## 6.4 Hamming Channel Encoder and Decoder

|                      |   |   |
|----------------------|---|---|
| <b>Students Name</b> | : | Luís Almeida (08/06/2018 - 10/07/2018)                                |
| <b>Goal</b>          | : | Implement a channel coder and decoder based on the Hamming algorithm. |

Hamming Channel Encoder/Decoder simulates a channel encoder and decoder that uses the Hamming Algorithm. The goal is to perform the encoding of supplied source data, then encode it using the Hamming Encoder, pass the encoded data through a channel that implements errors according to a defined probability and finally perform the decoding using the Hamming Decoder. In the end a BER is performed to detect the possibility of errors.

### 6.4.1 Introduction

The Hamming code is a linear block code, was developed by Richard Hamming, is used in signal processing and telecommunications. Its use allows the transfer and storage of data in a safe and efficient way.

In telecommunications the Hamming codes used are generalizations of Hamming Code (7,4). These can detect errors up to two bits and correct up to one bit. In contrast, the parity code cannot fix errors, and can only detect an odd number of errors. Due to its simplicity the Hamming codes are widely used in computer memory (ECC). In this context, it is common to use an extended Hamming code with an extra parity bit.

In mathematical terms, Hamming codes are a class of binary linear codes. For each integer  $r \geq 2$  there is a block length  $n = 2^r - 1$  and message length  $k = 2^r - r - 1$ . Therefore, the Hamming code rate is  $R = k/n = 1 - r/(2^r - 1)$ , which is as high as possible for codes with distance 3 and block length  $2^r - 1$ . The parity matrix of a Hamming code is constructed by listing all columns of length  $r$  that are linearly independent. Hamming codes are special because they are perfect codes, that is, they reach the highest rate for the codes with their block length and a minimum distance of 3 [1].

### 6.4.2 Theoretical Analysis

One of the most used Hamming Codes is the (7, 4), that encodes 4 data bits into 7 bit block (3 extra parity bits). The extra 3 bits will enable the receiver to detect up to 2 errors and correct one of them.

In order to encode a block of 4 data bits it is required to have a generator matrix  $G$ . Multiplying  $[d_1 \ d_2 \ d_3 \ d_4] \times G(d)$  will result in the desired  $1 \times 7$  encoded word.

The  $G$  matrix can be defined for the Hamming Code (7, 4) as follows:

$$G = [p|d]; \quad (6.58)$$

where,

$$p = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad (6.59)$$

$$d = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (6.60)$$

Also for decode it is required to create a decoding matrix  $H$ , and using the same example Hamming Code (7, 4), we can form the  $H$  matrix as follows.

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} \quad (6.61)$$

Using the  $H$  matrix we can detect the presence of errors in the received 7 bit block. If we multiply  $H$  by the 7 bit block the outcome will be a  $3 \times 1$  vector ( $err$ ) that if the sum of its values is different than zero it means an error was detected.

So to correct that error, we just need to compare the ( $err$ ) with the  $H$  matrix and search for the column that is equal to the obtained  $err$  vector. That indicates the bit that was received wrongly. To correct it, we just need to change the corresponding bit value.

After that to perform the decoding we just need to segment the last 4 bits from the 7 bit vector in order to recover the 4 data bits sent.

#### 6.4.3 Simulation Analysis

The project flow can be observed in Figure 6.25.

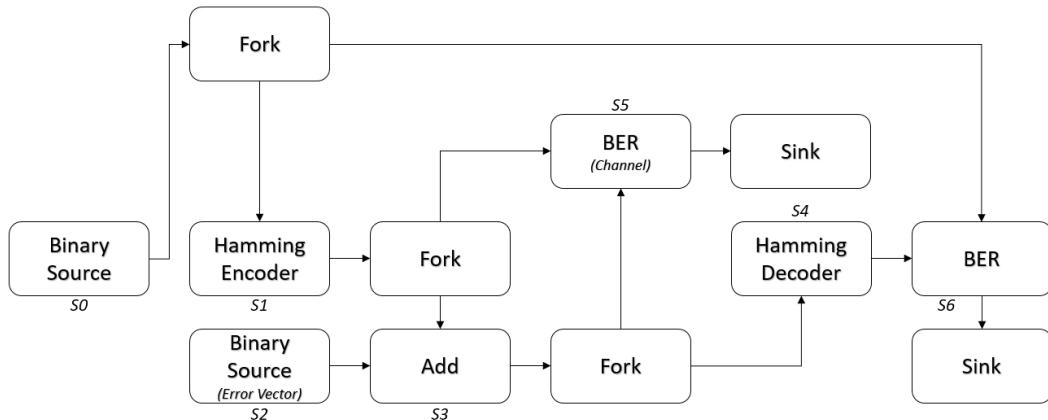


Figure 6.25: Hamming Encoder/Decoder Design Flow

Initially the project uses a Binary Source block in order to create a random number of bits that form the Data Signal  $S0$  (Figure 6.26).

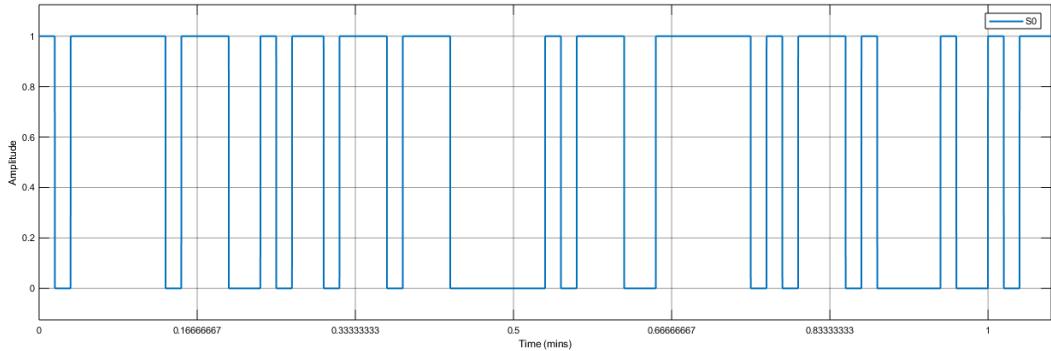


Figure 6.26: S0 Signal Example for Hamming Code (7, 4)

Next the  $S_0$  is feed into a Fork block in order to create a copy of the original signal,  $S_0$ , so it can be feed into the following processing chain and into the BER block.

After that  $S_0$  is encoded using the Hamming Encoder Block and we obtain the encoded signal  $S_1$ , according to the selected Hamming Code  $(n, k)$  (Figure 6.27).

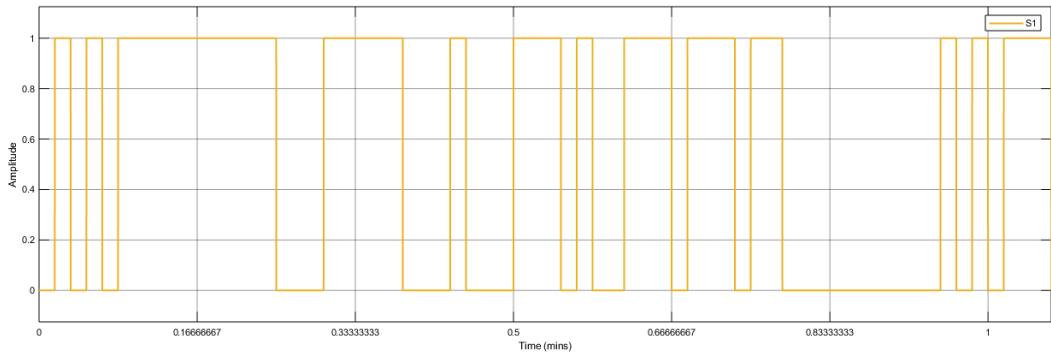


Figure 6.27: S1 Signal Example for Hamming Code (7, 4)

In order to simulate the occurrence of errors a second Binary Source block is used to generate the noise signal,  $S_2$ , that basically produces a one accordingly to a given percentage, where it represents an error (Figure 6.28).

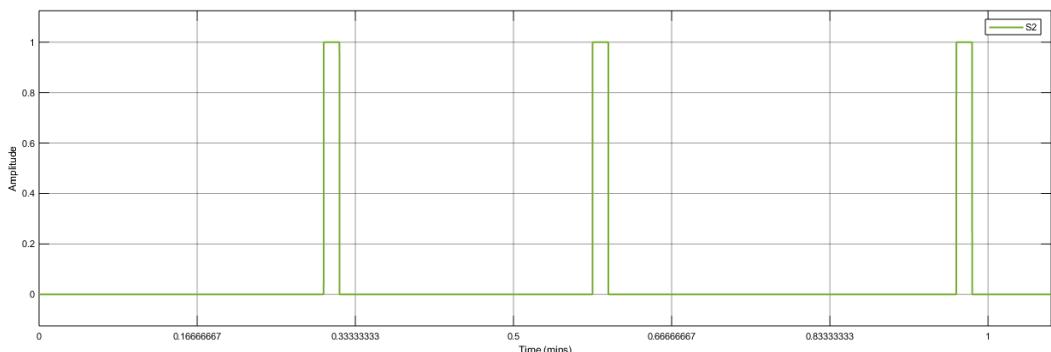


Figure 6.28: S2 Signal Example for Hamming Code (7, 4)

After that the data signal,  $S_0$ , is summed, using an add block, with the noise signal,  $S_2$ , and we obtain the data signal with errors,  $S_3$ , simulating a noise channel (Figure 6.29 and Figure 6.30).

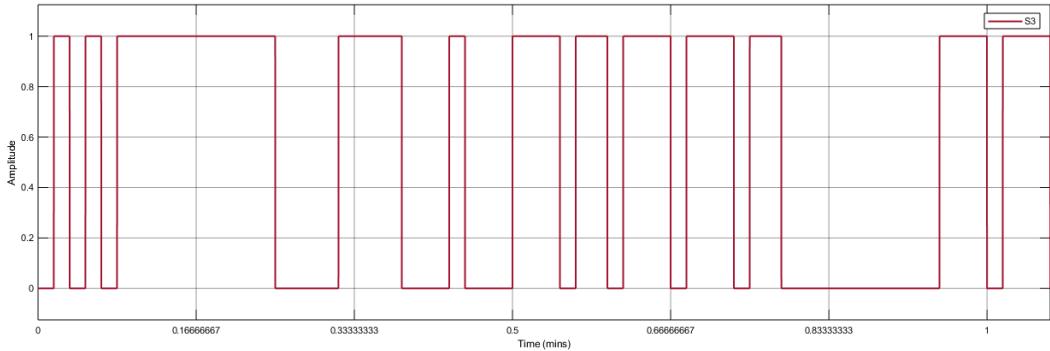


Figure 6.29:  $S_3$  Signal Example for Hamming Code (7, 4)

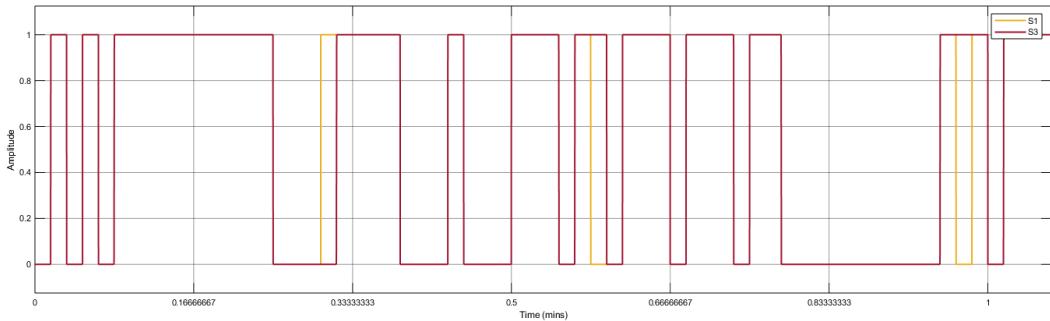


Figure 6.30: Comparison of  $S_1$  and  $S_3$  Signals

Then the obtained signal is decoded using the Hamming Decoder block, returning the decoded signal,  $S_4$  (Figure 6.31).

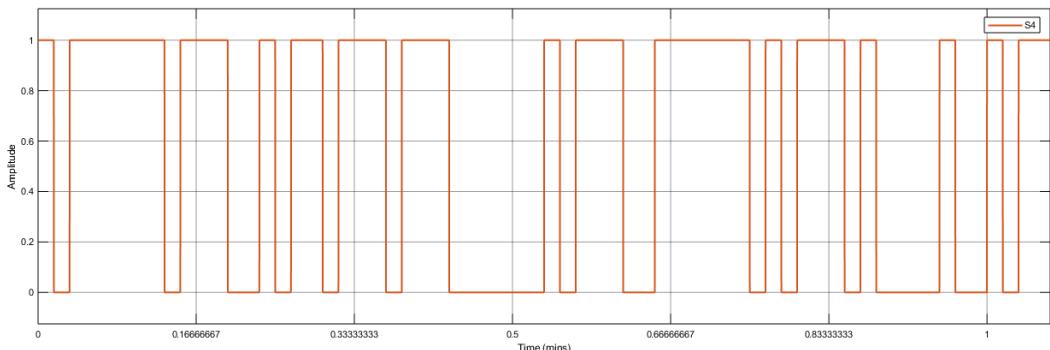


Figure 6.31:  $S_4$  Signal Example for Hamming Code (7, 4)

Finally a BER comparison using the BER block is performed in order to evaluate the encoding and decoding chain (Figure 6.32).

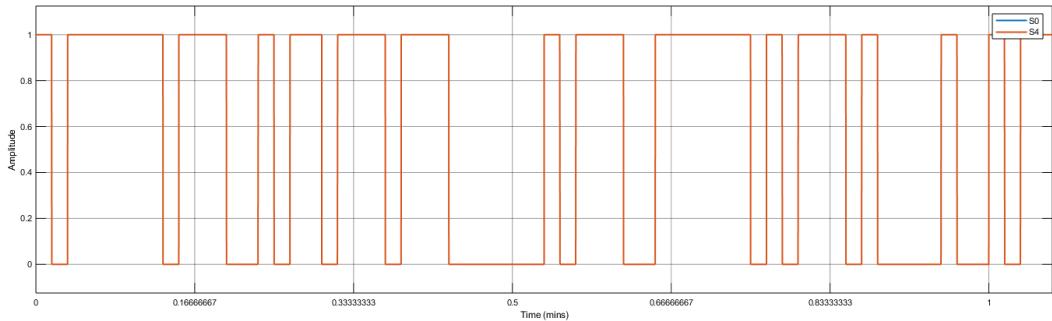


Figure 6.32: Comparison of S0 and S4 Signals (Original and Decoded Signals)

### BER Comparison

In this subsection we show the observed BER relations between the inputted error probability in the channel and the actual BER on the output of the channel (Figure 6.33) and the relation between the inputted probability of error of the channel and the BER on the output of the Hamming Decoder (Figure 6.34) .

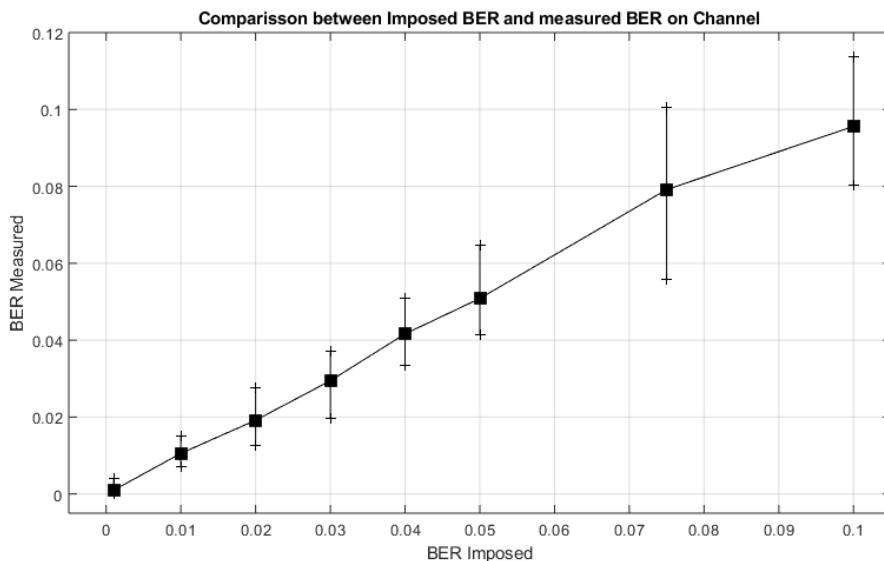


Figure 6.33: Comparison between Imposed BER and measured BER on Channel

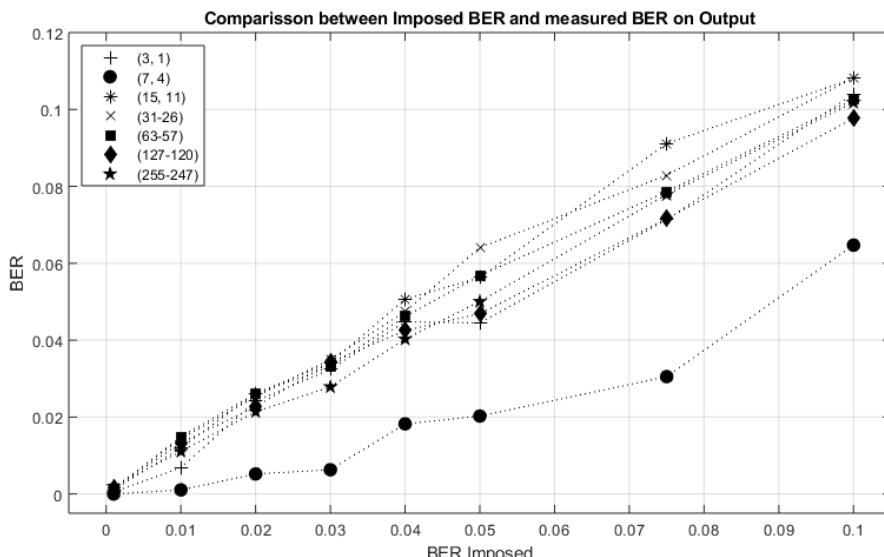


Figure 6.34: Comparison between Imposed BER and measured BER on Output of Decoder

## Required files

This project is composed by the header and source files described in following tables.

| Header Files               |          |        |
|----------------------------|----------|--------|
| File                       | Comments | Status |
| add_20180620.h             |          | ✓      |
| binary_source_20180523.h   |          | ✓      |
| bit_error_rate_20180424.h  |          | ✓      |
| fork_20180112.h            |          | ✓      |
| hamming_coder_20180608.h   |          | ✓      |
| hamming_decoder_20180608.h |          | ✓      |
| netxpto_20180418.h         |          | ✓      |
| sink_20180118.h            |          | ✓      |

| Source Files                                  |          |        |
|---|----------|--------|
| File  | Comments | Status |
| add_20180620.cpp                              |          | ✓      |
| binary_source_20180523.cpp                    |          | ✓      |
| bit_error_rate_20180424.cpp                   |          | ✓      |
| fork_20180112.cpp                             |          | ✓      |
| hamming_coder_20180608.cpp                    |          | ✓      |
| hamming_decoder_20180608.cpp                  |          | ✓      |
| netxpto_20180418.cpp                          |          | ✓      |
| sink_20180118.cpp                             |          | ✓      |
| eit_25828_hamming_channel_encoder_decoder.cpp |          | ✓      |

## System Input Parameters

In order to successfully run this project it is required to set the following input parameters:

- *probabilityOfZero\_ErrorVector* ( $p_0$ ) - Defines the probability for introducing errors (in order to simulate a non perfect channel). The probability of an error is  $p_1 = 1 - p_0$ .
- *hammingCode\_nBits n* - Defines the Hamming Code coded word size.
- *hammingCode\_kBits k* - Defines the Hamming Code data size.

The  $n$  and  $k$ , form the Hamming Code  $(n, k)$ , and the available combination of values for each of these variables can be viewed in the Hamming Encoder/Decoder in the Library Section.

## Inputs

This project doesn't take any input.

## Outputs

This project outputs the following signals:

- $S0.sgn$  - The source binary signal.
- $S1.sgn$  - The encoded signal using the Hamming Code  $(n, k)$ .
- $S2.sgn$  - The noise signal.
- $S3.sgn$  - The resulting signal of adding the encoded signal ( $S1.sgn$ ) with the noise signal ( $S2.sgn$ ).
- $S4.sgn$  - The decoded signal using the Hamming Code  $(n, k)$ .
- $S5.sgn$  - The BER signal on the channel.
- $S6.sgn$  - The BER signal after decoding.

## Open Issues

The only issue found, at the time of the construction of these blocks, was not related to the developed blocks, but regarding another block, the Binary Source (*binary\_source\_20180523.h*).

The mode *Random* doesn't work as intended. The block requires the user to define a probability of Zero for the output of the block, but if the output buffer becomes full on the next time the block is called again to continue producing bits, the sequence is exactly the same (Figure 6.35).

Observing the Figure below we can see a pattern that repeats itself over and over again instead of obtaining a completely random error vector.

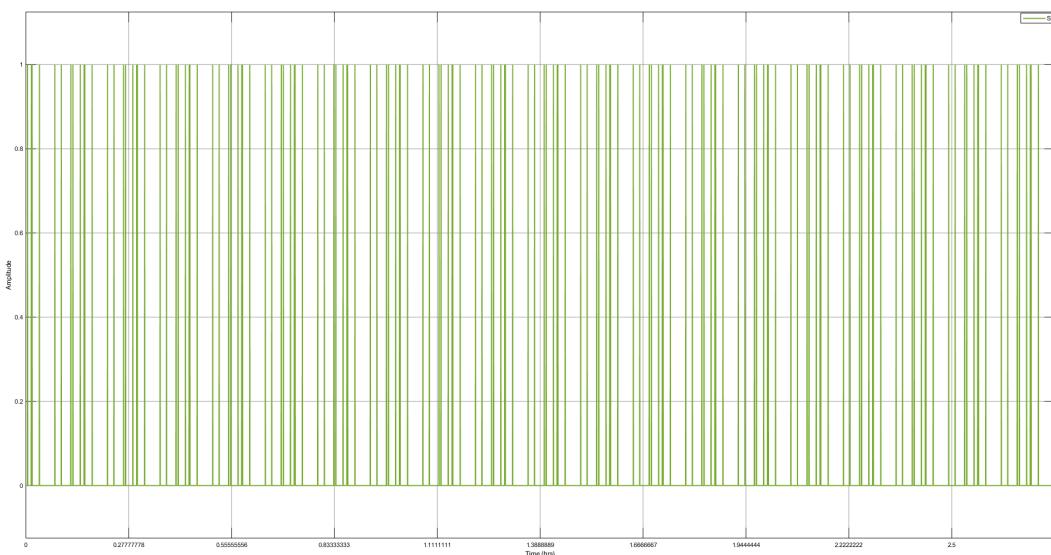


Figure 6.35: Binary Source - Random mode - Observed Error

This behavior needs to be resolved, not only to properly set this mode in the Binary Source, but also to improve the results of the developed blocks, since it is my belief that the curves observed in Figure 6.34 would improved, reducing the BER observed on the output of the decoder for the same imposed BER.

## References

- [1] Venkatesan Guruswami. *Introduction to Coding Theory - Notes 1: Introduction, linear codes*. 2010. URL: <https://www.cs.cmu.edu/venkatg/teaching/codingtheory/notes/notes1.pdf>.

## 6.5 Huffman Source Encoder and Decoder

|                      |   |   |
|----------------------|---|---|
| <b>Students Name</b> | : | Marina Jordao (21/06/2018)  |
| <b>Goal</b>          | : | Implement source code efficiency using a Huffman encoder and decoder. |
| <b>Directory</b>     | : | sdf/eit_45550_estimator_source_code_efficiency.                       |

The main goal of this Source Code Efficiency is to estimate the code efficiency provided by a Huffman encoder. First, the source entropy is calculated and then, a Huffman encoder is implemented, in order to encode the message that was generated by a binary source. Thereafter, the efficiency is estimated, by using entropy and message length. Lastly, a Huffman decoder is implemented in order to validate the results. It is intend to apply this strategy for a Huffman Encoder with 2, 3 and 4 order, for a binary code with 0 and 1.

### 6.5.1 Theoretical Analysis

To develop an efficiency estimator several blocks were used/developed. The block diagram with the several blocks used in this project can be seen in Figure 6.36. First, the Source block was used in order to provided the binary signal. Then, a Huffamn Encoder block was elaborated for 2, 3 and 4 orders. The efficiency estimator calculates the efficiency of the code and the Huffman Decoder will be decoded the signals for a 2,3 and 4 orders, to valide the results. The last block is the Sink, as expected.



Figure 6.36: Block diagram of source code efficiency implementation.

In the following sections each block will be explained in detailed.

#### Entropy Estimator

In the Entropy Estimator block the main goal is to calculated the entropy from a binary source for the case of a binary source composed by 0 and 1. In this sense, to calculated the entropy the equation 6.62 was applied.

$$H(x) = \sum_{i=1}^K P(a_i) \log_2 \frac{1}{P(a_i)} \quad (6.62)$$

This block does not have inputs variables.

#### Hufmman Encoder

The Hufman Encoder block aims to encode a binary signal, composed by 0 and 1, for a order of 2, 3 and 4. This block has 2 input variables, the probabilityOfZero and sourceOrder. The

probabilityOfZero variable is a double value with a range from 0 until 1. The sourceOrder variable is a integer value, where only 2, 3 and 4 values are accepted.

In this Huffman encoder, the probabilityOfZero will defined the type of codification, as well as, the source order. For a probability of zero less or equal than a probability of one, the encoder process is presented in tables 6.5, 6.7 and 6.9, for a source order of 2, 3 and 4 respectively. Otherwise, if probability of zero is greater than probability of one, the encoder process is shown in tables 6.6, 6.8 and 6.10, for a source order of 2, 3 and 4 respectively.

Table 6.5: Huffman Encoder 2 Order, when probability of Zero less or equal than probability of One

| <b>Message</b> | <b>Huffman Encoder Order 2</b> |
|----------------|--------------------------------|
| 00             | 111                            |
| 01             | 110                            |
| 10             | 10                             |
| 10             | 0                              |

Table 6.6: Huffman Encoder 2 Order, when probability of Zero greater than probability of One

| <b>Message</b> | <b>Huffman Encoder Order 2</b> |
|----------------|--------------------------------|
| 11             | 111                            |
| 10             | 110                            |
| 01             | 10                             |
| 00             | 0                              |

Table 6.7: Huffman Encoder 3 Order, when probability of Zero less or equal than probability of One

| <b>Message</b> | <b>Huffman Encoder Order 3</b> |
|----------------|--------------------------------|
| 000            | 1111111                        |
| 001            | 1111110                        |
| 010            | 111110                         |
| 011            | 11110                          |
| 100            | 1110                           |
| 101            | 110                            |
| 110            | 10                             |
| 111            | 0                              |

Table 6.8: Huffman Encoder 3 Order, when probability of Zero greater than probability of One

| <b>Message</b> | <b>Huffman Encoder Order 3</b> |
|----------------|--------------------------------|
| 111            | 1111111                        |
| 110            | 1111110                        |
| 101            | 111110                         |
| 100            | 11110                          |
| 011            | 1110                           |
| 010            | 110                            |
| 001            | 10                             |
| 000            | 0                              |

Table 6.9: Huffman Encoder 4 Order, when probability of Zero less or equal than probability of One

| <b>Message</b> | <b>Huffman Encoder Order 4</b> |
|----------------|--------------------------------|
| 0000           | 111111111111111                |
| 0001           | 111111111111110                |
| 0010           | 111111111111110                |
| 0011           | 111111111111110                |
| 0100           | 1111111111110                  |
| 0101           | 11111111110                    |
| 0110           | 1111111110                     |
| 0111           | 111111110                      |
| 1000           | 11111110                       |
| 1001           | 1111110                        |
| 1010           | 111110                         |
| 1011           | 11110                          |
| 1100           | 1110                           |
| 1101           | 110                            |
| 1110           | 10                             |
| 1111           | 0                              |

Table 6.10: Huffman Encoder 4 Order, when probability of Zero greater than probability of One

| Message | Huffman Encoder Order 4 |
|---------|-------------------------|
| 1111    | 11111111111111          |
| 1110    | 11111111111110          |
| 1101    | 11111111111110          |
| 1100    | 11111111111110          |
| 1011    | 111111111110            |
| 1010    | 111111111110            |
| 1001    | 1111111110              |
| 1000    | 111111110               |
| 0111    | 11111110                |
| 0110    | 11111110                |
| 0101    | 1111110                 |
| 0100    | 111110                  |
| 0011    | 11110                   |
| 0010    | 110                     |
| 0001    | 10                      |
| 0000    | 0                       |

### Efficiency Estimator

The Efficiency Estimator (Source Code Efficiency) block goal is to calculate the efficiency of the code, for a order code of 2, 3 and 4. This block has 2 input variables, the probabilityOfZero and sourceOrder. In this sense, to calculate the efficiency, the equation 6.63 was used, where L is length codeword.

$$\eta = \frac{Hr(x)}{L} \quad (6.63)$$

### Huffman Decoder

The Hufman Decoder block aims to decode signal from Huffman Encoder block, composed by 0 and 1, for a order of 2, 3 and 4. This block has 2 input variables, the probabilityOfZero and sourceOrder. The probabilityOfZero variable is a double value with a range from 0 until 1. The sourceOrder variable is a integer value, where only 2, 3 and 4 are accepted.

In this Huffman decoder, the probabilityOfZero will defined the type of descodification, as well as, the source order. For a probability of zero less or equal than a probability of one, the decoder process is presented in tables 6.11, 6.13 and 6.15, for a source order of 2, 3 and 4 respectively. Otherwise, if probability of zero is greater than probability of one, the decoder process is shown in tables 6.12, 6.14 and 6.16, for a source order of 2, 3 and 4 respectively.

Table 6.11: Huffman Decoder 2 Order, when probability of Zero less or equal than probability of One

| <b>Message</b> | <b>Huffman Decoder Order 2</b> |
|----------------|--------------------------------|
| 111            | 00                             |
| 110            | 01                             |
| 10             | 10                             |
| 0              | 11                             |

Table 6.12: Huffman Decoder 2 Order, when probability of Zero greater than probability of One

| <b>Message</b> | <b>Huffman Decoder Order 2</b> |
|----------------|--------------------------------|
| 111            | 11                             |
| 110            | 10                             |
| 10             | 01                             |
| 0              | 00                             |

Table 6.13: Huffman Decoder 3 Order, when probability of Zero less or equal than probability of One

| <b>Message</b> | <b>Huffman Decoder Order 3</b> |
|----------------|--------------------------------|
| 1111111        | 000                            |
| 1111110        | 001                            |
| 111110         | 010                            |
| 11110          | 011                            |
| 1110           | 100                            |
| 110            | 101                            |
| 10             | 110                            |
| 0              | 111                            |

Table 6.14: Huffman Decoder 3 Order, when probability of Zero greater than probability of One

| <b>Message</b> | <b>Huffman Decoder Order 3</b> |
|----------------|--------------------------------|
| 1111111        | 111                            |
| 1111110        | 110                            |
| 111110         | 101                            |
| 11110          | 100                            |
| 1110           | 011                            |
| 110            | 010                            |
| 10             | 001                            |
| 0              | 000                            |

Table 6.15: Huffman Decoder 4 Order, when probability of Zero less or equal than probability of One

| <b>Message</b>  | <b>Huffman Decoder Order 4</b> |
|-----------------|--------------------------------|
| 111111111111111 | 0000                           |
| 111111111111110 | 0001                           |
| 111111111111110 | 0010                           |
| 111111111111110 | 0011                           |
| 1111111111110   | 0100                           |
| 11111111110     | 0101                           |
| 1111111110      | 0110                           |
| 111111110       | 0111                           |
| 11111110        | 1000                           |
| 1111110         | 1001                           |
| 111110          | 1010                           |
| 11110           | 1011                           |
| 1110            | 1100                           |
| 110             | 1101                           |
| 10              | 1110                           |
| 0               | 1111                           |

Table 6.16: Huffman Decoder 4 Order, when probability of Zero greater than probability of One

| Message        | Huffman Decoder Order 4 |
|----------------|-------------------------|
| 11111111111111 | 1111                    |
| 11111111111110 | 1110                    |
| 11111111111110 | 1101                    |
| 11111111111110 | 1100                    |
| 11111111111110 | 1011                    |
| 111111111110   | 1010                    |
| 11111111110    | 1001                    |
| 1111111110     | 1000                    |
| 111111110      | 0111                    |
| 11111110       | 0110                    |
| 1111110        | 0101                    |
| 11110          | 0100                    |
| 1110           | 0011                    |
| 110            | 0010                    |
| 10             | 0001                    |
| 0              | 0000                    |

In table 6.41 are presented the input parameters of the system.

### 6.5.2 Simulation Analysis

The simulation implementation will be described in order to implement the Source Code Efficiency project. In Figure 6.37 the block diagram of simulation process is shown. For this simulation, the value of probabilityOfZero variable was 0.05.

First, the Binary source block is executed, in order to generate a sequence of random binary bits, which results in the S0 signal. In Figure 6.38 can be seen an example of a S0 signal, as a result of Binary Source code.

After acquired the S0 signal, a fork was made to copy this signal. Four copies were created, one copy of this signal was applied in Entropy Estimator block, other in Huffman Encoder 2 order, other in Huffman Encoder 3 order and the last copy was applied in Huffman Encoder 4 order.

The first copy was applied to Entropy Estimator block to estimate the entropy value of this random binary code, the resulting entropy signal from this block is shown in Figure 6.39. As expected, the entropy value converges to 1.

The second copy was applied in the Hufman Encoder, with a source order of 2. In Figure 6.40 is presented the S1 signal, which is the result of Hufman Encoder 2 order, when the S0 signal was encoded. Then, a fork was applied to copy the signal S1 in 2 signals, one to applied in the Efficiency Estimator block and other to applied in the Huffman Decoder block, using a source order of 2. From the Efficiency Estimator arises the signal X2, wich can be

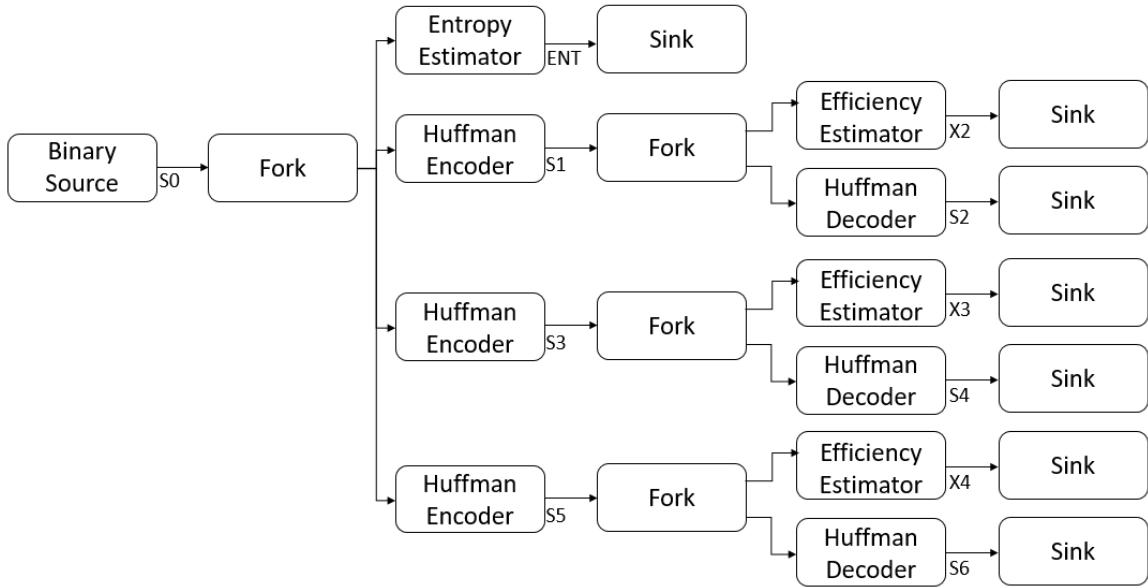
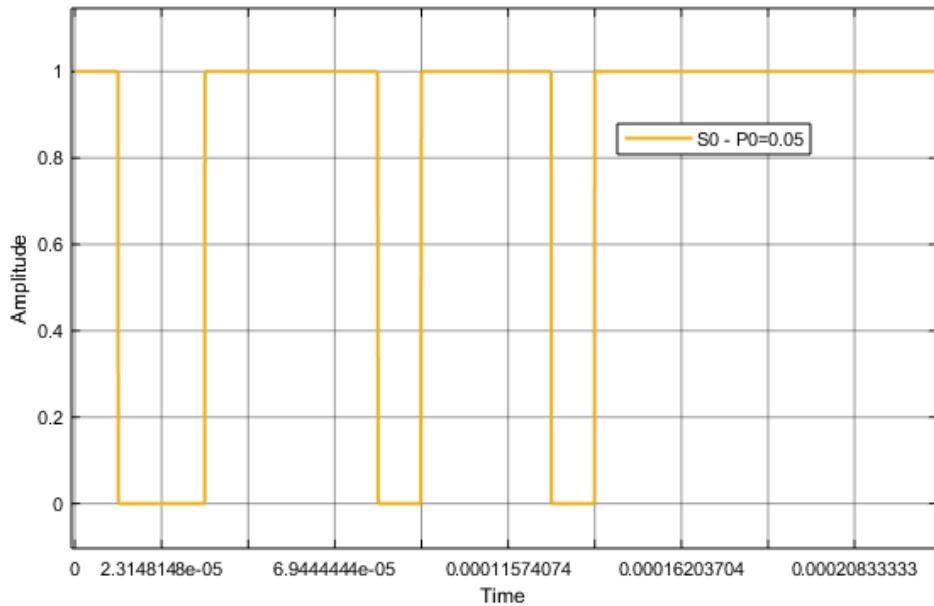


Figure 6.37: Simulation Block diagram.



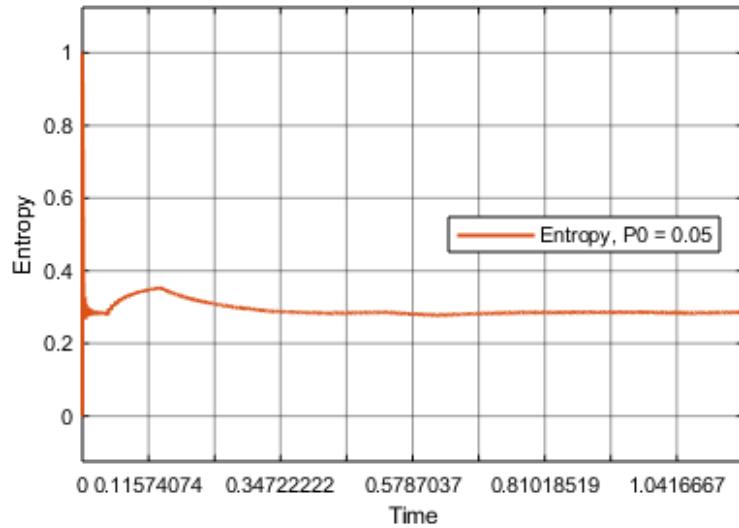


Figure 6.39: Entropy results for a binary source. The entropy theoretical value is 0.2864.

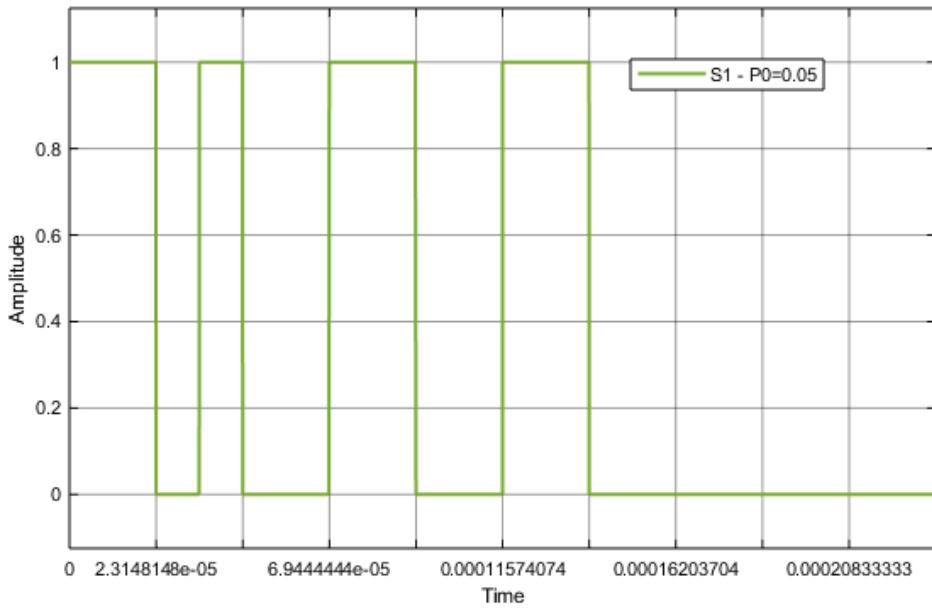


Figure 6.40: S1 Signal from Huffman Encoder 2 order.

block, using a source order of 3. From the Efficiency Estimator arises the signal X3, which can be seen in Figure 6.47 and the decoder signal S4 from Huffman Decoder block, for a order of 3 appears in Figure 6.43.

The fourth copy of signal S0 was applied in the Huffman Encoder, with a source order of 4. In Figure 6.44 is presented the S5 signal, which is the result of Huffman Encoder 4 order, when the S0 signal was encoded. Then, a fork was applied to copy the signal S5 in 2 signals, one to applied in the Efficiency Estimator block and other to applied in the Huffman Decoder

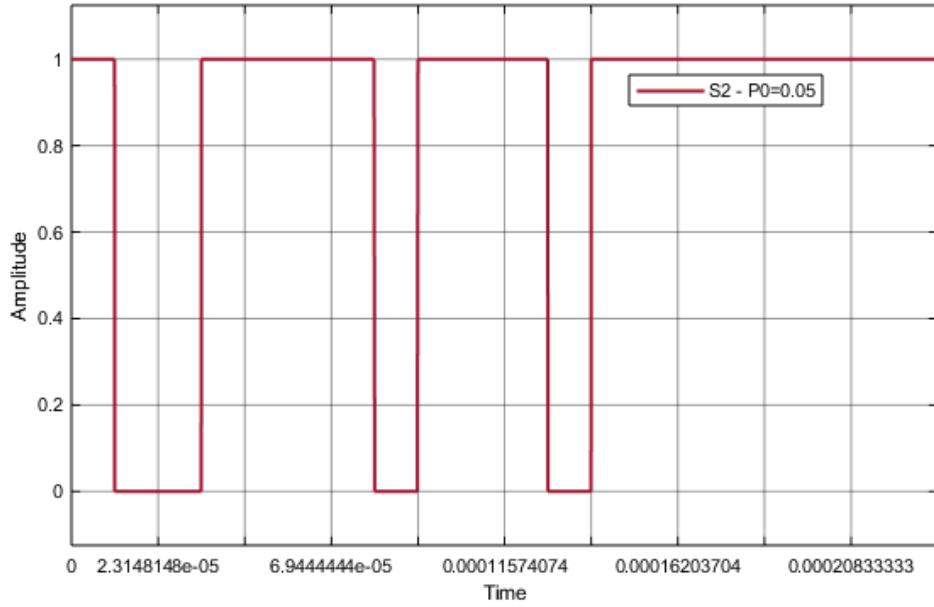


Figure 6.41: S2 Signal from Huffman Decoder 2 order.

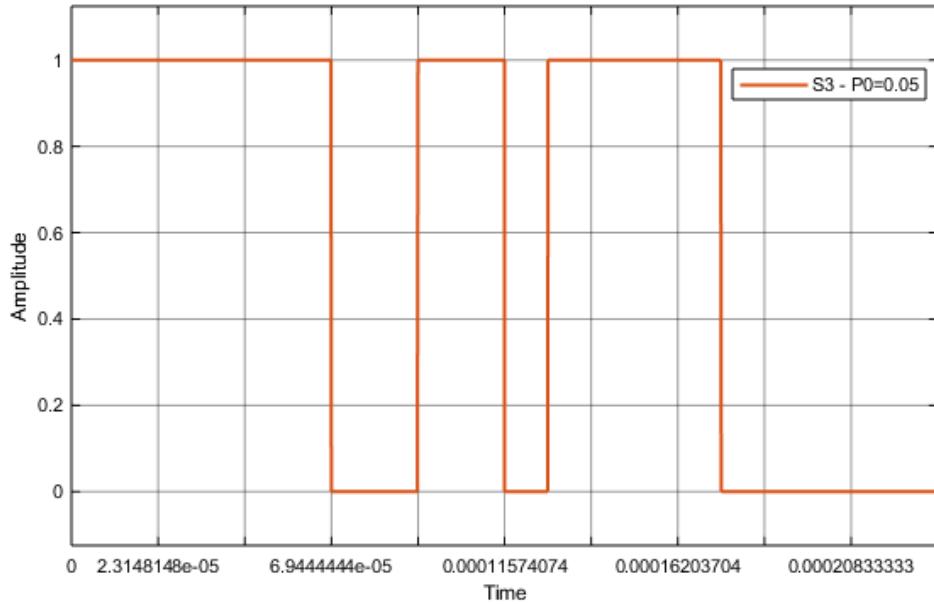


Figure 6.42: S3 Signal from Huffman Encoder 3 order.

block, using a source order of 4. From the Efficiency Estimator arises the signal X4, which can be seen in Figure 6.47 and the decoder signal S6 from Huffman Decoder block, for a order of 4 appears in Figure 6.45.

In order to validate that the encoder and decoder Huffman process was well developed for 2, 3 and 4 order, the original signal S0 was compared with signals S2, S4 and S6, which

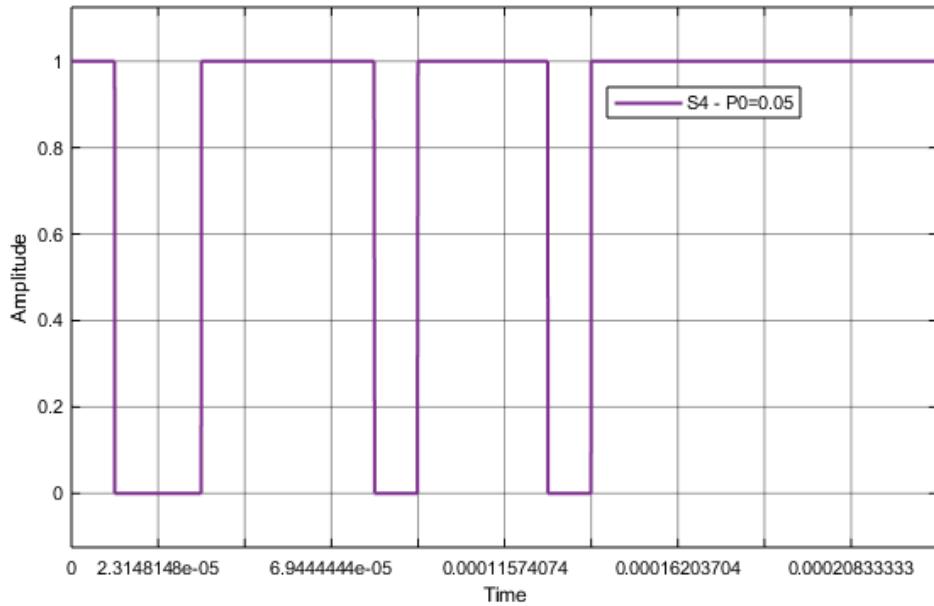


Figure 6.43: S4 Signal from Huffman Decoder 3 order.

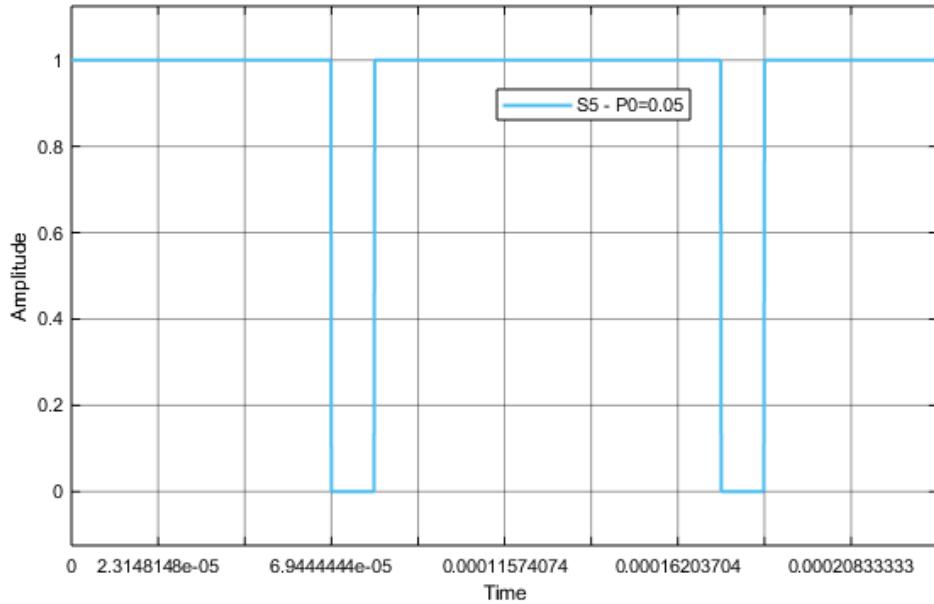


Figure 6.44: S5 Signal from Huffman Encoder 4 order.

are the decoded signals of a source order of 2, 3 and 4 respectively. This comparation of signals can be seen in Figure 6.46. As can be seen, the signals are overlapping, which proves that the coding and decoding process were well applied.

In Figure 6.47 can be seen the efficiency results for a source order of 2, 3 and 4 respectively. As expected, when the source order increases, the efficiency of the code decreases.

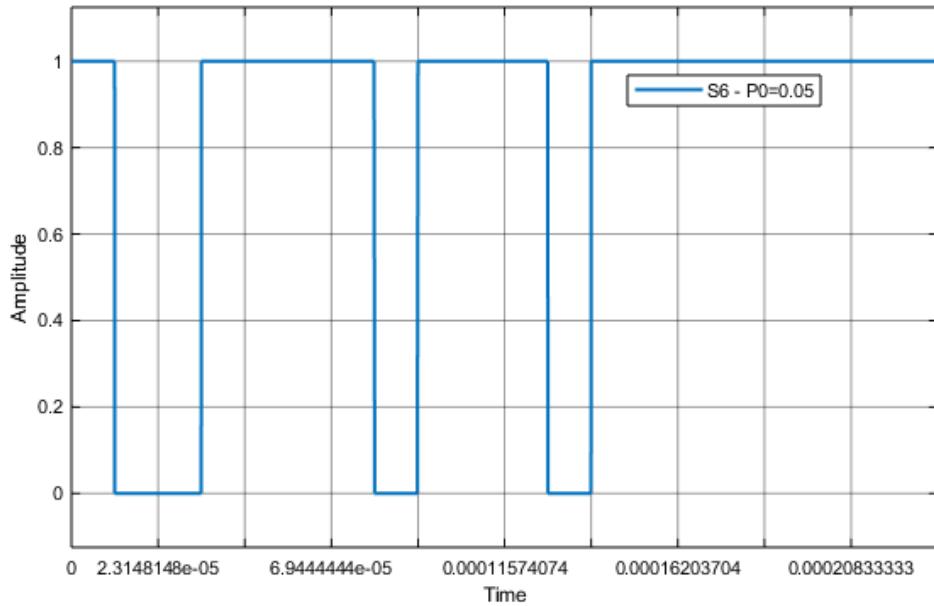


Figure 6.45: S6 Signal from Huffman Decoder 4 order.

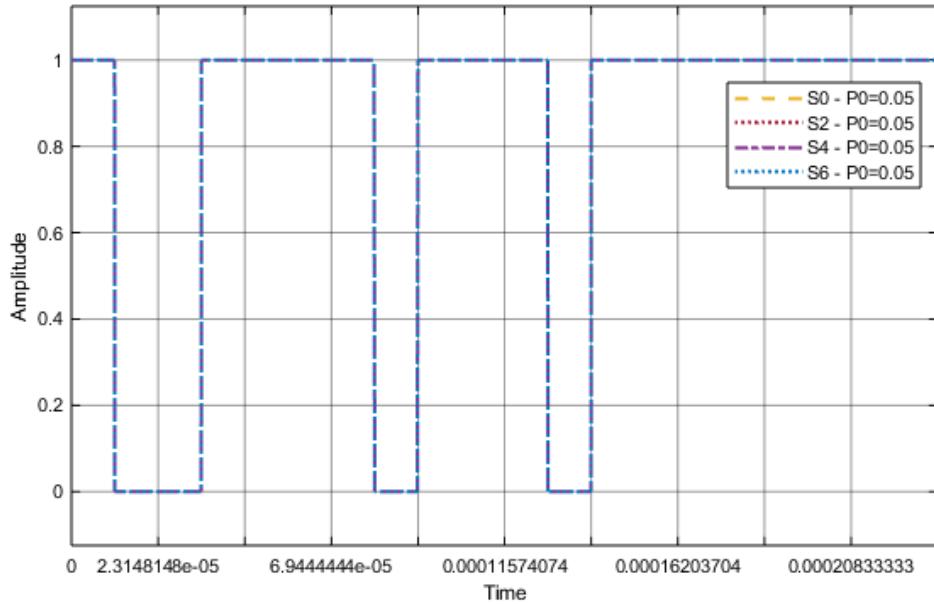


Figure 6.46: S0, S2, S4 and S6 signals comparation.

In the end, the sink block was used to empty the buffer.

The project is composed by several parts. Table 6.17 shows the header files used to implement the simulation presented in Figure 6.37. The source files are presented in table 6.44 and finally, in table 6.19 are shown the system signals, with the signal type information and description.

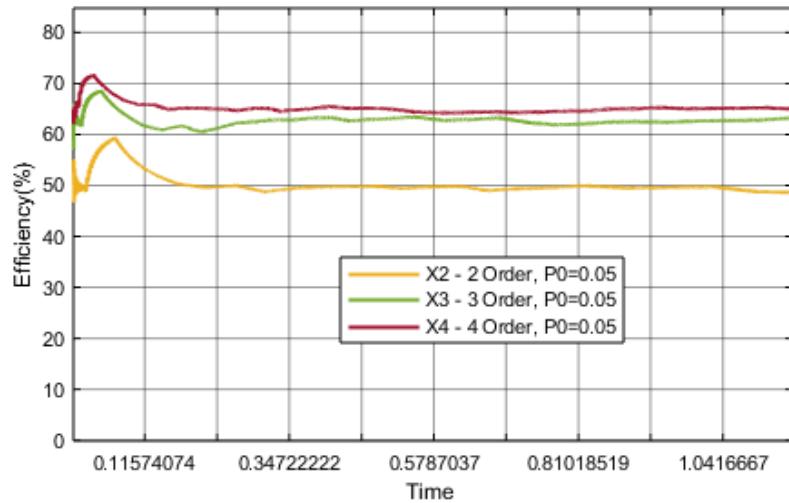


Figure 6.47: Efficiency results for 2, 3 and 4 orders using Huffman Encoder. The efficiency theoretical values for each order are 49.92%, 63.65% and 65.46%, respectively.

Table 6.17: Header Files

| File name                         | Description | Status |
|-----------------------------------|-------------|--------|
| binary_source_20180523.h          |             | ✓      |
| entropy_estimator_20180621.h      |             | ✓      |
| fork_20180112.h                   |             | ✓      |
| huffman_decoder_20180621.h        |             | ✓      |
| huffman_encoder_20180621.h        |             | ✓      |
| netxpto_20180418.h                |             | ✓      |
| sink_20180118.h                   |             | ✓      |
| source_code_efficiency_20180621.h |             | ✓      |

Table 6.18: Source Files

| File name  | Description | Status |
|--|-------------|--------|
| binary_source_20180523.cpp                         |             | ✓      |
| eit_45550_estimator_source_code_efficiency_sdf.cpp |             | ✓      |
| entropy_estimator_20180621.cpp                     |             | ✓      |
| fork_20180112.cpp                                  |             | ✓      |
| huffman_decoder_20180621.cpp                       |             | ✓      |
| huffman_encoder_20180621.cpp                       |             | ✓      |
| netxpto_20180418.cpp                               |             | ✓      |
| sink_20180118.cpp                                  |             | ✓      |
| source_code_efficiency_20180621.cpp                |             | ✓      |

Table 6.19: System Signals

| Signal name | Signal type                           | Description  |
|-------------|---------------------------------------|--|
| S0          | Binary                                | Binary signal which results from Source block                                  |
| ENT         | TimeContinuousAmplitudeContinuousReal | Entropy signal which results from Entropy Estimator block                      |
| S1          | Binary                                | Signal encoded by Huffman Encoder block, with order 2                          |
| X2          | TimeContinuousAmplitudeContinuousReal | Efficiency signal which results from Source Code Efficiency block with order 2 |
| S2          | Binary                                | Binary signal decoder by Huffman Decoder block, with order 2                   |
| S3          | Binary                                | Signal encoded by Huffman Encoder block, with order 3                          |
| X3          | TimeContinuousAmplitudeContinuousReal | Efficiency signal which results from Source Code Efficiency block with order 3 |
| S4          | Binary                                | Binary signal decoder by Huffman Decoder block, with order 3                   |
| S5          | Binary                                | Signal encoded by Huffman Encoder block, with order 4                          |
| X4          | TimeContinuousAmplitudeContinuousReal | Efficiency signal which results from Source Code Efficiency block with order 4 |
| S6          | Binary                                | Binary signal decoder by Huffman Decoder block, with order 4                   |

To run this simulation, different values can be applied to the probabilityOfZero variable.

## 6.6 Arithmetic Encoding & Decoding

|                      |   |   |
|----------------------|---|---|
| <b>Students Name</b> | : | Diogo Barros (46084)  |
| <b>Starting Date</b> | : | July 17, 2018   |
| <b>Goal</b>          | : | Integer implementation of Arithmetic encoding and decoding. |

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

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

### 6.6.1 Encoding Algorithm

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

The first reference to arithmetic encoding was made in [Abramson63]. However, the first practical implementations were only proposed in 1976 by Pasco [Pasco76] and Rissanen [Rissanen76]. The block diagram of the algorithm in its most simple floating point formulation is presented in Fig. 6.48.

The first step before the encoding starts is to compute the cumulative symbol probabilities of the source, given the probabilities of each individual symbol. The initial interval is set to [0,1) and is subdivided into sub intervals based on the symbol probabilities. The required resolution (number of bits) of the registers that represent the limits of the current interval is given by the lowest symbol probability as expressed in (6.64)

$$\text{Resolution} = \lceil (-\log_2(\min(P(x_n)))) \rceil + 1 \quad (6.64)$$

The encoding process starts by reading a symbol, identifying the corresponding sub interval in the current interval and updating the variables that save the current interval limits by using (6.65)-(6.67).

$$\text{delta} = \text{lim\_high} - \text{lim\_low} + 1 \quad (6.65)$$

$$\text{lim\_high} = \text{lim\_low} + \lfloor \text{delta} \cdot \text{count}(\text{symb} + 1) / \text{total\_count} \rfloor - 1 \quad (6.66)$$

$$\text{lim\_low} = \text{lim\_low} + \lfloor \text{delta} \cdot \text{count}(\text{symb}) / \text{total\_count} \rfloor \quad (6.67)$$

At this point we need to perform two operations. The first is to check if we are able to output any code bit. The second is to rescale the interval if its amplitude is small enough. These operations are done based on the three conditions (6.68), (6.69) and (6.70).

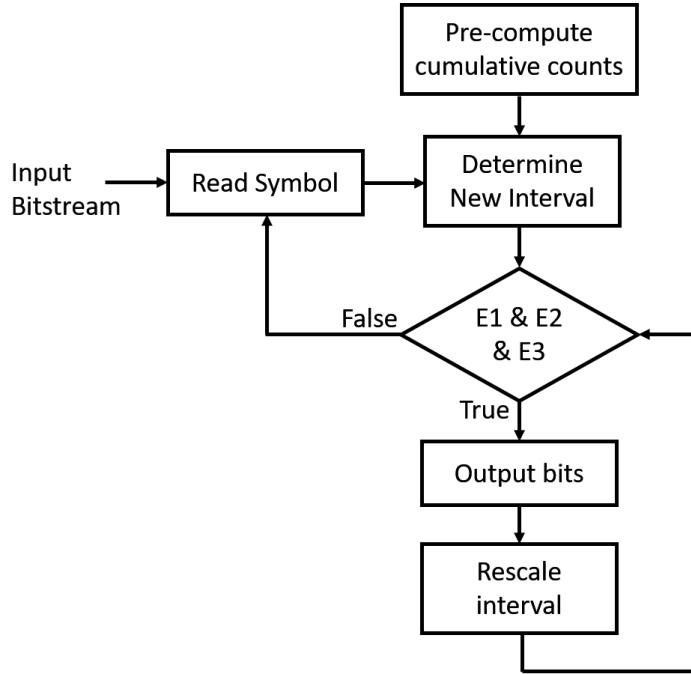


Figure 6.48: Block diagram of the algorithm for arithmetic encoding.

The algorithm keeps track of how many times condition (6.70) is met in succession and increments an internal extra bit counter.

$$E1 = (\text{lim\_low} < 0.5) \&\& (\text{lim\_high} < 0.5) \quad (6.68)$$

$$E2 = (\text{lim\_low} > 0.5) \&\& (\text{lim\_high} > 0.5) \quad (6.69)$$

$$E3 = (\text{lim\_low} > 0.25) \&\& (\text{lim\_high} < 0.75) \quad (6.70)$$

When (6.68) is met, the algorithm outputs a '0' and a number of extra '1' bits equal to the number of times (6.70) was met previously and the extra bit counter is reset. When (6.68) is met, the algorithm outputs a '1' and a number of '0' bits equal to the number of times (6.70) was met previously and the extra bit counter is reset. In the practical implementation, these conditions are tested by comparing the most significant bits of each register.

If any of these conditions is met, the amplitude of the interval is lower than 0.25 and needs to be rescaled by a factor of two, depending on which condition is met, as described in (6.71), (6.72) and (6.73).

$$E1 : \text{lim\_low} = 2 \cdot \text{lim\_low}; \text{lim\_high} = 2 \cdot \text{lim\_high}; \quad (6.71)$$

$$E2 : \text{lim\_low} = 2 \cdot (\text{lim\_low} - 0.5); \text{lim\_high} = 2 \cdot (\text{lim\_high} - 0.5); \quad (6.72)$$

$$E3 : \text{lim\_low} = 2 \cdot (\text{lim\_low} - 0.25); \text{lim\_high} = 2 \cdot (\text{lim\_high} - 0.25); \quad (6.73)$$

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

### 6.6.2 Decoding Algorithm

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

The decoding operations are the same as the ones performed by the coding algorithm with the exception of symbol identification and how the output bits are determined. The simplified block diagram of the encoder algorithm is presented in Fig. 6.49.

The decoder uses the same bit resolution as that of the encoder and starts by reading that number of bits from the coded bit stream, that becomes the "tag" value. The tag is then normalized as expressed in (6.74) and used to find the next symbol that was coded. This is done by finding the interval in the cumulative probability vector that contains it. Once the symbol is identified the code that corresponds to it can be sent to the output.

$$\text{tag\_norm} = \lfloor ((\text{tag} - \text{lim\_low} + 1) \cdot \text{total\_count} - 1) / (\text{lim\_high} - \text{lim\_low} + 1) \rfloor \quad (6.74)$$

The interval is updated in the same way as in the encoder and the conditions (6.68), (6.69) and (6.70) are computed to rescale the interval. Additionally the tag value is updated based on which of these conditions is met.

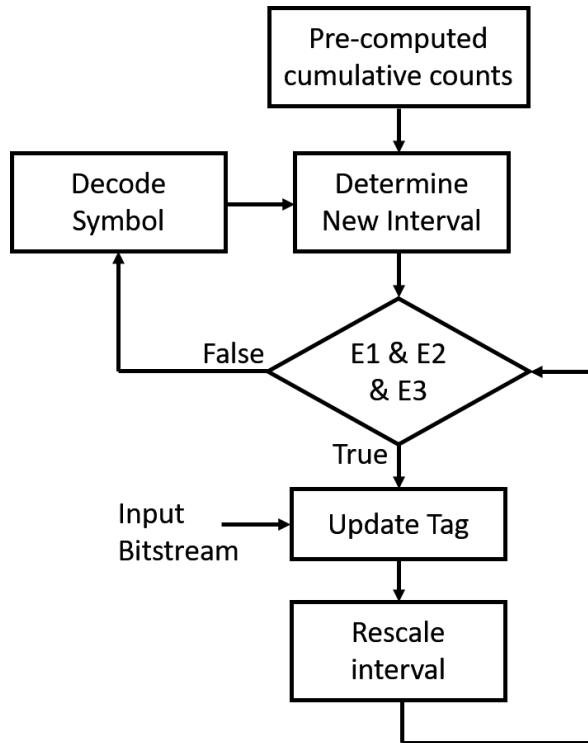


Figure 6.49: Block diagram of the algorithm for arithmetic decoding.

If either (6.68) or (6.69) is met, a new bit is read from the coded bit stream and added to the tag value multiplied by two. In the implemented code this is done right shifting the tag

bits by one (discarding the most significant bit) and adding the new bit to it. If (6.70) is met, the tag is updated in the same way but the new most significant bit is negated. When none of the conditions is met, a new symbol is decoded using the tag value normalized and the process continues until all symbols are decoded.

### 6.6.3 Encoding and Decoding Simulation Results

To test the implemented encoding and decoding algorithms, the system presented in Fig.6.50 was created in simulation. The system contains a binary source, the arithmetic encoding and decoding blocks and a bit-error-ratio computation block to ensure that the information is not degraded.

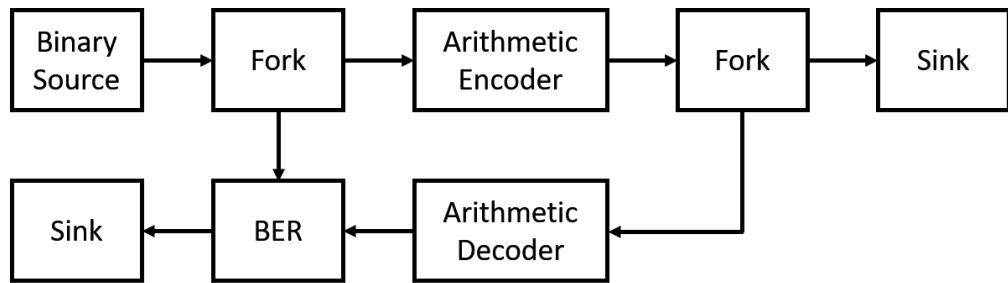


Figure 6.50: Block diagram of the floating point algorithm for arithmetic decoding.

A test simulation with a total of 45360 bits and a probability of zero of 0.1 was performed. the number of bits of the encoder was changed from 2 to 5 and the code efficiency, given by (6.75), was computed for each value. The results are presented in Fig.6.51. As expected, the code efficiency of the arithmetic algorithm is very close to optimum and increases as the number of coding bits increases.

$$\eta = \frac{H(n)}{L} \quad (6.75)$$

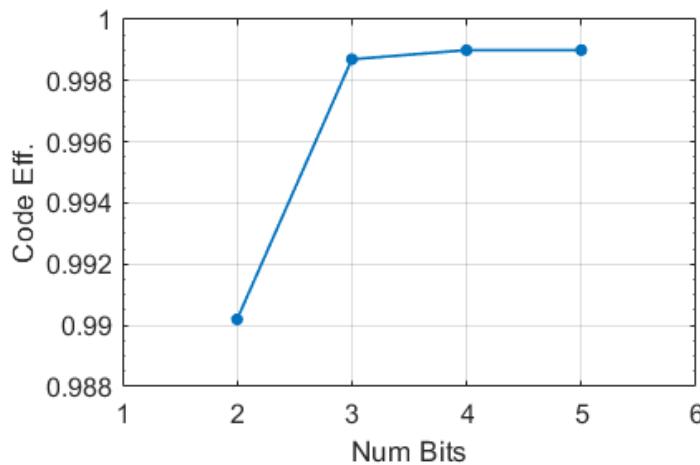


Figure 6.51: Code efficiency variation with the number of coding bits.

## 6.7 Mutual information estimation for a binary source

|                      |   |   |
|----------------------|---|---|
| <b>Student Name</b>  | : | Mariana Ramos   |
| <b>Starting Date</b> | : | July 24, 2018   |
| <b>Goal</b>          | : | Test a mutual information estimator in a binary source. |
| <b>Directory</b>     | : | sdf/eit_87071_mutual_information_estimator.             |

The main goal of this Mutual Information Estimator is to test the mutual information application in a binary source when it is connected to the receiver using a binary symmetric channel, which has a variable probability of error. The binary source should have equiprobable outputs, being  $P(X = 0) = P(X = 1) = \frac{1}{2}$ . However, this probability value can be set by the user. Moreover, the channel error probability can be set by the user and test the mutual information estimator for different error probabilities. The symbols are outputted by the binary source block with a certain probability. Then a add block follows the binary source, which will add errors in the sequence transmitted with a certain probability defined by the user. Mutual information estimator block receives the signal with errors as well as the sequence sent by the binary source. It will compare the two sequences, estimates channel error probability and  $P(X = 0)$  and finally calculates the entropy of the output symbols and the conditional entropy of the output symbols after observe the input symbols to calculate the mutual information between the two sequences. In sub section 6.7.2 are presented numerical results and the theoretical results for further comparison.

### 6.7.1 Theoretical Analysis

Mutual information is defined as a difference between the number of bits of information that the observer needs to determine the input channel symbol before observing the output bit symbol and the number of bits of information that he/she needs to determine the output bit symbol after observation. For a binary symmetric memoryless channel we can determine the mutual information using the formula:

$$I(X; Y) = H(Y) - H(Y|X), \quad (6.76)$$

where  $H(Y)$  is the entropy of the channel output symbols and  $H(Y|X)$  is the conditional entropy of the channel output symbols depending on the channel input symbols. Regarding with a binary memoryless symmetric channel equation 6.76 can be written as:

$$I(X; Y) = H(\bar{p}\alpha + p\bar{\alpha}) - H(p), \quad (6.77)$$

as it is explained in section 7.50. In equation 6.77  $p$  corresponds to the error probability of the binary symmetric memoryless channel and  $\alpha$  corresponds to the probability of the input channel symbol.

In this case, the channel input symbols are equiprobable being  $P(X = 0) = P(X = 1) = \alpha = \frac{1}{2}$ . The error probability of the channel can be set by the user in the current setup. This

way, if the error probability of the binary symmetric memoryless channel is lower than  $\frac{1}{2}$ ,  $\bar{p}\alpha + p\bar{\alpha} > p$  being the value calculated in equation 6.77 positive. When  $p = \frac{1}{2}$ , the arguments of both entropy functions (conditional entropy and entropy of the output channel symbols) are equal, which means that its difference is zero and the average of amount of information transferred by the channel is also zero as shown in figure 6.52.

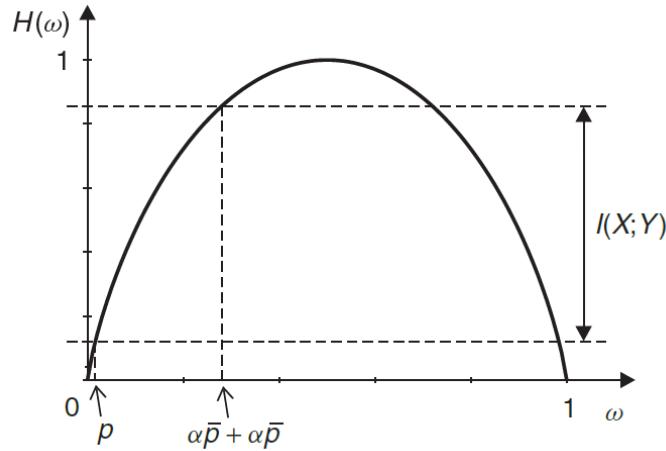


Figure 6.52: Graphical representation of average amount of information in the case of transmission using a binary symmetric memoryless channel. Figure from [1].

### 6.7.2 Simulation Analysis

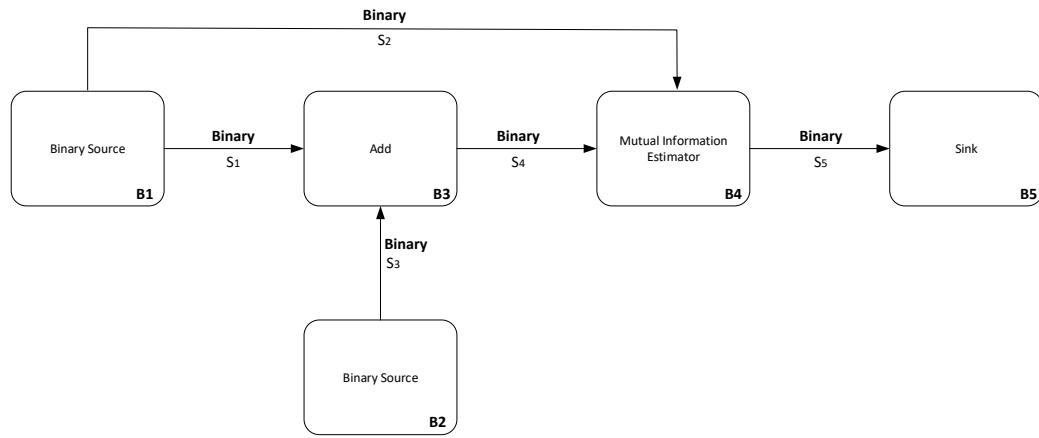


Figure 6.53: Block diagram of the simulation implemented.

In figure 6.53 is shown the block diagram of the system implemented to test the block of mutual information estimator. In this case, the transmitter is the binary source **B1** which outputs bit 0 and bit 1 with the same probability, 0.5, randomly. The binary symmetric

channel is represented by blocks **B2** and **B3**, being the first a binary source which outputs 0 with a probability  $1 - P_{\text{error}}$ , where  $P_{\text{error}}$  is defined by the user; and the second a adder block which introduces an error on the input bit sequence when the output of block **B2** is 1. Finally, **B4** is the block which estimates the mutual information between the input channel symbols and the output channel symbols and outputs a binary signal which takes value 1 when the input and output symbols are different and 0 otherwise. The functional description of this block is explained in detail in section 7.50.

In table 6.41 are presented the input parameters of the system.

Table 6.20: System Input Parameters

| Parameter    | Default Value |
|--------------|---------------|
| Perror       | 0.1           |
| m            | 0             |
| numberOfBits | 10            |

In table 6.42 are presented the system signals to implement the simulation presented in figure 6.312.

Table 6.21: System Signals

| Signal name | Signal type |
|-------------|-------------|
| S1          | Binary      |
| S2          | Binary      |
| S3          | Binary      |
| S4          | Binary      |
| S5          | Binary      |

Table 6.43 presents the header files used to implement the simulation as well as the specific parameters that should be set in each block. Finally, table 6.44 presents the source files.

Table 6.22: Header Files

| File name                               | Description | Status |
|---|-------------|--------|
| netxpto_20180418.h                      |             | ✓      |
| add_20180620.h                          |             | ✓      |
| binary_source_20180723.h                |             | ✓      |
| mutual_information_estimator_20180723.h |             | ✓      |
| sink.h                                  |             | ✓      |

Table 6.23: Source Files

| File name                                      | Description | Status |
|--|-------------|--------|
| netxpto_20180418.cpp                           |             | ✓      |
| add_20180620.cpp                               |             | ✓      |
| binary_source_20180723.cpp                     |             | ✓      |
| mutual_information_estimator_20180723.cpp      |             | ✓      |
| sink.cpp                                       |             | ✓      |
| eit_87071_mutual_information_estimator_sdf.cpp |             | ✓      |

The system described above was implemented and it were acquired  $1 \times 10^5$  symbols. Mutual information was calculated as well as the respective confidence intervals. The results are shown in table 6.24.

| $I(Y; X)$  | LB         | UB         | $H(Y X)$   | $H(Y)$     | $p$        | $\bar{p}\alpha + p\bar{\alpha}$ |
|------------|------------|------------|------------|------------|------------|---------------------------------|
| 0.92226600 | 0.92392500 | 0.92060600 | 0.07772800 | 0.99999400 | 0.00954000 | 0.50147100                      |
| 0.53179300 | 0.53488500 | 0.52870000 | 0.46820300 | 0.99999500 | 0.09975000 | 0.49871100                      |
| 0.27601000 | 0.27878100 | 0.27324000 | 0.72398300 | 0.99999400 | 0.20103000 | 0.50147700                      |
| 0.11982400 | 0.12183600 | 0.11781100 | 0.88017600 | 0.99999900 | 0.29909000 | 0.50052600                      |
| 0.02746750 | 0.02848050 | 0.02645450 | 0.97253100 | 0.99999800 | 0.40274000 | 0.50074900                      |
| 2.34E-08   | 9.71E-07   | -9.24E-07  | 1.00000000 | 1.00000000 | 0.49991000 | 0.50000000                      |
| 0.0285718  | 0.0296043  | 0.0275392  | 0.9714280  | 1.0000000  | 0.5991800  | 0.4999960                       |
| 0.1187460  | 0.1207510  | 0.1167410  | 0.8812540  | 1.0000000  | 0.7000300  | 0.5001840                       |
| 0.2729990  | 0.2757600  | 0.2702380  | 0.7269990  | 0.9999980  | 0.7974500  | 0.5008510                       |
| 0.5310040  | 0.5340970  | 0.5279110  | 0.4689960  | 1.0000000  | 0.9000000  | 0.4996960                       |
| 0.9182140  | 0.9199120  | 0.9165160  | 0.0817859  | 1.0000000  | 0.9898500  | 0.5001960                       |

Table 6.24: Results acquired for  $1 \times 10^5$  symbols acquisition.

After that the second line data from table 6.24 was chosen to represent graphically as shown in figure 6.54.

Results shown in figure 6.54 meet the theoretical representation shown in figure 6.52. Moreover, mutual information was calculated for a binary symmetric channel using different channel error probabilities ( $p$ ), theoretically, using equation 6.77. Both, theoretical and numerical values with the respective confidence intervals of mutual information were plotted in figure 6.55. As one can see in the figure, the numerical results meet the theoretical values of mutual information, which means that the mutual information estimator has the expected performance.

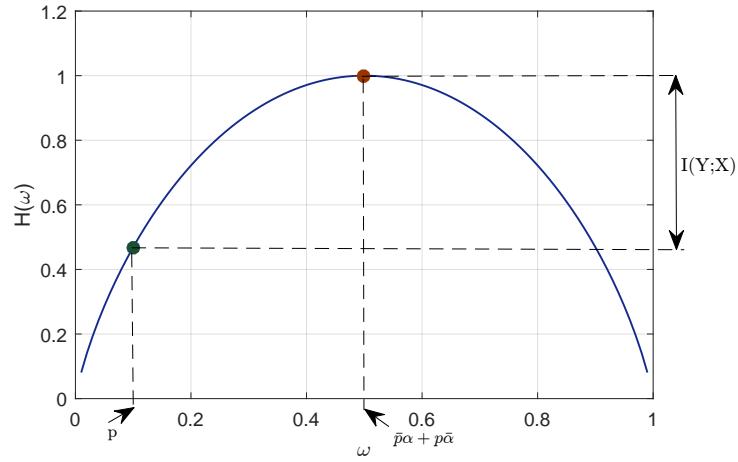


Figure 6.54: Numerical results representation for mutual information calculation.  $I(Y;X) = H(Y) - H(Y|X) = 0.5318$ ;  $H(Y|X) = 0.4680$ ;  $H(Y) = 1.0000$ ;  $p = 0.0998$ ;  $\bar{p}\alpha + p\bar{\alpha} = 0.4987$ .

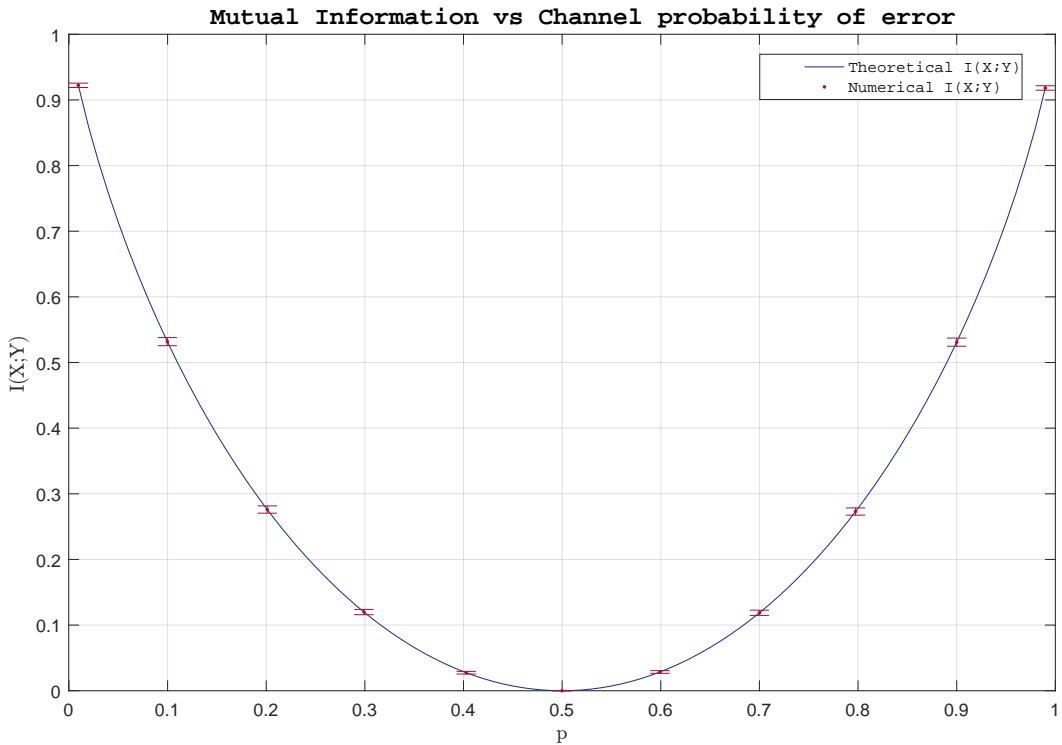


Figure 6.55: Numerical values of mutual information for different channel probability errors Vs theoretical mutual information.

## References

- [1] Krzysztof Wesolowski. *Introduction to digital communication systems*. John Wiley & Sons, 2009.

## 6.8 Dynamic Huffman Coder and Decoder

|                      |   |  |
|----------------------|---|--|
| <b>Students Name</b> | : | Cristiano Ferreira Gonçalves (10/06/2018 - 29/06/2018) |
| <b>Starting Date</b> | : | June 10, 2018  |
| <b>Goal</b>          | : | Huffman coding and decoding of text.                   |

Dynamic Huffman code can compress text very effectively for medium-long texts without previous knowledge of the characters probability.

### 6.8.1 Code Analysis

This code is stored and based on a Huffman tree, for example as the one of Figure 6.56. This tree stores in its leafs (bottom nodes) the characters. The code is attributed by the path from the root to each character, for each level of the tree is attributed an '1' if the path follows through the right soon or a '0' if it follows left.

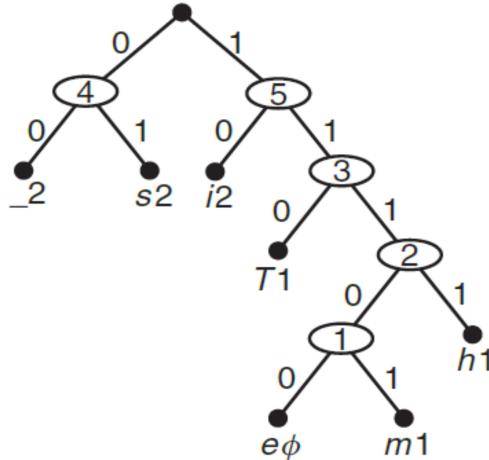


Figure 6.56: Ordered Huffman tree example (figure from [1]).

For each of the nodes 4 variables are stored:

- The character stored in the node;
- The frequency of the character (number of times it appeared);
- The pointer to the right soon node;
- The pointer to the left soon node.

When a new character is received two cases can happen:

- The character already exists and its frequency is incremented by one.
- The character does not exist on the tree and a new node is added for it.

For the last case, the node is added as right son of the previous empty node and the new empty node becomes the left son of the previous empty node.

After each modification the tree is reordered so its leafs from left to right and bottom-up are in ascending order of frequency. For the case described in Figure 6.56, it would be:

$e\emptyset \ m1 \ 1 \ h1 \ T1 \ 2 \ _2 \ s2 \ i2 \ 3 \ 4 \ 5$

The previous sequence is correctly ordered in terms of frequency. However for the example of Figure 6.57 the order would be:

$e\emptyset \ _2 \ 2 \ T1 \ h1 \ s2 \ i2 \ 3 \ 3 \ 5$

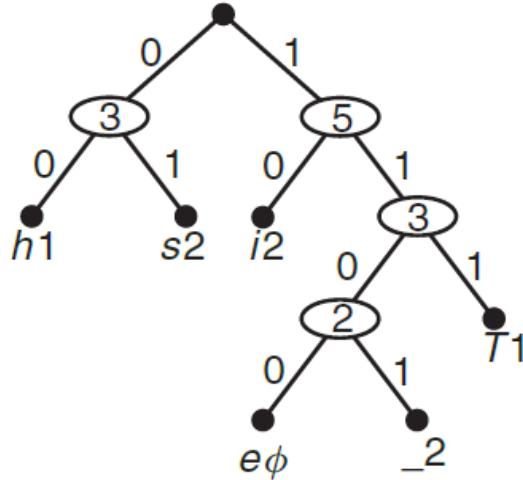


Figure 6.57: Not ordered Huffman tree example (figure from [1]).

For this case the nodes  $_2$  and  $h1$  should be swapped. This swap should be done between the first higher node ( $_2$  instead of 2) and the last lower one ( $h1$  instead of  $T1$ ) so the sub-tree becomes always ordered and the tree gets fully ordered faster.

### 6.8.2 Practical Test

The algorithm has been implemented and the respective program used to encode the following text:

"Nowadays, the power amplifier (PA) design paradigm is changing. Modern wireless networks push for PAs bandwidth, higher efficiencies and improved linearity requiring the use of more complex architectures [1][2]. Therefore, although the PA design methodology based on load-pull and S-parameter measurements obtained good results in the past [3], it is now very difficult to achieve the required figures of merit maintaining these design methodologies. This being the case, the use of CAD

programs with accurate nonlinear models becomes of paramount importance. Such models have been intensively described in literature [4][5], and are particularly interesting in the case of high frequency designs, mainly for the future mm-wave 5G networks, since at these frequencies the load-pull measurements are much harder to perform and the required equipment is very specific and expensive. The problem with this approach is that the obtained PA performance is strongly dependent on the accuracy of the used device models, which is strongly correlated with the quality of the measurements performed during the nonlinear model extraction [6]. Gallium Nitride (GaN) High Electron Mobility Transistors (HEMTs) are predominantly adopted on these new mm-wave state-of-art PAs due to their advantages in terms of power density, cut-off frequency and high thermal conductivity. However, these devices are still affected by many frequency dispersive phenomena as thermal and trapping effects [7]-[9]. These problems cause a substantial discrepancy between their static and dynamic characteristics. Thus, in order to properly model their behavior isodynamic measurements are necessary, which leads to more sophisticated measurement setups. There are two main approaches to characterize these new devices: (i) Continuous wave (CW) excitations to cover the entire I/V plane [10], which implies a complicated curve-fitting process to simultaneously extract the resistive and reactive components of the device. (ii) Pulsed I/V and S-parameter measurements [11][12], for which a system that allows the generation of very fast, high-power and arbitrary waveform pulses is necessary. In these last ones, the capability to generate arbitrary bias signals is of remarkable importance, because it allows to generate specific waveforms that are necessary to guarantee isothermal measurements and avoid dispersive phenomena [11]. There are already pulsed measurements systems able to generate 100 V and 7 A pulses with 600 ns settling time [13]-[15], and some modern pulsers that can handle 5000 W or even higher instantaneous power [16][17]. However, they are very expensive and unable to generate the necessary waveforms. Compared to the available ones, this paper presents a cost effective and more versatile measurement system, which allows the use of arbitrary waveform signals. This makes possible to obtain accurate and isodynamic pulsed measurements. Thus, the main contributions of this paper are the power head and respective characterization signal waveform design. To test the implemented system a 15 W GaN device from Wolfspeed was measured and shown to be mainly affected by drain lag and temperature effects. The second part of this paper is dedicated to the measurement setup and pulser implementation. The third section will be devoted to the waveform design. Lastly, in the fourth section, the measurements will be compared with I/V curves obtained from the integration of the small signal transconductance ( $g_m$ ) and output conductance ( $g_{ds}$ ). The good agreement obtained between all the sets of curves attests its isodynamic behavior."

This text has been encoded without any error in 60% of its original size, as shown in Figure 6.58. The total size decreased from 3.6KB to 2.2KB.

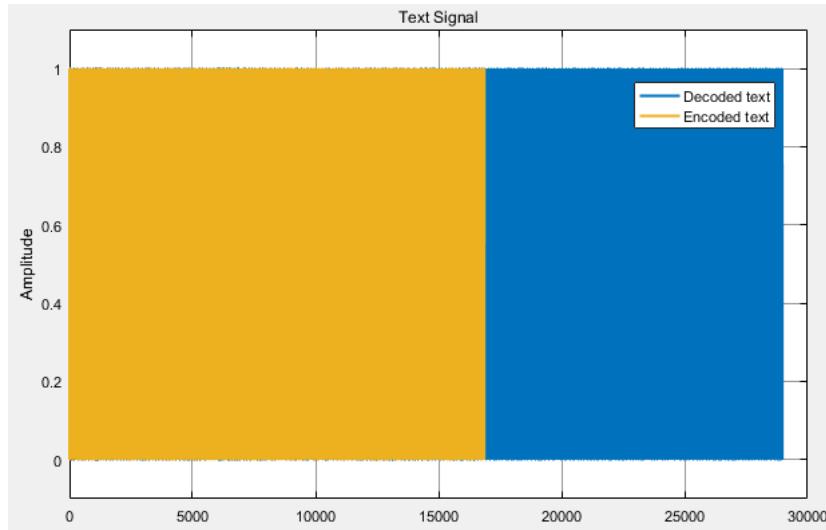


Figure 6.58: Matlab visualization of the encoded and decoded signals of the example text.

## References

- [1] Krzysztof Wesolowski. *INTRODUCTION TO DIGITAL COMMUNICATION SYSTEMS*. John Wiley and Sons, 2009.

## 6.9 M-QAM Transmission System

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

### 6.9.1 Introduction

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

For  $M = 4$  the M-QAM system can be reduced to a Quadrature Phase Shift Keying system (QPSK) system that uses four equispaced points in the constellation diagram, as shown in figure 6.59 where a Gray encoding is assumed [1].

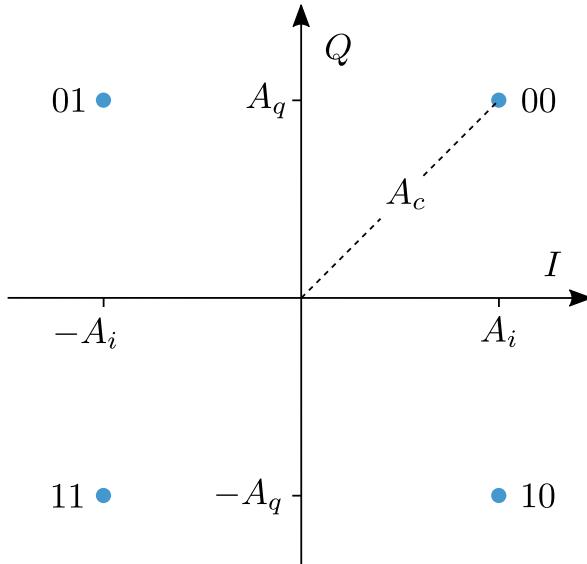


Figure 6.59: A QPSK constellation, assuming  $A = A_i = A_q = A_c/\sqrt{2}$ .

## 6.9.2 Theoretical Analysis

### 6.9.2.1 QPSK Transmitter

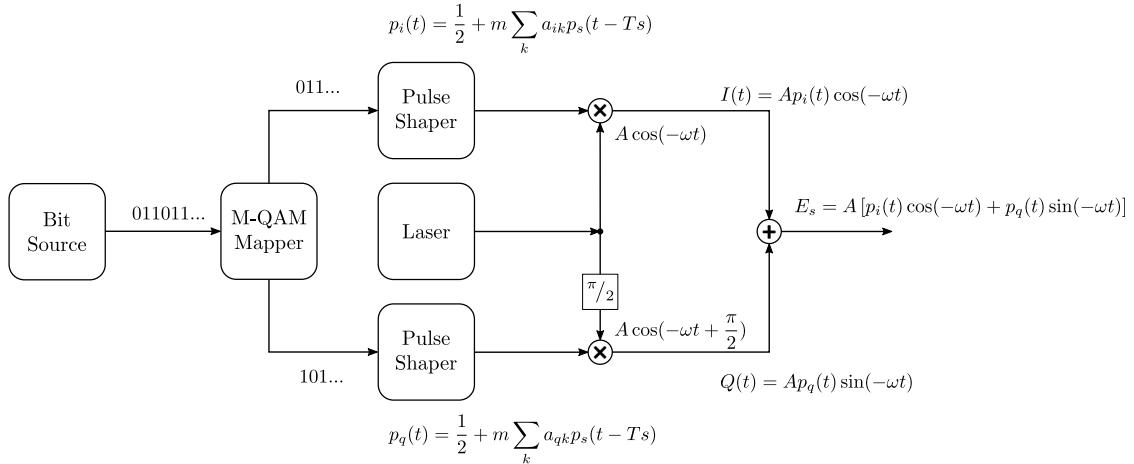


Figure 6.60: Transmitter diagram.

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

$$I(t) = p_i(t)A \cos(-\omega t) \quad (6.78)$$

$$Q(t) = p_q(t)A \cos\left(-\omega t + \frac{\pi}{2}\right) = p_q(t)A \sin(-\omega t) \quad (6.79)$$

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

$$p(t) = \frac{1}{2} + m \sum_k a_k p_s(t - T_s), \quad a_k = \begin{cases} -1 & \text{if bit } 1, \\ 1 & \text{if bit } 0 \end{cases} \quad (6.80)$$

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

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

Following from equations 6.78 and 6.79, we have [3, 4]:

$$\begin{aligned} E_s(t) &= I(t) + Q(t) = \\ &= p_i(t)A \cos(-\omega t) + p_q(t)A \sin(-\omega t) \\ &= p_i(t)A \cos(\omega t) - p_q(t)A \sin(\omega t) \\ &= A\sqrt{p_i^2(t) + p_q^2(t)} \cos(-\omega t + \phi_s(t)), \quad \phi_s(t) = \arctan(p_q(t), p_i(t)) \end{aligned} \quad (6.81)$$

The *arctan* function shown above is the two argument arctangent function, sometimes also called *atan2*. While the normal arctangent is defined only for the interval  $]-\frac{\pi}{2}, +\frac{\pi}{2}]$ , the two argument arctangent is a piecewise function defined in the interval  $]-\pi, +\pi]$  [5].

$$\arctan(x, y) = \begin{cases} \arctan(y/x), & \text{if } x > 0, \\ (\text{sign}\{y\})(\pi/2) & \text{if } x = 0 \text{ and } y \neq 0, \\ \pi + \arctan(y/x) & \text{if } x < 0, \\ 0, & \text{if } x = 0 \text{ and } y = 0 \end{cases} \quad (6.82)$$

### 6.9.2.2 QPSK Receiver

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

A typical configuration for this receiver is shown in Figure 6.61. The local oscillator laser uses the same frequency of the incoming signal. It is used in coherent detection to extract the phase and amplitude information contained in the received optical signal. This is done by mixing the LO with the incoming optical signal in a  $\pi/2$  optical hybrid. The optical hybrid then outputs 4 different optical signals, where the phase difference between the LO and the signal is at  $0, \pi/2, \pi, 3\pi/2$ . This difference is relative to their phase difference at the input of the optical hybrid. The signals with a difference of  $\pi$  between themselves are paired and sent to a pair of balanced photodiodes, where each pair measures one of the quadratures. The transimpedance amplifier then amplifies the signal, while converting the current to a voltage to be sampled and quantified in an ADC.

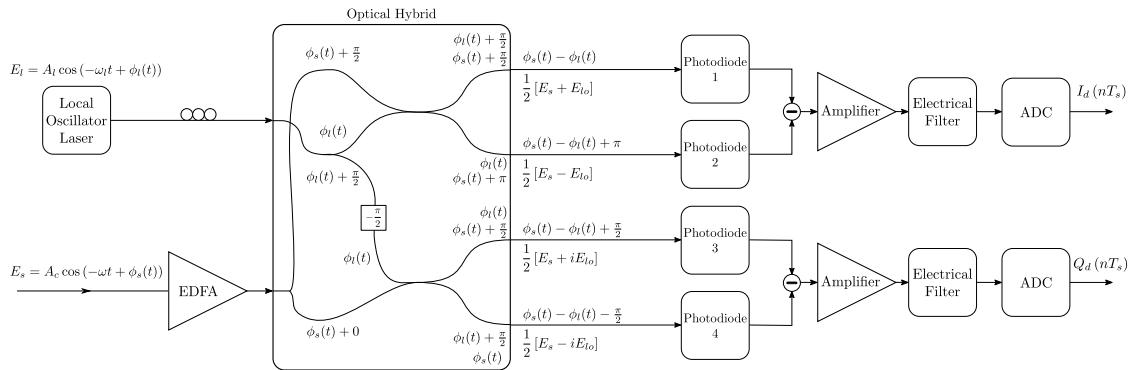


Figure 6.61: Local oscillator and receiver filters diagram.  $\phi_a$  and  $\phi_b$  represent the phase rotations happening inside the optical hybrid.

Before starting, we should clarify how the optical hybrid works. A possible configuration is shown in Figure 6.61, where the mixing is achieved resorting to a few directional couplers. Figure 6.62 shows an example of a directional coupler, where two fiber cores are close enough so that the space between them is comparable to their diameter. In these conditions,

the fundamental modes propagating through each fiber overlap partially, leading to power transfer from one core to the other [6].

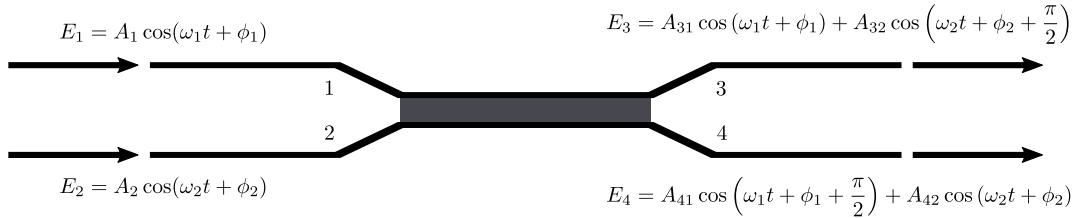


Figure 6.62: Directional coupler diagram.

Assuming a lossless directional coupler as shown Figure 6.62, the relation between input and output power is:

$$P_1 + P_2 = P_3 + P_4 \quad (6.83)$$

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

$$\begin{bmatrix} A_3 \\ A_4 \end{bmatrix} = \begin{bmatrix} \cos(kL) & i \sin(kL) \\ i \sin(kL) & \cos(kL) \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \quad (6.84)$$

Two things can be noticed here: first, the power ratios are dependent on the coupling length. This comes from the supermodes of the fiber coupler, which have different propagation constants [6]. This difference creates a relative phase shift proportional to the coupler length. In addition, a phase shift of  $\pi/2$  exists always between the two outputs. This is because the supermodes are the even and odd combinations of the fundamental modes of the coupler.

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

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

Coherent detection takes the product of the electric fields from the signal and a Local Oscillator, in order to measure the phase difference between them through interference. The electric field of the Local Oscillator is [7]:

$$E_{lo} = A_{lo} \cos(-\omega_{lo} t + \phi_{lo}) \quad (6.85)$$

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

$$P_s = k E_s^2 = k A_c^2 \cos^2(-\omega t + \phi_s) = k \frac{A_c^2}{2} = k \frac{(A\sqrt{2})^2}{2} = k A^2$$

$$P_{lo} = k E_{lo}^2 = k A_{lo}^2 \cos^2(-\omega_{lo} t + \phi_{lo}) = k \frac{A_{lo}^2}{2}$$

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

$$P_{\text{s\_out}} = G_o P_{\text{s\_in}} \quad (6.86)$$

$G_o$  is the EDFA optical gain, given by [8]:

$$G_o = \frac{G_{\text{small}}}{1 + \frac{P_{\text{in}}}{P_{\text{sat}}}} \quad (6.87)$$

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

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

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

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

$$I_1(t) = \frac{\eta}{4} \left[ G_o P_s(t) + P_{lo} + 2\sqrt{G_o P_s(t) P_{lo}} \cos(\phi_s(t)) \right] \quad (6.88)$$

$$I_2(t) = \frac{\eta}{4} \left[ G_o P_s(t) + P_{lo} - 2\sqrt{G_o P_s(t) P_{lo}} \cos(\phi_s(t)) \right] \quad (6.89)$$

$$I_3(t) = \frac{\eta}{4} \left[ G_o P_s(t) + P_{lo} + 2\sqrt{G_o P_s(t) P_{lo}} \sin(\phi_s(t)) \right] \quad (6.90)$$

$$I_4(t) = \frac{\eta}{4} \left[ G_o P_s(t) + P_{lo} - 2\sqrt{G_o P_s(t) P_{lo}} \sin(\phi_s(t)) \right] \quad (6.91)$$

$$(6.92)$$

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

$$\begin{aligned} V_i(t) &= G_e (I_1 - I_2) \\ &= G_e \eta \sqrt{G_o P_s P_{lo}} \cos(\phi_s(t)) \end{aligned} \quad (6.93)$$

$$\begin{aligned} V_q(t) &= G_e (I_3 - I_4) \\ &= G_e \eta \sqrt{G_o P_s P_{lo}} \sin(\phi_s(t)) \end{aligned} \quad (6.94)$$

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

Assuming perfect synchronization, if the signal is sampled with timing  $t = nT_s$ , the constellation can be reconstituted perfectly, if there is no intersymbol interference. In this case, according to equations 6.78 and 6.79,  $p_i(t)$  and  $p_q(t)$  are equal to either 1 or  $-1$ . It follows that  $\phi_s(t = nT_s) = \frac{\pi}{4} \pm m\frac{\pi}{2}$ . If the gain  $G_e$  is adjusted so that  $G_e \eta \sqrt{G_o P_s P_{lo}} = A_c$ , we get:

$$\begin{aligned} V_i(t) &= A_c \cos(\phi_s(t)), \\ &= \pm \frac{A_c}{\sqrt{2}} = \pm A_i = \pm A \quad t = nT_s \end{aligned} \quad (6.95)$$

$$\begin{aligned} V_q(t) &= A_c \sin(\phi_s(t)), \\ &= \pm \frac{A_c}{\sqrt{2}} = \pm A_q = \pm A \quad t = nT_s \end{aligned} \quad (6.96)$$

Looking at Figure 6.63, we can see that equations 6.95 and 6.96 describe the four points in the constellation.

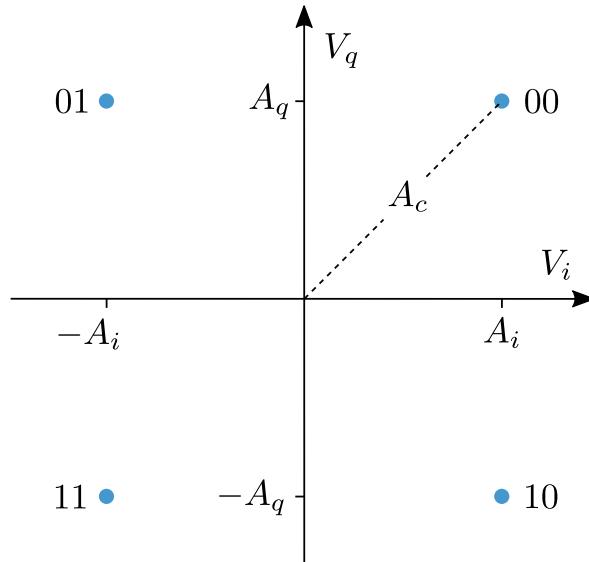


Figure 6.63: A QPSK constellation, assuming  $A = A_i = A_q = A_c/\sqrt{2}$ .

The previous explanation assumes that the polarization of both the transmitted signal and the local oscillator is aligned. This can be achieved by using a polarization controller, as shown in Figure 6.61. We have also neglected any link losses.

### 6.9.2.3 ISI

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

Each symbol is represented by its own pulse, which is dependent on the symbol period  $T_s$ . In each pulse, there is an optimal instant  $t_s$ , where we can sample the signal in order to perfectly identify the contained symbol. These instants are separated by a time equal to the symbol period. In order to negate intersymbol interference, the signal for each pulse should be equal to zero every time a different symbol is sampled, that is, at all instants  $t_s \pm nT_s$ . If this condition is verified, the value of the signal at  $t = t_s$  is dependent only on the symbol transmitted at that time, not being influenced by the surrounding pulses. This is shown in

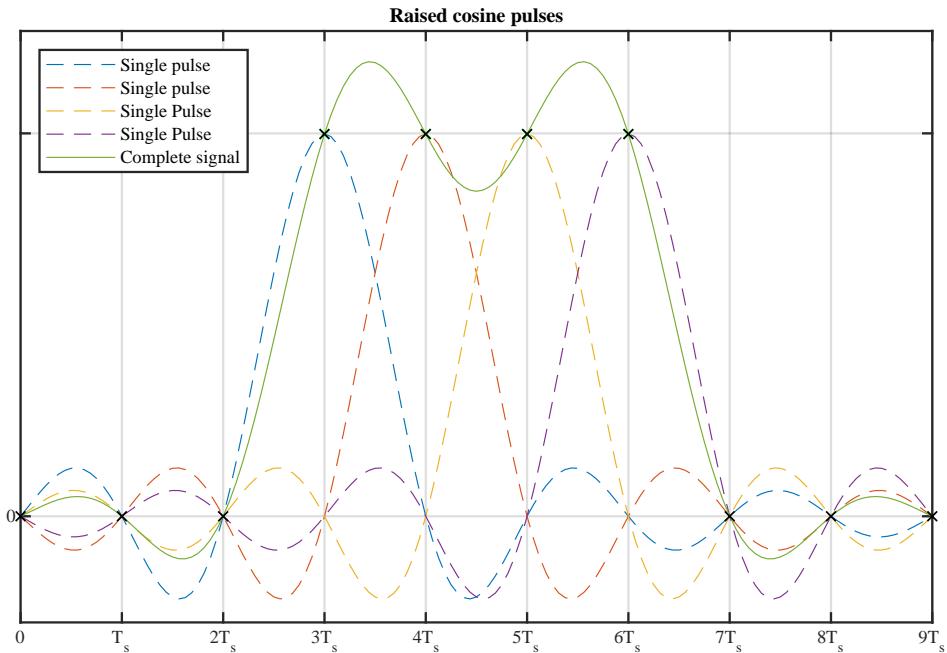


Figure 6.64: Signal generated by four sequential raised cosine pulses. It can be seen that at  $t = nT_s$ , every pulse except one equals exactly zeros, showing that there is no interference. In addition, at those times, the value of the signal is coincident with the peak of the individual pulses.

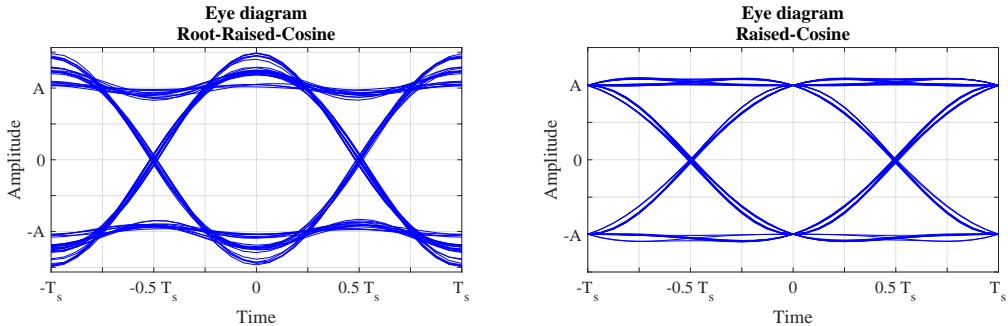


Figure 6.65: Eye diagrams of two signals without noise: on the left, shaped with a single root-raised-cosine filter and affected by ISI; on the right, shaped using a raised-cosine filter, showing no signs of ISI.

Figure 6.65, where we can see that at all instants  $nT_s$ , the response is zero for all pulses but one. Consequently, at those instants, the signal obtained from the combination of all pulses is exactly equal to the peak of the pulse transmitted at that time. Those instants are also shown at the center of the eye diagram shown in Figure 6.65, where  $t_s$  is represented, and perfect coincidence in the signal at sampling time is shown.

The raised-cosine is a shape commonly used for pulse shaping, as it does not create any intersymbol interference. This is shown in Figure 6.65. The time domain response for raised-cosine filter is given by [9]

$$h_{RC}(t) = \frac{\sin(\pi t/T_s)}{(\pi t/T_s)} \frac{\cos(\pi\beta t/T_s)}{1 - (4\beta^2 t^2/T_s^2)} \quad (6.97)$$

where  $\beta$  is the roll-off factor and  $T_s$  is the symbol period.

Optimal detection often requires the implementation of matched filters [10]. This is because in addition to avoiding ISI, it is desirable to maximize the SNR before sampling the signal, and in order to achieve both these objectives simultaneously, a matched filter should be used. This is done by implementing in the receiver a filter similar to the one used to initially shape the pulses. The pulse-shaping process becomes divided between the transmission and reception stages. Provided that an appropriate filter is chosen, the resulting signal can still be free of ISI, while reducing the noise power in the final sampled signal.

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

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

The root-raised-cosine is defined as a filter for which the square of the frequency response is equal to the frequency response of the raised cosine filter. Its time domain impulse

response given by[9]:

$$h_{RRC}(t) = \frac{(4\beta t/T_s) \cos [\pi(1+\beta)t/T_s] + \sin [\pi(1-\beta)t/T_s]}{(\pi t/T_s)[1 - (4\beta t/T_s)^2]} \quad (6.98)$$

#### 6.9.2.4 Noise

So far we have neglected to include any source of noise in this analysis. Several possible noise sources can be considered, either in the optical or in the electrical part of the system. As shown in Figure 6.61, for now we will only consider the existence of optical noise, and we will consider that the noise added is ASE which could originate from a EDFA. The ASE field can be represented as a sum of terms as follows [11, 12]:

$$E_{ASEx} = \sum_{k=-B_\nu}^{B_\nu} \sqrt{2n_{0o}\delta\nu} \cos ((-\omega + 2\pi k\delta\nu)t + \phi_{kx}) \quad (6.99)$$

$$E_{ASEy} = \sum_{k=-B_\nu}^{B_\nu} \sqrt{2n_{0o}\delta\nu} \sin ((-\omega + 2\pi k\delta\nu)t + \phi_{ky}) \quad (6.100)$$

where  $B_\nu = B_{\text{opt}}/(2\delta\nu)$ ,  $B_{\text{opt}}$  is the optical bandwidth and  $\delta\nu$  is small frequency interval. We can write an expression for the optical noise spectral density  $n_{0o}$  considering the use of a number  $N_A$  of cascaded EDFA's. Assuming that all amplifiers are similar and equally spaced, and that the gain in each EDFA is just enough to compensate the fiber losses in the distance between them, we have [13]:

$$n_{0o} = N_A n_{\text{sp}}(G_o - 1)h\nu \quad (6.101)$$

Here,  $n_{\text{sp}}$  is the EDFA's spontaneous emission factor,  $h\nu$  is the photon energy. In addition,  $\phi_{kx}$  and  $\phi_{ky}$  are independent uniform random variables representing random phases for each of the terms in the summation, and  $G_o$  is the amplifiers' optical gain as defined in equation 6.87. This field is added to equation 6.81, giving rise to three noise terms from different interactions:

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

For now, we will consider only the ASE-LO component, which is close enough to additive white Gaussian noise. In addition, it is typically the strongest, as usually the LO power is much greater than either the signal or the ASE. The variance, or noise power, generated by this component at each pair of balanced photodiodes is given by [11]:

$$\sigma_{\text{ASE-LO}}^2 = 2\eta^2 P_{\text{LO}} n_{0o} B_N = 2\eta^2 P_{\text{LO}} N_A n_{\text{sp}}(G_o - 1)h\nu B_N \quad (6.102)$$

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

$$\sigma_{\text{ASE-LO}}^2 = 2\eta^2 G_e^2 P_{\text{LO}} N_A n_{\text{sp}} (G_o - 1) h\nu B_N \quad (6.103)$$

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

$$x(t) = s(t) + n(t) \quad (6.104)$$

where the component  $s(t)$  corresponds to the received signal, and  $n(t)$  corresponds to the noise component. For  $t = nT_s$ , as mentioned in equations 6.95 and 6.96,  $s(t) = \pm A$ . The noise component, being AWGN, has constant power spectral density, and follows a Gaussian distribution with zero mean and variance equal to the noise power as described in equation 6.103.

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

Let us consider a signal as described in Equation 6.104, with  $a(t) = A_p p(t)$ , where  $A_p$  is the peak amplitude of the signal and  $p(t)$  is the shape of the pulse. In addition, let  $n(t)$  be AWGN of spectral density  $G_n(f) = n_0/2$ . Finally, assume that the filter at the receiver has a frequency response  $H(f)$ . The filter is similar to the shape of the pulse, but reversed in time and shifted by a time delay, such that it maximizes the SNR [2]. The energy contained in the pulse that enters the receiver filter depends on the pulse amplitude and on its shape, and it is given by:

$$E_p = \int_{-\infty}^{\infty} |A(f)|^2 df = A_p^2 \int_{-\infty}^{\infty} |P(f)|^2 df \quad (6.105)$$

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

$$A = \mathfrak{F}^{-1}[H(f)A(f)]|_{t=t_0+t_d} = A_p \int_{-\infty}^{+\infty} H(f)P(f)e^{+j\omega t_d} df \quad (6.106)$$

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

$$N_o = \int_{-\infty}^{+\infty} |H(f)|^2 G_n(f) df = \frac{n_0}{2} \int_{-\infty}^{+\infty} |H(f)|^2 df \quad (6.107)$$

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

$$\begin{aligned}\frac{A^2}{N_o} &= A_p^2 \frac{\left| \int_{-\infty}^{+\infty} |H(f)P(f)e^{j\omega t_d} df \right|^2}{\int_{-\infty}^{+\infty} |H(f)|^2 G_n(f) df} \\ &= A_p^2 \frac{\left| \int_{-\infty}^{+\infty} |H(f)P(f)e^{j\omega t_d} df \right|^2}{\frac{n_0}{2} \int_{-\infty}^{+\infty} |H(f)|^2 df}\end{aligned}\quad (6.108)$$

It can be shown that a matched filter maximizes the ratio above, so that it becomes [2] :

$$\frac{A^2}{N_o} = \frac{A_p^2}{n_0/2} \int_{-\infty}^{+\infty} |P(f)|^2 df = \frac{A_p^2}{n_0/2} \int_{-\infty}^{+\infty} p(t)^2 dt \quad (6.109)$$

Thus, substituting from equation 6.105, the following relation can be verified for the signal after the matched filter:

$$\frac{A^2}{N_o} = \frac{2E_b}{n_0} \quad (6.110)$$

Here,  $A$  is the amplitude of  $s(t)$  at the output of the matched filter,  $N_o$  is the corresponding noise power,  $E_b$  is the energy per bit and  $n_0/2$  is the two-sided noise spectral density, obtained from equation 6.103 from  $n_0 = \sigma_{\text{ASE-LO}}^2 / B_N$ .

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

#### 6.9.2.5 Bit error rate

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

As previously mentioned, the values sampled at sampling time  $t = nT_s$ , which are used to identify the received symbols, follow a Gaussian distribution. Using the constellation from Figure 6.59, with a decision boundary halfway between  $A$  and  $-A$ , an error occurs in two situations:

$$\begin{cases} x(t) < 0, & \text{if } s(t) = A \\ x(t) > 0, & \text{if } s(t) = -A \end{cases} \implies \begin{cases} n(t) < -A, & \text{if } s(t) = A \\ n(t) > A, & \text{if } s(t) = -A \end{cases} \quad (6.111)$$

This is illustrated in Figure 6.66, where the probability of error is shown by the colored area under the curves [10].

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

$$\begin{aligned}P_{be} &= P_0 P_{e0} + P_1 P_{e1} = \frac{1}{2} P_{e0} + \frac{1}{2} P_{e1} \\ &= \frac{1}{2} \left[ \Pr(n(t) < -A) + \Pr(n(t) > A) \right]\end{aligned}\quad (6.112)$$

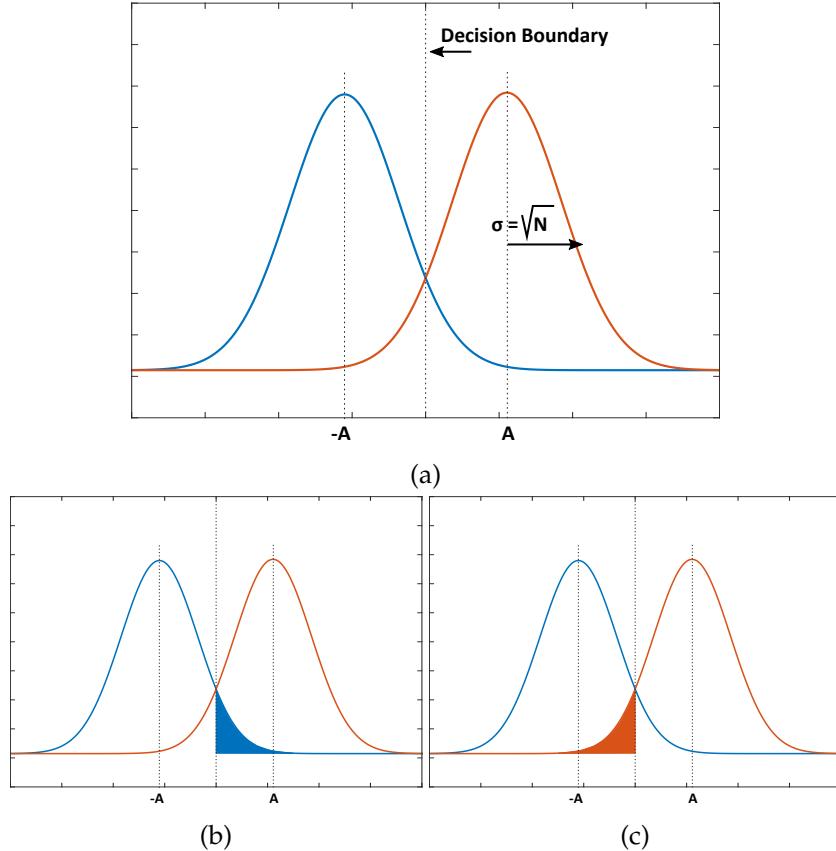


Figure 6.66: Probability density functions for  $x(t) = s(t) + n(t)$ , with  $s(t) = \pm A$  at the time of sampling, and  $n(t)$  as a Gaussian variable of zero mean and variance  $N$ . The colored areas below the curves on the right represent the probability of error for each transmitted bit.

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

$$P_{be} = Q\left(\frac{A}{\sqrt{N_o}}\right) = \frac{1}{2}\operatorname{erfc}\left(\frac{A}{\sqrt{2N_o}}\right) \quad (6.113)$$

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

We have worked on the probability of bit error for each quadrature independently. However, assuming a gray code, and knowing that the carriers are uncorrelated, an error in each carrier is independent from the other [9]. This can be verified by noting that the even bits are dependent only on one of the quadratures, and the odd bits depend on the other. Therefore, as half the bits are even and half are odd, and assuming that the constellation is symmetrical, it follows that the error rate is the same as for each quadrature independently.

$$\begin{aligned}
\text{BER} &= P_{\text{odd}}P_{\text{oddError}} + P_{\text{even}}P_{\text{evenError}} \\
&= \frac{1}{2}P_{\text{oddError}} + \frac{1}{2}P_{\text{evenError}} \\
&= \frac{1}{2} \left( Q \left( \frac{A_{\text{even}}}{\sqrt{N_{\text{even}}}} \right) + Q \left( \frac{A_{\text{odd}}}{\sqrt{N_{\text{odd}}}} \right) \right) \\
&= Q \left( \frac{A}{\sqrt{N_o}} \right)
\end{aligned} \tag{6.114}$$

Looking back at equation 6.110, we can also define the probability of error as a function of the energy per bit:

$$P_{be} = Q \left( \sqrt{\frac{2E_b}{n_0}} \right) = \frac{1}{2} \operatorname{erfc} \left( \sqrt{\frac{E_b}{n_0}} \right) \tag{6.115}$$

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

$$P_C = (1 - P_{be})^2 \tag{6.116}$$

From this, the probability of symbol error is:

$$\begin{aligned}
P_{Se} &= 1 - P_C \\
&= 1 - \left( 1 - Q \left( \frac{A}{\sqrt{N_o}} \right) \right)^2 \\
&= 2Q \left( \frac{A}{\sqrt{N_o}} \right) \left[ 1 - \frac{1}{2}Q \left( \frac{A}{\sqrt{N_o}} \right) \right] \\
&= 2Q \left( \sqrt{\frac{2E_b}{n_0}} \right) \left[ 1 - \frac{1}{2}Q \left( \sqrt{\frac{2E_b}{n_0}} \right) \right]
\end{aligned} \tag{6.117}$$

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

We can now write the expected bit error rate as a function of the system parameters. To get  $E_b$ , we can start by picking up the equations for the signal waveforms which are sampled and measured at the ADC, described in Equations 6.93 and 6.94.

$$E_b(t) = G_e^2 \eta^2 G_o P_s P_{lo} \frac{T_s}{2} \tag{6.118}$$

To get  $n_0$ , we start with equation 6.103, which gives the noise variance at the ADC's. The noise power spectral density will be the ration of this variance to the electrical bandwidth of the receiver. So we get:

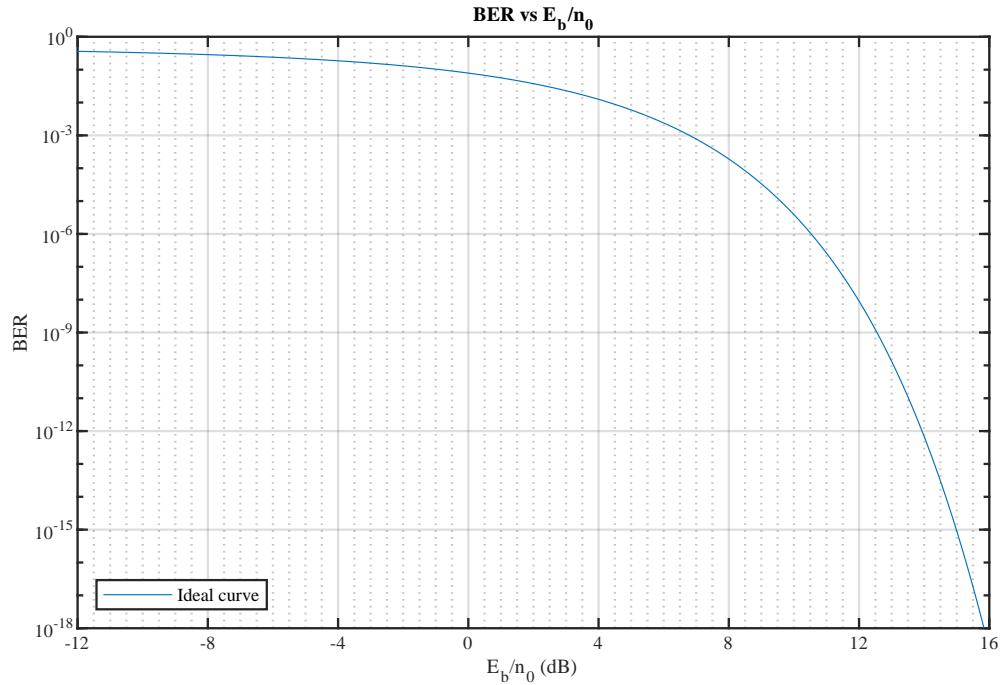


Figure 6.67: QPSK theoretical BER curve as a function of the ratio of energy per bit and the noise spectral density in dB.

$$\begin{aligned}
 n_0 &= \frac{2\eta^2 G_e^2 P_{\text{LO}} N_A n_{\text{sp}} (G_o - 1) h\nu B_N}{B_N} \\
 &= 2\eta^2 G_e^2 P_{\text{LO}} N_A n_{\text{sp}} (G_o - 1) h\nu \\
 &= 2\eta^2 G_e^2 P_{\text{LO}} n_{0o}
 \end{aligned} \tag{6.119}$$

### 6.9.3 Simulation Analysis

The M-QAM transmission system is a set of blocks that simulate the modulation, transmission and demodulation of an optical signal using M-QAM modulation. It is composed of several complex blocks: a transmitter, a receiver, a noise source, an addition block, a sink and a block that performs a Bit Error Rate (BER) measurement. The schematic representation of the system is presented in Figures 6.68 to 6.71. The simulation currently implements a QPSK system ( $M=4$ ).

#### 6.9.3.1 Functional description

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

The homodyne receiver requires two optical inputs: one of the signal itself, and another from a local oscillator perform the signal demodulation. It receives, processes and decodes the received signal and afterwards outputs the reconstructed bitstream. This signal is compared to the binary signal generated by the transmitter in order to estimate the Bit Error Rate (BER). The files used are summarized in tables 6.25 and 6.26. All the blocks and sub-blocks used are included in the tables.

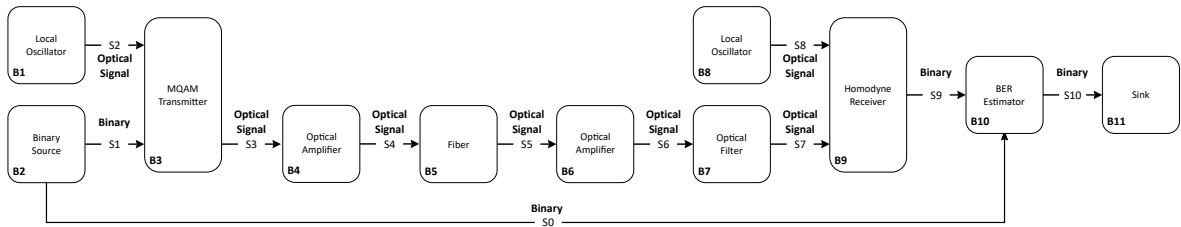


Figure 6.68: Top-Layer Schematic representation of the simulated MQAM system.

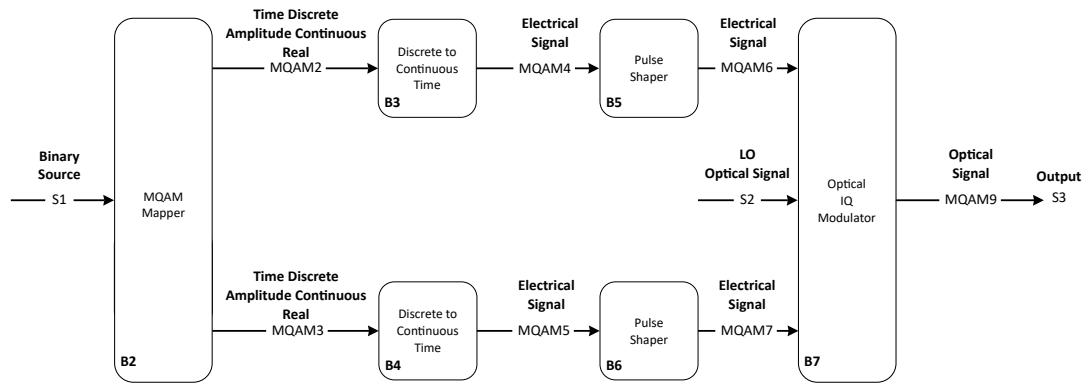


Figure 6.69: Schematic representation of the MQAM Transmitter block.

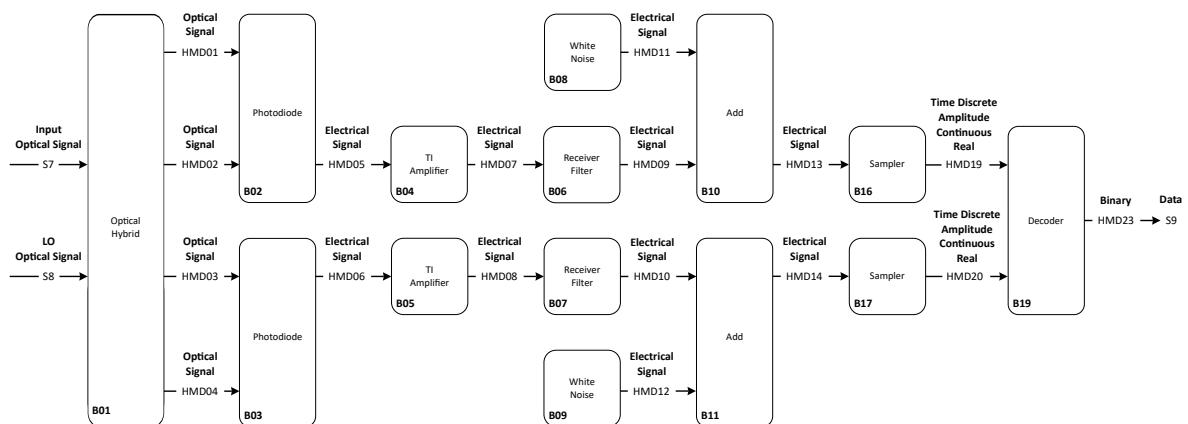


Figure 6.70: Simplified schematic representation of the Homodyne Receiver block.

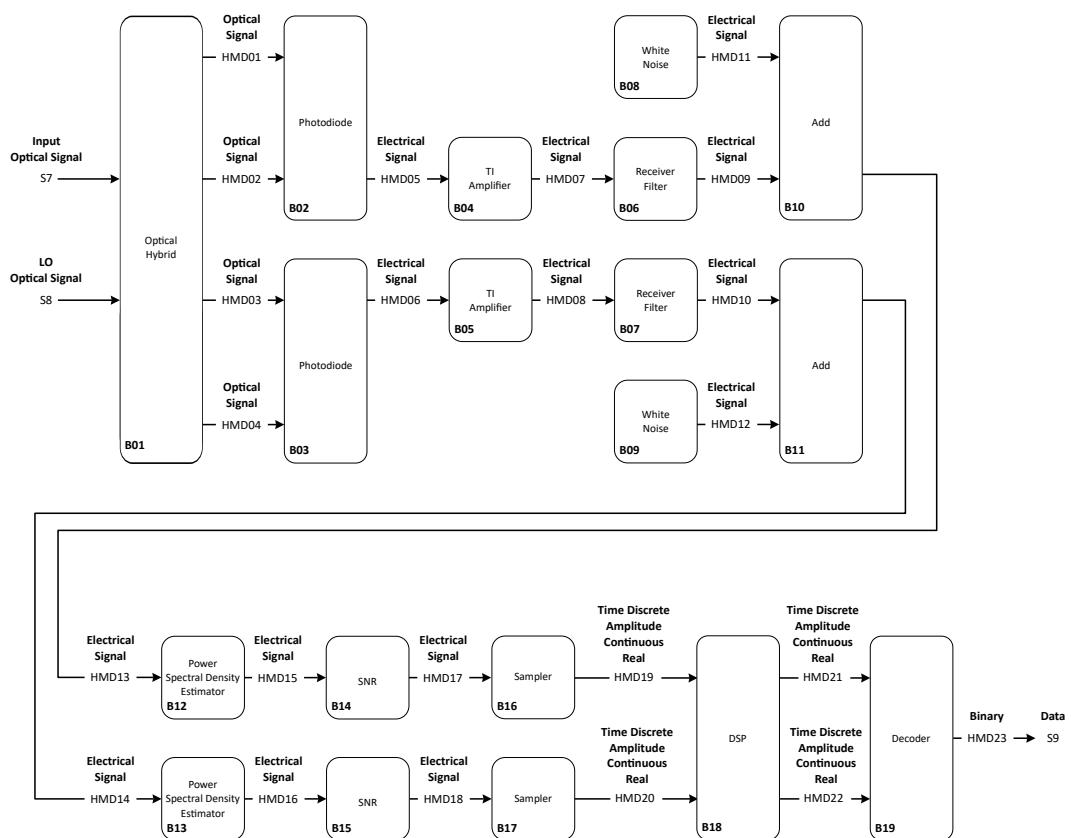


Figure 6.71: Schematic representation of the Homodyne Receiver block, with the DSP block (currently not implemented) and two optional blocks for measuring the SNR and power spectral density.

### 6.9.3.2 Required files

| Source Files                                  |              |        |
|---|--------------|--------|
| File  | Comments     | Status |
| add_20190215.cpp                              |              | ✓      |
| binary_source_20190215.cpp                    |              | ✓      |
| bit_error_rate_20190215.cpp                   |              | ✓      |
| decoder_20190215.cpp                          |              | ✓      |
| discrete_to_continuous_time_20190215.cpp      |              | ✓      |
| electrical_filter_20190215.h                  |              | ✓      |
| edfa_20190215.cpp                             |              | ✓      |
| fft_20180208.cpp                              |              | ✓      |
| fiber_20190215.cpp                            |              | ✓      |
| homodyne_receiver_withoutLO_20190215.cpp      | <sup>1</sup> | ✓      |
| ideal_amplifier_20190215.cpp                  |              | ✓      |
| iq_modulator_20190215.cpp                     |              | ✓      |
| local_oscillator_20190215.cpp                 |              | ✓      |
| m_qam_mapper_20190215.cpp                     |              | ✓      |
| m_qam_system_sdf.cpp                          | <sup>2</sup> | ✓      |
| m_qam_transmitter_20190215.cpp                |              | ✓      |
| netxpto_20190215.cpp                          | <sup>2</sup> | ✓      |
| optical_hybrid_20190215.cpp                   |              | ✓      |
| photodiode_old_20190215.cpp                   |              | ✓      |
| power_spectral_density_estimator_20190215.cpp |              | ✓      |
| pulse_shaper_20190215.cpp                     |              | ✓      |
| sampler_20190215.cpp                          |              | ✓      |
| sink_20190215.cpp                             |              | ✓      |
| snr_estimator_20190215.cpp                    |              | ✓      |
| super_block_interface_20190215.cpp            | <sup>2</sup> | ✓      |
| white_noise_20190215.cpp                      |              | ✓      |
| window_20180704.cpp                           |              | ✓      |

Table 6.25: <sup>1</sup> The library entry is under a different name, *m\_qam\_receiver*;

<sup>2</sup> No library entry as it is a main or general purpose file, not a specific block.

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

|   |              |   |
|---|--------------|---|
| discrete_to_continuous_time_20190215.h      |              | ✓ |
| electrical_filter_20190215.h                |              | ✓ |
| edfa_20190215.h                             |              | ✓ |
| fft_20180208.h                              |              | ✓ |
| fiber_20190215.h                            |              | ✓ |
| homodyne_receiver_withoutLO_20190215.h      | <sup>1</sup> | ✓ |
| ideal_amplifier_20190215.h                  |              | ✓ |
| iq_modulator_20190215.h                     |              | ✓ |
| local_oscillator_20190215.h                 |              | ✓ |
| m_qam_mapper_20190215.h                     |              | ✓ |
| m_qam_transmitter_20190215.h                |              | ✓ |
| netxpto_20190215.h                          | <sup>2</sup> | ✓ |
| optical_hybrid_20190215.h                   |              | ✓ |
| photodiode_old_20190215.h                   |              | ✓ |
| power_spectral_density_estimator_20190215.h |              | ✓ |
| pulse_shaper_20190215.h                     |              | ✓ |
| sampler_20190215.h                          |              | ✓ |
| sink_20190215.h                             |              | ✓ |
| snr_estimator_20190215.h                    |              | ✓ |
| super_block_interface_20190215.h            | <sup>2</sup> | ✓ |
| white_noise_20190215.h                      |              | ✓ |
| window_20180704.h                           |              | ✓ |

Table 6.26: <sup>1</sup> The library entry is under a different name, *m\_qam\_receiver*

<sup>2</sup> No library entry as it is a main or general purpose file, not a specific block.

### 6.9.3.3 Input Parameters

Table 6.27: Input parameters

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

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

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

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

#### 6.9.3.4 Output Files

Table 6.28: Output Files

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

### 6.9.3.5 Simulation results - ISI

In this section we will explore the signals generated during the simulation. The general scheme of the simulation is shown in Figures 6.68, 6.69 and 6.71. Every signal generated during the simulation is identified in those diagrams.

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

The parameters used to obtain the plots and eye diagrams in this section are displayed in Table 6.29.

Table 6.29: Simulation parameters

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

|            |      |  |
|------------|------|--|
| confidence | 0.95 |  |
|------------|------|--|

We will start by showing the signals present in the *m\_qam\_transmitter* block, shown in Figure 6.69. This is the block where the signals are generated and modulated, and remains unaltered for all cases shown here. Therefore, to avoid repetition, these signals will only be shown here, as their properties remain similar for all the other simulation that will be examined further on.

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

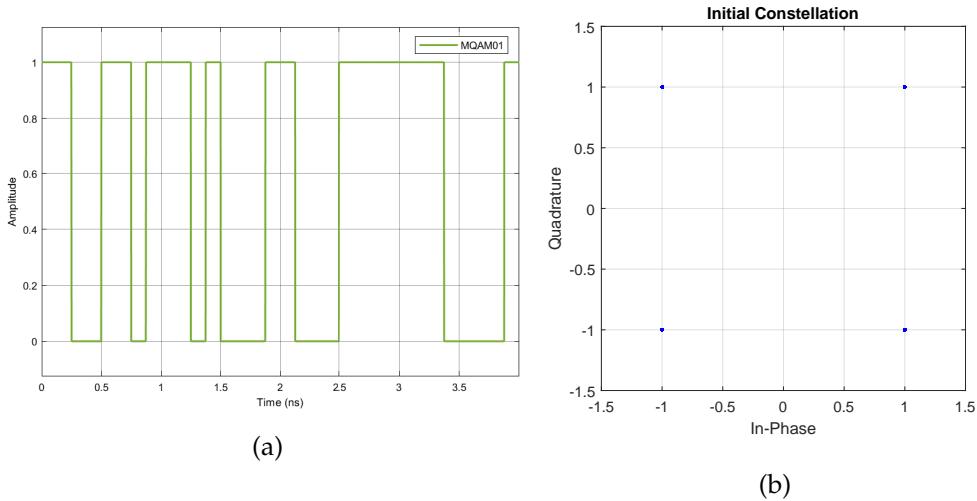


Figure 6.72: Eye diagram of initial bitstream MQAM01 (pseudorandom ones and zeros), and respective constellation to be used.

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

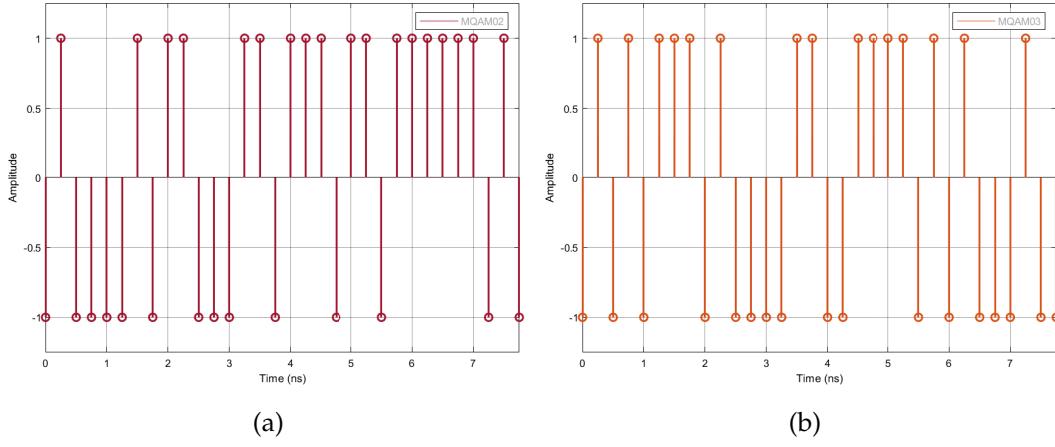


Figure 6.73: Signals MQAM02 (a) and MQAM03 (b), containing the values encoded and distributed to the in-phase and quadrature components.

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

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

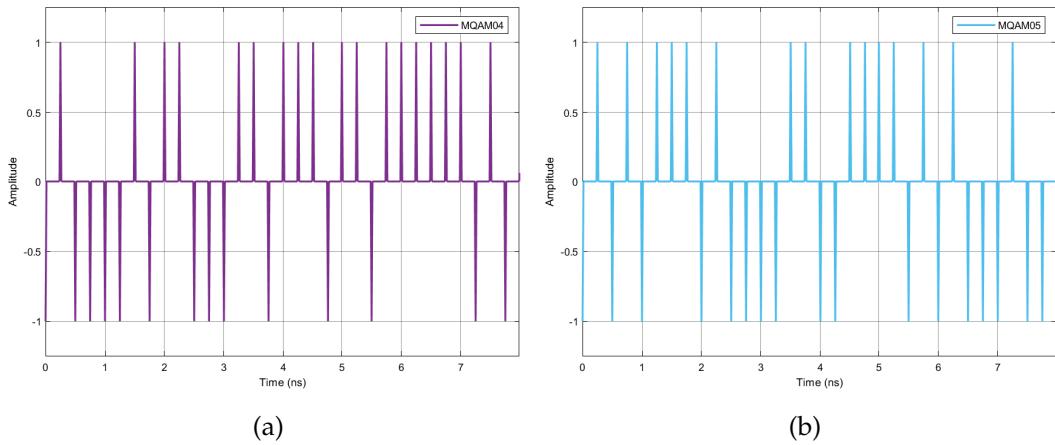


Figure 6.74: Signals MQAM04 (a) and MQAM05 (b), containing the values in a signal with continuous time.

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

chosen filter is a root-raised-cosine.

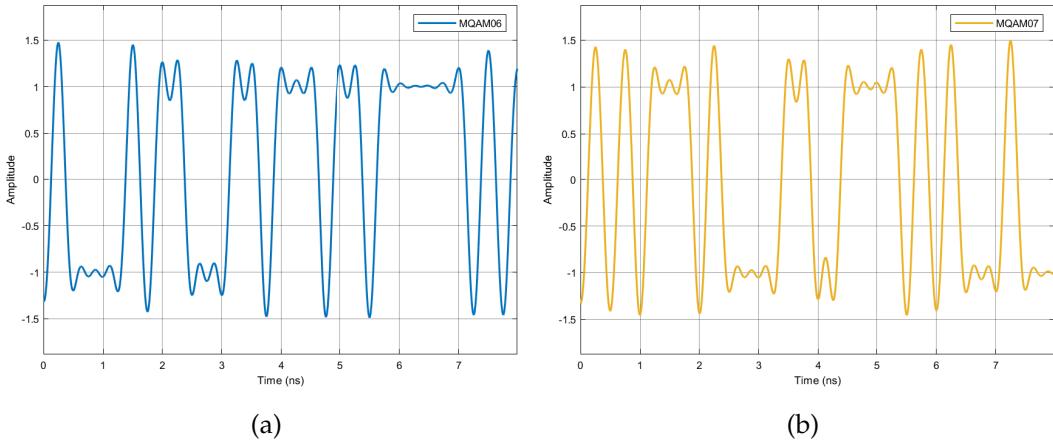


Figure 6.75: Signals MQAM06 (a) and MQAM07 (b), containing the signals to be modulated, already shaped with a root-raised-cosine filter.

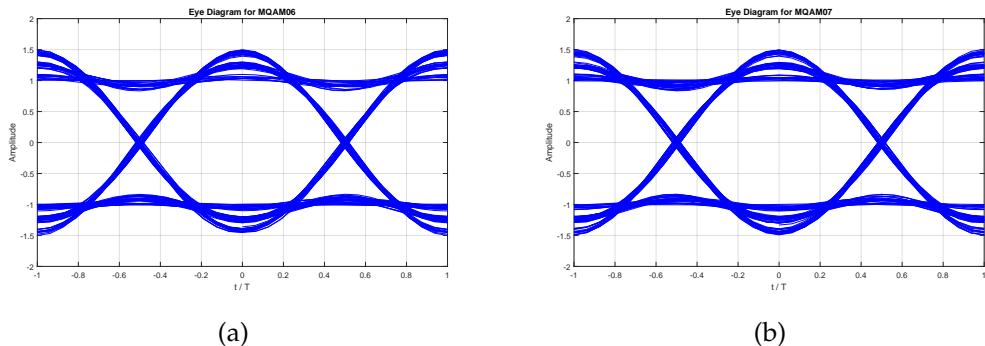


Figure 6.76: Eye diagrams for MQAM06 (a) and MQAM07 (b), shaped with root-raised-cosines.

It can be seen in the eye diagrams that the signal is not free from ISI. However, as mentioned in section 6.9.2.3, the shaping is done at the transmitter and at the receiver. We shall see further ahead that the signal will be free of ISI at sampling time.

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

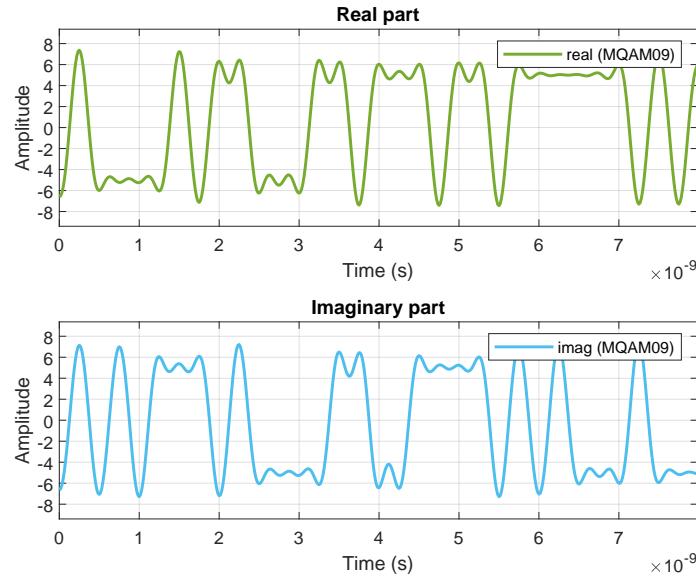


Figure 6.77: Signal MQAM09, modulated optical signal.

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

We shall now explore the process inside the *homodyne\_receiver\_withoutLO* block. The inputs of the block are mixed inside an *optical\_hybrid* block. These outputs are a mix of the transmitted optical signal and the local oscillator, as explained in section 6.9.2.2.

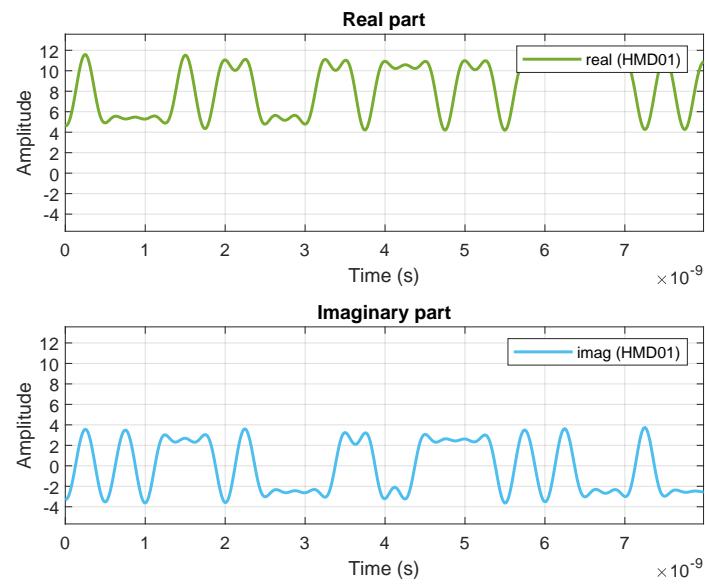


Figure 6.78: HMD01, output 1 optical hybrid.

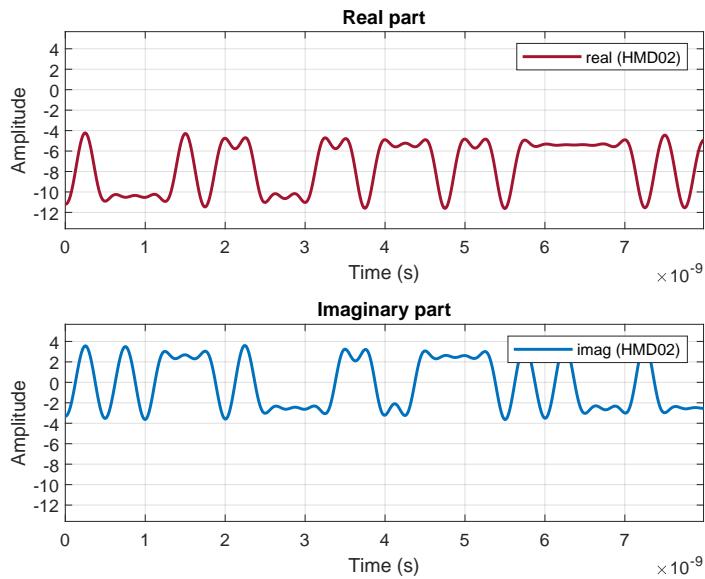


Figure 6.79: HMD02, output 2 of the optical hybrid.

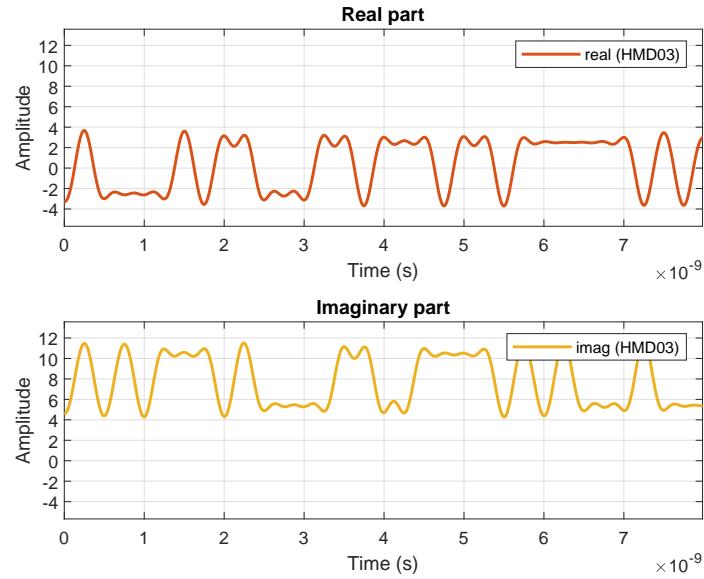


Figure 6.80: HMD03, output 3 of the optical hybrid.

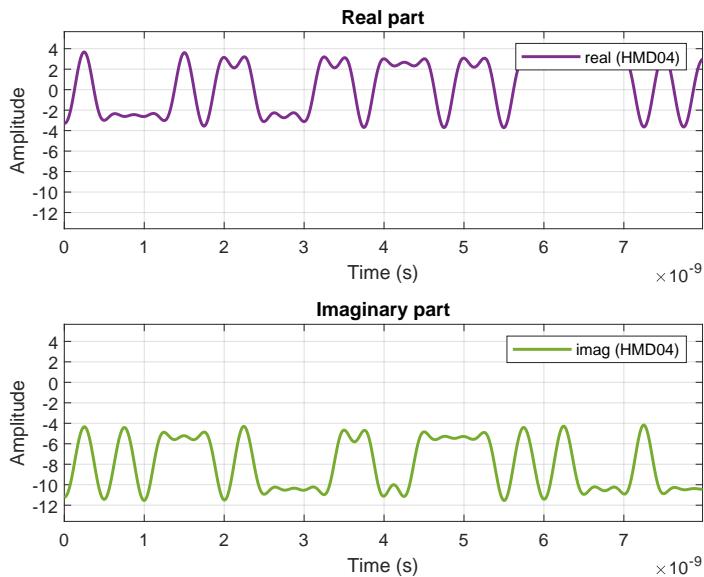


Figure 6.81: HMD04, output 4 of the optical hybrid.

These optical signals are then sent in pairs to two *photodiode* blocks, where they are detected (with a *responsivity* defined in the parameters) and subtracted. The output of the photodiode blocks is then directed to the *ti\_amplifier* blocks, which generate the signals shown in Figure 6.82. The *ti\_amplifier* usually does three things: it adds noise, amplifies the signal and noise, and lastly passes the signal and noise through a filter. However, as

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

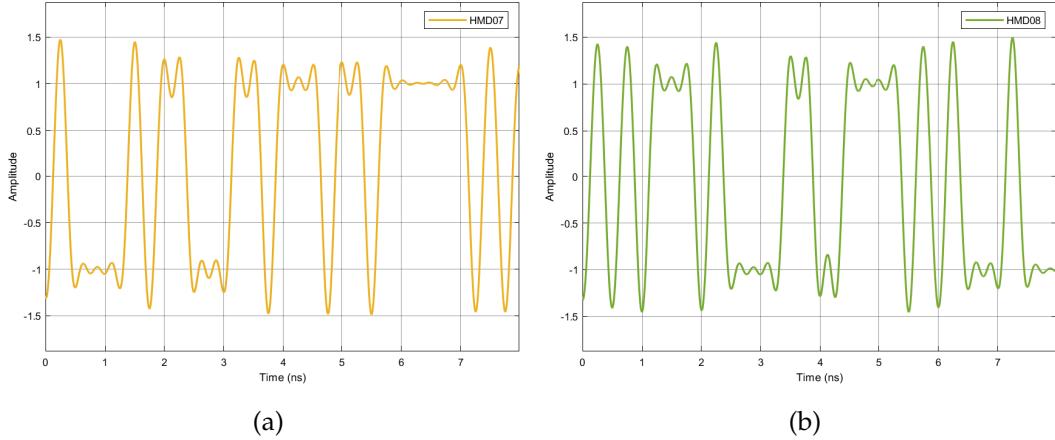


Figure 6.82: Signals HMD07 (a) and HMD08 (b), containing the amplified versions of the signals detected at the *photodiode* blocks

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

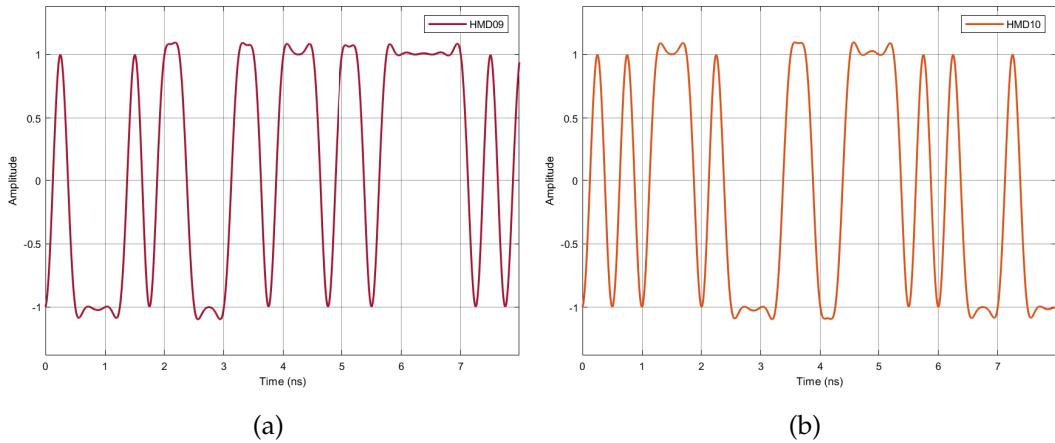


Figure 6.83: Signals HMD09 (a) and HMD10 (b), after the root-raised-cosine matched filter.

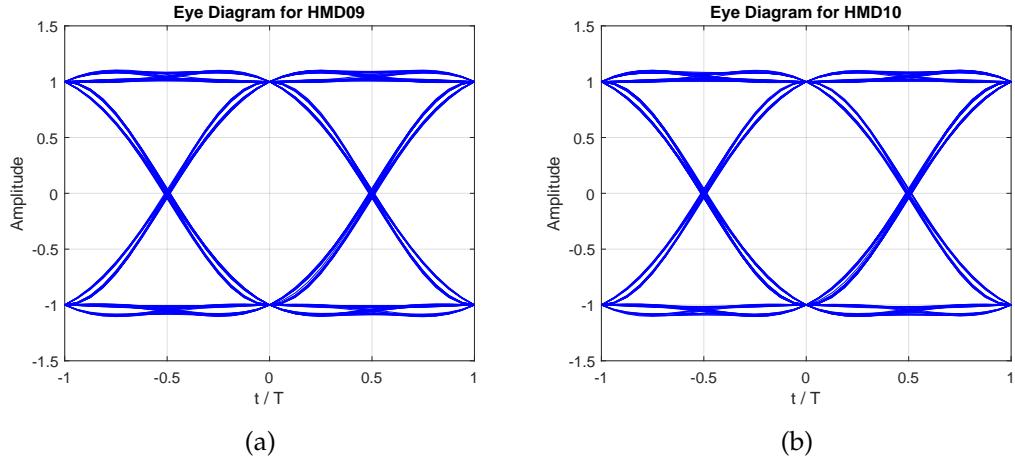


Figure 6.84: Eye diagrams for HMD09 (a) and HMD10 (b), after the root-raised-cosine matched filter. They are now following a raised-cosine shape.

For the purpose of this simulation, no electrical noise is added at the receiver. So HMD09 and HMD10 are effectively the signals that will be sampled. As can be seen from their eye diagrams in Figure 6.84, they suffer from no intersymbol interference, having always the same exact value at sampling time. This is then sampled and then decoded, transforming the received signal into a bitstream. As the transmission and reception are perfect, the received bitstream is exactly equal to the transmitted version.

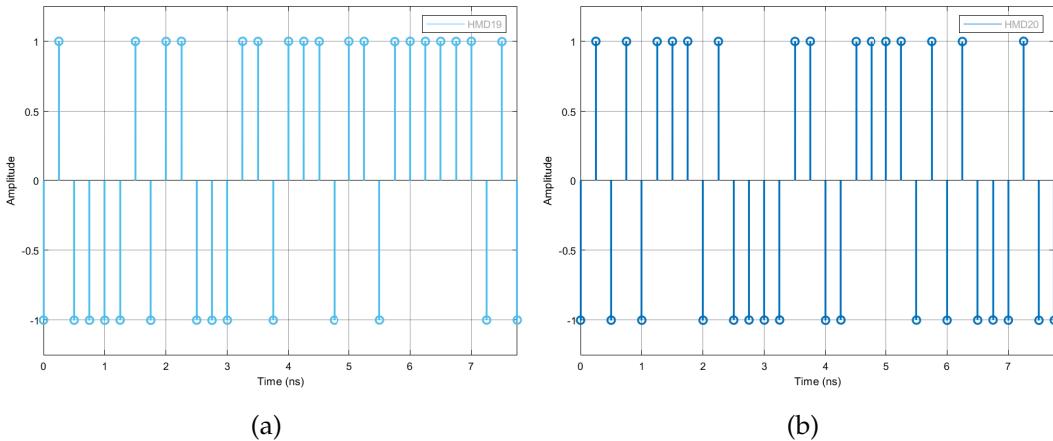


Figure 6.85: Signals HMD19 (a) and HMD20 (b), sampled at instants  $t = T_s$ .

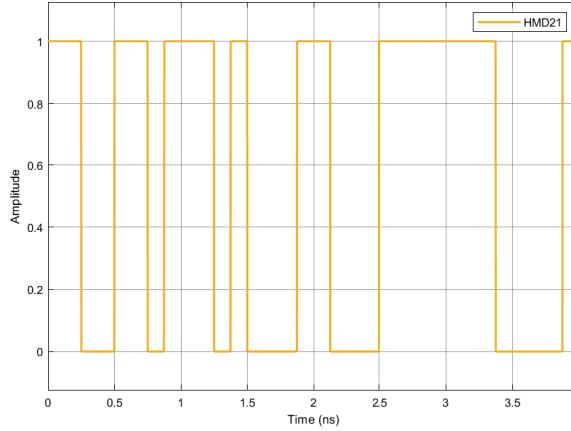


Figure 6.86: Signal HMD21, containing the decoded bitstream.

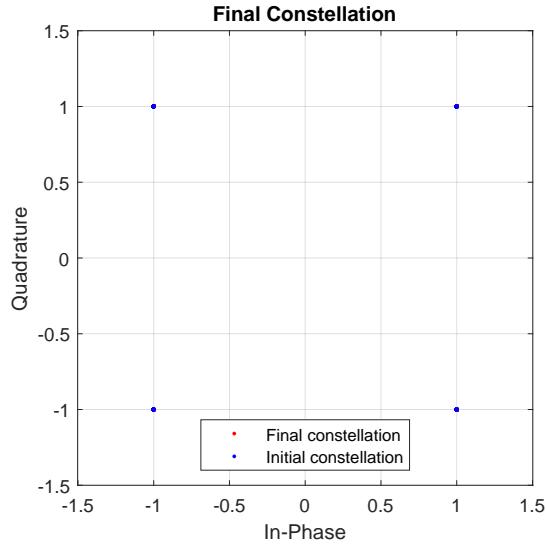


Figure 6.87: Constellation of the encoded and decoded signals. They are exactly equal due to lack of ISI and noise.

We can see that the received constellation is almost equal to the transmitted one, with all received bits having nearly no variation from their amplitude value. However, looking closely, we can still see a bit of the red points in figure 6.87. This is a consequence of the finite impulse response of the used filters. The ideal filters, which provide zero ISI, have infinite impulse response.

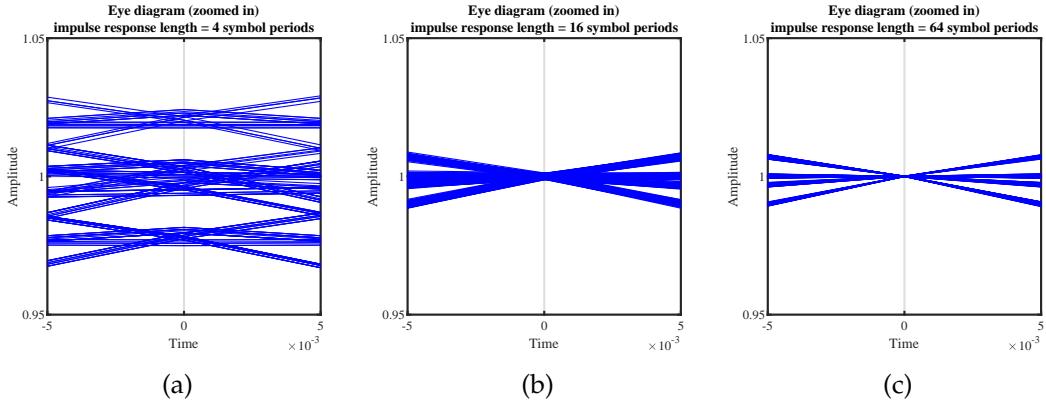


Figure 6.88: Zoom in on the eye diagram at sampling time. The figures used filter impulse responses with lengths of (a) 4, (b) 16 and (c) 64 symbol periods. As the filter impulse response increases in size, the vestigial amount of ISI diminishes.

However, the practical implementation of the FIR filter implies that the impulse response is finite, leading to very small variations at sampling time. This effect can be diminished by using a larger impulse response. We have conducted the simulations using impulse responses with a size of 16 symbol periods. We can see in Figure 6.88b that this effect is not particularly significant for this situation, and thus we can consider the intersymbol interference to be negligible.

#### 6.9.3.6 Simulation results - Thermal Noise

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

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

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

Table 6.30: Simulation parameters

| Parameter             | Value             | Units |
|-----------------------|-------------------|-------|
| numberOfBitsGenerated | $100 \times 10^3$ |       |
| samplingRate          | $64 \times 10^9$  | Hz    |

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

The transmitted signal is similar to the one shown in section 6.9.3.5, so it shall not be shown here. The starting constellation is also similar to the one in the previous section. Figure 6.89 shows the output of the photodiodes.

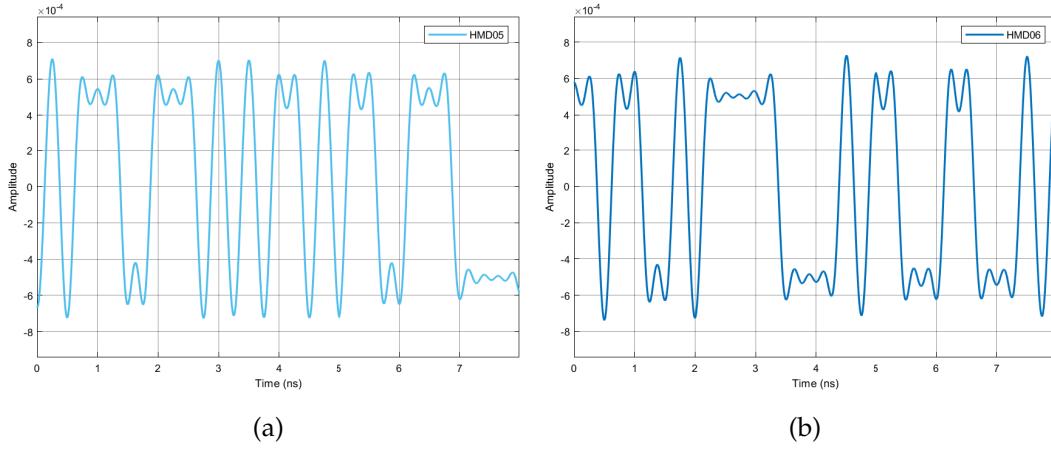


Figure 6.89: Signals HMD05 (a) and HMD06 (b), outputs of the photodiodes.

In this case no signal amplification is used, so there is no amplifier gain or noise. This means that the electrical signal will be much smaller than in the previous case. This can be verified by comparing Figures 6.90 and 6.83.

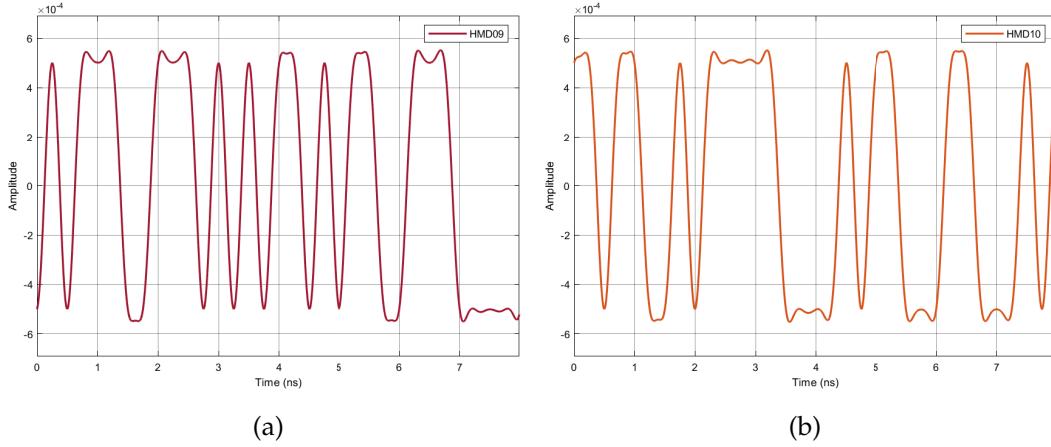


Figure 6.90: Signals HMD09 (a) and HMD10 (b), after the root-raised-cosine matched filter. They are now following a raised-cosine shape.

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

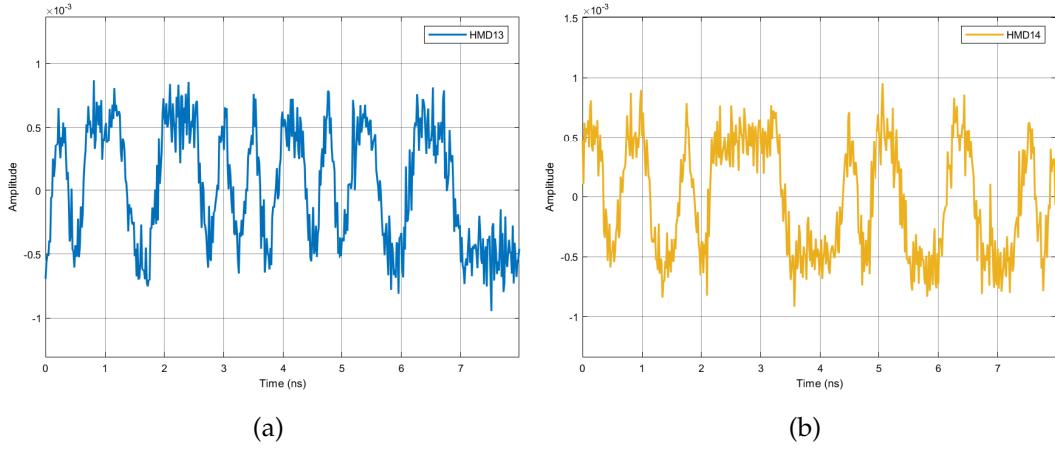


Figure 6.91: Signals HMD13 (a) and HMD14 (b), after adding thermal noise with an RMS voltage amplitude of  $1.6008 \times 10^{-4}$  V.

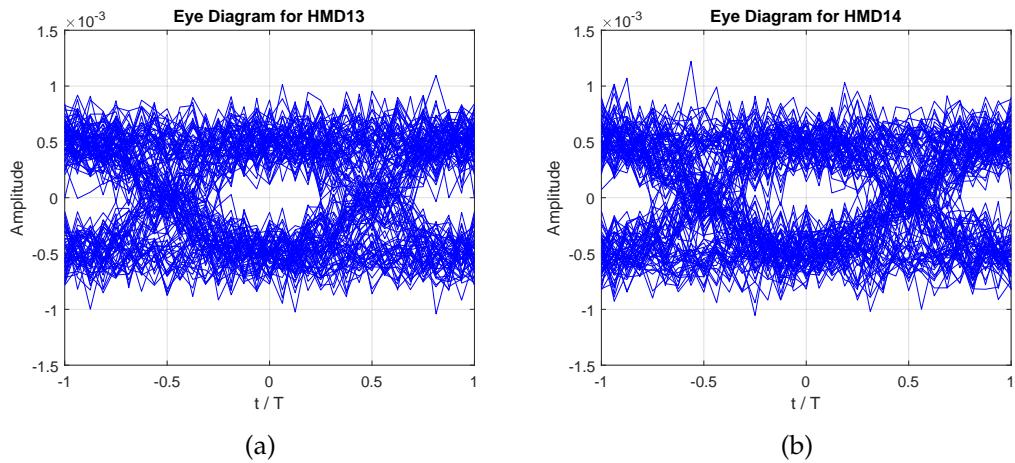
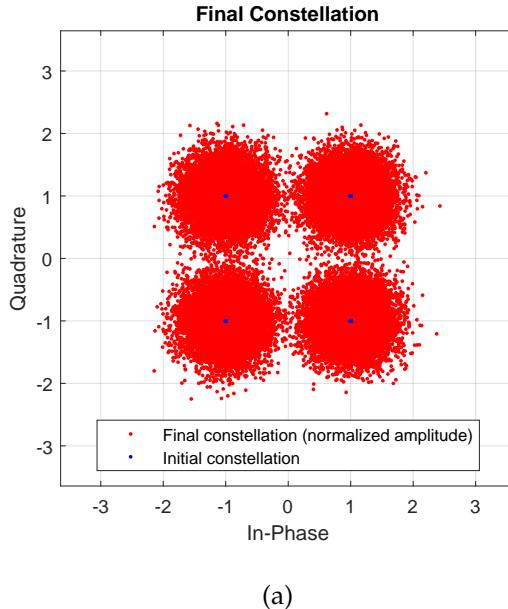


Figure 6.92: Eye diagrams of signals HMD13 (a) and HMD14 (b), after adding thermal noise with an RMS voltage amplitude of  $1.6008 \times 10^{-4}$  V.

Figure 6.93 shows the final received constellation. The received signal constellation is strongly affected by noise. The thermal noise is not a particularly strong source of noise. However, as the signal has not been amplified, they have comparable values.



(a)

Figure 6.93: Transmitted and received constellations. Signal optical output power of 0 dBm and 30 km of fiber length (-6 dBm at receiver input).

The signals shown so far had an optical output power of -6 dBm. By attenuating using various fiber lengths, we plot a BER curve to study receiver sensitivity. Receiver sensitivity is usually defined as the minimum signal optical power required to achieve a certain BER performance [14]. The target BER used to measure the sensitivity may vary, and is usually chosen considering a desired minimum performance. For instance, in some cases a BER smaller than  $10^{-12}$  may be desirable to ensure a very low error rate. In other cases, the requirement may be only  $10^{-3}$  in order for the FEC to be effective.

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

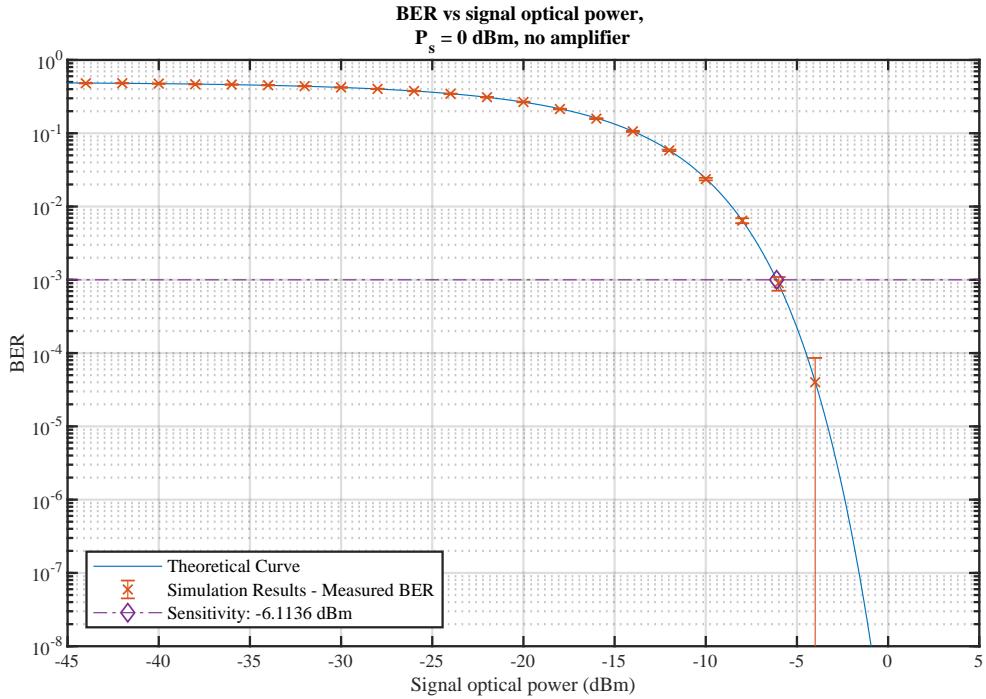


Figure 6.94: BER curve of the receiver with thermal noise. The sensitivity was obtained for a BER of  $10^{-3}$ .

The theoretical curve was obtained with

$$\text{BER} = \frac{1}{2} \operatorname{erfc} \left( \frac{A}{\sqrt{2N}} \right) \quad (6.120)$$

with

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

$$N = 4k_B TBR$$

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

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

The sensitivity considering the thermal noise establishes the baseline sensitivity upon which to compare the rest of them. In the mentioned conditions, we have measured a

sensibility of -6.1136 dBm when using no amplification. We can use this value to compare with the rest of the simulations, in order to quantify the increased performance provided by improvements on the system.

#### 6.9.3.7 Simulation results - Electrical Amplifier Noise

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

Table 6.31: Simulation parameters

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

|               |        |  |
|---------------|--------|--|
| bitSourceMode | Random |  |
| confidence    | 0.95   |  |

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

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

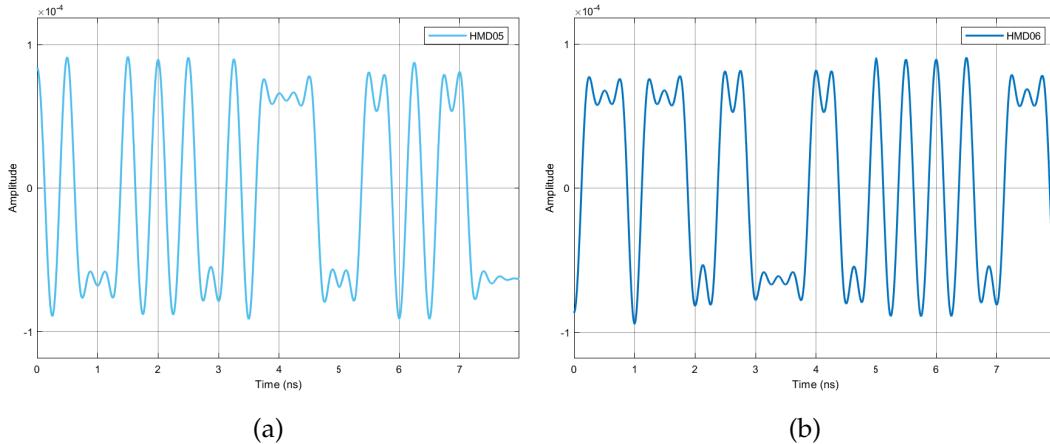


Figure 6.95: Signals HMD05 (a) and HMD06 (b), after detection in the photodiodes.

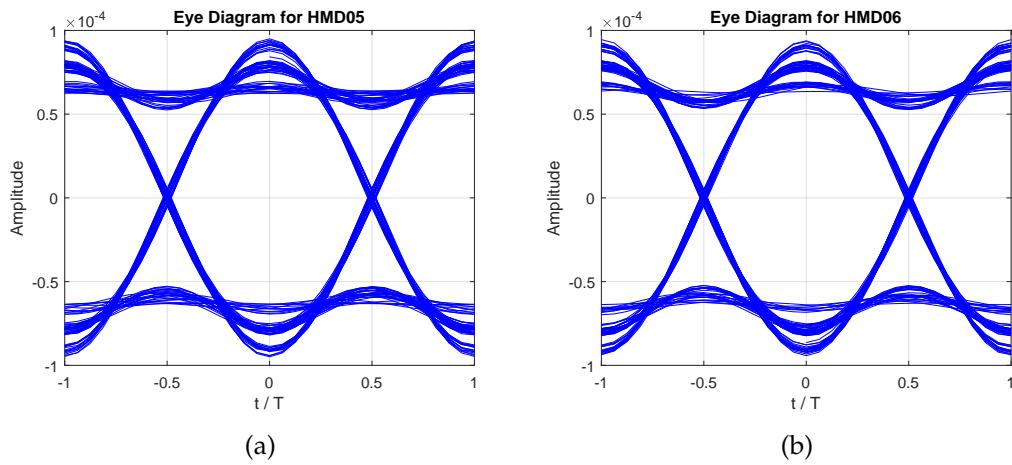


Figure 6.96: Eye diagrams of signals HMD05 (a) and HMD06 (b), after detection in the photodiodes. Shaped with a root-raised cosine filter.

The main difference between previous cases and this one is the existence of an amplifier

after the photodiodes. This amplifier has three properties which affect the signal: a gain, an input referred noise source, and a bandwidth. The bandwidth is much larger than the signal bandwidth, and therefore does not have any appreciable effects other than limiting the noise bandwidth. The main effects of the amplifier, as shown in Figures 6.97 and 6.98, are the amplification of the signal and the introduction of noise. The noise standard deviation increases proportionally to the amplifier gain.

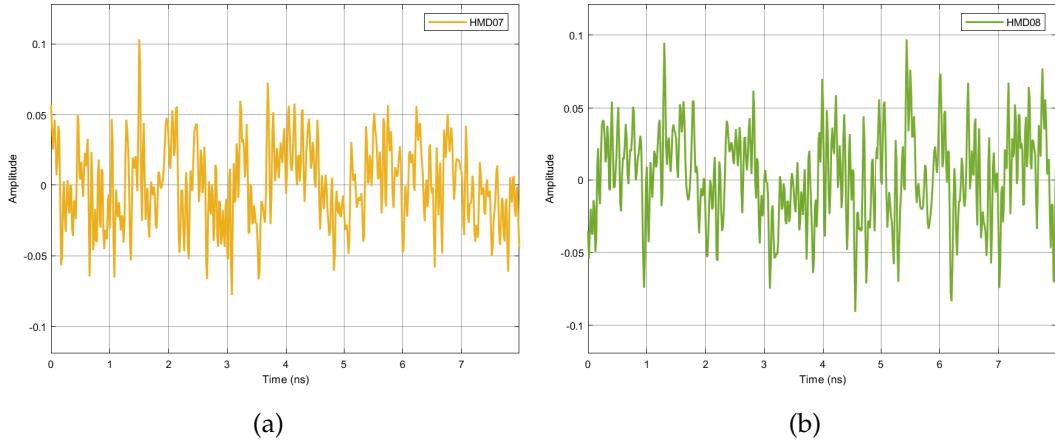


Figure 6.97: Signals HMD07 (a) and HMD08 (b), after detection in the photodiodes.

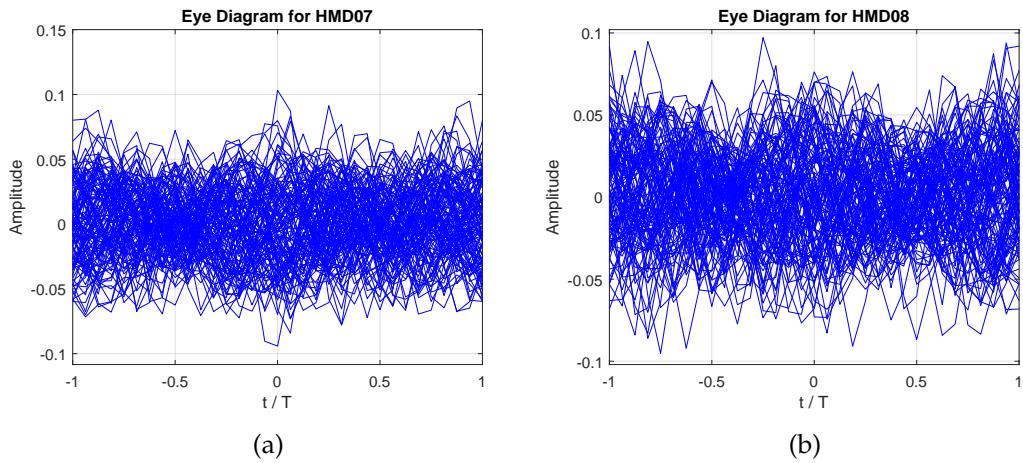


Figure 6.98: Eye diagrams of signals HMD07 (a) and HMD08 (b), after detection in the photodiodes.

The new noise source is placed prior to the matched filter. As such, the noise is strongly attenuated when the signal goes through the matched filter. This can be seen particularly well by comparing the eye diagrams of Figures 6.98 and 6.100. In the latter, the eye is widely open when compared to the former, and the variations in the signal due to noise are much smoother.

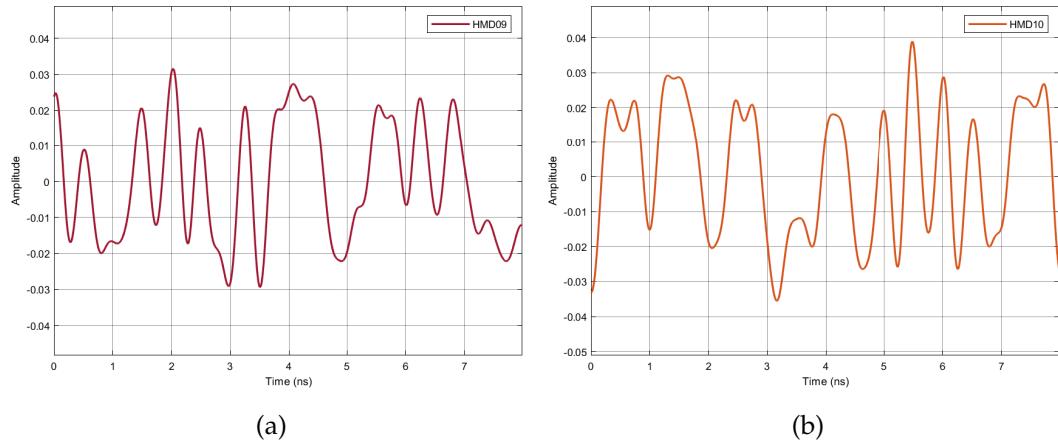


Figure 6.99: Signals HMD09 (a) and HMD10 (b), after detection in the photodiodes.

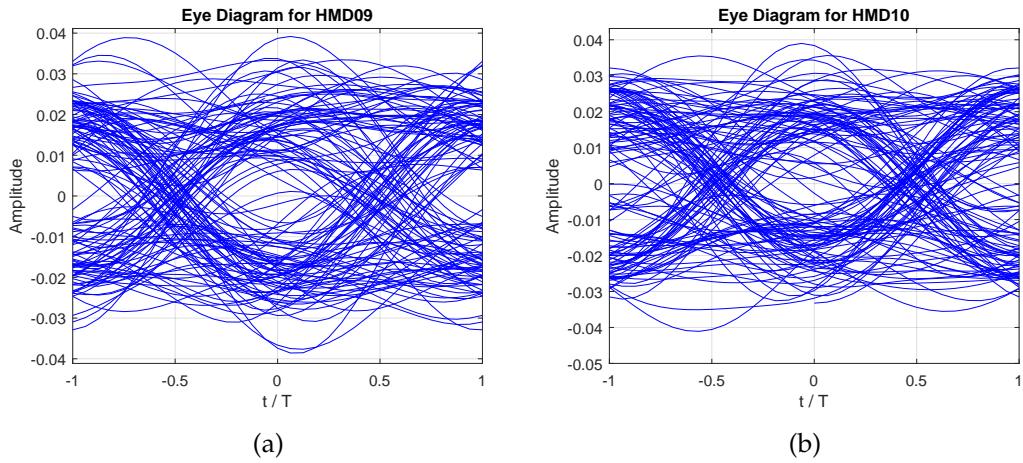


Figure 6.100: Eye diagrams of signals HMD09 (a) and HMD10 (b), after detection in the photodiodes.

Lastly, thermal noise is added prior to sampling. However, as can be seen in Figures 6.101 and 6.102, thermal noise is negligible in this case, as the signal has been amplified to several order of magnitude above the thermal noise level.

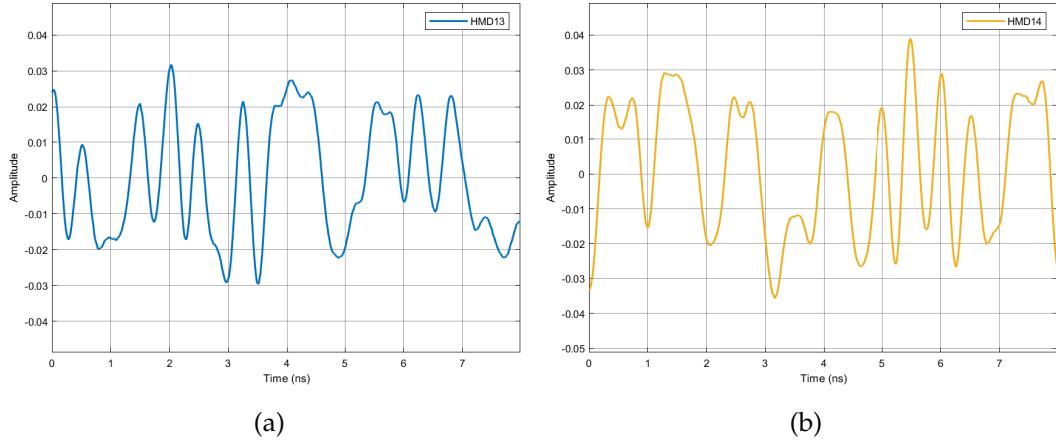


Figure 6.101: Signals HMD13 (a) and HMD14 (b), after detection in the photodiodes.

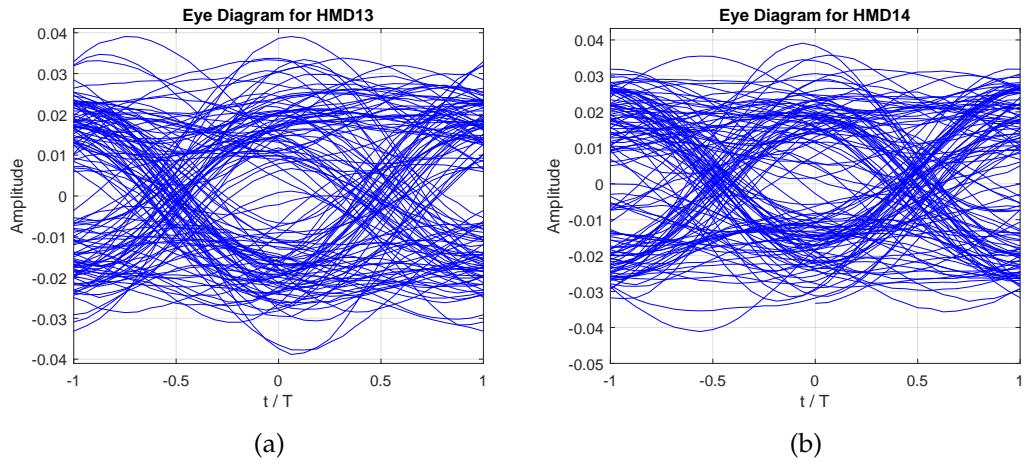


Figure 6.102: Eye diagrams of signals HMD13 (a) and HMD14 (b), after detection in the photodiodes.

We then get the constellation shown in Figure 6.103. Comparing with Figure 6.93 from the previous section, they appear to have a similar signal to noise ratio. While this might be true, it's worth noting that the constellation shown here has a much higher amplitude, equal to the constellation at the transmitter. In addition, it was obtained with a much weaker signal, -20 dBm, compared with 0 dBm in the previous example.

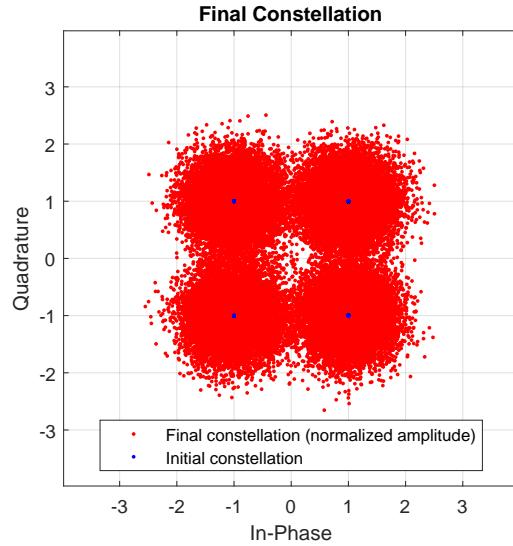


Figure 6.103: Final constellation. Signal optical output power of -20 dBm and 20 km of fiber length (-24 dBm at receiver input).

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

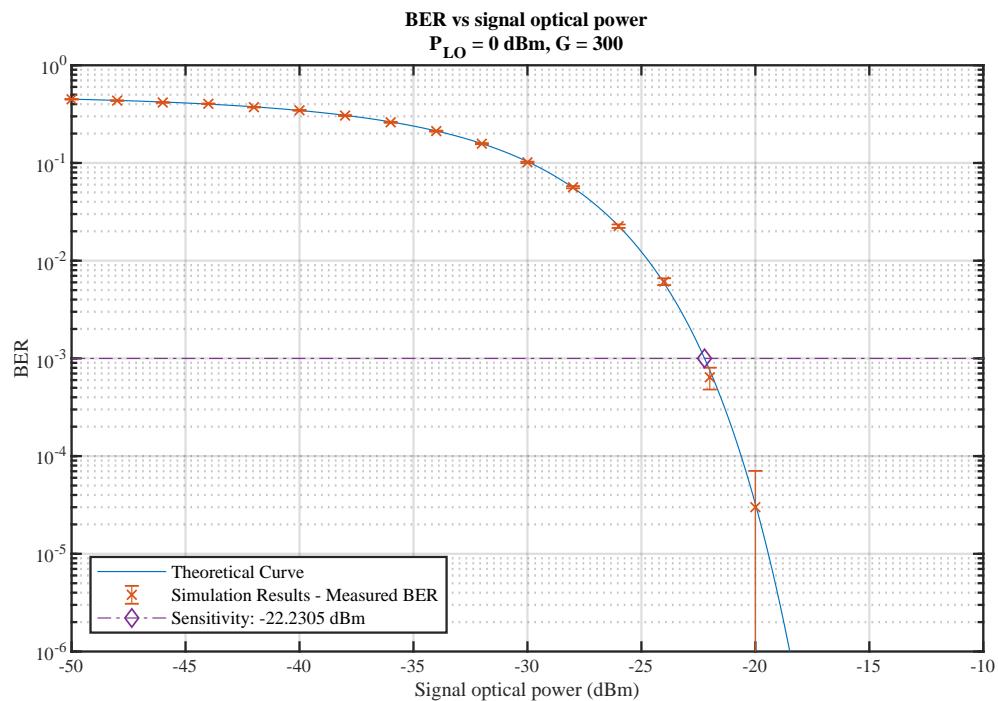


Figure 6.104: BER curve. LO power at 0 dBm, amplifier gain set at 300.

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

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

$$G_e = \frac{A_g}{\eta K \sqrt{P_s e^{-L\alpha} P_{LO}}} \quad (6.121)$$

The curve can now be obtained with Equation 6.120, with

$$\begin{aligned} A &= \eta G_e K \sqrt{P_s P_{LO} e^{-L\alpha}} \\ N &= 4k_B TBR + \left( \sqrt{n_{in} \frac{1}{T_s}} G_e K \right)^2 \end{aligned} \quad (6.122)$$

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

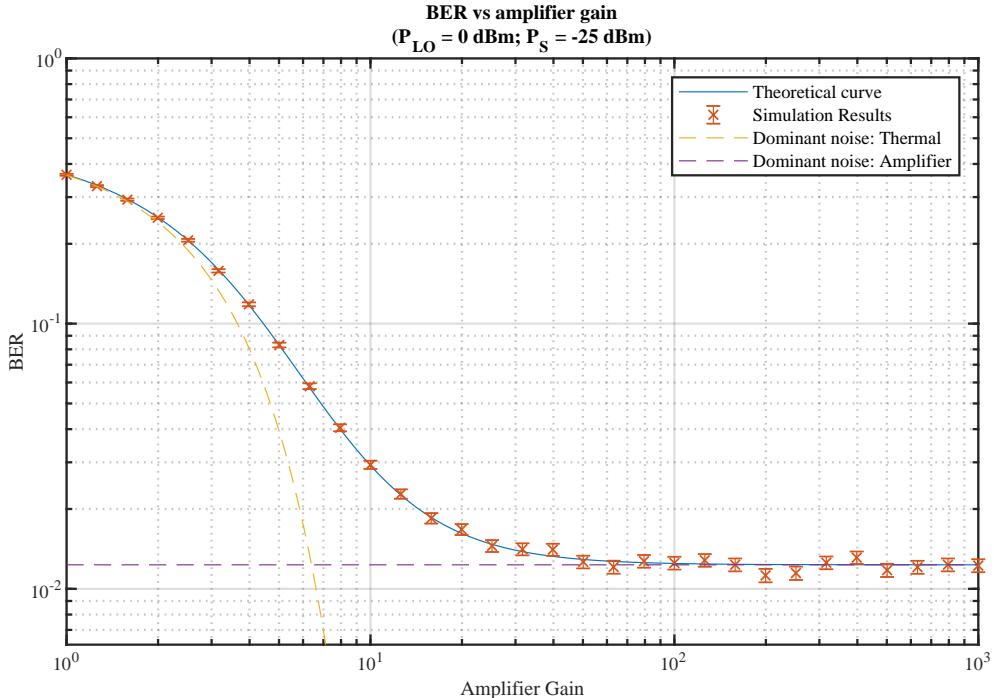


Figure 6.105: BER as a function of the transimpedance amplifier gain, along with the curves where the individual noise sources dominate. Signal output power of -25 dBm and local oscillator power of 0 dBm.

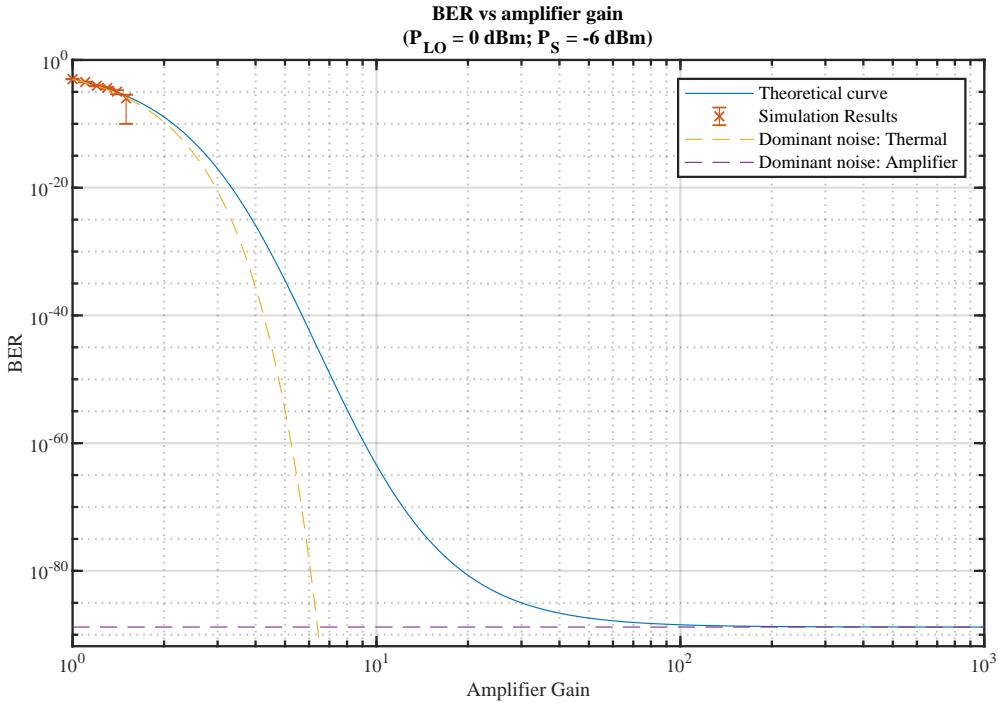


Figure 6.106: BER as a function of the transimpedance amplifier gain, along with the curves where the individual noise sources dominate. Signal output power of -6 dBm and local oscillator power of 0 dBm.

We can see the direct effect of the amplifier on the BER on Figure 6.105. The yellow curve is the expected behavior if thermal noise was the only noise source. It better demonstrates the system behavior at low gains, reflecting the case where the thermal noise overwhelms the amplifier-generated noise. As the gain increases, the thermal noise component becomes less significant when compared to the signal and to the amplifier noise. When the gain is high enough ( $10^2$  in the plot), thermal noise is negligible when compared to the amplifier output, and system follows the purple line. This line shows the expected behavior if the only noise present was due to the amplifier. At this point, the performance limiting factor is the relation between the current generated at the photodiodes and the input referred noise of the amplifiers. This ratio is constant and independent from the gain.

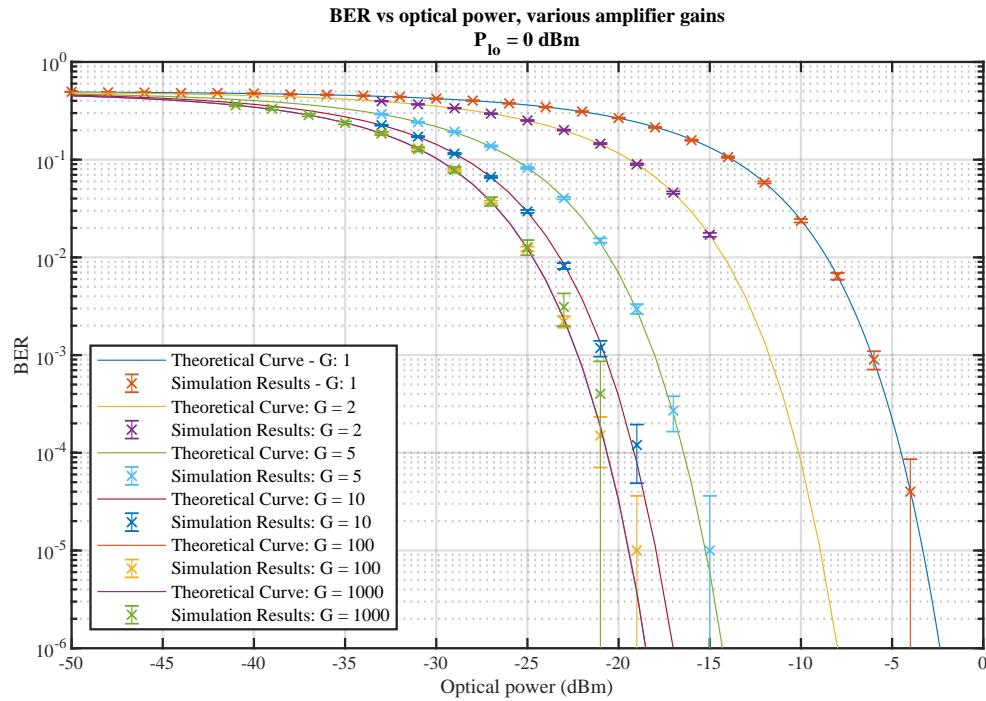


Figure 6.107: Comparison of BER curves with different gains. Local oscillator power of 0 dBm. We can see that when  $G=1$ , the curve is similar to the case without amplifier shown in Figure 6.94

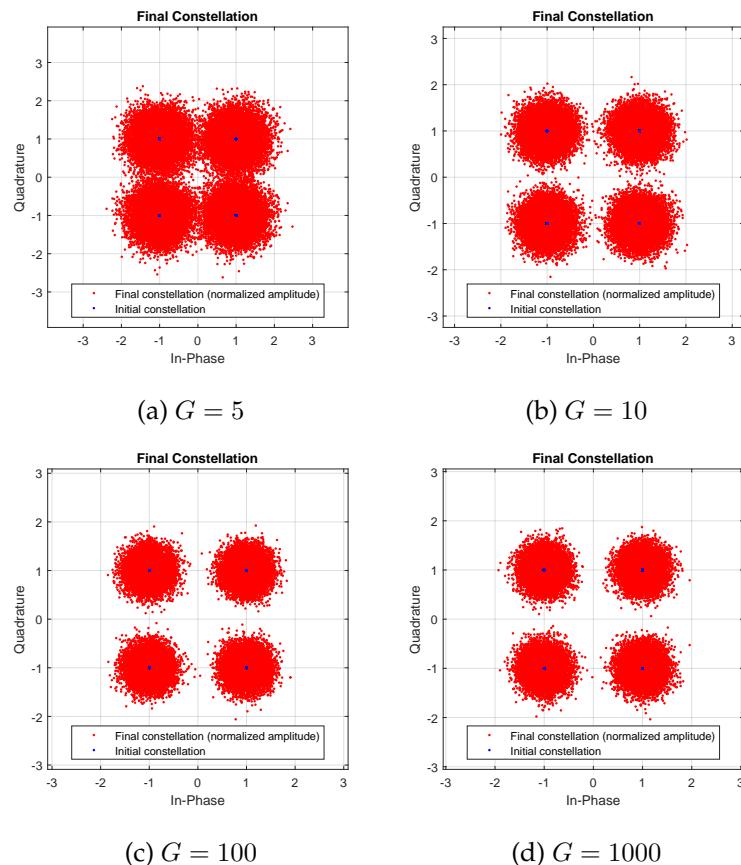


Figure 6.108: Final constellations for various local oscillator optical powers. Signal optical power at is -15 dBm, with 20 km of fiber length (-19 dBm at receiver input). It is easy to see that there does not appear to be a significant improvement between the constellations with the gains of 100 and 1000.

### 6.9.3.8 Simulation results - Increased LO

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

Table 6.32: Simulation parameters

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

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

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

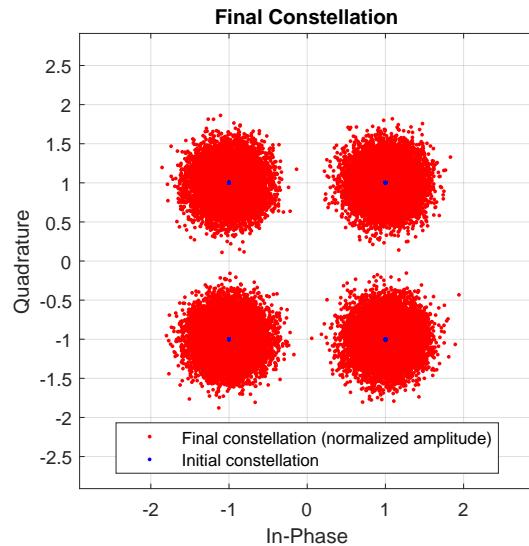


Figure 6.109: Final constellation (normalized). Signal optical output power of -20 dBm and 20 km of fiber length (-24 dBm at receiver input). Local oscillator at 5 dBm.

In Figure 6.109 we can see the final constellation obtained when choosing an output optical power of -20 dBm and setting the fiber length to 20 km. In conditions similar to Figure 6.103 it shows a much better constellation. This is all due to the higher local oscillator value.

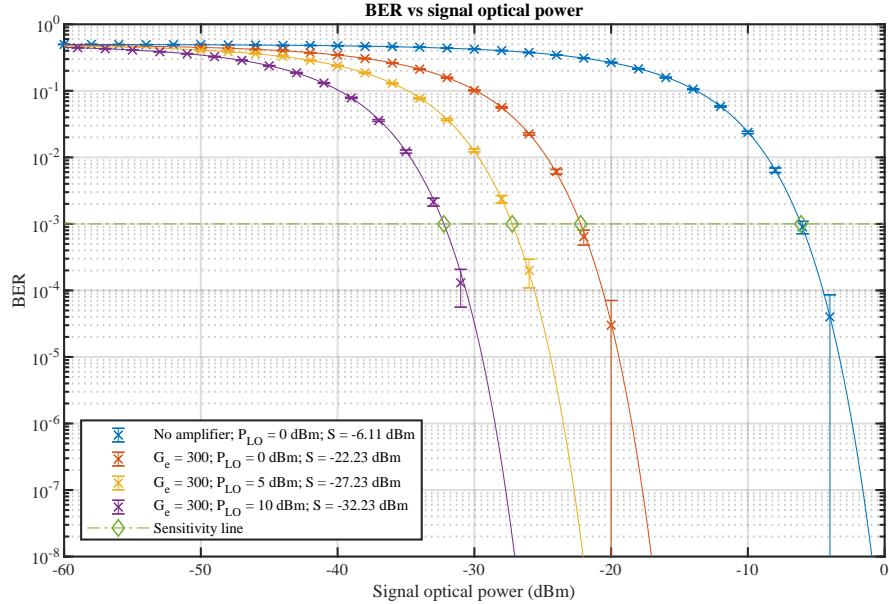


Figure 6.110: Comparison of BER curves. The conditions are described in the legend:  $G_e$  is the electrical amplifier gain,  $P_{LO}$  is the local oscillator power and S is the obtained sensitivity at a BER of  $10^{-3}$ .

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

The theoretical curve in Figure 6.110 was obtained according to the same equation as the second curve in the previous section. Notice that the new curve is approximately 5 dB to the left of curve with the amplifier and the LO at 0 dBm. This is because in the equations 6.120 and 6.122, the power of the local oscillator is as important as the optical signal power, and the amplitude at sampling time grows at the same rate with both quantities.

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

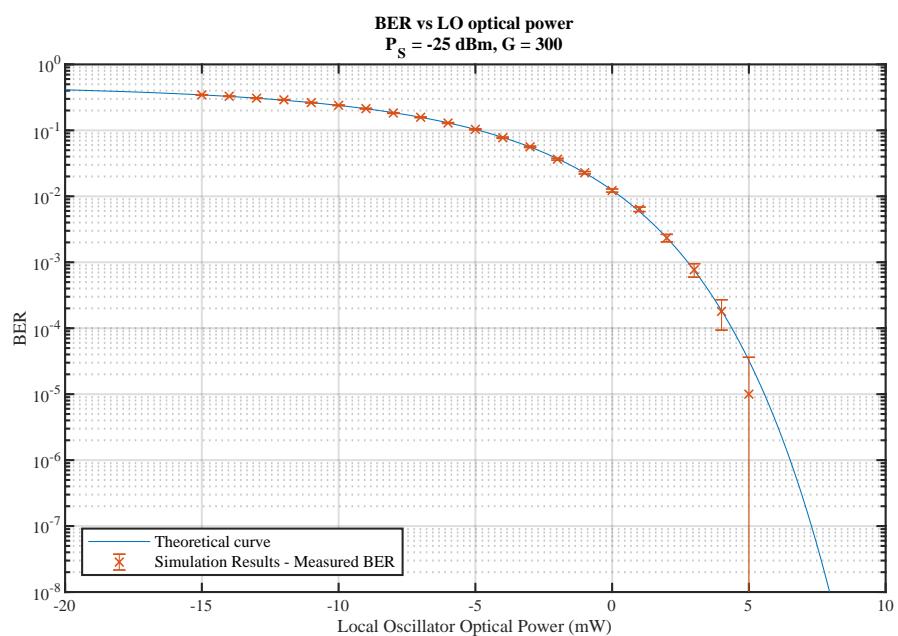


Figure 6.111: Variation of BER with the LO power. Signal optical power at receiver input is -25 dBm.

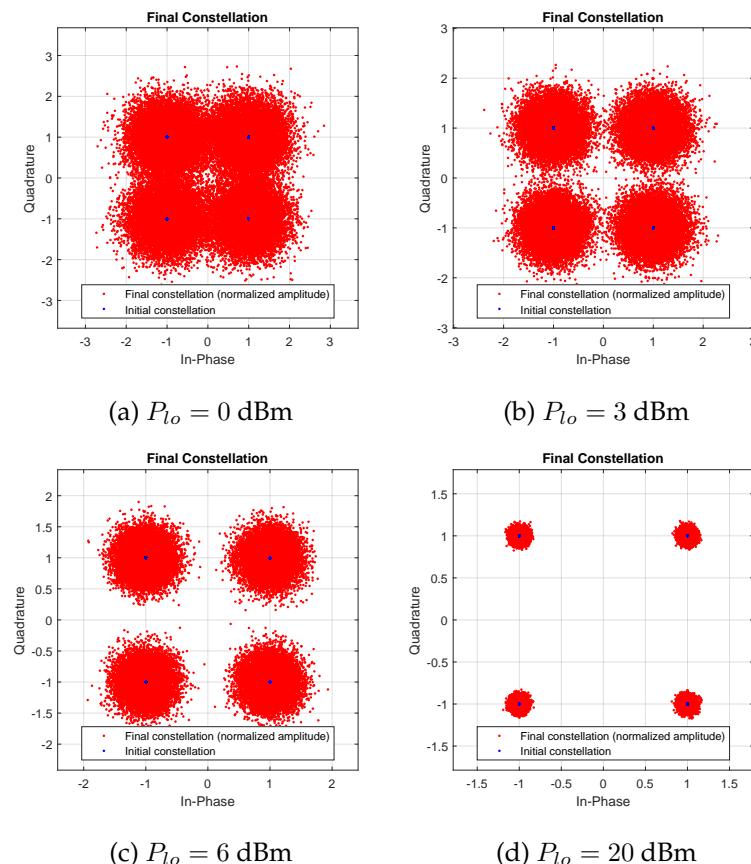


Figure 6.112: Final constellations for various local oscillator optical powers. Signal optical power at receiver input is -25 dBm.

### 6.9.3.9 Simulation results - Quantum Noise

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

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

$$\sigma_{\text{shot}}^2 = 2qIB = 2\eta P_{\text{opt}}B \quad (6.123)$$

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

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

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

$$A = \eta \sqrt{P_{\text{LO}}P_s} \quad (6.124)$$

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

$$N_{\text{shot i}} = 2\eta h f \frac{P_{\text{LO}}}{4} B \quad (6.125)$$

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

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

$$\begin{aligned} N_{\text{shot}} &= N_{\text{shot 1}} + N_{\text{shot 2}} \\ &\approx 2N_{\text{shot i}} = \\ &= 4\eta h f \frac{P_{\text{LO}}}{4} B \end{aligned} \quad (6.126)$$

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

$$\begin{aligned} A &= \eta G_e K \sqrt{P_s P_{\text{LO}} e^{-L\alpha}} \\ N &= 4k_B TBR + \left( \sqrt{\frac{1}{n_{in} T_s}} G_e K \right)^2 + \left( \sqrt{\frac{N_{\text{shot}}}{B} \frac{1}{T_s}} G_e K \right)^2 \end{aligned} \quad (6.127)$$

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

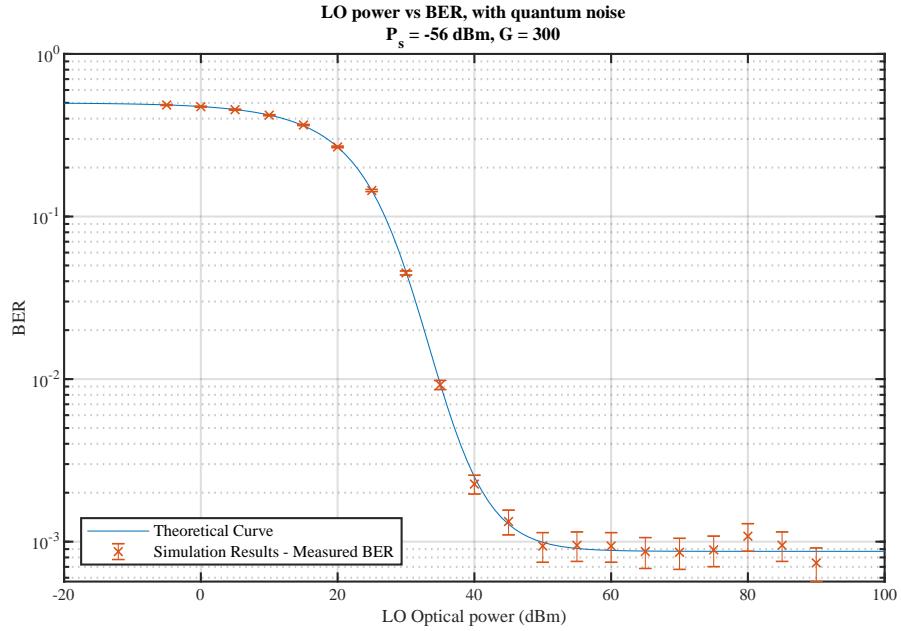


Figure 6.113: Variation of BER with the LO power, with shot noise. Signal optical power at receiver input is -56 dBm.

modulation scheme and type of detector. In the case of QPSK with homodyne detection this limit is given by [15, 13]:

$$\text{BER} = \frac{1}{2} \operatorname{erfc} \left( \sqrt{2\eta_q N_p} \right) \quad (6.128)$$

where  $\eta_{q,p}$  is the quantum efficiency of the receiver, and  $N_p$  is the number of photons per bit. We can convert the average number of photons per bit to dBm or vice versa:

$$P_{\text{dBm}} = 10 \log_{10} \left( \frac{hf}{T_b} N_p \times 10^3 \right) \quad (6.129)$$

$$N_p = \frac{10^{(P_{\text{dBm}}/10)} T_b \times 10^{-3}}{hf} \quad (6.130)$$

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

We can see that in this case the sensitivity at a BER  $10^{-3}$  is -56.1 dBm, which is at the quantum limit. This corresponds to an average of 2.4 photons per bit.

#### 6.9.3.10 Simulation Results - Optical Preamplifier

Lastly, we will look at the case where an optical preamplifier is used before the homodyne receiver. The EDFA block is used as a preamplifier and it has two effects: it outputs an ideally

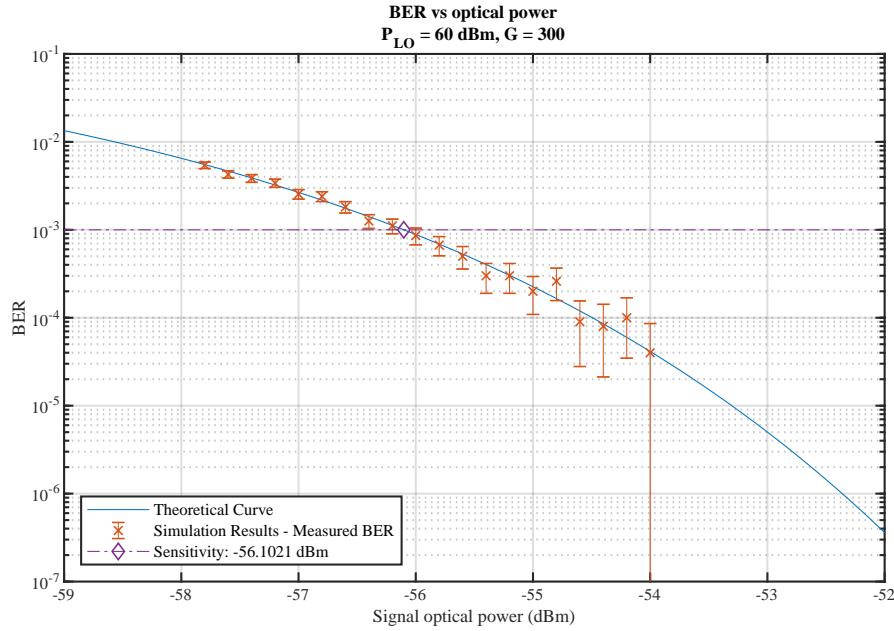


Figure 6.114: BER variation with the optical power.  $P_{lo} = 60 \text{ dBm}$ . Sensitivity obtained for a BER of  $10^{-3}$ .

amplifier version of the signal, by a gain defined in the variable *opticalGain\_dB*; and it adds white noise corresponding to the ASE noise produced in optical amplifiers. The generated noise spectral density is obtained from the noise figure  $N_F$  and optical gain  $G_o$  as follows:

$$\begin{aligned} n_{ASE} &= \frac{hc}{\lambda} (G_o - 1) n_{sp} \\ &= \frac{hc}{\lambda} (G_o - 1) N_F / 2 \end{aligned} \quad (6.131)$$

Here,  $n_{ASE}$  is the optical noise spectral density at the output of the EDFA block,  $h$  is the Planck constant,  $c$  is the speed of light,  $\lambda$  is the wavelength, and  $n_{sp}$  is the spontaneous emission factor of the EDFA.

It is important to notice that the EDFA block currently does not account for saturation behavior. Thus, the gain is always constant, independently of its optical input power or any other factors.

We can then obtain  $A$  and  $N$  for plotting the theoretical BER curves. For now we shall consider that the effects of the ASE noise is completely explained by the ASE-LO beat noise. There should also be noise from Signal-ASE and ASE-ASE interactions, but the ASE-LO noise is dominant and we can focus on it right now.

$$\text{BER} = \frac{1}{2} \operatorname{erfc} \left( \frac{A}{\sqrt{2N}} \right) \quad (6.132)$$

$$A = \eta G_e K \sqrt{G_o P_s P_{LO} e^{-L\alpha}}$$

$$N = N_{\text{Th}} + N_{\text{Amp}} + N_{\text{shot}} + N_{\text{ASE-LO}}$$

$N_{\text{Th}}$  is the thermal noise power,  $N_{\text{Amp}}$  is the electrical amplifier generated noise power,  $N_{\text{shot}}$  is the shot noise power and  $N_{\text{ASE-LO}}$  is the ASE-LO beat noise power. These noise components are defined as in the previous sections, and  $N_{\text{ASE-LO}}$  is given by [11]:

$$N_{\text{ASE-LO}} = 2 \frac{hc}{\lambda} (G_o - 1) \frac{N_F}{2} B \eta^2 K^2 G_e^2 P_{LO} \quad (6.133)$$

When using the optical preamplifier, depending on the gain, the ASE noise component should much higher than the others. So it can be expected that the other noise sources can be neglected in this situation, including the shot noise.

Therefore, using the optical preamplifier, the shot noise limit showed in the previous section is no longer relevant. Although this setup cannot achieve that level of sensitivity, it still offers a different advantage. While it is not capable the shot noise limited sensitivity, it can still greatly improve the system's performance, while requiring lower values of local oscillator power. These lower values are of no particular consequence in the simulation, but in the real world it can be very useful to use lower optical powers.

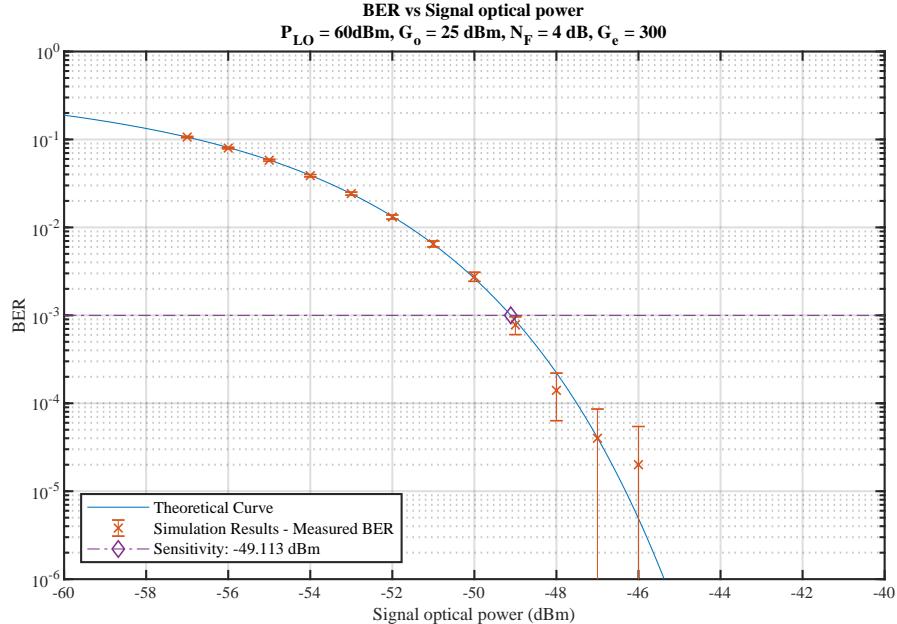


Figure 6.115

The sensitivity using  $P_{LO} = 60 \text{ dBm}$ ,  $G_o = 25 \text{ dB}$  and  $N_F = 4 \text{ dB}$  is  $-49.113 \text{ dBm}$ . As expected, this is higher than in the previous section. Nevertheless, it remains much lower than the the other cases.

Varying the optical gain only improves performance up to a certain point. We can see in Figure 6.116 that, keeping all the other conditions equal, there is no change in the BER at 40 dB and 80 dB of optical gain.

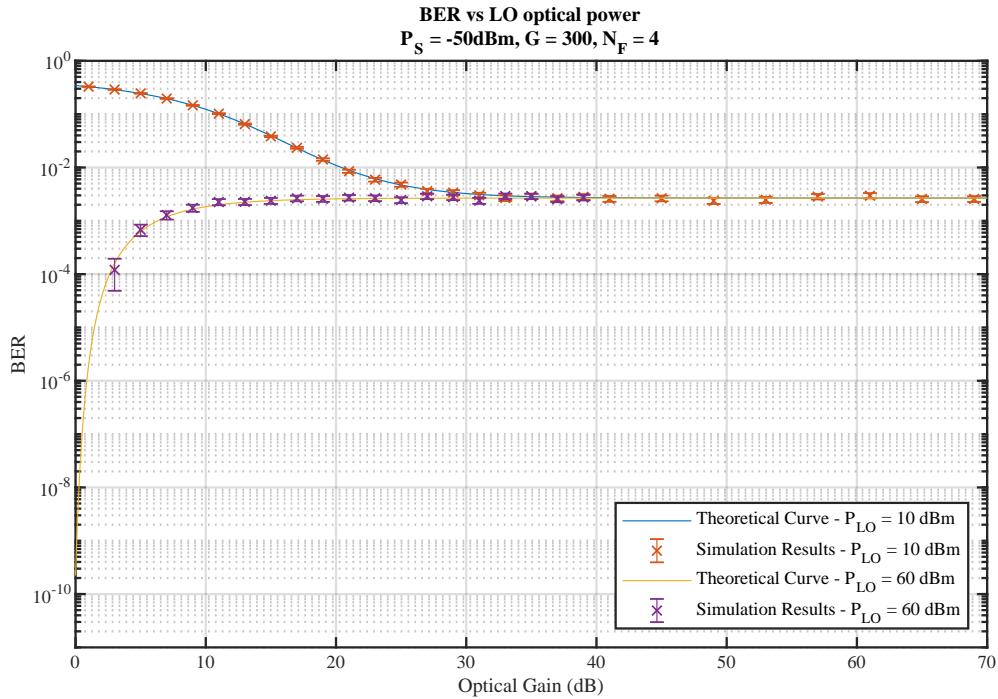


Figure 6.116

Similarly, increasing the local oscillator power improves performance, but only to a certain point. However, the limiting factor here remains the ASE-LO noise, which also grows with the local oscillator power.

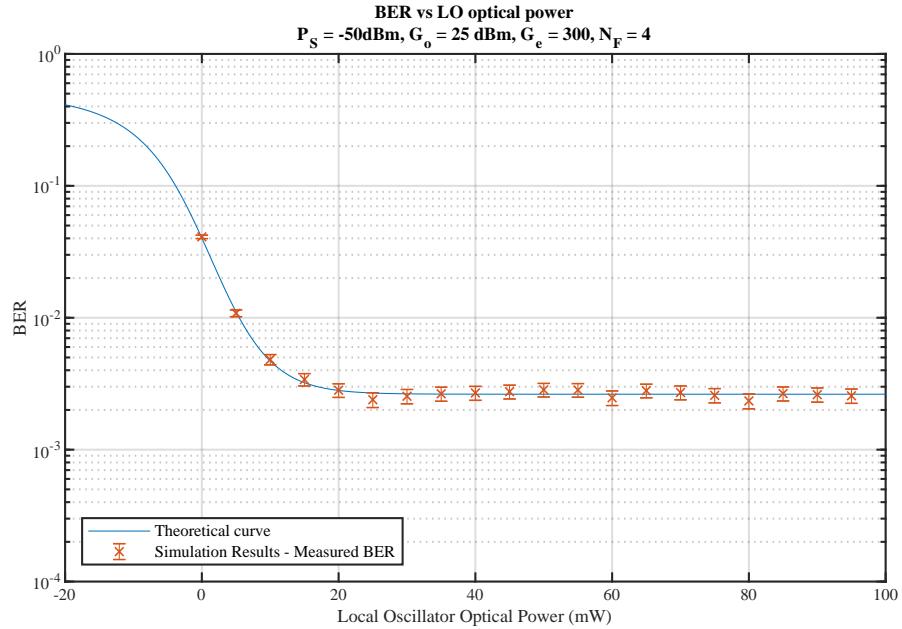


Figure 6.117

At this point it might seem that the benefits from the optical preamplifier are not great when compared with the shot noise limited sensitivity described in the previous sections. To understand why this is not entirely true, let us look at Figure 6.118. This is a two dimensional color coded map of the BER as a function of both the optical gain and local oscillator power.

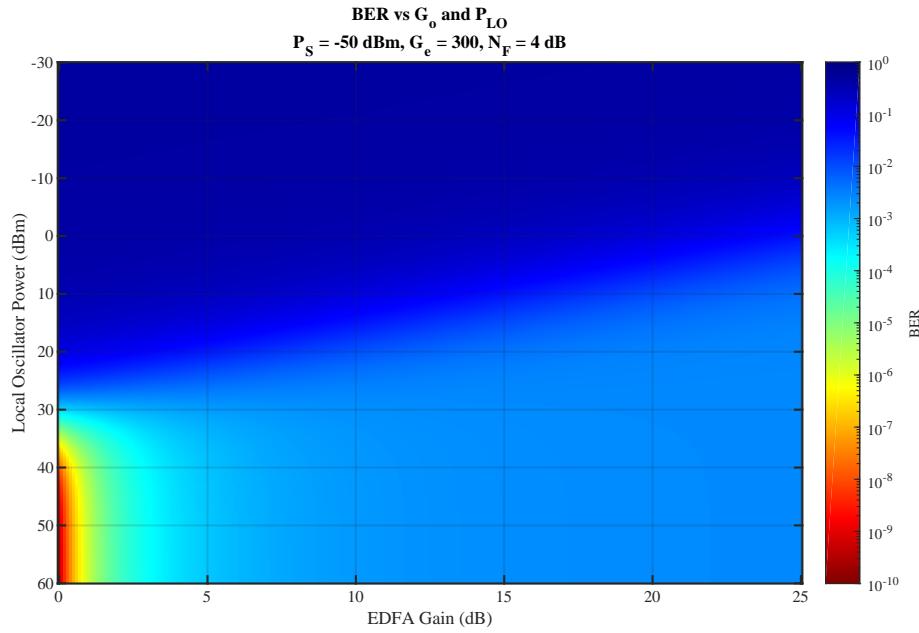


Figure 6.118: BER variation for different values of local oscillator power and optical gain. Theoretical values obtained with equations 6.132

The first thing to notice is the conclusion we had already reached: the BER when optical gain is zero and the LO power is very high ( $> 40$  dBm) is much lower than in any other circumstance. It is possible to reach BER around  $10^{-10}$  in these circumstances. On the other hand, for values of optical gain higher than 6 dB the BER does not go lower than around  $10^{-3}$ , independently of the local oscillator power.

The second thing to notice is that the local oscillator power required to reach a BER of  $10^{-3}$  diminishes with the increased optical gain. Without the preamplifier, the required local oscillator power for this would be around 30 dBm (1 W). Using a preamplifier with an optical gain of 20 dB, this value is lowered to 10 dBm, a much more reasonable value, and easier to work with in practice.

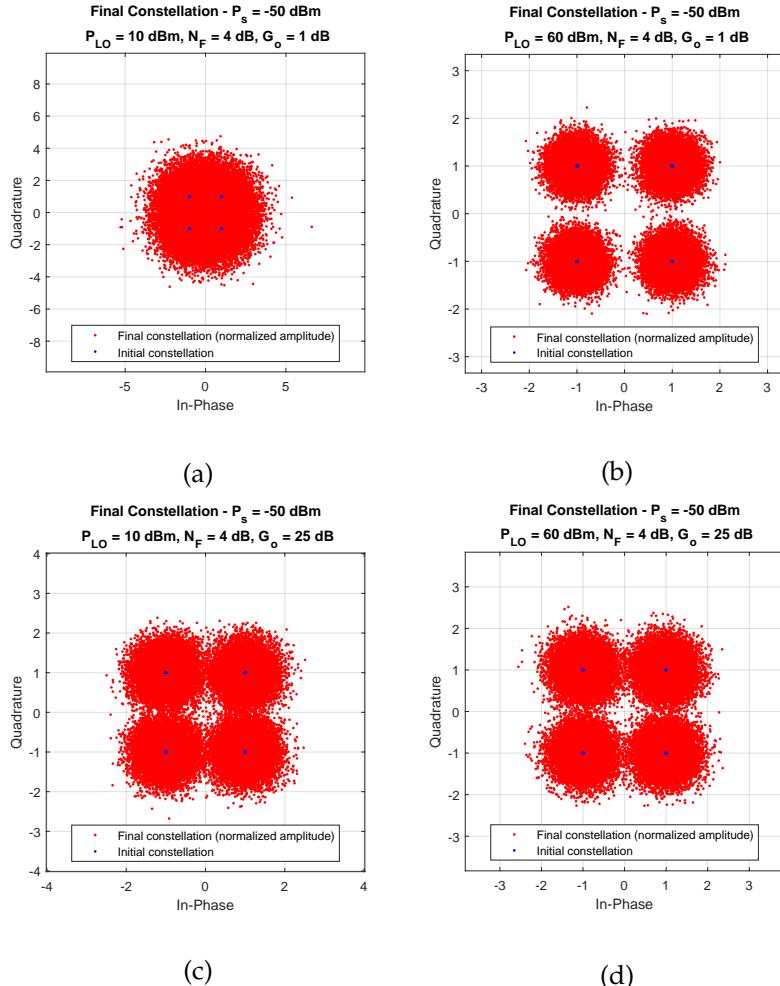


Figure 6.119: Final constellations for various local oscillator optical powers and optical amplifier gains. Signal optical power at receiver input is -50 dBm. Notice that the bottom two constellations are nearly equal, despite the 50 dB difference in local oscillator power.

We can compare these results with those from the previous cases. Immediately it can be seen that the sensitivity is the second best in the series, with only the shot noise limited curve having a lower sensitivity.

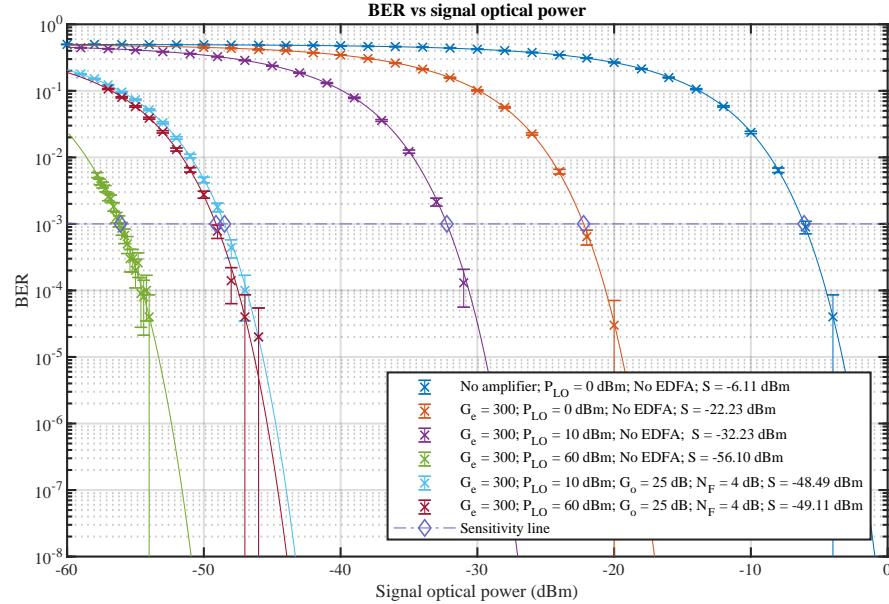


Figure 6.120: Comparison of BER curves of the various examples presented here.

Using an optical amplifier can compensate power losses in the signal path, but always ends up establishing a performance threshold above the shot noise limit. However, even if the performance is slightly worse by a few dB, the lower requirements for the local oscillator is often worth it.

### 6.9.4 Experimental Setup

The setup shown in Figure 6.121 was used to obtain experimental results to compare with the theory and validate the simulation. The list of devices used for the setup is available in Table 6.33. Tables 6.34 to 6.38 show the devices' specifications.

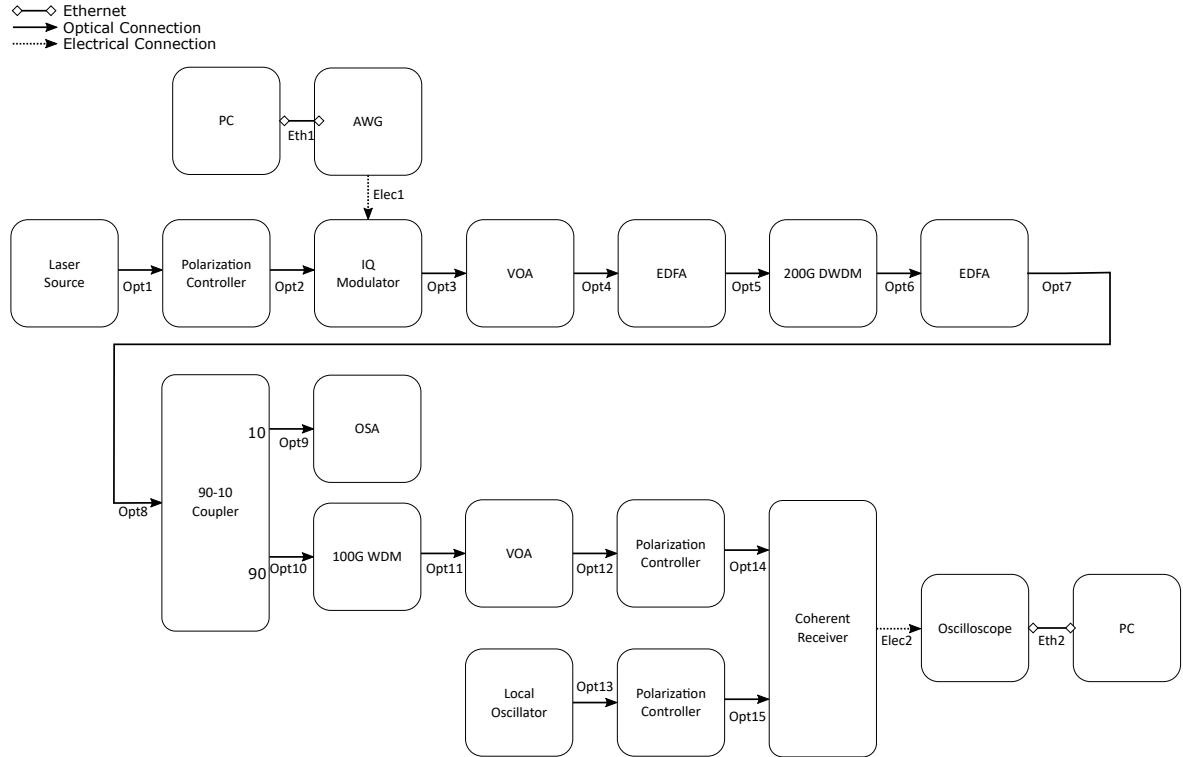


Figure 6.121: Experimental setup

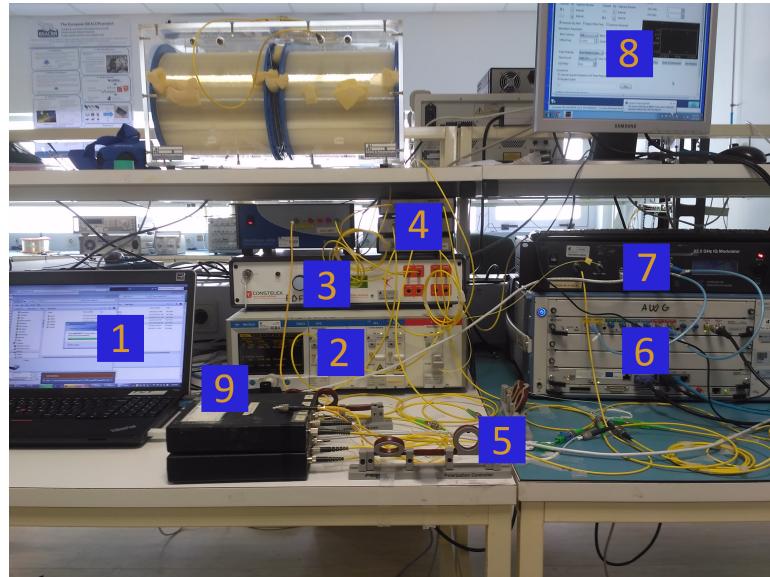


Figure 6.122: Photo of the experimental setup. 1-Computer; 2-TX Laser source; 3-EDFA; 4-Couplers and filters; 5-Polarization controllers; 6-AWG; 7-IQ Modulator; 8-AWG monitor; 9-USB controlled variable optical attenuator.

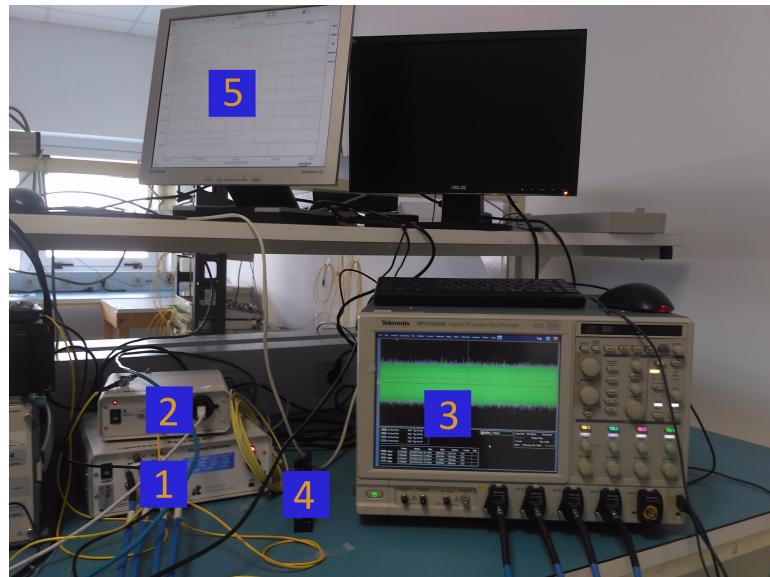


Figure 6.123: Photo of the experimental setup. 1-Coherent Receiver; 2-Local Oscillator laser source; 3-Oscilloscope; 4-Polarization controller; 5-OSA monitor.

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

Table 6.33: Devices in experimental setup.

Tables 6.34 to 6.38 list the relevant specifications for the devices used in the experimental setup.

#### 6.9.4.1 Device Specifications

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

Table 6.34

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

Table 6.35

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

Table 6.36

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

Table 6.37

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

Table 6.38

| <b>Keysight M8195A AWG</b> |  |
|----------------------------|--|
| Sample Rate                | 65 GSa/s                                 |
| Analog Bandwidth (typ.)    | 25 GHz                                   |
| Vertical Resolution        | 8 bits                                   |
| Amplitude (single ended)   | 75 mV <sub>PP</sub> to 1 V <sub>PP</sub> |
| Amplitude Resolution       | 200 $\mu$ V                              |
| Intrinsic Random Jitter    | < 200 fs                                 |
| Rise time (typical)        |  |
| 20% to 80%                 | 18 ps                                    |

Table 6.39

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

Table 6.40

#### 6.9.4.2 Considerations on signal quality and noise

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

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

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

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

It can be seen that there are more differences between the existing simulation and the lab data. By watching Figures 6.124 some of those differences can be seen. It was already mentioned that the noise is not white. A difference not yet mentioned is on the signals themselves. As can be seen, the signal from the lab, shaped with a root raised cosine filter with a roll-off of 0.05, is very well defined and has nearly constant spectral density in the spectrum. In the experimental case, however, the signal shape in the spectrum appears distorted.

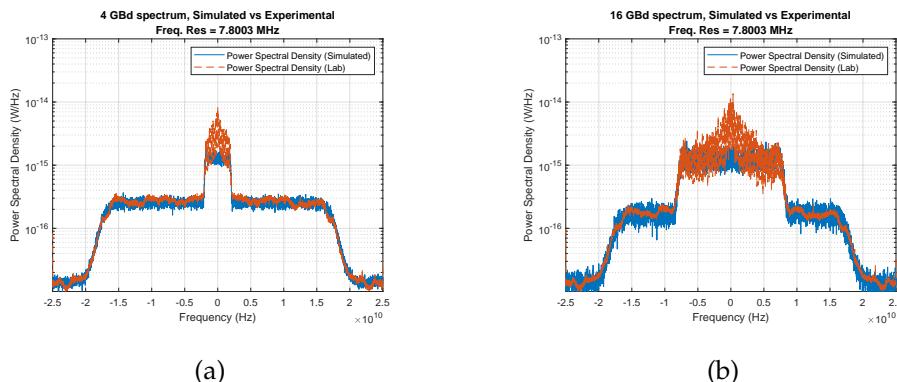


Figure 6.124: Comparison of spectra from simulated and experimental signals. 4 GBd signals on the left and 16 GBd on the right. The simulated and experimental signals produce the same BER.

Figure 6.125 shows the power spectrum of noise only signals, and their simulated counterparts, using the current simulation structure (optical white noise, electrical filter and

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

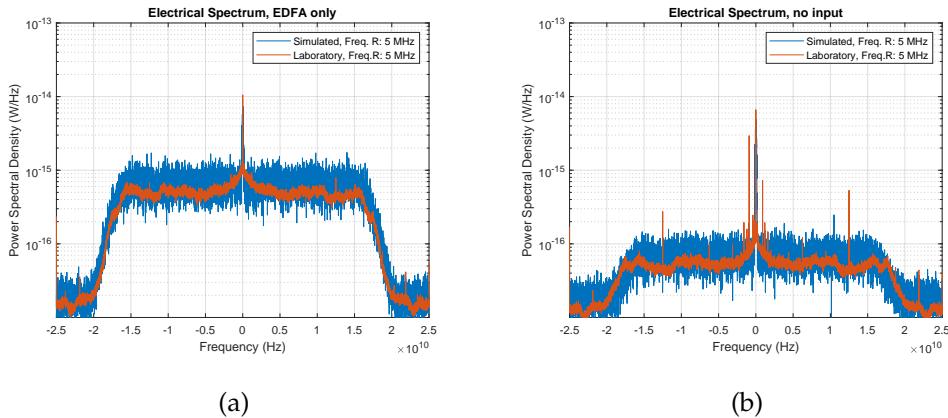


Figure 6.125: Power spectrum of the signal obtained when (a) the input signal is non-existing, and the local oscillator is turned off; (b) the input signal is made only of noise generated by the EDFA.

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

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

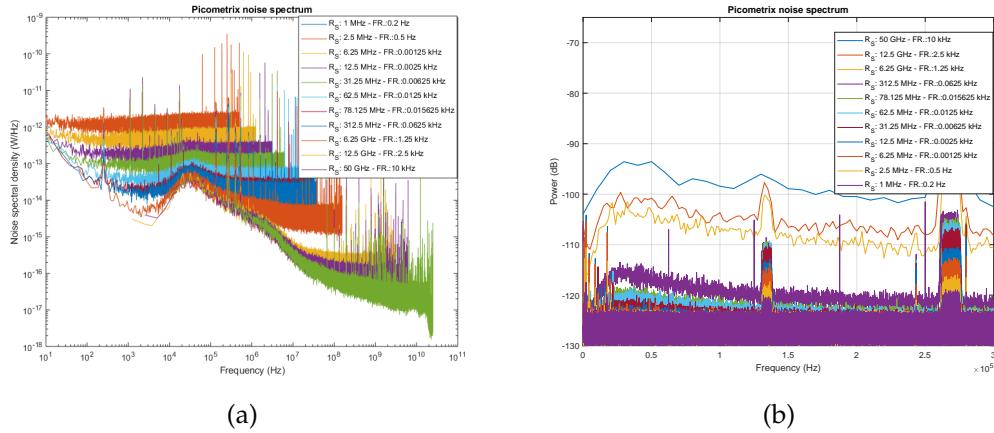


Figure 6.126: Noise spectra at various sample rates (a) Logarithmic plot, normalized to frequency resolution (b) Linear frequency scale, not normalized.

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

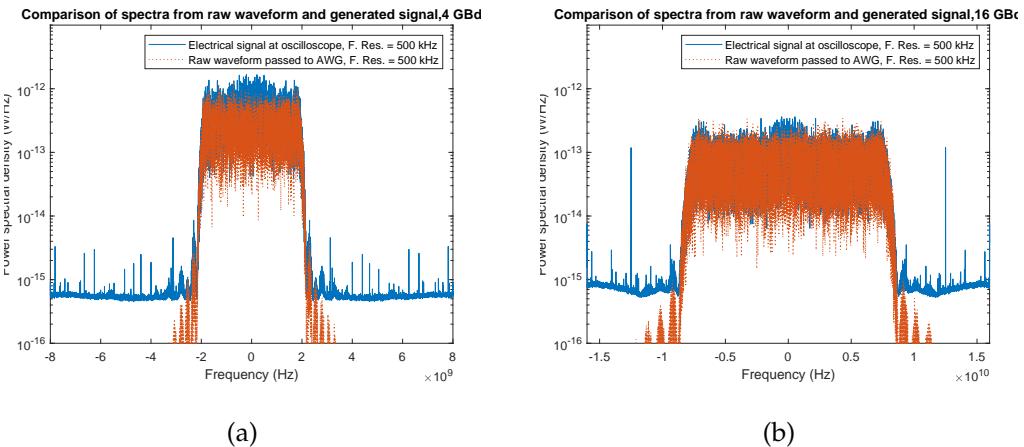


Figure 6.127: Spectra of electrical signals generated by the AWG compared to the original waveforms

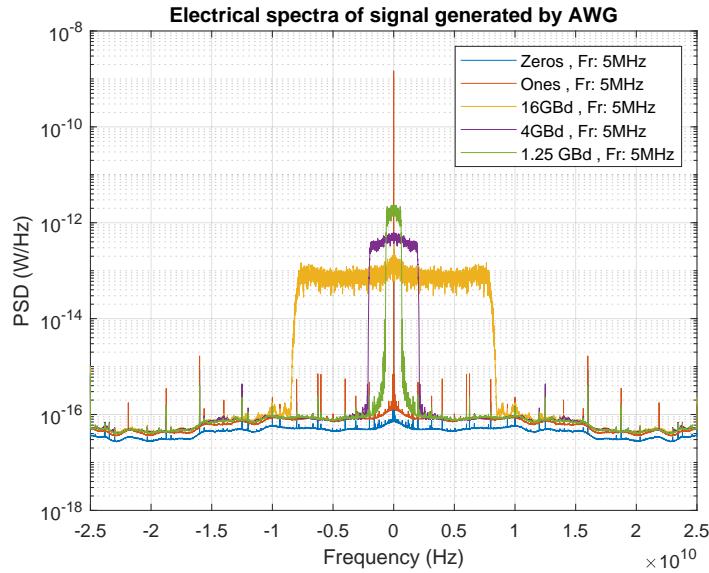


Figure 6.128: Spectra of the electrical signals generated by the AWG.

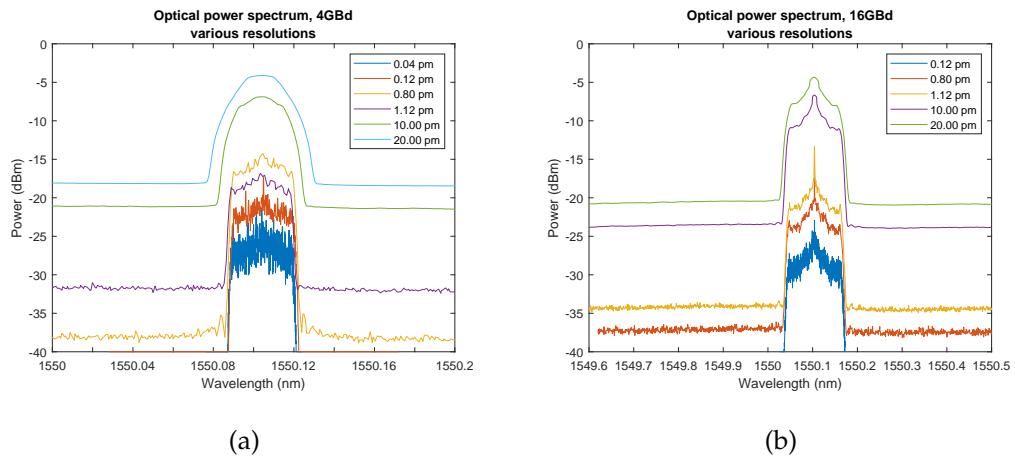


Figure 6.129: Optical spectra of the modulated signals at different resolutions.

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

#### 6.9.4.3 Considerations about parameters and results

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

simulation and the real world. In the following descriptions, keep in mind that the overall architecture of the setup is described in Figure 6.121, and the specifications of each particular device are in the tables following the figure. With this in mind, the possible origins for the values to use are:

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

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

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

It is worth noting that, when using the "Optical  $E_b/n_0$ ", it does not encompass all noise sources in the signal. This value only considers the optical noise generated by the EDFA, ignoring all electrical noise sources. Also, it is important to remember that this analysis considers that all nonlinearities in the signal can be neglected or compensated by DSP (a reasonable assumption used in [11]). It is also assumed that the receiver is perfectly balanced. While this is a bit more unrealistic, it is the case used in the simulation.

### 6.9.5 Homodyne Detection

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

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

Figure 6.130 shows the BER curves obtained for 16, 4 and 1.25 GBd signals. The theoretical curves were obtained resorting to equation 6.115.

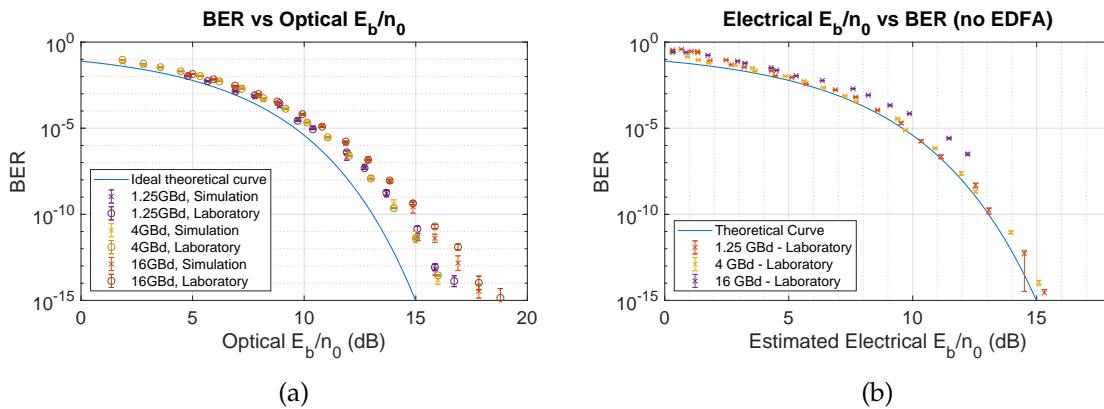


Figure 6.130: Comparison of theoretical BER curves against obtained results. Optical  $E_b/n_0$  calculated through the optical spectrum, electrical  $E_b/n_0$  estimated from the raw waveforms obtained at the oscilloscope, through spectral analysis of the acquired signal. Locally generated local oscillator .

The  $E_b/n_0$  used to plot the experimental data points in Figure 6.130 is estimated offline. Some further comments on the results follow in the sections of the respective symbol rate.

#### 6.9.5.1 16 GBd Signal

Figures 6.131 to 6.135 show the eye diagrams and the spectrum of the signals at every stage of the DSP process. The process is similar to the described for the intradyne configuration, but without phase or frequency corrections.

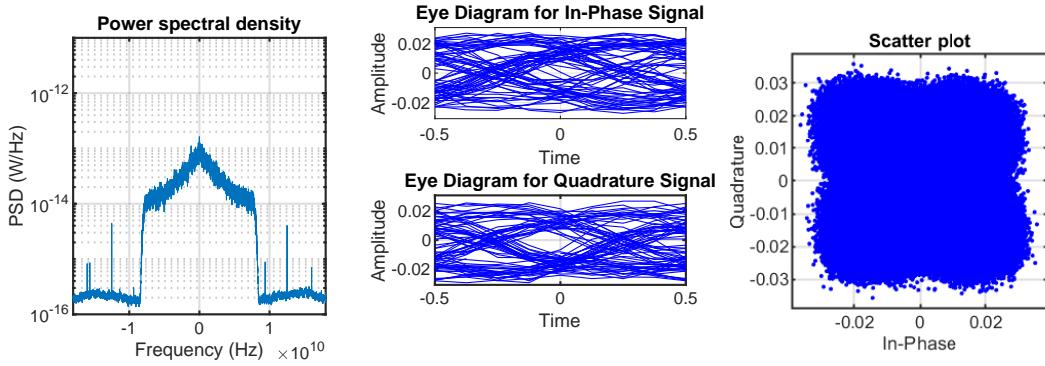


Figure 6.131: Initial spectrum, eye Diagram, and constellation of the original signal obtained at the oscilloscope. Input signal power immediately before the coherent receiver was -16.6 dBm.

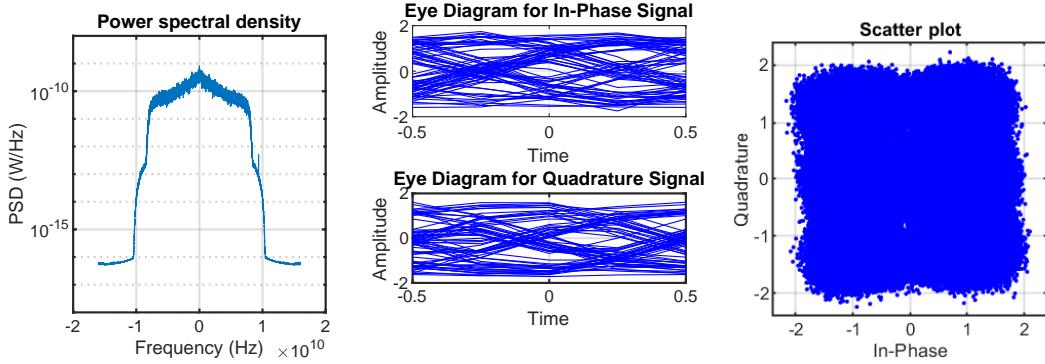


Figure 6.132: Spectrum, eye-diagram and constellation after applying front-end corrections, matched filtering and resampling to  $2S_R$ .

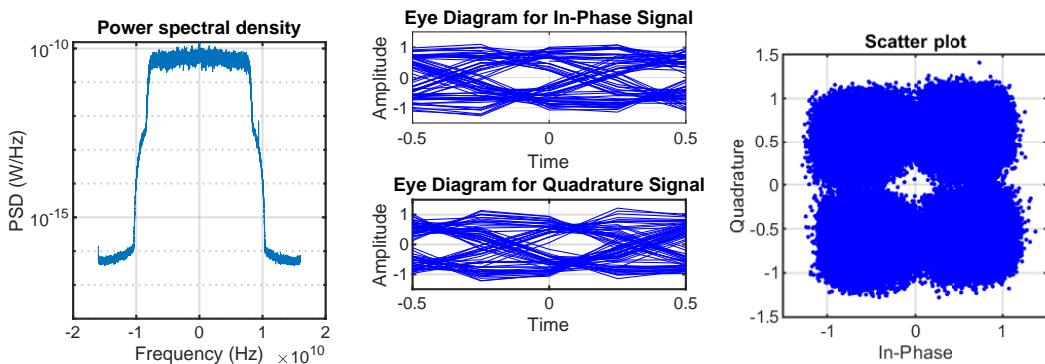


Figure 6.133: Spectrum, eye diagram and constellation after using a constant modulus algorithm.

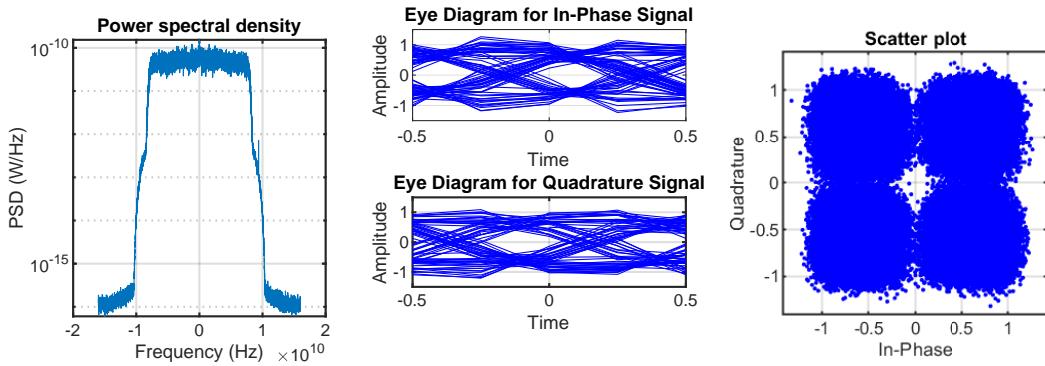


Figure 6.134: Spectrum, eye diagram and constellation after carrier-phase compensation.

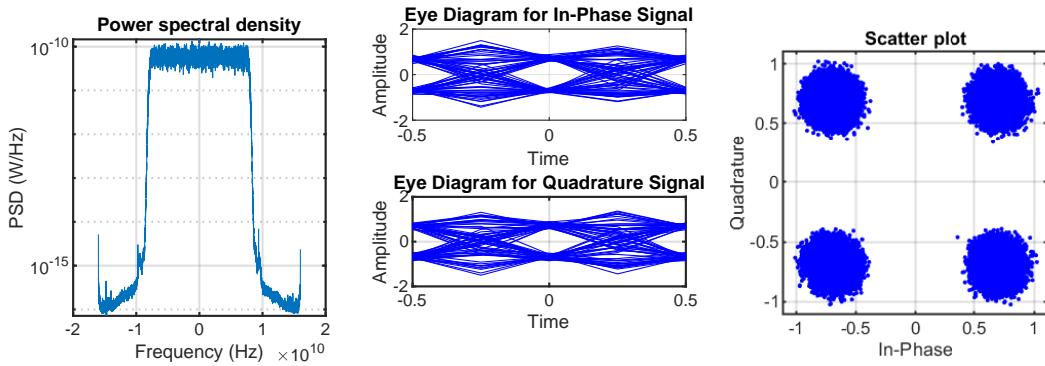


Figure 6.135: Spectrum, eye diagram and constellation after an adaptive equalizer.

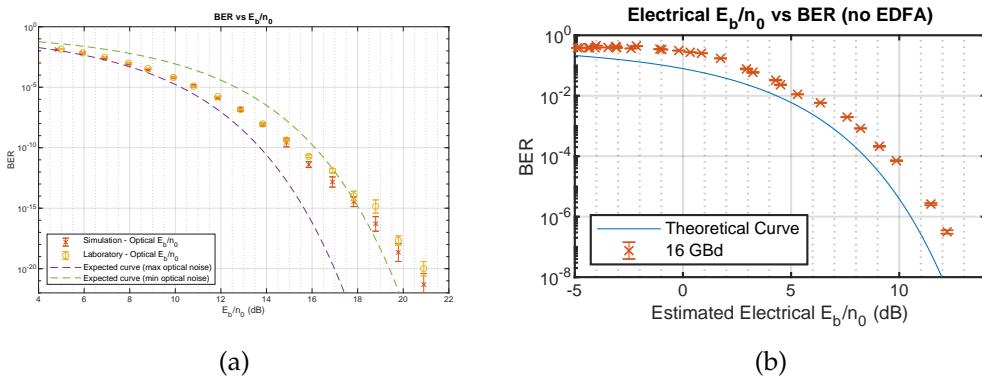


Figure 6.136: Comparison of expected BER, results from the simulation and from the lab setup, plotted against the  $E_b/n_0$  measured in the optical domain, as described in Sections ?? and 6.9.4.3.

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

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

#### 6.9.5.2 4 GBd Signal

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

Figures 6.137 to 6.141 show the eye diagrams and the spectrum of the signals at every stage of the DSP process. The process is similar to the described for the 16 GBd signal with homodyne receiver. However, some differences can be seen, particularly on the signal spectrum before the corrections. However, it can be seen that the distribution of points in the final constellation is not optimal, unlike in the 16 GBd case.

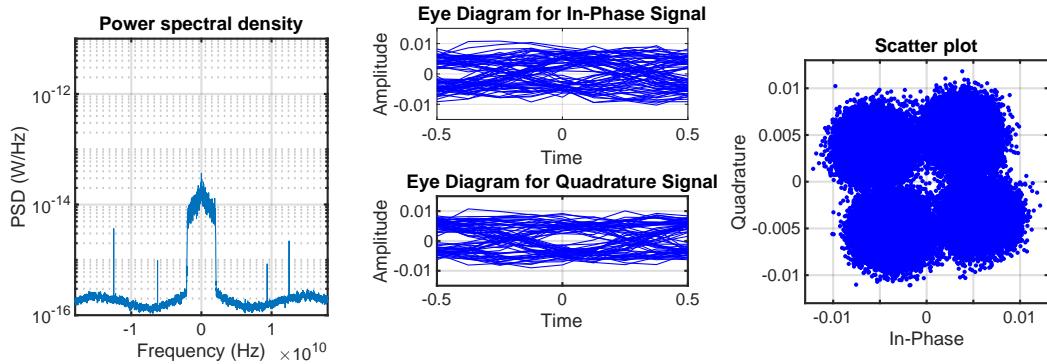


Figure 6.137: Initial spectrum, eye Diagram, and constellation of the original signal obtained at the oscilloscope. Input signal power immediately before the coherent receiver was -28.4 dBm.

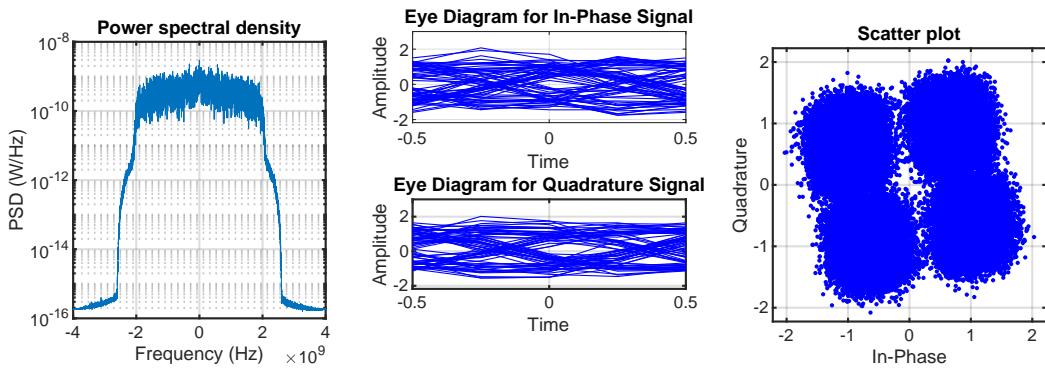


Figure 6.138: Spectrum, eye-diagram and constellation after applying front-end corrections, matched filtering and resampling to  $2S_R$ .

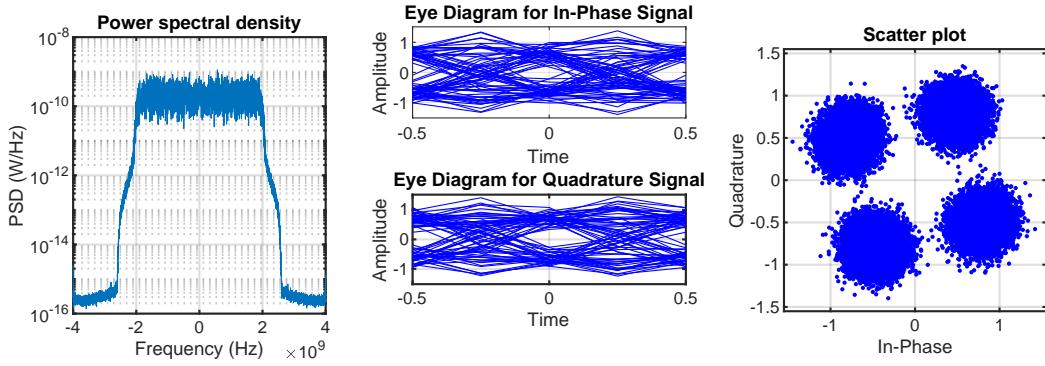


Figure 6.139: Spectrum, eye diagram and constellation after using a contant modulus algorithm.

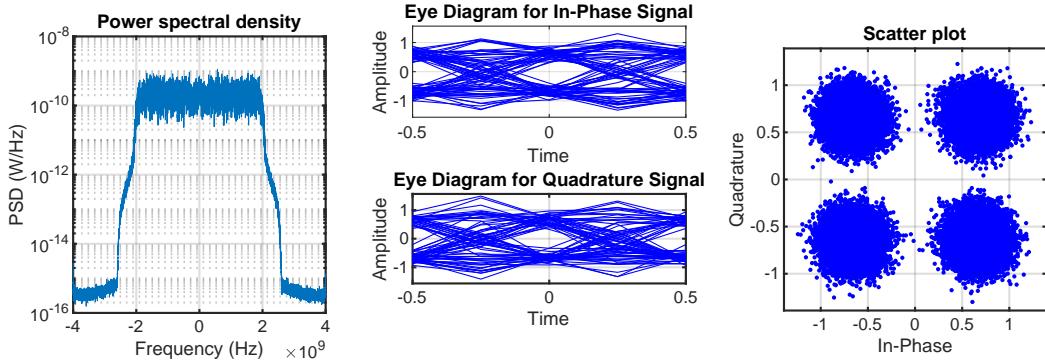


Figure 6.140: Spectrum, eye diagram and constellation after carrier-phase compensation.

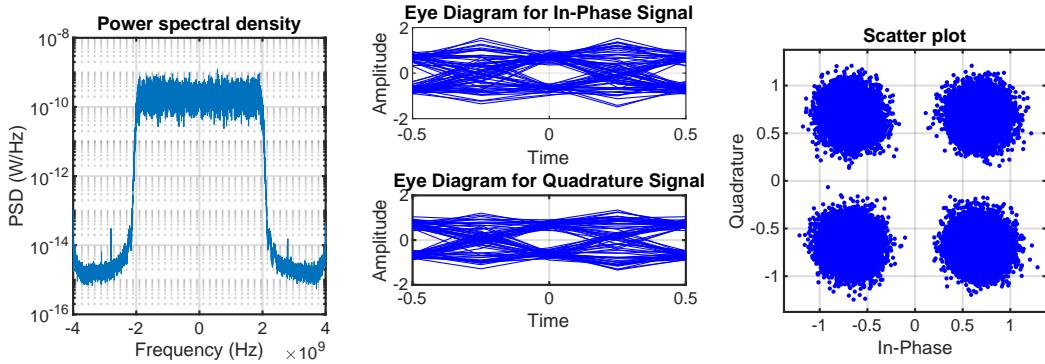


Figure 6.141: Spectrum, eye diagram and constellation after an adaptive equalizer.

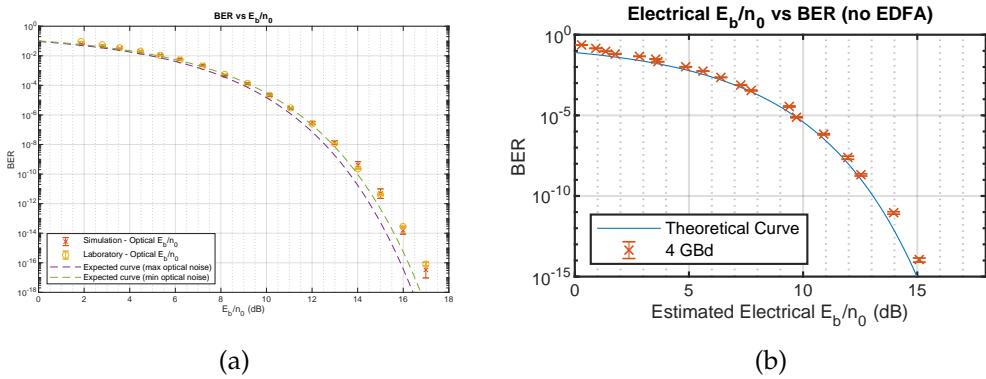


Figure 6.142: Comparison of expected BER, results from the simulation and from the lab setup, plotted against the  $E_b/n_0$  measured in the optical domain, as described in Sections ?? and 6.9.4.3.

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

### 6.9.5.3 1.25 GBd signal

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

Figures 6.143 to 6.147 show the eye diagrams and the spectrum of the signals at every stage of the DSP process. The process is similar to the described for the 16 GBd signal with homodyne receiver. However, some differences can be seen, particularly on the signal spectrum before the corrections. However, it can be seen that the distribution of points in the final constellation is not optimal, unlike in the 16 GBd case.

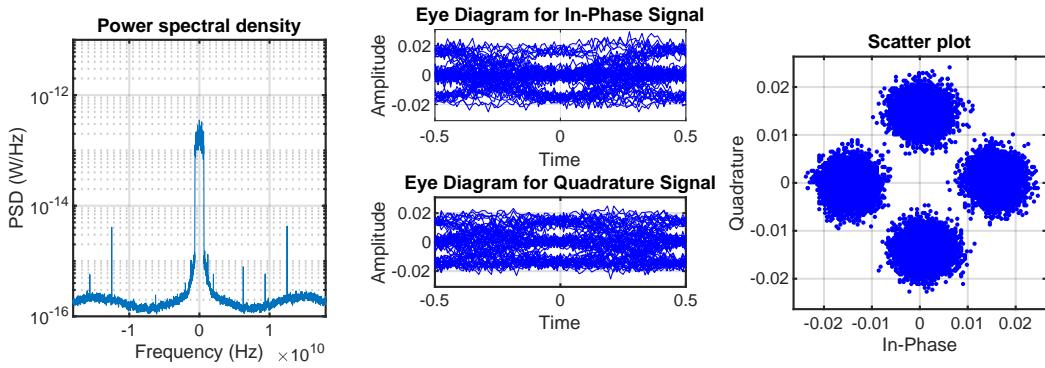


Figure 6.143: Initial spectrum, eye Diagram, and constellation of the original signal obtained at the oscilloscope. Input signal power immediately before the coherent receiver was -22.2 dBm.

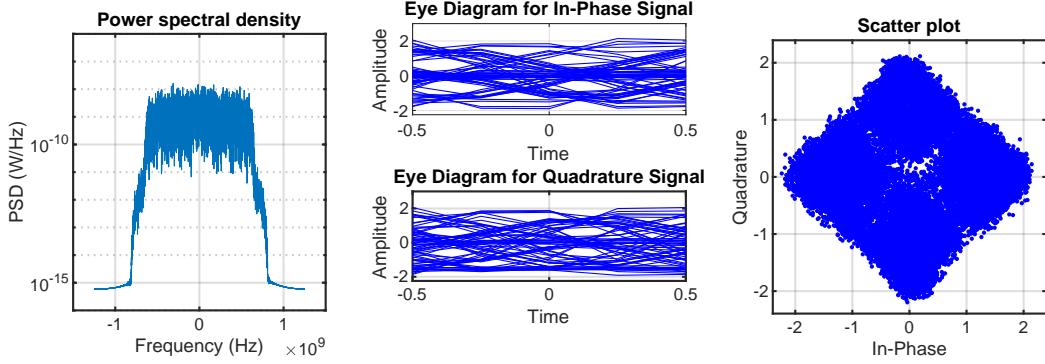


Figure 6.144: Spectrum, eye-diagram and constellation after applying front-end corrections, matched filtering and resampling to  $2S_R$ .

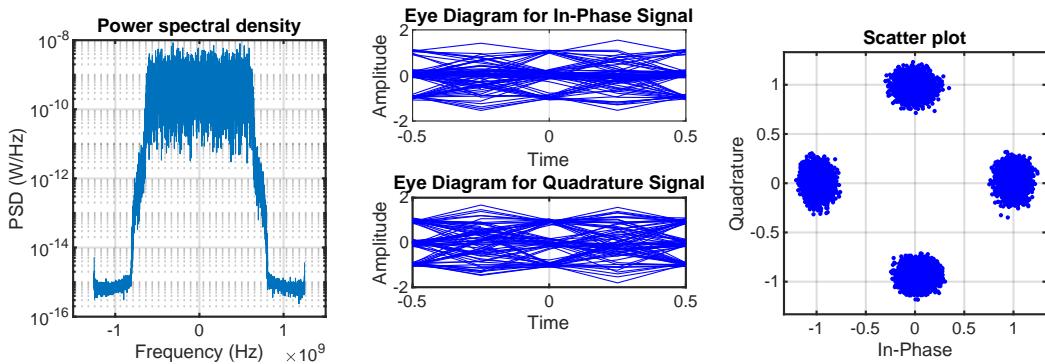


Figure 6.145: Spectrum, eye diagram and constellation after using a constant modulus algorithm.

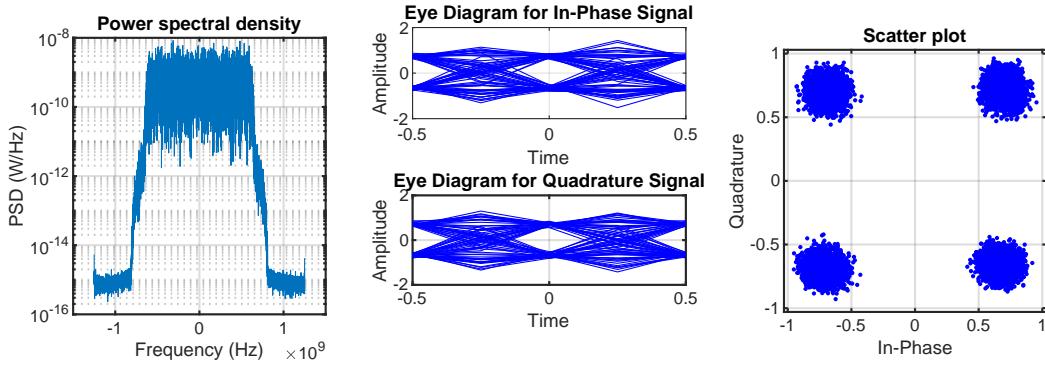


Figure 6.146: Spectrum, eye diagram and constellation after carrier-phase compensation.

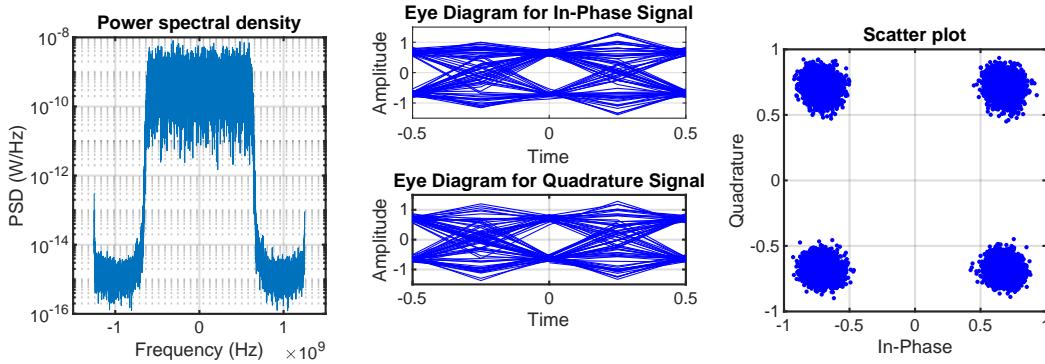


Figure 6.147: Spectrum, eye diagram and final constellation after an adaptive equalizer. The BER estimated from the Q-factor is  $9.2598 \times 10^{-29}$

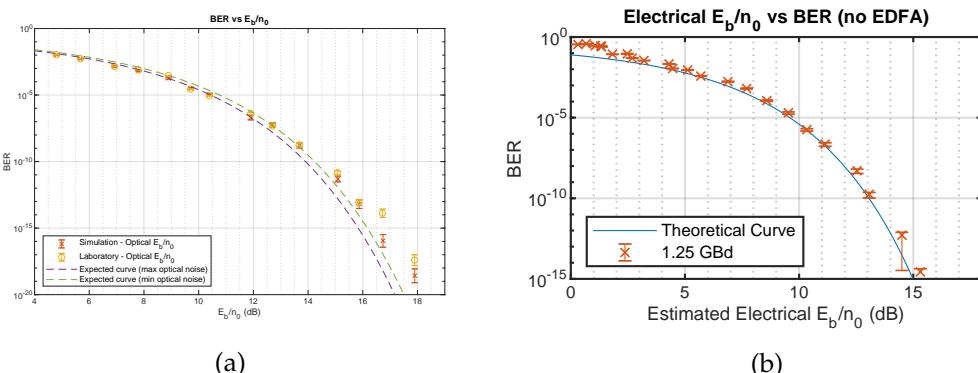


Figure 6.148: Comparison of expected BER, results from the simulation and from the lab setup, plotted against the  $E_b/n_0$  measured in the optical domain, as described in Sections ?? and 6.9.4.3.

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

good agreement with the experimental results.

### 6.9.6 Digital Signal Post-Processing

The stages of the DSP process applied to the experimental results of this setup are presented in Figure 6.149. After conversion to digital and combination in a complex vector, the signal

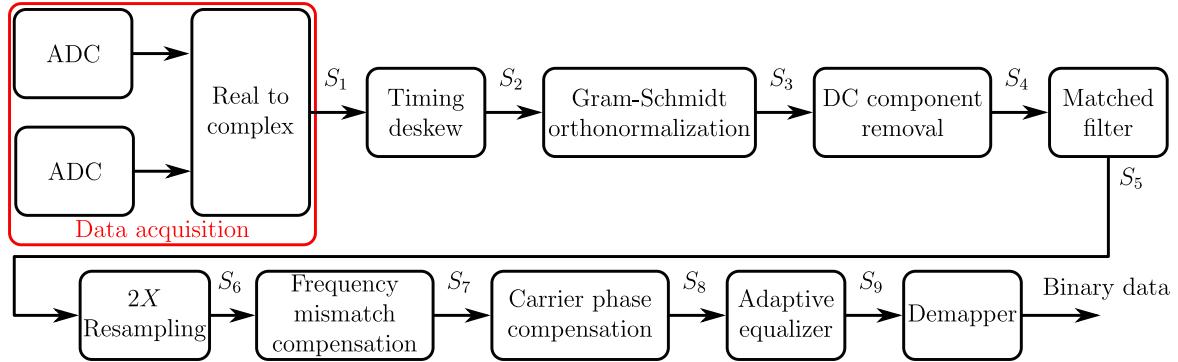


Figure 6.149: Block diagram of the DSP applied offline in the netxpto environment.

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

The constellation on which the DSP will be implemented is presented in Figure 6.150. This data was obtained from the .mat file **SNR=38\_trial8.mat**, included in the folder **\sdf\m\_qam\_system\_dsp\Data**. A short matlab code present in same folder, named **mat2txt.m**, reads the **.mat** file and outputs the received signal to a text file named **Input.txt**. This **.txt** file can be read into the netxpto environment by using the *LoadAscii* block, the code necessary for this is included below.

```

TimeContinuousAmplitudeContinuousComplex SIn{ "SIn.sgn" };

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

```

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

```

TimeContinuousAmplitudeContinuousComplex S1{ "S1.sgn" };

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

```

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

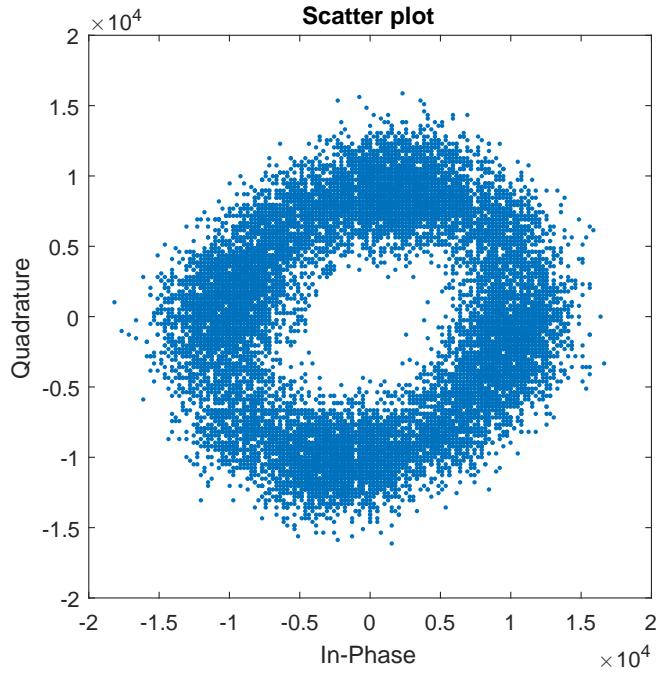


Figure 6.150: Constellation obtained at the oscilloscope from a signal with a baud rate of 1.25 Gbd and a SNR of 38 dB.

#### 6.9.6.1 Timing deskew

This DSP step takes a complex input signal and removes a given amount of timing skew between its real (in-phase) and imaginary (quadrature) parts. This DSP step follows the topology presented in Figure 6.151. The input signal  $S_{\text{in}}(t_n)$  is separated into its in-phase

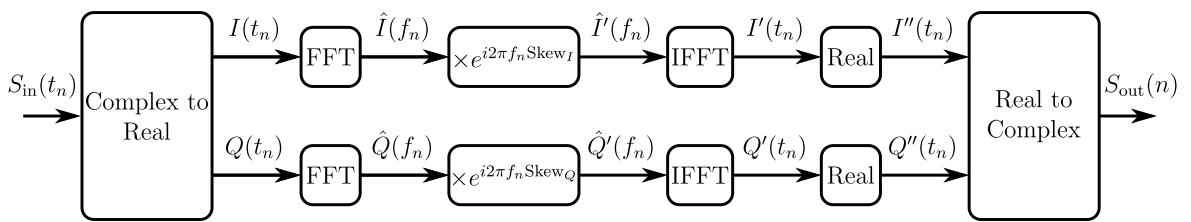


Figure 6.151: Block diagram representation of the deskew procedure.

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

$$\hat{I}'(f_n) = \hat{I}(f_n)e^{i2\pi f_n \text{Skew}_I}, \quad (6.134)$$

$$\hat{Q}'(f_n) = \hat{Q}(f_n)e^{i2\pi f_n \text{Skew}_Q}. \quad (6.135)$$

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

signal

$$S_{\text{out}}(t_n) = \text{real}(I'(t_n)) + i\text{real}(Q'(t_n)). \quad (6.136)$$

The effect of this DSP step is presented in Figure 6.152.

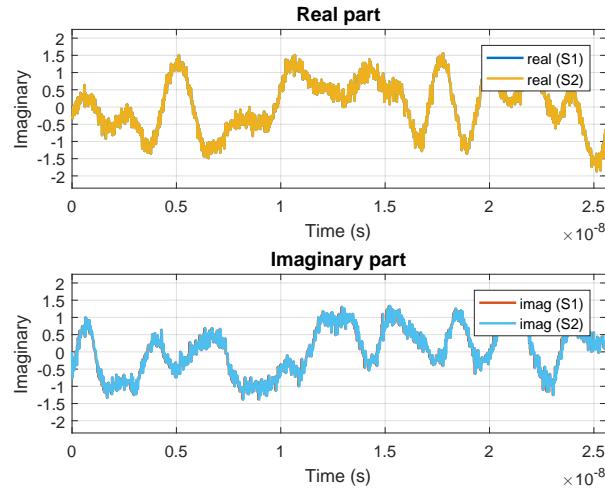


Figure 6.152: Signals before and after deskew DSP step.

#### 6.9.6.2 Gram-Schmidt orthonormalization

This step orthonormalizes the data by implementing a Gram-Schmidt algorithm. This implementation follows the topology presented in Figure 6.153. The input signal  $S_{\text{in}}(t_n)$  is

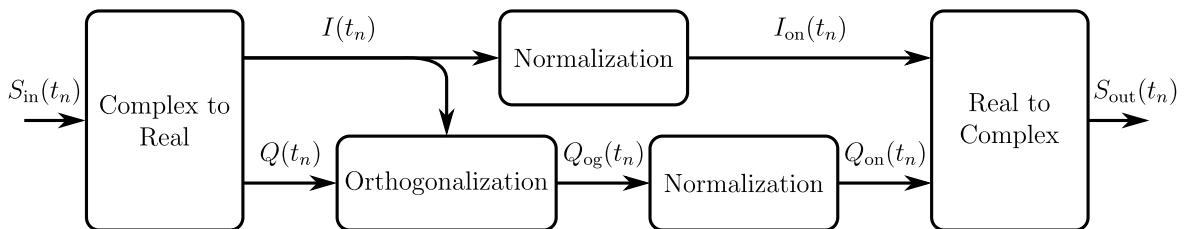


Figure 6.153: Block diagram representation of the Gram-Schmidt orthonormalization procedure.

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.:

$$P_I = \text{mean}(I^2(t_n)), \quad (6.137)$$

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

$$I_{\text{on}}(t_n) = \frac{I(t_n)}{\sqrt{P_I}}. \quad (6.138)$$

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

$$Q_{\text{og}}(t_n) = Q(t_n) - \frac{I(t_n)\text{mean}(I(t_n)Q(t_n))}{P_I}, \quad (6.139)$$

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

$$P_Q = \text{mean}(Q_{\text{og}}^2(t_n)), \quad (6.140)$$

$$Q_{\text{on}}(t_n) = \frac{Q_{\text{og}}(t_n)}{\sqrt{P_Q}}. \quad (6.141)$$

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

$$S_{\text{out}}(t_n) = I_{\text{on}}(t_n) + iQ_{\text{on}}(t_n) \quad (6.142)$$

The effect of this DSP step is presented in Figure 6.154.

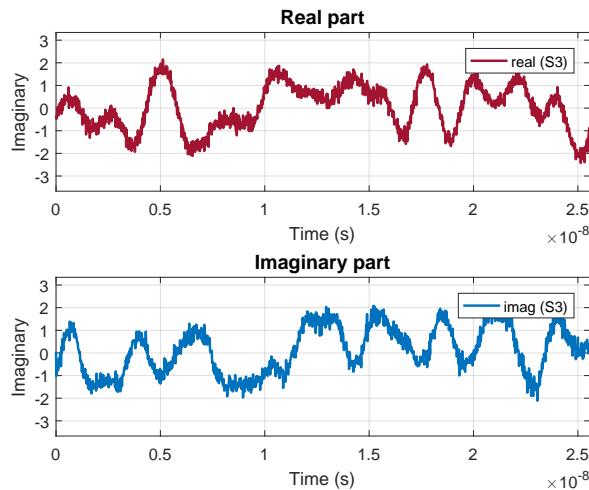


Figure 6.154: Signal after orthonormalization DSP step.

#### 6.9.6.3 DC component removal

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

$$S_{\text{out}}(t_n) = S_{\text{in}}(t_n) - \text{mean}(S_{\text{in}}(t_n)) \quad (6.143)$$

The effect of this DSP step is presented in Figure 6.155.

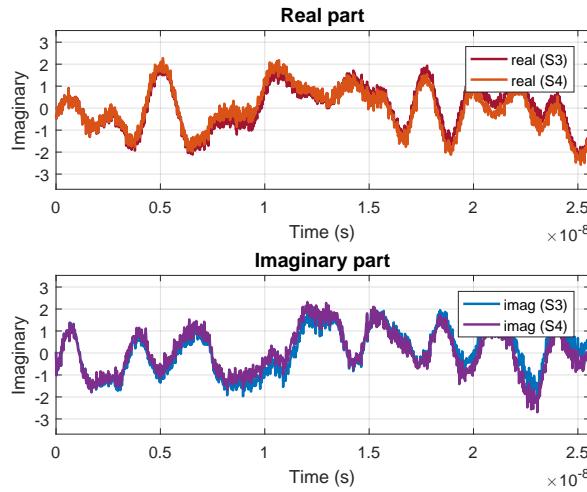


Figure 6.155: Signals before and after DC component removal DSP step.

#### 6.9.6.4 Matched filtering

This step implements some form of matched filtering. For the modulation of the signal corresponding to the constellation in Figure 6.150, the matched filter consists of a root raised cosine function. The filter can be applied either in time-domain, through convolution of the signal with the filtering function, or in frequency domain, through a simple multiplication.

The effect of this DSP step is presented in Figure 6.156.

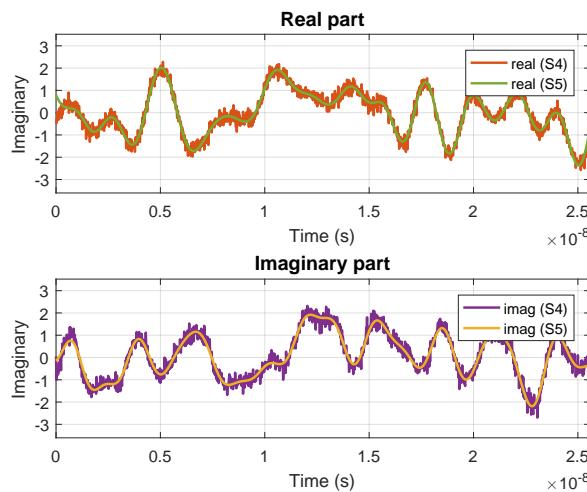


Figure 6.156: Signals before and after matched filtering.

### 6.9.6.5 $2\times$ resampling

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

### 6.9.6.6 Frequency mismatch compensation

This step compensates for the frequency mismatch between the local oscillators employed at the transmission and reception stages. Multiple methods for this are available, the one employed for the results presented here is the spectral method, a topological representation of which is presented in Figure 6.157. The input signal, which can be described as

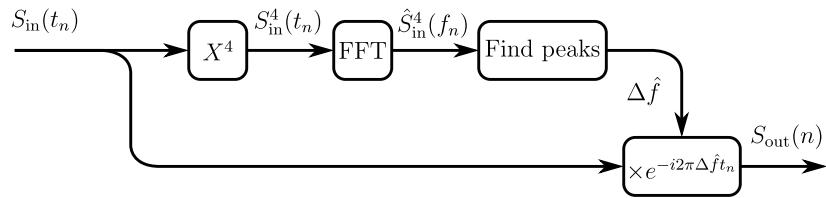


Figure 6.157: Block diagram representation of the spectral frequency mismatch estimation and compensation procedure.

$$S_{\text{in}}(t_n) = |S_{\text{in}}(t_n)|e^{i(2\pi\Delta f t_n + \theta(t_n) + \Delta\phi)}, \quad (6.144)$$

is powered by 4, yielding

$$S_{\text{in}}^4(t_n) = |S_{\text{in}}(t_n)|e^{i(2\pi 4\Delta f t_n + 4\theta(t_n) + 4\Delta\phi)}, \quad (6.145)$$

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

$$S_{\text{in}}^4(t_n) = |S_{\text{in}}(t_n)|e^{i(2\pi 4\Delta f t_n + \pi + 4\Delta\phi)}, \quad (6.146)$$

so this process effectively removes the QPSK modulation of the signal. The  $S_{\text{in}}^4(t_n)$  signal is then converted to the frequency domain, because of the removal of the phase modulation, its spectrum is expected to display a maximum at a frequency of  $4\Delta f$ , which is determined by some *find peaks* function, yielding the estimate  $\Deltâf$ . The input signal  $S_{\text{in}}$  is then multiplied by  $e^{-i2πΔ̂f t_n}$ . Assuming the estimate is a good one, this process will remove the effect of the frequency mismatch, yielding the following output signal

$$S_{\text{out}}(t_n) = S_{\text{in}}(t_n)e^{-i2πΔ̂f t_n} = |S_{\text{in}}(t_n)|e^{i(\theta(t_n) + \Delta\phi)} \quad (6.147)$$

### 6.9.6.7 Carrier phase compensation

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

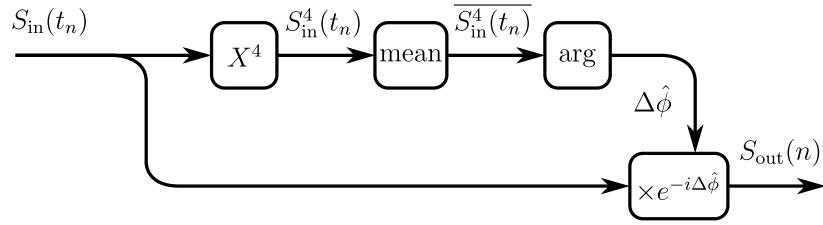


Figure 6.158: Block diagram representation of the blind phase mismatch estimation and compensation procedure.

of which is presented in Figure 6.158. The input signal, which at this stage can be described as

$$S_{\text{in}}(t_n) = |S_{\text{in}}(t_n)|e^{i(\theta(t_n)+\Delta\phi)}, \quad (6.148)$$

is powered by 4, yielding

$$S_{\text{in}}^4(t_n) = |S_{\text{in}}(t_n)|e^{i(4\theta(t_n)+4\Delta\phi(t_n))}, \quad (6.149)$$

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

$$S_{\text{in}}^4(t_n) = |S_{\text{in}}(t_n)|e^{i(\pi+4\Delta\phi(t_n))}, \quad (6.150)$$

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

$$S_{\text{out}}(t_n) = S_{\text{in}}(t_n)e^{-i\Delta\hat{\phi}} = |S_{\text{in}}(t_n)|e^{i\theta(t_n)} \quad (6.151)$$

#### 6.9.6.8 Adaptive equalizer

### 6.9.7 Open Issues

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

### 6.9.8 Future work

Extend this block to include other values of M.

## References

- [1] John G. Proakis and Masoud Salehi. *Digital Communications*. 5ed. McGraw-Hill, 2008. ISBN: 9780072957167.
- [2] A Bruce Carlson. *Communication Systems: An Introducton to Signals and Noise in Electrical Communication*. McGraw Hill, 1986.
- [3] Tom M Apostol. "Calculus, Vol. 1: One-Variable Calculus, with an Introduction to Linear Algebra". In: *Waltham, MA: Blaisdell* (1967).
- [4] Nay Oo and Woon-Seng Gan. "On harmonic addition theorem". In: *International Journal of Computer and Communication Engineering* 1.3 (2012), p. 200.
- [5] Tildon H Glisson. *Introduction to circuit analysis and design*. Springer Science & Business Media, 2011.
- [6] Govind P Agrawal. *Lightwave technology: components and devices*. Vol. 1. John Wiley & Sons, 2004.
- [7] Kazuro Kikuchi. "Fundamentals of coherent optical fiber communications". In: *Journal of Lightwave Technology* 34.1 (2016), pp. 157–179.
- [8] Takaaki Mukai and Yoshihisa Yamamoto. "Gain, frequency bandwidth, and saturation output power of AlGaAs DH laser amplifiers". In: *IEEE Journal of Quantum Electronics* 17.6 (1981), pp. 1028–1034.
- [9] Ha H Nguyen and Ed Shwedyk. *A First Course in Digital Communications*. Cambridge University Press, 2009.
- [10] Mischa Schwartz. *Information Transmission, Modulation and Noise*. McGraw-Hill College, 1990. ISBN: 9780070559097.
- [11] C Crognale. "Sensitivity and power budget of a homodyne coherent DP-QPSK system with optical amplification and electronic compensation". In: *Journal of Lightwave Technology* 32.7 (2014), pp. 1295–1306.
- [12] N Anders Olsson. "Lightwave systems with optical amplifiers". In: *Journal of Lightwave Technology* 7.7 (1989), pp. 1071–1082.
- [13] Govind P Agrawal. *Fiber-optic communication systems*. Vol. 222. John Wiley & Sons, 2012.
- [14] Rongqing Hui and Maurice O'Sullivan. *Fiber optic measurement techniques*. Academic Press, 2009.
- [15] John M Senior and M Yousif Jamro. *Optical fiber communications: principles and practice*. Pearson Education, 2009.

## 6.10 Low Baud M-QAM Transmission System

|                     |   |
|---------------------|---|
| <b>Student Name</b> | Andoni Santos (2018/01/03 - )<br>Ana Luisa Carvalho (2017/04/01 - 2017/12/31)   |
| <b>Goal</b>         | : Low baud rate M-QAM system implementation with BER measurement and comparison with theoretical and experimental values. |
| <b>Directory</b>    | : sdf/low_baud_m_qam_system   |

### 6.10.1 Experimental Results

#### 6.10.1.1 2 GBd Signal

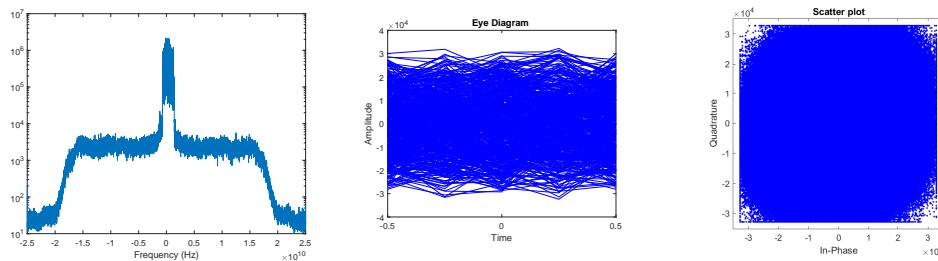


Figure 6.159: Eye Diagram and spectrum of the original signal obtained at the oscilloscope. Initial estimated SNR before any signal processing is 11.1265 dB.

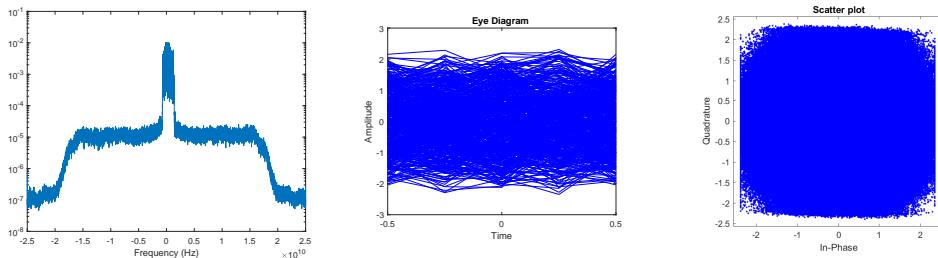


Figure 6.160: Eye Diagram and spectrum after front end corrections, including removal of DC component, deskew and Gram-Schmidt Orthonormalization.

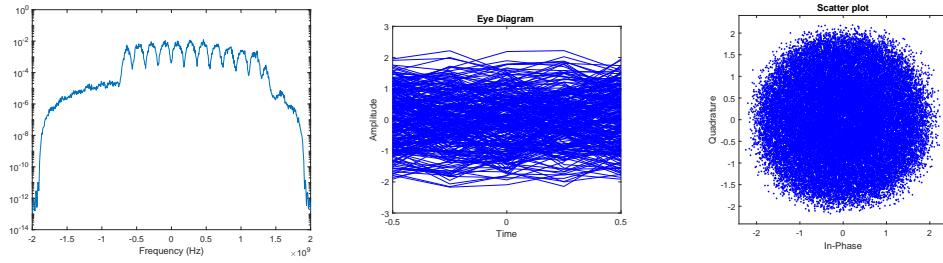


Figure 6.161: Eye Diagram and spectrum after applying the matched filter and downsampling to  $2S_R$ .

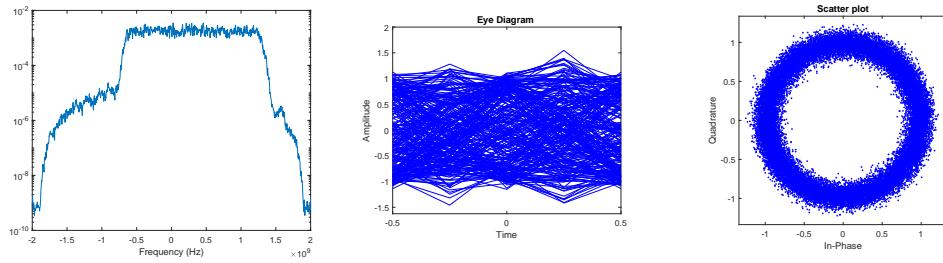


Figure 6.162: Eye Diagram and spectrum after the first stage of adaptive equalization, using a MIMO 2x2 CMA equalizer.

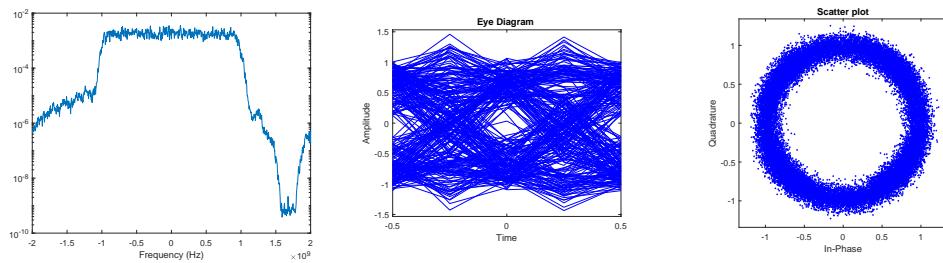


Figure 6.163: Eye Diagram and spectrum after frequency offset correction.

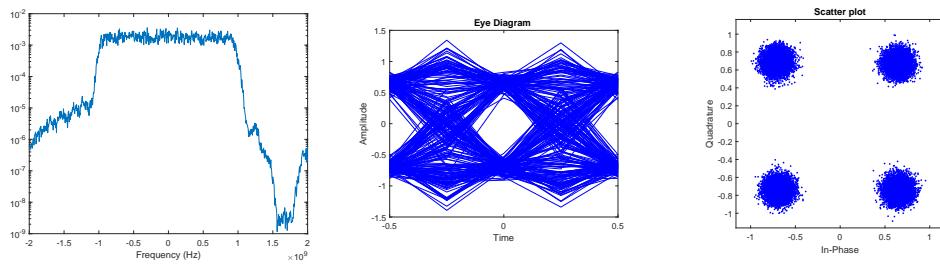


Figure 6.164: Eye Diagram and spectrum after the first stage of carrier-phase estimation.

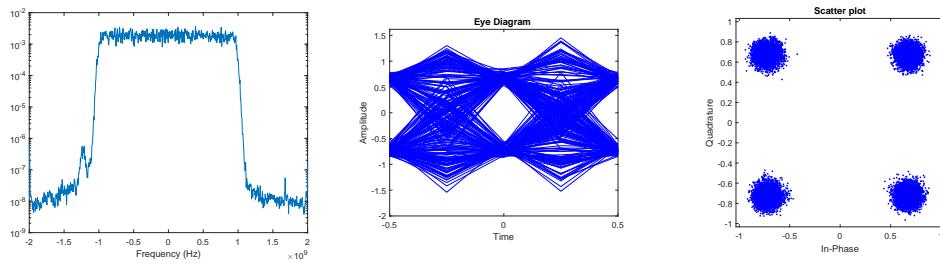


Figure 6.165: Eye Diagram and spectrum after the second stage of adaptive equalization, with a MIMO 4x4 LMS equalizer.

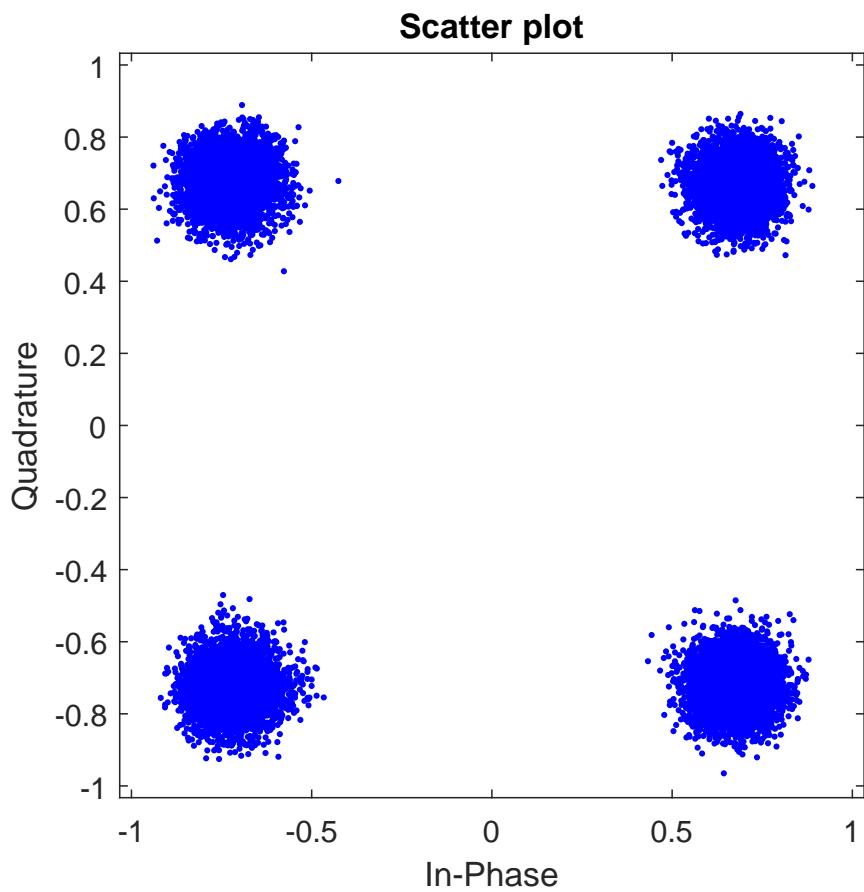


Figure 6.166: Final constellation obtained after processing. The BER in this case is 0. No BER curve shown because the number of samples was too low to obtain a BER higher than 0 at any tested SNR.

### 6.10.1.2 1 GBd Signal

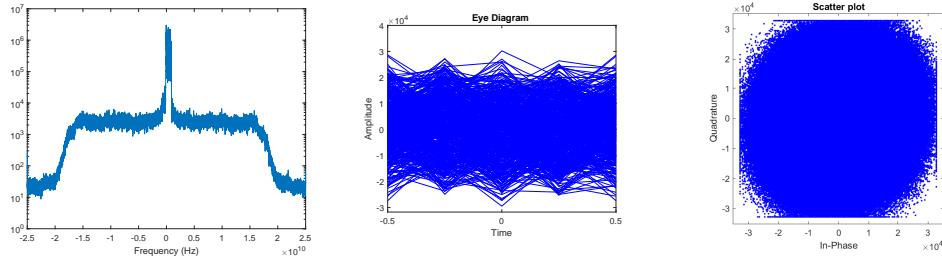


Figure 6.167: Eye Diagram and spectrum of the original signal obtained at the oscilloscope. Initial estimated SNR before any signal processing is 9.5887 dB.

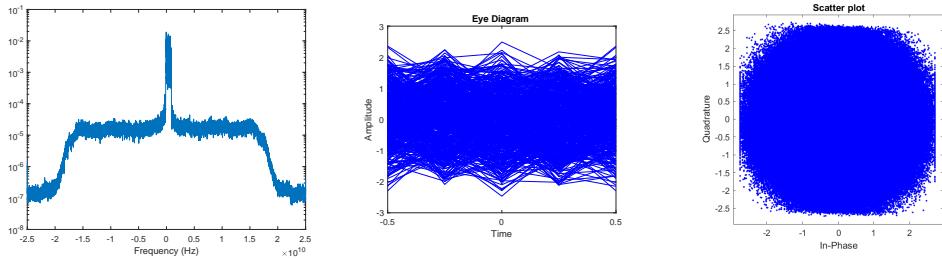


Figure 6.168: Eye Diagram and spectrum after front end corrections, including removal of DC component, deskew and Gram-Schmidt Orthonormalization.

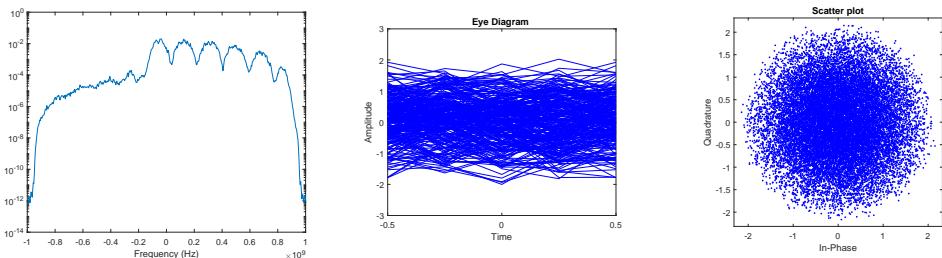


Figure 6.169: Eye Diagram and spectrum after applying the matched filter and downsampling to  $2S_R$ .

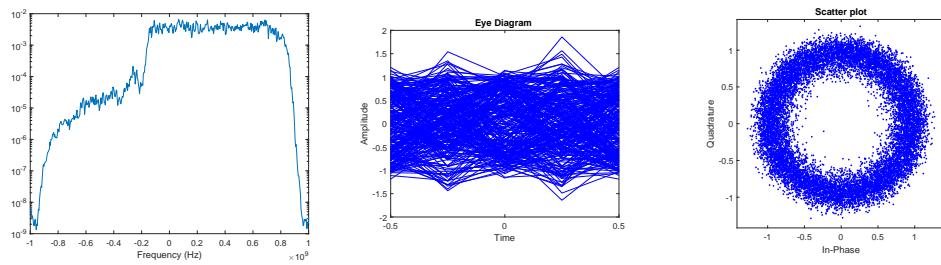


Figure 6.170: Eye Diagram and spectrum after the first stage of adaptive equalization, using a MIMO 2x2 CMA equalizer.

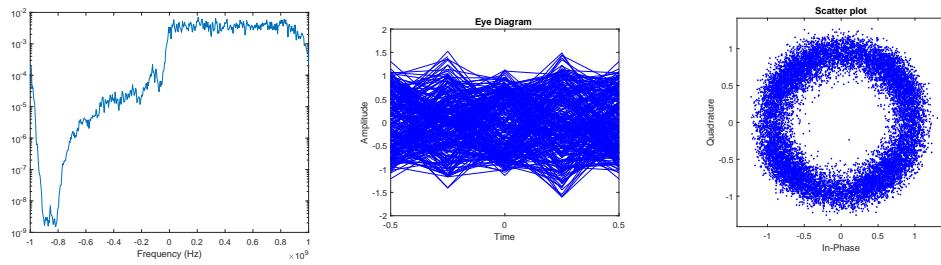


Figure 6.171: Eye Diagram and spectrum after frequency offset correction.

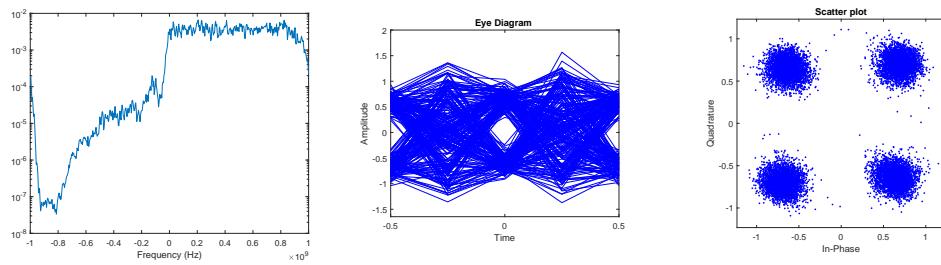


Figure 6.172: Eye Diagram and spectrum after the first stage of carrier-phase estimation.

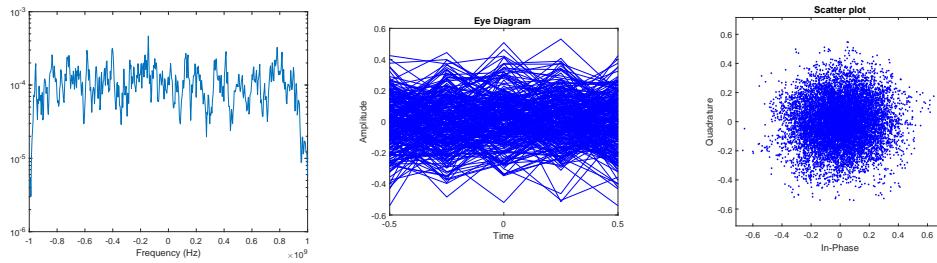


Figure 6.173: Eye Diagram and spectrum after the second stage of adaptive equalization, with a MIMO 4x4 LMS equalizer.

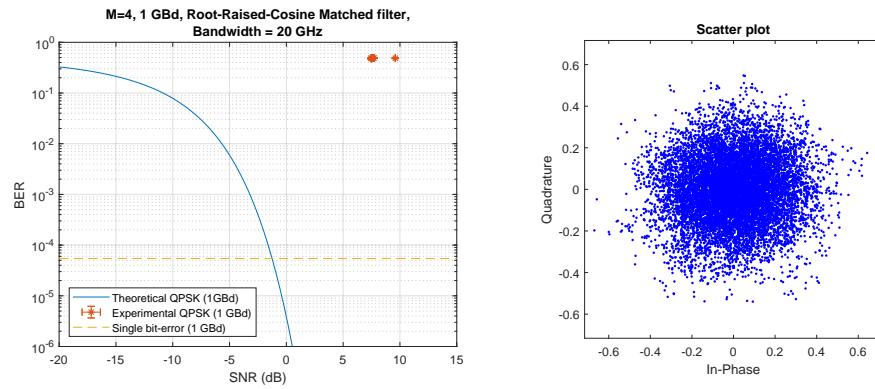


Figure 6.174: BER curve and final constellation obtained after processing. The BER in this case is 0.4866.

#### 6.10.1.3 1 GBd Signal without Downsampling before processing

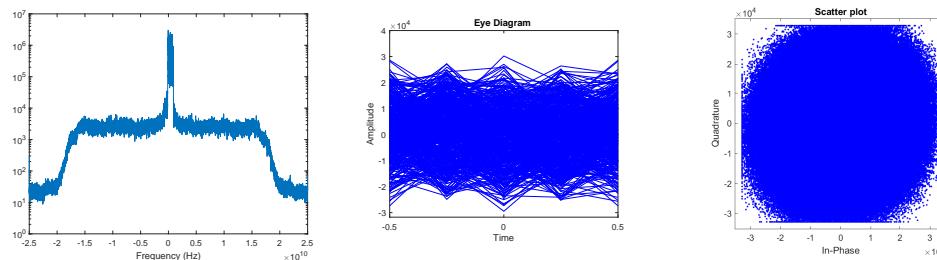


Figure 6.175: Eye Diagram and spectrum of the original signal obtained at the oscilloscope. Initial estimated SNR before any signal processing is 9.5887 dB.

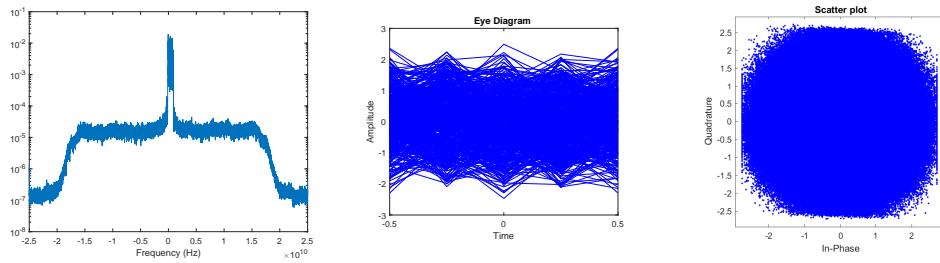


Figure 6.176: Eye Diagram and spectrum after front end corrections, including removal of DC component, deskew and Gram-Schmidt Orthonormalization.

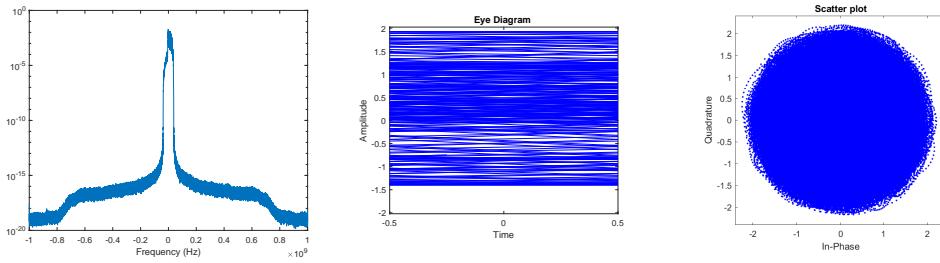


Figure 6.177: Eye Diagram and spectrum after applying the matched filter and downsampling to  $2S_R$ .

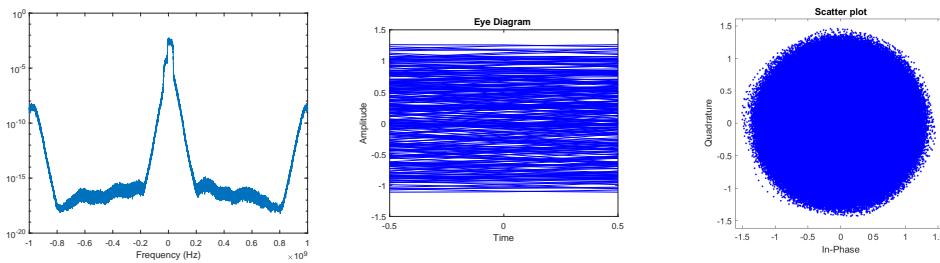


Figure 6.178: Eye Diagram and spectrum after the first stage of adaptive equalization, using a MIMO 2x2 CMA equalizer.

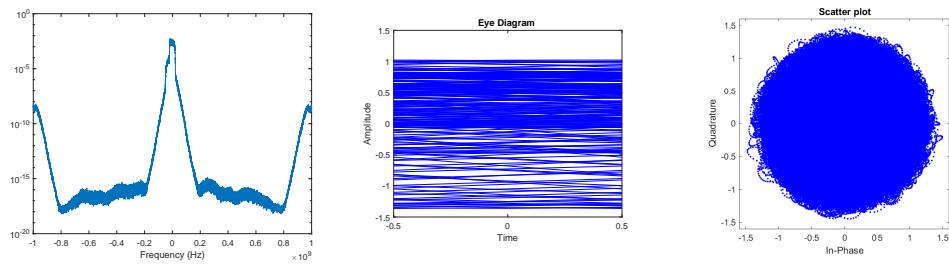


Figure 6.179: Eye Diagram and spectrum after frequency offset correction.

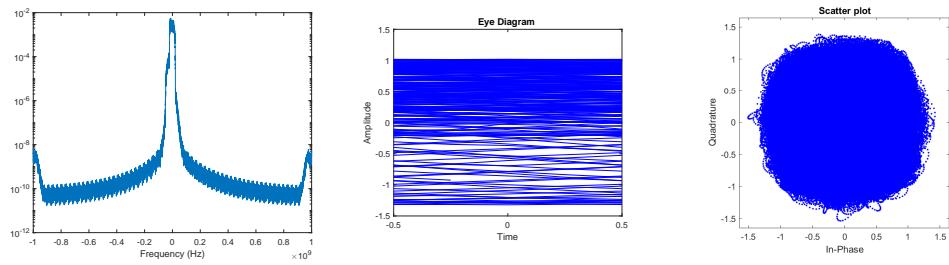


Figure 6.180: Eye Diagram and spectrum after the first stage of carrier-phase estimation.

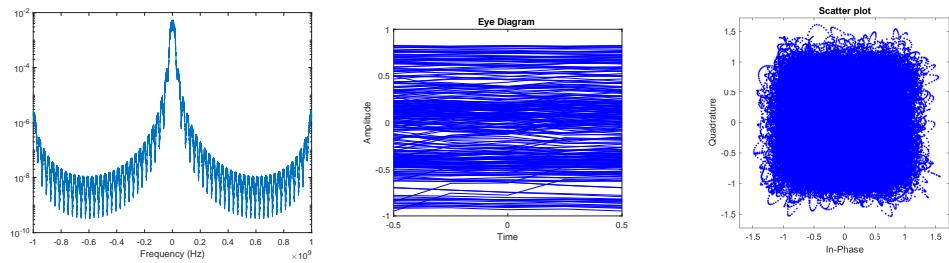


Figure 6.181: Eye Diagram and spectrum after the second stage of adaptive equalization, with a MIMO 4x4 LMS equalizer.

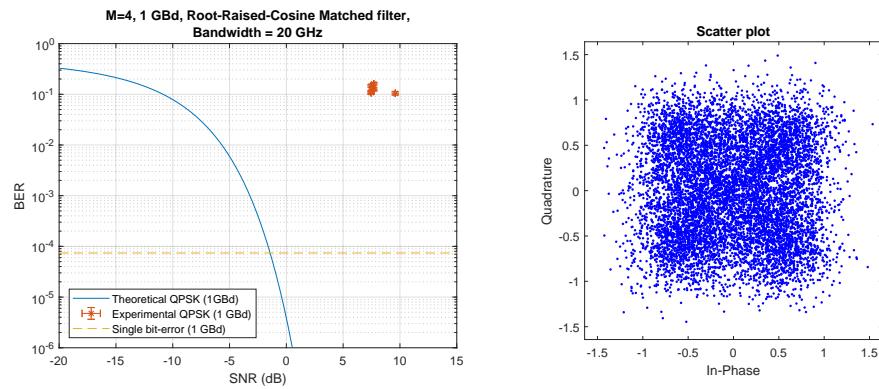


Figure 6.182: BER curve and final constellation obtained after processing. The BER in this case is 0.1044.

#### 6.10.1.4 1 GBd Signal with Upsampling before Frequency estimation

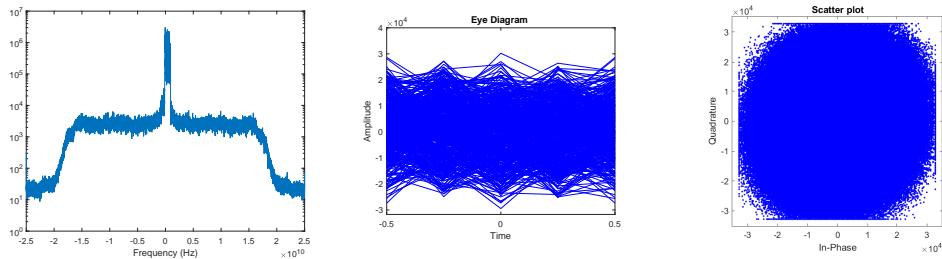


Figure 6.183: Eye Diagram and spectrum of the original signal obtained at the oscilloscope. Initial estimated SNR before any signal processing is 9.5887 dB.

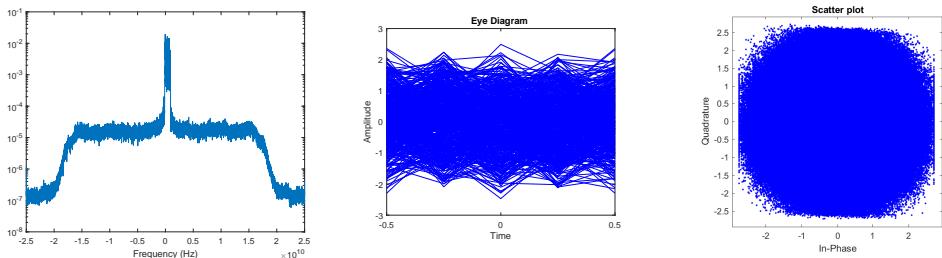


Figure 6.184: Eye Diagram and spectrum after front end corrections, including removal of DC component, deskew and Gram-Schmidt Orthonormalization.

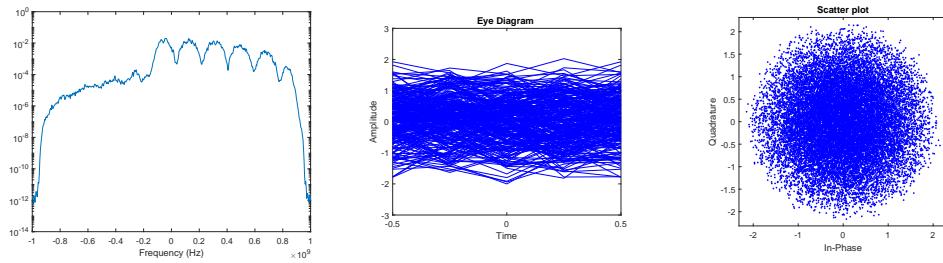


Figure 6.185: Eye Diagram and spectrum after applying the matched filter and downsampling to  $2S_R$ .

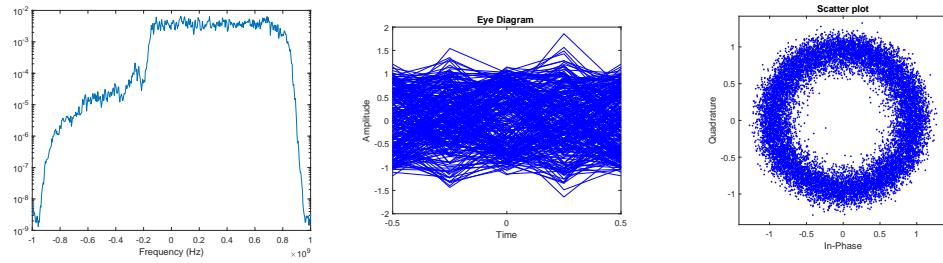


Figure 6.186: Eye Diagram and spectrum after the first stage of adaptive equalization, using a MIMO 2x2 CMA equalizer.

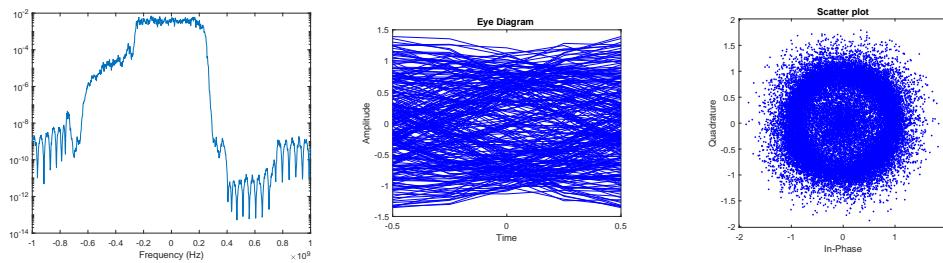


Figure 6.187: Eye Diagram and spectrum after frequency offset correction.

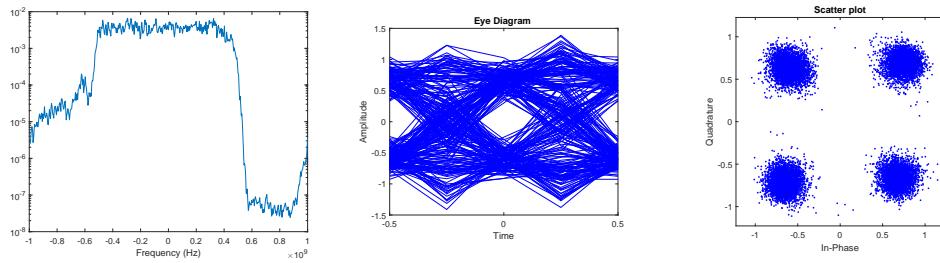


Figure 6.188: Eye Diagram and spectrum after the first stage of carrier-phase estimation.

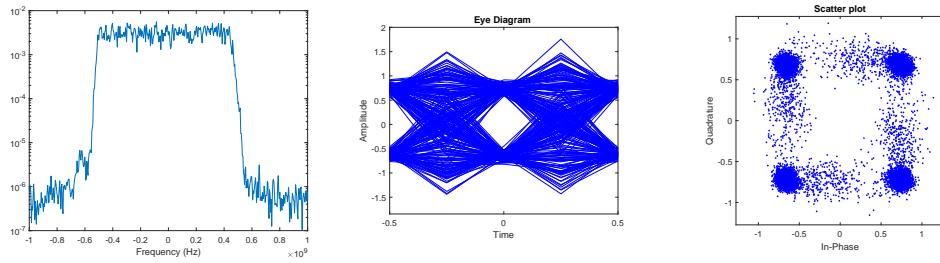


Figure 6.189: Eye Diagram and spectrum after the second stage of adaptive equalization, with a MIMO 4x4 LMS equalizer.

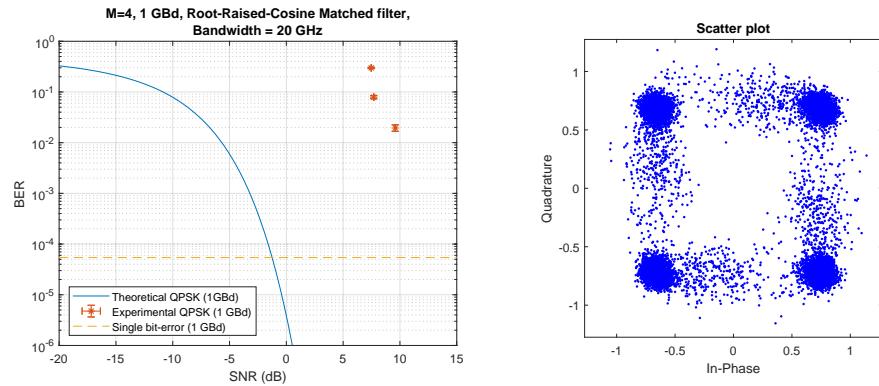


Figure 6.190: BER curve and final constellation obtained after processing. The BER in this case is 0.0195.

### 6.10.1.5 500 MBd Signal

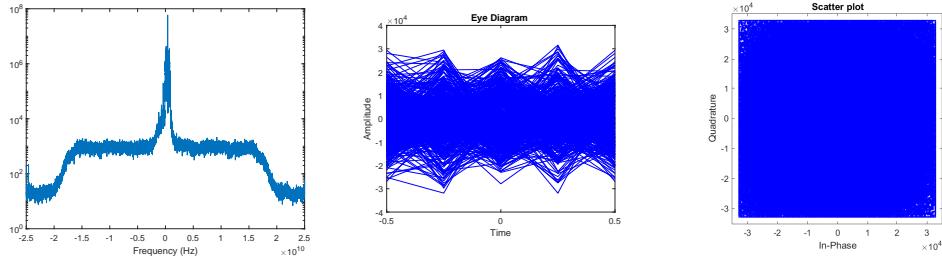


Figure 6.191: Eye Diagram and spectrum of the original signal obtained at the oscilloscope. Initial estimated SNR before any signal processing is 8.405 dB.

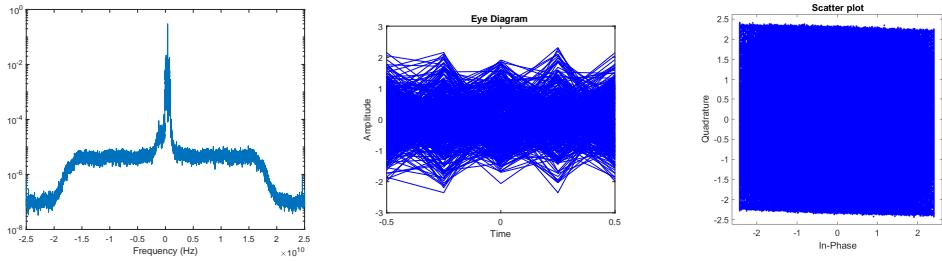


Figure 6.192: Eye Diagram and spectrum after front end corrections, including removal of DC component, deskew and Gram-Schmidt Orthonormalization.

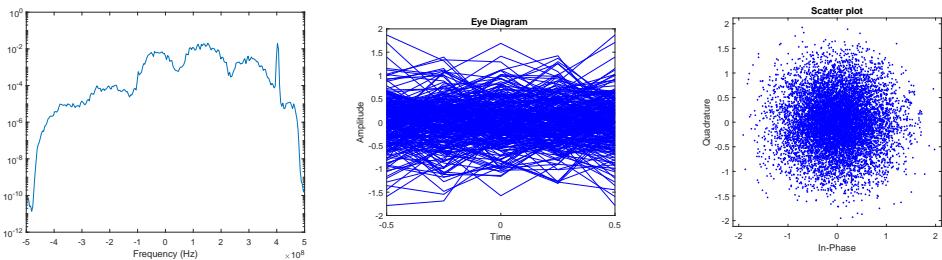


Figure 6.193: Eye Diagram and spectrum after applying the matched filter and downsampling to  $2S_R$ .

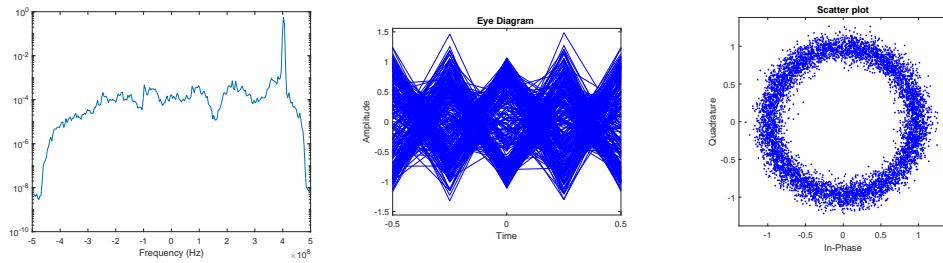


Figure 6.194: Eye Diagram and spectrum after the first stage of adaptive equalization, using a MIMO 2x2 CMA equalizer.

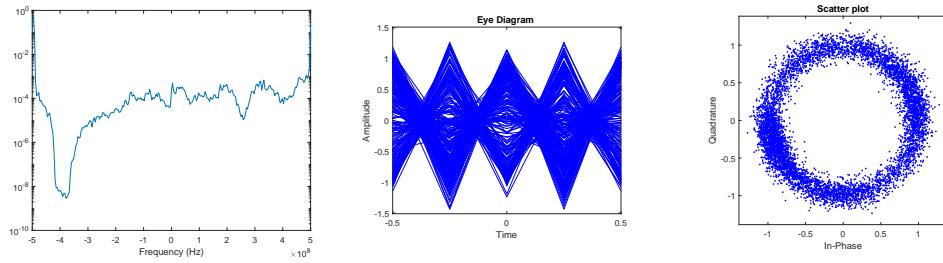


Figure 6.195: Eye Diagram and spectrum after frequency offset correction.

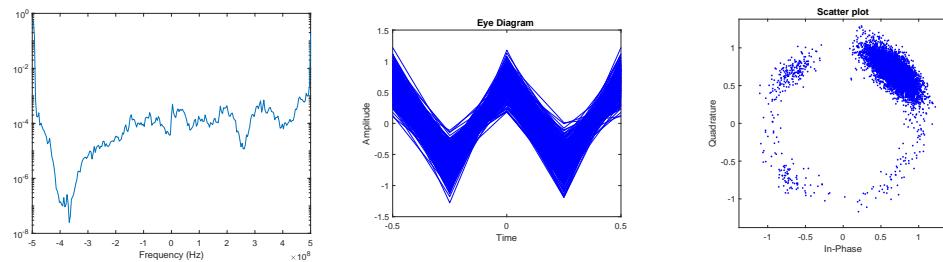


Figure 6.196: Eye Diagram and spectrum after the first stage of carrier-phase estimation.

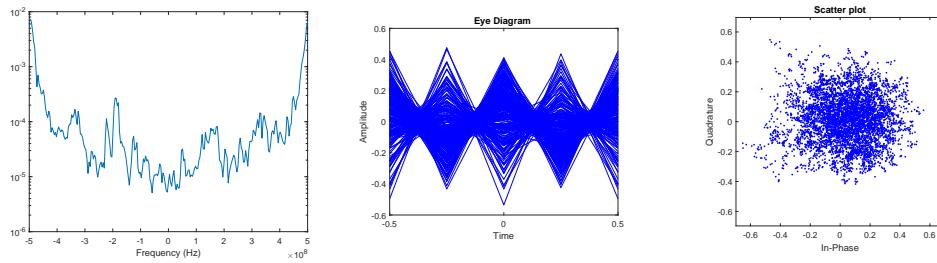


Figure 6.197: Eye Diagram and spectrum after the second stage of adaptive equalization, with a MIMO 4x4 LMS equalizer.

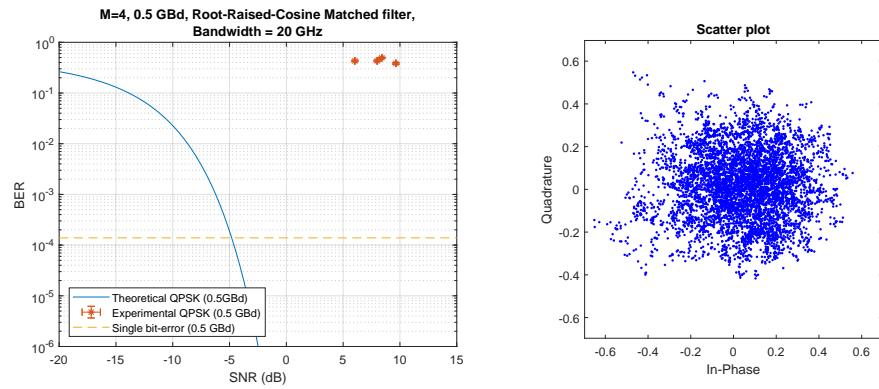


Figure 6.198: BER curve and final constellation obtained after processing. The BER in this case is 0.4939.

#### 6.10.1.6 500 MBd Signal without Downsampling before processing

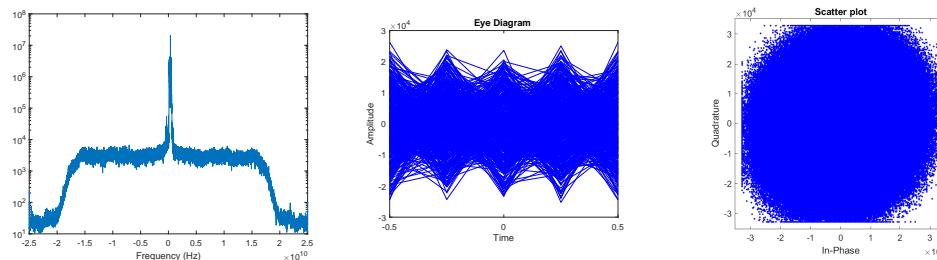


Figure 6.199: Eye Diagram and spectrum of the original signal obtained at the oscilloscope. Initial estimated SNR before any signal processing is 9.5887 dB.

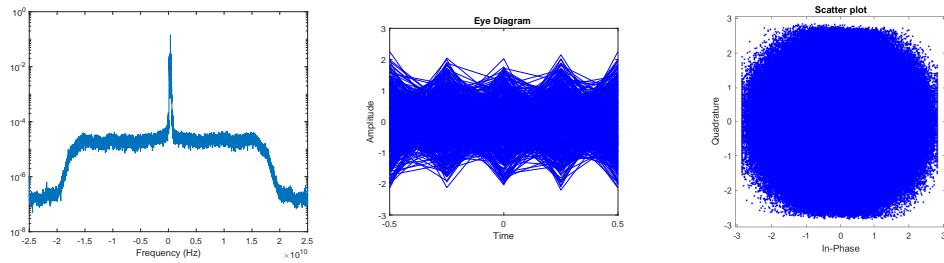


Figure 6.200: Eye Diagram and spectrum after front end corrections, including removal of DC component, deskew and Gram-Schmidt Orthonormalization.

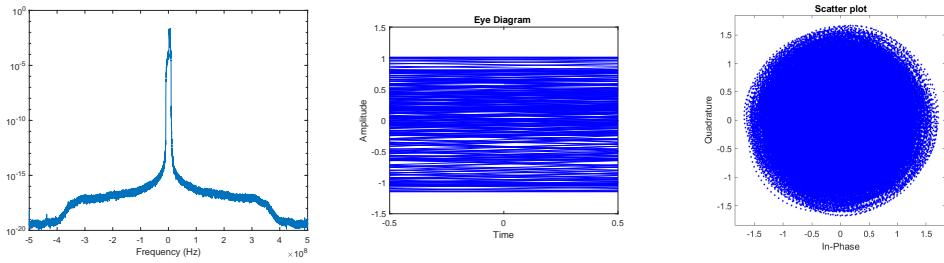


Figure 6.201: Eye Diagram and spectrum after applying the matched filter and downsampling to  $2S_R$ .

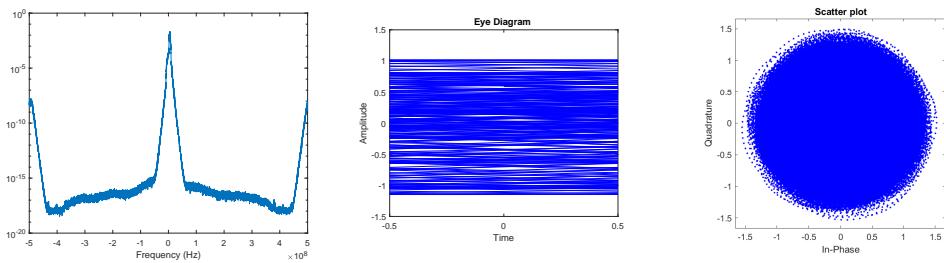


Figure 6.202: Eye Diagram and spectrum after the first stage of adaptive equalization, using a MIMO 2x2 CMA equalizer.

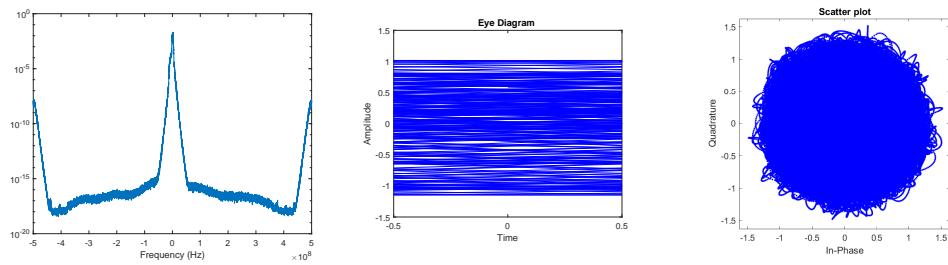


Figure 6.203: Eye Diagram and spectrum after frequency offset correction.

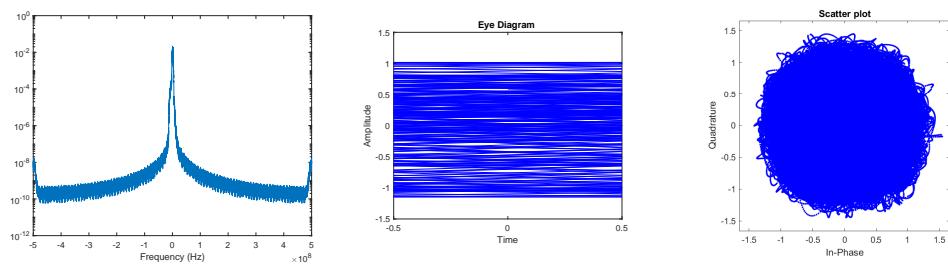


Figure 6.204: Eye Diagram and spectrum after the first stage of carrier-phase estimation.

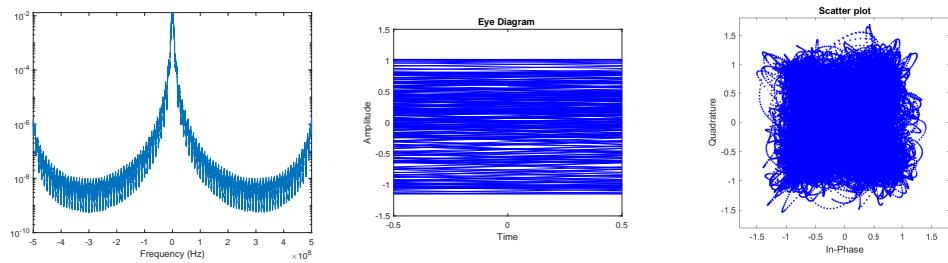


Figure 6.205: Eye Diagram and spectrum after the second stage of adaptive equalization, with a MIMO 4x4 LMS equalizer.

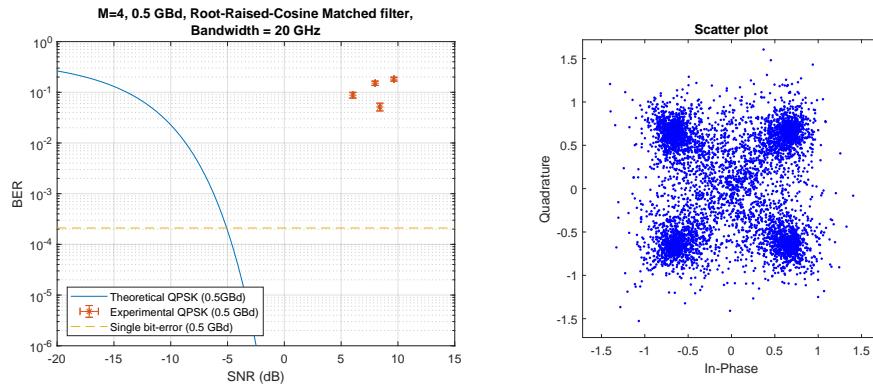


Figure 6.206: BER curve and final constellation obtained after processing. The BER in this case is 0.0514.

### 500 MBd Signal with Upsampling before Frequency estimation

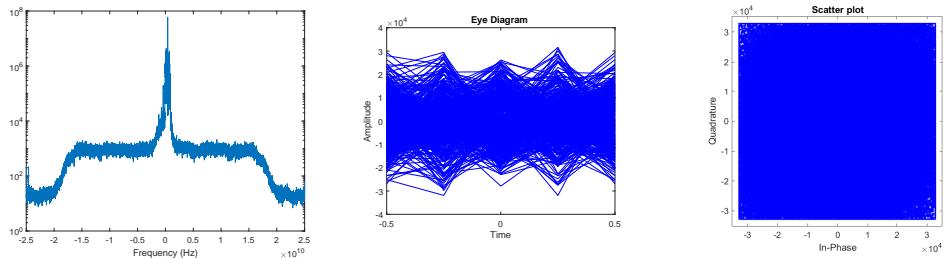


Figure 6.207: Eye Diagram and spectrum of the original signal obtained at the oscilloscope. Initial estimated SNR before any signal processing is 9.5887 dB.

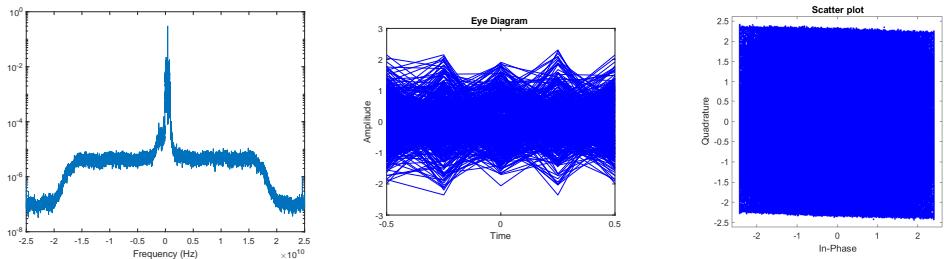


Figure 6.208: Eye Diagram and spectrum after front end corrections, including removal of DC component, deskew and Gram-Schmidt Orthonormalization.

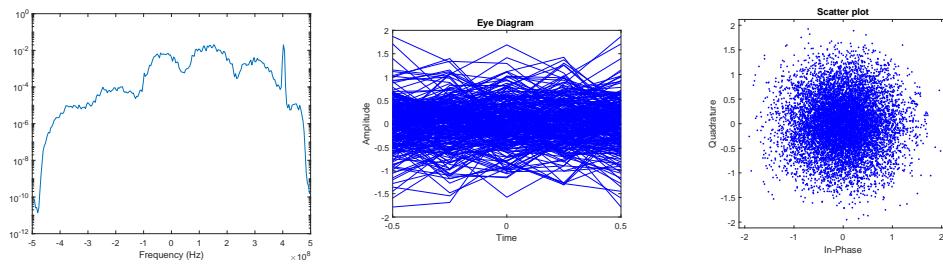


Figure 6.209: Eye Diagram and spectrum after applying the matched filter and downsampling to  $2S_R$ .

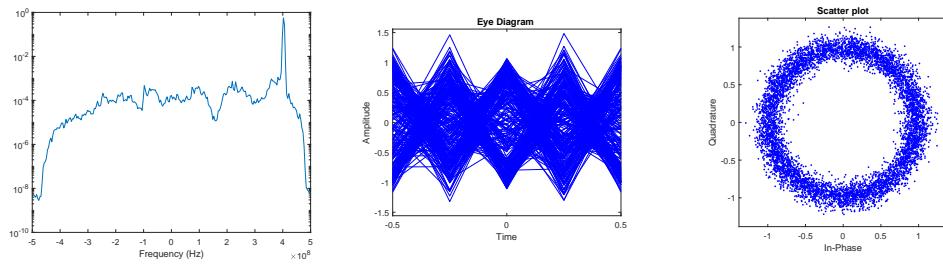


Figure 6.210: Eye Diagram and spectrum after the first stage of adaptive equalization, using a MIMO 2x2 CMA equalizer.

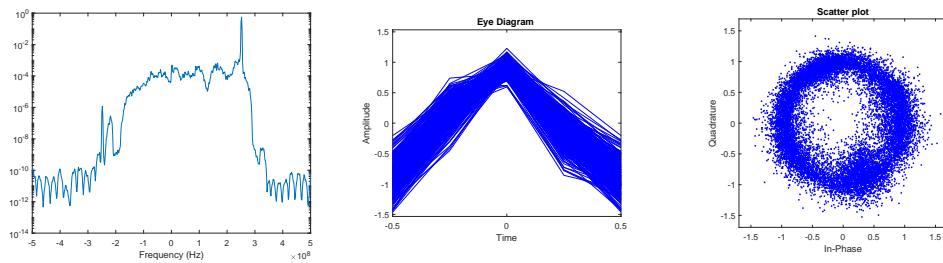


Figure 6.211: Eye Diagram and spectrum after frequency offset correction.

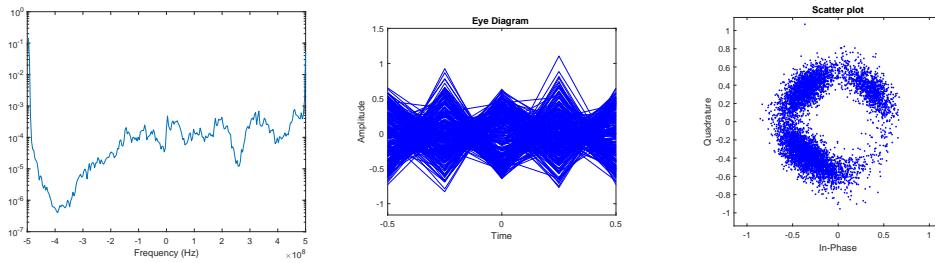


Figure 6.212: Eye Diagram and spectrum after the first stage of carrier-phase estimation.

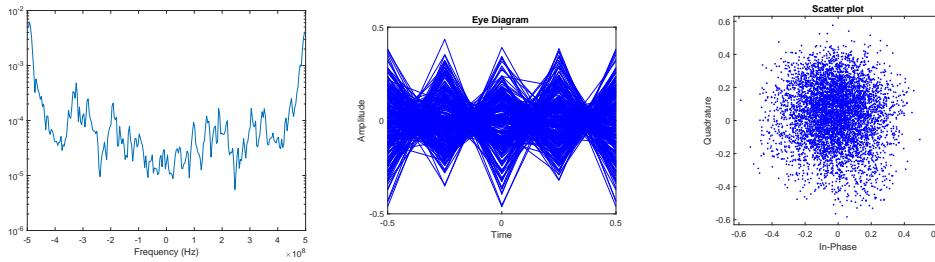


Figure 6.213: Eye Diagram and spectrum after the second stage of adaptive equalization, with a MIMO 4x4 LMS equalizer.

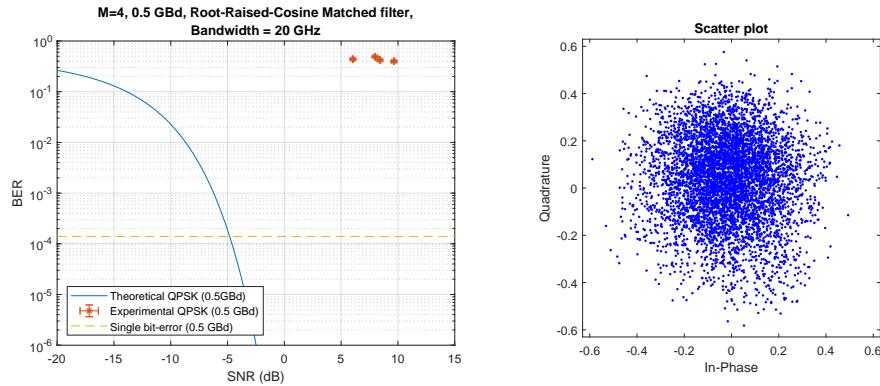


Figure 6.214: BER curve and final constellation obtained after processing. The BER in this case is 0.3741.

#### 6.10.1.7 Bias controller issues

During the next phase of experimental testing, where the symbol rate was reduced further, (2 GBd, 1GBd, 500MBd) there were issues with the bias controller of the IQ modulator.

Normally it requires manual intervention to start at the right values and afterwards it automatically sets the optimal bias values and keeps them in check. However, for these round of tests it did not perform as expected, instead requiring the bias values to be manually set and fixed in order to keep the acquisition stable. While this allowed circumventing the misbehavior of the bias controller, the values were likely not optimal, leading to irregularities in the acquired constellations. However, this can be somewhat compensated with DSP, by carrier-phase estimation, as shown in the following sections.

#### 6.10.1.8 2 GBd Signal, Homodyne Detection

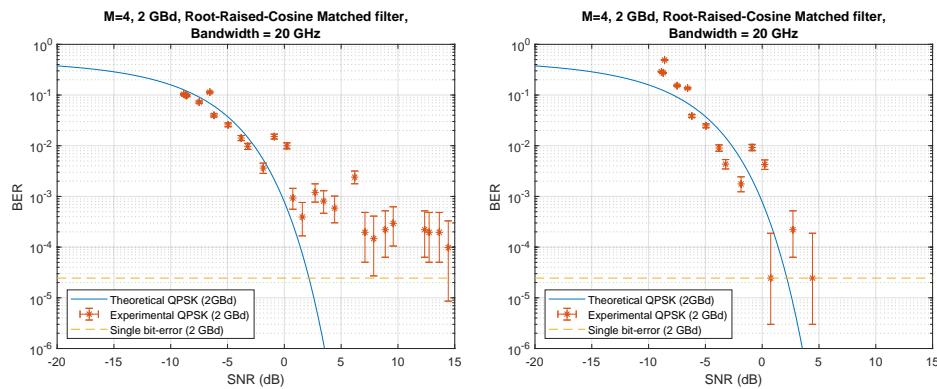


Figure 6.215: BER curves as a function of the SNR for the 2GBd signal. The curve on the left was obtained without any carrier phase recovery.

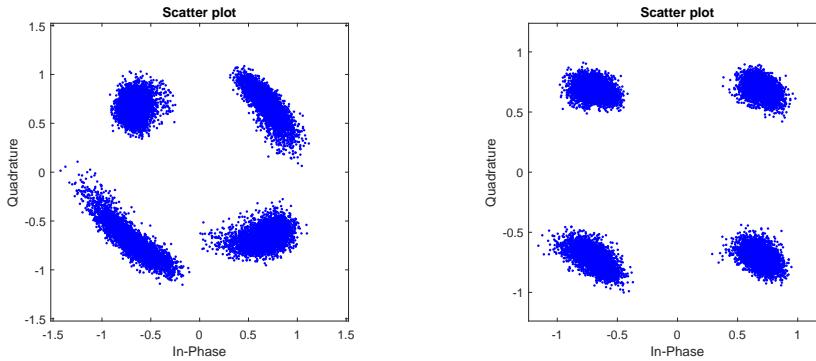


Figure 6.216: Constellation for the 2GBd signal with an estimated SNR of 14.43 dB. The constellation on the left was obtained with no carrier phase recovery.

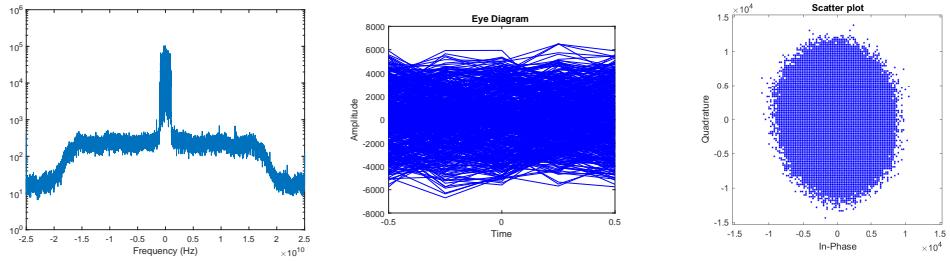


Figure 6.217: Eye Diagram and spectrum of the original signal obtained at the oscilloscope. Initial estimated SNR before any signal processing is 11.1265 dB.

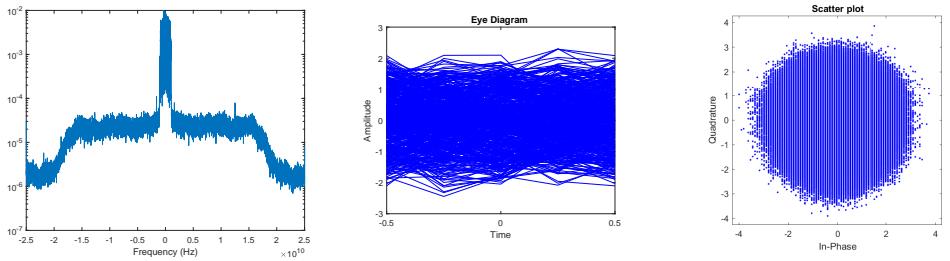


Figure 6.218: Eye Diagram and spectrum after front end corrections, including removal of DC component, deskew and Gram-Schmidt Orthonormalization.

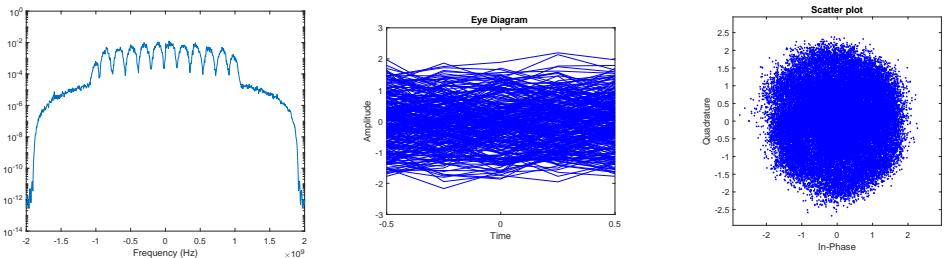


Figure 6.219: Eye Diagram and spectrum after applying the matched filter and downsampling to  $2S_R$ .

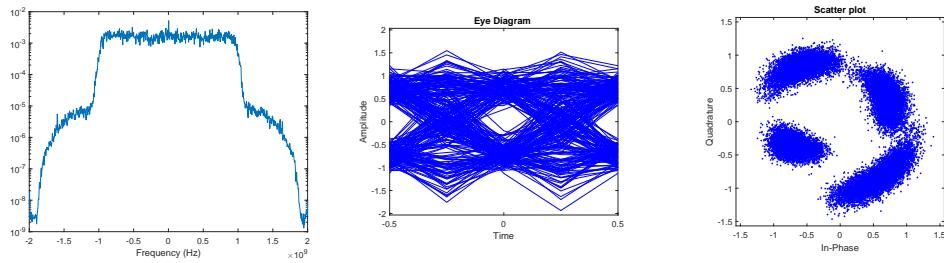


Figure 6.220: Eye Diagram and spectrum after the first stage of adaptive equalization, using a MIMO 2x2 CMA equalizer.

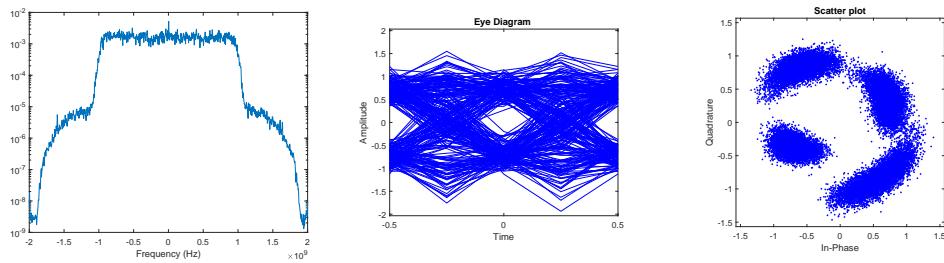


Figure 6.221: Eye Diagram and spectrum after frequency offset correction.

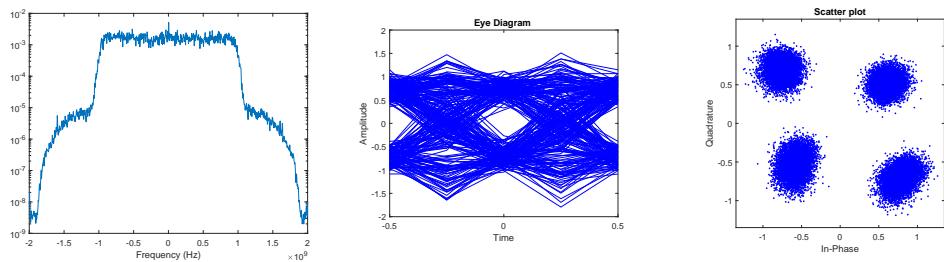


Figure 6.222: Eye Diagram and spectrum after the first stage of carrier-phase estimation.

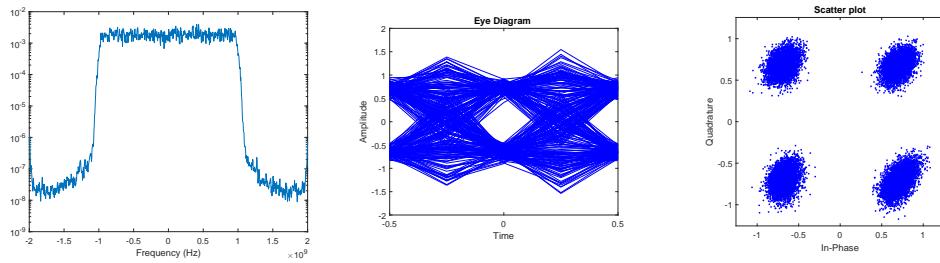


Figure 6.223: Eye Diagram and spectrum after the second stage of adaptive equalization, with a MIMO 4x4 LMS equalizer.

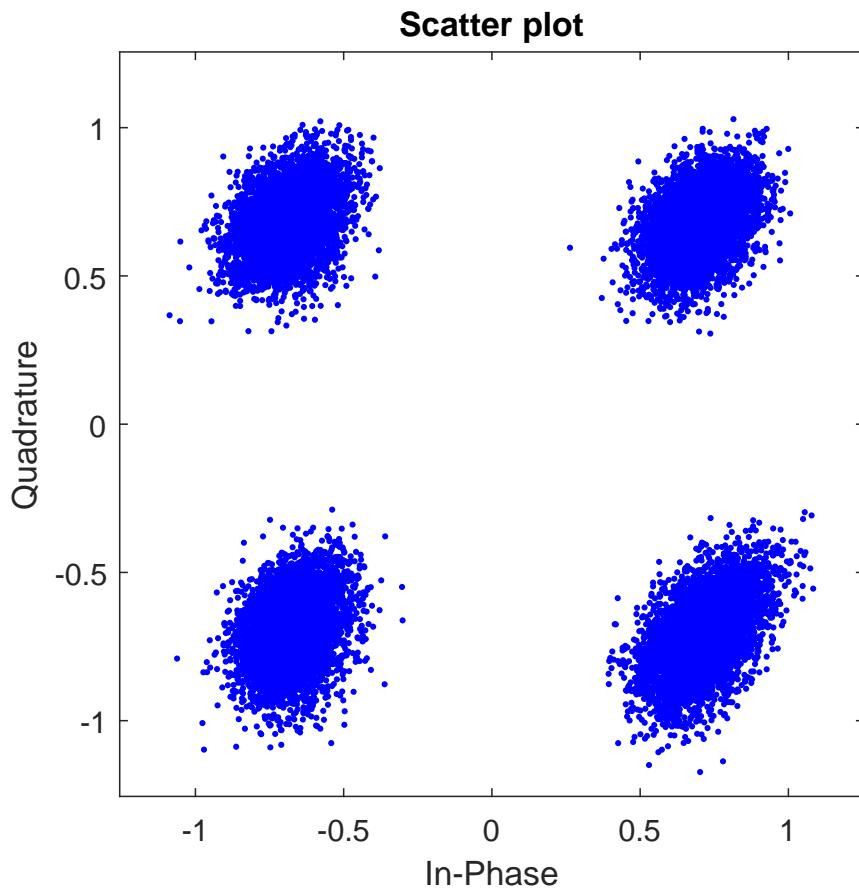


Figure 6.224: Eye Diagram after downsampling to 1 sample per symbol, and final constellation obtained after processing. Final SNR estimated through the moments method [1] is 12.3261 dB. The BER in this case is 0.

### 6.10.1.9 1 GBd Signal, Homodyne Detection

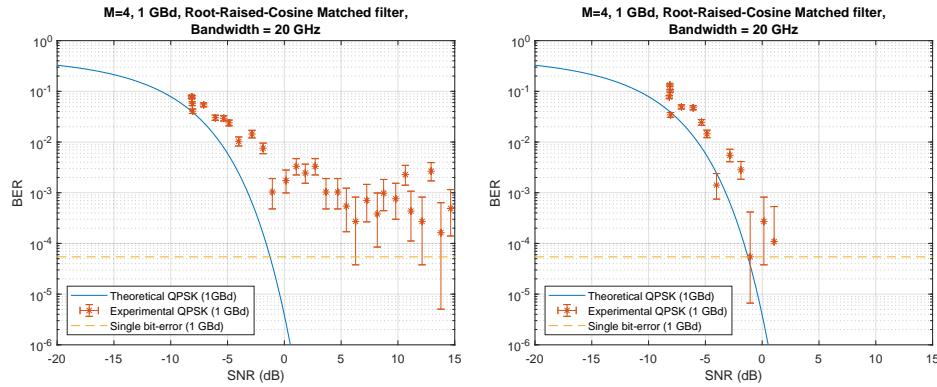


Figure 6.225: BER curves as a function of the SNR for the 1GBd signal. The curve on the left was obtained without any carrier phase recovery.

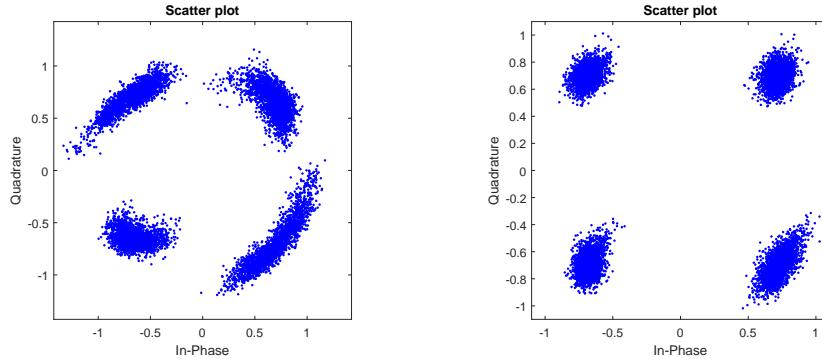


Figure 6.226: Constellation for the 1GBd signal with an estimated SNR of 14.6389 dB. The constellation on the left was obtained with no carrier phase recovery.

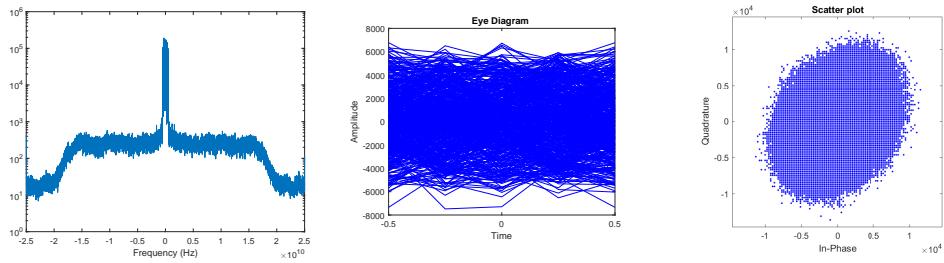


Figure 6.227: Eye Diagram and spectrum of the original signal obtained at the oscilloscope. Initial estimated SNR before any signal processing is 9.5887 dB.

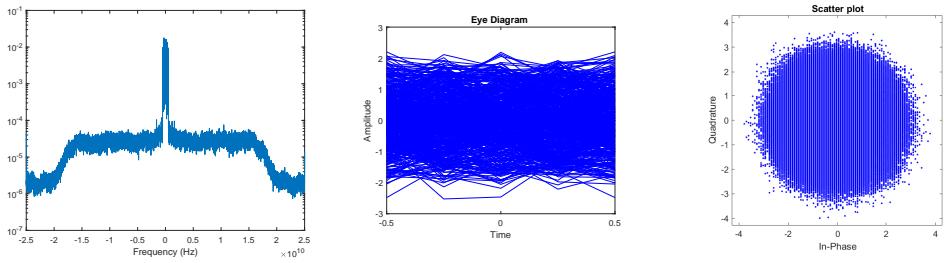


Figure 6.228: Eye Diagram and spectrum after front end corrections, including removal of DC component, deskew and Gram-Schmidt Orthonormalization.

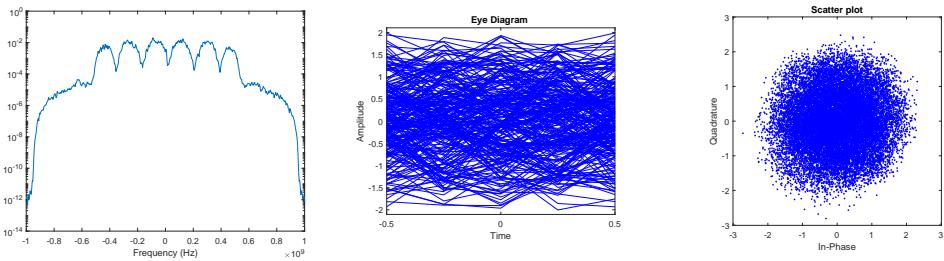


Figure 6.229: Eye Diagram and spectrum after applying the matched filter and downsampling to  $2S_R$ .

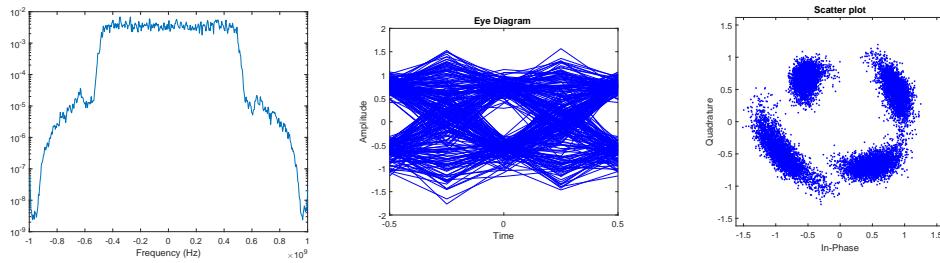


Figure 6.230: Eye Diagram and spectrum after the first stage of adaptive equalization, using a MIMO 2x2 CMA equalizer.

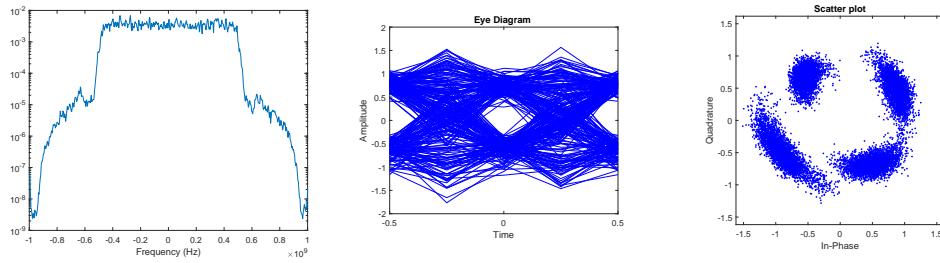


Figure 6.231: Eye Diagram and spectrum after frequency offset correction.

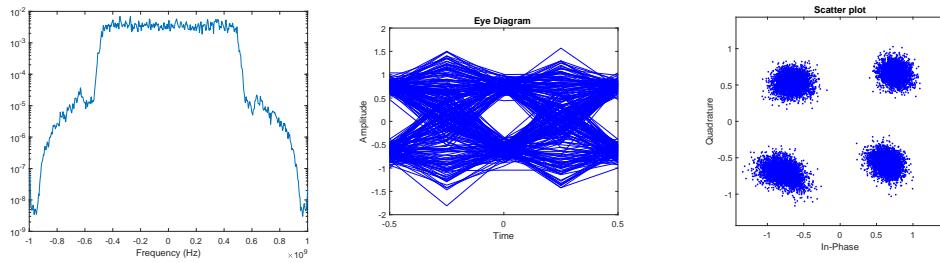


Figure 6.232: Eye Diagram and spectrum after the first stage of carrier-phase estimation.

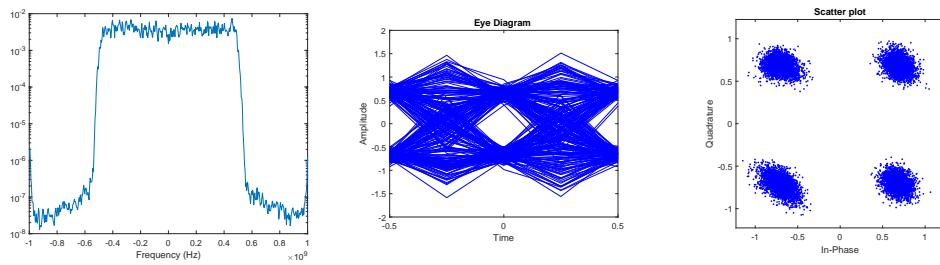


Figure 6.233: Eye Diagram and spectrum after the second stage of adaptive equalization, with a MIMO 4x4 LMS equalizer.

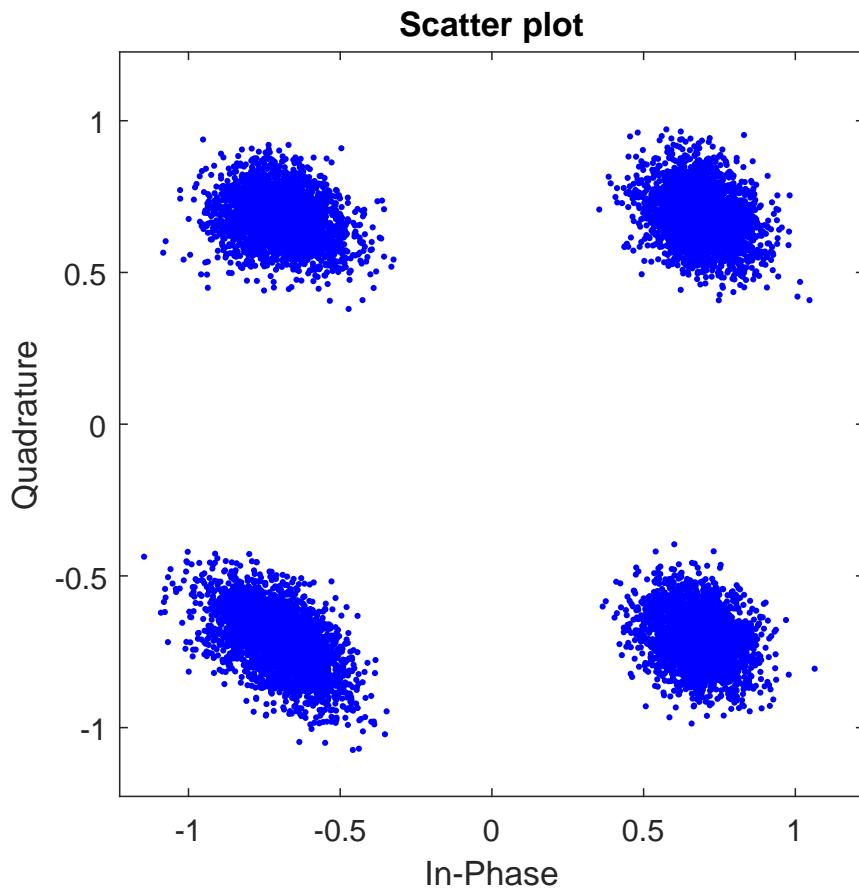


Figure 6.234: Eye Diagram after downsampling to 1 sample per symbol, and final constellation obtained after processing. Final SNR estimated through the moments method [1] is 12.3261 dB. The BER in this case is 0.4866.

### 6.10.1.10 500 MBd Signal, Homodyne Detection

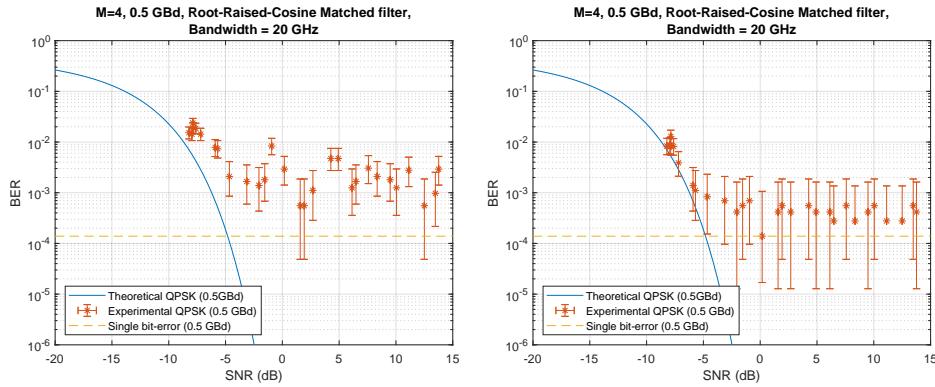


Figure 6.235: BER curves as a function of the SNR for the 500 MHz signal. The curve on the left was obtained without any carrier phase recovery.

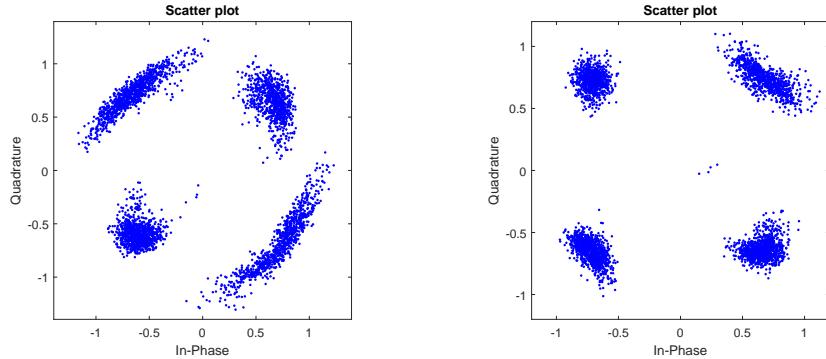


Figure 6.236: Constellation for the 500MBd signal with an estimated SNR of 14.6389 dB. The constellation on the left was obtained with no carrier phase recovery.

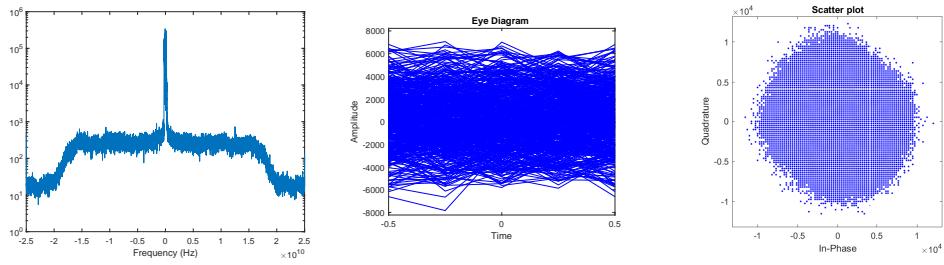


Figure 6.237: Eye Diagram and spectrum of the original signal obtained at the oscilloscope. Initial estimated SNR before any signal processing is 9.5887 dB.

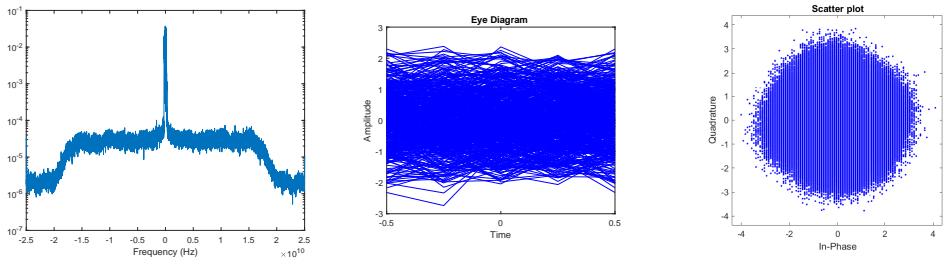


Figure 6.238: Eye Diagram and spectrum after front end corrections, including removal of DC component, deskew and Gram-Schmidt Orthonormalization.

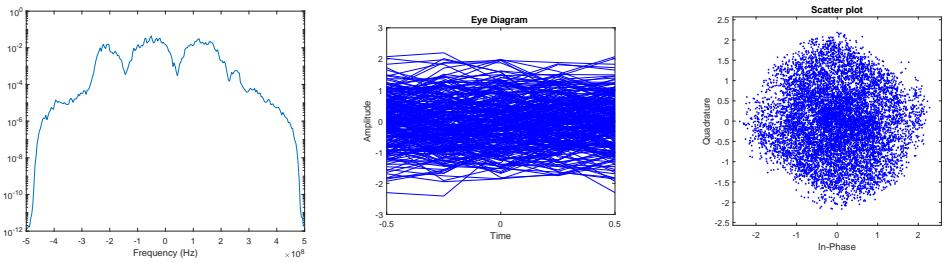


Figure 6.239: Eye Diagram and spectrum after applying the matched filter and downsampling to  $2S_R$ .

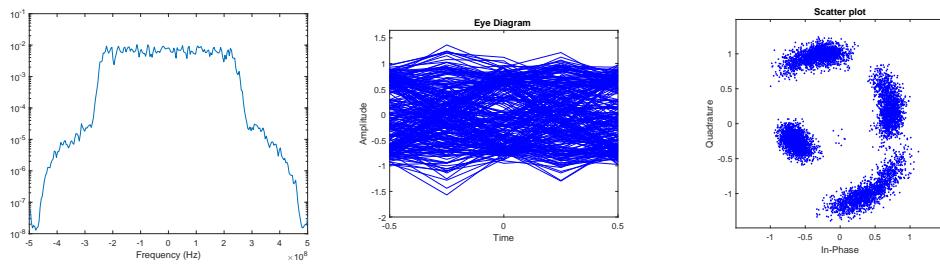


Figure 6.240: Eye Diagram and spectrum after the first stage of adaptive equalization, using a MIMO 2x2 CMA equalizer.

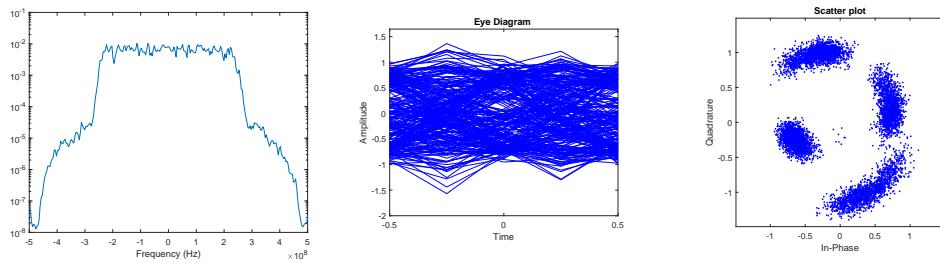


Figure 6.241: Eye Diagram and spectrum after frequency offset correction.

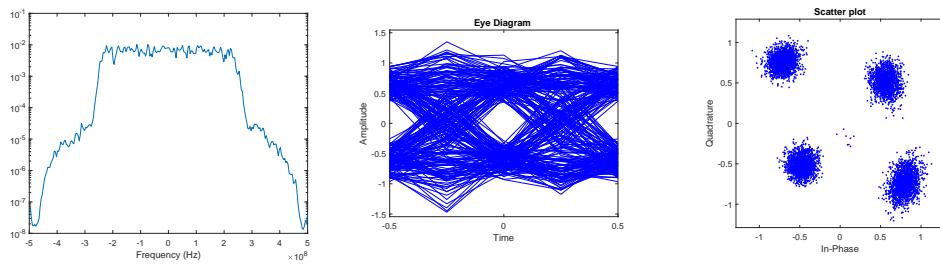


Figure 6.242: Eye Diagram and spectrum after the first stage of carrier-phase estimation.

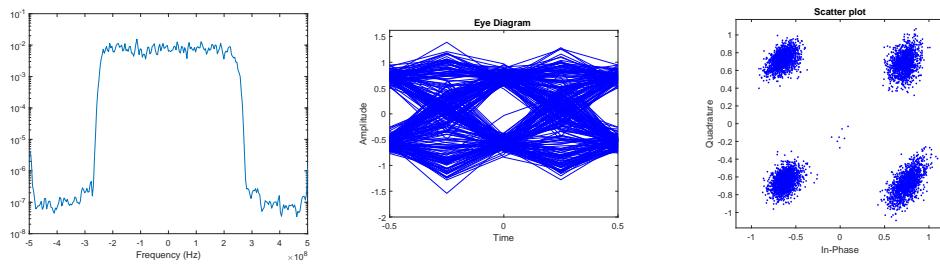


Figure 6.243: Eye Diagram and spectrum after the second stage of adaptive equalization, with a MIMO 4x4 LMS equalizer.

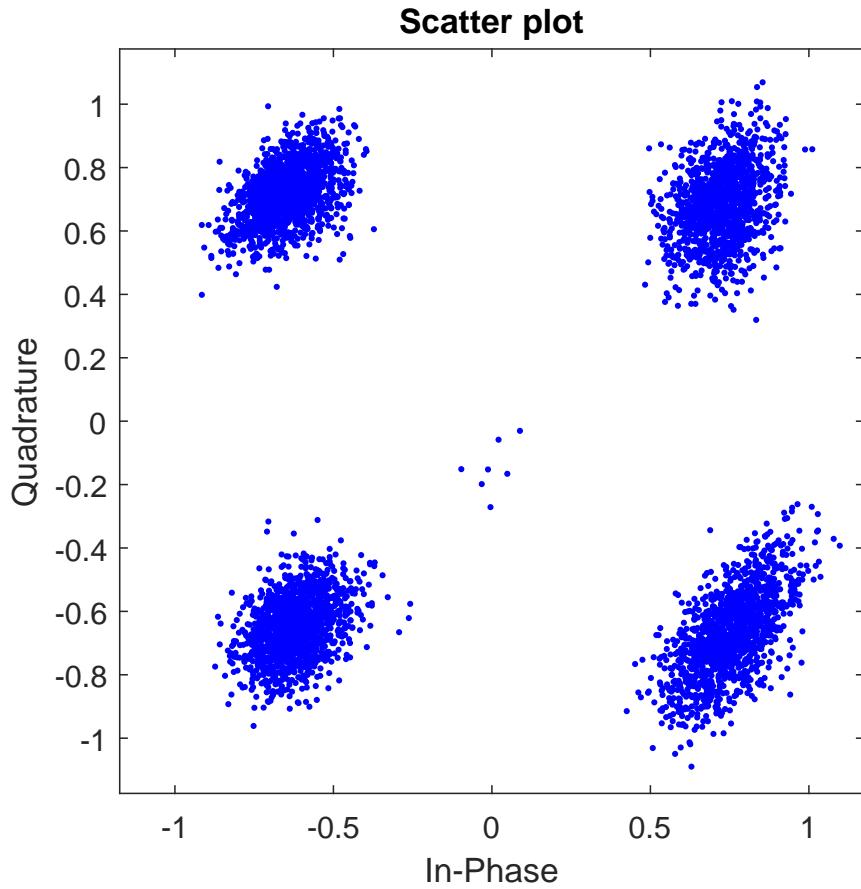


Figure 6.244: Eye Diagram after downsampling to 1 sample per symbol, and final constellation obtained after processing. Final SNR estimated through the moments method [1] is 12.3261 dB. The BER in this case is 0.4866.

### 6.10.2 Open Issues

The bias controller seems to have trouble when set to act automatically with lower baud rates. On intradyne configuration, frequency offset corrections are problematic as usually other processing occurs before which uses the expected frequency, such as matched filtering. Doing processing before frequency offset correction may compromise the ability to correct the offset. However, doing frequency correction first may not work as well.

## References

- [1] Rolf Matzner. "An SNR estimation algorithm for complex baseband signals using higher order statistics". In: *Facta Universitatis (Nis)* 6.1 (1993), pp. 41–52.

## 6.11 Kramers-Kronig Transceiver

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

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

### 6.11.1 Theoretical Analysis

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

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

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

where *p.v.* stands for *principle value* which is a standard method applied in mathematical applications by which an improper, and possibly divergent, integral is measured in a balanced way around singularities or at infinity [3]. A signal that satisfies the Kramers-Kronig relationship is also named as an analytical signal. This relationship imposes that the real and the imaginary parts of the signal are related to each other through Hilbert transform. Therefore, if we have the real part of the signal then the imaginary part can be calculated by its Hilbert transform.

For a signal that satisfies the Kramers-Kronig relationship, the real and imaginary part can be obtained only from the module. The following questions would give a comprehensive overview of Kramers-Kroning relation and the detailed mathematical calculation which depicts how phase can be extracted from the amplitude information.

### 6.11.1.1 1. What is the Hilbert transform?

If we consider a filter  $H(\omega)$ , described in Figure 6.245, that has a unity magnitude response for all frequencies and the phase response is  $\pi/2$  for all positive frequencies and  $-\pi/2$  for negative frequencies. The transfer function of this filter is given by

$$H(\omega) = -i \operatorname{sgn}(\omega) = \begin{cases} -i & \text{for } i > 0 \\ i & \text{for } i < 0 \\ 0 & \text{for } i = 0 \end{cases} \quad (6.152)$$

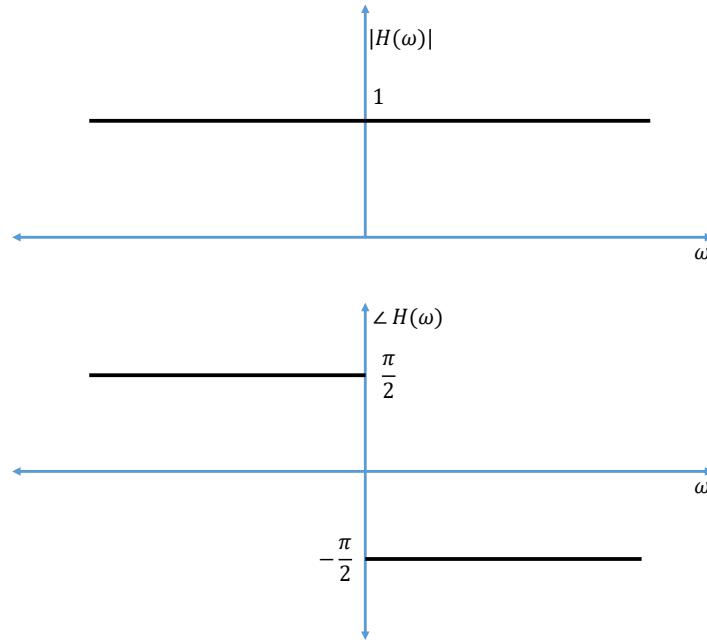


Figure 6.245: Magnitude and phase of Hilbert transform filter

The impulse response of this filter can be given as,

$$\begin{aligned} h(t) &= \mathcal{F}^{-1}[H(i\omega)] \\ &= -i\mathcal{F}^{-1}[\operatorname{sgn}(\omega)] \\ &= -i\left(\frac{i}{\pi t}\right) \\ &= \frac{1}{\pi t} \end{aligned} \quad (6.153)$$

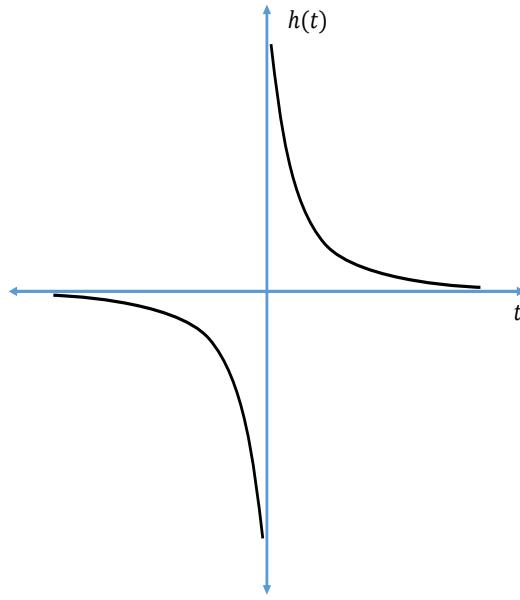


Figure 6.246: Impulse response  $h(t)$  of Hilbert transform filter

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

$$\begin{aligned}\hat{s}(t) &= s(t) \circledast h(t) \\ &= \int_{-\infty}^{\infty} \frac{s(u)}{\pi(t-u)} du\end{aligned}\tag{6.154}$$

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

$$\mathcal{F}[\hat{s}(t)] = H(\omega)S(\omega) = i \operatorname{sgn}(\omega)S(\omega)\tag{6.155}$$

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

#### 6.11.1.2 Example of rectangle function

Consider the rectangular signal  $r(t)$  and its Hilbert transformed  $\hat{r}(t)$  as shown in the Figure 6.247. The Fourier transformed version of the signal  $r(t)$  and  $\hat{r}(t)$  is described as  $R(f)$  and

$\hat{R}(f)$  and their magnitude and the phase spectrum are shown in Figure 6.248 and 6.249, respectively.

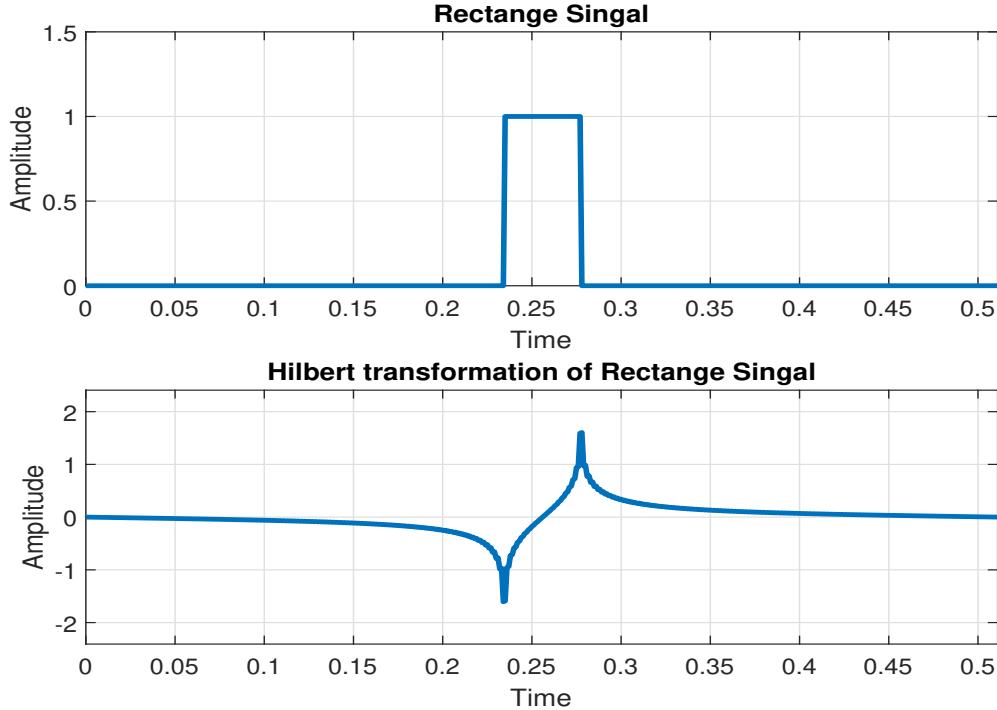


Figure 6.247: Rectangle signal  $r(t)$  and its Hilbert transformed  $\hat{r}(t)$

### 6.11.1.3 2. What is an analytical signal?

An analytic signal, i.e. a signal that satisfies the KK relationship, is a complex-valued signal that has no negative frequency components, and its real and imaginary parts are related to each other by the Hilbert transform.

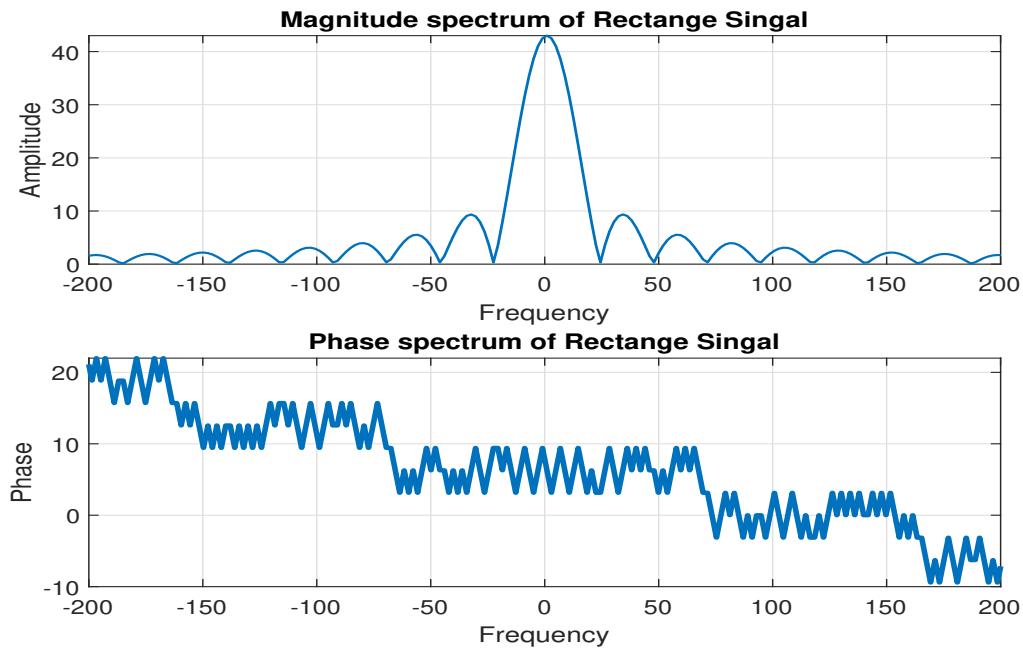
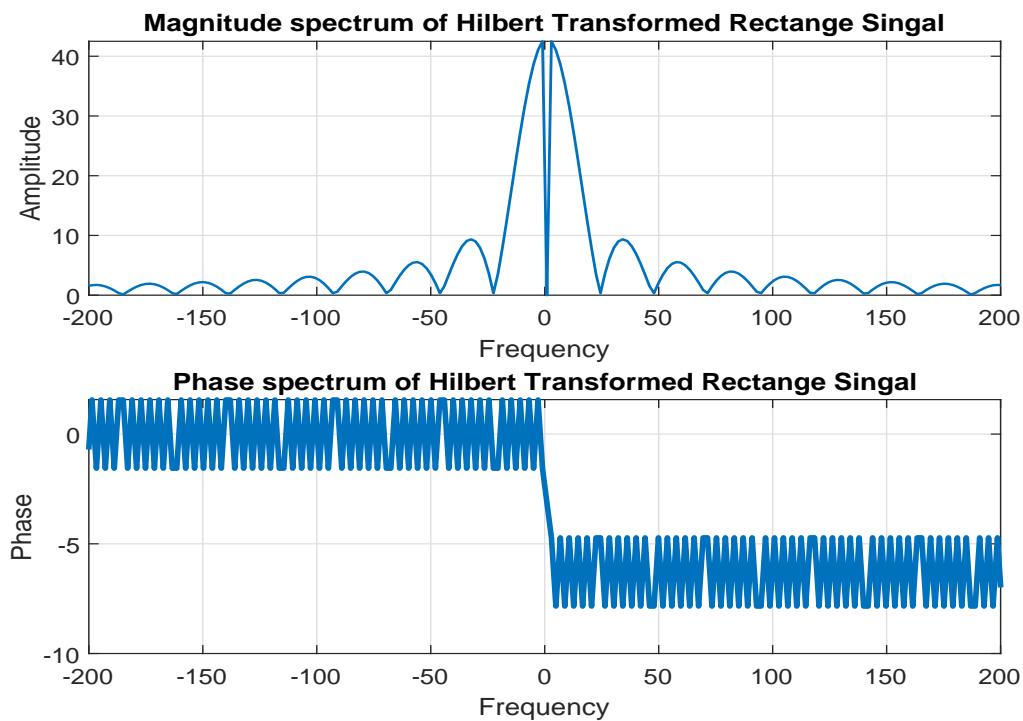
$$s_a(t) = s(t) + i\hat{s}(t) \quad (6.156)$$

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

$$s_a(t) = s_{a,r}(t) + i s_{a,i}(t) \quad (6.157)$$

then in the equation 6.157, the real and imaginary parts  $s_{a,r}(t)$  and  $s_{a,i}(t)$  are related through the Kramers-Kronig relation with each other. An intuitive way to analyze this relation is based on expressing its Fourier transform  $S_a(\omega)$  as follows,

$$S_a(\omega) = \frac{1}{2} [1 + \text{sgn}(\omega)] S_a(\omega) \quad (6.158)$$

Figure 6.248: Magnitude and phase spectrum of  $R(f)$ Figure 6.249: Magnitude and phase spectrum of  $\hat{R}(f)$

The equation 6.158 follows the SSB signal condition  $S_a(\omega) = 0$  for  $\omega < 0$ . Further, simplification of the signal can be summarized as follows:

$$\begin{aligned} S_a(\omega) &= \frac{1}{2}[1 + \text{sgn}(\omega)]S_a(\omega) \\ &= \frac{1}{2}S_a(\omega) + \frac{1}{2}\underline{\text{sgn}(\omega)S_a(\omega)} \end{aligned} \quad (6.159)$$

Taking inverse Fourier transform of the equation 6.159,

$$\begin{aligned} s_a(t) &= \mathcal{F}^{-1}\{S_a(\omega)\} \\ &= \frac{1}{2}s_a(t) + \frac{1}{2}\underline{\mathcal{F}^{-1}\{\text{sgn}(\omega)\} * s_a(t)} \end{aligned} \quad (6.160)$$

The underlined term in Equation 6.160 displays that multiplication in frequency domain converted into the convolution in the time domain. Further, IFT of the function  $\text{sgn}(\omega)$  given as  $(-i/\pi t)$ . As a consequence, we can further simplify our equation as,

$$\begin{aligned} s_a(t) &= \frac{1}{2}s_a(t) + \frac{1}{2}\left[\frac{i}{\pi t} * s_a(t)\right] \\ \frac{s_a(t)}{2} &= \frac{1}{2}\left[\frac{i}{\pi t} * s_a(t)\right] \\ s_a(t) &= i\left[\frac{1}{\pi t} * s_a(t)\right] \\ s_a(t) &= \frac{i}{\pi}p.v.\int_{-\infty}^{\infty} \frac{s_a(t')}{t-t'}dt' \end{aligned} \quad (6.161)$$

Using Equation 6.157 into Equation 6.161,

$$s_{s,r}(t) + i s_{a,i}(t) = \frac{i}{\pi}p.v.\int_{-\infty}^{\infty} \frac{s_a(t')}{t-t'}dt' \quad (6.162)$$

Therefore,

$$\begin{aligned} s_{s,r}(t) + i s_{a,i}(t) &= \frac{i}{\pi}p.v.\int_{-\infty}^{\infty} \frac{s_{a,r}(t') + i s_{a,r}(t')}{t-t'}dt' \\ s_{s,r}(t) + i s_{a,i}(t) &= -\frac{1}{\pi}p.v.\int_{-\infty}^{\infty} \frac{s_{a,i}(t')}{t-t'}dt' + \frac{i}{\pi}p.v.\int_{-\infty}^{\infty} \frac{s_{a,r}(t')}{t-t'}dt' \end{aligned} \quad (6.163)$$

which leads to,

$$\begin{aligned} s_{a,r}(t) &= -\frac{1}{\pi}p.v.\int_{-\infty}^{\infty} \frac{s_{a,i}(t')}{t-t'}dt' \\ s_{a,i}(t) &= \frac{1}{\pi}p.v.\int_{-\infty}^{\infty} \frac{s_{a,r}(t')}{t-t'}dt' \end{aligned} \quad (6.164)$$

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

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

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

$$s(t) = \frac{1}{2}[s(t) + i\hat{s}(t)] + \frac{1}{2}[s(t) - i\hat{s}(t)] \quad (6.165)$$

From Equation 9.15,

$$s(t) = s_a(t) + is_a^*(t) \quad (6.166)$$

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

$$\frac{1}{2}S_a(f) = \begin{cases} S(f) & \text{for } f > 0 \\ 0 & \text{for } f < 0 \end{cases} \quad (6.167)$$

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

$$F^{-1}\{S_a(f - f_0)\} = s_a(t)e^{i2\pi f_0 t} \quad (6.168)$$

Therefore, from the Euler's formula,

$$\begin{aligned} s_{ssb}(t) &= Re\{s_a(t)e^{i2\pi f_0 t}\} \\ &= Re\{[s(t) + i\hat{s}(t)][\cos(2\pi f_0 t) + i\sin(2\pi f_0 t)]\} \\ &= s(t)\cos(2\pi f_0 t) - \hat{s}(t)\sin(2\pi f_0 t) \end{aligned} \quad (6.169)$$

This Equation 6.169 displays the mathematical modeling of the upper sideband SSB signal. Similarly, we can generate lower sideband SSB signal by,

$$s_{ssb}(t) = s(t)\cos(2\pi f_0 t) + \hat{s}(t)\sin(2\pi f_0 t) \quad (6.170)$$

#### Discrete Hilbert transform

The equation 6.152 is extended from  $-\infty$  to  $\infty$  in the frequency domain. In the time domain the number of points must be limited (say, to  $2M + 1$ ), which is equivalent to adding rectangular window to the input signal. The Z-transform of windowed  $h(nt_s)$  is given as,

$$H(z) = \sum_{n=-M}^M h(nt_s)z^{-n} \quad (6.171)$$

By definition, the equation is not causal because summation starts from a negative value. In order to implement it into the practice, the equation 6.171 must be made causal. The FIR

design scheme can be used to achieve the discrete Hilbert transform.

**1:** Write the z transform of the function  $h(nt_s)$  as,

$$\begin{aligned} H(z) &= \sum_{n=-\infty}^{\infty} h(nt_s)z^{-n} \\ &= \sum_{n=\infty}^{-1} h(nt_s)z^{-n} + h(0) + \sum_{n=1}^{\infty} h(nt_s)z^{-n} \\ &= h(0) + \sum_{n=1}^{\infty} [h(-nt_s)z^n + h(nt_s)z^{-n}] \end{aligned} \quad (6.172)$$

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

$$\begin{aligned} H(e^{i2\pi ft_s}) &= H_r(e^{i2\pi ft_s}) + iH_i(e^{i2\pi ft_s}) \\ &= \sum_{n=-\infty}^{\infty} h(nt_s)e^{i2\pi nft_s} \\ &= h(0) + \sum_{n=1}^{\infty} [h(-nt_s)\cos(2\pi nft_s) + ih(-nt_s)\sin(2\pi nft_s) + \\ &\quad h(nt_s)\cos(2\pi nft_s) + ih(nt_s)\sin(2\pi nft_s)] \end{aligned} \quad (6.173)$$

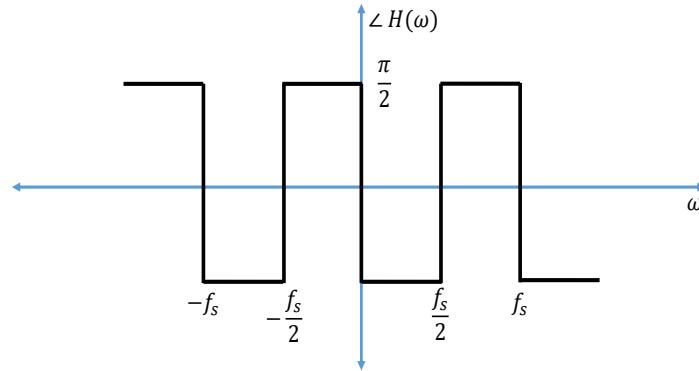
where,

$$\begin{aligned} H_r(e^{i2\pi ft_s}) &= h(0) + \sum_{n=1}^{\infty} [h(-nt_s) + h(nt_s)]\cos(2\pi nft_s) \\ H_i(e^{i2\pi ft_s}) &= \sum_{n=1}^{\infty} [h(-nt_s) + h(nt_s)]\sin(2\pi nft_s) \end{aligned} \quad (6.174)$$

**2:** When the sampling frequency is  $f_s$ , the transfer function  $H(f)$  is limited to the bandwidth  $\frac{f_s}{2}$ . Due to the periodic property of sampling, the Hilbert transfer function is actually as shown in Figure 7.16. This can be represented by Fourier series as,

$$H_i(e^{i2\pi ft_s}) = \sum_{n=1}^{\infty} b_n \sin(2\pi nft_s) \quad (6.175)$$

$$\begin{aligned} b_n &= \frac{2}{f_s} \int_{-f_s/2}^{f_s/2} H(e^{i2\pi nft_s}) \sin(2\pi nft_s) df \\ &= \frac{2}{f_s} \left[ \int_{-f_s/2}^0 \sin(2\pi nft_s) df + \int_0^{f_s/2} \sin(2\pi nft_s) df \right] \\ &= \frac{1}{n\pi} [-2 + 2\cos(n\pi)] \\ &= \begin{cases} 0 & n = \text{even} \\ \frac{-4}{n\pi} & n = \text{odd} \end{cases} \end{aligned} \quad (6.176)$$

Figure 6.250: Periodic representation of  $H(f)$ 

In the above equation, the relation of  $f_s t_s = 1$  is used.

3: The equation 6.152 has only the imaginary part, therefore,  $H_r(f) = 0$  and  $H_i(f) \neq 0$ . This condition can be fulfilled if,

$$h(n) = 0 \text{ and } h(-nt_s) = -h(nt_s) \quad (6.177)$$

Using this relationship,  $H_i$  in equation 6.174 can be written as

$$H_i(e^{j2\pi n f t_s}) = -2 \sum_{n=1}^{\infty} h(nt_s) \sin(2\pi n f t_s) \quad (6.178)$$

Comparing equations 6.178 and 6.175, we can obtain

$$h(nt_s) = -\frac{b_n}{2} \quad (6.179)$$

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

$$h(nt_s) = \begin{cases} 0 & n = \text{even} \\ \frac{2}{n\pi} & n = \text{odd} \end{cases}$$

$$h(-nt_s) = \begin{cases} 0 & n = \text{even} \\ -\frac{2}{n\pi} & n = \text{odd} \end{cases} \quad (6.180)$$

4: As mentioned before, the results obtained in the equations 6.180 are not causal. In order to make them causal, a simple shift in time domain can be used. The value of  $n$  in  $h(nt_s)$  is windowed from  $-M$  to  $M$  as shown in equation 6.171. The shift in time domain is equivalent to multiplying the result of equation 6.171 by  $Z^{-M}$  and substituting with  $k = n + M$ , the new results is,

$$H(z) = \sum_{n=-M}^M h(nt_s) Z^{-(n+M)} = \sum_{k=0}^{2M} h(kt_s - Mt_s) Z^{-k} \quad (6.181)$$

### Graphical explanation SSB signal generation

This section describes the generation of SSB signal using Hilbert transformation method (Phase Shift Method). Consider a message signal  $m(t)$  with its frequency domain spectrum  $|M(F)|$  as shown in Figure 6.251. From the Figure 6.251, we can see that both the side are scaled by factor '1' which means it represents the original signal.

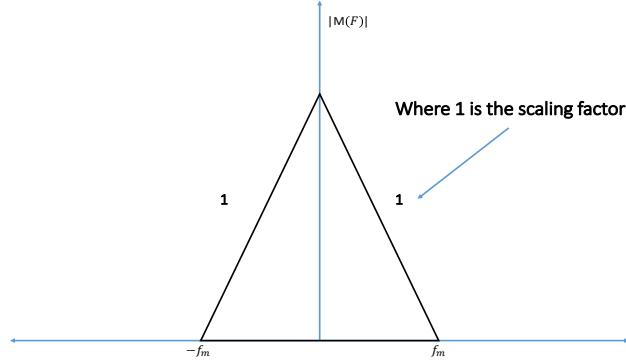


Figure 6.251: An original baseband signal

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

$$x(t) = m(t)\cos(2\pi f_c t) \quad (6.182)$$

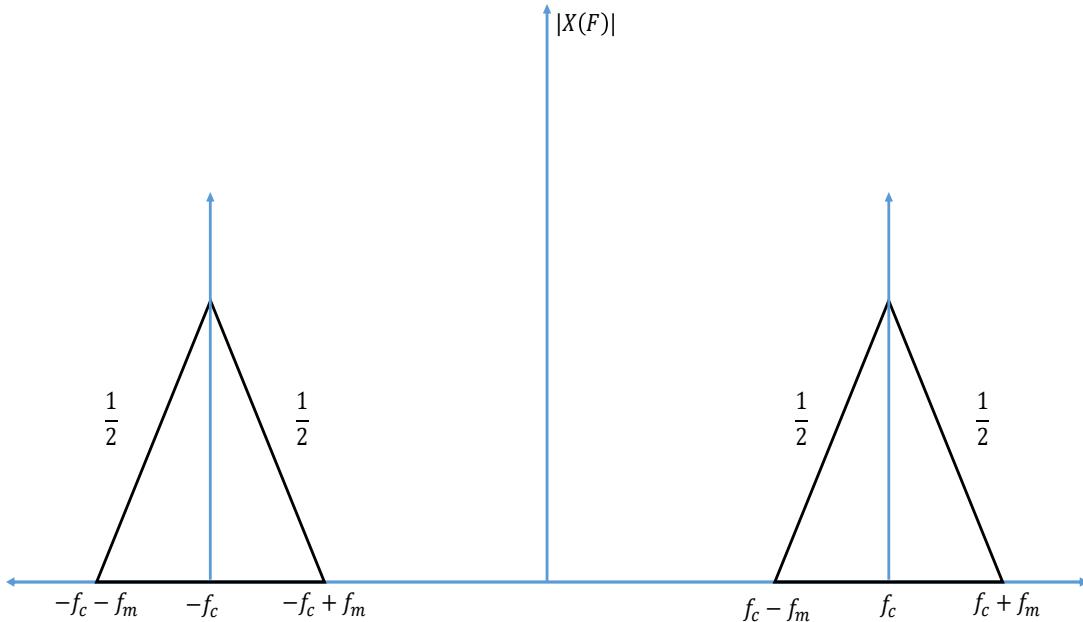
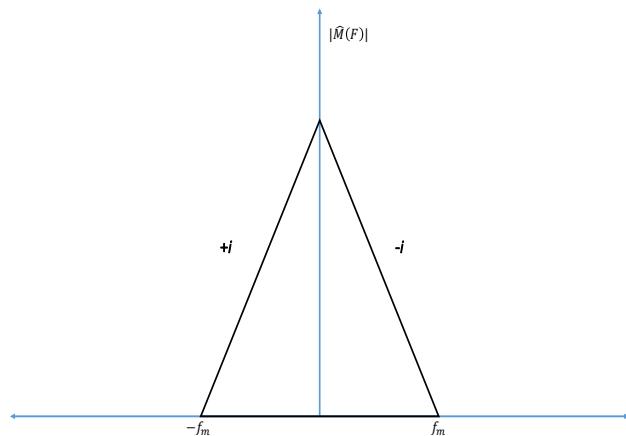
Frequency domain representation (see Figure 6.252) of the equation 6.182 can be given as,

$$X(F) = \frac{1}{2}M(f - f_c) + \frac{1}{2}M(f + f_c) \quad (6.183)$$

Here in equation 6.183, we can observe that each sideband is scaled by  $\frac{1}{2}$  on the frequency spectrum. The figure displays the frequency domain representation of the modulated signal  $X(F)$ .

Next, we will discuss something more interesting which is called as a Hilbert transform of the original message signal  $m(t)$ . As we discussed earlier, in the frequency domain, the Hilbert transformed signal  $\hat{M}(f)$  can be achieved by multiplying the Fourier transformed signal  $M(F)$  with  $[-i\text{sgn}(F)]$  (see Figure 6.253). Suppose we modulate the Hilbert transformed message signal  $\hat{m}(t)$  with the  $\sin(2\pi f_c t)$  (quadrature phase carrier), then we get the following results:

$$\begin{aligned} \hat{m}(t)\sin(2\pi f_c t) &= \hat{m}(t)\frac{e^{i2\pi f_c t} - e^{-i2\pi f_c t}}{2} \\ &= \hat{m}(t)\frac{e^{i2\pi f_c t}}{2} - \hat{m}(t)\frac{e^{-i2\pi f_c t}}{2} \\ &= \frac{\hat{M}(f - f_c)}{2i} - \frac{\hat{M}(f + f_c)}{2i} \\ &= \frac{-i}{2}\hat{M}(f - f_c) + \frac{i}{2}\hat{M}(f + f_c) \end{aligned} \quad (6.184)$$

Figure 6.252: Frequency spectrum of modulated signal  $X(F)$ Figure 6.253: Spectrum of Hilbert transformed message signal  $\hat{M}(F)$ 

The detailed explanation of the equation 6.184 has been given in the Figure 6.254 and 6.255. Figure 6.254 displays the spectrum of the  $\hat{M}(f + f_c)$  and  $\hat{M}(f - f_c)$  for the positive and negative frequencies respectively. The final equation resolution of equation displays that both positive and negative side of the spectrum multiplied with  $\frac{i}{2}$  and  $\frac{-i}{2}$  respectively. Finally, the spectrum of the signal  $\hat{m}(t)\sin(2\pi f_c t)$  can be given as Figure 6.255.

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

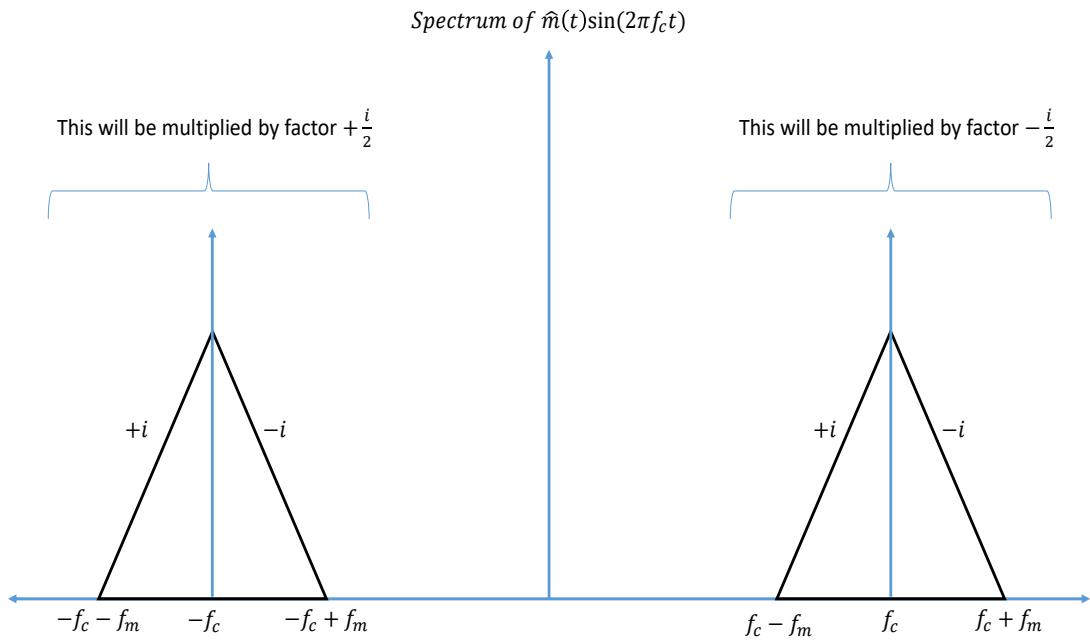


Figure 6.254: Hilbert transformed modulated signal

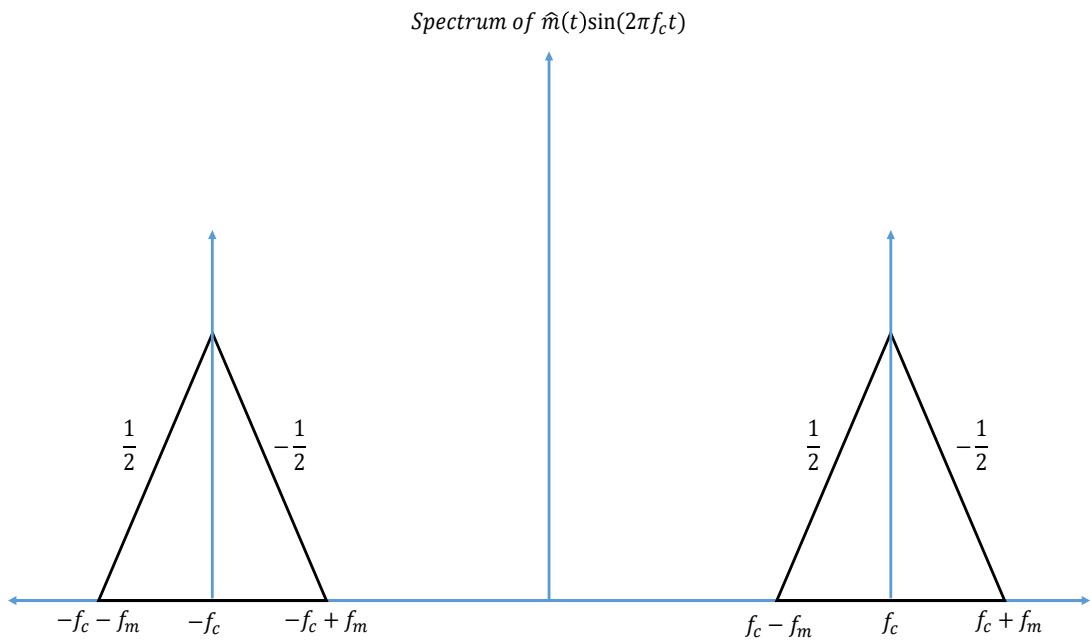


Figure 6.255: Hilbert transformed modulated signal

the upper sideband SSB signal as follows,

$$u(t) = m(t)\cos(2\pi f_c t) - \hat{m}(t)\sin(2\pi f_c t) \quad (6.185)$$

From the above discussion, the spectrum of the Equation 6.185 can be given by the Figure 6.256. Similarly, for the lower sideband SSB can be generated by Equation,

$$u(t) = m(t)\cos(2\pi f_c t) + \hat{m}(t)\sin(2\pi f_c t) \quad (6.186)$$

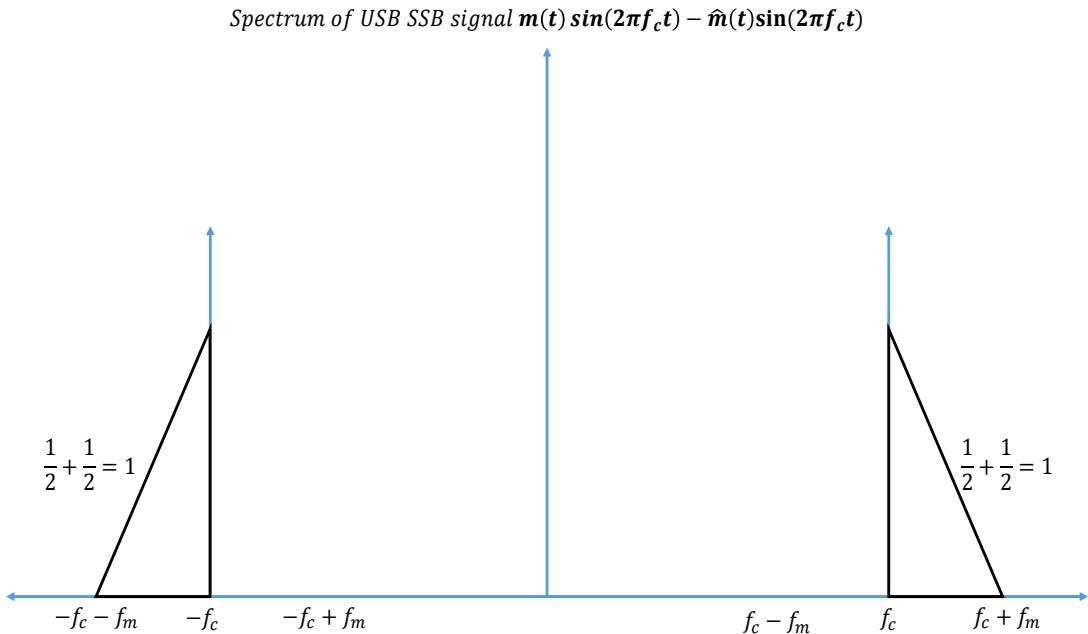


Figure 6.256: SSB signal spectrum

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

A necessary and sufficient condition for a complex signal  $E(t)$  to be minimum phase is that the curve described in a complex plane by  $A(t)$  when  $t \rightarrow -\infty$  to  $t \rightarrow \infty$  does not encircle the origin [4]. A minimum-phase signal has a useful property that the natural logarithm of the magnitude of the frequency response is related to the phase angle of the frequency response by the Hilbert transform.

For instance, if we consider a complex data-carrying signal  $E_s(t)$  with its optical bandwidth  $B$  and the LO is assumed to be a continuous-wave (CW) signal with its amplitude of  $E_o$  and the frequency which coincides with the left edge of the information-carrying signal spectrum and the constructed signal  $E(t)$  as,

$$E(t) = E_o \exp(i\pi Bt) + E_s(t) \quad (6.187)$$

where  $E_o$  is a constant; Here,  $E(t)$  is a minimum phase if and only if the winding number of its trajectory into the complex plane is zero i.e. the representation of the signal in the complex plane do not circle the origin. The condition  $|E_o| > |E_s(t)|$  is sufficient for guaranteeing minimum phase property [3]. Therefore, when  $E(t)$  is minimum phase signal, its phase  $\phi(t)$  can be reconstructed from its magnitude  $|E(t)|$  by means of using Kramers-Kronig relationship. In other words,  $E_o$  should be large enough to ensure that the signal presented by Equation 6.188 becomes a minimum phase signal.

$$E(t) \exp(-i\pi Bt) = E_o + E_s(t) \exp(i\pi Bt) \quad (6.188)$$

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

$$E_s(t) = \left[ \sqrt{I} \exp[i\phi_E(t)] - E_o \right] \exp(i\pi Bt) \quad (6.189)$$

$$\phi_E(t) = \frac{1}{2\pi} p.v. \int_{-\infty}^{\infty} dt' \frac{\log[I(t')]}{t - t'} \quad (6.190)$$

### 6.11.2 Numerical Validations

This section contains the numerical analysis of the Kramers-Kronig transceiver for both the complex and real valued signal. In the case I, the complex-valued signal is analyzed numerically as shown in Figure 6.257. It shows that first the generated QAM signal passed through the RRC (Root-Raised Cosine) filter and added a CW tone. This signal is then unconverted such that the CW tone coincide with the DC component, this up-converted signal called as minimum phase SSB signal. At the receiver end, the minimum phase signal is direct-detected using a single photo detector and the full complex signal recovered using the KK (Kramers-Kronig) algorithm.

#### Case I : 16-QAM signal

Consider a 16-QAM complex signal  $E_s(t) = I(t) + jQ(t)$  generated using the simulator as shown in Figure 6.258. The detail of the generated signal  $I(t)$  and  $Q(t)$  are given in the table.

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

The constellation diagram and the magnitude spectrum of the signal are shown in the Figure 6.259a and 6.259b respectively. The signal  $E_s(t)$  is confined within the bandwidth of  $\sim 10.5$  GHz. A continuous wave (CW) tone is required at the edge (either lower or higher) of the information bandwidth to make the signal to be a minimum phase signal. In this example, a

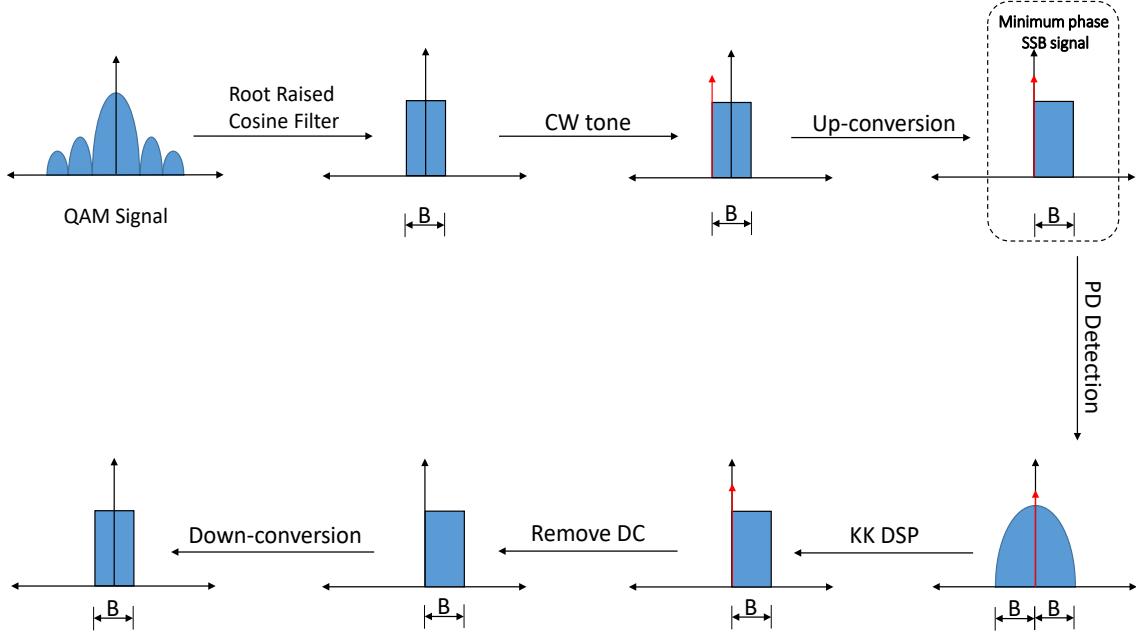


Figure 6.257: Generation and detection minimum phase SSB signal for QAM

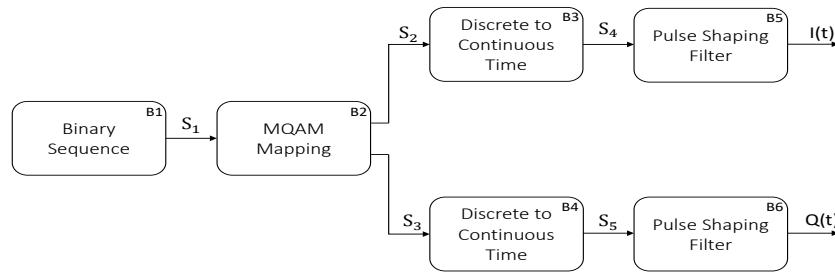


Figure 6.258: set-up to generate a 16-QAM modulation signal

CW tone has been added at the lower edge of the information bandwidth as shown In Figure 6.260b. The mathematical model of the CW tone added signal can be written as,

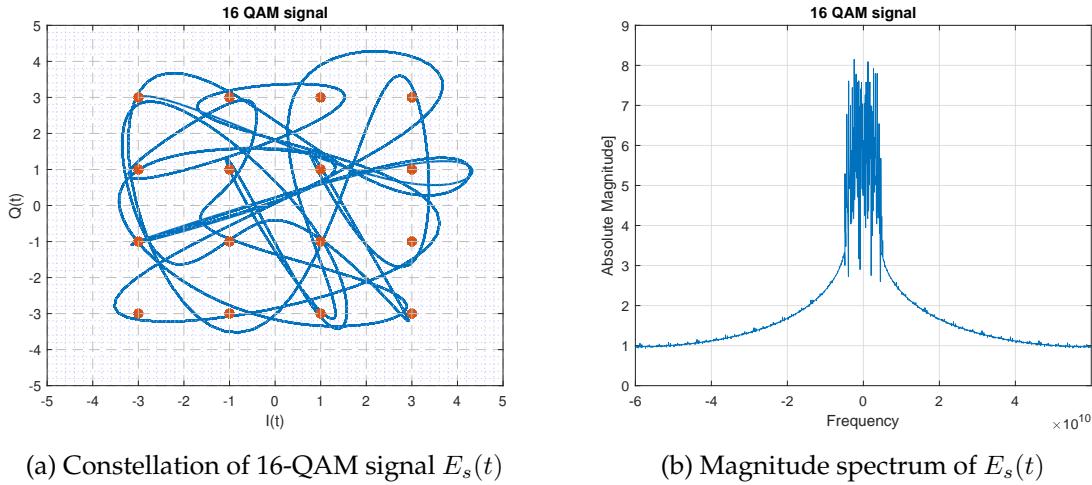
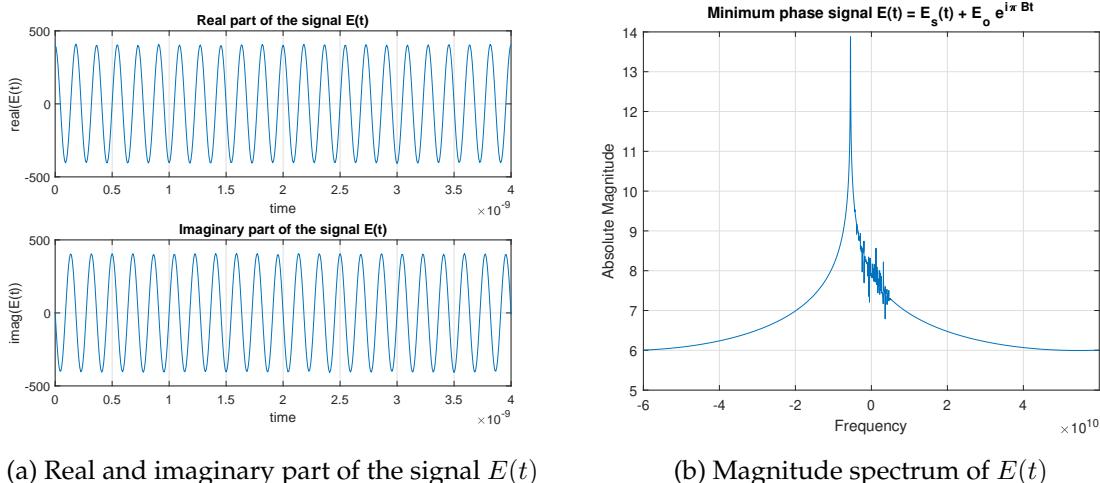
$$E(t) = E_s(t) + E_o e^{-i\pi Bt} \quad (6.191)$$

where the frequency of the CW is -5.5 GHz (half of the information bandwidth) and amplitude  $E_o \geq |E_s(t)|$  (It should be greater than 6dB of the information spectrum).

The frequency shifted version (frequency up-converted signal) of the minimum phase signal can be written in the following form.

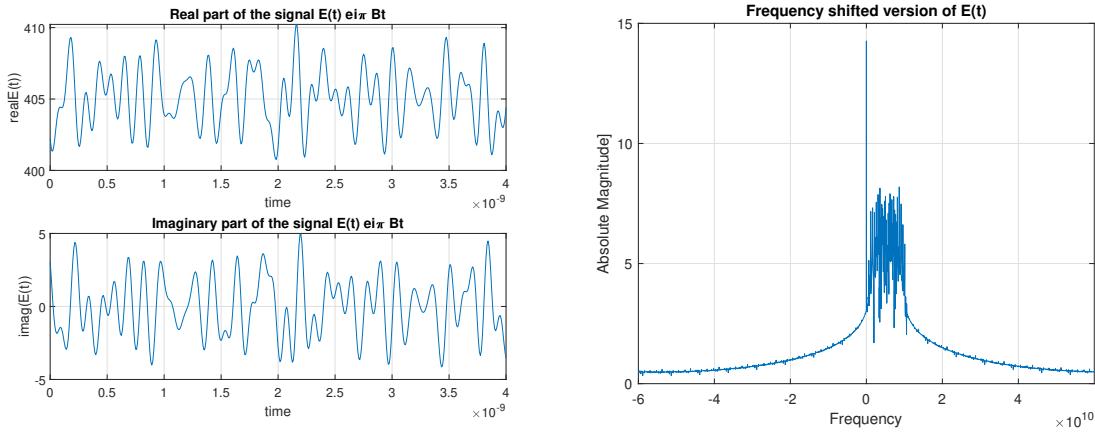
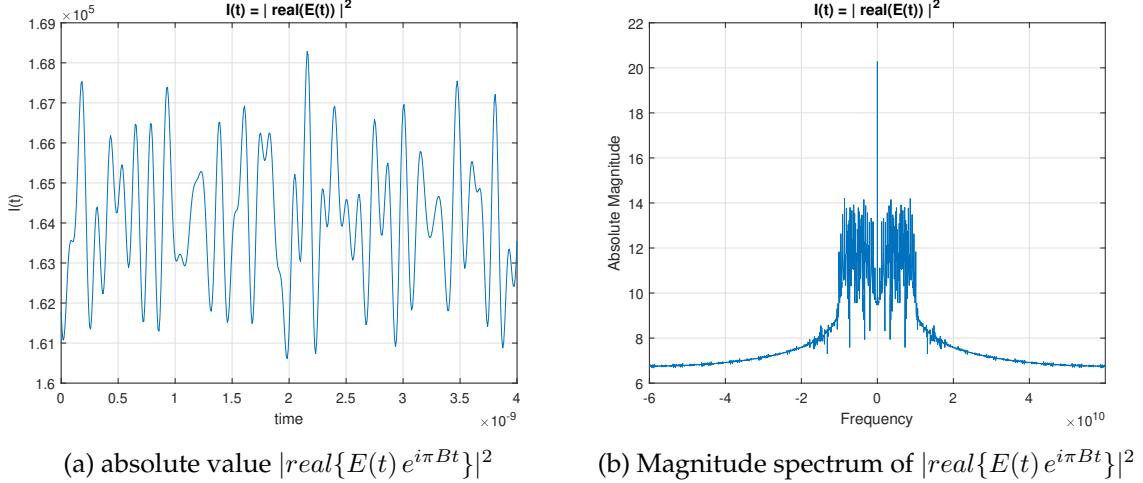
$$E(t) e^{i\pi Bt} = E_s(t) e^{i\pi Bt} + E_o \quad (6.192)$$

Figure 6.261 displays that the real part of the frequency shifted signal is non-negative. As long as this non-negativity of the real part is preserved, the signal can be fully recovered

Figure 6.259: 16-QAM signal  $E_s(t) = I(t) + iQ(t)$ Figure 6.260: 16-QAM minimum phase signal  $E(t) = E_s(t) + E_o e^{-i\pi Bt}$ 

from its intensity. Therefore, if we sample the optical signal with the sampling frequency equals to the twice of the information signal bandwidth then we can achieve the resulting spectrum similar to 6.261b and we can recover the full spectrum using its magnitude value.

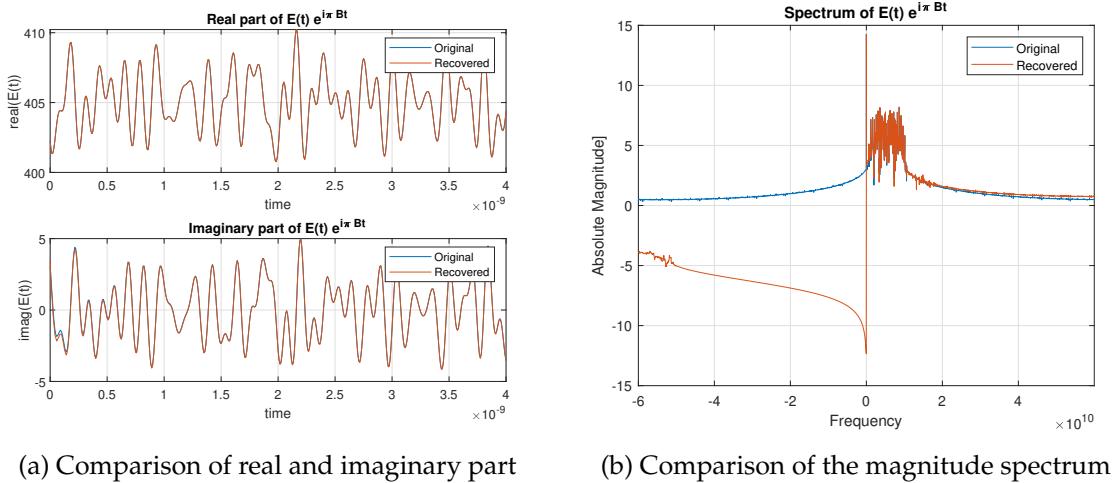
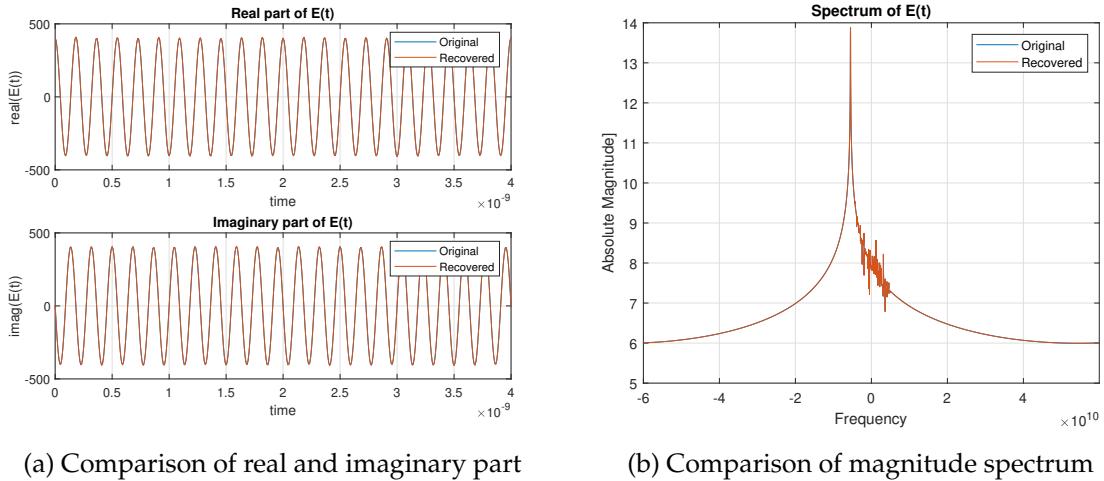
Next, if we take the square of the modulus of the signal as  $|real\{E(t)e^{i\pi Bt}\}|^2$  then the resultant output is as shown in Figure 6.262. By employing Kramers-Kronig algorithm on the  $|real\{E(t)e^{i\pi Bt}\}|^2$ , we can recover the full  $E(t)e^{i\pi Bt}$  as shown in Figure 6.263. Next, the signal is down-converted (see Figure 6.264) and then the CW tone has been removed (see Figure 6.265) to recover the original complex signal  $E_s(t) = I(t) + iQ(t)$ .

(a) Real and imaginary part of the signal  $E(t) e^{i\pi Bt}$ (b) Magnitude spectrum of  $E(t) e^{i\pi Bt}$ Figure 6.261: Frequency shifted version of  $E(t)$  as  $E(t) e^{i\pi Bt} = E_s(t) e^{i\pi Bt} + E_o$ (a) absolute value  $|\text{real}\{E(t) e^{i\pi Bt}\}|^2$ (b) Magnitude spectrum of  $|\text{real}\{E(t) e^{i\pi Bt}\}|^2$ Figure 6.262: Frequency shifted minimum phase signal  $E(t) e^{i\pi Bt}$ 

## Case II : PAM signal

In case of PAM signal, we can use the idea displayed in the Figure 6.266 to generate minimum phase SSB signal. The figure displays the idea of generating SSB signal using the Hilbert transformation method. The same thing can be achieved by using the SSBOF (Single Side Band Optical Filter) which makes use of the simplicity of amplitude modulation instead of using IQ modulator in the Hilbert filter method. Therefore, in the real-valued signal, a less complex configuration has been recently proposed [5] which facilitates to generating minimum phase SSB signal (see Figure 6.274).

Consider a 4-PAM real signal  $E_s(t) = I(t)$  generated using the simulator as shown in Figure 6.267. The detail of the signal  $I(t)$  is given in the following table.

Figure 6.263: Comparison between original and recovered  $E(t) e^{i\pi Bt}$ Figure 6.264: Comparison between original and recovered  $E(t) e^{i\pi Bt}$ 

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

The baseband signal  $I(t)$  and its magnitude spectrum are displayed as shown in Figure 6.268a and 6.268b respectively. The signal is confined within the bandwidth of approximately  $\sim 22$  GHz. A DC bias is applied to the signal  $E_s(t)$  to ensure the signal becomes non-negative. The mathematical model of the signal can be written as,

$$E_{nn}(t) = I(t) + E_o \quad (6.193)$$

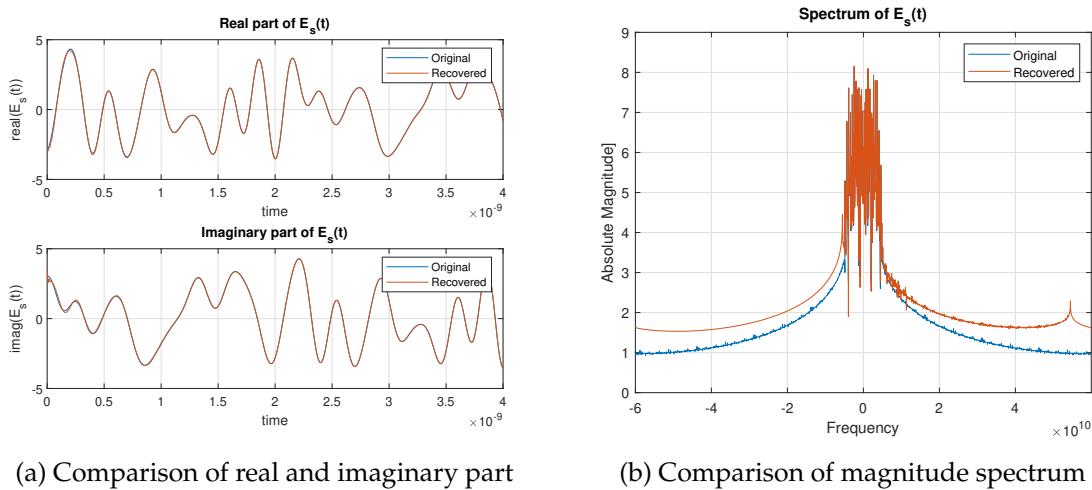
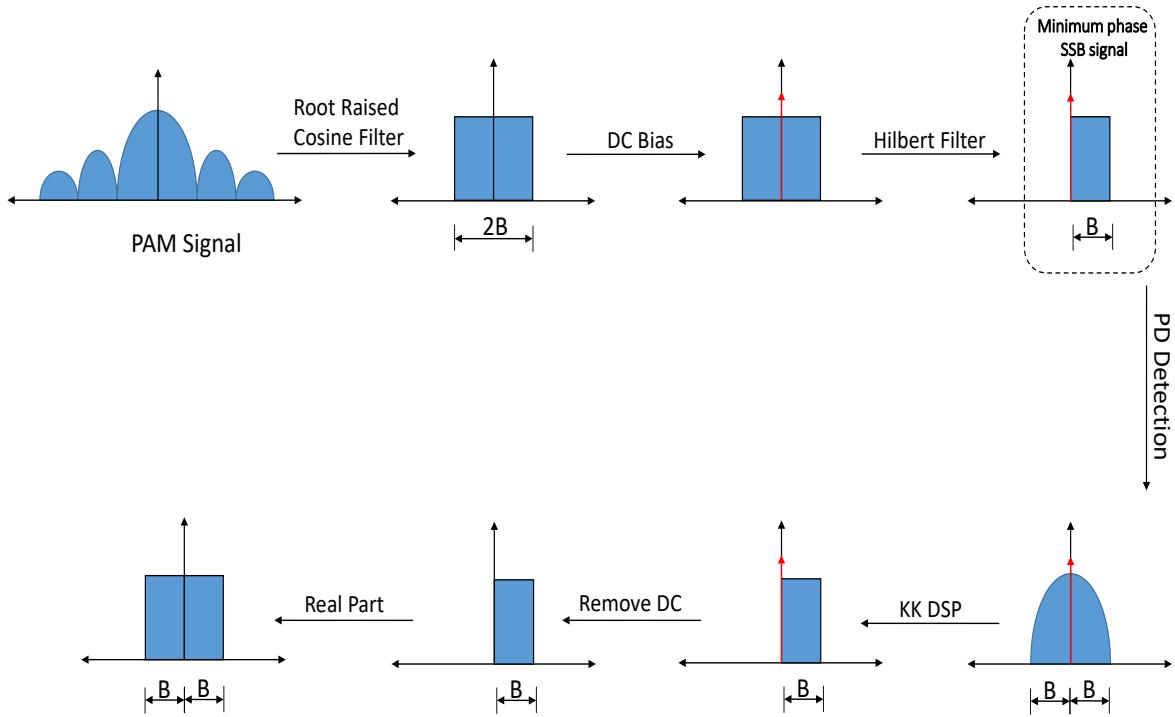
Figure 6.265: Comparison between original and recovered  $E_s(t)$ 

Figure 6.266: Generate minimum phase SSB of PAM signal

where  $E_o$  is the DC bias which should be large enough to ensure that the signal should become non-negative as shown in Figure 6.269. Next, this signal has been passed through the Hilbert filter which generates the minimum phase analytical SSB signal (Practically same thing can be achieved by using SSBOF) as shown in Figure 6.270. The mathematical model

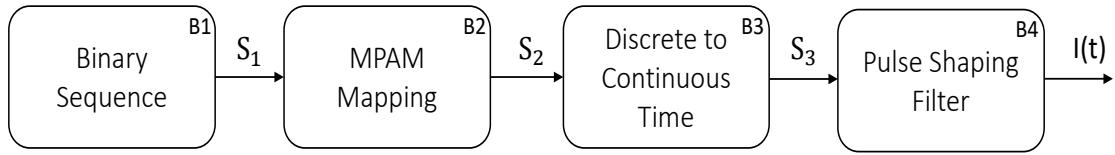
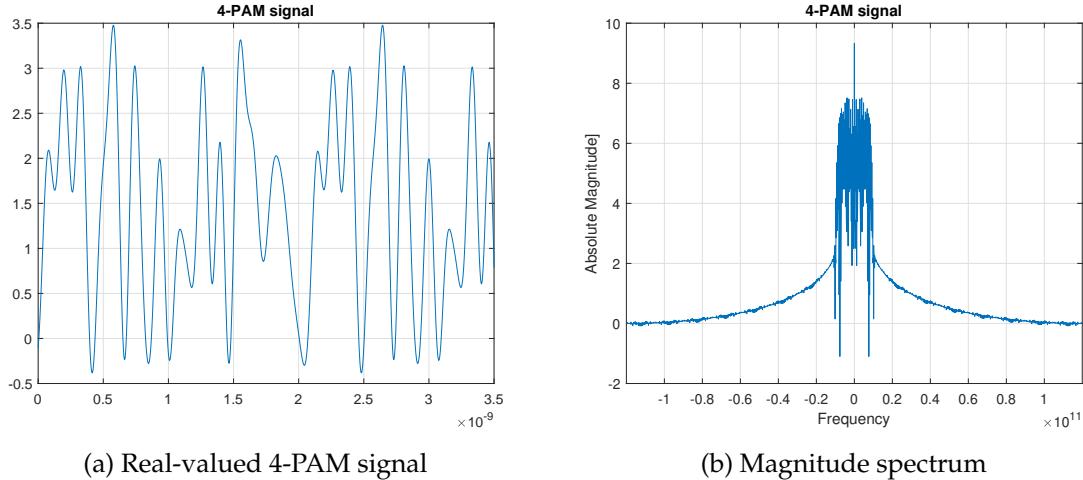


Figure 6.267: set-up to generate 4-PAM modulation signal

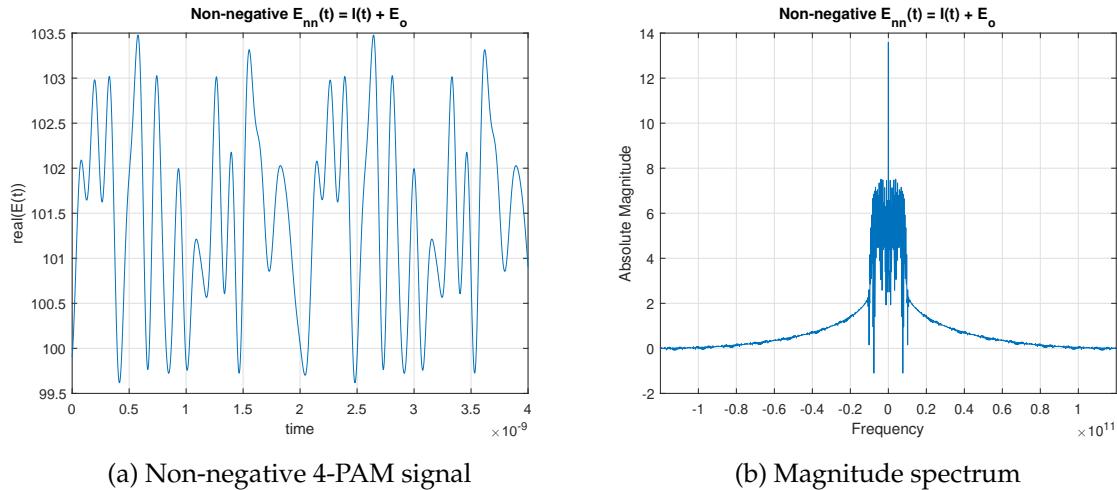
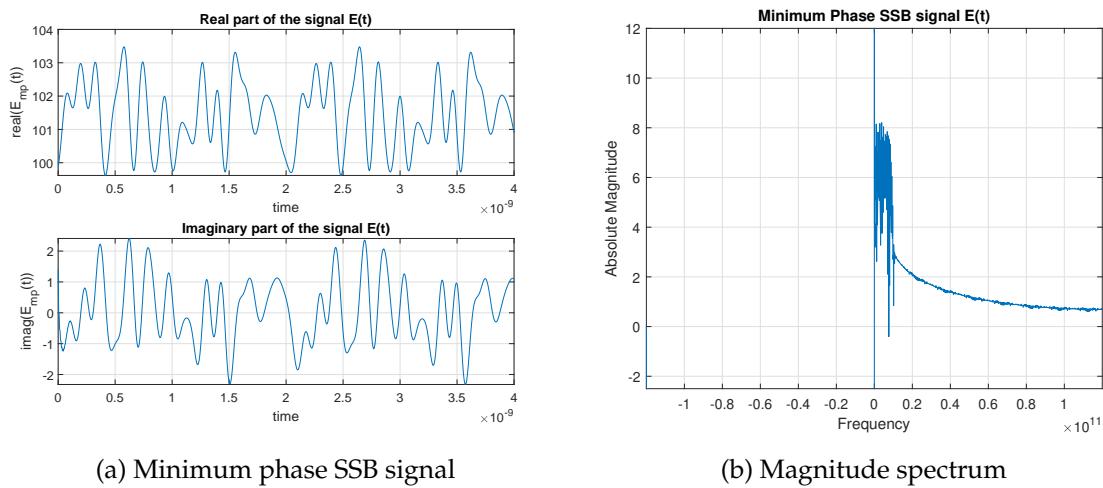
of the minimum phase SSB signal can be written as,

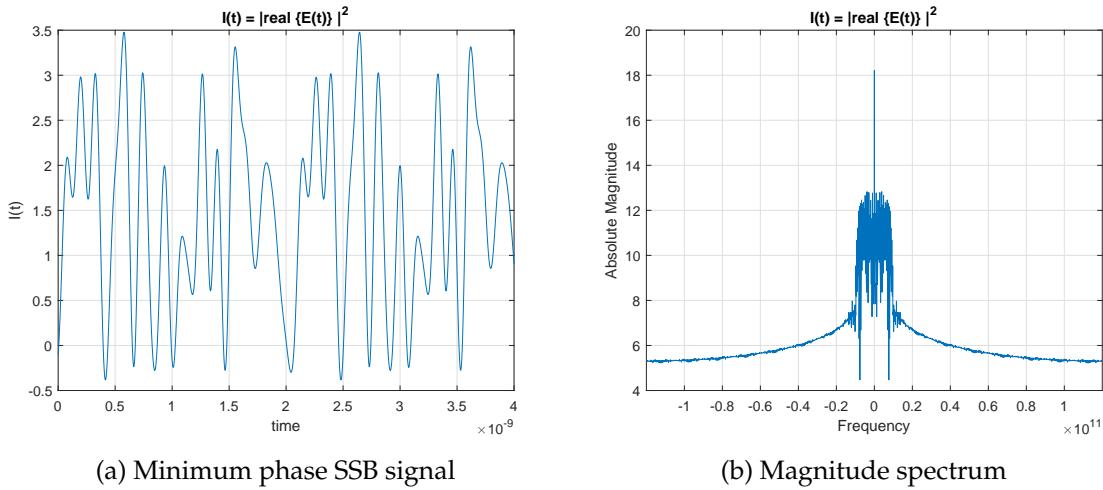
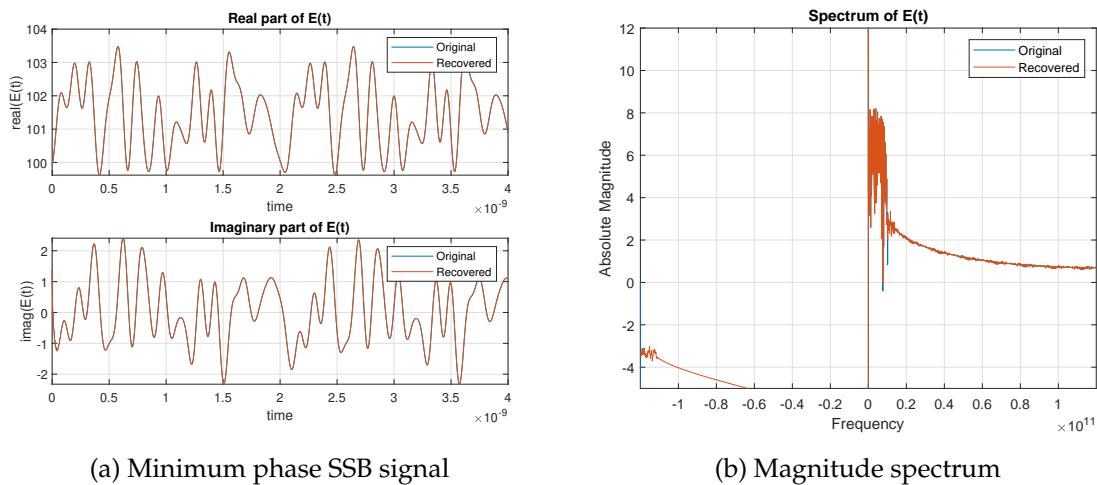
$$E(t) = E_s(t) + E_o \quad (6.194)$$

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

Figure 6.268: 4-PAM signal  $E_s(t) = I(t)$ 

Hilbert transformed of the signal  $I(t)$ . This SSB minimum phase signal  $E(t)$  is launched into the optical fiber for the transmission. At the receiver end, a photo detector will detect the intensity of the signal  $|real\{E(t)\}|^2$  (see Figure 6.271). By employing the Kramers-Kronig algorithm on the  $|real\{E(t)\}|^2$ , we can recover the full complex signal  $E(t)$  as shown in Figure 6.272. From the recovered complex signal, we can select only real part and removing the DC bias content, we can recover the signal  $I(t)$  as shown in Figure 6.273.

Figure 6.269: Non-negative real-valued information signal  $E(t)$ Figure 6.270: SSB minimum phase signal  $E(t)$

Figure 6.271: Detected SSB minimum phase signal  $I(t) = |\text{real}\{E(t)\}|^2$ Figure 6.272: Recovered non-negative signal  $E(t)$

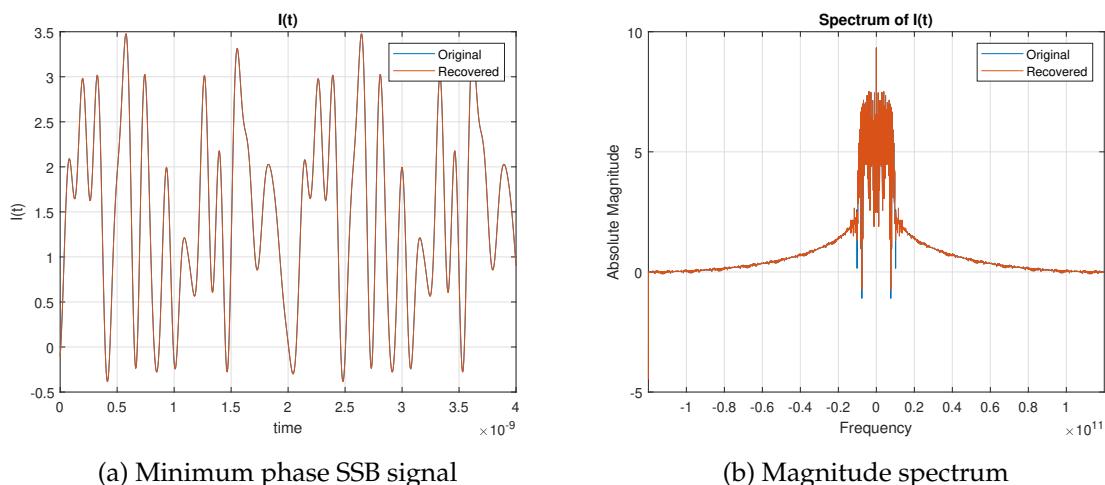


Figure 6.273: Comparison between original and recovered signal  $I(t)$

### Transceiver configurations of KK detection

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

#### 1. Transmitter configurations for real-valued signal

In the quest for the cost-effectiveness and highly performing transmission solutions, a simplified transmitter setup has been proposed which helps to transmit the real-valued signal for the KK detection [5]. The transmitter setup in configuration 1 is shown in the Figure 6.274 which consists of a laser followed by a single-drive MZM and a sharp optical filter (OF) suppressing one of the optical sidebands. The modulator is driven by an arbitrary waveform generator or we can feed any real-valued signal, i.e. an RF signal to it. The MZM bias level was set in the vicinity of the middle between the quadrature and null-points. The driving signal amplitude was adjusted so that it never crosses the MZM null-point, thus preventing the optical field from assuming negative values. This setup offers the reduced complexity and low cost transmitter configuration that relies on the amplitude modulation using a single drive MZM and a single-sideband optical filtering using SSBOF. However, it demands the stringent configurations of the SSBOF with a very steep roll-off of  $\sim 40\text{dB}/0.1\text{ nm}$ .

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

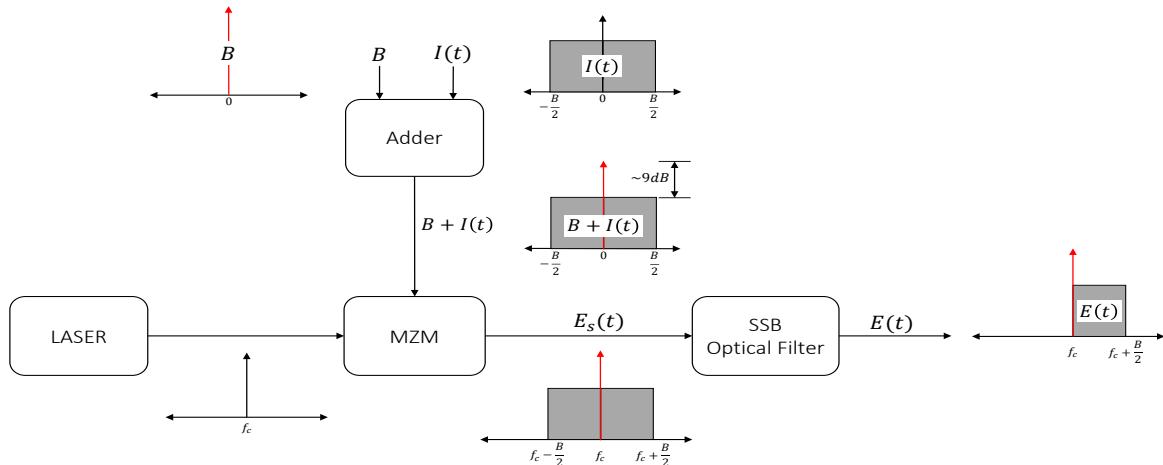


Figure 6.274: Transmitter setup for the real-valued signal : Configuration 1

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

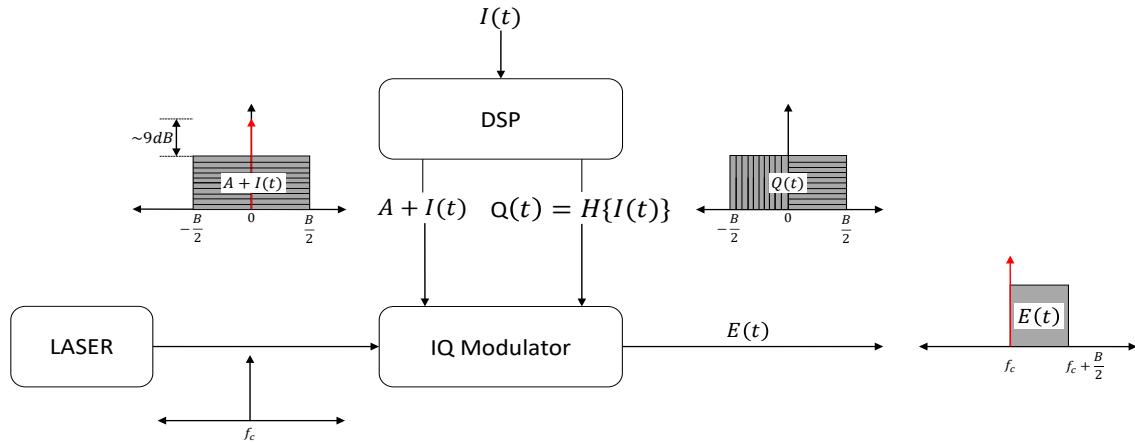


Figure 6.275: Transmitter setup for the real-valued signal : Configuration 2 ( $H\{\cdot\}$  represents the Hilbert transformation)

mentioning that the resultant SSB signal is a minimum phase because the real part  $A + I(t)$  is non-negative, where  $A$  is the DC bias which should be high enough to ensure the non-negativity of the signal. The quadrature part of the signal contains the Hilbert transformed version of the  $H\{I(t)\}$  as mentioned before in the discussion. It is important to mention that the Hilbert transform of  $H\{I(t)\}$  or  $H\{A + I(t)\}$  will lead to the same result since the Hilbert transform of any signal does not reflect the DC part in the resultant signal (see Equation 6.152).

## 2. Transmitter configurations for complex-valued signal

The various transmitter configurations for the KK detection scheme are shown in the Figure 6.276 and 6.277. In the configuration 1, the transmitter consists of a laser and an IQ modulator fed by two DACs for data generation and modulation. The CW tone can be generated optically. In order to generate a CW tone, we split a portion of unmodulated laser carrier and pass it through the frequency shifter ( $\Delta f$ ) and added it with the IQ modulated signal as shown in the Figure 6.276. The CW tone does not need to be synchronized with the signal in order work with the KK algorithm.

Since the method displayed in Figure 6.276 involves the complexity of optical domain, an alternative method (See Figure 6.277) was proposed where the CW tone added in the digital domain to get rid of the frequency shifter. In the proposed configuration 2, the CW tone with an amplitude  $E_o$  and its frequency approximately coincides with the lower-frequency edge of the information carrying bandwidth added in the digital domain to both  $I(t)$  and  $Q(t)$  channels. Next, these digital signals are converted into the analog form using DAC and supplied to the IQ modulator as shown in Figure 6.277. Generating and adding the tone digitally before DACs reduces the DAC resolution for the information bearing signal and affects both attainable interface rate and spectral efficiency [6]. The following table displays the main difference between all the transmitter configurations to generating the minimum

phase SSB signal:

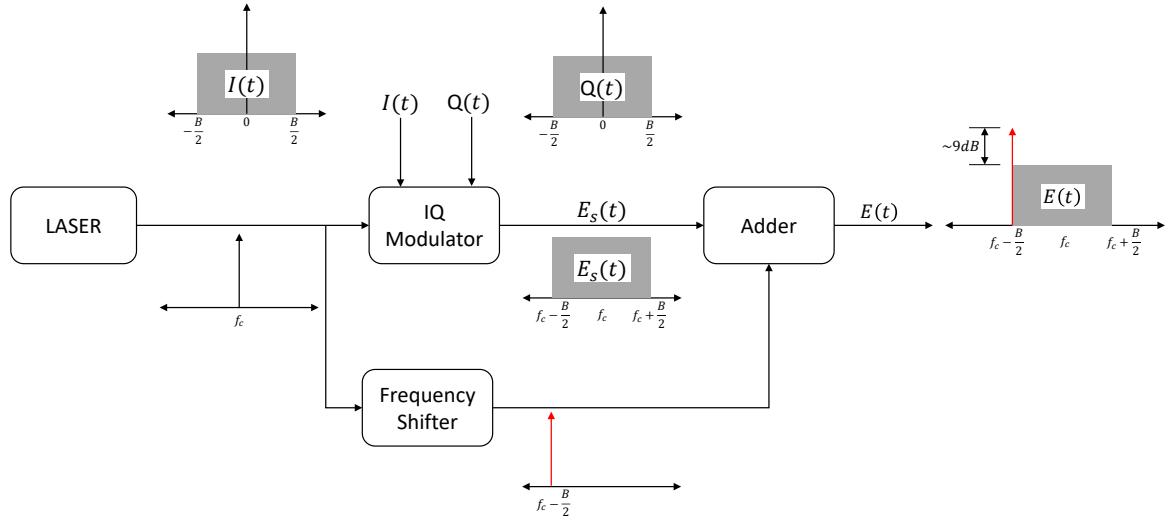


Figure 6.276: Transmitter setup : Configuration 3

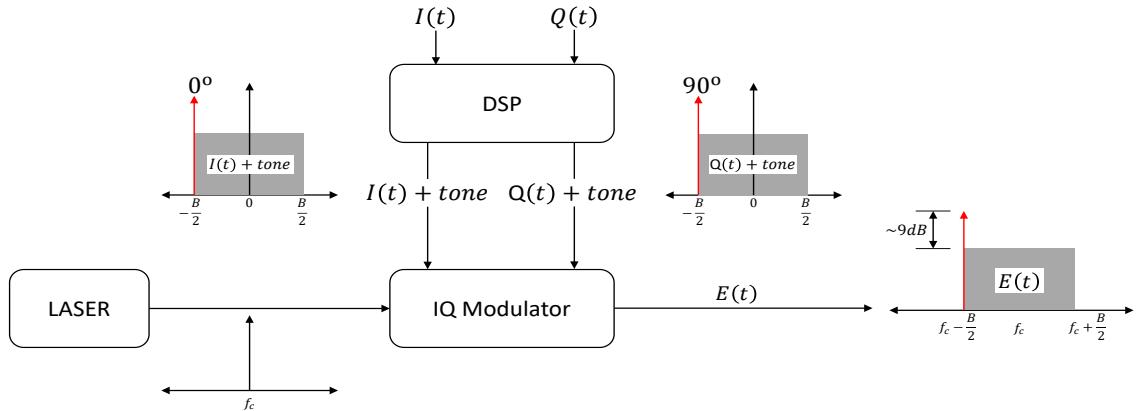


Figure 6.277: Transmitter setup : Configuration 4

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

### Receiver architecture for the KK detection

At the receiver end, the signal is detected by direct detection method and full complex envelope recovered by employing the Kramers-Kronig algorithm. The dotted box in the Figure 6.278 displays the DSP of the kramres-Kronig algorithm. First, the square root data of the photo-detected signal is calculated and then converted into the frequency domain after computing its logarithmic value. In the frequency domain, we can most conveniently apply the Hilbert transformation algorithm. Here,  $\text{sgn}(\omega)$  is the sign function, which is equal to 1 for  $\omega > 0$ , and 0 for the  $\omega = 0$  and -1 for  $\omega < 0$ . Finally, the Hilbert transformed signal is converted into the time domain and then its exponential value calculated to get the phase information of the detected signal. Finally, we can recover the full complex signal and applied to the post-processing section to recover the data.

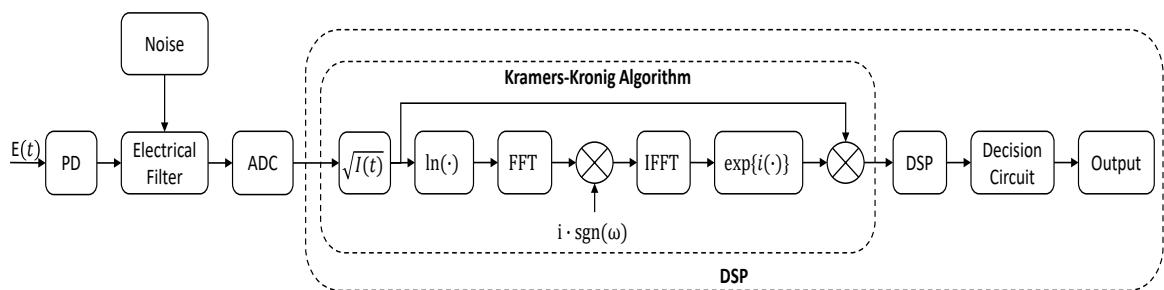


Figure 6.278: Receiver setup for the real-valued signal

### Upsampling free KK algorithm

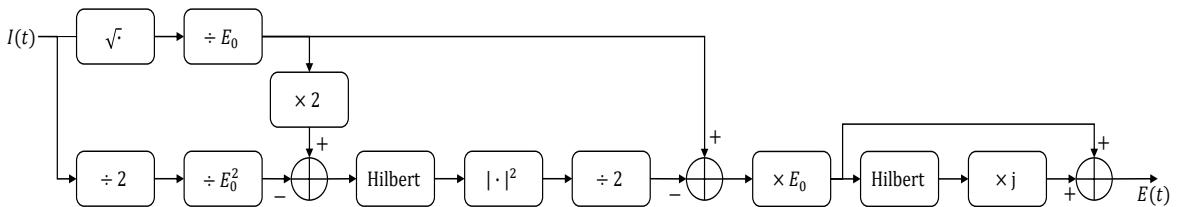


Figure 6.279: Up sampling free KK algorithm

### AM-AM and AM-PM analysis

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

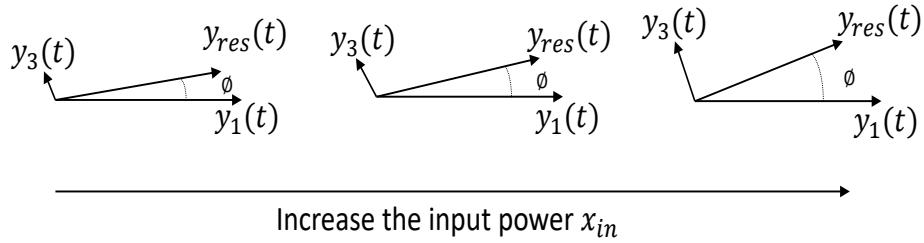


Figure 6.280: Illustration of AM-AM and AM-PM distortion

input signal (see Figure 6.280 ). where,  $y_1(t)$  is the linear component;  $y_3(t)$  is the third-order signal correlated distortion component;  $y_r(t)$  is the resultant output component; and  $\phi$  is the resultant output phase. The figure shows that addition of the signal correlated third-order components ( $y_3(t)$ ) to the linear components ( $y_1(t)$ ) constitutes a vector addition, which means that variation in input amplitude will produce changes in output amplitude, but also in output phase. Alternatively, we can also illustrate the AM-AM and AM-PM behavior of the nonlinear device as shown Figure 6.281.

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

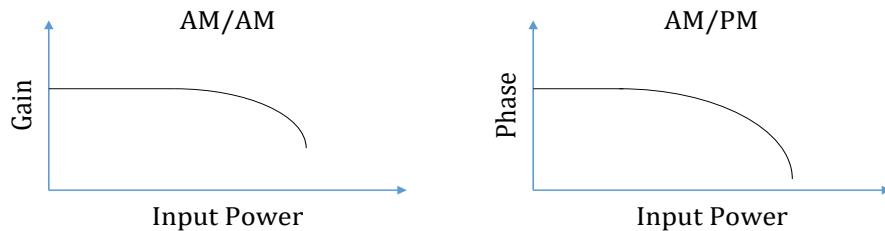


Figure 6.281: Illustration of AM-AM and AM-PM distortion

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

### Proposed Method

The proposed approach is based on the differential measurement between two branches, one composed by MZM and the other one made up by using linear element. This method uses the Wilkinson power divider at the input end, and a 90 degree hybrid coupler at the output end. The scattering matrix for the 90 degree hybrid coupler (see Figure 6.282) can be written as,

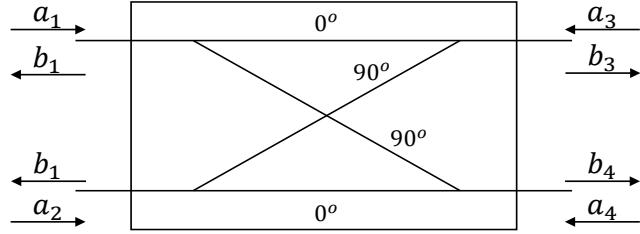


Figure 6.282: 90 degree hybrid coupler

$$S = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & -j & 1 \\ 0 & 0 & 1 & -j \\ -j & 1 & 0 & 0 \\ 1 & -j & 0 & 0 \end{bmatrix} \quad (6.195)$$

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

$$b_3 = -j \frac{|a_1|}{\sqrt{2}} + \frac{|a_2|}{\sqrt{2}} \cos(\phi) + j \frac{|a_2|}{\sqrt{2}} \sin(\phi) \quad (6.196)$$

$$b_4 = \frac{|a_1|}{\sqrt{2}} - j \frac{|a_2|}{\sqrt{2}} \cos(\phi) + \frac{|a_2|}{\sqrt{2}} \sin(\phi) \quad (6.197)$$

where  $\phi$  is the phase difference between  $a_1$  and  $a_2$ , if  $a_1$  is assumed to define a reference phase. From Equation 6.11.2, it is possible to obtain the power at the output ports of the coupler:

$$\begin{aligned} P_3 &= |b_3|^2 = \frac{|a_1|^2}{2} + \frac{|a_2|^2}{2} - |a_1||a_2| \sin(\phi) \\ &= \frac{P_1}{2} + \frac{P_2}{2} - \sqrt{P_1 P_2} \sin(\phi) \end{aligned} \quad (6.198)$$

$$\begin{aligned} P_4 &= |b_4|^2 = \frac{|a_1|^2}{2} + \frac{|a_2|^2}{2} + |a_1||a_2| \sin(\phi) \\ &= \frac{P_1}{2} + \frac{P_2}{2} + \sqrt{P_1 P_2} \sin(\phi) \end{aligned} \quad (6.199)$$

where,  $P_1 = |a_1|^2$  and  $P_2 = |a_2|^2$  are the incident power at the input port 1 and 2 respectively. The value of  $\phi$  can be extracted by subtracting Equation 6.11.2 and 6.11.2,

$$P_4 - P_3 = 2\sqrt{P_1 P_2} \sin(\phi) \quad (6.200)$$

and inverting 6.11.2,

$$\phi = \sin^{-1} \left( \frac{P_4 - P_3}{2\sqrt{P_1 P_2}} \right) \quad (6.201)$$

If we consider the  $90^\circ$  degree coupler is lossless then we can write,

$$P_2 = P_3 + P_4 - P_2 \quad (6.202)$$

which leads to

$$\phi = \sin^{-1} \left( \frac{P_4 - P_3}{2\sqrt{P_1(P_3 + P_4) - P_1^2}} \right) \quad (6.203)$$

### Measurement Setup

bla bla bla bla

### 6.11.3 Simulation Analysis

The transmitter setup includes various blocks to generating the minimum phase SSB signal. The presented block B1 randomly generates the sequence of binary numbers 0 and 1 as a  $S_1$  signal. The generated binary sequence  $S_1$  was 16-QAM mapped which outputs two discrete data signals  $S_2$  and  $S_3$ , which are converted into the continuous time form using the B3 and B4 blocks, respectively. The continuous time signals  $S_4$  and  $S_5$  are passed through the pulse shaping filter blocks B5 and B6 respectively to get rid of inter-symbol-interference in the communication system. The output of the pulse shaping filter blocks are represented by signal  $S_6$  and  $S_7$ .

The Real to complex block accepts two real signal to generate the complex output. Here, we have supplied  $S_6$  and  $S_7$  signal to generate a complex signal  $S_8$ . The oscillator block B8 will generate complex CW tone whose frequency coincides at the left edge of the information signal ( $S_9$ ) spectrum. The signals  $S_8$  and  $S_9$  are supplied to the B9 adder blocks which outputs the signal  $S_{10}$ . The signal  $S_{10}$  is then converted into minimum phase signal by mixer block B11. The mixer block will accept the CW tone added information signal  $S_{10}$  and the complex LO signal  $S_{11}$  and generates the minimum phase signal  $S_{12}$ . The output of the mixer blocks B11 supplied to the ideal IQ modulator block B12 which outputs the signal  $S_{13}$  same as an input signal  $S_{12}$ .

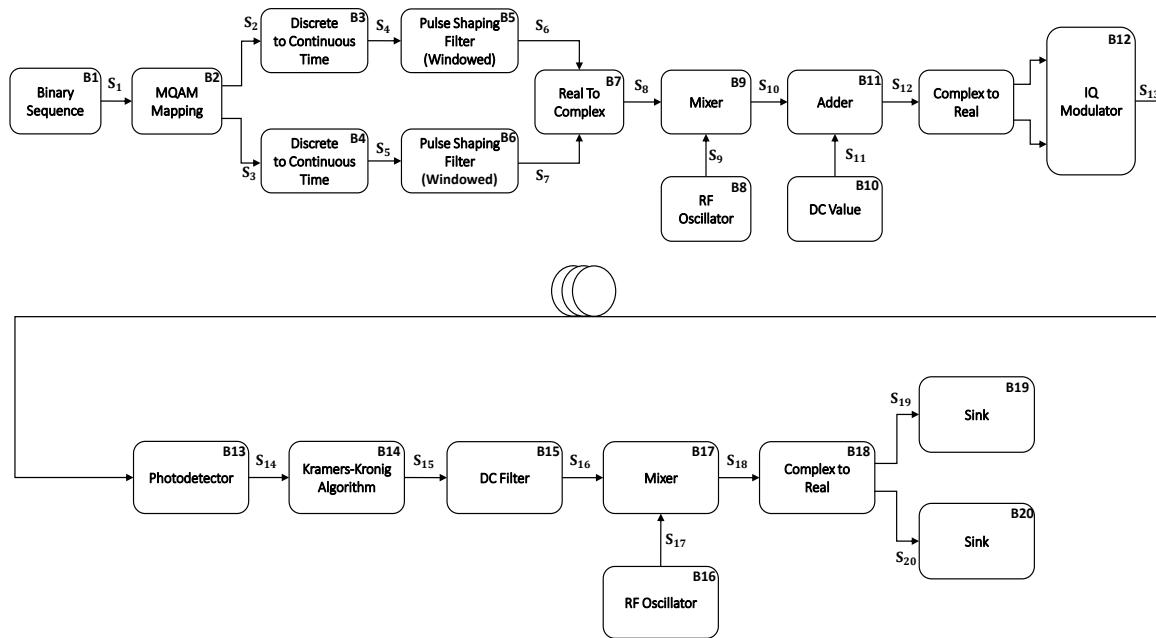


Figure 6.283: Transceiver simulation setup

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

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

## Signal analysis

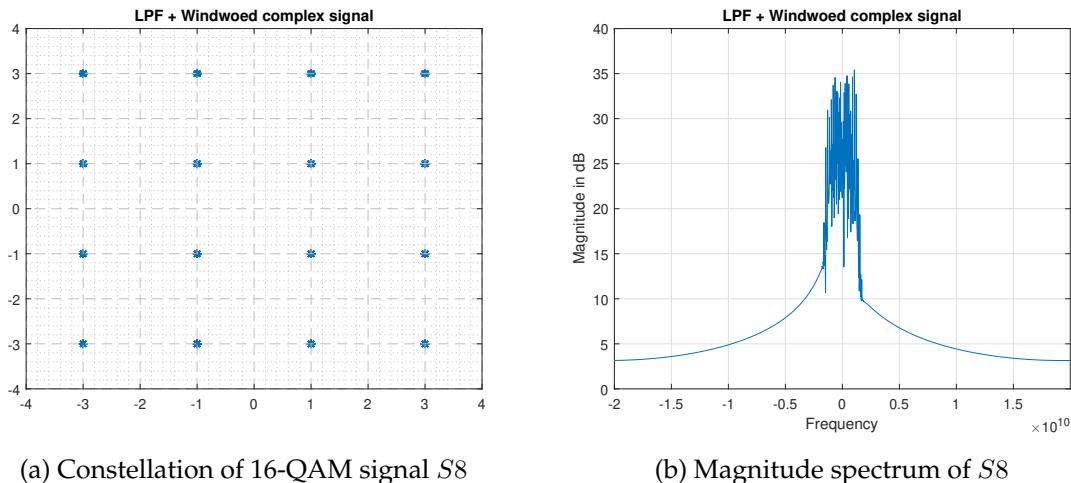


Figure 6.284: 16-QAM signal  $S_8$

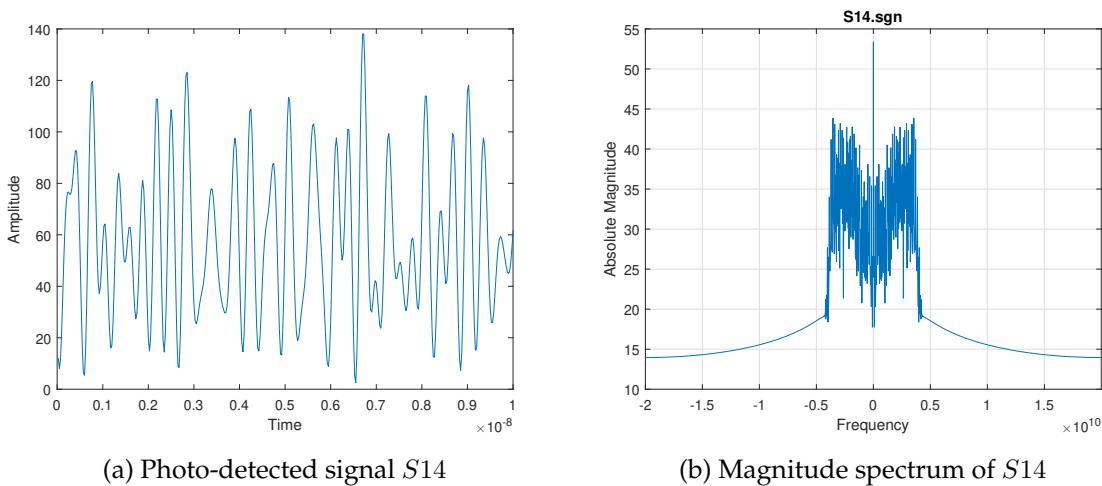
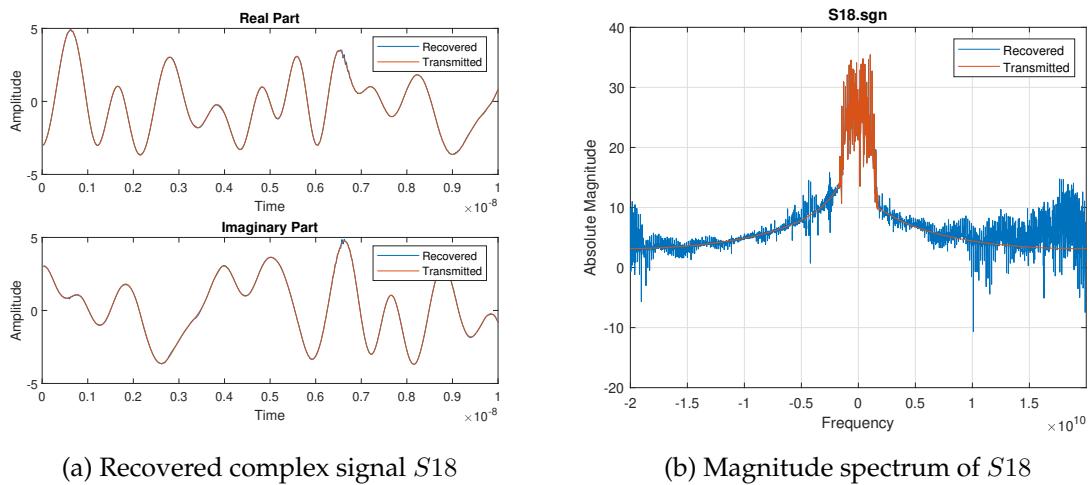
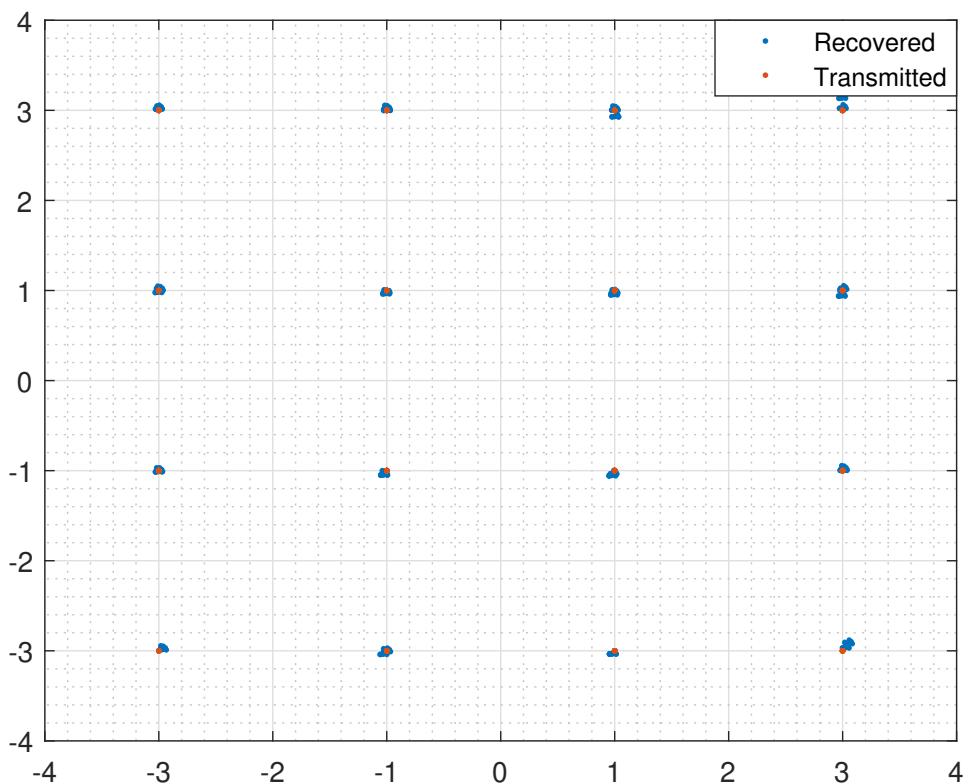


Figure 6.285: Photo-detected signal  $S_{14}$

Figure 6.286: Recovered signal  $S18$ Figure 6.287: Constellation of the recovered ( $S18$ ) and transmitted ( $S8$ ) complex signal

#### 6.11.4 Iterative method

Under certain conditions, a signal can be reconstructed from a partial information (phase or magnitude) in the time or frequency domain. A minimum or maximum phase signal, in particular, can be recovered from the phase or magnitude of its Fourier transform. Conventionally, the signal reconstruction algorithm involves applying Hilbert transform to the log-magnitude or phase to obtain the unknown component.

As discussed earlier, the conventional approach includes the nonlinear operations which demand upsampling of the signal and very high CW tone power yielding the requirement of high speed DSP and receiver sensitivity penalty, respectively. To overcome these problems, an alternative iterative algorithm can be used for reconstructing a minimum phase signal from the magnitude or phase data. Here, we begin with discussion of a number of equivalent conditions for a sequence to be minimum phase and then utilize these condition to develop iterative reconstruction algorithm.

#### The minimum phase condition

If we restrict Z-transform of a sequence of the sequence  $h(n)$  to be a rational function of the form as,

$$H(z) = Az^{n_o} \frac{\prod_{k=1}^{M_i} (1 - a_k z^{-1}) \prod_{k=1}^{M_o} (1 - b_k z)}{\prod_{k=1}^{P_o} (1 - c_k z^{-1}) \prod_{k=1}^{P_o} (1 - d_k z)} \quad (6.204)$$

where  $|a_k|$ ,  $|b_k|$ ,  $|c_k|$  and  $|d_k|$  are less than or equal to unity,  $z^{n_o}$  is a linear phase factor, and  $A$  is a scale factor. When  $h(n)$  is stable (i.e.  $\sum_n h(n) < \infty$ ),  $|c_k|$  and  $|d_k|$  are strictly less than one.

A complex function  $H(z)$  of a complex variable  $z$  is defined to be minimum phase if it and its reciprocal  $H^{-1}(z)$  are both analytic for  $|z| \geq 1$ . A minimum phase sequence is then defined as a sequence whose z-transform is minimum phase. The minimum phase condition excludes poles or zeros on or outside the unit circle in the z-plane or at infinity. The minimum phase condition excludes poles and zeros on or outside the unit circle in the z-plane and at infinity. Therefore, the factors  $(1 - b_k z)$  corresponding to zeros on or outside the unit circle and the factors of the form  $(1 - d_k z)$  corresponding to poles on or outside the unit circle will not present. Moreover,  $n_o = 0$  will exclude the poles or zeros at infinity. Thus, for the minimum phase  $H(z)$ , Equation 6.204 reduces to,

$$H(z) = A \frac{\prod_{k=1}^{M_i} (1 - a_k z^{-1})}{\prod_{k=1}^{P_o} (1 - c_k z^{-1})} \quad (6.205)$$

where,  $a_k$  and  $c_k$  are both strictly less than unity. From Equations 6.204, the two conditions can be formulated for a signal to be a minimum phase. Later, these conditions will be used in developing the iterative algorithm.

**Minimum phase condition A:** Consider a stable  $h(n)$  and  $H(z)$  rational in the form of Equation 6.204 with no zeros on the unit circle. A necessary and sufficient condition for  $h(n)$  to be minimum phase is that  $h(n)$  be causal, i.e.  $h(n) = 0$  for  $n < 0$ , and  $n_o$  in Equation 6.204 be zero.

From the Equation 6.205, it seems that these conditions are necessary. To show that they are sufficient, we show that they force Equation 6.204 to Equation 6.205. Clearly, factors of the form  $(1 - d_k z)$  with  $|d_k| < 1$  in the denominator will introduce poles outside the unit circle which would violate the causality condition. Also, with  $n_o = 0$ , factors  $(1 - b_k z)$  would introduce the positive powers of  $z$  in the Laurent expansion of  $H(z)$ , requiring  $h(n)$  to have some non-zero values for negative values of  $n$ , which again violates the causality condition. Therefore, these factors cannot be present and with  $n_o = 0$ , Equation 6.204 reduces to Equation 6.205. Finally, because our conditions assumes  $h(n)$  is stable and that  $H(z)$  has no zeros on the unit circle,  $h(n)$  is minimum phase.

**Minimum phase condition B:** Consider a stable  $h(n)$  and  $H(z)$  rational in the form of Equation 6.204 with no zeros on the unit circle. A necessary and sufficient condition for  $h(n)$  to be minimum phase is that  $h(n) = 0$  for  $n < 0$  and  $h(0) = A$  where  $A$  is the scaling factor in Equation 6.204.

Again from Equation 6.205, it follows that these conditions are necessary since Equation 6.205 has no poles or zeros outside the unity circle and infinity, guaranteeing causality, and from the initial value theorem

$$h(0) = \lim_{z \rightarrow \infty} H(z) = A$$

. To demonstrate that provided conditions are sufficient we note that causality of  $h(n)$  will eliminate factors of the form  $(1 - d_k z)$  in the denominator of Equation 6.204. Furthermore, since the conditions require that  $h(n)$  be causal, the initial value theorem can be applied with the result that,

$$h(0) = \lim_{z \rightarrow \infty} H(z) = \lim_{z \rightarrow \infty} Az^{n_o} \prod_{k=1}^{M_o} (1 - b_k z)$$

since  $h(0) = A$ ,

$$\lim_{z \rightarrow \infty} z^{n_o} \prod_{k=1}^{M_o} (1 - b_k z) = 1$$

and since  $|b_k| < 1$ , it requires that  $n_o = 0$  and the  $b_k$  equals to zero and thus Equation 6.204 reduces to Equation 6.205.

### Iterative algorithm of signal reconstruction from its magnitude

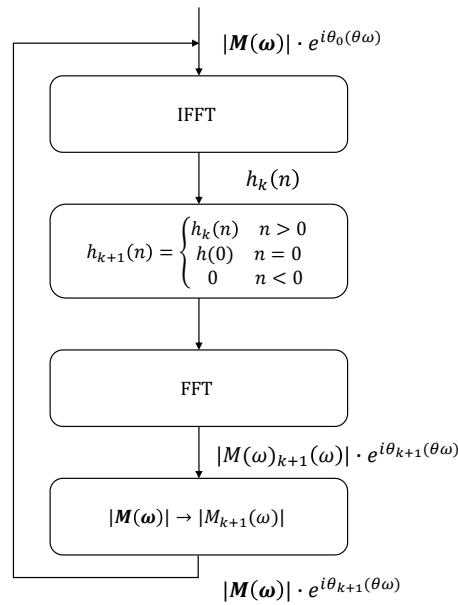


Figure 6.288: Iterative method for frequency domain magnitude data

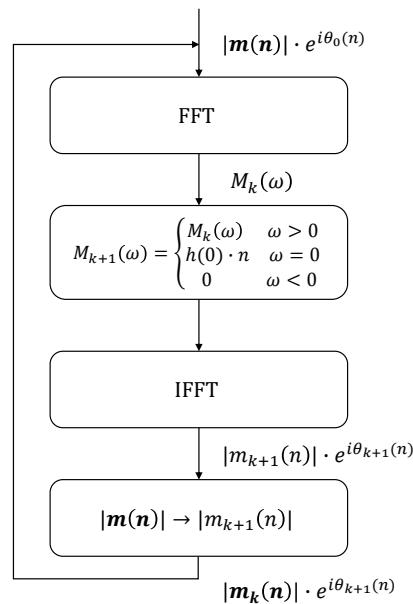


Figure 6.289: Iterative method for time domain magnitude data

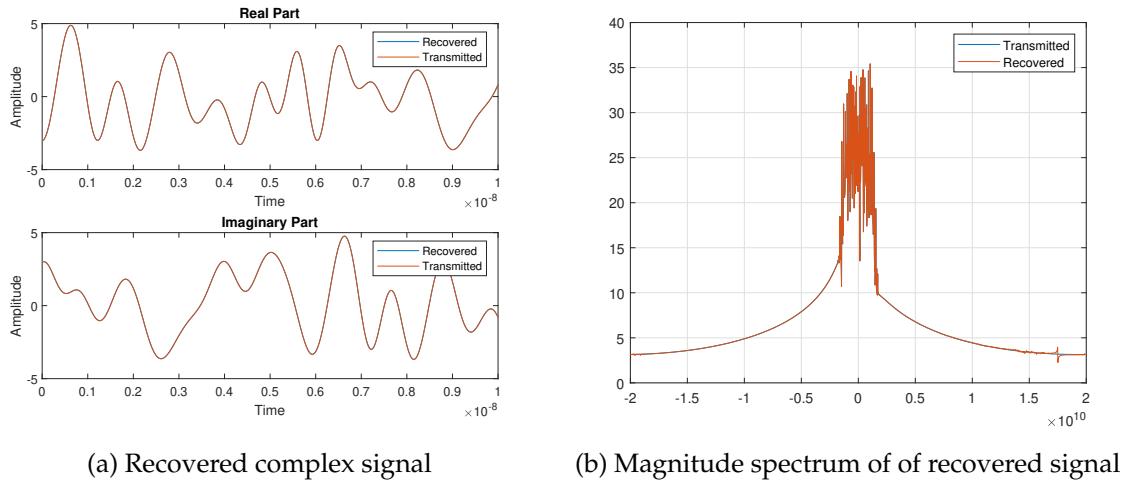


Figure 6.290: Recovered signal using iterative method

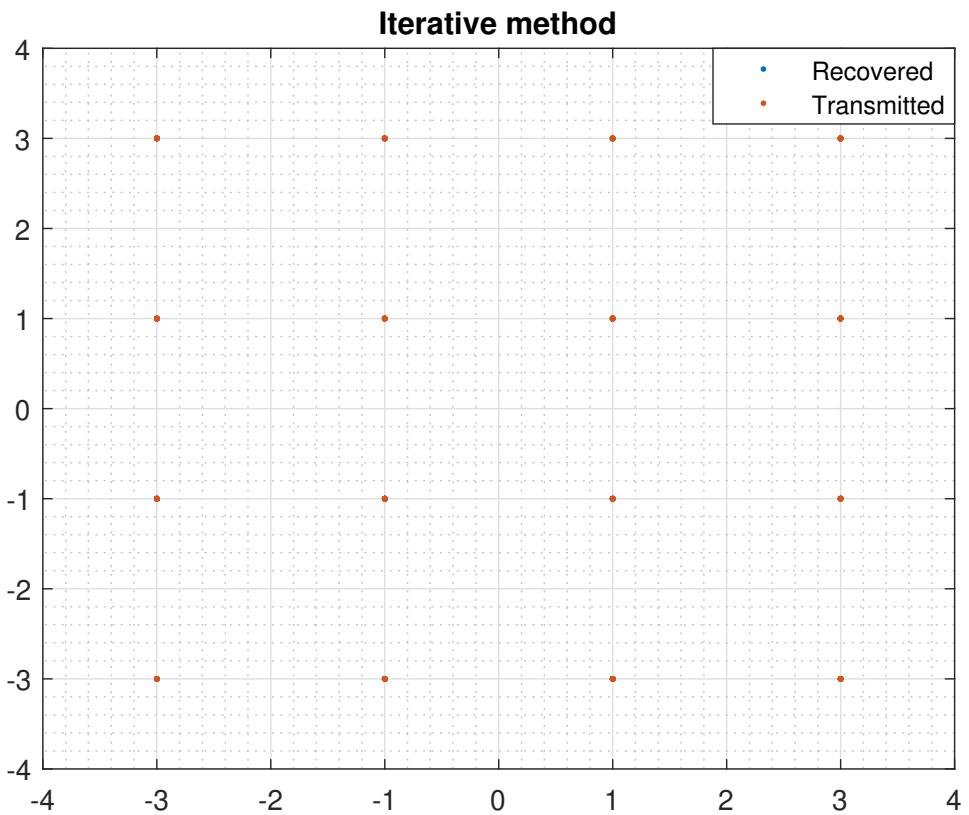


Figure 6.291: Constellation of the recovered and transmitted signal using iterative method

## References

- [1] Antonio Mecozzi, Cristian Antonelli, and Mark Shtaif. "Kramers-Kronig coherent receiver". In: *Optica* 3.11 (Nov. 2016), p. 1220. ISSN: 2334-2536. DOI: [10.1364/OPTICA.3.001220](https://doi.org/10.1364/OPTICA.3.001220). URL: <https://www.osapublishing.org/abstract.cfm?URI=optica-3-11-1220>.
- [2] Antonio Mecozzi. "Retrieving the full optical response from amplitude data by Hilbert transform". In: *Optics Communications* 282.20 (Oct. 2009), pp. 4183–4187. ISSN: 0030-4018. DOI: [10.1016/j.optcom.2009.07.025](https://doi.org/10.1016/j.optcom.2009.07.025). URL: <https://www.sciencedirect.com/science/article/pii/S0030401809006907>.
- [3] Matilde Legua and Luis Sánchez-Ruiz. "Cauchy Principal Value Contour Integral with Applications". In: *Entropy* 19.5 (May 2017), p. 215. ISSN: 1099-4300. DOI: [10.3390/e19050215](https://doi.org/10.3390/e19050215). URL: <http://www.mdpi.com/1099-4300/19/5/215>.
- [4] Antonio Mecozzi. "A necessary and sufficient condition for minimum phase and implications for phase retrieval". In: *IEEE TRANSACTIONS ON INFORMATION THEORY* 13.9 (2014). URL: <https://pdfs.semanticscholar.org/1ae2/690a2a435f94b74320d14f135f3e4928f08a.pdf>.
- [5] M. Presi et al. "Transmission in 125-km SMF with 3.9 bit/s/Hz spectral efficiency using a single-drive MZM and a direct-detection Kramers-Kronig receiver without optical CD compensation". In: *Optical Fiber Communication Conference*. Washington, D.C.: OSA, Mar. 2018, Tu2D.3. ISBN: 978-1-943580-38-5. DOI: [10.1364/OFC.2018.Tu2D.3](https://doi.org/10.1364/OFC.2018.Tu2D.3). URL: <https://www.osapublishing.org/abstract.cfm?URI=OFC-2018-Tu2D.3>.
- [6] S. T. Le et al. "8–256Gbps Virtual-Carrier Assisted WDM Direct-Detection Transmission over a Single Span of 200km". In: *2017 European Conference on Optical Communication (ECOC)*. IEEE, Sept. 2017, pp. 1–3. ISBN: 978-1-5386-5624-2. DOI: [10.1109/ECOC.2017.8346088](https://doi.org/10.1109/ECOC.2017.8346088). URL: <https://ieeexplore.ieee.org/document/8346088/>.

## 6.12 DSP Laser Phase Noise Compensation

|                     |   |   |
|---------------------|---|---|
| <b>Student Name</b> | : | Celestino Martins (08/01/2018 - )   |
| <b>Goal</b>         | : | DSP algorithms for laser phase noise compensation applied in optical coherent receiver systems. |
| <b>Directory</b>    | : | sdf/dsp_laser_phase_compensation  |

Laser phase noise is one of the fundamental impairments in coherent optical systems, since it can severely limit the synchronization between transmitter and receiver in demodulation and detection of the transmitted data. This limitation can be even more critical in high data rate optical system using higher order modulation formats, M-QAM, where data is encoded in the amplitude and phase of an optical carrier. These requirements impose that an accurate carrier phase recovery is required [1].

In coherent optical systems, carrier phase recovery has been performed primarily in the electrical domain as part of the digital signal processing (DSP), using both feedforward and feedback-based algorithms. Nevertheless, the researches experiments have shown that feedforward carrier phase recovery schemes are more tolerant to laser phase and facilitate the parallel implementation in a hardware unit, owing to the high parallelization and pipelining required in a real ASIC implementation.

### 6.12.1 Theoretical Analysis

Generally, laser phase noise can be modeled as a Wiener process [1] described as,

$$\phi(k) = \phi(k - 1) + \Delta\phi(k), \quad (6.206)$$

where,  $\Delta\phi(k)$  is an independent and identically distributed random Gaussian variable with zero mean and variance given as,

$$\sigma^2 = 2\pi(\Delta f T), \quad (6.207)$$

where  $\Delta f$  corresponds to the sum of linewidth of the signal and local oscillator lasers and  $T$  is the symbol period.

Typically, laser phase noise compensation in coherent optical receivers is performed by feedforward algorithms based on the well-known Viterbi-Viterbi (VV) algorithm [2, 1] or blind phase search algorithm (BPS) [3].

#### 6.12.1.1 Viterbi-Viterbi Algorithm

VV algorithm is a n-th power feed-forward approach employed for uniform angular distribution characteristic of m-PSK constellations, where the information of the modulated phase is removed by employing the n-th power operation on the received symbols. The algorithm implementation diagram is shown in Figure 7.10, starting with M-th power operation on the received symbols. In order to minimize the impact of additive noise in the estimation process, an average sum of  $2N + 1$  symbols is considered. The resulting estimated

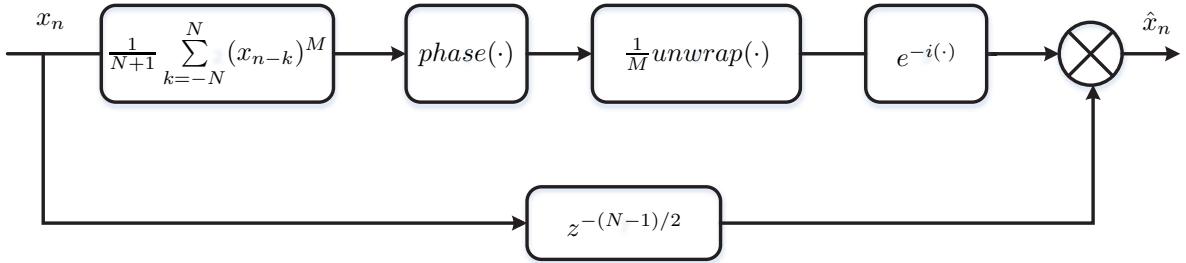


Figure 6.292: Block diagram of Viterbi-Viterbi algorithm for carrier phase recovery.

phase noise is then submitted to a phase unwrap function in order to extend the angles from  $-\infty$  to  $\infty$ . 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.

#### 6.12.1.2 Blind Phase Search Algorithm

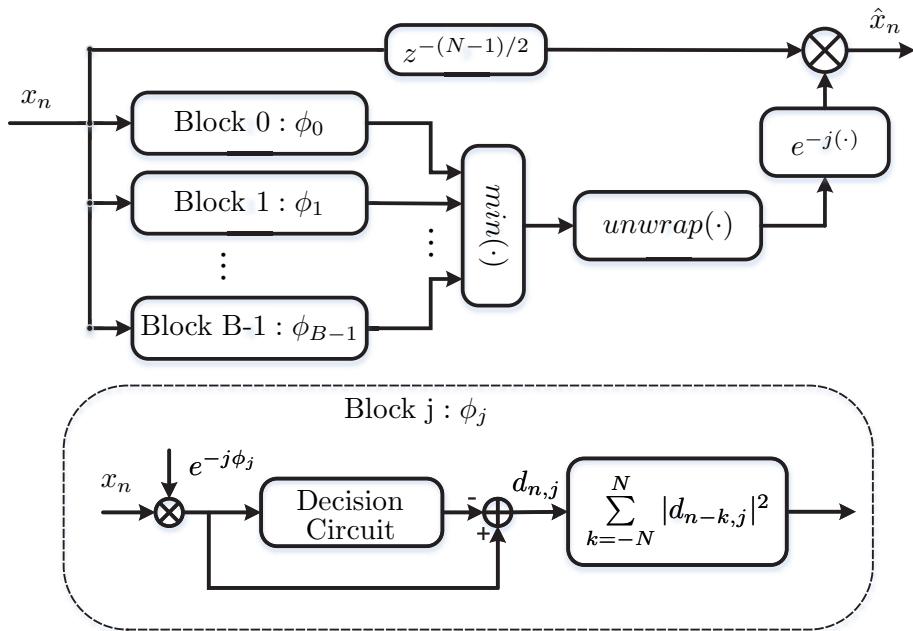


Figure 6.293: Block diagram of blind phase search algorithm for carrier phase recovery.

An alternative to the VV phase noise estimator is the so-called BPS algorithm, in which the operation principle is shown in the Figure 7.11. Firstly, a block of  $2N + 1$  consecutive received symbols is rotated by a number of  $B$  uniformly distributed test phases defined as,

$$\phi_b = \frac{b}{B} \frac{\pi}{2}, b \in \{0, 1, \dots, B - 1\}. \quad (6.208)$$

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.

### 6.12.2 Simulation Analysis

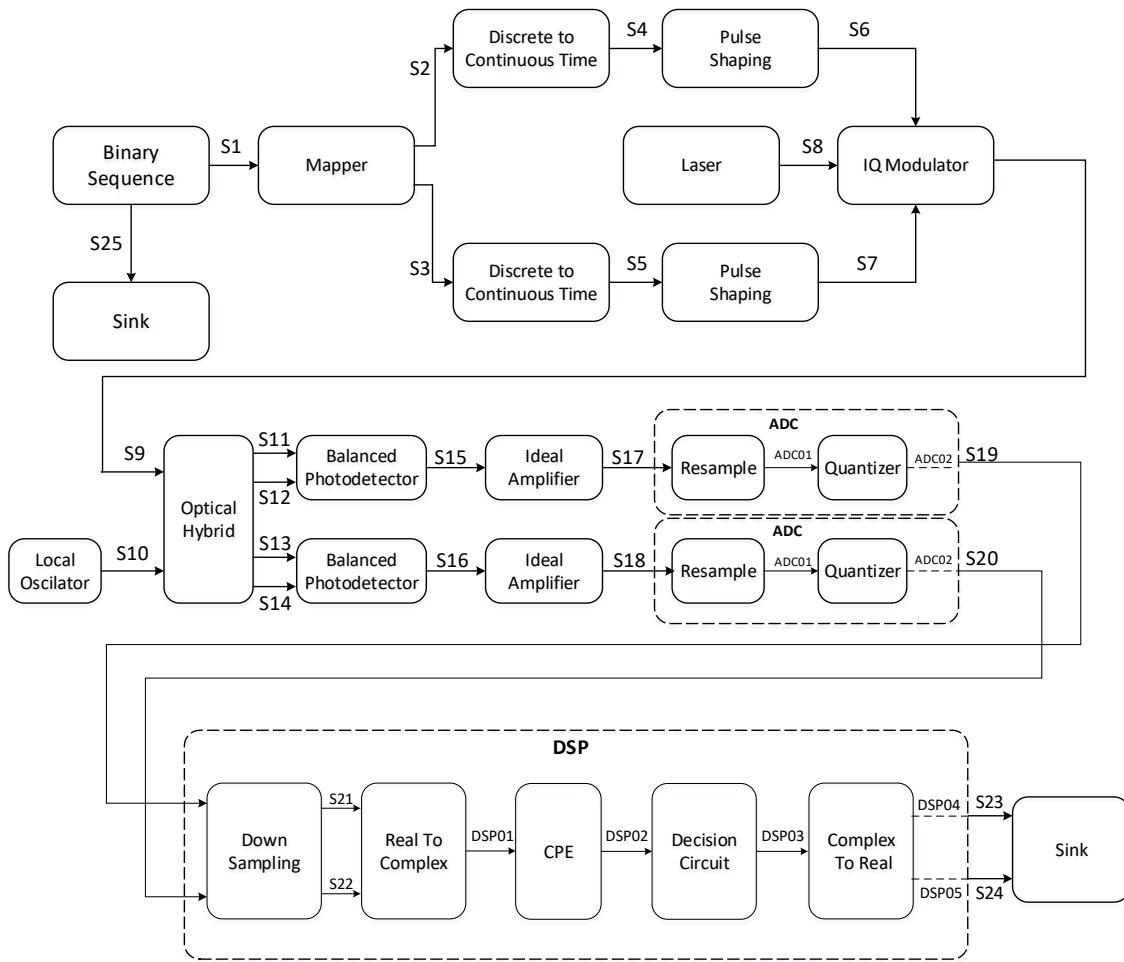


Figure 6.294: Simulation setup for the QPSK transmission system for the simulation of laser phase noise.

Figure 6.294 depicts the simulation setup for a QPSK transmission system, including the simulation of carrier phase noise. As can be observed in the figure 6.294, we have employed an IQ modulation which modulates the signal after pulse shaping,  $S_6$  and  $S_7$ , on the laser carrier,  $S_8$ . The output signal,  $S_9$ , is then fed to coherent receiver front-end, comprising optical hybrid, photodiode and ideal amplifier. The output signal to the coherent receiver front-end,  $S_{17}$  and  $S_{18}$ , are fed to the ADC super blocks, which comprises resample

block followed by quantizer block. The two real output signals of ADC blocks,  $S19$  and  $S20$ , are fed into DSP super block, where is performed the compensation of transmission impairments and recover of transmitted data. The DSP super block is composed by a down-sampling block followed by a real to complex block, which combine the two input real signal into a complex signal and then fed to CPE block, where the laser phase noise compensation is performed. The complex output signal of CPE block is then converted into two real signals using complex to real block. The two outputs,  $S23$  and  $S24$ , is then fed to the Sink block.

### 6.12.3 Simulation Results

The time-domain output signal of IQ modulator  $S9$  is presented in figure 6.295 (a) and its corresponding constellation diagram is shown in figure 6.295 (b). From the figure 6.295 (b) it can be clearly observed the effect of laser phase noise, where the signal  $S9$  presents a rotated constellation points. In general, the amount of the rotation is directly proportional to by the laser linewidth and the signal symbol period.

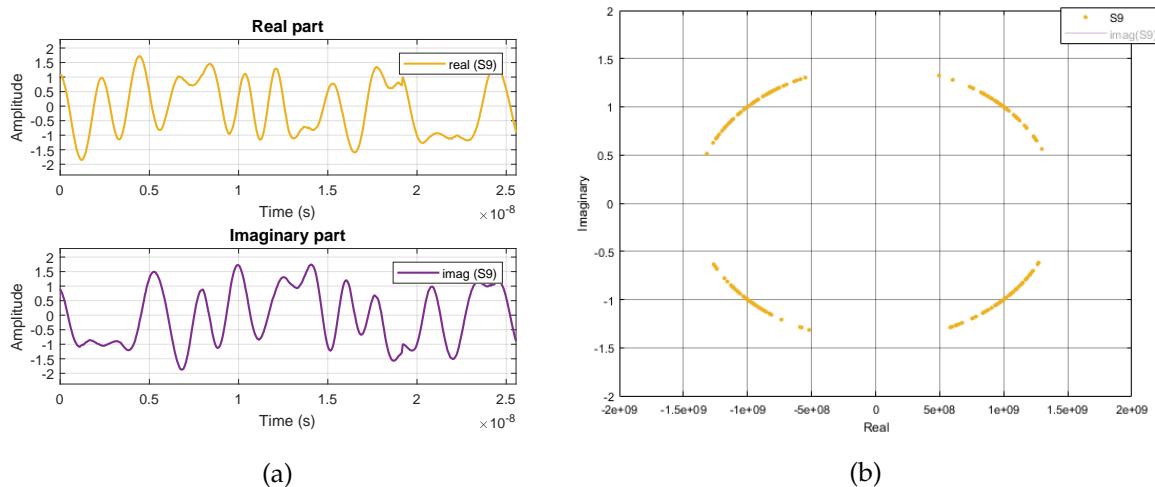


Figure 6.295: Output signal of IQ modulator,  $S9$ , including the laser phase noise. (a) Time-domain; (b) Constellation diagram.

The four signals components corresponding to in-phase and quadrature signal obtained by the optical hybrid are presented in figure 6.296 and figure 6.297, respectively.

The outputs of the two balanced photodiodes are presented in the figure 6.298.

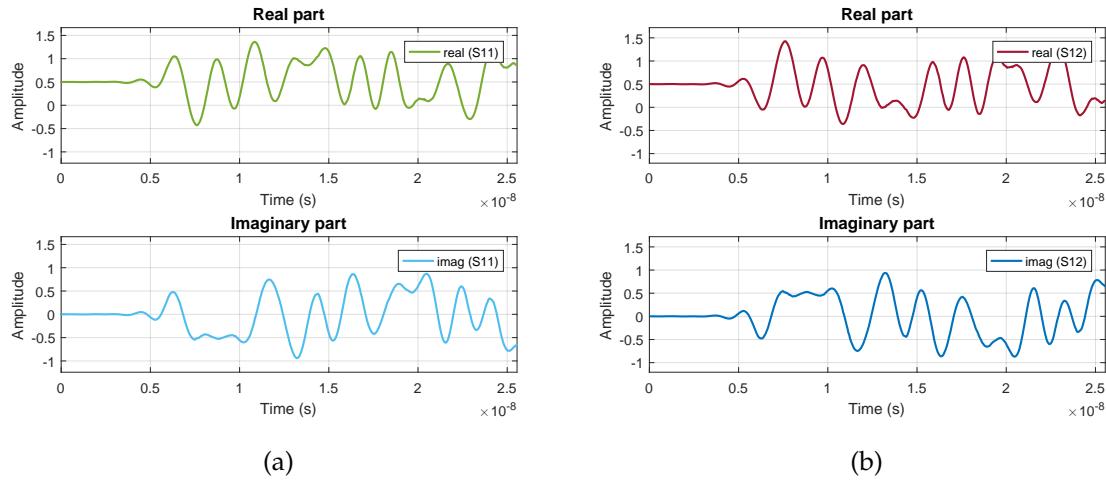


Figure 6.296: The in-phase output of optical hybrid block. (a)  $S_{11}$ ; (b)  $S_{12}$ .

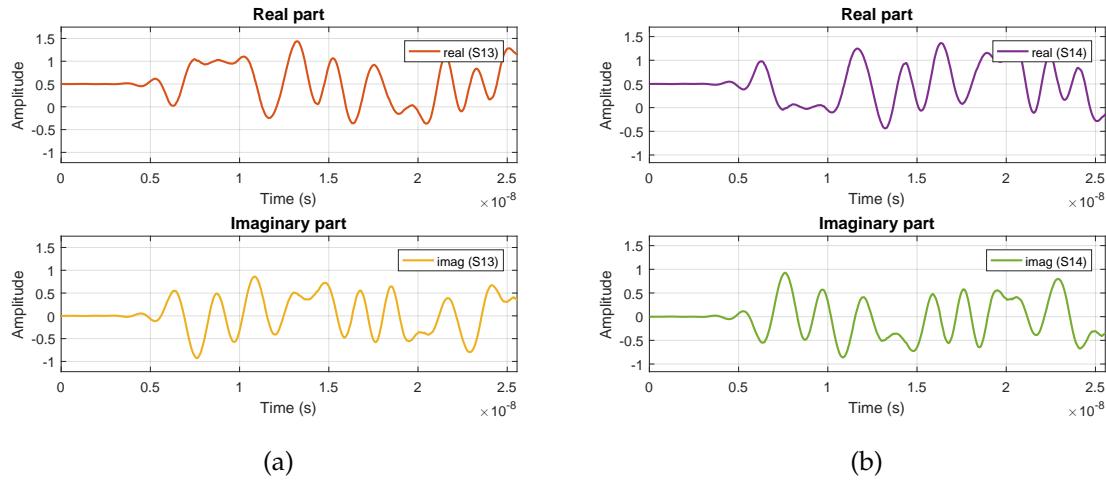


Figure 6.297: The quadrature output of optical hybrid block. (a)  $S_{13}$ ; (b)  $S_{14}$ .

The signal at the output of the amplifier is shown in the figure 6.299.

The signal after the super block ADC is shown in the figure 6.300

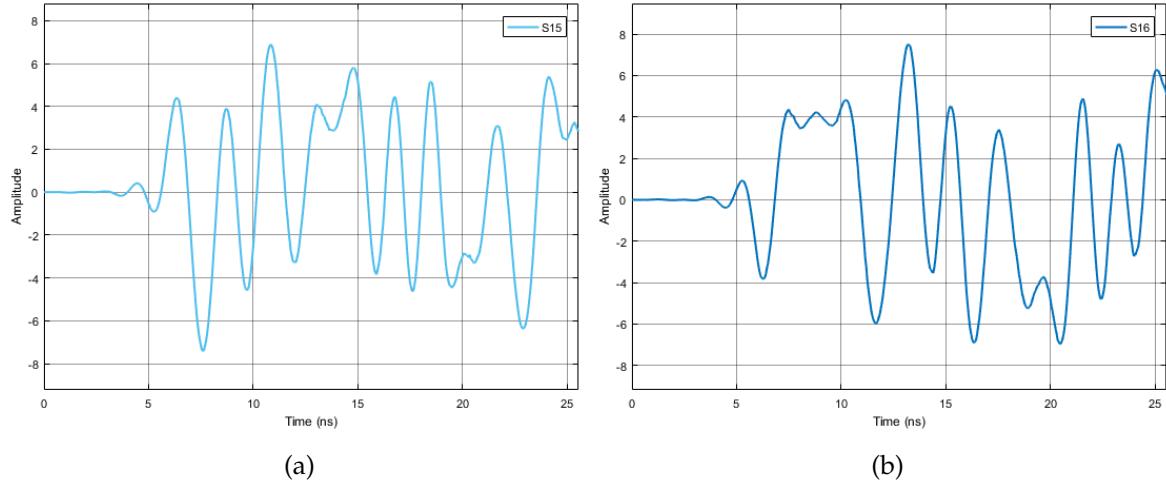


Figure 6.298: The output signal of balanced photodiodes block. (a) In-phase component,  $S15$ ; (b) Quadrature component,  $S16$ .

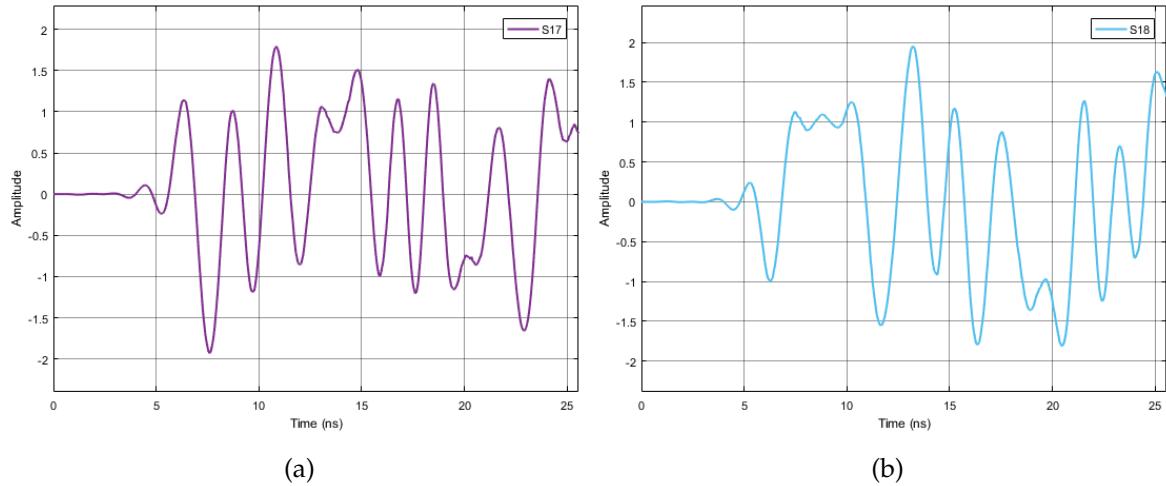


Figure 6.299: The output signal of the ideal amplifier block. (a) In-phase component,  $S17$ ; (b) Quadrature component,  $S18$ .

Figure 6.301 and Figure 6.302 show the signal after phase noise compensation using VV and BPS algorithms, respectively. As we can note in the figure 6.301 (b) and figure 6.302 (b), the correct constellation diagram points is recovered, which indicates that the laser phase noise is effectively compensated.

#### 6.12.4 VHDL Implementation

For the hardware implementation of BPS the algorithm is slightly modified in order to provide less complexity implementation. Figure 6.303 illustrates the modified version, where the decoded output symbol  $\hat{x}_n$  is selected from a set of  $B$  decoded symbols previously obtained after decision circuit,  $\hat{x}_{n,b}$ , each corresponding to a given test phase. The symbol

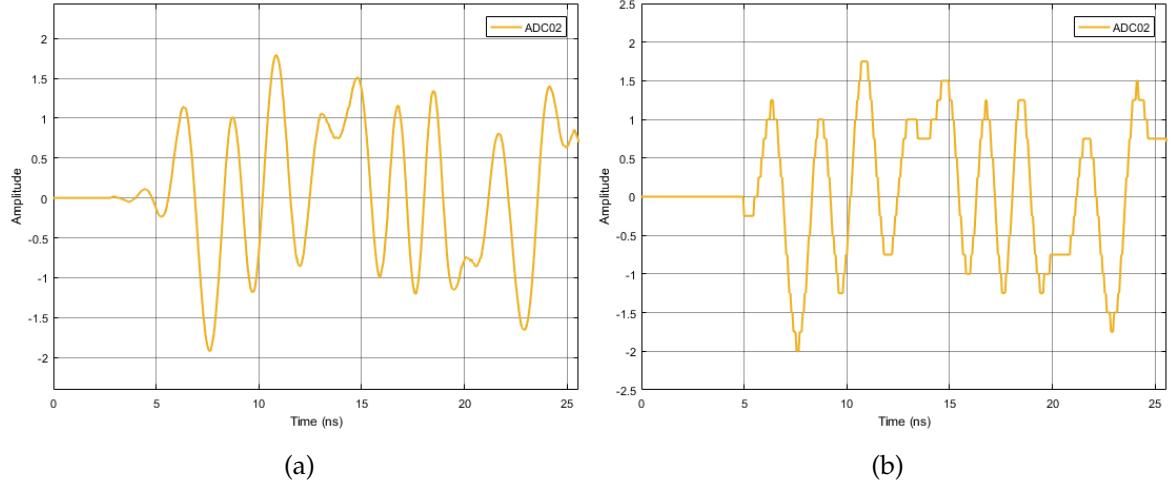


Figure 6.300: The output signal of the ADC super block, which includes the resample and quantizer blocks. (a) 8-bits quantization signal; (b) 4-bits quantization signal.

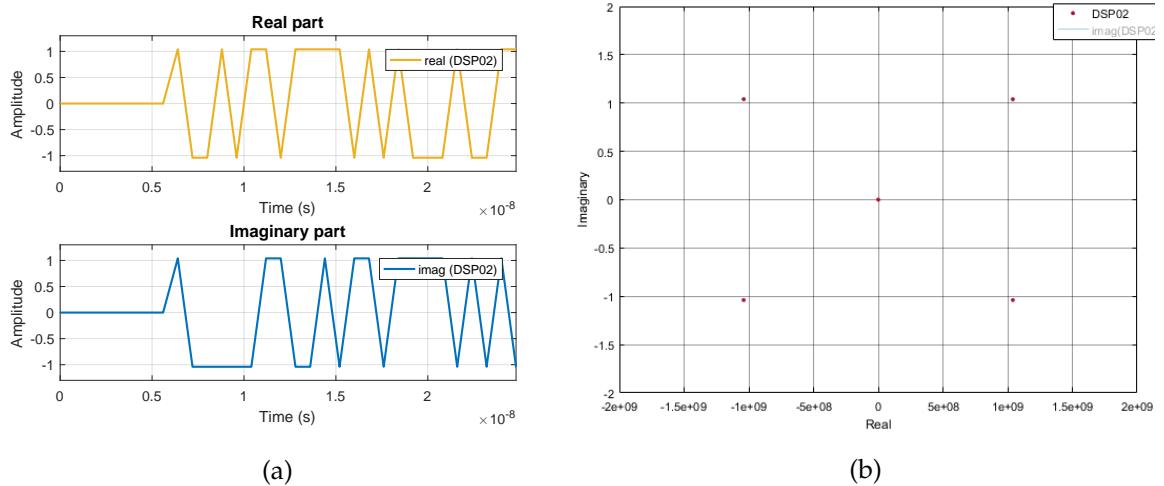


Figure 6.301: The signal after DSP block including VV algorithm for laser phase noise compensation. (a) Time-domain signal; (b) Constellation diagram.

selection is performed by a switch which is controlled by the index of the minimum distances sum.

According to the Figure 6.303, the top level implementation diagram of BPS algorithm in Very high speed integrated circuit (VHSIC) Hardware Description Language (VHDL) is shown in Figure 6.304. The design is based on the interconnection of several entities blocks, which performs a given function and describe the interface signals into and out of the design unit. The input signal to the top level entity block *CPE\_BPS* is provided by testbench, which defines the stimuli for the simulation of unit under test (UUT). The input signal is loaded from an external file, converted into *std\_logic\_vector* type and then sent to the UUT. The output of UUT of *std\_logic\_vector* type is converted integer type and then save into external

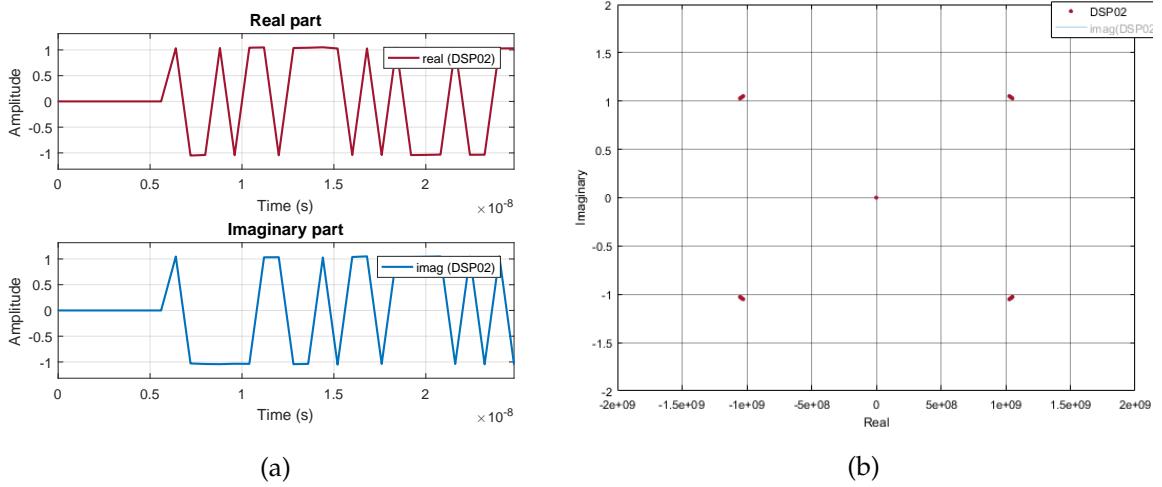


Figure 6.302: The signal after DSP block including BPS algorithm, using 64 test phases, for laser phase noise compensation. (a) Time-domain signal; (b) Constellation diagram.

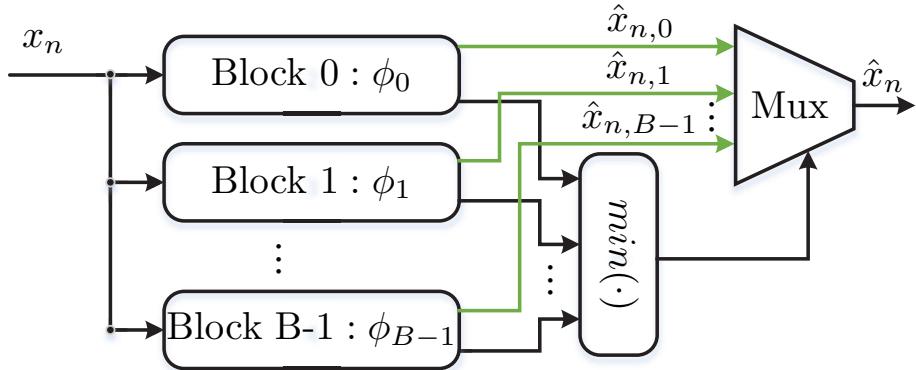


Figure 6.303: Block diagram of blind phase search algorithm for carrier phase recovery.

file.

#### 6.12.4.1 Top Level Implementation of BPS Entity Block

In *CPE\_BPS* block, the input signal is firstly delayed one clock cycle to be synchronized with the output of *2P-ROM-0*, where the exponential of test phases are saved. The output of *2P-ROM-0* and the delayed input signal correspond to the input of *Distance Calculation* block, where the distance to the closest constellation points in the original constellation is calculated. The VHDL implementation diagram of *Distance Calculation* block is presented in the Figure 6.305 and the detail is presented later. The outputs of *Distance Calculation* correspond to two array of *std\_logic\_vector* which can be interpreted as matrixes N by M, where N is the number of test phase and M is the number of parallel processed sample. One output holds the values of distance calculation and the other output hold the ROM address for the corresponding decision from *DecisionCircuit* of square distance calculation block,

Figure 6.306. The two  $N \times M$  matrixes outputs of *Distance Calculation* are converted to  $M \times N$  matrixes using *Array2Array* entity block. Then, the output holding the values of distance calculation correspond to the input of *MinArray* entity block, where the minimum distance for each  $M$  input samples are determined. For complexity reduction purpose, the output of *MinArray* is a *std\_logic\_vector* holding  $M$  indexes of minimum distance values. In parallel to the minimum distance search, the matrix holding the ROM addresses is delayed to be synchronized with the output of *MinArray*, which are then the input of *ROM Adress Selection* entity block. Based on the output of *MinArray*, the block *ROM Adress Selection* selects the addresses of decoded output symbol from IQ signal ROM. The  $M$  addresses obtained from *ROM Adress Selection* are used to select the decoded output symbol from IQ signal ROM.

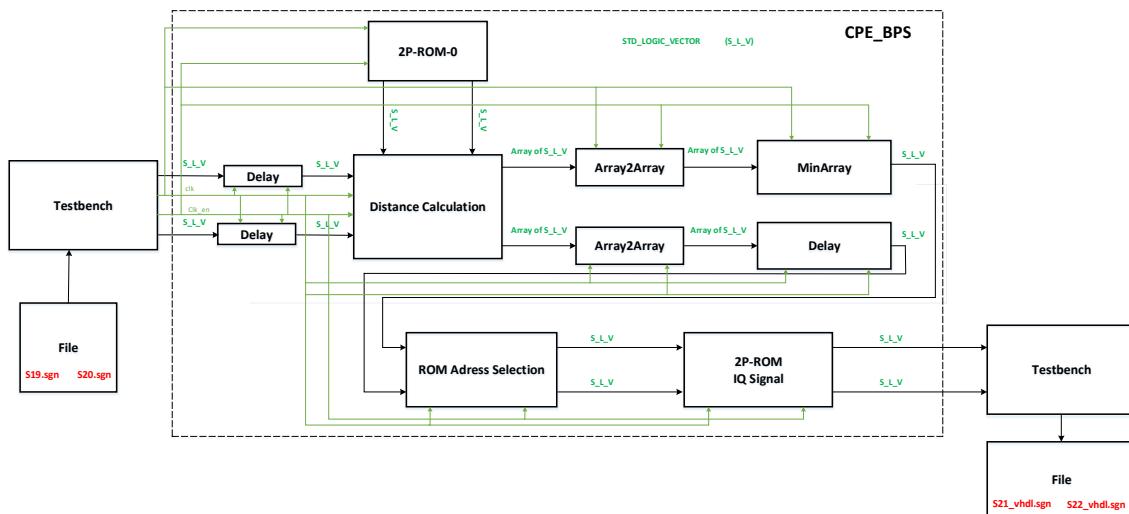


Figure 6.304: Top level entity block for VHDL implementation of BPS.

#### 6.12.4.2 Implementation of Distance Calculation Entity Block

The *Distance Calculation* entity block is interfaced by four inputs and two outputs of *std\_logic\_vector* type. The inputs correspond to the real and imaginary of input samples and test phase. The outputs are the corresponding distance between the rotated input sample to the closest constellation points and the ROM address of corresponding symbol decision. *Distance Calculation* entity block is composed by the interconnection of 3 entity blocks. Firstly, the square distance are calculated and the ROM address of symbol decision are determined by *Distance Calculation* entity block. The consecutive calculated distances are buffered using *Buffer* entity block, which is then the input of *Average* entity block for the average calculation. In parallel, the ROM address output is delayed to be synchronized with the output of *Average* entity block.

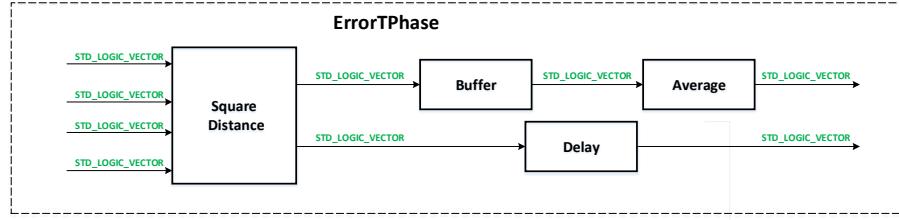


Figure 6.305: VHDL implementation of sub-block of CPE for the distance calculation between the signal and  $N$  test phases.

#### 6.12.4.3 Implementation of Square Distance Calculation Entity Block

The *Square Distance Calculation* entity block has the same inputs and outputs as *Distance Calculation* entity block. The first operation correspond to the complex multiplication between the input sample and the test phase. The result is then fed to the *Decision Circuit* entity block, where the symbol decision is performed. The output of *Decision Circuit* is the ROM address of corresponding decoded symbol. The ROM outputs and the delay version of complex multiplication are then fed to the *Complex Subtraction* entity block followed by its absolute value calculation. The output of *Absolute Value* entity block is then registered, which is the output of *Square Distance Calculation* entity block in parallel with the delayed ROM address of decoded symbol.

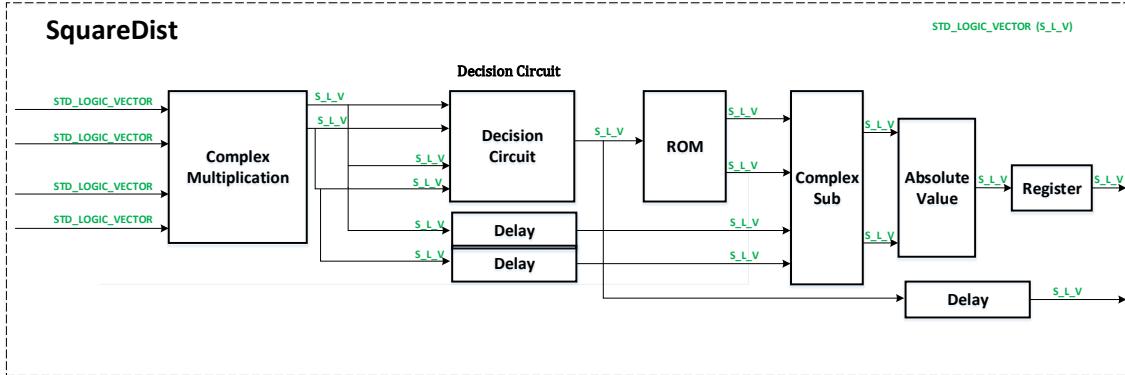


Figure 6.306: VHDL implementation of sub-block of CPE for the square distance calculation between the signal and one test phase.

#### 6.12.5 VHDL Simulation Results

In order to validate the the VHDL implementation of CPE we have used open-source simulator GHDL. According to the Figure 6.294, the inputs signal to VHDL CEP block are the signals S21 and S22, which correspond to the real and imaginary part, respectively. The corresponding outputs S23\_vhdl and S24\_vhdl are compared with S23\_V2 and S24\_V2, respectively, which are the simulator output based on floating-point operation. The signals S23\_V2 and S24\_V2 are obtained according to the Figure 6.303, where the decoded symbol is

directly selected from a set of  $B$  decoded symbol. The in-phase and quadrature component of the output signal at DSP block using this version of implementation is presented in Figure 6.307.

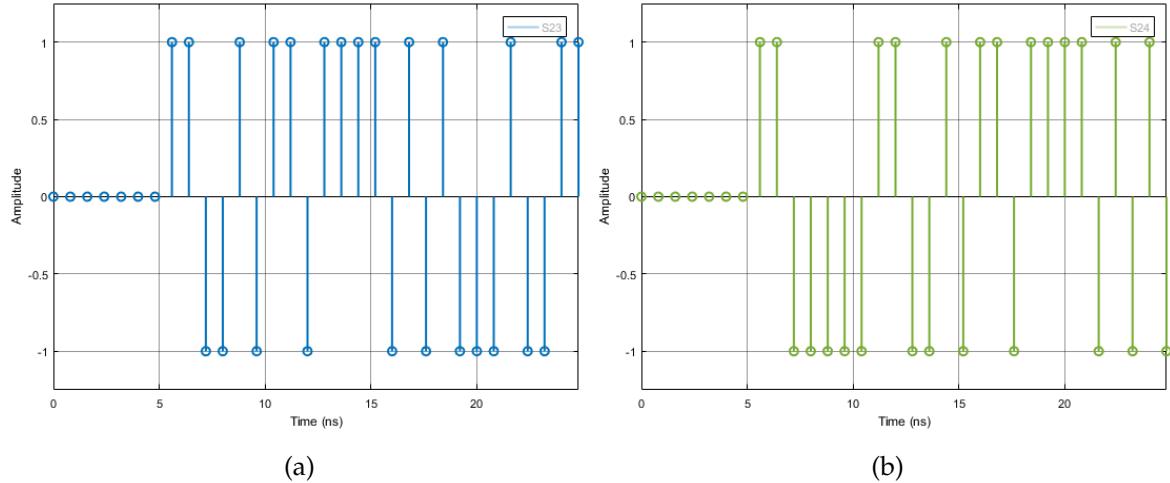


Figure 6.307: The output signal of the DSP super block including CPE using 32 test phases. (a) In-phase component; (b) Quadrature component.

The outputs of VHDL implementation of BPS is shown in Figure 6.308. It should be noted that a given delay is introduced by the VHDL implementation, which impose the first symbols between Figure 6.307 and Figure 6.308 are different.

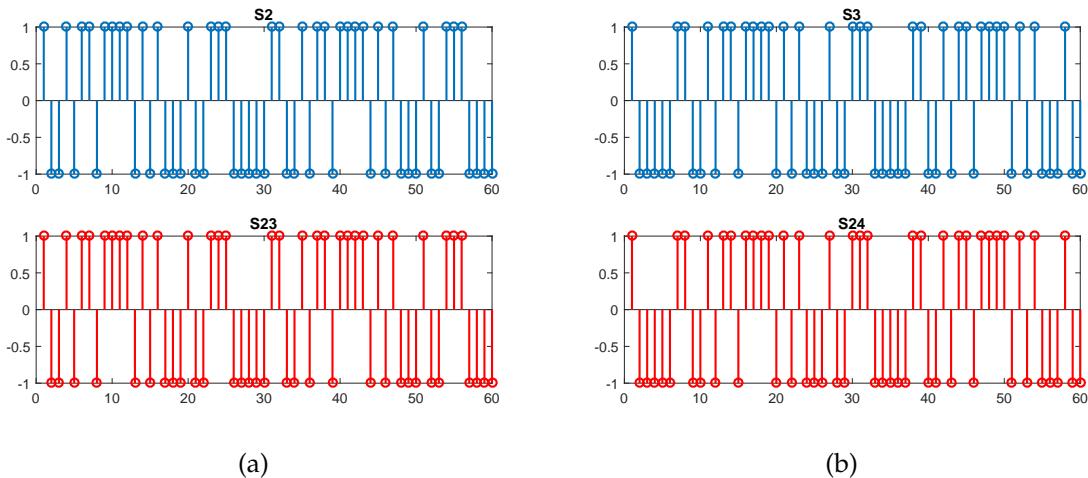


Figure 6.308: VHDL simulation output signal after CPE block in comparison with the signal  $S2$  and  $S3$ . (a) In-phase component; (b) Quadrature component.

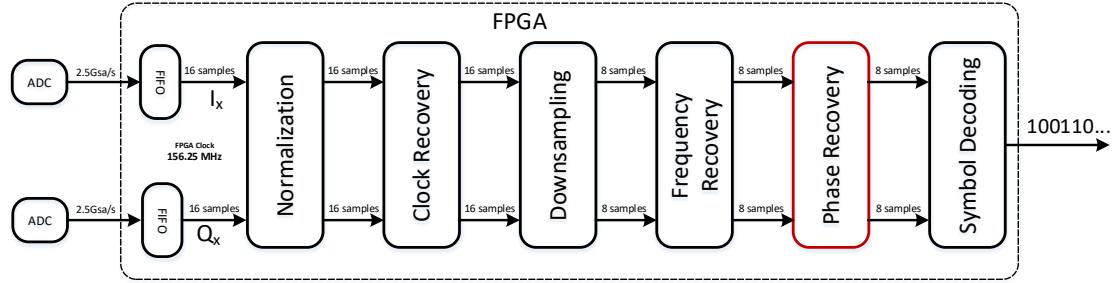


Figure 6.309: Simplified DSP for coherent receiver using single-polarization.

### 6.12.6 DSP Architectures for Coherent Receiver

Figure 6.309 illustrates a simple DSP schematic for coherent receiver, which is composed by, normalization, clock recovery, downsampling, frequency recovery, phase recovery and finally the decoding block.

Since the data rates used in the optical communications systems is high, in this case 2.5 Gsa/s, and the standard clock frequency used for FPGA-based DSP applications is, in this case, 156.25 MHz the DSP needs to process various samples per each clock cycle. Considering that the signal is sampled at 2.5 Gsa/s, a degree-of-parallelization,  $N$ , of 16 in the DSP is required to process such signal. This means that the DSP needs to process 16 samples of the signal in parallel per clock cycle. The output samples of ADC are firstly stored in a FIFO, which is then sent to the DSP blocks of coherent receiver.

#### 6.12.6.1 Normalization

The first DSP block is the Normalization block and is used to control electrical amplitude fluctuations and thus mitigate digital overflows or loss of resolution. It should be noted that a simple digital overflow in an internal signal of the subsequent block can strongly degrade the algorithms operations. This block ensures that the amplitude of the digital signal spans over the entire excursion of the DSP bit-width resolution, improving the dynamic range. The algorithm is designed with a feedback architecture based on integral control with the transfer function  $\frac{\mu}{s}$  in the s-domain approach, where  $\mu$  corresponds to the integral gain, enabling to dynamically adjusts the signal amplitude through a measured amplitude error. The amplitude error is given by:

$$e(k) = e(k-1) + \mu \left[ d - \frac{1}{2} \sum_{n=0}^1 |y(kN+2-n)| \right], \quad (6.209)$$

where  $y(k)$  is the normalized sampled signal and  $d$  is the reference absolute amplitude. Note that only two samples per DSP clock cycle are used for the error estimation, since varying amplitude optical signal is much slower than the DSP clock cycle.

Directory:

`\sdf\ dsp_laser_phase_compensation\VHDL\DSP_Functions\NormalizationSP_IT`

### 6.12.6.2 Clock Recovery

The Clock Recovery block is included after Normalization block for the estimation of the ideal sampling instant for digital coherent receiver, due to reason that the symbol rate of the transmitted signal may not match an exact integer multiple of the receiver ADC clock. The implemented Clock Recovery algorithm is based on Gardner algorithm [Zhou11], a widely used technique due to its simplicity and carrier independency properties and its operation requires two sample per symbol. In the first step of algorithm the signal interpolation is performed, where the sampling instant is adjusted, which is them followed by the timing error detector (TED), where it is estimate the error that has to be corrected. For the interpolation it can be used linear interpolation or cubic interpolation [Zhou11], while for TED is used Gardner algorithm based on signal amplitude or signal power [Meng03].

Directory:

`\sdf\ dsp_laser_phase_compensation\VHDL\ DSP_Functions\ ClockRecoveryFeedback_IT`

### 6.12.6.3 Downsampling

In the following it is included the Downsampling block, which reduce the number of sample per symbol to one sample per symbol.

Directory:

`\sdf\ dsp_laser_phase_compensation\VHDL\ DSP_Functions\ Downsample_IT`

### 6.12.6.4 Frequency Recovery

The Frequency Recovery block is included in the following to estimate and compensate the frequency offset induced by free-running optical lasers. The proposed implementation is a differential phase-based method with a feedforward design [Savory08]. This method is based on the  $M^{th}$ -order power schemes working with one SPS, which is a nonlinear method used to remove the phase modulation from the input signal before performing the frequency estimation. This algorithm can be implemented using two different architectures, based on feedforward or feedback control, wherein the feedforward configuration is the most popular approach.

Directory:

`\sdf\ dsp_laser_phase_compensation\VHDL\ DSP_Functions\ FrequencyRecovery_IT`

### 6.12.6.5 Phase Recovery

Phase Recovery block is used after Frequency Recovery to perform the laser phase noise estimation and compensation. The implementation is based on both the well-known VV algorithm and blind phase search BPS.

Directory:

\sdf\ dsp\_laser\_phase\_compensation\VHDL\ DSP\_Functions\ ViterbiViterbi\_IT  
 \sdf\ dsp\_laser\_phase\_compensation\VHDL\ DSP\_Functions\ CPE\_BPS\_PhU

#### 6.12.6.6 Symbol Decoding

Finally, the symbol decoding is performed including the bit error count. Due to the phase ambiguity of the blind phase recovery algorithm, differential encoding and decoding can be used to solve this issue [Savory08].

Directory:

\sdf\ dsp\_laser\_phase\_compensation\VHDL\ DSP\_Functions\ BERT\_DQPSK\_IT

#### 6.12.7 Open Issues

In order to read the outputs signals of VHDL simulation, S23\_vhdl and S24\_vhdl, we have followed the following procedures:

- Firstly, we should note that in testbench we convert the 8-bits output sample to a char, which is then save with double precision in a txt file. Each char is saved with 64-bits, where 8-bits output samples values are repeated 8 times. Therefore, to obtain the 8-bit value of output sample we down-sample the S23\_vhdl and S24\_vhdl with down-sampling factor of 8;
- Then, we use the function *valConvert2decPar* to convert the fixed-point 8-bits representation to the double precision. As the input parameter it requires the number of bits used for the fractional part and the total number of bits representation.

#### 6.12.8 FMC + FPGA

In this section we present the basic steps for the integration of FMC card with FPGA board. The main goal of this section is to work as a guide so that it facilitates the this integration.

##### 6.12.8.1 Software Installation

- Install ISE or Vivado;
- Install 4DSP software (<http://www.4dsp.com/FMC/BSP>) choosing the corresponding FMC version (FMC125,FMC230,...). It is required windows 7 32 bits;
- Install Visual Studio 2012 and follow the configurations steps in *4FM\_Get\_Started\_Guide* documentation.

#### 6.12.8.2 Check the FMC and FPGA board integration

Firstly, the FMC should be connected to the FPGA board (Any doubts, you should talk to Eng. Prata) and then the board is connected to the computer through ethernet and usb cable. More details can be obtained in *4FM\_Get\_Started\_Guide* documentation.

After this steps, open "Chipscope", and click on the small icon on the top (left side) to detect the FPGA device. After a few seconds a window appears ... click on "ok". If it does not detect, you should go to Computer, Properties and left side on top click on Device Manager with the FPGA turned on it should appear the 4DSP drive properly installed, otherwise look for solutions on the net for "4DSP USB Cable".

#### 6.12.8.3 FPGA Programming

Before the calibration process, we need to load the bit stream into the FPGA. This can be performed in the following steps:

- Open Analyzer Chipscope and click on the small icon on the top left side to detect the FPGA device;
- Right click on the particular detected hardware and go to configure;
- Select the bitstream and load it into FPGA. After this step the FPGA is programmed.
- GO to File, Open project and then select the corresponding chipscope project.

#### 6.12.8.4 ADC Calibration

Calibration is required to obtain optimum performance when two or more ADC converters are interleaved. In this case the calibration is performed such that the converters have the same characteristics in terms of the offset, the gain, and the phase shifting. Proper calibration will greatly improve noise performance (SNR) and dynamic range (SFDR). Usually we perform the first time calibration process to obtain the optimum calibration points for offset, gain and phase. Then, we use these optimum values in a pre-build C++ software to generate a an executable file, which can be always used when the calibration is required (usually when we run the fmc+board).

- Load the .bit files to the FPGA using the Chipscope. Choose the bit file that reflects FMC card and FPGA board (ex. *428\_vc707\_fmc125\_cal.bit*). The path for this is *C :\ProgramFiles\4dsp\Common\Firmware\Recovery*;
- Open cmd;
- From a command window browse to *C :\ProgramFiles\4DSP\FMCBoardSupportPackage\Bins* and run the application that matches your FMC card (ex. *Fmc1251\_Calibration.exe*);

- In the cmd run the .exe calibration file as *Fmc1251\_Calibration.exe* 1 VC707, and identify the device index(0,1,2,3,...). Then run *Fmc1251\_Calibration.exe* 1 VC707 1 2, according to the identified device index, in this case is set to 1.
- Follow the instructions on the screen. At some point it will ask for an input of sine wave 0.5Vpp and 450 MHz. After that, it will follow and when the calibration is completed it is generated the optimum calibration points.

#### 6.12.8.5 Generate .exe files from the C++ code

After obtaining the optimum calibration points we can generate the .exe file for the calibration.

- Go to the folder *C:\Program Files(x86)\4dsp\FMCBoardSupportPackage\Refs\Software\FMC125\_Calibration;*
- Open the C++ project using Microsoft Visual Studio 2012;
- In the main, change the values of parameters offset, gain and phase by the new values obtained using after calibration;
- Rebuild the project to obtain the new .exe file, which can be found in the debug folder. Copy this .xe file to a working directory for future calibration.

#### 6.12.8.6 User mode

After obtaining the new calibration .exe the calibration process is performed as:

- Open cmd;
- Browse to the folder where is located the executable file for the calibration;
- In the cmd run the .exe calibration file as *Fmc1251\_Calibration.exe* 1 VC707, and identify the device index(0,1,2,3,...). Then run *Fmc1251\_Calibration.exe* 1 VC707 1 2, according to the identified device index, in this case is set to 1.
- After the *Fmc1251\_Calibration.exe* 1 VC707 1 2 command, you need reset the FPGA in the two push buttons (reset buttons) until the Errors parameter in the ChipScope shows changes;
- In principle you need make several resets in order to obtain the convergence of the algorithms (do this until Errors change);
- Then, it is OK.

## References

- [1] E. Ip and J. M. Kahn. "Feedforward Carrier Recovery for Coherent Optical Communications". In: *Journal of Lightwave Technology* 25.9 (Sept. 2007), pp. 2675–2692. ISSN: 0733-8724. DOI: [10.1109/JLT.2007.902118](https://doi.org/10.1109/JLT.2007.902118).
- [2] A. Viterbi. "Nonlinear estimation of PSK-modulated carrier phase with application to burst digital transmission". In: *IEEE Transactions on Information Theory* 29.4 (July 1983), pp. 543–551. ISSN: 0018-9448. DOI: [10.1109/TIT.1983.1056713](https://doi.org/10.1109/TIT.1983.1056713).
- [3] T. Pfau, S. Hoffmann, and R. Noe. "Hardware-Efficient Coherent Digital Receiver Concept With Feedforward Carrier Recovery for  $M$ -QAM Constellations". In: *Journal of Lightwave Technology* 27.8 (Apr. 2009), pp. 989–999. ISSN: 0733-8724. DOI: [10.1109/JLT.2008.2010511](https://doi.org/10.1109/JLT.2008.2010511).

## 6.13 Quantum Random Number Generator

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

True random numbers are indispensable in the field of cryptography [1]. There are two approaches for random number generation: the pseudorandom generation which are based on an algorithm implemented on a computer, and the physical random generators which consist in measuring some physical observable with random behaviour. Since classical physics description is deterministic, all classical processes are in principle predictable. Therefore, a true random number generator must be based on a quantum process [2].

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

### 6.13.1 Theoretical Analysis

One of the optical processes available as a source of randomness is the splitting of a polarized single photon beam. The principle of operation of the random generator is shown in figure 6.310. Each individual photon coming from the source is linearly polarized at  $45^\circ$  and has equal probability of be found in the horizontal (H) or in the vertical (V) output of the PBS. Quantum theory estimates for both cases that the individual choices are truly random, independent one from each other , and with a probability of  $1/2$ .

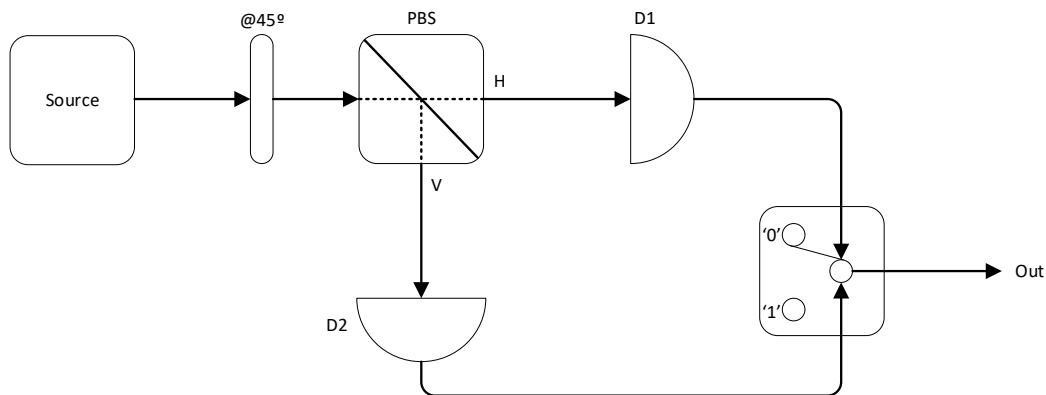


Figure 6.310: Source of randomness with a polarization beam splitter PBS where the incoming light is linearly polarized at  $45^\circ$  with respect to the PBS. Figure adapted from [2].

From a classical approach, the information is stored as binary bits that can take the logical

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

$$|\psi\rangle = C_0|0\rangle + C_1|1\rangle, \quad (6.210)$$

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

$$|\psi\rangle = C_0|0\rangle + C_1|1\rangle \quad (6.211)$$

$$= C_0|0^\circ\rangle + C_1|90^\circ\rangle. \quad (6.212)$$

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

$$|\psi\rangle = \begin{pmatrix} C_0 \\ C_1 \end{pmatrix}, \quad (6.213)$$

which will be used in simulation analysis.

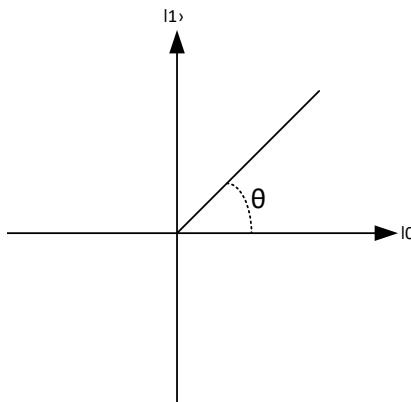


Figure 6.311: Representation of polarization states of a qubit in a bi-dimensional space.

As one can see in figure 6.311 the amplitude coefficients can be written as a function of  $\theta$ :

$$C_0 = \cos(\theta) \quad (6.214)$$

$$C_1 = \sin(\theta). \quad (6.215)$$

According with the setup presented in figure 6.310 and considering the polarization angle  $\theta = 45^\circ$ , the single photon has the probability of reach **D1** and outputs a "0" is equals to  $|\cos(\theta)|^2$  and the probability of reach **D2** and outputs a "1" is equals to  $|\sin(\theta)|^2$ , which in the case  $\theta = 45^\circ$  both have the same value equals to 0.5.

### 6.13.2 Simulation Analysis

The simulation diagram of the setup described in the previous section is presented in figure 6.312. The linear polarizer has an input control signal (S1) which allows to change the rotation angle. Nevertheless, the only purpose is to generate a time and amplitude continuous real signal with the value of the rotation angle in degrees. In addition, the photons are generated by single photon source block at a rate defined by the clock rate. At the end of the simulation there is a circuit decision block which will outputs a binary signal with value "0" if the detector at the end of the horizontal path clicks or "1" if the detector at the end of the vertical path clicks.

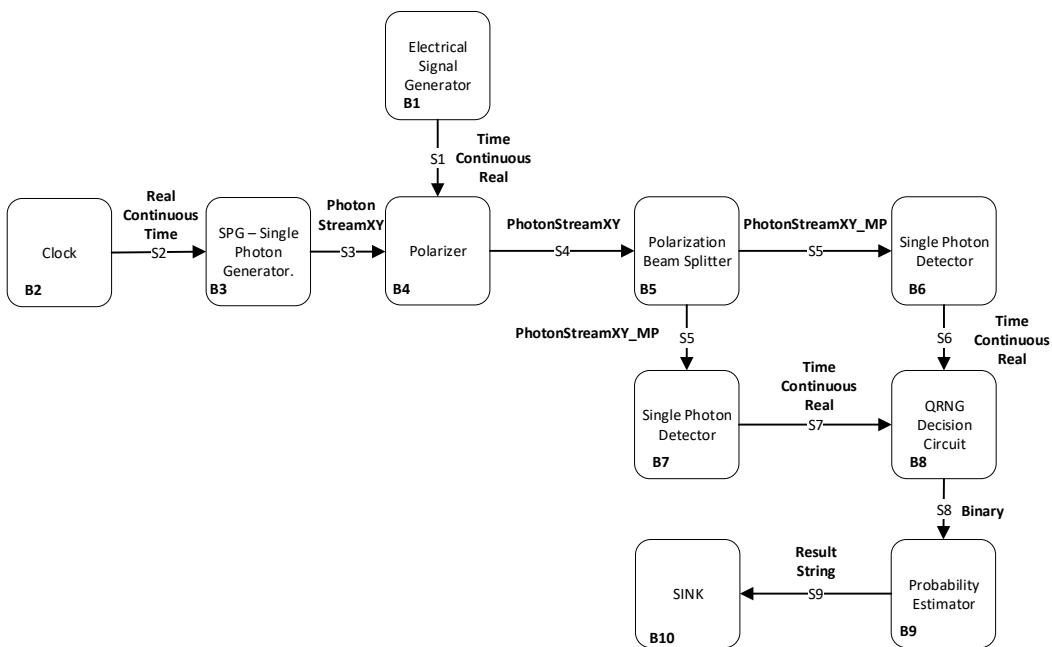


Figure 6.312: Block diagram of the simulation of a Quantum Random Generator.

In table 6.41 are presented the input parameters of the system.

Table 6.41: System Input Parameters

| Parameter                | Default Value |
|--------------------------|---------------|
| RateOfPhotons            | 1e6           |
| NumberOfSamplesPerSymbol | 16            |
| PolarizerAngle           | 45.0          |

In table 6.42 are presented the system signals to implement the simulation presented in figure 6.312.

Table 6.42: System Signals

| Signal name | Signal type                           |
|-------------|---------------------------------------|
| S1          | TimeContinuousAmplitudeContinuousReal |
| S2          | TimeContinuousAmplitudeContinuousReal |
| S3          | PhotonStreamXY                        |
| S4          | PhotonStreamXY                        |
| S5          | PhotonStreamXYMP                      |
| S6          | TimeContinuousAmplitudeContinuousReal |
| S7          | TimeContinuousAmplitudeContinuousReal |
| S8          | Binary                                |
| S9          | Binary                                |

Table 6.43 presents the header files used to implement the simulation as well as the specific parameters that should be set in each block. Finally, table 6.44 presents the source files.

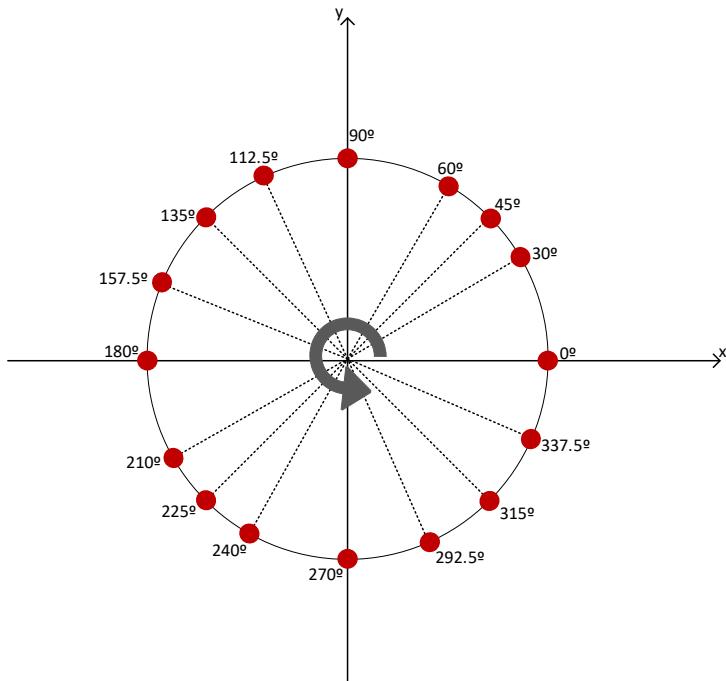
Table 6.43: Header Files

| File name                              | Description                    | Status |
|--|--------------------------------|--------|
| netxpto_20180118.h                     |                                | ✓      |
| electrical_signal_generator_20180124.h | setFunction(), setGain()       | ✓      |
| clock_20171219.h                       | ClockPeriod(1 / RateOfPhotons) | ✓      |
| polarization_beam_splitter_20180109.h  |                                | ✓      |
| polarizer_20180113.h                   |                                | ✓      |
| single_photon_detector_20180111.h      | setPath(0), setPath(1)         | ✓      |
| single_photon_source_20171218.h        |                                | ✓      |
| probability_estimator_20180124.h       |                                | ✓      |
| sink.h                                 |                                | ✓      |
| qrng_decision_circuit.h                |                                | ✓      |

Table 6.44: Source Files

| File name                                | Description | Status |
|--|-------------|--------|
| netxpto_20180118.cpp                     |             | ✓      |
| electrical_signal_generator_20180124.cpp |             | ✓      |
| clock_20171219.cpp                       |             | ✓      |
| polarization_beam_splitter_20180109.cpp  |             | ✓      |
| polarizer_20180113.cpp                   |             | ✓      |
| single_photon_detector_20180111.cpp      |             | ✓      |
| single_photon_source_20171218.cpp        |             | ✓      |
| probability_estimator_20180124.cpp       |             | ✓      |
| sink.cpp                                 |             | ✓      |
| qrng_decision_circuit.cpp                |             | ✓      |
| qrng_sdf.cpp                             |             | ✓      |

Lets assume, for an angle of  $45^\circ$ , a number of samples  $N = 1 \times 10^6$  and the expected probability of reach each detector of  $\hat{p} = 0.5$ . We have an error margin of  $E = 1.288 \times 10^{-3}$ , which is acceptable. This way, the simulation will be performed for  $N = 1 \times 10^6$  samples for different angles of polarization shown in figure 6.313 with different error margin's values since the expected probability changes depending on the polarization angle.

Figure 6.313: Angles used to perform the qrng simulation for  $N = 1 \times 10^6$  samples.

For a quantum random number generator with equal probability of obtain a "0" or "1" the polarizer must be set at  $45^\circ$ . This way, we have 50% possibilities to obtain a "0" and 50% of possibilities to obtain a "1". This theoretical value meets the value obtained from the simulation when it is performed for the number of samples mentioned above.

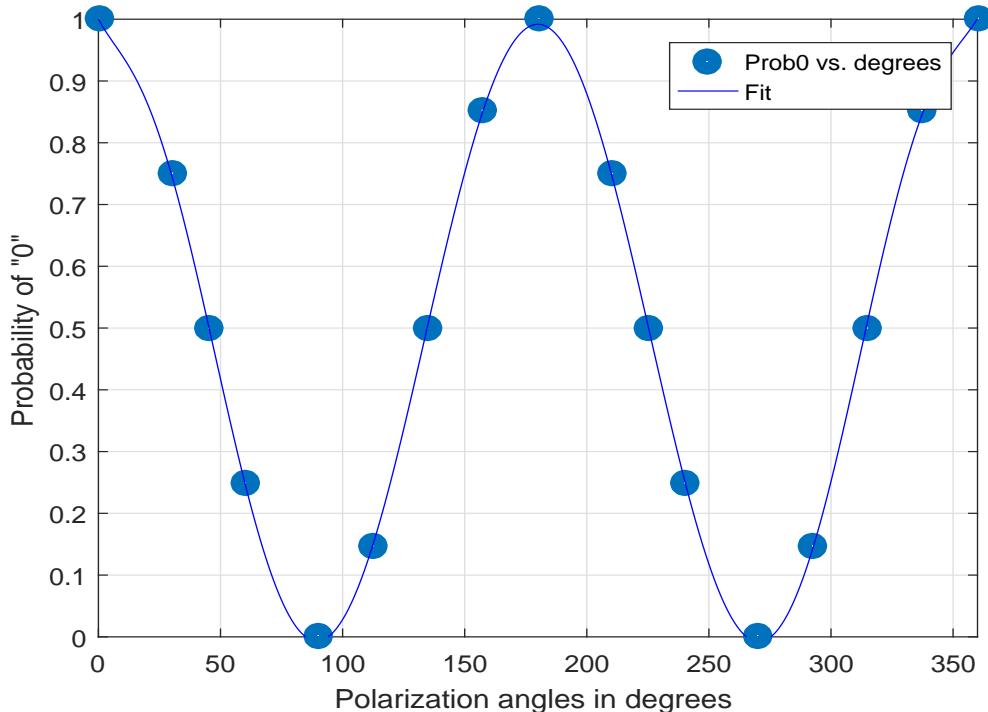


Figure 6.314: Probability of outputs a number "0" depending on the polarization angle.

Figure 6.314 shows the probability of a single photons reaches the detector placed on Horizontal axis depending on the polarization angle of the photon, and this way the output number is "0". The following table shows the goodness of the fit:

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

On the other hand, figure 6.315 shows the probability of a single photon reaches the detector placed on Vertical component of the polarization beam splitter, and this way the output number is "1". As we can see in the figures the two detectors have complementary probabilities, i.e the summation of both values must be equals to 1. One can see that "Probability of 1" behaves almost like a sine function and "Probability of 0" behaves almost like a cosine function with a variable angle.

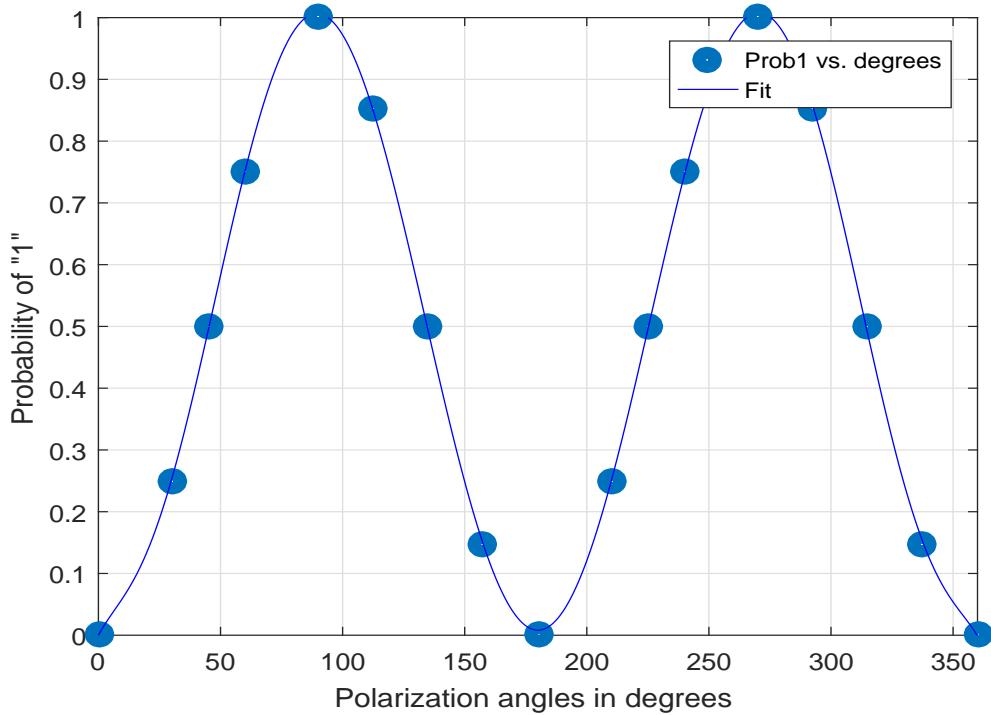


Figure 6.315: Probability of outputs a number "1" depending on the polarization angle.

The goodness of the fit presented in figure 6.315 is shown in the following table:

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

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

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

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

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

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

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

where,

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

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

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

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

where,

$$v = n - m,$$

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

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

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

where,

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

### 6.13.3 Experimental Analysis

In order to have a real experimental quantum random number generator, a setup shown in figure 6.316 was built in the lab. To simulate a single photon source we have a CW-Pump laser with 1550 nm wavelength followed by an interferometer Mach-Zenhder in order to have a pulsed beam. The interferometer has an input signal given by a Pulse Pattern Generator. This device also gives a clock signal for the Single Photon Detector (APD-Avalanche Photodiode) which sets the time during which the window of the detector is open. After the MZM there is a Variable Optical Attenuator (VOA) which reduces the amplitude of each pulse until the probability of one photon per pulse is achieved. Next, there is a polarizer controller followed by a Linear Polarized, which is set at 45°, then a Polarization Beam Splitter (PBS) and finally, one detector at the end of each output of the PBS. The output signals from the detector will be received by a Processing Unit. Regarding to acquired the output of the detectors, there is an oscilloscope capable of record  $1 \times 10^6$  samples.

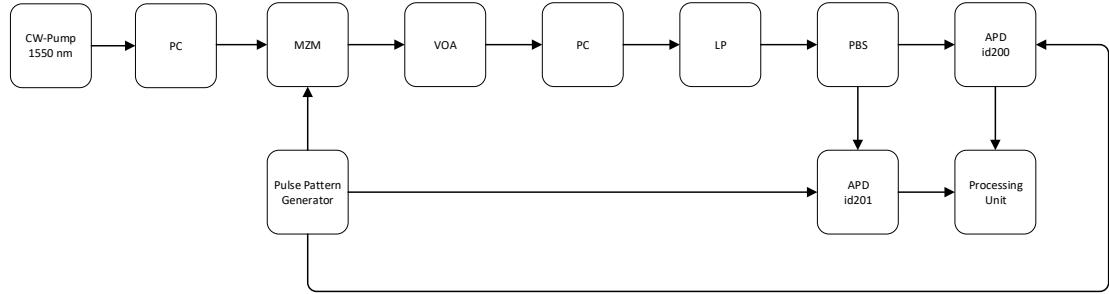


Figure 6.316: Experimental setup to implement a quantum random number generator.

#### 6.13.3.1 IDQuantique detector

The detector used in the laboratory is the Thorlabs PDB 450C. This detector consists of two well-matched photodiodes and a transimpedance amplifier that generates an output voltage (RF OUTPUT) proportional to the difference between the photocurrents of the photodiodes. Additionally, the unit has two monitor outputs (MONITOR+ and MONITOR-) to observe the optical input power level on each photodiode separately.

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

$$S_n = e^{-\mu} \frac{\mu^n}{n!}, \quad (6.216)$$

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

$$P = 1 - S_0 = 1 - e^{-\mu}. \quad (6.217)$$

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

$$P_{click} = P_{det} + P_{dc} + P_{det}P_{dc}, \quad (6.218)$$

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

$$P_{det} = 1 - e^{-\eta_D \mu}. \quad (6.219)$$

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

$$f_{click} = f_{trigger} P_{click} \rightarrow P_{click} = \frac{f_{click}}{f_{trigger}}. \quad (6.220)$$

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

$$\mu = -\frac{1}{\eta_D} \ln \left[ 1 - \frac{1}{1 - P_{dc}} \left( \frac{f_{click}}{f_{trigger}} - P_{dc} \right) \right] \quad (6.221)$$

#### **6.13.4 Open Issues**

- Experimental Implementation.
- Random number validation/standardization.

## References

- [1] GE Katsoprinakis et al. "Quantum random number generator based on spin noise". In: *Physical Review A* 77.5 (2008), p. 054101.
- [2] Thomas Jennewein et al. "A fast and compact quantum random number generator". In: *Review of Scientific Instruments* 71.4 (2000), pp. 1675–1680. DOI: [10 . 1063 / 1 . 1150518](https://doi.org/10.1063/1.1150518).
- [3] Mark Fox. *Quantum Optics, an Introduction*. Oxford, University Press, 2006.

## 6.14 BB84 with Discrete Variables

|                      |  |
|----------------------|--|
| <b>Students Name</b> | : Mariana Ramos (7/11/2017 - 9/4/2018)         |
|                      | Kevin Filipe (7/11/2017 - 10/11/2017)          |
| <b>Starting Date</b> | : November 7, 2017                             |
| <b>Goal</b>          | : BB84 implementation with discrete variables. |

BB84 is a key distribution protocol which involves three parties, Alice, Bob and Eve. Alice and Bob exchange information between each other by using a quantum channel and a classical channel. The main goal is continuously build keys only known by Alice and Bob, and guarantee that eavesdropper, Eve, does not gain any information about the keys.

### 6.14.1 Protocol Analysis

|                      |   |
|----------------------|---|
| <b>Students Name</b> | : Kevin Filipe (7/11/2017 - 10/11/2017) |
| <b>Goal</b>          | : BB84 - Protocol Description           |

BB84 protocol was created by Charles Bennett and Gilles Brassard in 1984 [1]. It involves two parties, Alice and Bob, sharing keys through a quantum channel in which could be accessed by a eavesdropper, Eve. A basic model is depicted in figure 6.317.

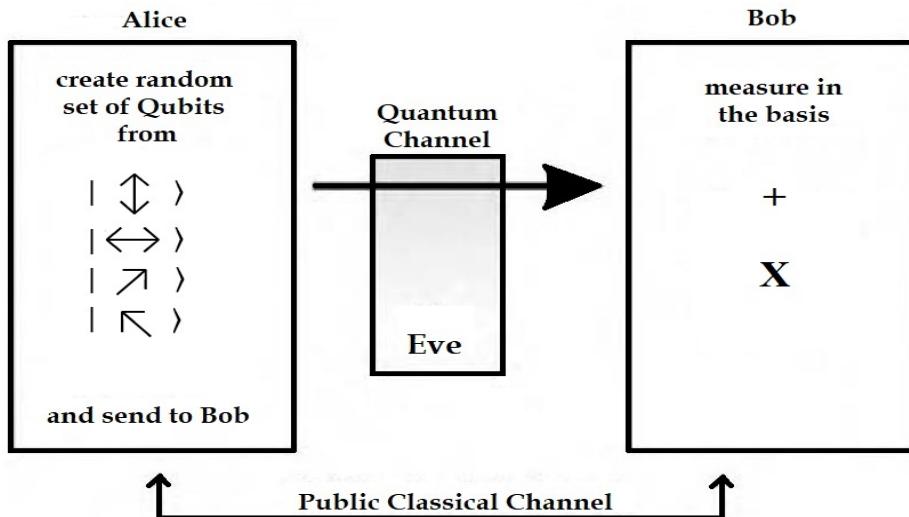


Figure 6.317: Basic QKD Model. Alice and Bob are connected by 2 communication channels, a public quantum channel and a authenticated classical channel, with an eavesdropper, Eve (figure adapted from [2]).

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

| Bit | <i>Rectilinear Basis, +</i> | <i>Diagonal Basis, ×</i> |
|-----|-----------------------------|--------------------------|
| 0   | 0                           | -45                      |
| 1   | 90                          | 45                       |

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

Table 6.45: Initial Parameters.

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

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

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

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

2. Alice transmits a train of photons,  $S_{AB}$ , obtained by encoding the bits,  $R_{A1}$  with the respective photon polarization state  $R_{A2}$ .

$$S_{AB} = \{\rightarrow, \uparrow, \nwarrow, \rightarrow, \nwarrow, \nearrow, \nearrow, \uparrow, \nwarrow, \nearrow, \nwarrow, \uparrow, \nwarrow, \rightarrow, \rightarrow, \uparrow, \nearrow, \rightarrow, \nearrow, \uparrow\}.$$

3. Bob generates a random string,  $R_B$ , to receive the photon trains with the correspondent basis.

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

4. Bob performs the incoming photon states measurement,  $M_B$ , with its generated random basis,  $R_B$ . If the two photon detectors don't click, means the bit was lost during transference due to attenuation. If both photon detectors click, a false positive was detected. In the measurements,  $M_B$ , the no-click in both detectors is represented by a -1 and the false positives to -2. The measurements done in rectilinear or diagonal basis are represented by 0 or 1, respectively. This is represented 6.318

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

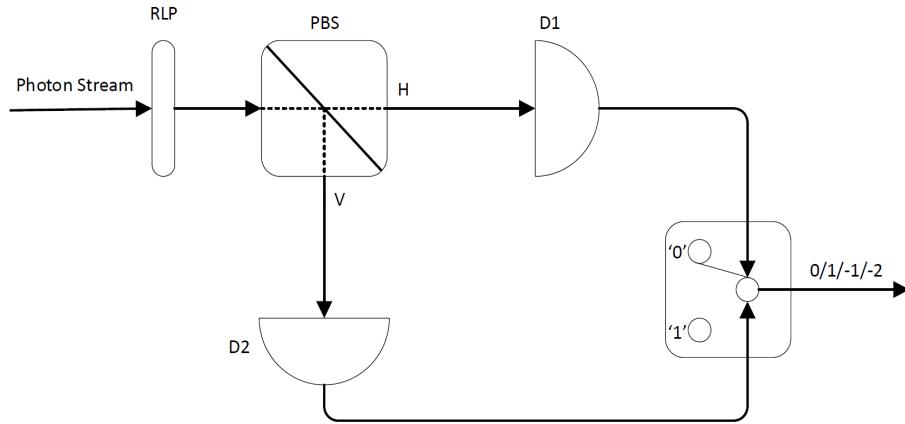


Figure 6.318: Single-Photon Detection block with false-positives, -2, and attenuation, -1, detection depending on D1 and D2 output.

5. After the measurement, Bob sends to Alice, using the classical channel, the used basis values,  $R_B$  with the attenuation, -1, and false positives,-2.
6. Alice performs a modified negated XOR, generating a sequence that detects when the same basis she used  $B_{AB}$ .

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

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

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

8. Alice then by having knowledge of  $R_{A2}$  and  $B_{AB}$  performs a scrambling algorithm over the deduced key. It is generated a matrix  $M \times N$ , according to the input parameter. Assuming a scrambling matrix of 3x4, 6.46. And being the scramble key represented as  $KS_{AB}$

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

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

Table 6.46: Scrambling matrix

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | - | - |

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

To determine the QBER, it is necessary to know the confidence interval parameter,  $\alpha$  and the QBER limit, in which states the maximum allowed QBER by the user. Then to verify if the channel is reliable or not, the flowchart presented in figure 6.319.

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

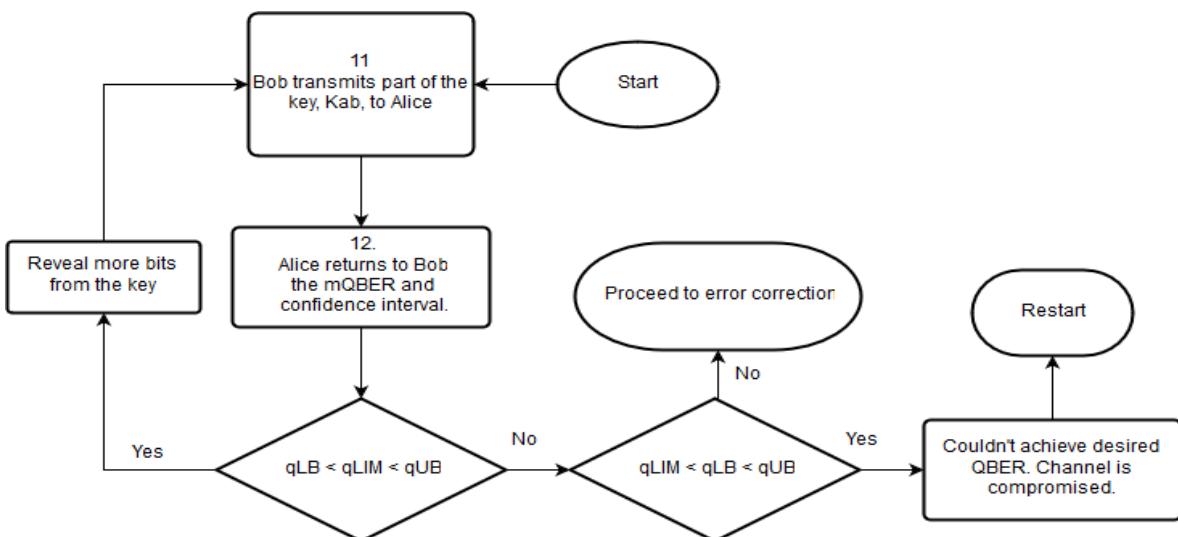


Figure 6.319: Flowchart to determine if the channel is reliable or not.

### 6.14.2 Simulation Analysis

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

In this sub section the simulation setup implementation will be described in order to implement the BB84 protocol. In figure 6.320 a top level diagram is presented. Then it will be presented the block diagram of the transmitter block (Alice) in figure 6.321 and the receiver block (Bob) in figure 6.322. In a first approach, we do not consider the existence of eavesdropper.

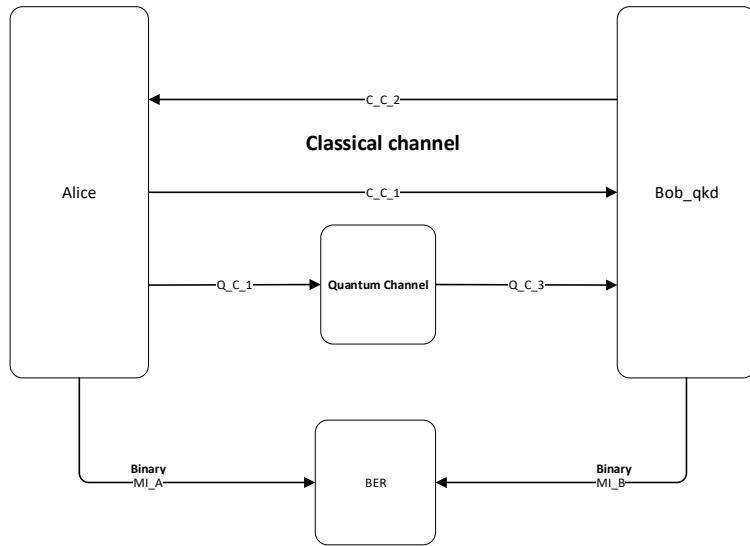


Figure 6.320: Simulation diagram at Alice's side

Figure 6.320 presents the top level diagram of our simulation. The setup contains two parties Alice and Bob, where the communication between them is done throughout two authenticated classical channels and one public quantum channel. In a first approach we will perform the simulation without eavesdropper presence. Furthermore, for bit error rate calculation between Alice and Bob.

In figure 6.321 one can observe a block diagram of the simulation at Alice's side. As it is shown in the figure, Alice must have one block for random number generation which is responsible for basis generation to polarize the photons, and for key random generation in order to have a random state to encode each photon. Furthermore, she has a Processor block for all logical operations: array analysis, random number generation requests, and others. This block also receives the information from Bob after it has passed through a fork's block. In addition, it is responsible for set the initial length  $l$  of the first array of photons which will send to Bob. This block also must be responsible for send classical information

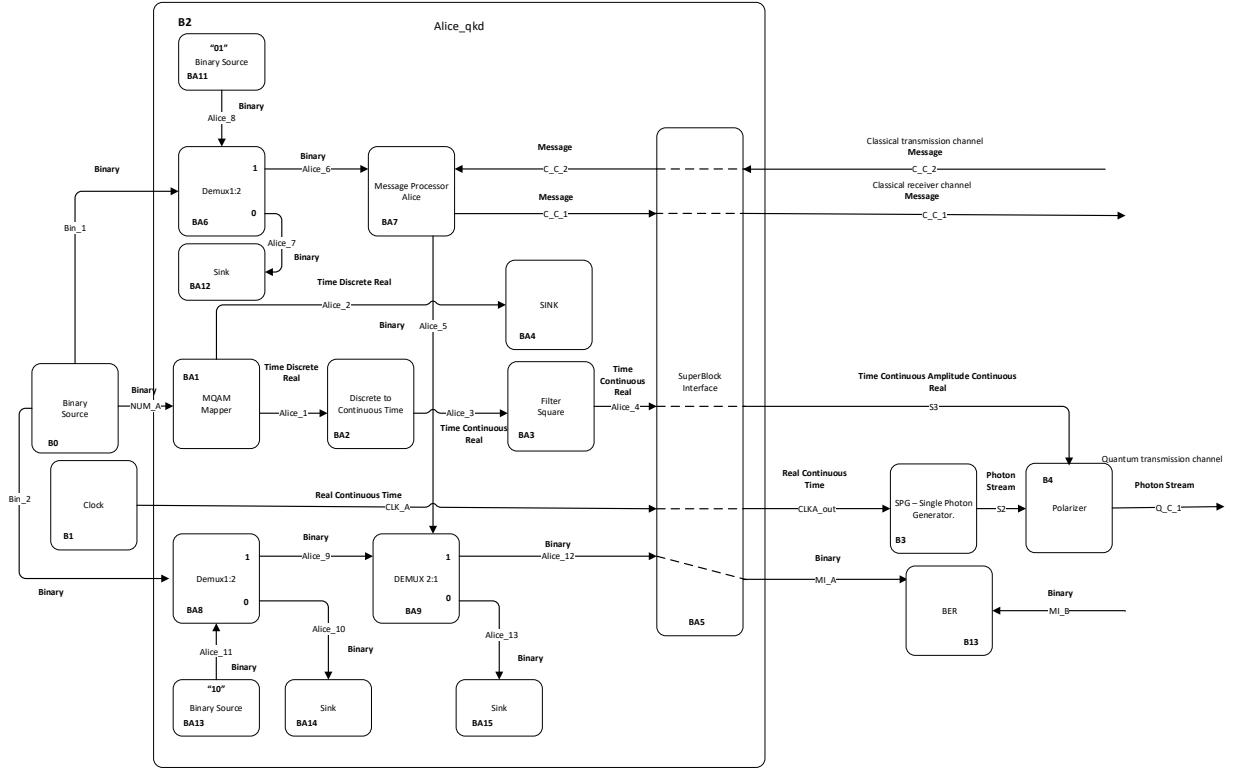


Figure 6.321: Simulation diagram at Alice's side

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

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

In figure 6.321 one can observe a block diagram of the transmitter. As it is shown in the figure, the transmitter must have one block for random number generation (binary source) which is responsible for basis generation to polarize the photons, and for key random generation in order to have a random state to encode each photon. This block has three outputs which will be inputs for the super block Alice. Furthermore, Alice block is responsible for all logical operations: random single photons state values generation, receive and send messages to the receiver Bob by using the classical channels, binary output for mutual information calculations. Each block of the super block is described in Library chapter. Finally, Alice block will also send a real continuous time signal to single photon generator (clock sets the rate of photons generation), in order to generate photons polarized in the horizontal axis by default. Therefore, the transmitter has one more block, the polarizer

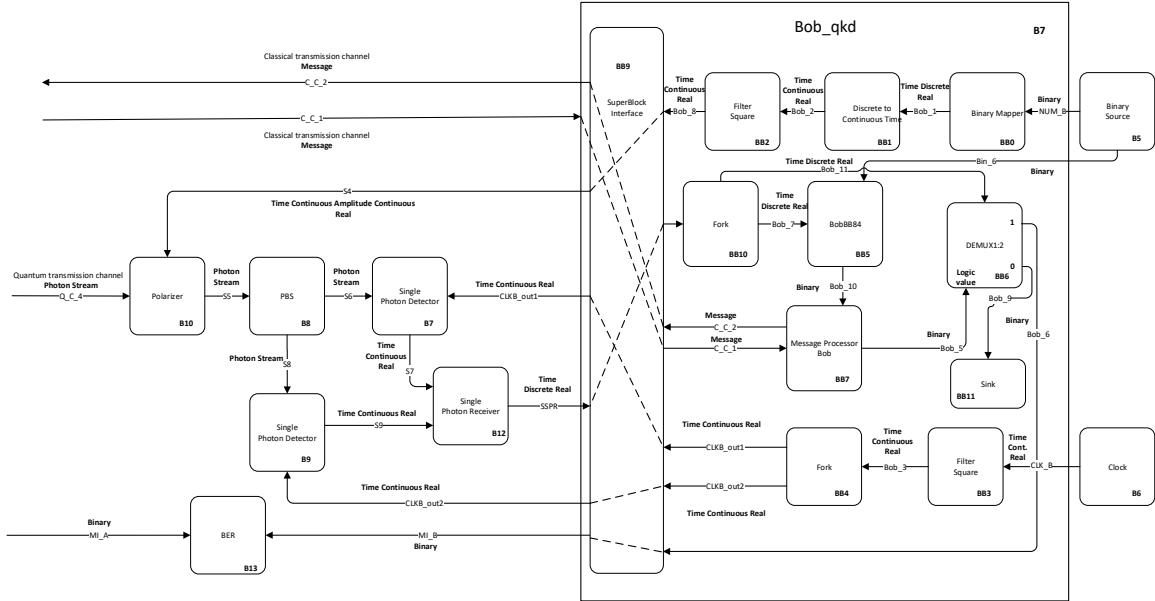


Figure 6.322: Simulation diagram at Bob's side

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

In figure 6.322 one can see a block diagram of the simulation for receiver (Bob). The receiver has one block for Random Number Generation which is responsible for randomly generate basis values which Bob will use to measure the photons sent by Alice throughout the quantum channel. Like transmitter, the receiver has the Bob block responsible for receive and send messages through the classical channel, receive single photons values detection from the single photon detectors, provides a clock signal to the detectors and send binary values for mutual information calculation. Furthermore, the receiver has two blocks for single photon detection (one for horizontal detection and other for vertical detection) which receives from Bob block a real continuous time signal which will set the detection window for the detector and outputs for Bob block the result value for detection. In addition, there is a polarizer which receives from Bob block a time continuous real signal which provides information about the rotation angle. If the basis chosen by Bob is the diagonal basis he sends "45°", otherwise sends "0°". The polarization beam splitter divides the input photon stream in horizontal component and vertical component.

Table 6.47: System Signals

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

Table 6.54 presents the system signals as well as them type.

Table 6.48: System Input Parameters

| Parameter                    | Default Value               | Description  |
|------------------------------|-----------------------------|--|
| rateOfPhotons                | 1000 photons/s              | Number of photon per sample.   |
| iqAmplitudeValues            | {-45,0},{0,0},{45,0},{90,0} | Possible photon states.  |
| numberOfSamplesPerSymbol     | 16                          | Number of samples per symbol.  |
| detectorWindowTimeOpen       | 0.2 ms                      | smaller than 1 ms  |
| detectorPulseDelay           | 0.7 ms                      | in units of ms   |
| detectorProbabilityDarkCount | 0.0                         | Probability of dark counts in single-photon detector.                            |
| rotationAngle                | 0.0                         | Polarization angle in XY axis to introduce in Deterministic SOP changes.         |
| elevationAngle               | 0.0                         | Polarization angle in Poincare sphere to introduce in Deterministic SOP changes. |
| fiberLength                  | 10 km                       | Length of the optical fibre in km.   |
| fiberAttenuation             | 0.2 dB/km                   | Attenuation of the optical fibre in dB/km.                                       |

Table 6.49: Header Files

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

Table 6.50: Source Files

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

#### 6.14.2.1 Simulation Results

Figure 6.323 represents the block diagram of the first simulation performed between Alice and Bob. This simulation intends to simulate the communication protocol between Alice and Bob until they do the Basis Reconciliation. At this time, it is not taken into account any attack from an eavesdropper. However, as one can learn from theoretical protocol analysis, the attenuation due the fiber losses, dark counts probabilities from single photon detectors and the SOP drift over the quantum channel are all taken into account.

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

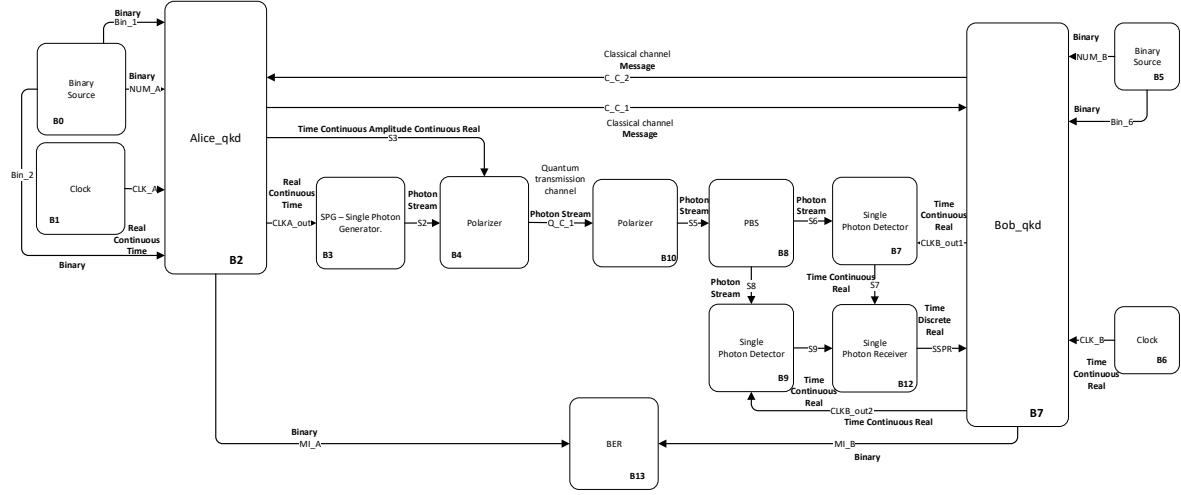


Figure 6.323: Diagram block of simulation performed between Alice and Bob until Basis Reconciliation.

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

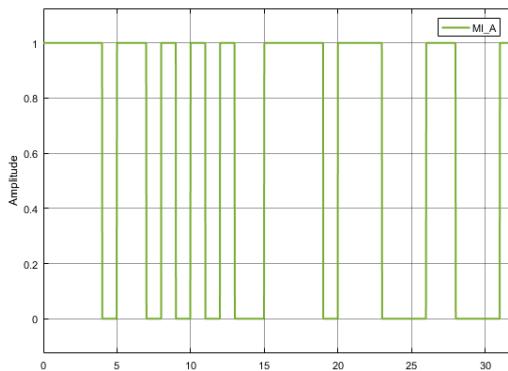


Figure 6.324: MI\_A signal.

Figure 6.324 and figure 6.325 represent the sequence of bits which will be used by Alice to encode the messages and the sequence of bits used by Bob to decode the message when no errors in quantum channel are taken into account, respectively. As one can see the two

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

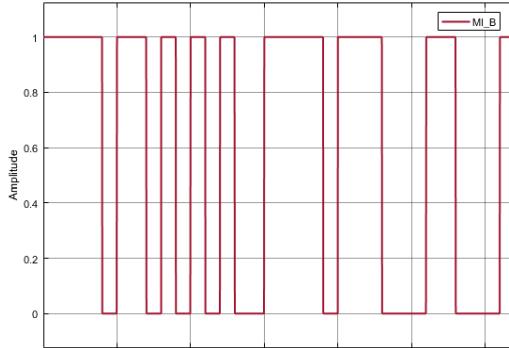


Figure 6.325: MI\_B signal.

As one can see in figure 6.340 a block which calculates QBER is connected to Alice and Bob. This block calculates the QBER between the measurements that Bob performed with the same basis as Alice, based on method described in [3]. Thus, as expected, the QBER is 0% when no errors are taken into account.

Next, some errors due the changes in state of polarization of the single photons transmitted between Alice and Bob were added. This way, a polarization rotator in the middle of the quantum channel was added, which is controlled by a SOP modulator block as it is shown in figure 6.326 with modelled with deterministic [4] and stochastic [5] methods. Additional information about the blocks presented in this quantum channel can be found in library chapter.

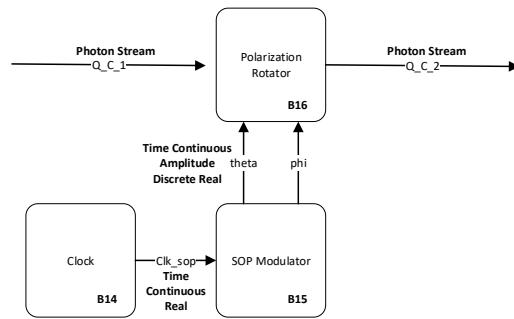


Figure 6.326: Quantum channel diagram.

Now, it is important to calculate the QBER as a function of the rotation angle  $\theta$ . In order to do that, it was simulated a deterministic SOP modulation, in which the  $\theta$  angle varies over the time. In figure 6.369 is presented the variation in the value of QBER with respect with theta changes from  $0^\circ$  to  $45^\circ$ . Theoretically, QBER corresponds to the probability of errors in

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

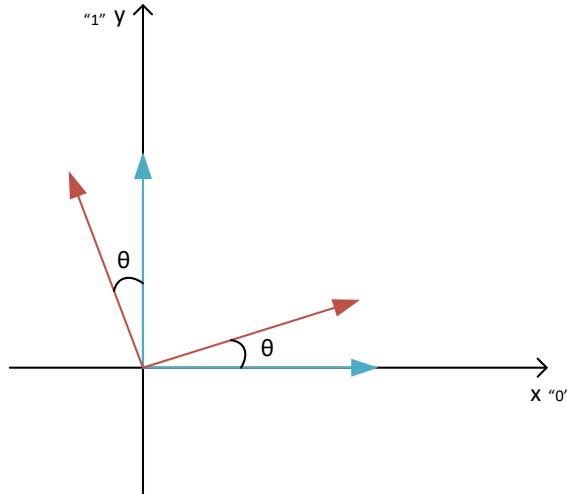


Figure 6.327: Representation of two orthogonal states rotated by an angle  $\theta$ .

Figure 6.368 presents the graphical representation of two orthogonal states rotated by an angle  $\theta$ . This rotation is induced by the SOP modulator block which selects a deterministic  $\theta$  and  $\phi$  angles that do not change over the time. This same rotation is applied for all sequential samples. From figure 6.368 the theoretical QBER can be calculated using the following equation:

$$QBER = P(0)P(1|0) + P(1)P(0|1). \quad (6.222)$$

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

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

This way,

$$QBER = \frac{1}{2}\sin^2(\theta) + \frac{1}{2}\sin^2(\theta) \quad (6.223)$$

$$QBER = \sin^2(\theta). \quad (6.224)$$

In figure 6.369 are represented two curves: QBER calculated from simulated data and QBER calculated using theoretical model from equation 6.242. Furthermore, the cross correlation coefficient between the two signals was calculated using a function from MATLAB `xcorr(x,y,'coeff')` which the result is 99.92%. From that, we can conclude that the QBER calculated from simulated data follows the theoretical curve with high correlation.

Nevertheless, the error bars presented in figure 6.369 were calculated based on a confidence interval of 95%.

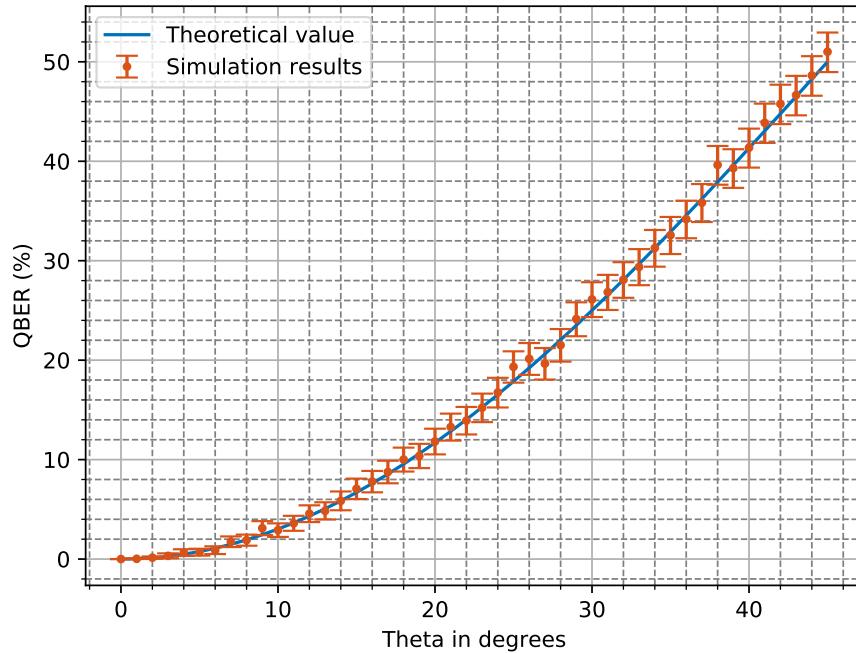


Figure 6.328: QBER evolution in relation with deterministic SOP drift.

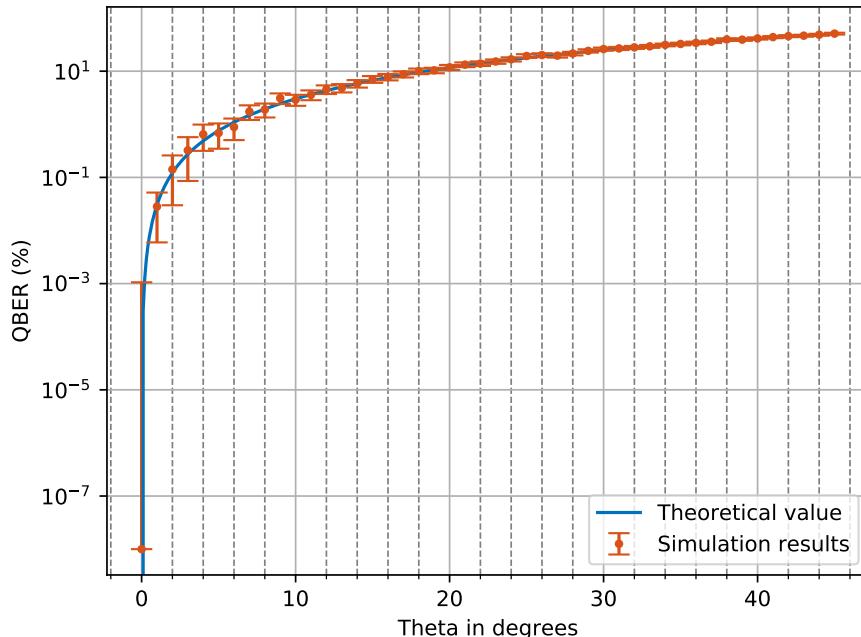


Figure 6.329: QBER evolution in relation with deterministic SOP drift in log scale.

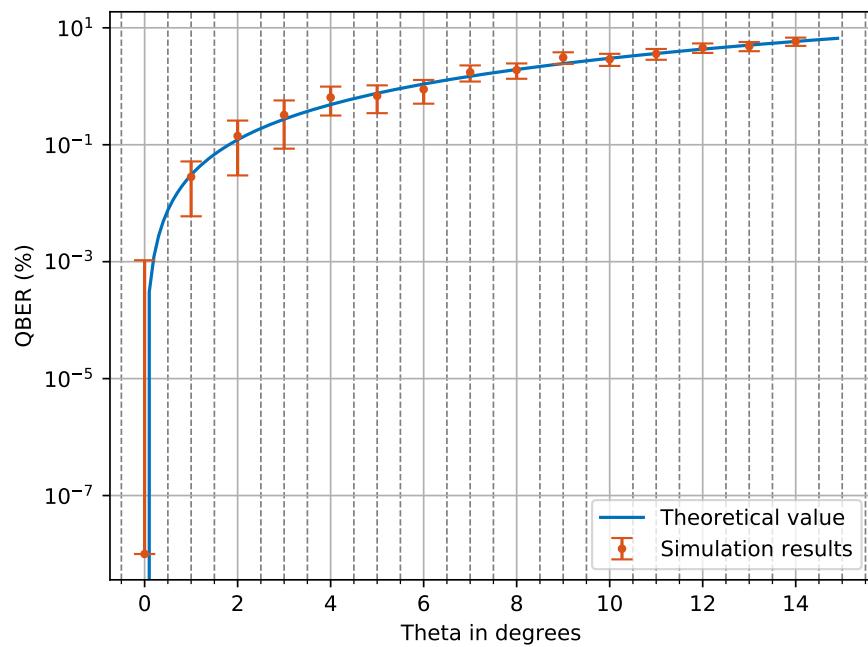


Figure 6.330: QBER evolution in relation with deterministic SOP drift scaled.

### 6.14.3 Open Issues

1. Estimation of the photons received rate (raw key rate), from the SSPR signal.
2. Estimation of the key rate after basis reconciliation, from MI\_B.
3. Include the random polarization rotations in the communication channel.
4. Implementation of the control system for polarization rotations.
5. Implementation of a QBER estimation protocol.
6. Insertion of a new block in Bob\_qkd super block, after the DEMUX\_2\_1 so that we can choose the bits that are going to be used in BER estimation.
7. Implementation of the scrambling algorithm in order to spread the errors.
8. Implementation of the cascade for error correction.
9. Implementation of the output which represents the final key that is built.
10. Introduce EVE in simulation as shown in figure 6.331.

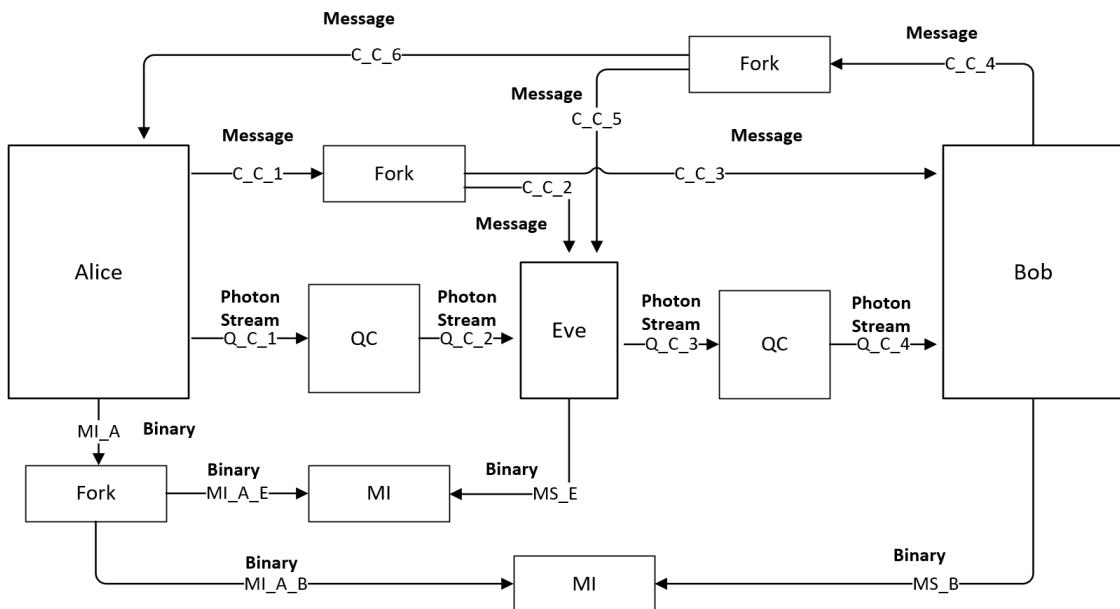


Figure 6.331: Simulation diagram at a top level

11. Analyze different strategies for Eve.
12. Experimental Implementation.

## References

- [1] Charles H Bennet. "Quantum cryptography: Public key distribution and coin tossing". In: *Proc. of IEEE Int. Conf. on Comp., Syst. and Signal Proc., Bangalore, India, Dec. 10-12, 1984.* 1984.
- [2] Christopher Gerry and Peter Knight. *Introductory quantum optics*. Cambridge university press, 2005.
- [3] Nelson J Muga, Mário FS Ferreira, and Armando N Pinto. "QBER estimation in QKD systems with polarization encoding". In: *Journal of Lightwave Technology* 29.3 (2011), pp. 355–361.
- [4] Nelson J Muga and Armando N Pinto. "Extended Kalman filter vs. geometrical approach for stokes space-based polarization demultiplexing". In: *Journal of Lightwave Technology* 33.23 (2015), pp. 4826–4833.
- [5] Cristian B Czegledi et al. "Polarization drift channel model for coherent fibre-optic systems". In: *Scientific reports* 6 (2016), p. 21217.

## 6.15 Quantum Oblivious Key Distribution with Discrete Variables

|                     |   |   |
|---------------------|---|---|
| <b>Authors Name</b> | : | Mariana Ramos (2017/09/18 - )   |
|                     | : | Armando Pinto (2017/09/18 - )   |
| <b>Goal</b>         | : | Quantum oblivious key distribution (QOKD) implementation with discrete variables. |
| <b>Directory</b>    | : | sdf/ot_with_discrete_variables.   |

Oblivious Transfer (OT) is a fundamental primitive in multi-party computation. The one-out-of-two OT consists in a communication protocol between Alice and Bob. At the beginning of the protocol Alice has two messages  $m_1$  and  $m_2$  and Bob wants to know one of them,  $m_b$ , without Alice knowing which one, i.e. without Alice knowing  $b$ , and Alice wants to keep the other message private, i.e. without Bob knowing  $m_{\bar{b}}$ . Therefore two conditions must be fulfilled:

1. The protocol must be concealing, i.e at the beginning of the protocol Bob does not know nothing about Alice's messages, while at the end of the protocol Bob will learn the message  $m_b$  chosen by him.
2. The protocol is oblivious, i.e Alice cannot learn anything about Bob's choice, bit  $b$ , and Bob cannot learning nothing about the other message  $m_{\bar{b}}$ .

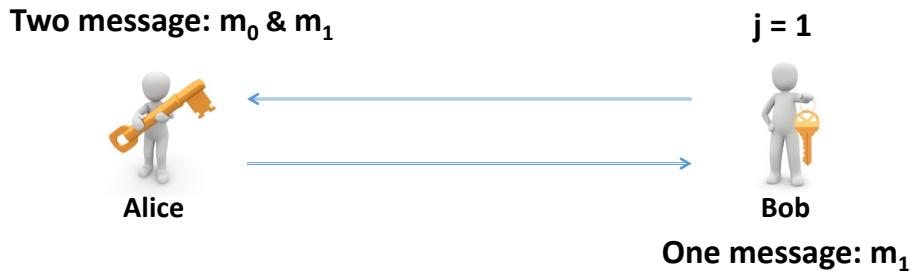


Figure 6.332: 1-out-of-2 Protocol

In order to implement OT between two parties (Alice and Bob) they must be able to exchange continuously oblivious keys, i.e a QOKD system must exist between them.

### 6.15.1 Theoretical Description

#### 6.15.1.1 Quantum Oblivious Key Distribution System (QOKD)

In this section we are going to describe the Quantum Oblivious Key Distribution system (QOKD). The QOKD system enables two parties (Alice and Bob) to share a set of keys. These



Figure 6.333: 1-out-of-2 Protocol

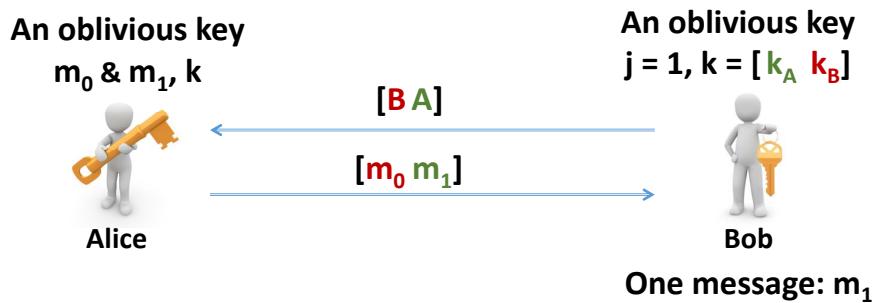


Figure 6.334: 1-out-of-2 Protocol

keys have the particularity of being half right and half wrong. Only Bob knows which are right and wrong keys.

Considering a discrete variables implementation, both Alice and Bob agree with the following correspondence, where + corresponds to *Rectilinear Basis* and  $\times$  corresponds to *Diagonal Basis*,

| Basis |          |
|-------|----------|
| 0     | +        |
| 1     | $\times$ |

Alice and Bob also agree with the bit correspondence for each direction for each basis. For *Rectilinear basis*, "+",

|   | Basis "+"               |
|---|-------------------------|
| 0 | $\rightarrow (0^\circ)$ |
| 1 | $\uparrow (90^\circ)$   |

and for *Diagonal Basis*, " $\times$ ",

|   | Basis "x"              |
|---|------------------------|
| 0 | $\searrow (-45^\circ)$ |
| 1 | $\nearrow (45^\circ)$  |

1. The first step is to establish for both Alice and Bob the block length  $l$ . In this case, lets assume  $l = 16$ . Alice randomly generate a bit sequence with length  $l$ . Therefore, she must define two sets randomly:  $S_{A1}$  which contains the basis values; and  $S_{A2}$ , which contains the key values.

In that case, lets assume she generates the following sets  $S_{A1'}$  and  $S_{A2'}$ :

$$S_{A1'} = \{0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1\},$$

$$S_{A2'} = \{1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1\}.$$

2. Next, Alice sends to Bob throughout a quantum channel  $l$  photons encoded using the basis defined in  $S_{A1'}$  and according to the key bits defined in  $S_{A2'}$ .

Therefore, in the current example, Alice sends the following photons,

$$\begin{aligned} S_{AB} &= \{\uparrow, \uparrow, \nearrow, \searrow, \rightarrow, \rightarrow, \searrow, \nearrow, \uparrow, \rightarrow, \searrow, \nearrow, \downarrow, \uparrow, \nearrow\} \\ &= \{90^\circ, 90^\circ, 45^\circ, -45^\circ, -45^\circ, 0^\circ, 0^\circ, -45^\circ, 45^\circ, 90^\circ, 0^\circ, -45^\circ, 45^\circ, -45^\circ, 90^\circ, 45^\circ\}. \end{aligned}$$

3. Bob also randomly generates  $l = 16$  bits, which are going to define his measurement basis,  $S_{B1'}$ . Lets assume,

$$\begin{aligned} S_{B1'} &= \{0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1\} \\ &= \{+, \times, \times, +, +, \times, +, \times, +, \times, +, +, +, \times\}. \end{aligned}$$

Bob will get  $l$  results:

$$S_{B2'} = \{1, -, \underline{0}, 0, -, 1, \underline{1}, -, 1, -, 1, 0, 1, 1, \underline{0}, 1\}.$$

The " $-$ " corresponds to no clicks in Bob's detector, due to attenuation. The underlined values are bits which were measured with a correct basis but an error has occurred due to imperfections in the quantum communication system.

4. Bob is going to send a " $-1$ " or a hash value to Alice for each measurement that he performed, thereby being " $-1$ " the measurements which correspond to no clicks. In this case, we are going to assume that the hash value is calculated using the SHA-256 algorithm [1]. In detail, Bob has two sets  $S_{B1'}$  and  $S_{B2'}$  and he is going to generate the set  $S_{BH1}$  with  $l$  values (" $-1$ " or hash values calculated for each position of  $S_{B1'}$  with the correspondent position of  $S_{B2'}$ ). Therefore, Bob will send to Alice the following set:

$$S_{BH1} = \{S_1, -1, S_2, S_3, -1, S_4, S_5, -1, S_6, -1, S_7, S_8, S_9, S_{10}, S_{11}, S_{12}\}.$$

5. Since Alice has received the confirmation of measurement from Bob, i.e after Alice has received  $S_{BH1}$ , she sends throughout a classical channel the basis which she has used to codify the photons updated with the information about the no received photons,

$$S_{A1'} = \{0, -1, 1, 1, -1, 0, 0, -1, 1, -1, 0, 1, 1, 0, 1\}$$

Due to attenuation, the previous sets are reduced to the length 12 and they shall be replaced by the following:

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

$$S_{A2} = \{1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1\},$$

$$S_{B1} = \{0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1\},$$

$$S_{B2} = \{1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1\}$$

Note that  $S_{B2}$  still has errors.

6. In order to know which photons were measured correctly, Bob does the operation  $S_{B3} = S_{B1} \oplus S_{A1}$ . In the current example,

$$\begin{array}{c|cccccccccccc} S_{B1} & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ \hline S_{A1} & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ \oplus & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \end{array}$$

In this way, Bob gets

$$S_{B3} = \{1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1\}.$$

When Bob uses the right basis he gets the values correctly, apart from possible errors in transmission, when he uses the wrong basis he just guess the value. The values "1" correspond to the values he measured correctly and "0" to the values he just guessed. Thus, Bob is building two sets of keys, one with correct basis measurements values and other with the wrong basis measurement values that he just guessed.

Thus, Bob has two pair of sets, one for the right basis,

$$S_{B_{rp}} = \{1, 2, 5, 6, 8, 11, 12\},$$

$$S_{B_{rb}} = \{1, 0, 1, 1, 0, 0, 1\},$$

where  $S_{B_{rp}}$  is the set of positions and  $S_{B_{rb}}$  is the set of bit values he measured for each position. The other pair is for photons he measured with the wrong basis and then he just guessed the values,

$$S_{B_{wp}} = \{3, 4, 7, 9, 10\},$$

$$S_{B_{wb}} = \{0, 1, 1, 1, 1\},$$

where  $S_{B_{wp}}$  is the set of positions and  $S_{B_{wb}}$  is the set of bit values he measured for each position.

Nevertheless, due to errors in transmission, some bits in  $S_{B_{rb}}$  may be not right.

At this point, in order to test Bob's honesty and to estimate the QBER of the channel, Alice is going to ask Bob to open some pairs of the Bob's sets. The definition of the protocol to test Bob's honesty is still an open issue. However, depending on the QBER estimated by her, Alice must have a parameter to set the number of right position she wants to open, i.e she must open a minimum number of right position in order to guarantee a minimum QBER. This will increase the security of the protocol. Alice chooses some positions to open and tells Bob which positions she wants to open. Bob sends to Alice the pairs she chose and then these pairs are eliminated from them sets. Lets assume she asked to open the positions 10, 11 and 12. If she concludes Bob is not being honest, she stops the protocol and they must start it again. Otherwise, the protocol continues. Lets assume Alice has verified these pairs using the hash function committed by Bob and concluded that he is being honest. Therefore, she sends to Bob the QBER estimated by her.

Now, Bob has the previous sets replaced by the following,

$$\begin{aligned} S_{B_{rp}} &= \{1, 2, 5, 6, 8\} \\ S_{B_{rb}} &= \{1, 0, 1, 1, 0\} \\ S_{B_{wp}} &= \{3, 4, 7, 9\} \\ S_{B_{wb}} &= \{0, 1, 1, 1\} \end{aligned}$$

Bob is going to use a modified version of *Cascade algorithm* to correct the errors due transmission.

#### 6.15.1.2 Modified version of Cascade Algorithm

The Cascade algorithm is often used with a key set where all values are supposed right. In this case, Bob has two pairs of sets, one with the position and bit values of photon he measured with the correct basis and other with position and bit values of photon he measured with the wrong basis. He only needs to apply the Cascade algorithm in the set that he measured the photons correctly [2]. However, he must apply a modified version of the Cascade in the other set in order to keep in secret from Alice which set corresponds to right and which set corresponds to wrong measurements.

Bob randomly generates a bit value. If he gets 0, he will send to Alice the set  $\{S_{B_{rp}}, S_{B_{wp}}\}$ . Otherwise, if he gets 1 he will send the set  $\{S_{B_{wp}}, S_{B_{rp}}\}$ . This guarantee that Alice does not know which is the right or wrong set. Lets assume this random bit is "0" and he sends  $\{S_{B_{rp}}, S_{B_{wp}}\}$ .

- (a) Bob starts by applying the normal cascade to the set  $S_{B_{rb}}$ . After both know the error estimative Bob determine if the error rate is above the fail threshold. If it is

truth they must start the procedure again. Lets assume the estimated error rate is acceptable. Bob and Alice use a random permutation which is represented in figure 6.335 for a larger number of bits (agreed at the beginning) by applying it to the shifted keys, in order to guarantee the spread out of the error bits randomly and to separate consecutive errors from each other.

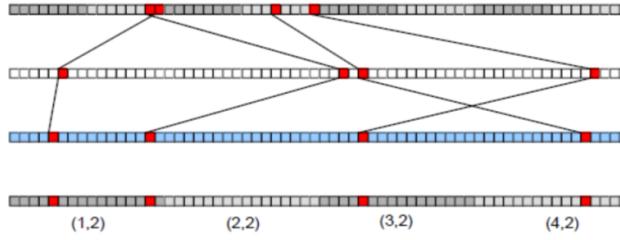


Figure 6.335: Cascade Algorithm - permutation

- (b) Bob and Alice divide all the shifted key bits into blocks of  $N$  bits depending on the estimated error rate in order to have one or no error per block. In general, the sets of keys are too large and it is easier to explain the algorithm based on a larger number of bits. Therefore, figure 6.336 represents the typical cascade initial steps. However, in this case, the set to be corrected only has five bits, therefore they divide the set in two sub-blocks, one with 3 bits and other with 2 bits.

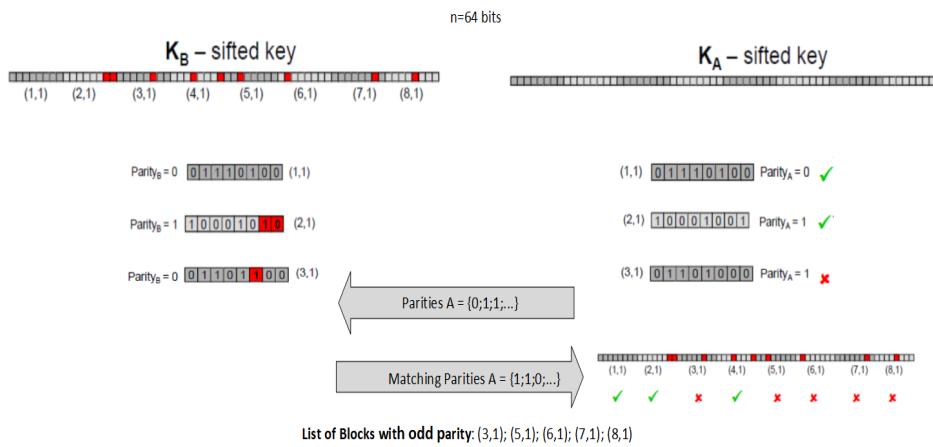


Figure 6.336: Cascade Algorithm

- (c) They use a classical channel to compare the block parities. For blocks with different parities, an odd number of errors must exist, otherwise an even number of errors would mask each other. Thus, the block in which the parities disagree is divided in half into two smaller blocks of length  $\frac{N}{2}$ , and another parity check is performed on the first sub-block, as one can see in figure 6.337. As it was referred above, there is at least one error in one sub-block being the error location revealed

by the parity of one sub-block. In other words, if the parity of the first sub-block passes, the error will be in the second sub-block. The sub-block with error will be sub-divided until the error is found.

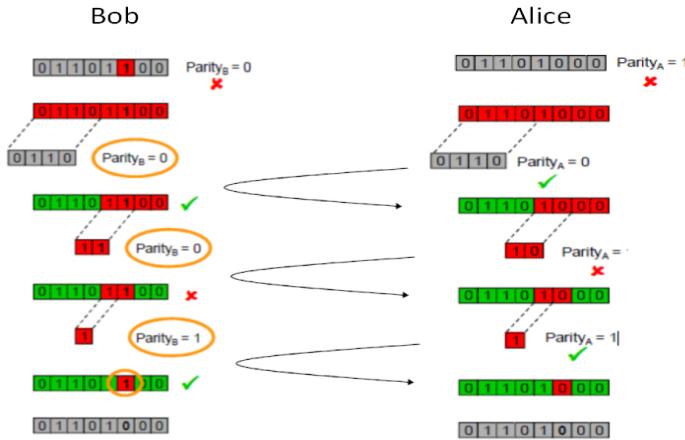


Figure 6.337: Cascade Algorithm - example of error correction

- (d) When the error is corrected, the last bit of the block is discard in order to prevent the gain of additional information by Bob.

In this case, lets assume the set of right positions was corrected with the algorithm described above and it will be replaced by the following:

$$\begin{aligned} S_{B_{rp}} &= \{1, 2, 5, 6, 8\} \\ S_{B_{rb}} &= \{1, 1, 0, 1, 0\} \end{aligned}$$

In order to test Alice's honesty, Bob must verify if the QBER sent by Alice is a realistic value. If it is not he stops the protocol and they must start again. Otherwise, the protocol continues.

After that, Bob needs to apply the Fake Cascade to the set  $S_{B_{wb}}$ . The main goal of this step is to convince Alice she is performing the real Cascade but she is not.

- (a) First of all, based on the positions contained in  $S_{B_{wb}}$ , Bob must build an array with the correspondent bits in a random order and informs Alice the order of positions. In order to best explain this version of the algorithm, lets assume a larger set of bits.

Bob sends to Alice throughout a classical channel the new positions order as if it were the permutation step represented in figure 6.335 in real Cascade algorithm.

- (b) Assuming each of them has a set with 32 bits randomly organized by Bob, they divide the supposed shifted keys in blocks with N bits according to the estimated

error rate. As the QBER is the same as for real cascade, Bob will assume the same number of errors, even if he starts for this modified version he can know the number of errors from QBER estimated by Alice.

- (c) Bob and Alice use a classical channel to compare the block parities. Alice sends to Bob her parity list. Based on Alice's parity list, Bob sends a block list with odd parities, i.e the blocks position in which parity supposed disagree. This list is randomly built based on the number of errors considered by Bob, i.e if he considered five errors from QBER estimative, he will distributed them randomly and after that he will fill the remaining spaces with even parities. Bob sends to Alice the set with the list of odd parities, i.e the list of sub-sets he has different parities than Alice.
- (d) The blocks with errors will be consecutively divided until they found the supposed errors. Since we have assumed there were five errors, this is the number of errors that Alice must supposedly correct.

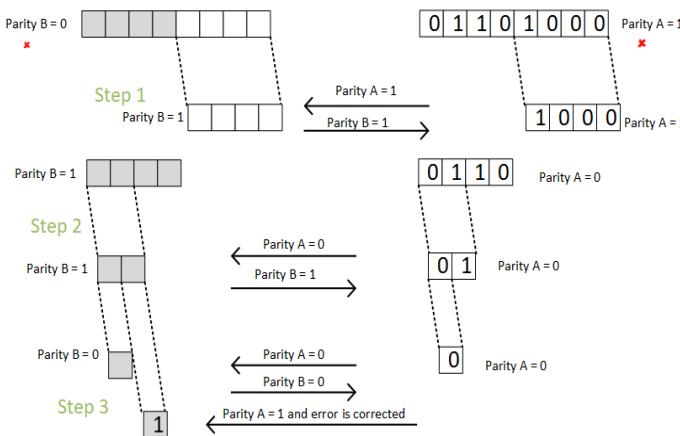


Figure 6.338: Fake cascade - example of error correction

Lets assume one of the blocks with error and analyse figure 6.338. Bob starts with a set filled with random bits, therefore we do not need to know which bits are. Alice starts by dividing her set in half with two blocks with  $N$  bits.

**Step 1:** Bob chooses one of the to blocks and informs Alice she must send the parity of this block. Lets assume he chose sub-block 2. She sends the parity and Bob is going to send his parity, which after know Alice's block parity he send the opposite parity. As referred in normal Cascade, there must be one error or no error in each block. Thus, since the parities disagree, the error must be in seconde block. They start the procedure to correct it.

**Step 2:** Bob divides again the sub-block in half with  $\frac{N}{2}$  bits and asks Alice for the parity of the first sub-block. She sends her parity equals to 0 and Bob sends to her the opposite parity again.

**Step 3:** They divide the sub-block in half again and Bob asks Alice for the parity

of the first bit. Alice sends to him the parity equals to her. As the error is not in the first bit, it must be in the second, therefore Bob is able to correct this bit with the information sent by Alice.

Note that Bob make his choice of which half analyse first using a random bit generator result. If he got "0" he starts with the first half of the sub-block, otherwise, if he got "1", he starts with the second half. In addition, they must discard the last bit of each block and sub-block in which fake Cascade were applied in order to guarantee that Bob does not gain additional information.

In this case, after apply the fake Cascade to  $S_{B_{wb}}$ , lets assume,

$$\begin{aligned} S_{B_{wp}} &= \{3, 4, 7, 9\} \\ S_{B_{wb}} &= \{0, 1, 1, 0\} \end{aligned}$$

If Bob starts by applying the fake Cascade, he must test Alice's honesty at the beginning of the real Cascade application, based on the number of errors he has. If he thinks that the QBER sent by Alice is unrealistic, he stops the protocol at this point.

7. When Alice sends to Bob a photons set, they are building a set of pairs (array positions and bit values which correspond to measured photons at Bob's side and to the key bit with the photon was encoded at Alice's side). The main goal is to guarantee that Bob has the same number of right and wrong pairs. In addition, they must know information about  $t$  (represented in figure 6.339) which corresponds to the points where the previous condition is verified.

Since Bob has sent to Alice the information about the smallest set, in this example, Alice know that there are four pairs of wrong positions and five pairs of right positions. Alice must destroy one of the right pairs by asking Bob to open it. Therefore, at  $t = 8$  both know that there are the same number of right and wrong pairs thereby being the main goal guaranteed.

As we can see in figure 6.339, unlike Bob, Alice does not know which positions corresponds to right or wrong measurements performed by Bob. They have been building these sets during all protocol.

#### 6.15.1.3 1-out-of-2 Oblivious Transfer Protocol with QOKD system

At this time, we are going to describe the oblivious transfer protocol with detail. As it was referred at the beginning, Alice sends two messages to Bob and he wants to know one of them. Alice does not know which message Bob wants and Bob only know the message he wants, i.e he does not know anything about the other message. Furthermore, only Alice knows information about messages  $m_0$  and  $m_1$ . In this case, lets assume the following two messages with size  $s = 4$ ,  $m_0 = \{0011\}$  and  $m_1 = \{0001\}$ . Alice must guarantee  $t = s \times 2$ . In order to do that, she must destroy the remaining pairs. In this case, there is no need to do that because they have a set for  $t = 8$  with the same number of wrong and right pairs.

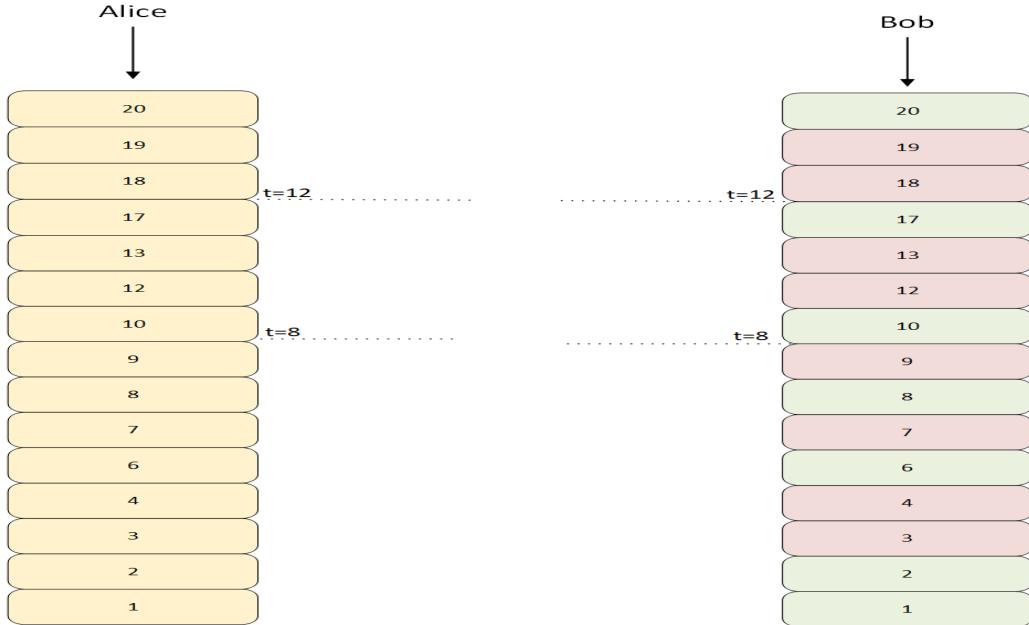


Figure 6.339: Alice and Bob key sets.

1. Bob defines two new sub-sets,  $I_0$  and  $I_1$ .  $I_0$  is a set of values with photons array positions which Bob just guessed the measurement since he did not measure them with the same basis as Alice,  $I_1$  is a set of values with photons array positions which Bob measured correctly since he used the same basis as Alice used to encoded them. The position of the pairs of each right and wrong message are in the keys sets that they have been building during the protocol.

In this example, the message size is 4. Since, at this time  $t = 10$  and we have 5 right pairs and 5 wrong pairs, Alice ask to Bob to open one right pair and one wrong pair in order to both have exactly the message's size number of right and wrong pairs. Lets assume that Alice opened two pairs, position 15 which is a wrong measurement and position 10 which is a right measurement. We have now  $t = 8$ .

Next, Bob defines two sub-sets with size  $s = 4$ :

$$I_0 = \{3, 4, 7, 9\},$$

and

$$I_1 = \{1, 2, 6, 8\},$$

where  $I_0$  is the sequence of positions in which Bob was wrong about basis measurement and  $I_1$  is the sequence of positions in which Bob was right about basis measurement. Bob sends to Alice the set  $S_b$

Thus, if Bob wants to know  $m_0$  he must send to Alice throughout a classical channel the set  $S_0 = \{I_1, I_0\}$ , otherwise if he wants to know  $m_1$  he must send to Alice throughout a classical channel the set  $S_1 = \{I_0, I_1\}$ .

2. Alice is sure about Bob's honesty, since she knows he only has 4 right basis to measure the photons. In addition, Alice cannot know which message Bob chose because she did not know the order that he sent the sets.
3. Lets assume Bob sent  $S_0 = \{I_1, I_0\}$ . Alice defines two encryption keys  $K_0$  and  $K_1$  using the values in positions defined by Bob in the set sent by him. In this example, lets assume:

$$K_0 = \{1, 1, 1, 0\}$$

$$K_1 = \{0, 0, 0, 1\}.$$

Alice does the following operations:

$$m = \{m_0 \oplus K_0, m_1 \oplus K_1\}.$$

$$\begin{array}{c|cccc} m_0 & 0 & 0 & 1 & 1 \\ \hline K_0 & 1 & 1 & 1 & 0 \\ \oplus & 1 & 1 & 0 & 1 \end{array}$$

$$\begin{array}{c|cccc} m_1 & 0 & 0 & 0 & 1 \\ \hline K_1 & 0 & 0 & 0 & 1 \\ \oplus & 0 & 0 & 0 & 0 \end{array}$$

Adding the two results,  $m$  will be:

$$m = \{1, 1, 0, 1, 0, 0, 0, 0\}.$$

After that, Alice sends to Bob the encrypted message  $m$  through a classical channel.

4. When Bob receives the message  $m$ , in the same way as Alice, Bob uses  $S_{B1}$ , values of positions given by  $I_1$  and  $I_0$  and does the decrypted operation. In this case, he does following operation:

$$\begin{array}{c|cccccccc} m & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ \oplus & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{array}$$

The first four bits corresponds to message 1 and he received  $\{0, 0, 1, 1\}$ , which is the right message  $m_0$  and  $\{0, 1, 1, 0\}$  which is a wrong message for  $m_1$ .

#### 6.15.1.4 Nearest Private Query

The Nearest Private Query is another example of QOKD application [Xu2017]. In this case, there are also two parties: a user (who we called Bob) and a data owner (who we called Alice). Bob has an input secret parameter  $x$  and Alice has a private data set  $A$ .

Lets assume Bob's secret input parameter  $x = 8$  and Alice's data set  $A = \{1, 2, 3, 6, 7, 10, 11, 14\}$ .

Bob wants to know which element  $(x_i)$  is the closest to  $x$  in Alice data set  $A$ , without revealing his secret  $x$ . Alice does not know which is the Bob's secret parameter, and Bob does not know any information about Alice's set except the closest element  $x_i$  to his  $x$ .

**Step 1** Alice generates a new set with  $N = 2^n$  elements,  $D(j)$  for  $j = 0, 1, \dots, N - 1$ , in which  $D(j) = x_l$  being  $x(l)$  the closest element to  $j$  in  $A$ .  $n$  is the number of bits that Alice needs to represent each element of her data set.

| $j$    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
|--------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| $D(j)$ | 1 | 1 | 2 | 3 | 3 | 6 | 6 | 7 | 7 | 10 | 10 | 11 | 11 | 14 | 14 | 14 |
|        | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
|        | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0  | 0  | 0  | 0  | 1  | 1  | 1  |
|        | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
|        | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0  | 0  | 1  | 1  | 0  | 0  | 0  |

**Step 2** Bob and Alice have a set of keys  $K_i^*$  and  $K_i$ , respectively, with 64 elements where 60 are bits resulted from wrong basis measurement and 4 resulted from correct basis measurement. This allows Alice to know that Bob is being honest. They start from QOKD with symmetric keys and then Alice is being asking Bob to destroy pairs of keys until they have a set with the characteristics above.

**Step 3** Bob sends to Alice the set  $S$  with all positions referred random measurements except the position bits at  $j = 8$ .

**Step 4** Alice encoded all bits with the corresponding bit keys at the positions sent by Bob, and then she sends the encoded message to Bob.

**Step 5** Bob receives the encoded message and apply an operation XOR based on the correspondent bits to the positions he sent to Alice above.

**Step 6** At the end, Bob gets the message he wanted, the closest element to 8, which corresponds to message 0, 1, 1, 1 or 7 and he remains know nothing about the other elements of Alice's data set. In the same way, Alice does not know which element Bob wants to know.

#### 6.15.1.5 QKD from QOKD

All Quantum Key Distribution systems can be obtained from the QOKD system presented in this report. The Quantum Oblivious Key Distribution system allows to obtain symmetric

secret keys and symmetric or asymmetric oblivious keys.

In fact, since Alice and Bob has the same set of keys and at some tab Alice knows that there are the same number of right and random bits, she can obtain any combination from this set by asking Bob to destroy some pairs.

QOKD has a big advantage over other QKD systems because of its versatility. However, the biggest disadvantage is related with a large number of photons consumption which can became the communication rate slower.

### 6.15.2 Simulation Analysis

First of all, the protocol will be simulated and then a experimental setup will be built in the laboratory.

The main goal of this simulation is to demonstrate that Bob was able to learn correctly message  $m_b$  and he does not know the message  $m_{\bar{b}}$ .

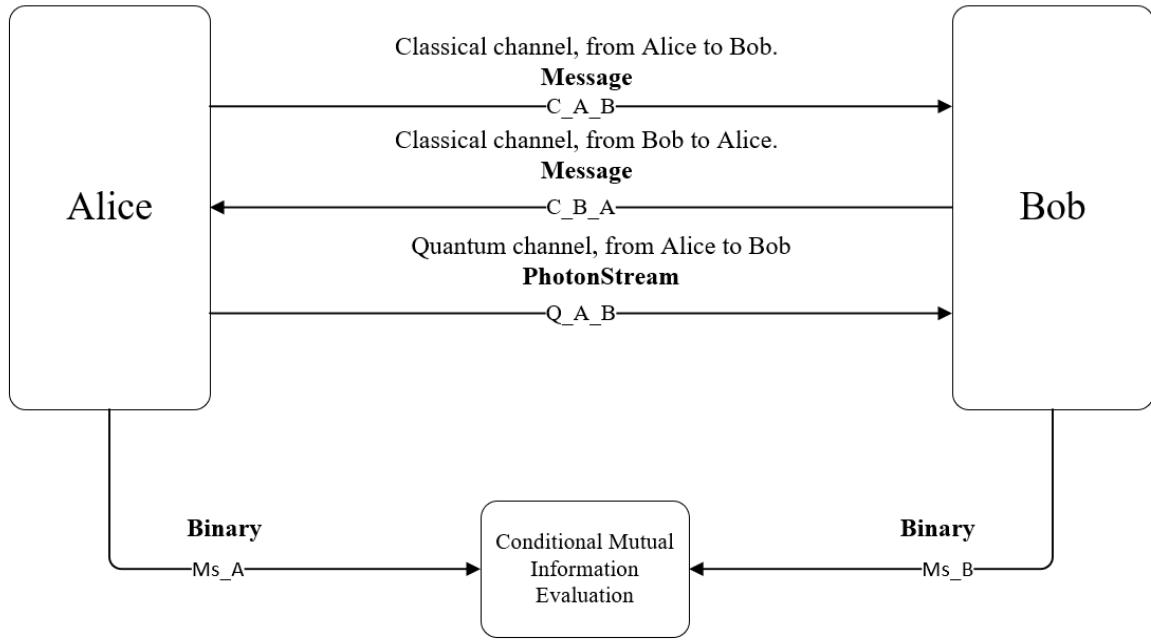


Figure 6.340: Simulation diagram at a top level

As one may see in figure 6.340 this simulation will have three top level blocks. Two of them are Alice and Bob and they are connected through two classical channels and one quantum channel. In addition, a third block will be implemented to calculate the *Mutual Information*. The mutual information (MI) between Alice and Bob is defined in terms of their joint distribution.

1. In figure 6.341 a block diagram of the simulation at Alice's side is shown. Alice has two additional blocks at her inputs: one block for random number generation which is responsible for basis generation to choose the polarization of the photons having

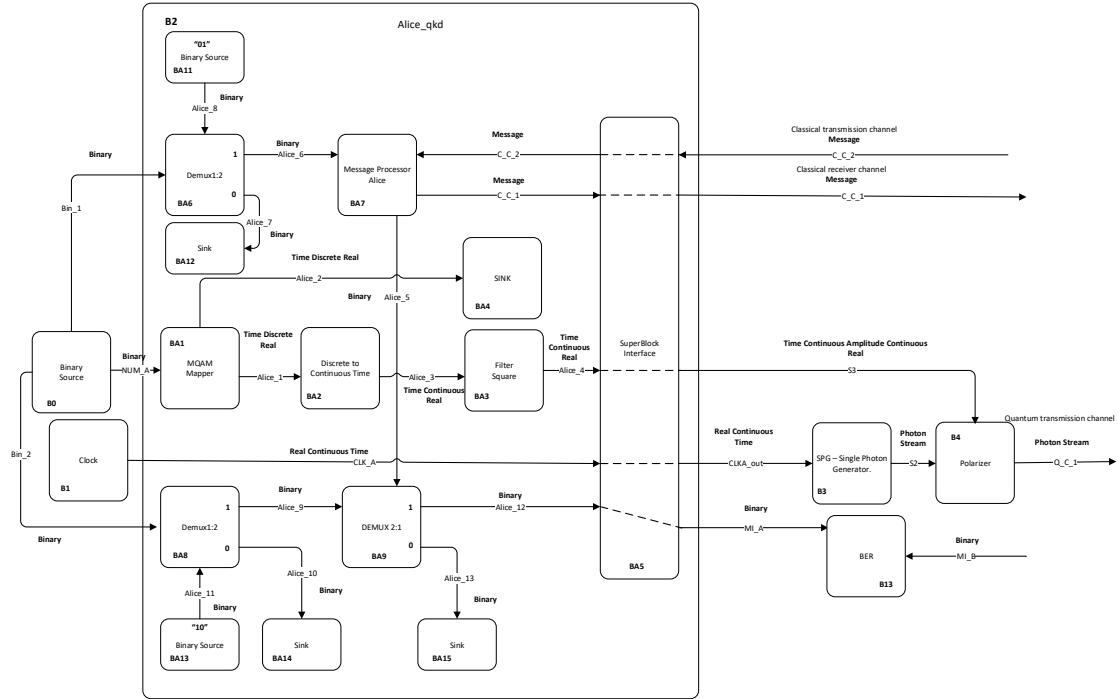


Figure 6.341: Simulation diagram at Alice's side.

a random state to encode each photon, and a second block to generate a clock signal which will keep Alice and Bob synchronized. The main block is a *Alice\_qkd* block responsible to process all information: classical messages to send to Bob and manage the quantum information to be send to Bob. The main block has an output to control the quantum channel and all the information which is sent through this channel. Alice has two more blocks for quantum tasks: the single photon generator and the polarizer block which is responsible to encode the photons generated from the previous block and send them throughout a quantum channel from Alice to Bob. Moreover, the polarizer is controlled by the main bloc, *Alice\_qkd*.

Finally, Alice's processor has an additional output signal to Mutual Information top level block, *MI\_A*.

2. In figure 6.342 a block diagram of the simulation at Bob's side is shown. From this side, Bob has one block for Random Number Generation which is responsible for randomly generate basis values which Bob will use to measure the photons sent by Alice throughout the quantum channel. Furthermore, this Block will generate the random bits that Bob needs in Modified Version of Cascade Algorithm. Like Alice, Bob has a main block, *Bob\_qkd*, responsible for all processing tasks, i.e Hash function generation, analysing functions, requests for random number generator block, etc.

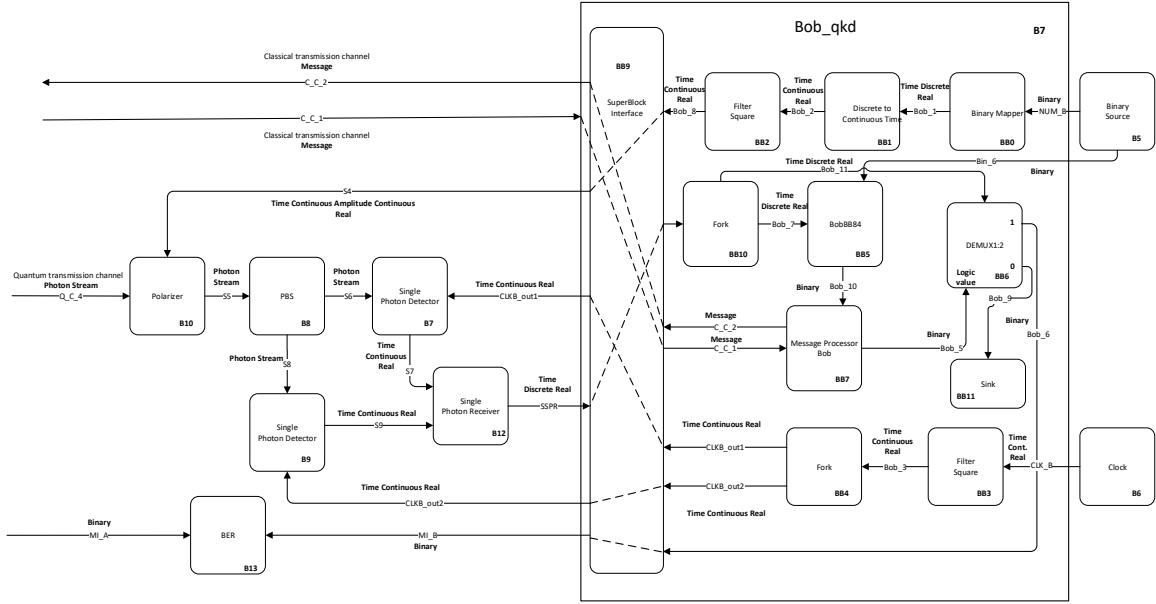


Figure 6.342: Simulation diagram at Bob's side.

Additionally, it receives information from Alice throughout a classical channel and from a quantum channel but he only sends information to Alice throughout a classical channel. Furthermore, Bob has an additional set of blocks for single photon detection which receives from processor block a real discrete time signal, in order to obtain the basis it should use to measure the photons. The photon receiver block also has an output signal built based on the clicks from both detectors, which is sent to Bob's main block.

Finally, Bob's processor has an output to Mutual Information top level block,  $MI_B$ .

### 3. Mutual Information calculation

Table 6.51: System Signals

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

Table 6.52: System Input Parameters

| Parameter                    | Default Value               | Description   |
|------------------------------|-----------------------------|---|
| rateOfPhotons                | 1000 photons/s              | Number of photon per sample.                          |
| iqAmplitudeValues            | {-45,0},{0,0},{45,0},{90,0} | Possible photon states.                               |
| numberOfSamplesPerSymbol     | 16                          | Number of samples per symbol.                         |
| detectorWindowTimeOpen       | 0.2 ms                      | smaller than 1 ms                                     |
| detectorPulseDelay           | 0.7 ms                      | in units of ms  |
| detectorProbabilityDarkCount | 0.0                         | Probability of dark counts in single-photon detector. |
| fiberLength                  | 10 km                       | Length of the optical fibre in km.                    |
| fiberAttenuation             | 0.2 dB/km                   | Attenuation of the optical fibre in dB/km.            |

Table 6.53: Header Files

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

Table 6.54: Source Files

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

### 6.15.2.1 Simulation Results

First, a simulation were performed using a deterministic model to change the state of polarization of a single-photon throughout an optical fibre. This state of polarization varies according with an angle  $\theta$  inserted by the user.

Figure 6.368 presents the graphical representation of two orthogonal states rotated by an angle  $\theta$ . This rotation is induced by the SOP modulator block which selects a deterministic  $\theta$  and  $\phi$  angles that do not change over the time. This same rotation is applied for all sequential samples. From figure 6.368 the theoretical QBER can be calculated using the following equation:

$$QBER = P(0)P(1|0) + P(1)P(0|1). \quad (6.225)$$

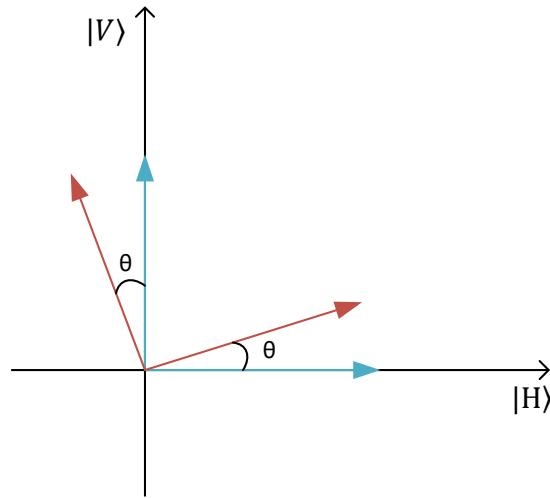


Figure 6.343: Representation of two orthogonal states rotated by an angle  $\theta$ .

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

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

This way,

$$QBER = \frac{1}{2}\sin^2(\theta) + \frac{1}{2}\sin^2(\theta) \quad (6.226)$$

$$QBER = \sin^2(\theta). \quad (6.227)$$

In figure 6.369 are represented two curves: QBER calculated from simulated data and QBER calculated using theoretical model from equation 6.242. Now, we can conclude that the QBER calculated from simulated data follows the theoretical curve. Nevertheless, the error bars presented in figure 6.369 were calculated based on a confidence interval of 95%.

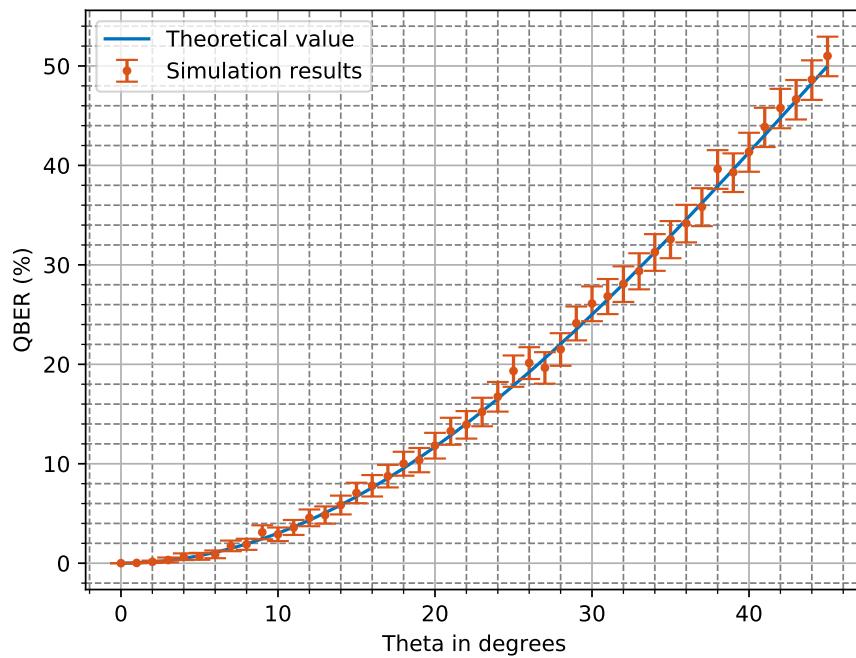


Figure 6.344: QBER evolution in relation with deterministic SOP drift.

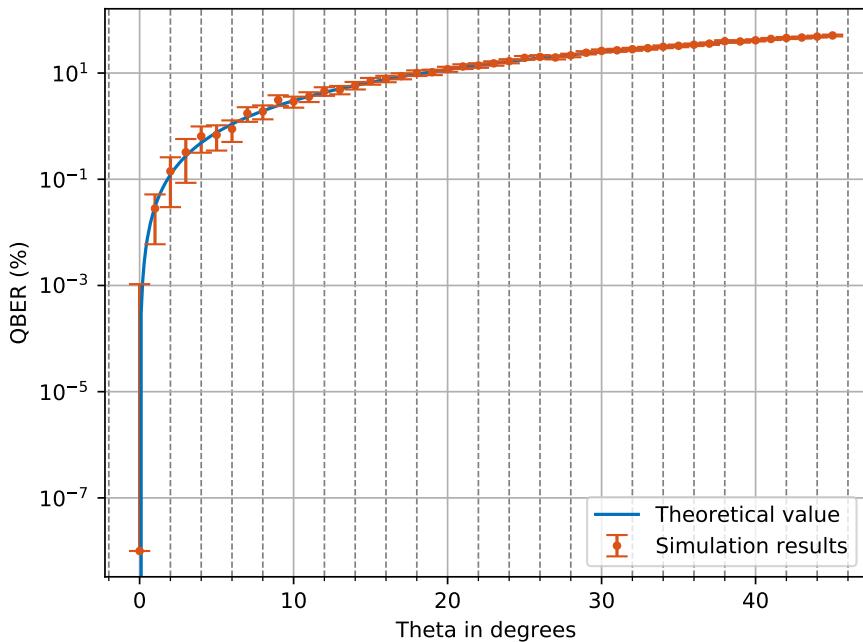


Figure 6.345: QBER evolution in relation with deterministic SOP drift in log scale.

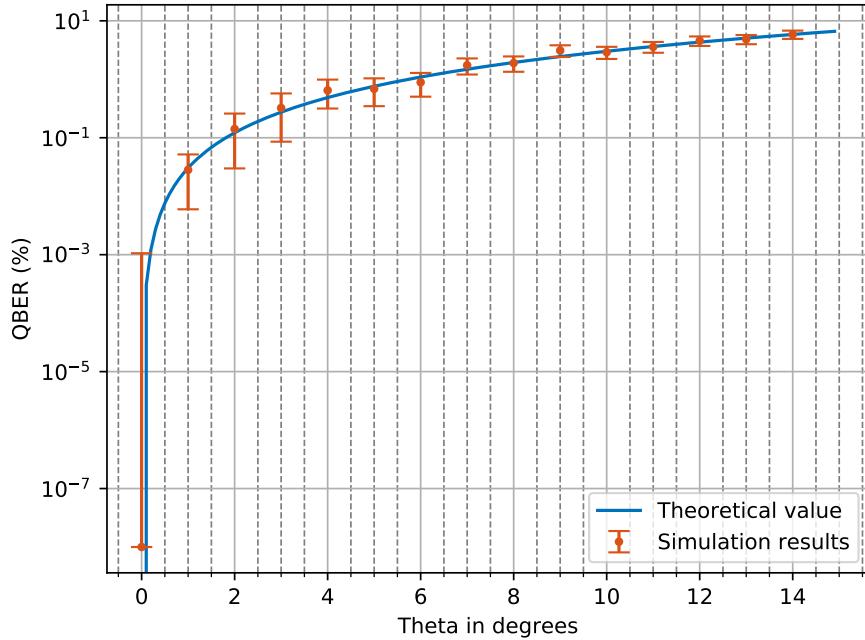


Figure 6.346: QBER evolution in relation with deterministic SOP drift scaled.

These results show the QBER evolution in the case when the single photons are linear polarized. However, due birefringence effects the polarization of the single photons may suffer changes and when they reach the detection system they may be circular or elliptical polarized. This way, another cases of interest are the QBER theoretical evolution for circular and elliptical polarization. Figure 6.372 shows the representation of circular polarization where the two amplitudes of both components  $E_{0x}$  and  $E_{0y}$  are equal to  $E_0$  although lagged by  $\phi = -\frac{\pi}{2} + 2m\pi$  with  $m = 0, \pm 1, \pm 2, \dots$ , i.e:

$$E_x(t) = E_0 \cos(\theta(t)) \quad (6.228)$$

and

$$E_y(t) = E_0 \sin(\theta(t)). \quad (6.229)$$

So, from equation 6.241 we have to calculate the conditional probability  $P(0|1)$ , since  $P(1|0) = 1 - P(0|1)$  and  $P(0) = P(1) = \frac{1}{2}$ :

$$\begin{aligned}
 P(0|1) &= \int_{-\infty}^{+\infty} \cos^2(\theta(t)) dt \\
 &= \int_0^{2\pi} \cos^2(\theta) d\theta \\
 &= \int_0^{2\pi} \frac{1 + \cos(2\theta)}{2} d\theta \\
 &= [\frac{\theta}{2} + \frac{1}{4} \sin(2\theta)]_0^{2\pi} \\
 &= \pi
 \end{aligned} \quad (6.230)$$

This way, from equation 6.241 QBER for circular polarized single photons should be:

$$\text{QBER} = \frac{1}{2}\pi + \frac{1}{2}(1 - \pi) = \frac{1}{2}. \quad (6.231)$$

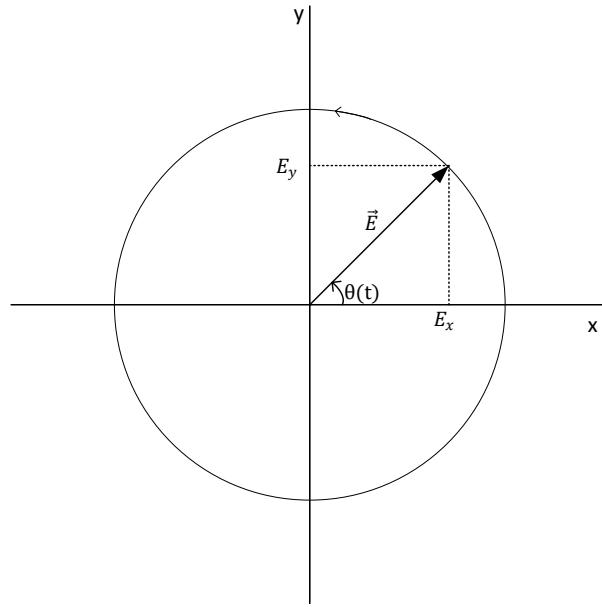


Figure 6.347: Representation of circular polarization.

The second case of interest is the elliptical polarization, which is the most general way to represent a state of polarization. From this representation the linear and circular polarization can be obtained being particular cases of elliptical polarization. At a fixed time instant the state of polarization can be represented as shown in figure 6.373. The ellipse has a minor axis with length  $\mathbf{b}$  and a major axis with length  $\mathbf{a}$ . The ellipticity angle  $\varphi$  and the azimuth angle  $\theta$  as well as the handedness (right and left defined by the sign of  $\varphi$ ) are the parameters necessary to describe the elliptical polarization.  $\theta$  is the angle of deviation of the ellipse regarding with the x-axis, and  $\varphi$  is defined by the following equation:

$$\tan\varphi = \frac{b}{a}.$$

From figure 6.373:

$$\begin{aligned} E_y &\sim \sqrt{1-a} \\ E_x &\sim \sqrt{a}, \end{aligned} \quad (6.232)$$

where  $a = \frac{1}{2}(1 + \cos(2\theta)\cos(2\varphi))$ . This way, the conditional probability  $P(0|1)$  can be calculated by:

$$\begin{aligned}
 P(0|1) &= \int_{-\infty}^{+\infty} E_x(t)^2 dt \\
 &= \int_{-\infty}^{+\infty} \frac{1}{2}(1 + \cos(2\theta) \cos(2\varphi)) dt \\
 &= \int_0^\pi \frac{1}{2}(1 + \cos(2\theta) \cos(2\varphi)) d\theta \\
 &= \frac{\theta}{2} + \frac{1}{4} \sin(2\theta) \cos(2\varphi) \Big|_{\theta=0}^{\theta=\pi} \\
 &= \frac{\pi}{2}
 \end{aligned} \tag{6.233}$$

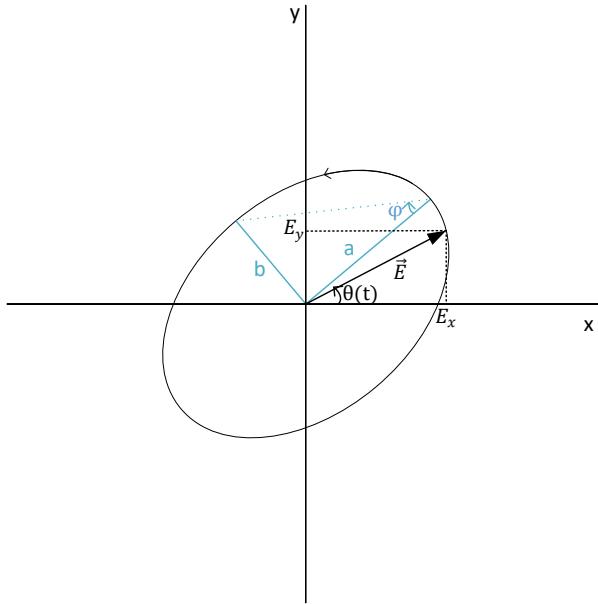


Figure 6.348: Representation of elliptical polarization.

This way, from equation 6.241 QBER for elliptical polarized single photons should be:

$$\text{QBER} = \frac{1}{2} \frac{\pi}{2} + \frac{1}{2} \left(1 - \frac{\pi}{2}\right) = \frac{1}{2}. \tag{6.234}$$

The elliptical polarization collapses for linear polarization in the cases that  $\varphi = 0$  and  $\theta = 0 + m\phi$ , where  $m = 0, 1, 2, \dots$ . Moreover, it collapses for circular polarization when  $\theta = \pm\frac{\pi}{2}$  and  $\varphi = \pm\frac{\pi}{4}$ .

A stochastic model to simulate the polarisation drift through the quantum channel was implemented based on the work presented in [3], which is explained in detail in section 7.65.

The  $\Delta p$  value was fixed at some values and the autocorrelation ( $A(t)$ ) over time was calculated of a SOP drift during the time interval  $\Delta T$ , where:

$$A(t) = \mathbb{E}[\mathbf{r}_k^T \mathbf{r}_{k+t}], \quad (6.235)$$

$\mathbf{r}_k = \mathbf{J}_k S_{in}$ , where  $S_{in}$  is a constant input photon state, which in this case we assume a photon polarized at 90 projected in Stokes space. The autocorrelation (ACF) is a correlation calculated between a signal and a delayed version of itself, i.e. autocorrelation measures the similarity between two signals. In this case the signal is represented using a matrix formulation.

The ACF calculated for different values of  $\Delta p$  are plotted in figure 6.374.

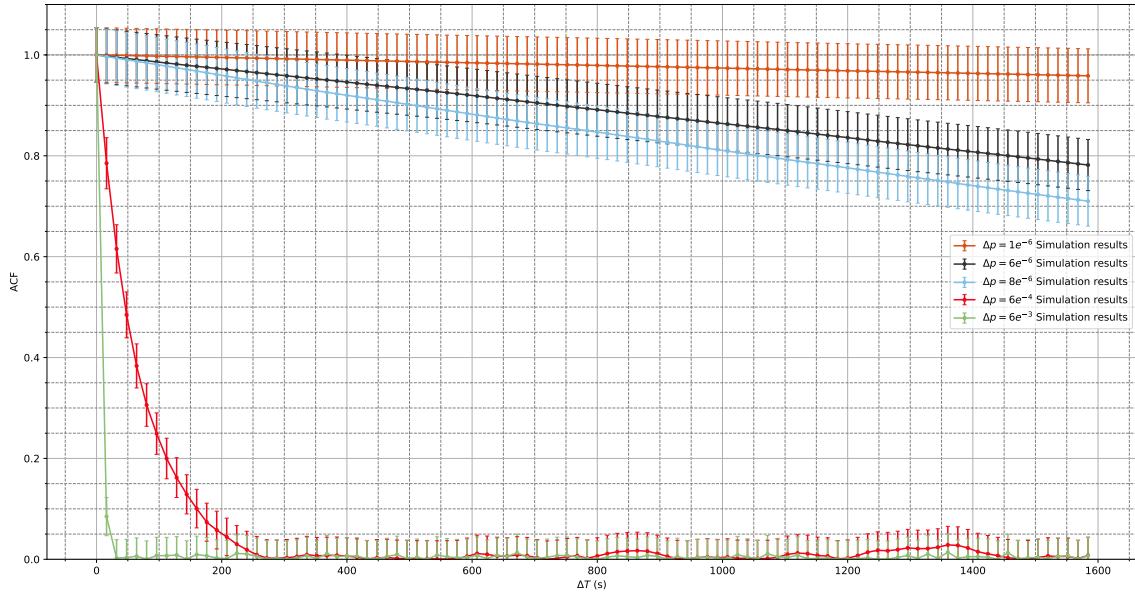


Figure 6.349: Autocorrelation values calculated for different values of  $\Delta p$  for  $\Delta t = 1600$  seconds.

As one can see in the figure when the  $\Delta p$  value increases the autocorrelation decreases faster than for low values of  $\Delta p$ , as which was expected by theoretical results. Moreover, the Stokes parameters for each different value of  $\Delta p$  were plotted in Poincare sphere, which shows the velocity of the change of the SOP when the  $\Delta p$  parameter increases. In addition, the theoretical evolution of ACF was calculated by [3]:

$$A(t) \approx \|S_{in}\|^2 e^{-8\pi\Delta T \Delta p}. \quad (6.236)$$

Next, we will present plots with a comparison between the numerical ACF calculated from data from simulation with theoretical values for each  $\Delta p$ , as well as, the respective plot in Poincare sphere and at the end the histograms of  $\vec{\alpha} = (\alpha_1, \alpha_2, \alpha_3)$  parameters for each  $\Delta p$ .

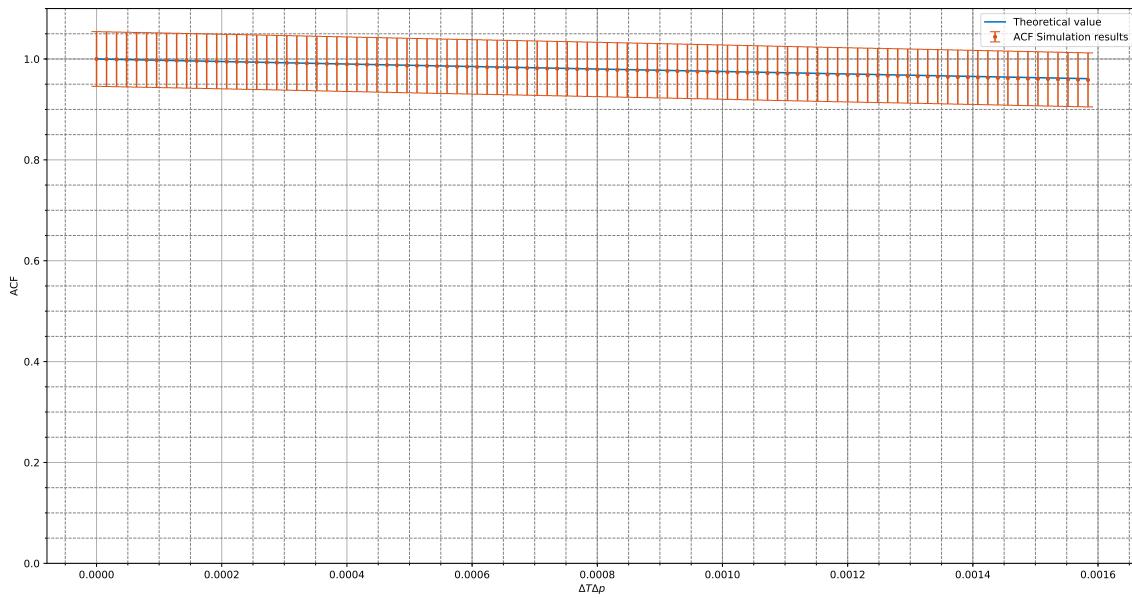


Figure 6.350: Comparison between ACF values calculated from theoretical formula and ACF values obtained from simulation data for  $\Delta p = 1e^{-6}$  during a period of 1600 seconds.

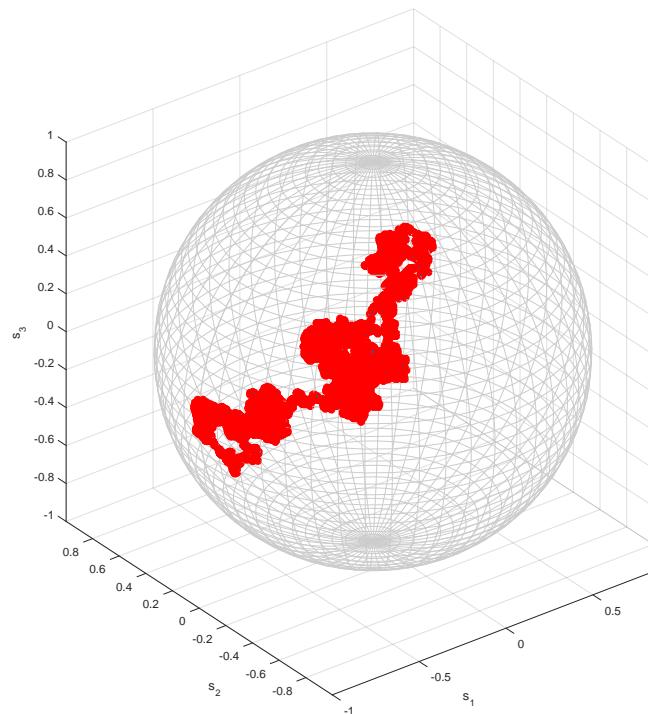


Figure 6.351: Representation of Stokes parameters in Poincare sphere for  $\Delta p = 1 \times 10^{-6}$  for a period of 3 hours.

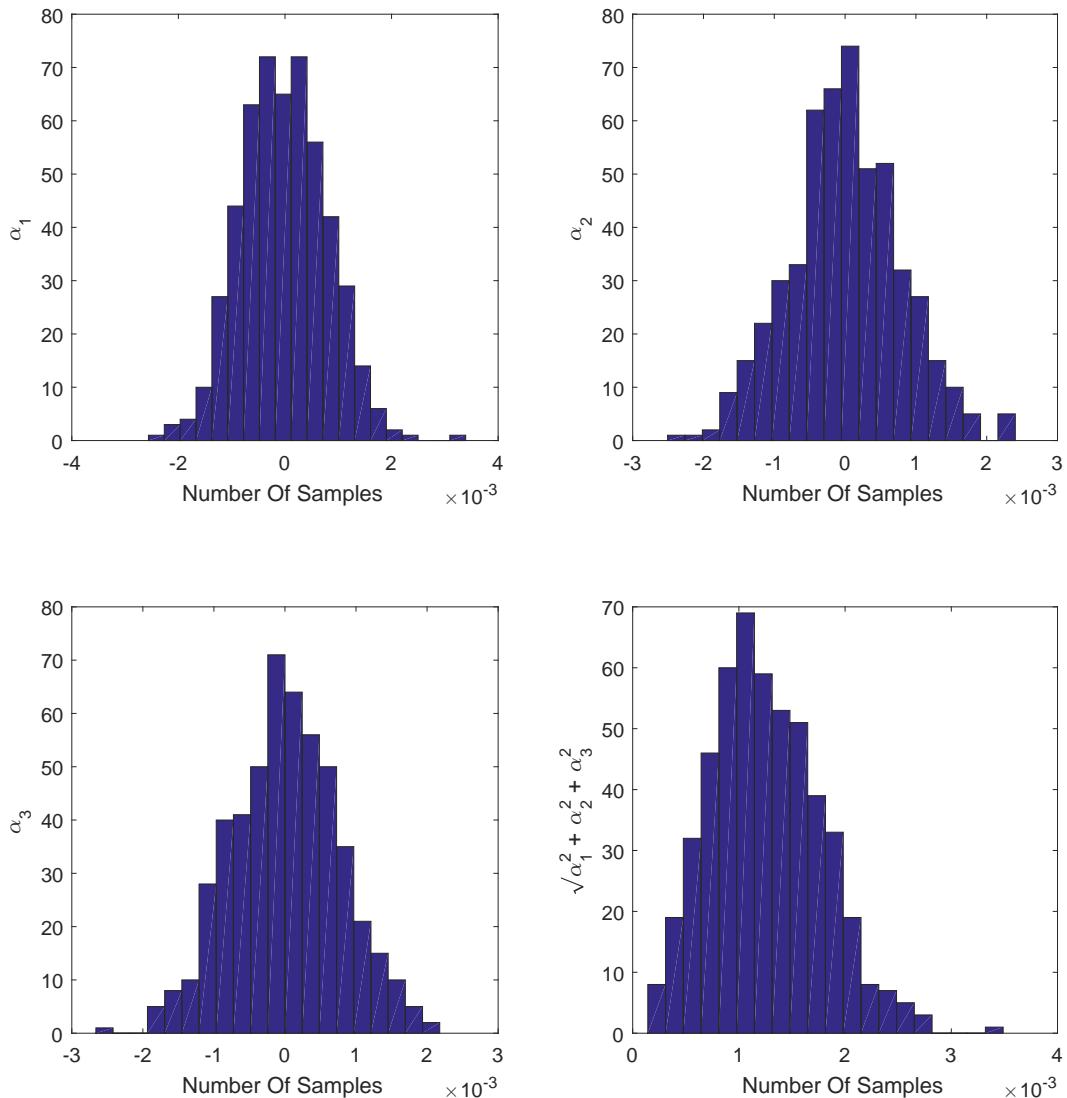


Figure 6.352: Histograms of  $(\alpha_1, \alpha_2, \alpha_3)$  parameters for  $\Delta p = 1 \times 10^{-6}$  using 100 samples.

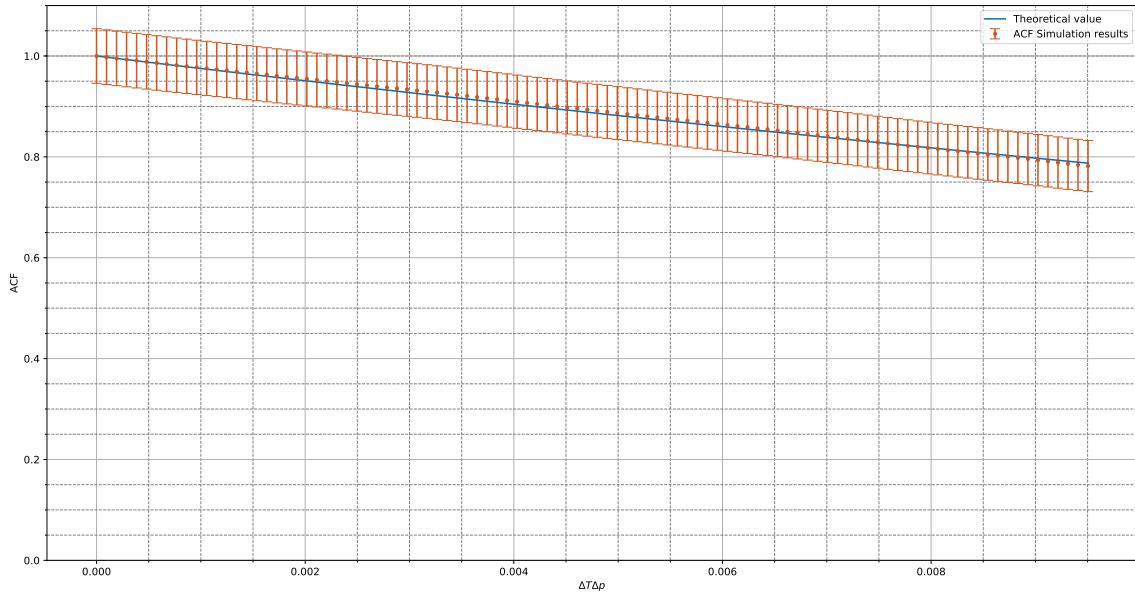


Figure 6.353: Comparison between ACF values calculated from theoretical formula and ACF values obtained from simulation data for  $\Delta p = 6e^{-6}$  during a period of 1600 seconds.

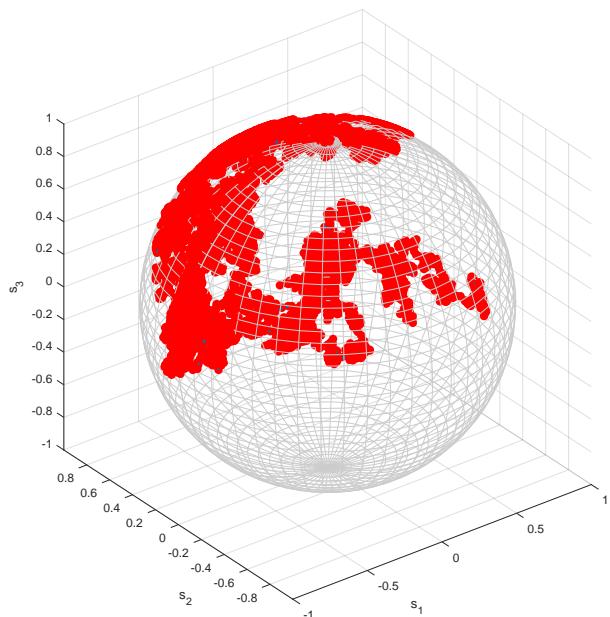


Figure 6.354: Representation of Stokes parameters in Poincare sphere for  $\Delta p = 6 \times 10^{-6}$  for a period of 3 hours.

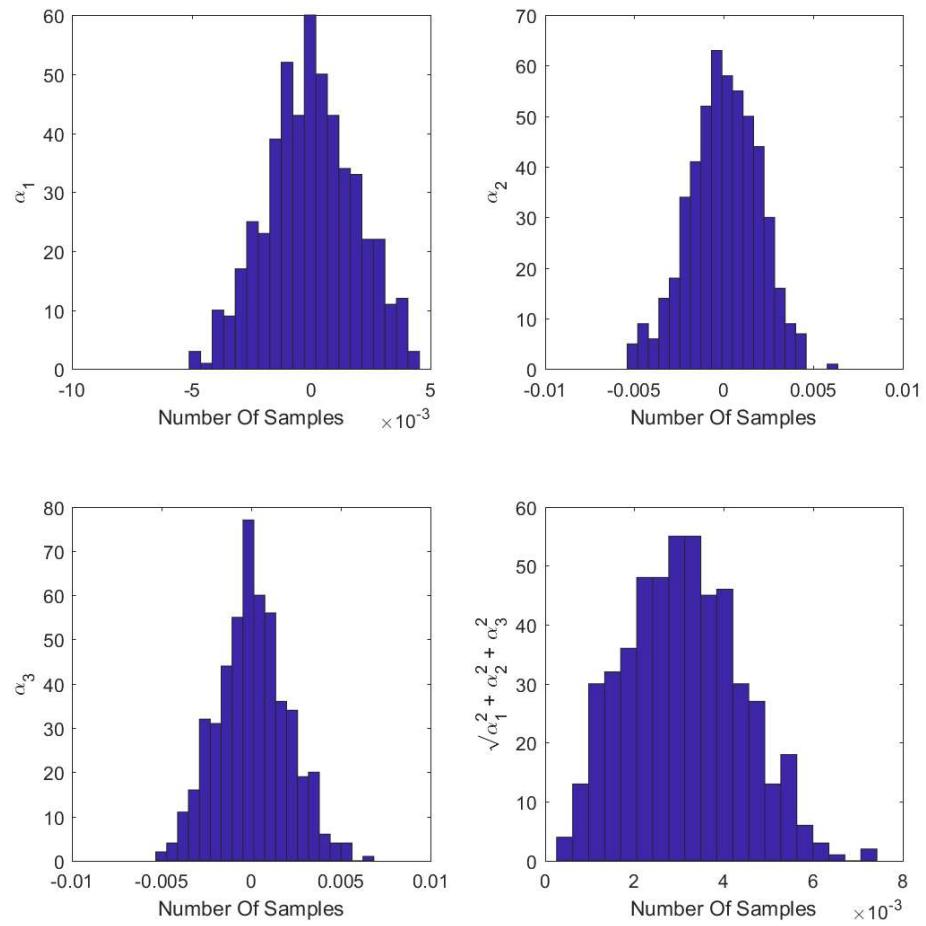


Figure 6.355: Histograms of  $(\alpha_1, \alpha_2, \alpha_3)$  parameters for  $\Delta p = 6 \times 10^{-6}$  using 100 samples.

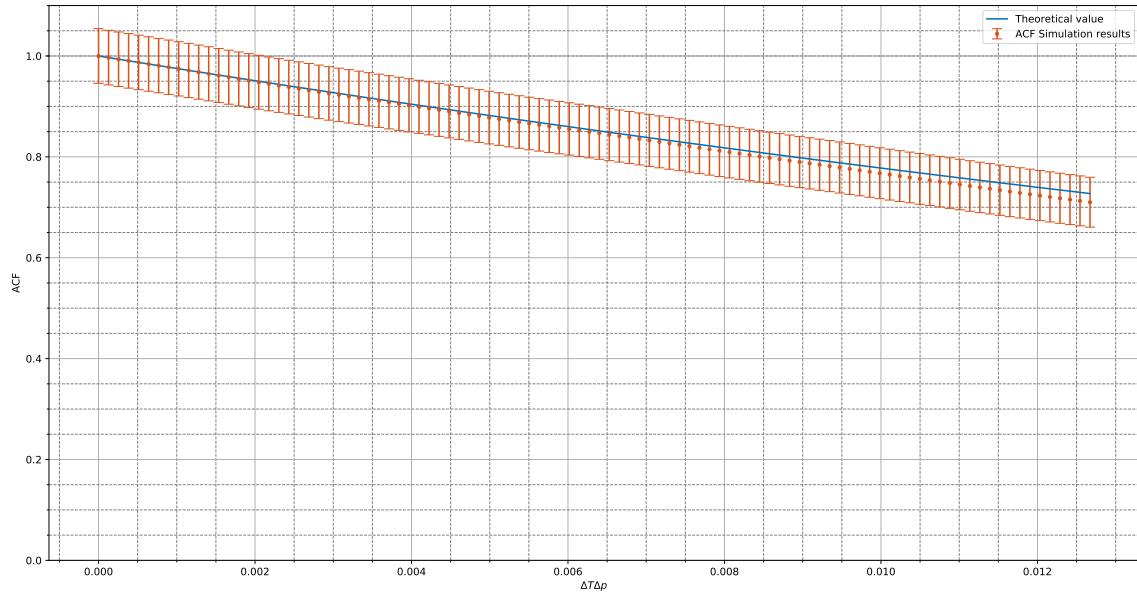


Figure 6.356: Comparison between ACF values calculated from theoretical formula and ACF values obtained from simulation data for  $\Delta p = 8e^{-6}$  during a period of 1600 seconds.

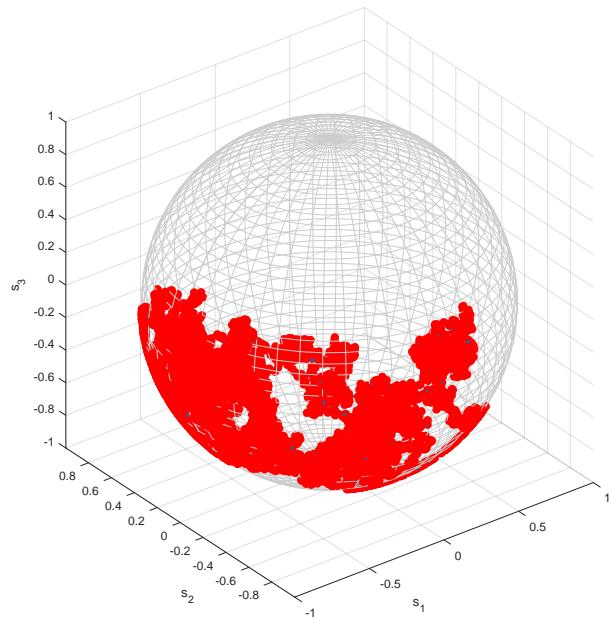


Figure 6.357: Representation of Stokes parameters in Poincare sphere for  $\Delta p = 8 \times 10^{-6}$  for a period of 3 hours.

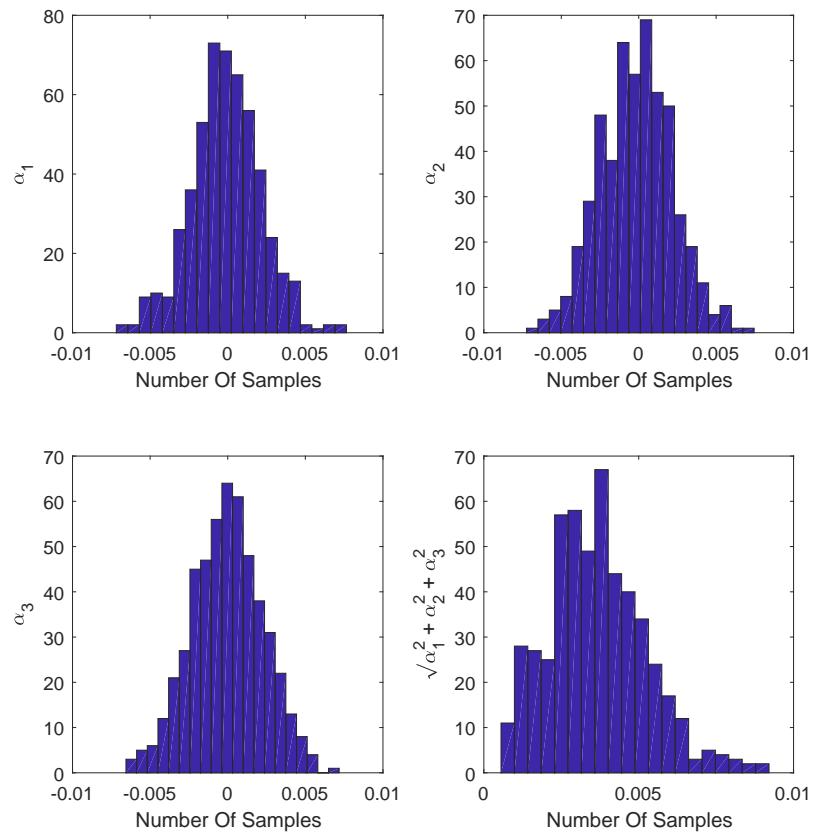


Figure 6.358: Histograms of  $(\alpha_1, \alpha_2, \alpha_3)$  parameters for  $\Delta p = 8 \times 10^{-6}$  using 100 samples.

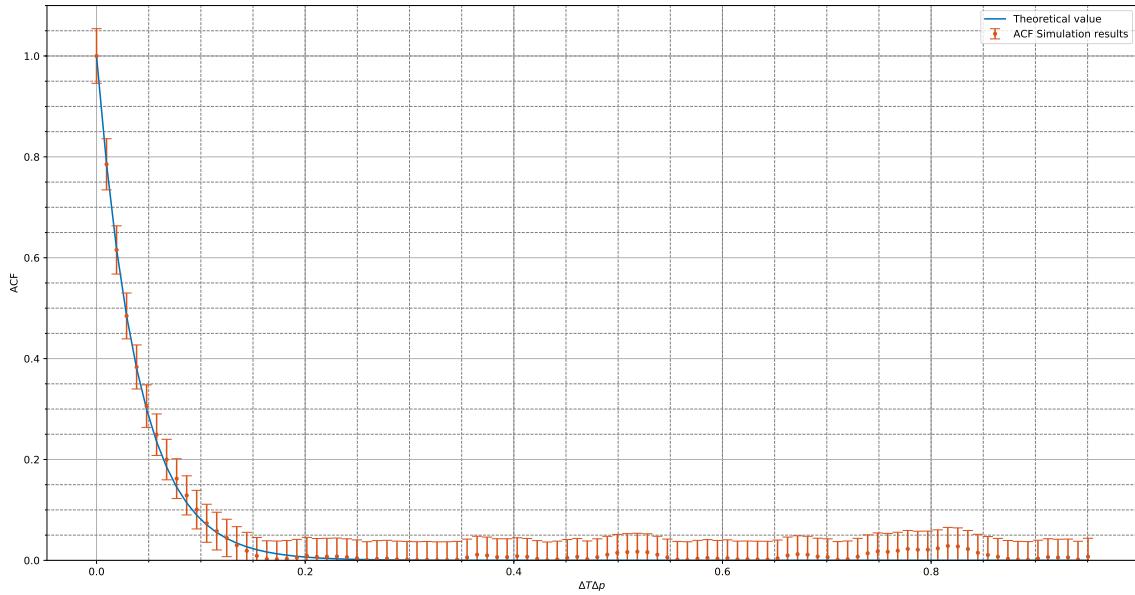


Figure 6.359: Comparison between ACF values calculated from theoretical formula and ACF values obtained from simulation data for  $\Delta p = 6e^{-4}$  during a period of 1600 seconds.

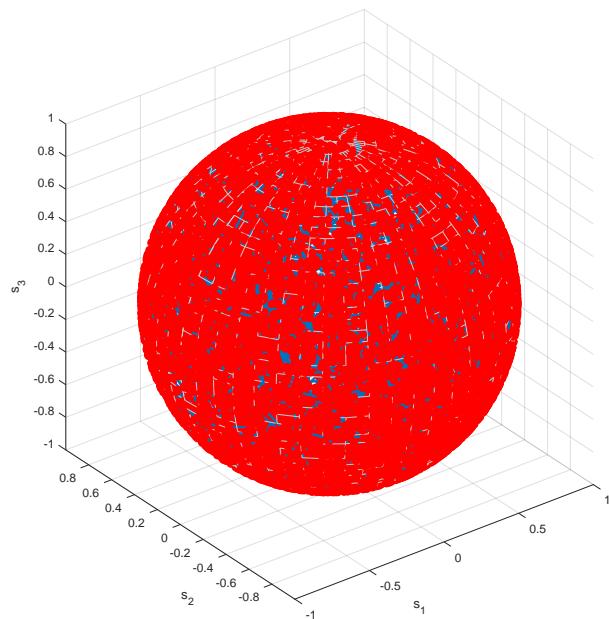


Figure 6.360: Representation of Stokes parameters in Poincare sphere for  $\Delta p = 6 \times 10^{-4}$  for a period of 3 hours.

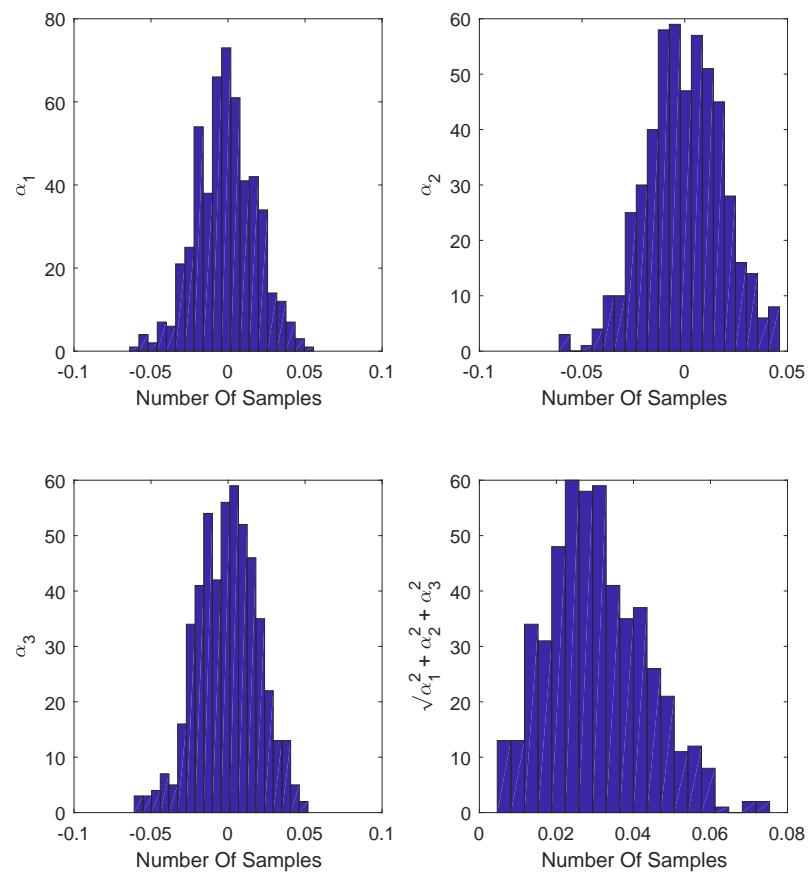


Figure 6.361: Histograms of  $(\alpha_1, \alpha_2, \alpha_3)$  parameters for  $\Delta p = 6 \times 10^{-4}$  using 100 samples.

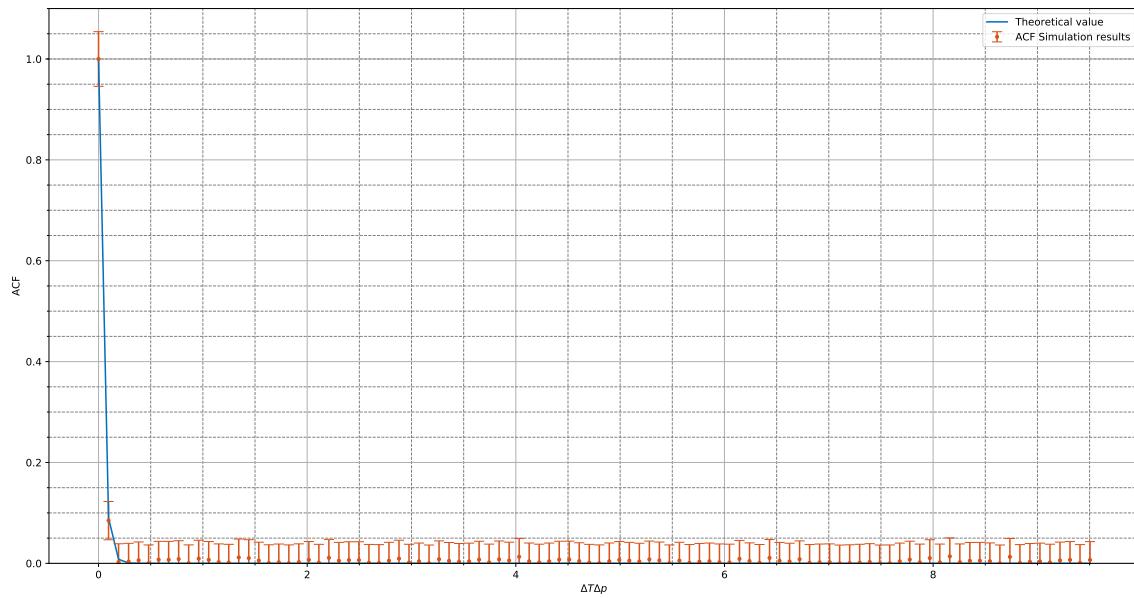


Figure 6.362: Comparison between ACF values calculated from theoretical formula and ACF values obtained from simulation data for  $\Delta p = 6e^{-3}$  during a period of 1600 seconds.

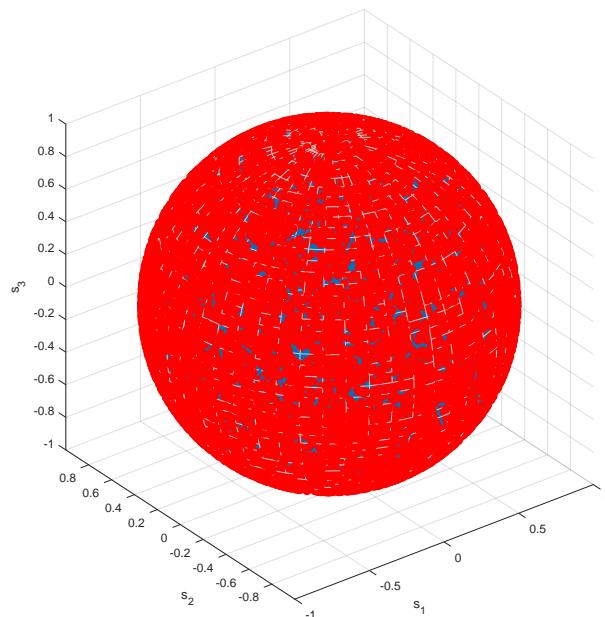


Figure 6.363: Representation of Stokes parameters in Poincare sphere for  $\Delta p = 6 \times 10^{-3}$  for a period of 3 hours.

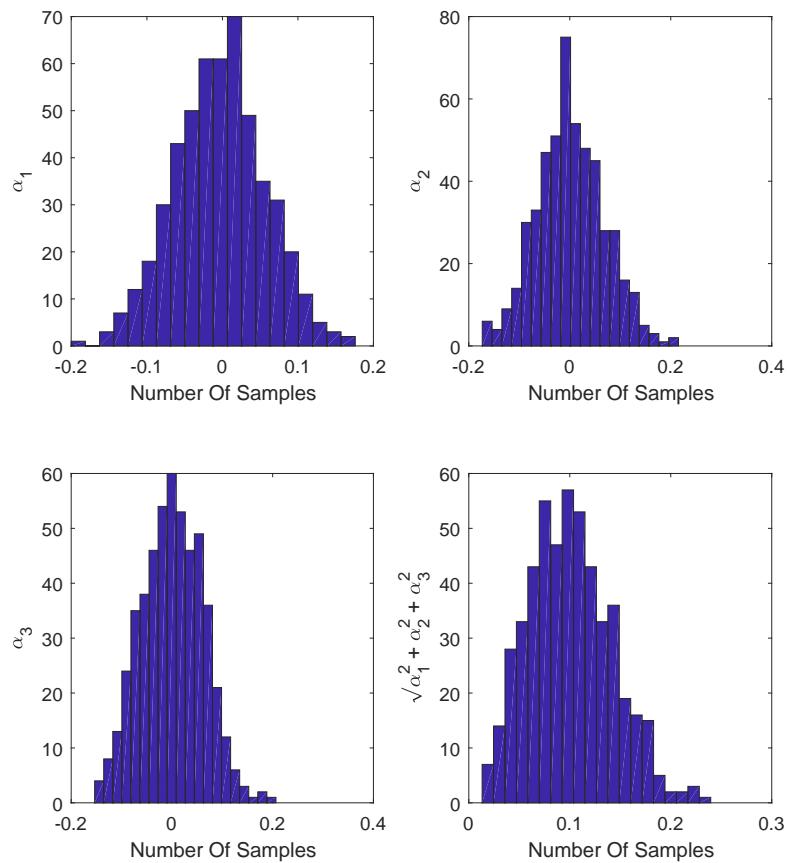


Figure 6.364: Histograms of  $(\alpha_1, \alpha_2, \alpha_3)$  parameters for  $\Delta p = 6 \times 10^{-3}$  using 100 samples.

Figure 6.365 shows the evolution of QBER over time for three different values of  $\Delta p$ .

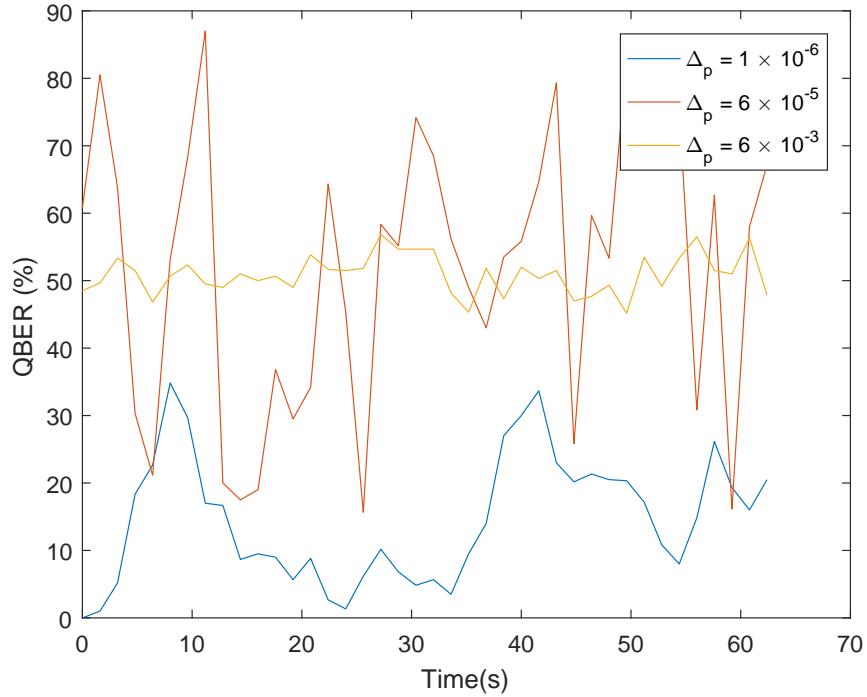


Figure 6.365: QBER evolution over time for three different values of  $\Delta p$ .

### Experimental $\Delta p$ measurement

The parameter  $\Delta p$  may be related to PMD of the optical fiber. The autocorrelation in time may be written as:

$$A(t) = e^{-\frac{|\Delta t|}{t_d}}, \quad (6.237)$$

where  $t_d$  is the typical drift time for absolute polarization states. This coefficient is unique for each installed fiber and can be related with the polarization linewidth parameter  $\Delta p$ , which also depends of the installation of the optical fiber and it is also unique for each fiber. Comparing equation 6.256 with equation ?? we may find the relation:

$$\frac{1}{t_d} \sim \Delta p. \quad (6.238)$$

From the work presented in [4] and [5] we can write the expression for  $t_d$  as:

$$t_d = \frac{2t_0}{3w^2 D_p^2 z}, \quad (6.239)$$

where  $D_p$  is the PMD coefficient (in  $ps/\sqrt{km}$ ) of the fiber,  $w$  is the carrier frequency,  $t_0$  is a measure of the drift time of the index difference in birefringence element used to model the fiber and  $z$  is the length of the fiber. This way, we can write the  $\Delta p$  as:

$$\Delta p \sim \frac{3w^2 D_p^2 z}{2t_0}. \quad (6.240)$$

### 6.15.3 Experimental Setup

In figure 6.366 are presented the experimental setup to be performed in the lab. The main goal is to build an experimental setup in which Alice and Bob communicate through two classical channels and one quantum channel that will have only one direction (Alice to Bob).

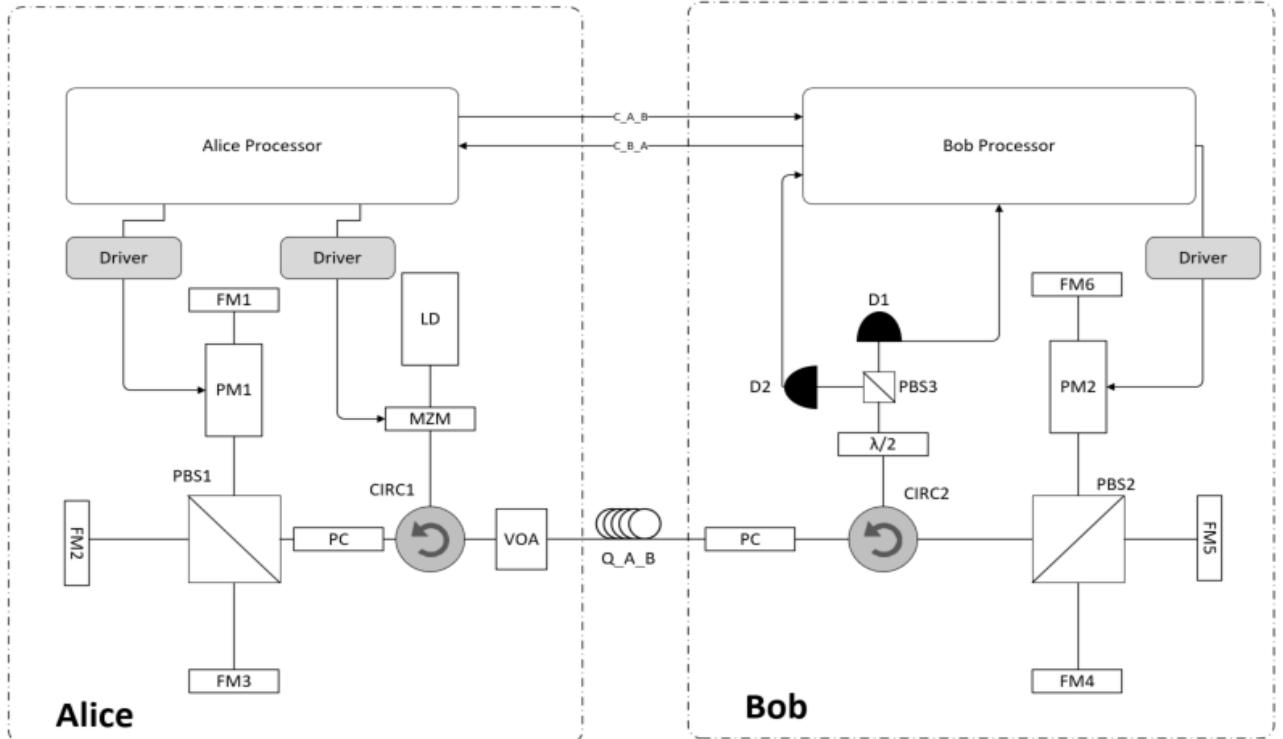


Figure 6.366: QOKD Experimental setup

The laser **LD** emits light with a wavelength of 1550 nm and then the light passes throughout a Mach-Zenhder Modulator (**MZM**) in order to have pulsed light. The light is polarized with a polarizer controller and gets a linear polarization of  $45^\circ$ . When the light reaches the polarization beam splitter, **PBS1**, single photon pulses in  $45^\circ$  linearly polarization state

$$|in\rangle = |45\rangle = \frac{\sqrt{2}}{2}(|H\rangle + |V\rangle)$$

and each pulse is divided in PBS1 into two orthogonal components  $P_x$  and  $P_y$ .  $P_x$  is directly transmitted in **FM2** direction and it is reflected by it with its stated rotated, which means it has a new direction  $|V\rangle$ . This way, when it reaches again **PBS1** is reflected to **FM3** direction and it happens the same being the  $P_x$  component transmitted through the **PBS1** in **FM1** direction passing through the phase modulator and suffers a certain phase shift. Relatively to vertical component which reaches the **PBS1** and was reflected to **FM1** direction and passes through the phase modulator, **PM**, that applies a phase shift to it, than it follows to the **FM1** and is reflected rotated by  $90^\circ$ , meaning that it follows to **FM3** and it is reflected at its original state. At the end, the two components recombine in **PSB1** and the single photon

pulse follows its path with a polarization state defined by the phase shift applied in **PM**.

In table 6.55 is present the phases shifts that must be applied at Phase Modulator in order to get the polarized photons chose by Alice.

Table 6.55: Different Alice's output polarization states based on shift phases applied in Phase Modulator.

| Alice: $(\phi_1, \phi_2)$         | Output Polarization |
|-----------------------------------|---------------------|
| $(\frac{\pi}{2}, 0)$              | Vertical            |
| $(\frac{\pi}{2}, \frac{3\pi}{2})$ | Horizontal          |
| $(0, 0)$                          | $-45^\circ$         |
| $(0, \frac{\pi}{2})$              | $45^\circ$          |

In table 6.56 are described all components needed to build the experimental setup.

Table 6.56: List of material

| Material Name                         | Quantity | Status    |
|---------------------------------------|----------|-----------|
| Laser semiconductor 1550nm            | 1        | ✓         |
| Manual polarization controller        | 5        |           |
| Faraday Mirror (FM)                   | 6        |           |
| Mach-Zehnder Modulator                | 1        | ✓ 2.56GHz |
| Single Photon Detector                | 2        | ✓         |
| Phase modulator                       | 2        |           |
| Four-port polarization beam splitter  | 2        |           |
| Three-port polarization beam splitter | 1        |           |
| Half-wave plate                       | 1        |           |
| Optical circulator                    | 2        |           |
| Variable Optical Attenuator           | 1        | ✓         |
| Computer                              | 1        |           |

#### 6.15.4 Open Issues

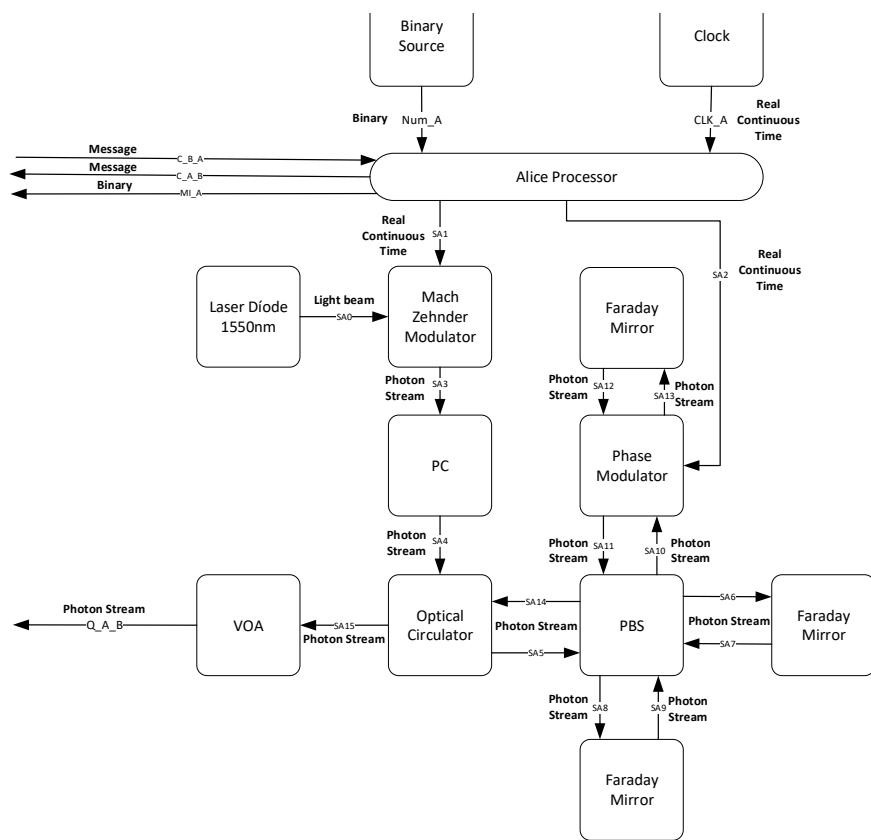


Figure 6.367: Simulation diagram - Alice's side - Similar with experimental setup.

## References

- [1] Tie-Ming Liu et al. "Researching on Cryptographic Algorithm Recognition Based on Static Characteristic-Code". In: *Security Technology: International Conference, SecTech 2009, Held as Part of the Future Generation Information Technology Conference, FGIT 2009, Jeju Island, Korea, December 10-12, 2009. Proceedings*. Ed. by Dominik Ślęzak et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 140–147. ISBN: 978-3-642-10847-1. DOI: [10.1007/978-3-642-10847-1\\_18](https://doi.org/10.1007/978-3-642-10847-1_18). URL: [https://doi.org/10.1007/978-3-642-10847-1\\_18](https://doi.org/10.1007/978-3-642-10847-1_18).
- [2] Gilles Brassard and Louis Salvail. "Secret-Key Reconciliation by Public Discussion". In: *Advances in Cryptology — EUROCRYPT '93: Workshop on the Theory and Application of Cryptographic Techniques Lofthus, Norway, May 23–27, 1993 Proceedings*. Ed. by Tor Helleseth. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 410–423. ISBN: 978-3-540-48285-7. DOI: [10.1007/3-540-48285-7\\_35](https://doi.org/10.1007/3-540-48285-7_35). URL: [https://doi.org/10.1007/3-540-48285-7\\_35](https://doi.org/10.1007/3-540-48285-7_35).
- [3] Cristian B Czegledi et al. "Polarization drift channel model for coherent fibre-optic systems". In: *Scientific reports* 6 (2016), p. 21217.
- [4] Magnus Karlsson, Jonas Brentel, and Peter A Andrekson. "Long-term measurement of PMD and polarization drift in installed fibers". In: *JOURNAL OF lightwave technology* 18.7 (2000), p. 941.
- [5] Álvaro J Almeida et al. "Continuous Control of Random Polarization Rotations for Quantum Communications". In: *Journal of Lightwave Technology* 34.16 (2016), pp. 3914–3922.

## 6.16 Discrete Variables Polarization Encoding

|                      |   |   |
|----------------------|---|---|
| <b>Student Name</b>  | : | Mariana Ramos   |
| <b>Starting Date</b> | : | June 20, 2018   |
| <b>Goal</b>          | : | Study of polarization drift throughout optical fiber with discrete variables. |
| <b>Directory</b>     | : | sdf/dv_polarization_encoding_system.  |

### 6.16.1 Theoretical Analysis

#### 6.16.1.1 Deterministic Rotation around S3

First, a simulation were performed using a deterministic model to change the state of polarization of a single-photon throughout an optical fibre. This state of polarization varies according with an angle  $\theta$  inserted by the user.

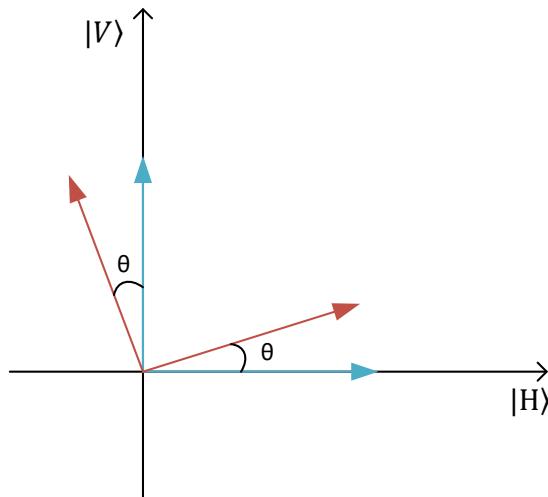


Figure 6.368: Representation of two orthogonal states rotated by an angle  $\theta$ , where  $|V\rangle$  corresponds to bit "1" and  $|H\rangle$  corresponds to bit "0".

Figure 6.368 presents the graphical representation of two orthogonal states rotated by an angle  $\theta$ . This rotation is induced by the SOP modulator block which selects a deterministic  $\theta$  and  $\phi$  angles that do not change over the time. This same rotation is applied for all sequential samples. From figure 6.368 the theoretical QBER can be calculated using the following equation:

$$QBER = P(0)P(1|0) + P(1)P(0|1). \quad (6.241)$$

Assuming,

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

This way,

$$QBER = \frac{1}{2}\sin^2(\theta) + \frac{1}{2}\sin^2(\theta) \quad (6.242)$$

$$QBER = \sin^2(\theta). \quad (6.243)$$

In figure 6.369 are represented two curves: QBER calculated from simulated data and QBER calculated using theoretical model from equation 6.242. Now, we can conclude that the QBER calculated from simulated data follows the theoretical curve. Nevertheless, the error bars presented in figure 6.369 were calculated based on a confidence interval of 95%.

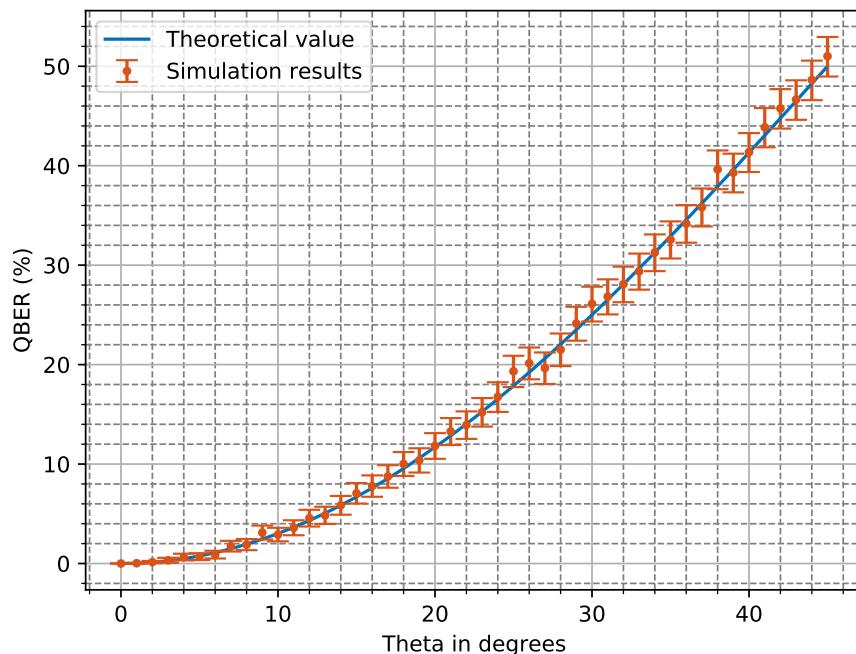


Figure 6.369: QBER evolution in relation with deterministic SOP drift.

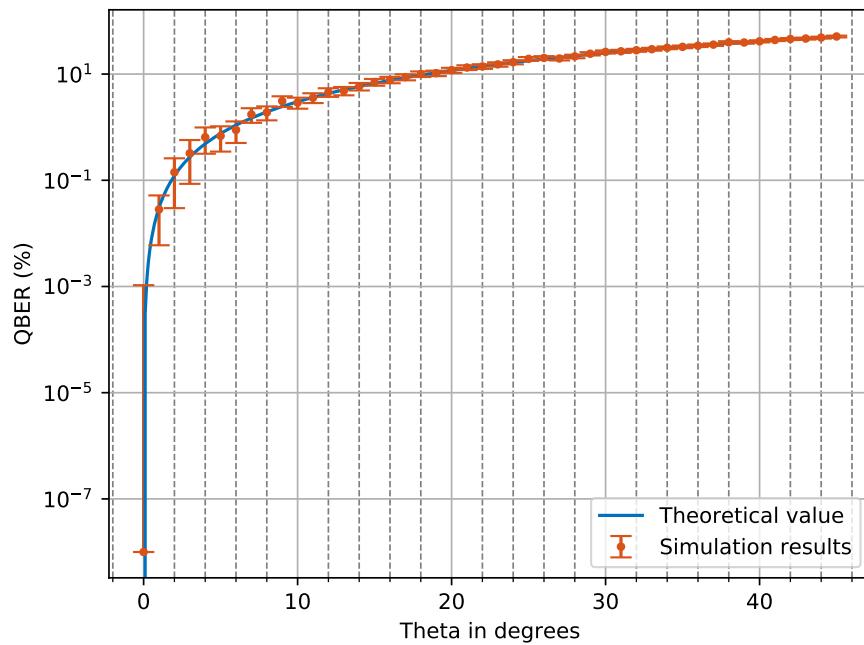


Figure 6.370: QBER evolution in relation with deterministic SOP drift in log scale.

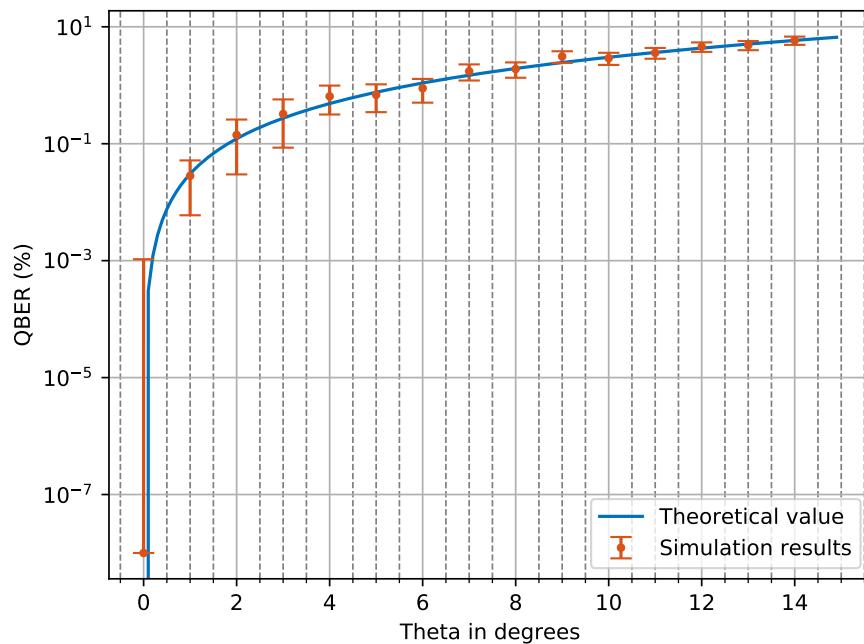


Figure 6.371: QBER evolution in relation with deterministic SOP drift scaled.

### 6.16.1.2 Deterministic Rotation around S1 and S2

The above results show the QBER evolution in the case when the single photons are linear polarized. However, due birefringence effects the polarization of the single photons may suffer changes and when they reach the detection system they may be circular or elliptical polarized. This way, another cases of interest are the QBER theoretical evolution for circular and elliptical polarization. When electromagnetic wave travels in the  $\hat{z}$  direction, it oscillates perpendicular to the propagation direction. The polarization of the wave is managed by the electric field evolution in the plane XY, which means that each SOP can be represented in a linear basis,

$$\mathbf{E}(x, y, t) = E_x \hat{x} \cos(wt) + E_y \hat{y} \cos(wt + \varphi), \quad (6.244)$$

where  $\varphi = \varphi_y - \varphi_x$ . One interest representation of the electric field is the Jones vector:

$$\mathbf{E} = E_0 \begin{pmatrix} \cos \theta \\ \sin \theta e^{j\varphi} \end{pmatrix}, \quad (6.245)$$

where  $E_0 = \sqrt{E_x^2 + E_y^2}$  is the field amplitude in the Cartesian coordinate system. With this factor of normalization the state of polarization can be described uniquely by  $(\theta, \varphi)$  angles as shown in figure 6.373. Moreover, the evolution of the ellipse depends on the signal of  $\varphi$ .

Figure 6.372 shows the representation of circular polarization where the two amplitudes of both components  $E_x$  and  $E_y$  are equal to  $E_0$  although lagged by  $\phi = -\frac{\pi}{2} + 2m\pi$  with  $m = 0, \pm 1, \pm 2, \dots$ , i.e:

$$E_x(t) = E_0 \cos(\theta(t)) \quad (6.246)$$

and

$$E_y(t) = E_0 \sin(\theta(t)). \quad (6.247)$$

So, from equation 6.241 we have to calculate the conditional probability  $P(0|1)$ , since  $P(1|0) = 1 - P(0|1)$  and  $P(0) = P(1) = \frac{1}{2}$ :

$$\begin{aligned} P(0|1) &= \int_{-\infty}^{+\infty} \cos^2(\theta(t)) dt \\ &= \int_0^{2\pi} \cos^2(\theta) d\theta \\ &= \int_0^{2\pi} \frac{1 + \cos(2\theta)}{2} d\theta \\ &= [\frac{\theta}{2} + \frac{1}{4} \sin(2\theta)]_0^{2\pi} \\ &= \pi \end{aligned} \quad (6.248)$$

This way, from equation 6.241 QBER for circular polarized single photons should be:

$$\text{QBER} = \frac{1}{2}\pi + \frac{1}{2}(1 - \pi) = \frac{1}{2}. \quad (6.249)$$

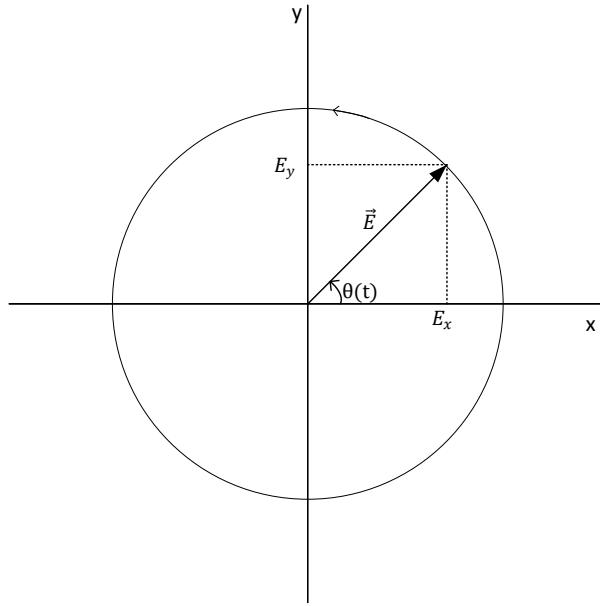


Figure 6.372: Representation of circular polarization.

#### 6.16.1.3 Deterministic Rotation around S1, S2 and S3

With the elliptical polarization description, we can calculate the theoretical QBER when a specific state is transmitted. Lets assume,

$$\text{QBER} = P(0)P(1|0) + P(1)P(0|1) \quad (6.250)$$

be the general expression to calculate QBER. We also assume that our link is symmetric, which means that we only can transmit one of the two possible symbols, with the same probability.

$$P(0) = P(1) = \frac{1}{2} \quad (6.251)$$

and the error probability is the same

$$P(0|1) = P(1|0). \quad (6.252)$$

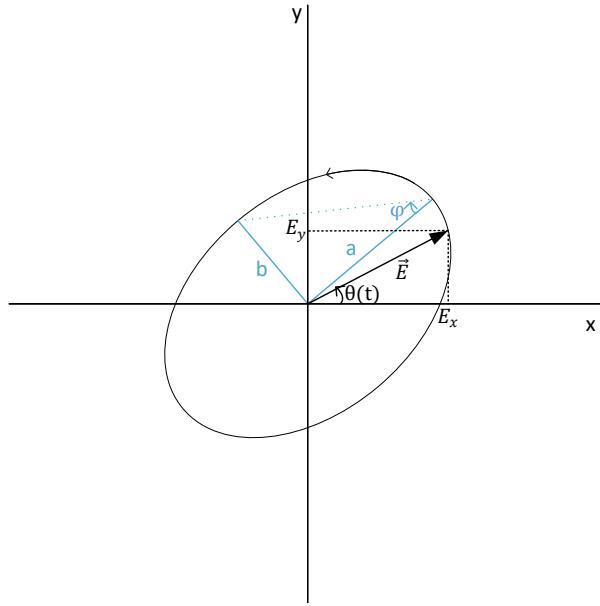


Figure 6.373: Representation of elliptical polarization.

At a fixed time instant the state of polarization can be represented as shown in figure 6.373. The ellipse has a minor axis with length  $b$  and a major axis with length  $a$ . The ellipticity angle  $\varphi$  and the azimuth angle  $\theta$  as well as the handedness (right and left defined by the sign of  $\varphi$ ) are the parameters necessary to describe the elliptical polarization.  $\theta$  is the angle of deviation of the ellipse regarding with the x-axis, and  $\varphi$  is defined by the following equation:

$$\tan \varphi = \frac{b}{a}.$$

From figure 6.373:

$$\begin{aligned} E_y &\sim \sqrt{1-a} \\ E_x &\sim \sqrt{a}, \end{aligned} \tag{6.253}$$

where  $a = \frac{1}{2}(1 + \cos(2\theta) \cos(2\varphi))$ . For an elliptical polarization theoretical QBER will be calculated using equation 6.250 by the following formula,

$$\text{QBER} = 1 - \frac{1}{2}(1 + \cos \theta \cos \varphi). \tag{6.254}$$

Elliptical polarization is the most general case and from that we can obtain the representation of the circular and linear polarization states. Elliptical polarization have no restrictions about angles  $(\theta, \varphi)$ . On the contrary, circular polarization is the most restrictive case, which requires  $\theta = \pm \frac{\pi}{4}$  and  $\varphi = \pm \frac{\pi}{2}$ . In addition, the case of linear polarization is less restrictive than the circular polarization, letting  $\theta$  takes any value and  $\varphi = m\pi$ , where  $m$

is an integer. As result, the restriction values of  $\theta$  and  $\varphi$  in equation 6.254 we obtain for circular polarization QBER=  $\frac{1}{2}$ , which means the detection is completely random, and for linear polarization QBER=  $\sin \theta^2$ , which depends only the rotation angle.

#### 6.16.1.4 Stochastic Rotations

A stochastic model to simulate the polarisation drift through the quantum channel was implemented based on the work presented in [1], which is explained in detail in section 7.65.

The  $\Delta p$  value was fixed at some values and the autocorrelation ( $A(t)$ ) over time was calculated of a SOP drift during the time interval  $\Delta T$ , where:

$$A(t) = \mathbb{E}[\mathbf{r}_k^T \mathbf{r}_{k+t}], \quad (6.255)$$

$\mathbf{r}_k = \mathbf{J}_k S_{in}$ , where  $S_{in}$  is a constant input photon state, which in this case we assume a photon polarized at 90 projected in Stokes space. The autocorrelation (ACF) is a correlation calculated between a signal and a delayed version of itself, i.e. autocorrelation measures the similarity between two signals. In this case the signal is represented using a matrix formulation.

The ACF calculated for different values of  $\Delta p$  are plotted in figure 6.374.

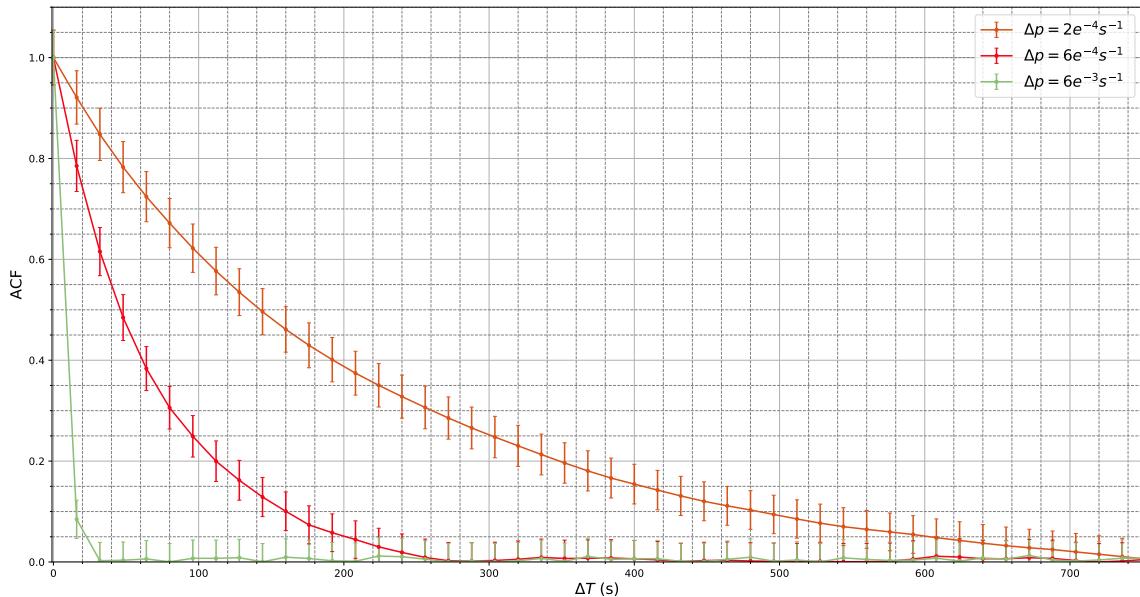


Figure 6.374: Autocorrelation values calculated for different values of  $\Delta p$  for  $\Delta t = 1600$  seconds.

As one can see in the figure when the  $\Delta p$  value increases the autocorrelation decreases faster than for low values of  $\Delta p$ , as which was expected by theoretical results. Moreover, the Stokes parameters for each different value of  $\Delta p$  were plotted in Poincare sphere, which

shows the velocity of the change of the SOP when the  $\Delta p$  parameter increases. In addition, the theoretical evolution of ACF was calculated by [1]:

$$A(t) \approx \|S_{in}\|^2 e^{-8\pi\Delta T \Delta p}. \quad (6.256)$$

Next, we will present plots with a comparison between the numerical ACF calculated from data from simulation with theoretical values for each  $\Delta p$ .

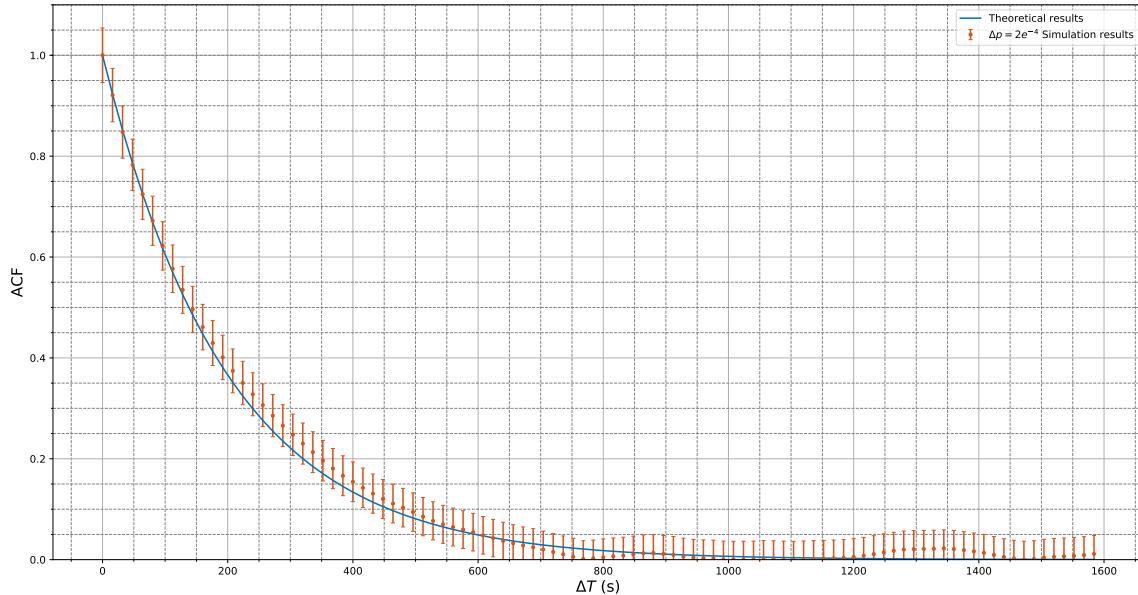


Figure 6.375: Comparison between ACF values calculated from theoretical formula and ACF values obtained from simulation data for  $\Delta p = 2e^{-4}$  during a period of 1600 seconds.

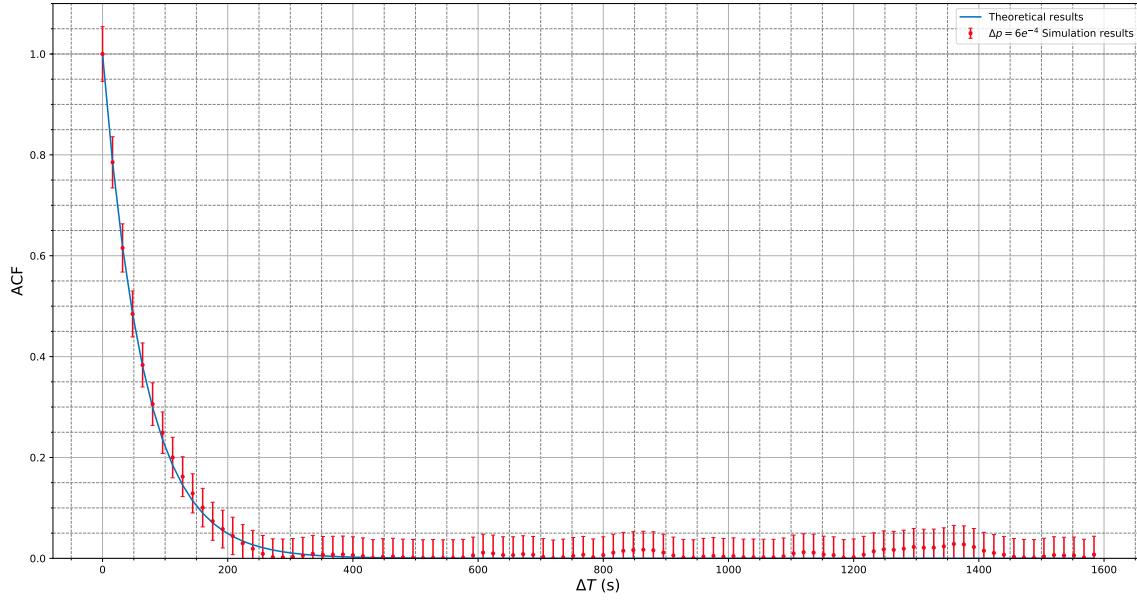


Figure 6.376: Comparison between ACF values calculated from theoretical formula and ACF values obtained from simulation data for  $\Delta p = 6e^{-4}$  during a period of 1600 seconds.

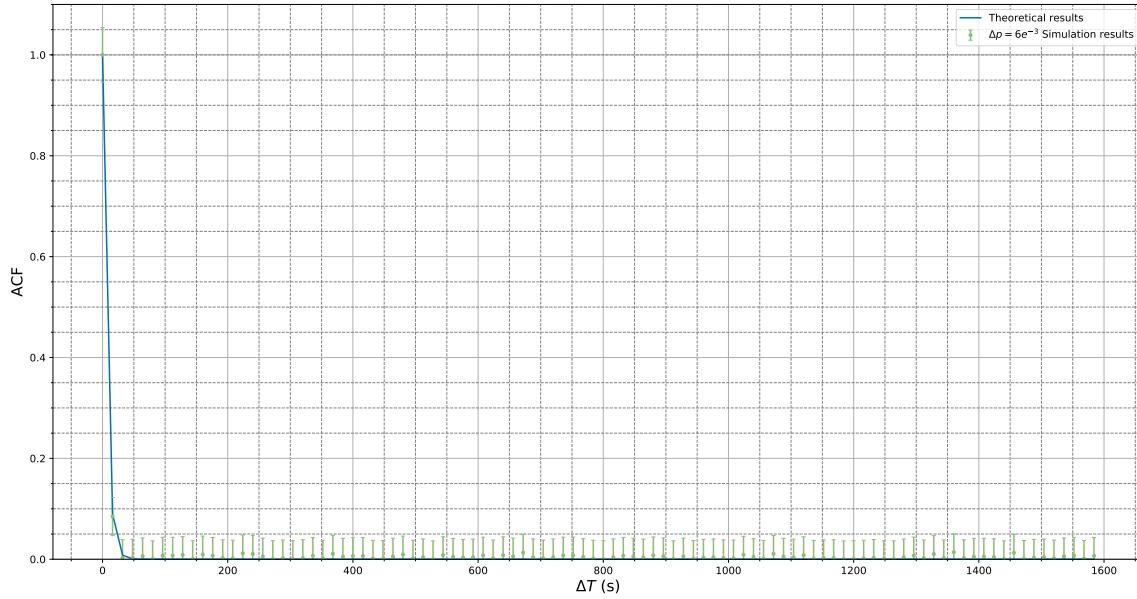


Figure 6.377: Comparison between ACF values calculated from theoretical formula and ACF values obtained from simulation data for  $\Delta p = 6e^{-3}$  during a period of 1600 seconds.

Next, we will fix the  $\Delta p = 2e^{-4}$  and we will represent the evolution over time of the state of polarization in Poincare sphere, as it is shown in figure 6.378. As time goes by, the state of polarization evolves due birefringence effects of the optical fibre covering the all area of the

Poincaré sphere when the time interval is large enough. When it happens, the probability of error in the detector system is 50% since the state of polarization can be found in any point of the sphere having a behaviour completely random.

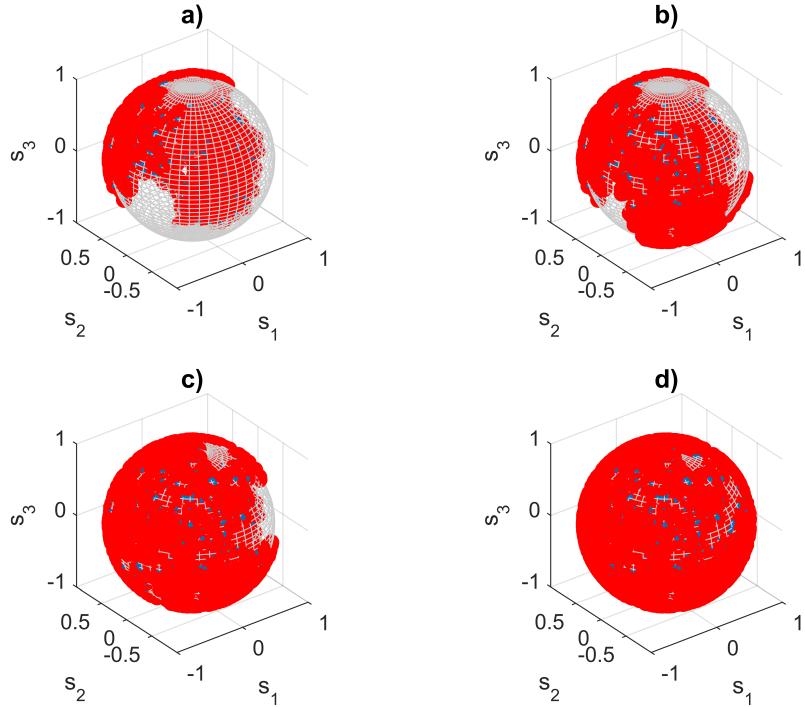


Figure 6.378: SOP evolution over time for a  $\Delta p = 2e^{-4}$

In figure 6.379 are represented the histograms of the parameters  $(\alpha_1, \alpha_2, \alpha_3)$ , which are the parameters that introduce the randomness of the model implemented.

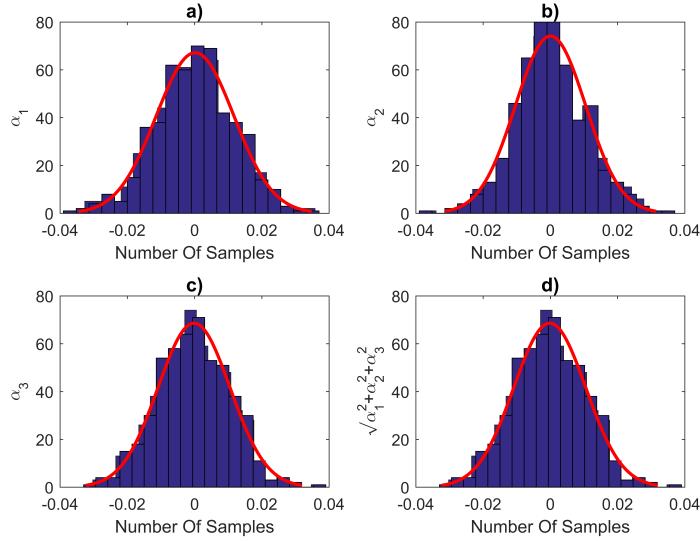


Figure 6.379: Histograms of  $(\alpha_1, \alpha_2, \alpha_3)$  for a  $\Delta p = 2e^{-4}$

Figure 6.365 shows the evolution of QBER over time for three different values of  $\Delta p$ .

### Experimental $\Delta p$ measurement

The parameter  $\Delta p$  may be related to PMD of the optical fiber. The autocorrelation in time may be written as:

$$A(t) = e^{-\frac{|\Delta t|}{t_d}}, \quad (6.257)$$

where  $t_d$  is the typical drift time for absolute polarization states. This coefficient is unique for each installed fiber and can be related with the polarization linewidth parameter  $\Delta p$ , which also depends of the installation of the optical fiber and it is also unique for each fiber. Comparing equation 6.256 with equation 6.257 we may find the relation:

$$\frac{1}{t_d} \sim \Delta p. \quad (6.258)$$

From the work presented in [2] and [3] we can write the expression for  $t_d$  as:

$$t_d = \frac{2t_0}{3w^2 D_p^2 z}, \quad (6.259)$$

where  $D_p$  is the PMD coefficient (in  $ps/\sqrt{km}$ ) of the fiber,  $w$  is the carrier frequency,  $t_0$  is a measure of the drift time of the index difference in birefringence element used to model the fiber and  $z$  is the length of the fiber. This way, we can write the  $\Delta p$  as:

$$\Delta p \sim \frac{3w^2 D_p^2 z}{2t_0}. \quad (6.260)$$

### 6.16.2 Algorithm for polarization compensation

The proposed algorithm to compensate the polarization drift through an optical fiber link is based on a rigorous method to estimate the quantum bit error rate (QBER) proposed in [3]. According with theoretical equation 6.254 the QBER depends on the values  $\theta$  and  $\varphi$  which represent a state of polarization on Poincaré sphere. Lets assume the single photon is horizontal linear polarized being the Stokes parameters [100]. The QBER distribution on Poincaré sphere will be as shown in figure 6.380. This way, we can have an idea of the SOP

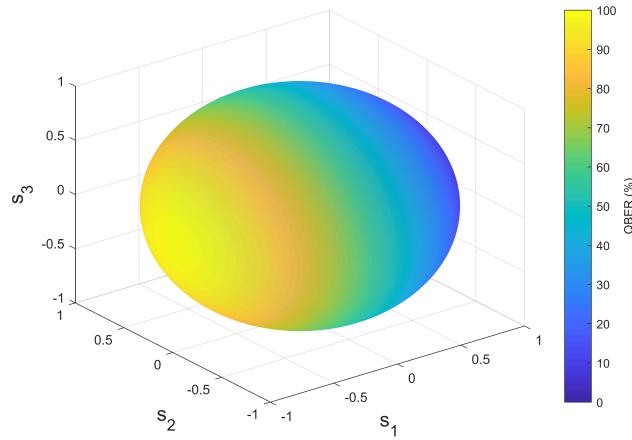


Figure 6.380: QBER distribution on Poincaré sphere.

position on the sphere knowing the value of QBER since for a specific value of QBER we can draw a meridian on the sphere.

Lets assume an example where the transmitted sequence has  $N_F$  (data frame size) bits and which is divided in two parts, one with control bits with size  $N_c$  and other with data bits with size  $N_d$ . Furthermore, two other value must be defined to guarantee the security and effectiveness of the communication protocol, a  $\text{QBER}_{\text{Min}}$  and a  $\text{QBER}_{\text{Max}}$ . The algorithm runs as follow:

1. In a first step an initial  $\widehat{\text{QBER}}$  is estimated as described in [3],

$$\widehat{\text{QBER}} = \frac{e_c}{N_c}, \quad (6.261)$$

where  $e_c$  is the number of errors in the control sequence of bits. Lets assume the initial QBER estimated is  $\widehat{\text{QBER}} = 10\%$  being possible draw the meridian in figure 6.381.

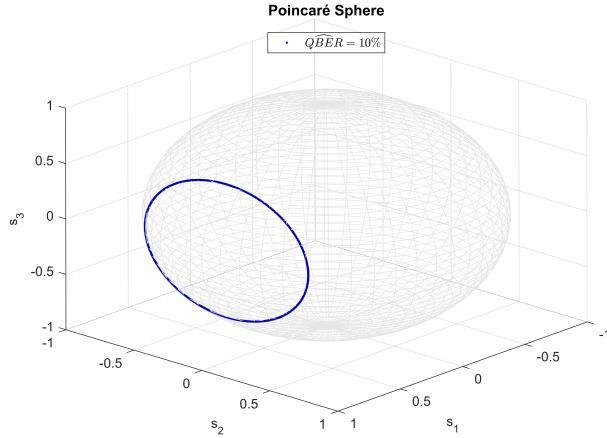


Figure 6.381:  $\widehat{QBER}$  representation on Poincaré sphere.

2. At this point three cases should be considered:

- (a)  $\widehat{QBER} < QBER_{Min}$ , which is an acceptable case and the rotation applied to compensate the polarization drift should be done using  $\theta = 2 \times \frac{\theta_{Max}}{6}$  and  $\varphi = 2 \times \frac{\varphi_{Max}}{6}$ , where  $\theta_{Max}$  and  $\varphi_{Max}$  are the maximum angles of the SOP positions over the meridian drawn.
- (b)  $QBER_{Min} < \widehat{QBER} < QBER_{Max}$ , where the compensation rotation should follow the rules defined in the previous step.
- (c)  $\widehat{QBER} > QBER_{Max}$ , where the rotation should be higher using the angles  $\theta = 2 \times \frac{\theta_{Max}}{2}$  and  $\varphi = 2 \times \frac{\varphi_{Max}}{2}$ .

Depending on the situation a rotation in EPC must be done using the appropriate angles. Lets assume a  $QBER_{Max} = 8\%$  and a  $QBER_{Min} = 1\%$ . In theory, to perform the rotation three matrices should be applied in order to rotated the points around each of the three axis which represent the Stokes Parameters of the SOP ( $S_1$ ,  $S_2$  and  $S_3$ )[4],

$$\begin{aligned} R_{S_1} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\varphi) & -\sin(\varphi) \\ 0 & \sin(\varphi) & \cos(\varphi) \end{bmatrix} \\ R_{S_2} &= \begin{bmatrix} \cos(\varphi) & 0 & \sin(\varphi) \\ 0 & 1 & 0 \\ -\sin(\varphi) & 0 & \cos(\varphi) \end{bmatrix} \\ R_{S_3} &= \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \quad (6.262)$$

Since  $\widehat{QBER} > QBER_{Max}$  the rotation should be performed using  $\theta = 2 \times \frac{\theta_{Max}}{2}$  and  $\varphi = 2 \times \frac{\varphi_{Max}}{2}$ . The result is shown in figure 6.382.

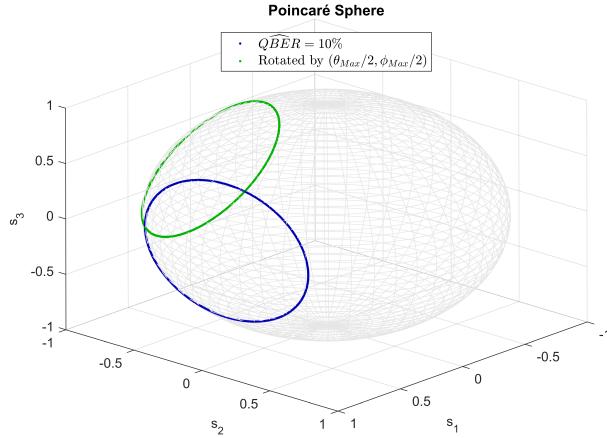


Figure 6.382: QBER distribution on Poincaré sphere rotated by  $\theta$  around  $S_3$  and by  $\varphi$  around  $S_1$  and  $S_2$ .

3. The next step is to estimate again the  $\widehat{QBER}$  and also draw other meridian on the sphere. As presented in figure 6.383 the rotated curve from the previous step intercepts the curve of the new  $\widehat{QBER}$  estimation in two points. These two points are the two possible positions of our SOP. This way, since we know the two possible locations, we should choose one of them and apply the appropriate location to move the SOP to a point where the QBER is zero. The  $\widehat{QBER}$  is estimated again and two situations must be taken into account:
  - (a) The  $\widehat{QBER}$  decreases to a value around zero and then this is the last step of the algorithm iteration.
  - (b) The  $\widehat{QBER}$  does not decrease to a value around zero and then the other intersection point should be chosen and a rotation to a point where the QBER is zero should be applied. In this case, the algorithm has one more step than the previous situation.

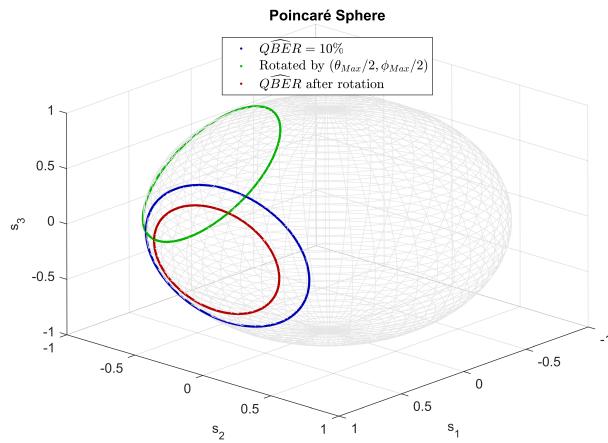


Figure 6.383:  $\widehat{QBER}$  representation on Poincaré sphere estimated after rotation.

### 6.16.3 Simulation Analysis - Algorithm for polarization compensation

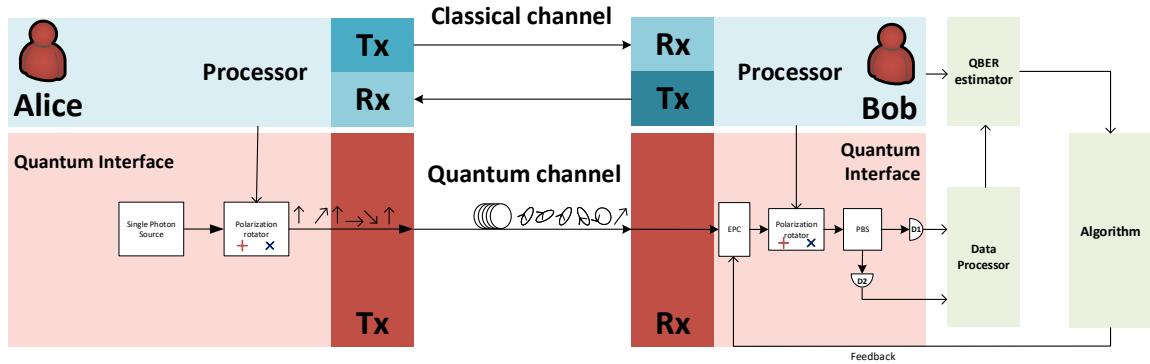


Figure 6.384: Simulation setup to test the algorithm to compensate polarization drift.

### 6.16.4 Experimental Setup

In figure 6.385 is presented the experimental setup to be performed in the lab for single basis encoding. The main goal is to build an experimental setup in which Alice sends to Bob possible states from a single orthogonal basis randomly. Bob always uses the same basis to measure the single photons sent by Alice.

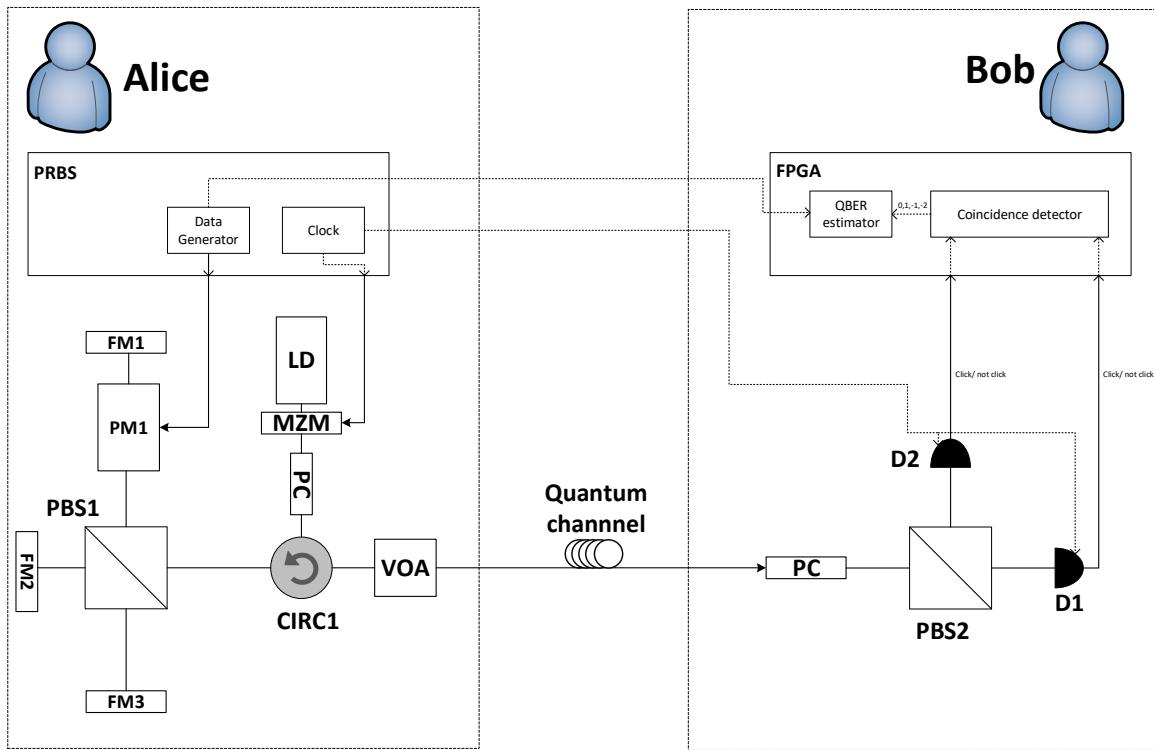


Figure 6.385: QOKD Experimental setup for single basis encoding.

Moreover, using this setup we will be able to estimate the QBER between the bit sequence sent by Alice and the bit sequence received by Bob. PRBS block will allow Alice to generate a bit sequence randomly to encode the single photons using two possible states. In addition, this block will provide a clock signal for all devices which need to be synchronized as well as to modulate the laser beam at Alice's side. In FPGA module, there is a coincidence detector block which will analyse the outputs from single photon detectors. There are four possible scenarios:

- Detector D1 clicks and detector D2 does not click. In this case the output from coincidence detector will be "0".
- Detector D2 clicks and detector D1 does not click. In this case the output from coincidence detector will be "1".
- None of the detectors click due attenuation effects in quantum channel. In this case the output from coincidence detector will be "-1".
- Both detectors click due dark counts inherent detectors specifications. In this case the output from coincidence detector will be "-2".

Furthermore, FPGA module should have an additional block regarding with QBER estimation. This block has two inputs, one from Alice data generator with the bit sequence

sent by Alice and another one from coincidence detector. This block should discard the bits that Bob measured as "-1" and "-2".

#### 6.16.4.1 Phase-modulator calibration

In order to find the voltage values to have a certain phase induced by phase-modulator, the experimental setup in figure 6.386.

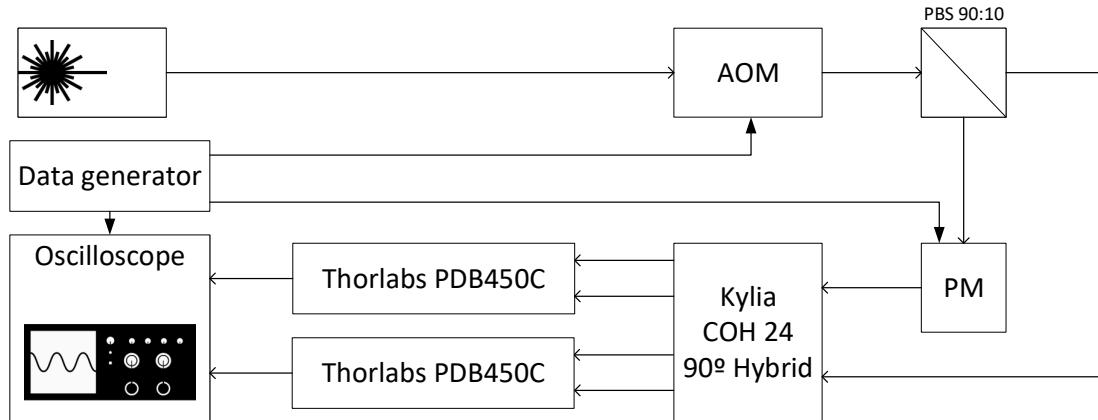


Figure 6.386: Experimental setup for phase-modulator characterization.

Data generator feed acousto-optic modulator (AOM) with a pulse signal at 1.0MHz, 5V amplitude, 250ns width, and delayed by 700ns. This way, the AOM output signal is a pulsed signal. This enters in PBS and 90% of the power follows directly to hybrid. The other 10% follows the path to phase modulator, which is fed with a square signal which the amplitude induces a certain phase change.

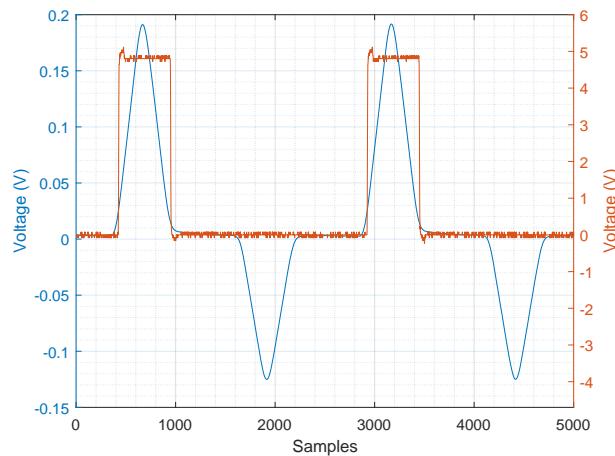


Figure 6.387: Blue: phase-modulator optical output-signal. Orange: phase-modulator input electric signal.

figure 6.387 shows that the phase change is only applied alternately between two consecutive pulses. Moreover, as measure the absolute phase is very difficult, we measure the phase difference between two consecutive pulses. In order to do that, both real and imaginary part of the signal are measured for 419 pulses when a voltage of 3V is input in phase-modulator.

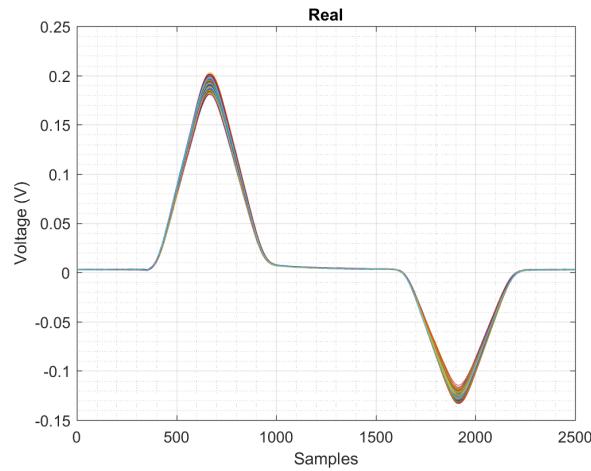


Figure 6.388: Real part of the obtained signal.

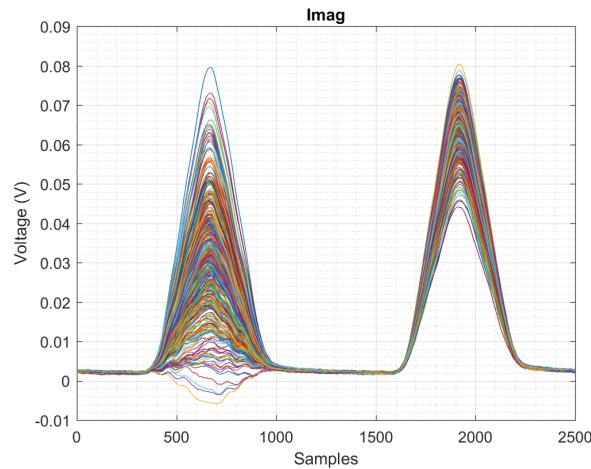


Figure 6.389: Imaginary part of the obtained signal.

The average real and imaginary signals are shown in figure 6.390.

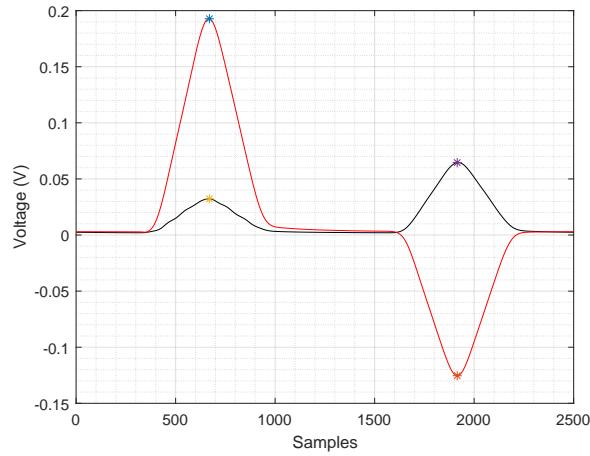


Figure 6.390: Average real and imaginary of the obtained signal.

After that, the two peaks were plotted in IQ diagram so the phase difference between the two peaks could be calculated.

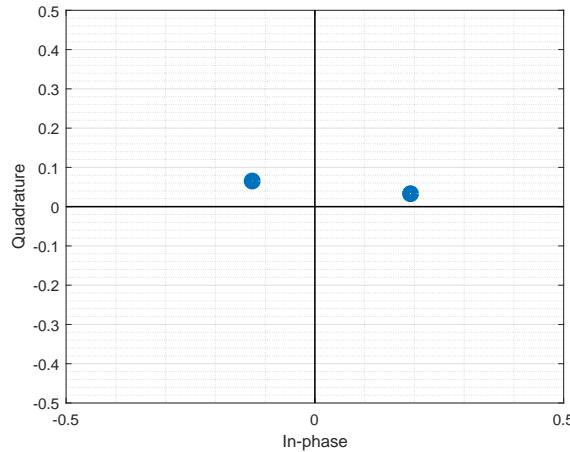


Figure 6.391: IQ diagram for the two consecutive pulses.

This procedure was done for different phase-modulator input voltages. Figure 6.392 shows the relation between the phase-modulator input voltage with the phase-change induced in the signal. It is a linear relation, so finding  $V_\pi$  we can find all the other voltage values.

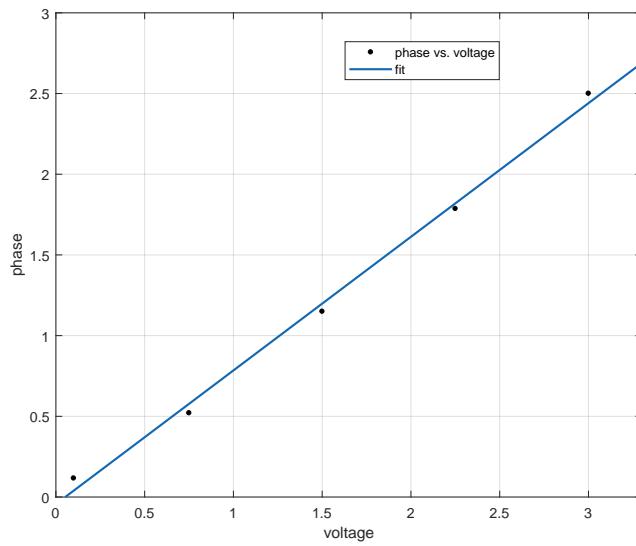


Figure 6.392: Phase change dependency on phase-modulator input voltage. Fit equation:  $y = 0.828x - 0.04362$ .

Figure 6.393 shows the relation between the phase-change and the homodyne real output voltage.

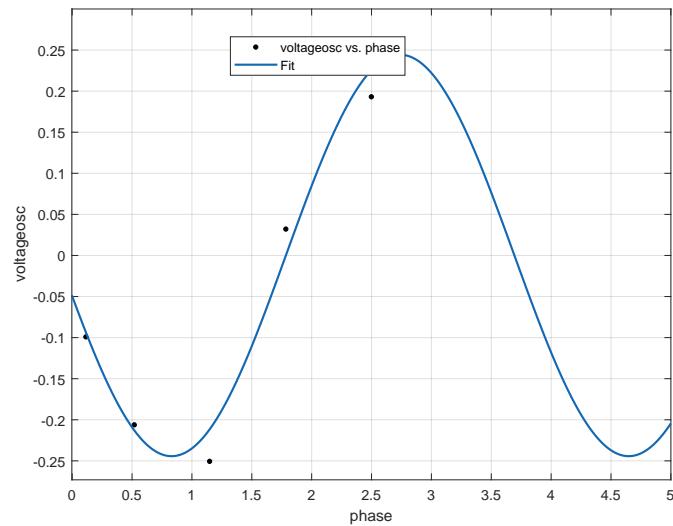


Figure 6.393: Phase change dependency on real homodyne detector output voltage. Fit equation:  $y = 0.2443 \sin(1.646x + 3.346)$ .

### 6.16.4.2 Alice's Interferometer

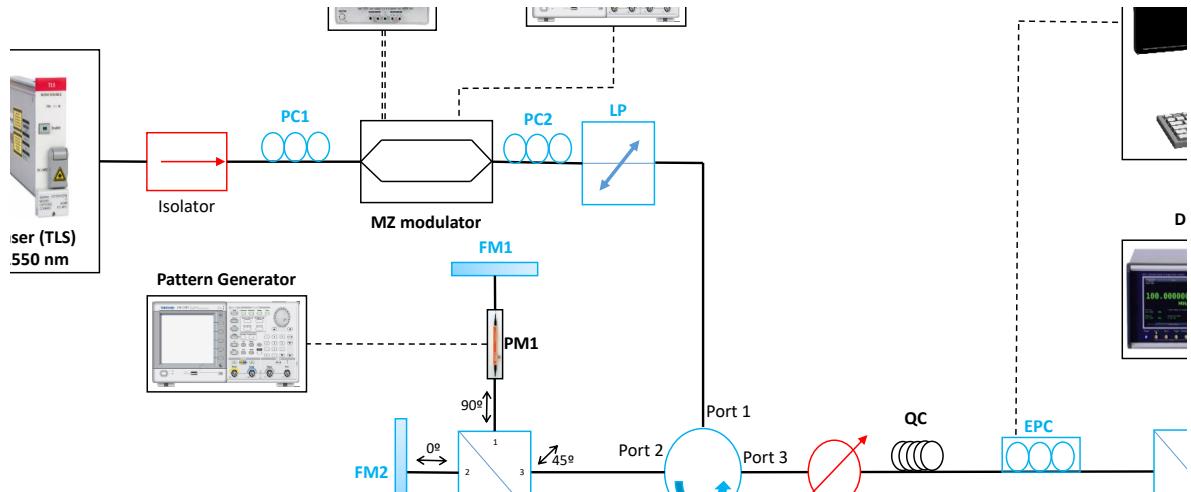


Figure 6.394: Schematic of our transmitter system (Alice). MZ, Mach-Zehnder modulator; CIRC1, optical circulator; PBS1, PBS2, polarization beam splitter; D1,D2, Single-photon detectors; FM1-FM3, faraday mirror; PM1, phase modulator; VOA, variable optical attenuator; EPC, electronic polarization controller; LP, linear polarizer; PC1, PC2, polarization controller.

Table 6.57: Number of received bits required for each boundary limit.

| Polarization States |  |
|---------------------|--|
| +45                 | $\sqrt{2}( e_x\rangle +  e_y\rangle)/2$  |
| -45                 | $\sqrt{2}( e_x\rangle -  e_y\rangle)/2$  |
| Right circular      | $\sqrt{2}( e_x\rangle + i e_y\rangle)/2$ |
| Left circular       | $\sqrt{2}( e_x\rangle - i e_y\rangle)/2$ |

### 6.16.5 Open Issues

## References

- [1] Cristian B Czegledi et al. "Polarization drift channel model for coherent fibre-optic systems". In: *Scientific reports* 6 (2016), p. 21217.
- [2] Magnus Karlsson, Jonas Brentel, and Peter A Andrekson. "Long-term measurement of PMD and polarization drift in installed fibers". In: *JOURNAL OF lightwave technology* 18.7 (2000), p. 941.
- [3] Álvaro J Almeida et al. "Continuous Control of Random Polarization Rotations for Quantum Communications". In: *Journal of Lightwave Technology* 34.16 (2016), pp. 3914–3922.
- [4] JP Gordon and H Kogelnik. "PMD fundamentals: Polarization mode dispersion in optical fibers". In: *Proceedings of the National Academy of Sciences* 97.9 (2000), pp. 4541–4550.

â—«

## 6.17 Discrete Variables Polarization Encoding Bob Processor

|                     |   |   |
|---------------------|---|---|
| <b>Student Name</b> | : | Jaymin Patel (July 1, 2018)                       |
| <b>Goal</b>         | : | Implement the BER estimation in FPGA.             |
| <b>Directory</b>    | : | sdf/dv_polarization_encoding_system_bob_processor |

### 6.17.1 System overview

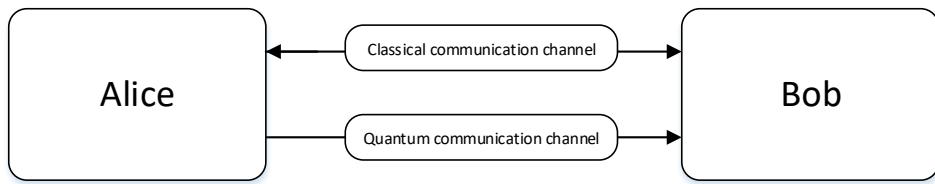


Figure 6.395: Overall system

The overall system consist of two entities communicating through optical and electrical channel, later being classical and bidirectional while former being quantum and unidirectional. The same data is transmitted through quantum and classical channels and compared with the reference of classical channel at the receiver-Bob side, based on this Qunatum Bit Error Rate QBER is calculated.

The transmitter consist of pseudo random data generator, clock source, laser and modulator. It sends the same bits on both the channels simultaneously. The receiver decodes the data on optical channel and compares it with the data received from classical channel, if the difference is noticed the error is considered to be occurred. The modulation consist polarisation change, which at the detection time decoded back to the data. The optical path or decoding may induce some error and to calculate the rate of it bob processor is implemented.

### 6.17.2 Brief Alice overview

Alice side the data is generated pseudo randomly and transmitted towards Bob via two channels, first quantum optical and second classical electrical. The laser source is modulated and information is encoded in the form of photon's one of the two polarisations. Each polarisation represents bit 0 or 1. The finer details of this set-up is out of the bounds of this chapter.

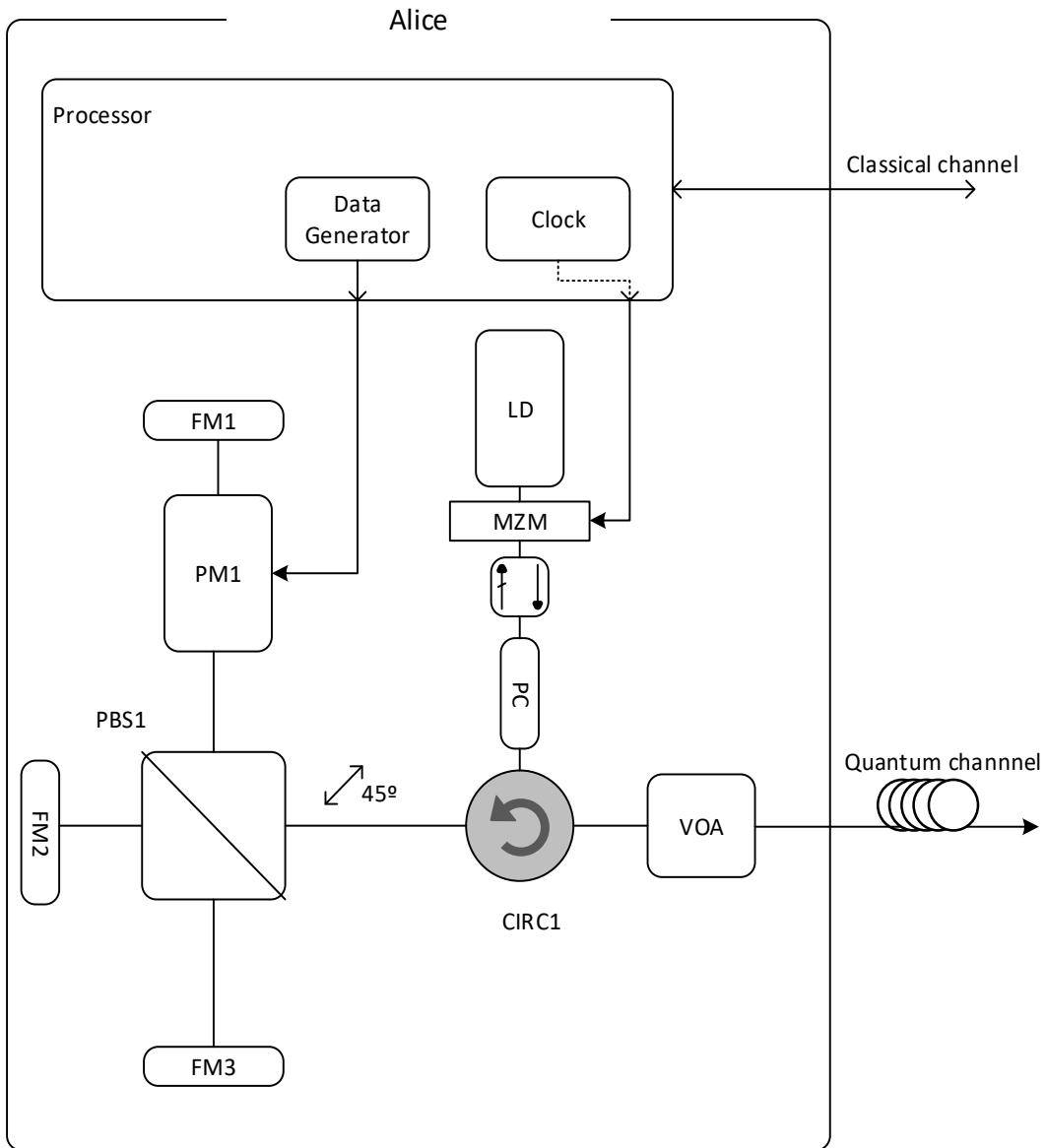


Figure 6.396: Alice overview

### 6.17.3 Bob overview

The receiver-Bob have two inputs in the form of classical and quantum communication channel. The optical signal is converted back into the electrical signal with the help of the setup consisting of polarisation controller, polarisation beam splitter and single photon detectors. After this conversion the Bob processor compare the data received from electrical channel. Bob processor triggers the SPDs at regular intervals to receive the incoming photon stream.

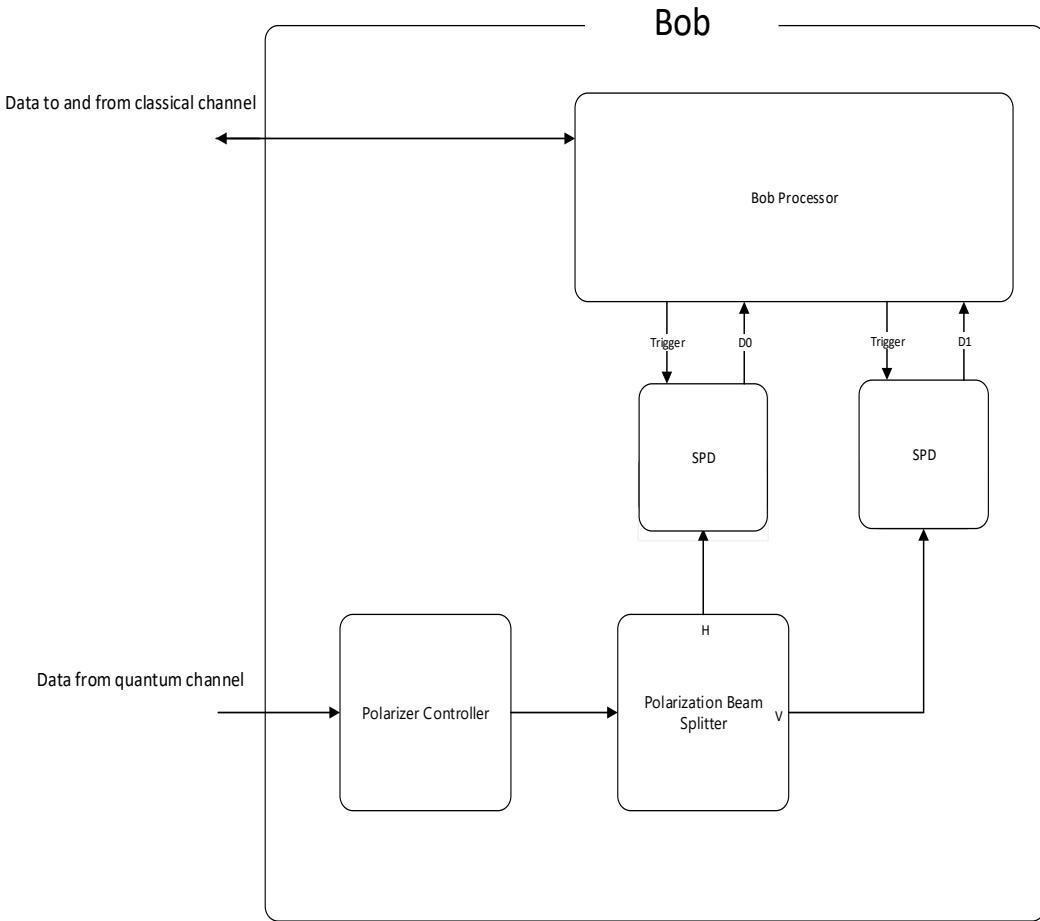


Figure 6.397: Overall system block diagram

The incoming photons are polarised in one of the two ways to represent the bit 0 or bit 1. The Bob end separates the photons with different polarisation and sends it to the detectors where they are converted into the electrical pulses. The Bob processor triggers the detectors to open the incoming window and then also reads the output, based on it the processor takes the decision whether it has received valid bit or an error has been occurred. Upon valid bit reception it proceed to comparison and calculation of error rate.

#### 6.17.4 Bob processor overview

Bob processor is responsible to calculate BER and triggering the detectors. The bob processor has two inputs, the pseudo random data transmitted via electrical signal and demodulated data coming from single photon detectors. This detector needs trigger pulse to activate them so as to capture the transmitted photon. Bob processor is designed to trigger the detectors. It captures the output and interprets the two outputs. If both the detector gives output or none of them gives, processor skips the bit. In the case of either detector gives output the processor consider it as valid bit received, then it compares the received bit with the bit

received on electrical signal. If they differs then error count is increased.

To synchronise with the transmitted data from electrical channel the Bob processor expects certain frame structure, which consists of header of continuous 10 1's followed by 90 bits of data. If the error rate while receiving the header is high then the remaining frame is dropped and processor waits for the next header. The threshold error rate is configurable via 'config' file.

The processor compares the received data over electrical channel with the data received from optical channel, if both are different then error is stored and after certain no of bit transmission it displays the bit error rate. This rate indicates the error per no of bit transmitted.

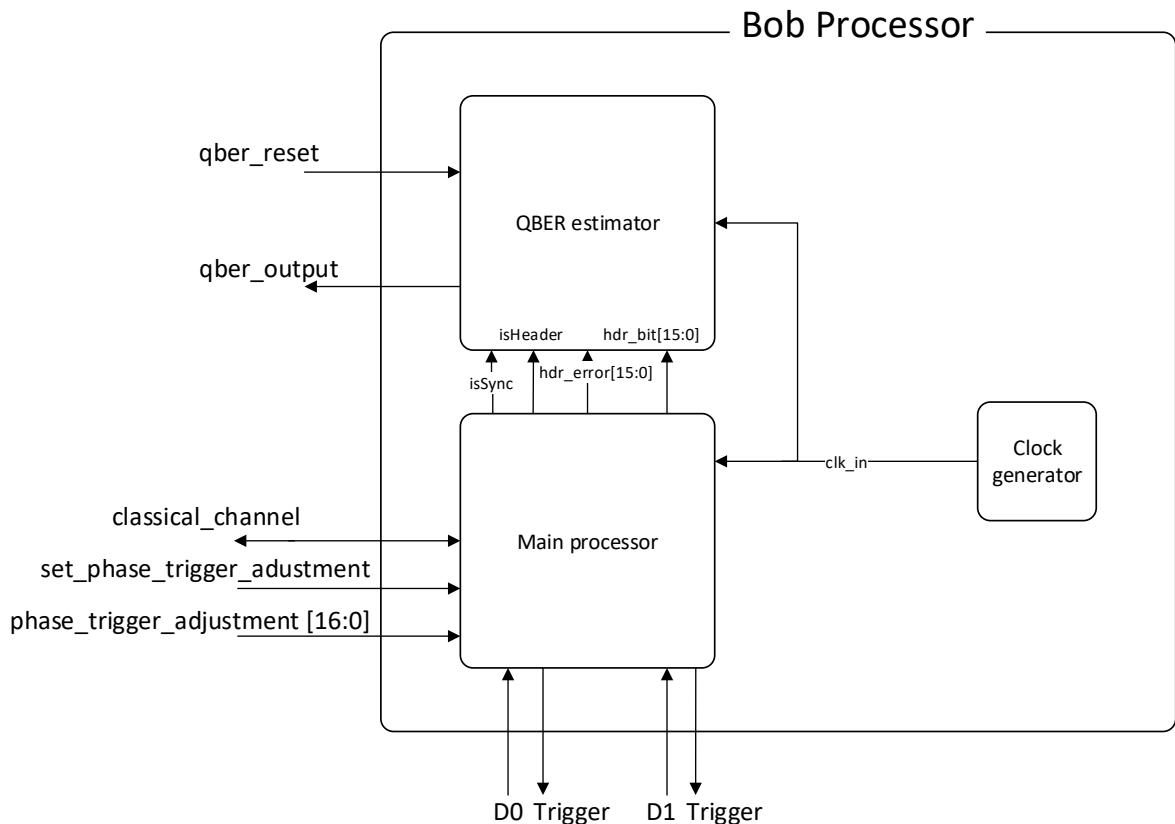


Figure 6.398: Bob processor block diagram

#### 6.17.5 Frame structure

The Bob processor is designed to receive the data in certain frame format which consist of header and data part. The header part is 10 bits long and the data part is 90 bits. The processor compares the header bits and senses the error rate in header beats, if it is higher than the predefined rate, data part is omitted in BER calculation.

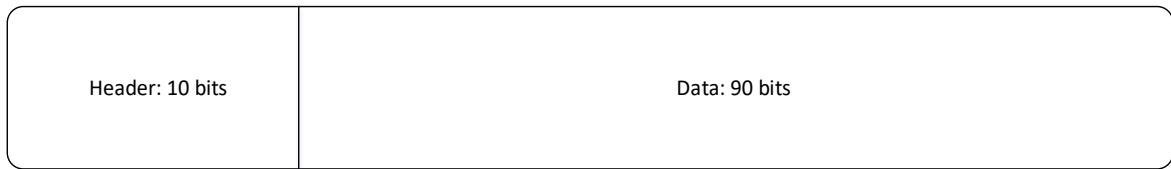


Figure 6.399: Frame structure

### 6.17.6 Configuration file

The parameters like error rate in header and many others can be configured via 'config.txt' file. The processor will read the file and sets the following parameters accordingly. The format of this file is ASCII.

The threshold error rate for header is the error rate detection the reception errors in header beyond this level will result in discard of the data section.

This file is stored in on board NVM. The file system and processor stores, reads and modifies the configuration file. This file can be uploaded on the device via serial port.

Header Bits: the header with which the incoming bits will be compared. Clock: the input frequency of the clock Header Phase: the sync of header with the rising or falling edge of the clock. Trigger Phase: trigger sync with rising or falling edge of clock.

The header and the data length is also included in this file. The format of the file will be as following.

Header Bits: The data to be expected in header.

Header Basis: The header basis.

Clock: The input source clock of all hardware blocks having 1MHz frequency.

Header Phase: The header sync with clock's rising or falling edge.

Trigger Phase: The trigger time adjustment.

Errors for async state: The no of errors that would lead FSM into async state.

Errors for sync state: The no of errors that would lead FSM into sync state.

Frame count to go in async state: The no of consecutive frames with errors more than mentioned above will lead FSM to async state.

Frame count to go in sync state: The no of consecutive frames with errors more than mentioned above will lead FSM to sync state.

Configuration Header

#####

Header Bits: 1111111111

Header Basis: 1111111111

Lower Bound: 1e-3

Confidence Level: 0.95

Clock: 1E6

Header Phase: 0  
 Trigger Phase: 0  
 Errors for async state: 5  
 Errors for sync state: 2  
 Frame count to go in async state: 2  
 Frame count to go in sync state: 2

### 6.17.7 VHDL Implementation of Bob processor

### 6.17.8 Main processor

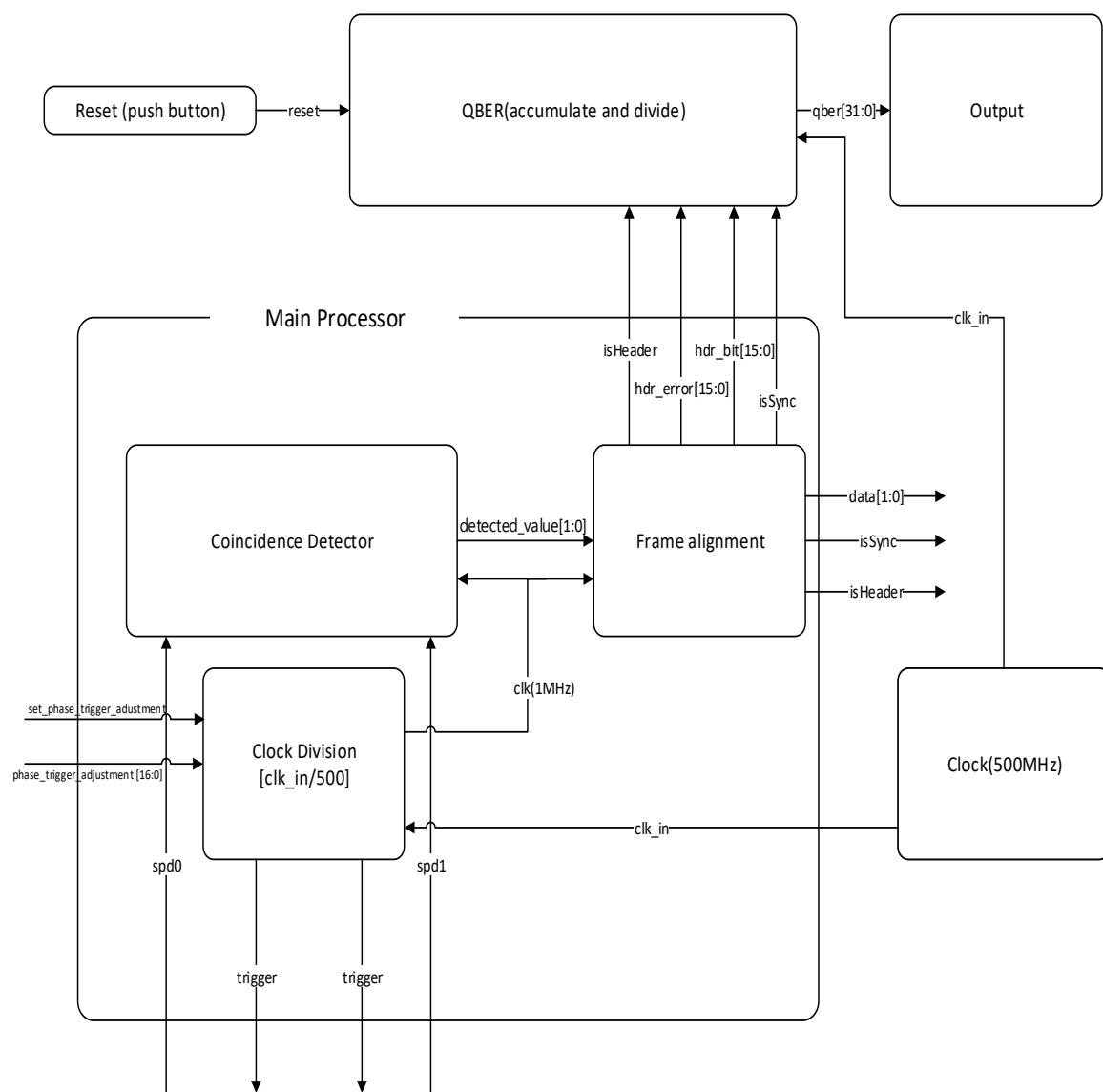


Figure 6.400: Main processor

Main processor receives output from single photon detectors and produces data; QBER for header is calculated. The counter is counting the incoming bits to differentiate the header and data part. The QBER block is basically the division of correctly received bit by the errors occurred in transmission. The main processor extracts data from two sensors and also differentiate between header and data. The main processor also generates trigger signals for the single photon detectors.

#### 6.17.9 Main processor: Coincedencedetector

This entity generates data on '*Data\_vec*' according to the two inputs '*D0*' and '*D1*'. The output ranges from 0 to 3 representing either 0 or 1 reception, the invalid situation where both or none detector gives output. The output is stored in a buffer for further processing.

```
-----
-- Authors: Jaymin Rasikbhai Patel
-----

LIBRARY ieee;
USE ieee.std_logic_1164.all;
--USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
use ieee.math_real.all;
-----

ENTITY Coincedencedetector IS

PORT(
    clk      : IN std_logic;
    SPD0     : IN std_logic;
    SPD1     : IN std_logic;
    Data_out : OUT std_logic;
    InValid  : OUT std_logic;
    din      : OUT std_logic_vector(1 DOWNTO 0)
    --Trig0   : OUT bit;
    --Trig1   : OUT bit;
);
END Coincedencedetector;

ARCHITECTURE Struct1 OF Coincedencedetector IS

BEGIN

P1: process(clk)
begin
    if rising_edge(clk) then
        --Trig0 <= '1';                                --trigger the SPD as soon as the clock is high --TODO : to
        -- discuss exactly how the SPD will be triggered
        --Trig1 <= '1';
        if(SP0 = '0') AND (SPD1 = '1') then --detect the output of SPD
            Data_out <= '0';
            InValid <= '0';
            din <= "01"; --x"1"; --din <= x"00";
        elsif (SPD0 = '1') AND (SPD1 = '0') then
            Data_out <= '1';
            InValid <= '0';
        end if;
    end if;
end process;

```

```

din <= "10" ; --x"2"; --din <= x"01";
elsif (SPD0 = '1') AND (SPD1 = '1') then
  Data_out <= '0';
  InValid <= '1';
  din <= "11"; --x"3"; --din <= x"FF";
else
  Data_out <= '1';
  InValid <= '1';
  din <= "00" ; --x"0"; --din <= x"fe";
end if;
end if;
--wait until clk = '0';
--Trig0 <= '0';
--Trig1 <= '0';
end process P1;

END Struct1;

```

### 6.17.10 Main processor: Frame alignment

The FIFO is read by this block and it has two states 'sync' and 'async'. The error count in each header is monitored and if it is higher than 5 counts in 2 consecutive frames the state is changed to async from initial sync state. The state machine in async state tries to go to sync state by getting 2 or less errors in 2 consecutive frames. In the sync state it provides the output to QBER estimator block.

This block reads FIFO and provides output to QBER estimator. The signal '*isSync*' states the current status of FSM. The '*isHeader*' differentiates between header and data output. The '*hdr\_bit[15 : 0]*' is the number of valid bit received. '*hdr\_error[15 : 0]*' is the number of error in reception. These previous two numbers are currently used in estimation of QBER. '*data[1 : 0]*' is the data bits received.

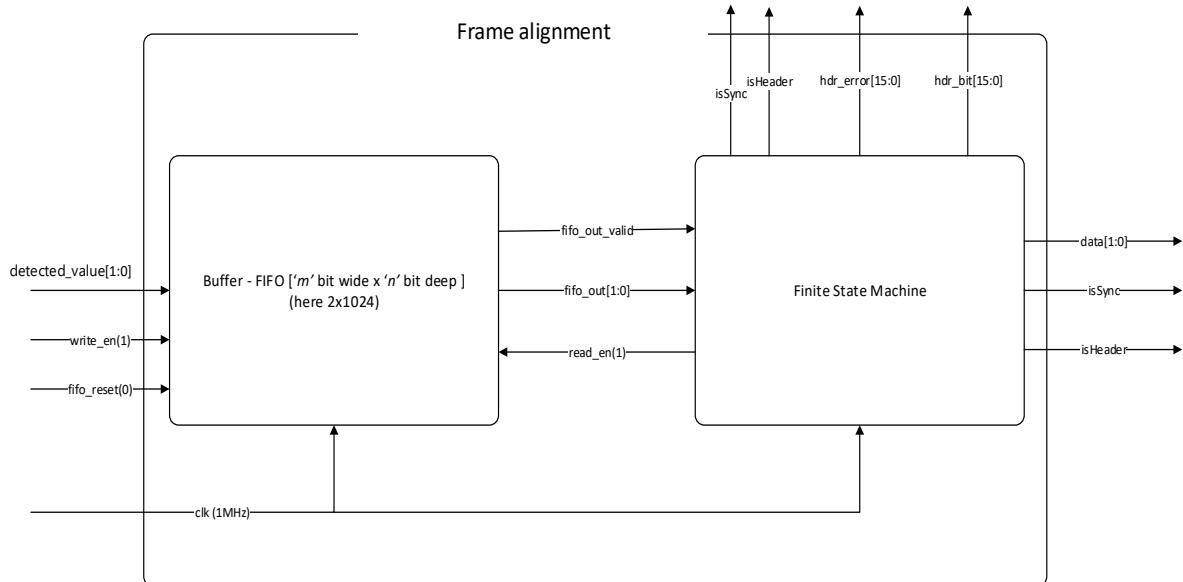


Figure 6.401: Frame Alignment block

The frame alignment is functionally described by FSM which is depicted as follows. The states and transition conditions can be configured via configuration file depending on the experimental set up.

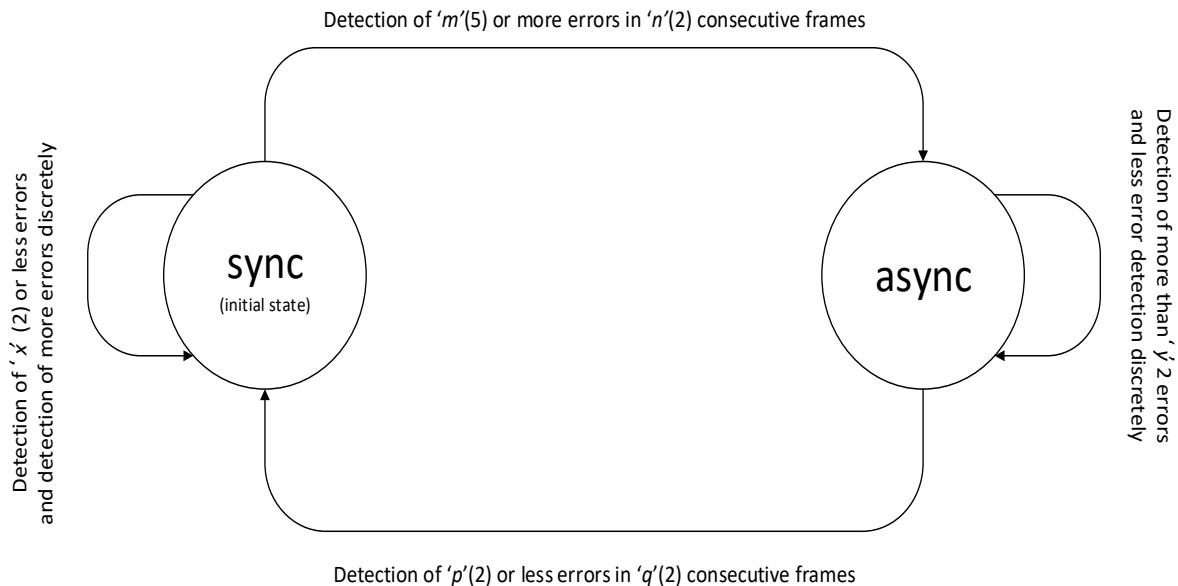


Figure 6.402: Frame Alignment - Finite State Machine

This simulation shows FSM going out of sync when it receives error higher than the pre-set no in consecutive frames. The '*outvalid*' is not raised once it goes in '*async*' state.



Figure 6.403: Frame Alignment

```
-----
-- Company:
-- Engineer:
--
-- Create Date: 14:03:58 11/15/2018
-- Design Name:
-- Module Name: q4_Frame_sync - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE ieee.std_logic_unsigned.all;
use ieee.math_real.all;
use ieee.Numeric_Std.all;
use IEEE.STD_LOGIC_ARITH.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
```

```

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity q4_Frame_sync is
  GENERIC(
    Nbit    : integer := 16;
    Hdrlen : integer := 10;
    Datalen : integer := 20;
    Framerlen : integer := 30;

    async_error_no : integer := 4;
    async_error_frames : integer := 2;
    sync_error_no : integer := 2;
    sync_error_frames : integer := 2
  );
  Port ( fifo_out  : in STD_LOGIC_VECTOR (1 downto 0);
         rd_en      : out STD_LOGIC;

         clk          : in STD_LOGIC;
         out_valid   : out STD_LOGIC;

         error        : out STD_LOGIC_VECTOR (Nbit-1 DOWNTO 0);
         header       : OUT std_logic := '1';
         hdr_bit     : out STD_LOGIC_VECTOR (Nbit-1 DOWNTO 0));
end q4_Frame_sync;

architecture Behavioral of q4_Frame_sync is

  signal header_sig           : STD_LOGIC;
  signal out_valid_sig        : STD_LOGIC := '1' ;
  signal isSynced             : STD_LOGIC := '1';
  signal bit_count_temp       : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');
  signal error_count_temp     : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');
  signal both_click_count_temp : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');
  signal no_click_count_temp  : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');

  signal bit_count_valid_temp : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0'); --ever increasing
  signal error_count_frame    : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0'); --ever increasing

  signal async_frame_count_sig : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');
  signal sync_frame_count_sig : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');
  signal async_frame_conseq_sig : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');
  signal sync_frame_conseq_sig : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');

begin
P1: process(clk,header_sig)
begin

  if rising_edge(clk) then
    rd_en <= '1'; --fetch data from FIFO buffer

    if bit_count_temp < Hdrlen then --header parsing
      bit_count_temp <= bit_count_temp + "0000000000000001";
      header <= '1'; header_sig <= '1'; out_valid_sig <= '1';
      if fifo_out = "10" and fifo_out_valid = '1' then --fifo_out_valid : we are NOT reading beyond
        written content in FIFO
        error_count_temp <= error_count_temp + "0000000000000001" ;
    end if;
  end if;
end process;
end;

```

```

        error_count_frame <= error_count_frame + "0000000000000001" ;
        bit_count_valid_temp <= bit_count_valid_temp + "0000000000000001" ;
    elsif rd_data = "01" then --as of now header consists of all 1s
        bit_count_valid_temp <= bit_count_valid_temp + "0000000000000001" ;
    elsif rd_data = "11" then
        both_click_count_temp <= both_click_count_temp + "0000000000000001" ;
    elsif rd_data = "00" then
        no_click_count_temp <= no_click_count_temp + "0000000000000001" ;
    end if; --if fifo_out = x"2" then

    elsif (bit_count_temp >= Hdrlen) AND (bit_count_temp < (Framelen-1)) then --data part parsing
        bit_count_temp <= bit_count_temp + "0000000000000001";
        header <= '0'; header_sig <= '0';

    if out_valid_sig = '1' then
        out_valid_sig <= '0';

        if async_frame_conseq_sig > 0 then
            async_frame_conseq_sig <= async_frame_conseq_sig - "0000000000000001";
        end if;
        if sync_frame_conseq_sig > 0 then
            sync_frame_conseq_sig <= sync_frame_conseq_sig - "0000000000000001";
        end if;

        if error_count_temp >= async_error_no AND async_frame_count_sig >= async_error_frames AND
            isSynced = '1' then -- going out of sync
            isSynced <= '0';
            async_frame_count_sig <= "0000000000000000";
        elsif error_count_temp >= async_error_no AND async_frame_count_sig < async_error_frames AND
            isSynced = '1' then --errors getting reported, may the state go in async
            hdr_error <= error_count_frame;
            hdr_bit <= bit_count_valid_temp;
            valid_header <= '1'; --out_valid_sig <= '0';
            if async_frame_conseq_sig > 0 then --only consequent frames having sufficient errors will be
                considered
                async_frame_count_sig <= async_frame_count_sig + "0000000000000001";
            elsif async_frame_conseq_sig = 0 then --when encountering high errors for the first time
                async_frame_conseq_sig <= std_logic_vector(to_unsigned(async_error_frames, Nbit));
            end if;

        elsif error_count_temp <= sync_error_no AND sync_frame_count_sig >= sync_error_no AND isSynced =
            '0' then -- synchronised!
            isSynced <= '1';
            sync_frame_count_sig <= "0000000000000000";
        elsif error_count_temp < sync_error_no AND sync_frame_count_sig < sync_error_no AND isSynced =
            '0' then --errors are less, may the state go in sync
            if sync_frame_conseq_sig > 0 then --only consequent frames with low errors drages the state in
                sync
                sync_frame_count_sig <= sync_frame_count_sig + "0000000000000001";
            elsif sync_frame_conseq_sig = 0 then --when encounter low error for the first time
                sync_frame_conseq_sig <= std_logic_vector(to_unsigned(sync_error_frames, Nbit));
            end if;
        elsif isSynced = '1' then
            error <= error_count_frame;
            bits <= bit_count_valid_temp;
            --if header_sig = '0' and out_valid_sig = '1' then
            valid_header <= '1'; --out_valid_sig <= '0';
            --end if;
        end if; --error_count_temp > async_error_no AND async_frame_count_sig > async_error_frames AND
            isSynced = 1 then -- going out of sync

    end if;

```

```

else
    bit_count_temp <= "0000000000000000";
    error_count_temp <= "0000000000000000";
end if; --if bit_count_temp < Hdrlen then

end if; --if rising_edge(clk) then

if falling_edge(clk) then
    --rd_en <= '0';           --if this pulse is not wide enough for FIFO , have to move this from here
    if out_valid_sig = '0' then
        valid_header <= '0';   --if this pulse is not wide enough for divider, have to move this from
        here
        --out_valid_sig <= '0';
    end if;
end if; --if falling_edge(clk) then

--if falling_edge (header_sig) then

--end if;

end process P1;
end Behavioral;

```

### 6.17.11 Main processor: FIFO buffer

The output of coincidence detector is the two bit value representing either 0 or 1 or the remaining invalid states of SPDs. The bit stream is stored in a 2x1024 FIFO buffer. The frame synchroniser is reading from this buffer and giving error and bit counts to QBER estimator.

```

-----
-- Company:
-- Engineer:
--
-- Create Date: 14:01:48 11/13/2018
-- Design Name:
-- Module Name: fifo_qber - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```

```

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity fifo_qber is
  Port ( clk : in STD_LOGIC;
         reset : in STD_LOGIC;
         detected_value : in STD_LOGIC_VECTOR (1 downto 0);
         fifo_out : out STD_LOGIC_VECTOR (1 downto 0);
         write_enable : in STD_LOGIC;
         full : out STD_LOGIC;

         empty : out STD_LOGIC;
         valid_header : out STD_LOGIC;
         read_enable : in STD_LOGIC);

end fifo_qber;

architecture Behavioral of fifo_qber is
COMPONENT circular_buf
  PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
    full : OUT STD_LOGIC;

    empty : OUT STD_LOGIC;
    valid : OUT STD_LOGIC
  );
END COMPONENT;
begin

your_instance_name : circular_buf
  PORT MAP (
    clk => clk,
    rst => reset,
    din => detected_value,
    wr_en => write_enable,
    rd_en => read_enable,
    dout => fifo_out,
    full => full,

    valid => valid_header,
    empty => empty
  );
end Behavioral;

```

### 6.17.12 Main processor: Clock management and trigger

The clock source provides high frequency clock of 500 MHz and the clock divider is used to generate 1 MHz clock and this clock is used as the input to other logical blocks. The

output 'trigger' is generated once in the slow clock period and has brief pulse width of the high speed clock period. In this case pulse of 2 nanosecond is generated after every 1 microsecond.

### 6.17.13 Main processor: Trigger phase adjustment

The trigger can be delayed or advanced by specific no of periods of fast clock by a push button. The input value is represented by sliding switches on board. This shift is made once per push button switch press. The amount of time delay or advance in trigger signal can be tuned in experimental set up.

```
-----
-- Authors: Jaymin Rasikbhai Patel
-----

LIBRARY ieee;
USE ieee.std_logic_1164.all;
--USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
use ieee.math_real.all;
-----

ENTITY Q3_freqdiv IS
  GENERIC(
    Nbit    : integer := 16;
    inputfreq : integer := 500;
    halfinputfreq : integer := 250;
    doubleinputfreq : integer := 1000
  );
  PORT(
    clk_in      : IN std_logic;
    trigger0     : OUT std_logic;
    trigger1     : OUT std_logic;
    clk          : OUT std_logic := '1';
    set_phase_trigger_adustment : in std_logic;
    phase_trigger_adjustment : in std_logic_vector (Nbit-1 DOWNTO 0)
  );
END Q3_freqdiv;

ARCHITECTURE Struct3 OF Q3_freqdiv IS

  signal bit_count_temp : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');

BEGIN
  P3: process(clk_in, set_phase_trigger_adustment)
  begin
    if rising_edge(clk_in) then
      bit_count_temp <= bit_count_temp + "0000000000000001";
      if bit_count_temp < halfinputfreq then
        clk <= '1';
        trigger0 <= '0'; trigger1 <= '0';
      elsif (bit_count_temp >= halfinputfreq) AND (bit_count_temp < (inputfreq-1)) then
        clk <= '0';
      else

```

```

        bit_count_temp <= "0000000000000000";
        trigger0 <= '1'; trigger1 <= '1';
    end if;
end if;
if rising_edge(set_phase_trigger_adustment) then
    bit_count_temp <= bit_count_temp + phase_trigger_adjustment;
end if;
end process P3;

END Struct3;

```

#### 6.17.14 QBER estimator

This block divides the two input signals and produces the error rate. The result is always going to be less than 1 and greater than 0, so the output is considered as fraction with 0 integer part. The output is the fraction spanning from bit 0 to 15. This is the binary representation of fraction number. In this type the fraction is normalised over the maximum range so decimal figure can be known by dividing it with 65536 ( $2^{16}$ ). The division result is generated in one clock. The divider block has two outputs one is the result and other indicates the validity. This block also accumulates errors and data bits. The reset is used to clear these counts.

The '*ans*' is the 32 bit fractional number with most significant 16 bit as integer and rest as fractional part.

---

```

-- Company:
-- Engineer:
--
-- Create Date: 12:40:18 10/24/2018
-- Design Name:
-- Module Name: divvhdl - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity q3_h_divvhdl_ip is
    Port (

```

---

```

    hdr_error : in STD_LOGIC_VECTOR (15 downto 0);
    hdr_bit : in STD_LOGIC_VECTOR (15 downto 0);
    n_v      : in STD_LOGIC;
    clk      : in STD_LOGIC;
    d_v      : in STD_LOGIC;
    qber     : out STD_LOGIC_VECTOR (31 downto 0);
    ans_v    : out STD_LOGIC);
end q3_h_divvhdl_ip;

architecture Behavioral of q3_h_divvhdl_ip is
COMPONENT div
PORT (
    clk          : IN STD_LOGIC;
    s_axis_divisor_tvalid : IN STD_LOGIC;
    s_axis_divisor_tdata : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    s_axis_dividend_tvalid : IN STD_LOGIC;
    s_axis_dividend_tdata : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    m_axis_dout_tvalid : OUT STD_LOGIC;
    m_axis_dout_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END COMPONENT;
begin
your_instance_name : div
PORT MAP (
    clk => clk,
    s_axis_divisor_tvalid => d_v,
    s_axis_divisor_tdata => hdr_bit,
    s_axis_dividend_tvalid => n_v,
    s_axis_dividend_tdata => hdr_error,
    m_axis_dout_tvalid => ans_v,
    m_axis_dout_tdata => qber
);
end Behavioral;

```

### 6.17.15 Output

The output is going to be displayed on the LCD and updated every time when QBER generates the output. The display block makes the normalised output of the QBER again in the human readable format.

### 6.17.16 Simulation of Bob processor

Current simulation shows no of errors and transmitted bits. The following figure shows the signal named ber\_out; which currently shows the QBER in header.

The following simulation shows one scenario where system is receiving higher errors in consecutive frames forcing it to go in '*async*' state. This can be seen by '*ber\_valid*' signal not getting high after few header reception. The transition to '*sync*' state can also be observed by tweaking the input and having lower errors in few constitutive headers.

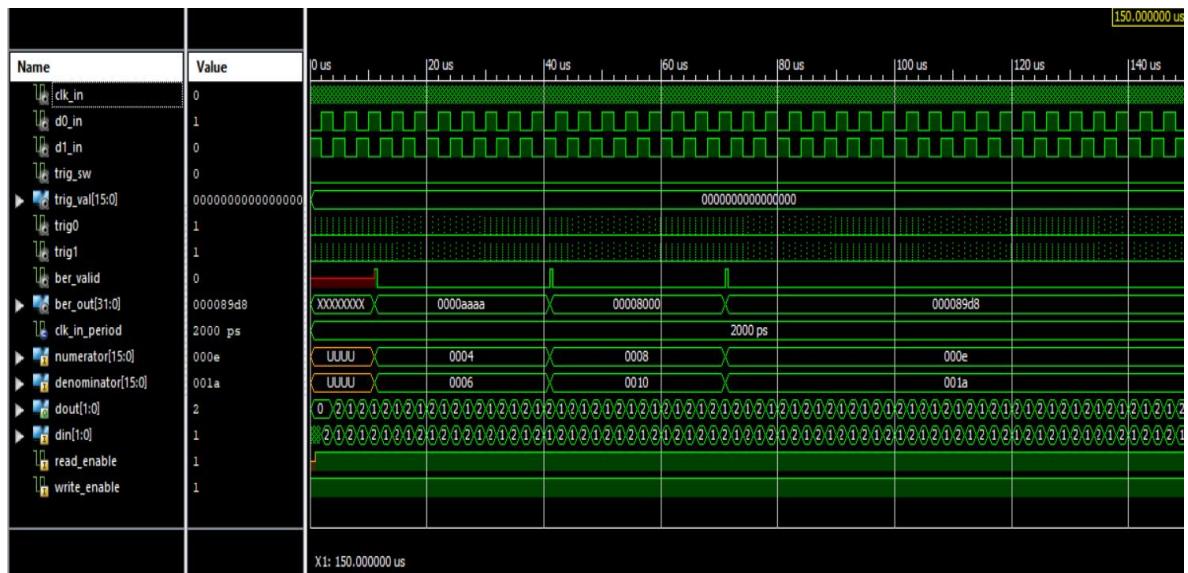


Figure 6.404: Simulation Result

### 6.17.17 VHDL code

### 6.17.18 Main entity

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.all;

ENTITY Q4_main_processor IS
  GENERIC(
    Nbit    : integer := 16;
    Hdrlen : integer := 10;
    Datalen : integer := 20;
    Framelen : integer := 30;

    async_error_no : integer := 4;
    async_error_frames : integer := 2;
    sync_error_no : integer := 2;
    sync_error_frames : integer := 2;

    inputfreq : integer := 500;
    halfinputfreq : integer := 250;
    doubleinputfreq : integer := 1000
  );
  PORT(
    clk_in      : IN std_logic;
    D0_in       : IN std_logic;
    D1_in       : IN std_logic;
    trig0       : OUT std_logic;
    trig1       : OUT std_logic;
    ...
  );
END ENTITY;

```

```

set_phase_trigger_adustment : in std_logic;
phase_trigger_adjustment : in std_logic_vector (Nbit-1 DOWNTO 0);
ber_valid      : out std_logic;

BER_out       : OUT std_logic_vector (31 DOWNTO 0)

);

END Q4_main_processor;

ARCHITECTURE Struct OF Q4_main_processor IS

-- Internal signal declarations
SIGNAL clk, fifo_read_enable_sig, fifo_out_valid_sig,hdr_out_valid_sig,data_out_sig,valid_check :
std_logic ;
SIGNAL hdr_sig, fifo_reset, full_sig,empty_sig : std_logic ;
signal fifo_wr_en_sig : std_logic := '1';

signal hdr_bits_count_sig : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');
signal hdr_error_count_sig : std_logic_vector (Nbit-1 DOWNTO 0):= (others => '0');
signal data_out_sig_vec,rd_data_sig,data_vec_sig : std_logic_vector(1 DOWNTO 0);
-- Component Declarations

COMPONENT Coincedencedetector

PORT(
clk      : IN std_logic;
SPD0    : IN std_logic;
SPD1    : IN std_logic;
Data_out  : OUT std_logic;
din     : OUT std_logic_vector(1 DOWNTO 0);
InValid   : OUT std_logic

);

END COMPONENT;

component Q3_freqdiv
GENERIC(
Nbit      : integer := 16;
inputfreq : integer := 500;
halfinputfreq : integer := 250;
doubleinputfreq : integer := 1000
);
PORT(
clk_in      : IN std_logic;
trigger0    : out std_logic;
trigger1    : out std_logic;
clk         : OUT std_logic := '1';
set_phase_trigger_adustment : in std_logic;
phase_trigger_adjustment : in std_logic_vector (Nbit-1 DOWNTO 0)
);
end component;

COMPONENT q4_Frame_sync
GENERIC(
Nbit      : integer := 16;
Hdrlen : integer := 10;
Datalen : integer := 90;
Frameolen : integer := 100;
async_error_no : integer := 4;
async_error_frames : integer := 2;
sync_error_no : integer := 2;
sync_error_frames : integer := 2

```

```

);
PORT(
    fifo_out      : in STD_LOGIC_VECTOR (1 downto 0);
    rd_en        : out STD_LOGIC;
    valid_header : in STD_LOGIC;
    clk          : in STD_LOGIC;
    valid_header : out STD_LOGIC;
    hdr_error    : out STD_LOGIC_VECTOR (Nbit-1 DOWNTO 0);
    header       : OUT std_logic := '1';
    hdr_bit      : out STD_LOGIC_VECTOR (Nbit-1 DOWNTO 0)
);
END COMPONENT;

COMPONENT fifo_qber

PORT(
    clk : in STD_LOGIC;
    reset : in STD_LOGIC;
    detected_value : in STD_LOGIC_VECTOR (1 downto 0);
    fifo_out : out STD_LOGIC_VECTOR (1 downto 0);
    write_enable : in STD_LOGIC;
    full : out STD_LOGIC;
    empty : out STD_LOGIC;
    valid_header : out STD_LOGIC;
    read_enable : in STD_LOGIC
);
END COMPONENT;

COMPONENT q3_h_divvhdl_ip
Port (
    hdr_error : in STD_LOGIC_VECTOR (Nbit-1 downto 0);
    hdr_bit : in STD_LOGIC_VECTOR (Nbit-1 downto 0);
    n_v      : in STD_LOGIC;
    clk      : in STD_LOGIC;
    d_v      : in STD_LOGIC;
    qber     : out STD_LOGIC_VECTOR (31 downto 0);
    ans_v    : out STD_LOGIC);
end COMPONENT;

BEGIN

CD_1 : Coincedencedetector
PORT MAP(
    clk      => clk,
    SPD0    => D0_in,
    SPD1    => D1_in,
    Data_out => data_out_sig,
    din     => data_out_sig_vec,
    InValid => valid_check
);

DEMX_1 : Q3_freqdiv
GENERIC MAP(
    Nbit    => Nbit,
    inputfreq => inputfreq,
    halfinputfreq => halfinputfreq,
    doubleinputfreq => doubleinputfreq
)
PORT MAP(
    clk_in      => clk_in,
    trigger0    => trig0,
    trigger1    => trig1,
    clk         => clk,

```

```

        set_phase_trigger_adustment => set_phase_trigger_adustment,
        phase_trigger_adjustment => phase_trigger_adjustment
    );
}

FRMSYN_1 : q4_Frame_sync
GENERIC MAP(
    Nbit      => Nbit,
    Hdrlen   => Hdrlen,
    Datalen  => Datalen,
    Framelen => Framelen,
    async_error_no => async_error_no,
    async_error_frames => async_error_frames,
    sync_error_no => sync_error_no,
    sync_error_frames => sync_error_frames
)
PORT MAP(
    fifo_out      => rd_data_sig,
    rd_en         => fifo_read_enable_sig,
    valid_header  => fifo_out_valid_sig,
    clk           => clk,
    valid_header  => hdr_out_valid_sig,
    hdr_error     => hdr_error_count_sig,
    header        => hdr_sig,
    hdr_bit       => hdr_bits_count_sig
);

FIFO_1 : fifo_qber
PORT MAP(
    clk          => clk,
    reset        => fifo_reset,
    detected_value => data_out_sig_vec,
    fifo_out     => rd_data_sig,
    write_enable => fifo_wr_en_sig,
    full         => full_sig,
    empty        => empty_sig,
    valid_header => fifo_out_valid_sig,
    read_enable  => fifo_read_enable_sig
);

Q3_divIP : q3_h_divvhdl_ip
PORT MAP(
    clk          => clk_in,
    hdr_error   => hdr_error_count_sig,
    hdr_bit     => hdr_bits_count_sig,
    n_v         => hdr_out_valid_sig,
    d_v         => hdr_out_valid_sig,
    qber        => BER_out,
    ans_v       => ber_valid
);
END Struct;

```

### 6.17.19 Open Issues

The implementation in FPGA with file reading is to be done.

## 6.18 Quantum Noise

|                     |   |   |
|---------------------|---|---|
| <b>Contributors</b> | : | Diamantino Silva, (2017-01-23 - ...)                                      |
|                     | : | Armando Pinto (2017-01-23 - ...)  |
| <b>Goal</b>         | : | Simulation and measurement of quantum noise in double homodyne detection. |

Quantum noise is an intrinsic property of light, a manifestation of the vacuum field fluctuations [1]. Contrarily to the majority of noise sources, that are overcomed by better equipment, quantum noise has a lower bound, given by the Heisenberg uncertainty principle, which cannot be broken. This property can be useful in some areas, such in quantum cryptography, were many protocols depend on it to ensure their security. The objective of this work is to develop a numerical model for the quantum noise in a homodyne detection and to validate the numerical model with experimental results.

$$\hat{a} |n\rangle = \sqrt{n} |n-1\rangle \quad (6.263), \quad \hat{a}^\dagger |n\rangle = \sqrt{n+1} |n+1\rangle \quad (6.264), \quad \hat{n} |n\rangle = n |n\rangle \quad (6.265)$$

### 6.18.1 Theoretical Analysis

In this section, we assume the following transmission system

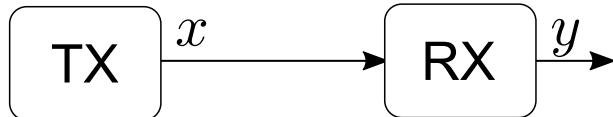


Figure 6.405: Model of a linear transmission system

We can approximate the output by the following relation [2]

$$y = tx + z \quad (6.266)$$

$y$  is the received signal,  $x$  is the transmitted signal,  $t = \sqrt{\eta T}$ , where  $\eta$  accounts for the detector efficiency and  $T$  accounts for the transmittance of the channel, and the total noise is modeled by the random variable  $z$ . We are also going to assume that the random variable  $z$  is normal distributed with mean 0 and variance  $\sigma_z^2 = \eta T \xi + N_0 + v_{el}$  which is the superposition of all noise sources assumed to be independent from each other and independent from the signal  $x$ . These sources are the shot noise characterized by a gaussian variable with mean 0 and variance  $N_0$ , the detector's electronic noise characterized by a gaussian variable with mean 0 and variance  $v_{el}$  and the excess noise, which is the equivalent noise added in the channel, accounted at the transmitter output and characterized by a gaussian variable with

mean 0 and variance  $\xi$ . We are going to assume that  $x$  is a gaussian distributed random variable with mean 0 and variance  $\sigma_x^2$ . With the previous assumptions we have

$$\langle x \rangle = \langle z \rangle = \langle y \rangle = 0 \quad (6.267)$$

with variances (and covariances)

$$\begin{aligned} \langle x^2 \rangle &= \sigma_x^2 \\ \langle xy \rangle &= t \langle x^2 \rangle + \langle xz \rangle = \sqrt{\eta T} \sigma_x^2 \\ \langle y^2 \rangle &= t^2 \langle x^2 \rangle + 2t \langle xz \rangle + \langle z^2 \rangle = \eta T \sigma_x^2 + \eta T \xi + N_0 + v_{el} \end{aligned} \quad (6.268)$$

in which  $\langle xz \rangle = 0$ , given that  $x$  and  $z$  are defined as independent variables.

### 6.18.2 Numerical Analysis

To test the previous model, we simulate an optical communication system. Diagram 6.406 shows the block setup used in the simulation

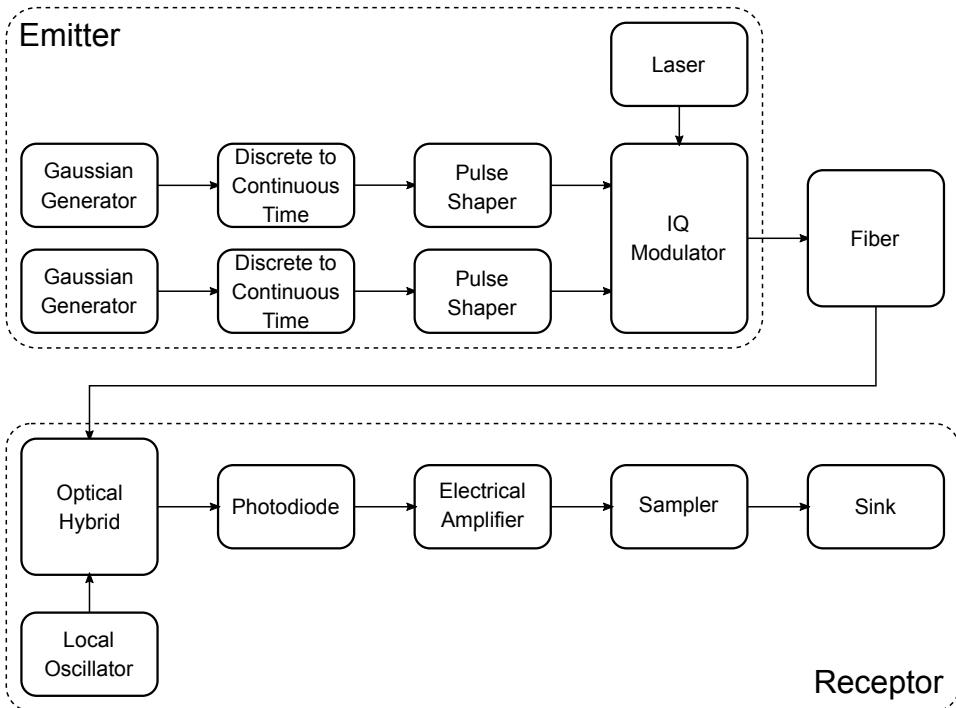


Figure 6.406: Block diagram of the simulation

The simulation starts by generating two random numbers  $x_I$  and  $x_Q$ , both following gaussian distributions with the same mean 0 and same variance  $\sigma_x^2$ . An optical signal with I and Q modulation is generated, accordantly with  $x_I$  and  $x_Q$ , and sent through the fiber to

the receptor. The receptor will perform a double balanced homodyne detection, outputing the following two currents proportional to each quadrature [3]

$$i_I(t) = R\sqrt{P_S P_{LO}} \cos [\theta_S(t) - \theta_{LO}(t)] \quad i_Q(t) = R\sqrt{P_S P_{LO}} \sin [\theta_S(t) - \theta_{LO}(t)] \quad (6.269)$$

in which  $i_I$  is the current proportional to the in-phase component,  $i_Q$  is the current proportional to the quadrature component,  $P_S$  is the signal's instantaneous power,  $P_{LO}$  is the local oscillator instantaneous power,  $R = \eta e/\hbar\omega$  is the detector responsivity,  $\theta_S$  is the signal phase and  $\theta_{LO}$  is the local oscillator phase.

Assuming a local oscillator power much higher than the signal power, the expected variance for both currents is [4]

$$(\Delta i)^2 = \frac{1}{2} R^2 P_\lambda [4\mu^2 P_{LO} N_0 + P_{LO}\mu(1-\mu)] \quad (6.270)$$

in which  $P_\lambda = \frac{hc}{\lambda\Delta t}$  and  $\Delta t$  is the sample duration. For continuous wave coherent states,  $N_0 = \frac{1}{4}$ .

In our simulation, the input of a photodiode is a sequence of samples, corresponding to the mean amplitude of the electric field during the sample duration. Using this input, we can obtain the incident average power in the photodiode with  $\bar{P}(t) = |A(t)|^2$  which corresponds to the average number of photon in sample duration  $\bar{n} = \frac{\bar{P}(t)}{P_\lambda}$ . To model the shot noise present in a photodiode detection, we will assume that the number of photons detected during a given interval,  $n(t)$ , is a random variable following a Poisson distribution with mean  $\bar{n}$ . The output current of the photodiode will be  $i(t) = RP_\lambda n(t)$ .

The homodyne detection consists in three fundamental steps: mixing the signal with a local oscillator in a balanced beam splitter, converting each of the two output beams into a current and finally subtracting one current from the other. In our simulation, we simulate a double balanced homodyne detector, which is composed by two balanced homodyne detectors, each one for measuring one of the signals quadrature. To keep our analysis simple, we will focus on the in-phase homodyne detector, which have the following electric fields incident to each of it's photodides Assuming that the photocurrents of the two photodiodes are stochastically independent, then:

$$(\Delta i(t))^2 = R^2 P_\lambda^2 ((\Delta n_1(t))^2 + (\Delta n_2(t))^2) = \frac{1}{2} R^2 P_\lambda^2 (\bar{n}_S + \bar{n}_{LO}) \quad (6.271)$$

These two results are in agreement with the previous obtained results 6.269 and 6.270 if  $\mu = 1$ .

Before constructing a complete continuous variable transmission system we started by simulating a simplified setup, by removing all sources of noise, to test the generation of quantum noise in the photodiodes.

The constelation was simplified to one single state  $|\sqrt{\bar{n}_S/2} + i\sqrt{\bar{n}_S/2}\rangle$ , the fiber has no loss and no noise and all the electronic elements had their noises turned off.

The simulations has two main input parameters  $\bar{n}_S$  and  $\bar{n}_{LO}$ . The following plots show

the current variance  $(\Delta i(t))^2$  in shot noise units, in function of the input signal number of photons per sample, for the  $I$  and  $Q$  quadratures. Each curve corresponds to a local oscillator number of photons per sample.

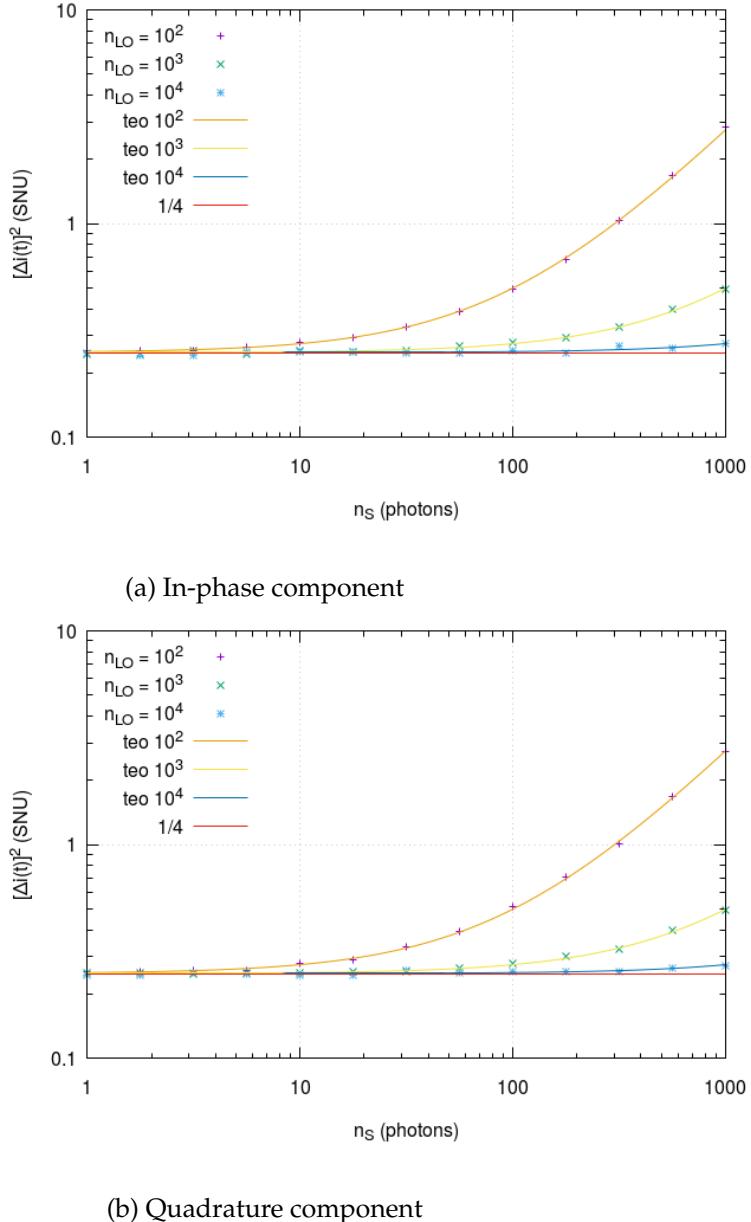


Figure 6.407: Current variance in function of the signal  $\bar{n}_S$  and the local oscillator  $\bar{n}_{LO}$  number of photons.

For each pair  $(\bar{n}_S, \bar{n}_{LO})$ , 10000 signals were used to simulate the transmission. We see that the simulation results follow the theoretical prediction. Also, for higher  $\bar{n}_{LO}$  values, the variance becomes independent of the signal's number of photons, converging to the theoretical value  $\frac{1}{4}$ .

### 6.18.3 Experimental Analysis

#### 6.18.3.1 Measurement of quantum noise

The different noise sources that compose the total noise, scale differently in function of the local oscillator power. Electronic noise does not scale at all, quantum noise scales with the root square and classical noise scales linearly.

## References

- [1] Mark Fox. *Quantum Optics: An Introduction*. Oxford University Press, 2006.
- [2] Tao Wang et al. "Practical performance of real-time shot-noise measurement in continuous-variable quantum key distribution". In: *Quantum Information Processing* 17.1 (2018), p. 11.
- [3] Govind P Agrawal. *Fiber-optic communication systems*. "Third". John Wiley & Sons, 2002.
- [4] Rodney Loudon. *The Quantum Theory of Light*. Oxford University Press, 2000.

## 6.19 Frequency and Phase Recovery in CV-QC Systems

|                      |   |
|----------------------|---|
| <b>Student Name</b>  | : Daniel Pereira (01/05/2017 - )  |
| <b>Goal</b>          | : Theoretical and simulation study of techniques for frequency and phase recovery in CV-QC systems.         |
| <b>Directory</b>     | : sdf/cv_system   |
| <b>Related Links</b> | : <a href="https://www.overleaf.com/14348245sjvzbkzpxwxf">https://www.overleaf.com/14348245sjvzbkzpxwxf</a> |

Continuous Variable Quantum Communications (CV-QC) can encode information in the phase of weak coherent states. The weak nature of these states makes frequency and phase noise a substantial impairment, as such it is important to implement optimal techniques for their compensation.

### 6.19.1 Classical Frequency and Phase Recovery - State of the art

The traditional method for compensating noise in classical coherent communications is to use some form of phase-locked loop (PLL), synchronizing the frequency and the phase of the local oscillator (LO) with that of the transmitter laser [1]. Alternatively, frequency offset compensation and phase recovery can be performed in the digital domain by applying digital signal processing (DSP) [2]. It has been shown that DSP is more tolerant to laser phase noise than PLL based techniques [1]. This work will be focused on the study of DSP techniques for Continuous Variables Quantum Communications (CV-QCs) systems.

#### 6.19.1.1 Frequency Mismatch Compensation Techniques

Frequency mismatch occurs when the central frequencies of the transmission and reception local oscillators aren't equal, as illustrated in Figure 6.408. A frequency offset of  $\Delta f = f_{\text{LO}_{Tx}} - f_{\text{LO}_{Rx}}$  introduces a phase of  $\omega_I T_n$  to the  $n$ th symbol relative to the 0th symbol, where

$$\omega_I = 2\pi \frac{f_{\text{LO}_{Tx}} - f_{\text{LO}_{Rx}}}{2} \quad (6.272)$$

is the intermediate frequency.

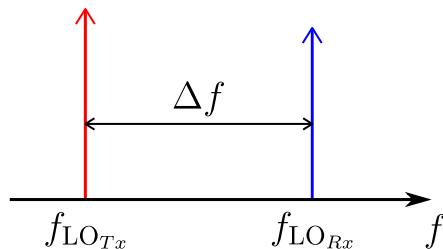


Figure 6.408: Visual representation of an optical frequency offset.

In order to show the action of the various frequency mismatch compensation techniques we will assume that an optical communication system has resulted in a signal given by

$$x(n) = x_{\text{sym}}(n)e^{i(\omega_I T n + \Delta\theta(n))}, \quad (6.273)$$

where  $x_{\text{sym}}(n)$  is the complex modulated signal,  $\Delta\theta(n)$  is the phase noise contribution and  $T$  is the symbol period. Compensation is accomplished by first estimating  $\omega_I$ , yielding  $\omega_{\text{est}}$ , and then removing the added phase for each symbol via [2]

$$\tilde{x}(n) = x(n)e^{-i\omega_{\text{est}}Tn}, \quad (6.274)$$

Frequency estimation can be accomplished through a blind frequency search method [3], in which the frequency is scanned over a predetermined range, symbol decisions made and the minimum square error used as the frequency-selection criteria. Alternatively the frequency can be estimated by evaluating the spectrum of the  $m$ th power of the input signal [4], which exhibits a peak at the frequency  $m\omega_I$ . A third option involves the usage of training symbols inserted periodically with the data payload [5]. These training symbols are used to remove the applied phase modulation. The resulting signal  $s(n)$  is used to obtain an estimate of the offset frequency as

$$\omega_{\text{est}} = \frac{1}{T} \arg \left\{ \sum_{n=1}^L s(n)s^*(n-1) \right\} \quad (6.275)$$

### 6.19.1.2 Phase Mismatch Compensation Techniques

#### Deterministic Phase Mismatch

Phase mismatch occurs when the two employed oscillators have differing optical phases, as illustrated in Figure 6.409. This causes the phase modulation applied to one of the signals to be misread in the detection scheme.

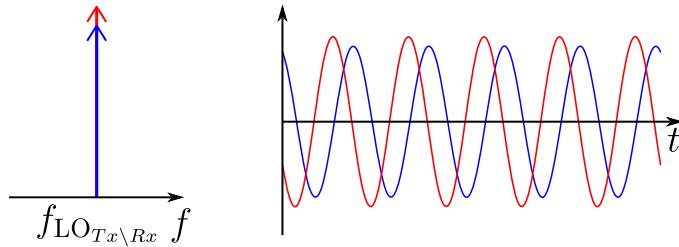


Figure 6.409: Frequency (left) and time (right) domain representation of a phase mismatch between two cosines of equal frequency.

#### Phase Noise

Phase noise arises from the non-zero linewidth of the transmission and reception local

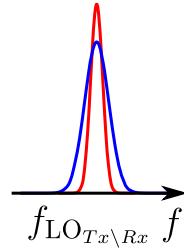


Figure 6.410: Phase noise arises from the non-zero linewidth of the transmission and reception oscillators, being present even if their central frequencies match.

oscillators, as illustrated in Figure 6.410. Phase noise consists of a time evolving phase mismatch, which can be modelled as a Weiner process described as

$$\phi(t_k) = \phi(t_{k-1}) + \Delta\phi(k), \quad (6.276)$$

where  $\Delta\phi(k)$  are phase increments, which are independent and identically distributed random Gaussian variables with zero mean and variance

$$\sigma^2 = 2\pi\Delta\nu|t_k - t_{k-1}|, \quad (6.277)$$

where  $\Delta\nu$  is the laser linewidth and  $t_{k/k-1}$  are the sampling instants.

In order to show the action of the various phase recovery techniques we assume that a perfect frequency mismatch compensation has been accomplished for the signal (6.273), having resulted in

$$x(n) = x_{\text{sym}}(n)e^{i\Delta\theta(n)}, \quad (6.278)$$

where, again,  $x_{\text{sym}}(n)$  is the complex, phase-modulated signal and  $\Delta\theta(n)$  is the phase noise contribution.

The phase mismatch can be compensated either through blind estimation [6] or through pilot-aided estimation [7]. In the blind estimation method a  $m$ th-order non-linearity is used to remove the phase modulation, yielding an estimate of the phase mismatch [2]. The phase estimation is given by [6]

$$\Delta\theta_{\text{est}}(n) = \frac{1}{m} \arg \left\{ \frac{1}{2L+1} \sum_{l=-L}^L f(l) x^m(n+l) \right\}, \quad (6.279)$$

where  $f(l)$  is a weighting function. The  $m$ th power removes the phase modulation, thus the only remaining phase is due to the phase noise contribution. The sum  $-\frac{1}{2L+1} \sum_{l=-L}^L f(l) e^{i\Delta\theta(n+l)}$  functions then as a weighted average of the phase noise contributions up to  $L$  symbols before and  $L$  symbols after the current one.

In the pilot aided scheme, pre-agreed on symbols are inserted, time multiplexed, with the data payload at a pilot rate of  $1/P$  (meaning one pilot symbol is inserted after  $P - 1$  data symbols). The phase mismatch is determined from the pilot symbols and this estimation is used to compensate the phase mismatch on the data carrying symbols [7].

A alternative blind estimation scheme is proposed in [8], taking advantage of the unmodulated  $x^m$  signal to perform both frequency and phase mismatch compensation. The phase added by both the frequency and the phase mismatch is estimated as

$$\Delta\theta_{\text{est}}(n) = \Delta\theta_{\text{est}}(n-1) + \mu \arg \left\{ \frac{1}{L} \sum_{l=0}^{L-1} x^m(n-l)[x^m(n-l-1)]^* \right\}, \quad (6.280)$$

for every symbol except for the first/last  $L$  symbols, where  $\mu$  corresponds to the integral gain. The frequency and phase mismatch compensation is accomplished by taking the uncompensated signal (6.273) and performing

$$x_{\text{rec}}(n) = x(n)e^{-i\sum_{k=L}^n \Delta\theta_{\text{est}}(k)T_s} \quad (6.281)$$

### 6.19.2 Quantum Frequency and Phase Recovery - State of the art

Due to the low amplitude of the quantum coherent states employed in CV-QC, any auxiliary processes, such as frequency and phase mismatch compensation, need to be employed in a non-intrusive manner. Another consequence of the low amplitude quantum coherent states is the necessity for high sensitivity, low thermal noise coherent receivers [9]. This requirement limits the bandwidth of the receivers [10, 11], which in turn limits the maximum operational baud rate of CV-QC systems. To avoid the added noise from disparities between the transmission and reception LO lasers, the first CV-QC protocols used a high intensity signal sent polarization multiplexed with the quantum signal, to be used as the LO in the detection scheme [12, 13]. This co-propagating technique presents a security risk, with attacks proposed that tamper with the LO's wavelength [14] or the shape of its pulses [15, 16, 17]. Locally generated LO (LLO) CV-QC protocols were proposed simultaneously in [18] and [19]. The usage of two different laser sources for the signal and the LO necessitates the application of frequency and phase recovery techniques.

#### 6.19.2.1 Frequency Mismatch Compensation Techniques

In [18, 19, 20] the frequency mismatch was assumed to be minimal, accomplished by employing high quality tunable laser sources. The resulting noise introduced in this situation is treated as extra phase mismatch.

Given the low amplitude of the quantum pulses, frequency offset compensation methods cannot be applied directly to the quantum pulses [19]. To overcome this, a high intensity reference can be shared, with the application of both blind frequency search and frequency domain methods to this high intensity reference having been proposed [21].

#### 6.19.2.2 Phase Mismatch Compensation Techniques

Given their low intensity, phase recovery methods cannot be applied directly to the quantum pulses [19]. In [18, 19] phase recovery is based on a pilot aided estimation scheme, with the pilot signal consisting of high intensity reference pulses and low intensity quantum

pulses carved sequentially from the continuously running transmission local oscillator. This scheme is dubbed LLO-sequential.

In [20] two LLO CV-QC schemes were proposed. The first one is similar to the pilot assisted phase recovery proposed in [18, 19], with the difference being in that the reference pulses are extracted from the same wavefront as the quantum pulses and time multiplexed with the latter by using a delayline, for this reason being dubbed LLO-delayline. In this setup the reception local oscillator is also pulsed and subjected to the delayline scheme. The second scheme proposed in [20] consisted of displacing the modulated constellation in phase space by a constant amplitude and a phase determined by the instantaneous phase of the transmission laser. This last scheme is dubbed LLO-displacement.

In [22] an alternative solution is presented in which the pilot signal is prepared from a carrier-suppressed optical single-sideband and sent polarization and frequency multiplexed with the quantum signal.

### 6.19.3 Open issues

- Available frequency and phase recovery DSP techniques for classical systems do not function well at low baud rates. Techniques that work at baud rates of the order of a few MBd are required.
- Available techniques do not function well for high frequency mismatches and low baud-rates. Robust techniques capable of compensating for large offset frequencies are necessary.
- Available CV-QC DSP techniques do not function well for large linewidths and low baud-rates. Techniques more capable of working at high phase noise levels are required.

### 6.19.4 Novelty

A novel frequency mismatch compensation technique is proposed which can operate at low baud rates, is resistant to large frequency deviations and large laser linewidths. In theoretical tests, our technique performs well in the presence of strong phase noise, yielding EVM results close to perfect frequency mismatch compensation (that is, compensation using the actual frequency mismatch value) for symbol rates as low as 1 kHz and for frequency mismatches as high as  $10^9$  Hz. The other studied techniques fail for symbol rates below 1 GHz and for frequency mismatches above  $10^7$  Hz.

### 6.19.5 Novel Frequency Mismatch Compensation Technique

Our novel frequency mismatch compensation technique 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 LLO CV-QC scheme is presented in Figure 6.411. At the receiver stage, after coherent detection, the signal is described by

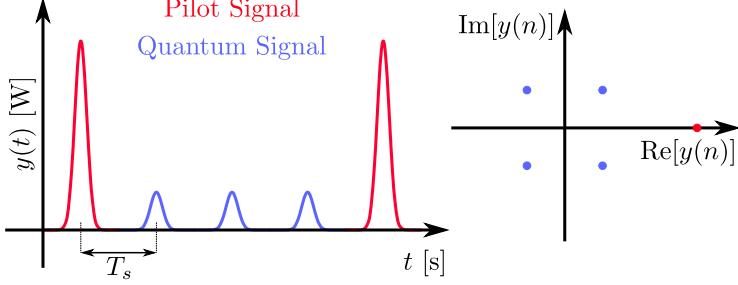


Figure 6.411: Time dependence (left) and constellation (right) at the output of the modulation stage of a pilot-assisted LLO CV-QC. Pilot and signal pulses/points identified by color. Pilot rate in the time dependence image is meant only as illustrative, actual pilot rate may be higher or lower.

$$x(t) = \begin{cases} x_q(t) = |x_q| \Pi(t) e^{i(\omega_I t + \phi(t) + \phi_0 - \varphi(t) - \varphi_0 + \theta(t))}, & n \neq jP \\ x_p(t) = |x_p| \Pi(t) e^{i(\omega_I t + \phi(t) + \phi_0 - \varphi(t) - \varphi_0)}, & n = jP \end{cases}, \quad n, j \in \mathbb{N}, \quad (6.282)$$

where  $x_q(t)/x_p(t)$  is the time continuous quantum/pilot signal,  $P$  is the pilot rate,  $\omega_I$  is the offset frequency,  $\phi(t)/\varphi(t)$  is the instantaneous phase of the transmitter/receiver laser and  $\theta(t)$  is the phase modulation at the sampling instant  $t$  and  $\phi_0/\varphi_0$  is the initial phase of the transmitter/receiver laser.  $\Pi(t) = \sum M(t + nT_s)$  describes the time continuous amplitude modulation applied to both the quantum and pilot signals, while  $M(t)$  describes the shape of each individual pulse. Sampling signal (6.282), assuming  $\Pi(nT_s) = 1$  for any  $n$ , results in

$$x(n) = x(t = nT_s) = \begin{cases} x_q(n) = |x_q| e^{i(\omega_I nT_s + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0 + \theta(nT_s))}, & n \neq jP \\ x_p(n) = |x_p| e^{i(\omega_I nT_s + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0)}, & n = jP \end{cases}, \quad n, j \in \mathbb{N}, \quad (6.283)$$

where  $x_q(n)/x_p(n)$  corresponds to the quantum/pilot constellation. The relative phase between the pilot and quantum signal constellations is known, so the pilot constellation can be used as a reference point from which to recover the phase encoded in the quantum signal without having to recover the absolute phase difference between the transmission and reception LOs.

Our novel technique functions by first multiplying an incremented pilot constellation by the complex conjugate of the unincremented pilot constellation yielding

$$x_{\text{freq}}(n = p(n + P)x_p^*(n)) = |x_p|^2 e^{i(\omega_I PT_s + \phi((n+P)T_s) - \phi(nT_s) - (\varphi((n+P)T_s) - \varphi(nT_s)))}, \quad (6.284)$$

with the frequency estimation obtained by taking the expected value of the complex

argument of  $x_{\text{freq}}(n)$  and dividing it by  $PT_s$ . Explicitly

$$\begin{aligned}\omega_{\text{est}} &= E \left[ \frac{1}{PT_s} \arg(x_{\text{freq}}(n)) \right] \\ &= E \left[ \omega_I + \frac{\phi((n+P)T_s) - \phi(nT_s) - (\varphi((n+P)T_s) - \varphi(nT_s)) + 2k\pi}{PT_s} \right] \\ &= \omega_I + k\pi(PT_s)^{-1}, \quad n = jP, \quad n, j \in \mathbb{N},\end{aligned}\quad (6.285)$$

where the  $k\pi(PT_s)^{-1}$  term is due to the non-injectiveness of the  $\arg$  function, forcing the value of  $\omega_{\text{est}}PT_s$  to remain within the  $[0, 2\pi]$  interval. The value of  $k$  is bounded by

$$0 \leq \omega_{\text{est}}PT_s < 2\pi \iff -\frac{\omega_I PT_s}{2\pi} \leq k < -\frac{\omega_I PT_s}{2\pi} + 1, \quad k \in \mathbb{Z}. \quad (6.286)$$

$\frac{\phi((n+P)T_s) - \phi(nT_s) - (\varphi((n+P)T_s) - \varphi(nT_s))}{PT_s}$  is a normally distributed random variable with null mean and variance given by

$$\text{Var} \left[ \frac{\phi((n+P)T_s) - \phi(nT_s) - (\varphi((n+P)T_s) - \varphi(nT_s))}{PT_s} \right] = \frac{4\pi\Delta\nu PT_s}{(PT_s)^2} = 4\pi\Delta\nu(PT_s)^{-1}. \quad (6.287)$$

The frequency mismatch compensated signals are obtained by multiplying the constellations in (6.283) by  $e^{-i\omega_{\text{est}}t_n}$ , resulting in

$$\begin{aligned}\bar{x}(n) &= \begin{cases} \bar{x}_q(n) = |x_q| e^{i(-nk\pi P^{-1} + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0 + \theta(nT_s))}, & n \neq jP \\ \bar{x}_p(n) = |x_p| e^{i(-nk\pi P^{-1} + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0)}, & n = jP \end{cases}, \\ &\quad n, j \in \mathbb{N} \quad -\frac{\omega_I PT_s}{2\pi} \leq k \leq -\frac{\omega_I PT_s}{2\pi} + 1.\end{aligned}\quad (6.288)$$

The phase modulated signal can then be recovered by estimating the instantaneous phase difference using the pilot signal, this step depends on the applied LLO scheme.

The exactness of  $\omega_{\text{est}}$  can be evaluated through a confidence interval. Assuming that the variance is not well known, being estimated from the experimental results, the confidence interval is given by

$$P \left( \omega_I - t \frac{s}{\sqrt{n}} < \omega_{\text{est}} < \omega_I + t \frac{s}{\sqrt{n}} \right) = 1 - \alpha, \quad (6.289)$$

where  $t$  is the  $1 - \frac{\alpha}{2}$ 'th percentile of the Student's t-distribution and  $s^2$  is the estimated value of the variance. For an offset frequency of  $\omega_I = 10$  MHz, assuming a pilot rate  $P = 2$  and that  $n = 2 \times 10^5$  samples were used, using a variance estimate given by  $s^2 = 2\pi\Delta\nu T_s^{-1}$ , the 99 % confidence interval is less than 1 % of  $\omega_I$ .

### 6.19.6 Implementation issues

The proposed compensation technique is compatible with all pilot assisted phase mismatch compensation techniques, with few differences observed for each case. In the interest of demonstrating both this compatibility and the distinctions between the different pilot assisted LLO techniques, we now proceed with a detailed study of multiple pilot assisted CV-QC methods.

### 6.19.6.1 Optimal amplitude of Pilot Signal

Given the sensitivity of CV-QC protocols to phase noise, the Pilot Signal's amplitude should be optimized in order to minimize the uncertainty of the measured instantaneous phase difference between the transmission and reception LOs. Assuming that the quadrature noise does not change meaningfully for an increasing amplitude, the phase uncertainty will drop as the Pilot signal's amplitude increases. This effect is explained graphically in Figure 6.412.

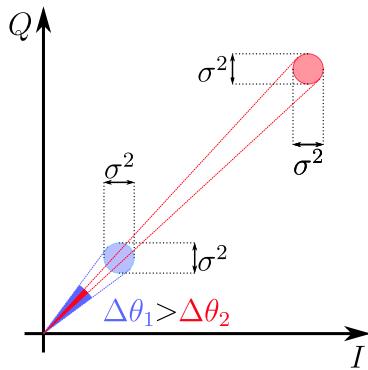


Figure 6.412: The phase uncertainty for Gaussian distributions with the same variance drops with an increase in amplitude.

Given the low intensity of the quantum signal, a very powerful LO is necessary in order to extract any phase information from it. Given this, the pilot signal, having an amplitude much larger than the quantum signal, can very easily saturate the output of the coherent receiver. The results of this study are presented in Figure 6.413. The values for the gain, thermal noise and saturation voltage were extracted from [23].

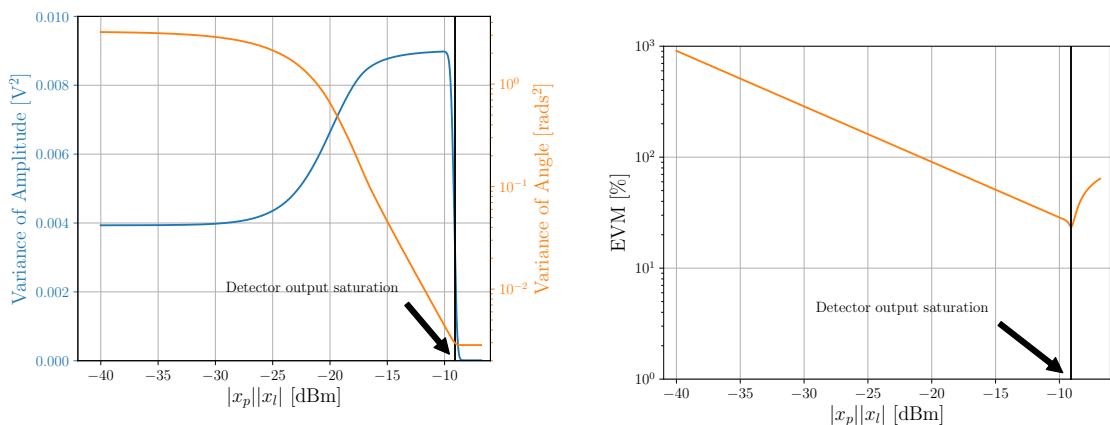


Figure 6.413: Evolution of the phase variance (left) and EVM (right) with the product of the amplitudes of the pilot signal  $|x_p|$  and LO  $|x_l|$ .

### 6.19.6.2 LLO-sequential

The initially proposed LLO schemes, a simplified diagram of which is presented in Figure 6.414, carved the pilot and signal pulses sequentially from the free running transmission local oscillator, thus being dubbed LLO-sequential. Coherent detection is performed with a similarly free running reception local oscillator. The phase difference between the transmission and reception laser for each signal symbol is estimated from an average of the instantaneous phase of the previous and ensuing pilot pulses.

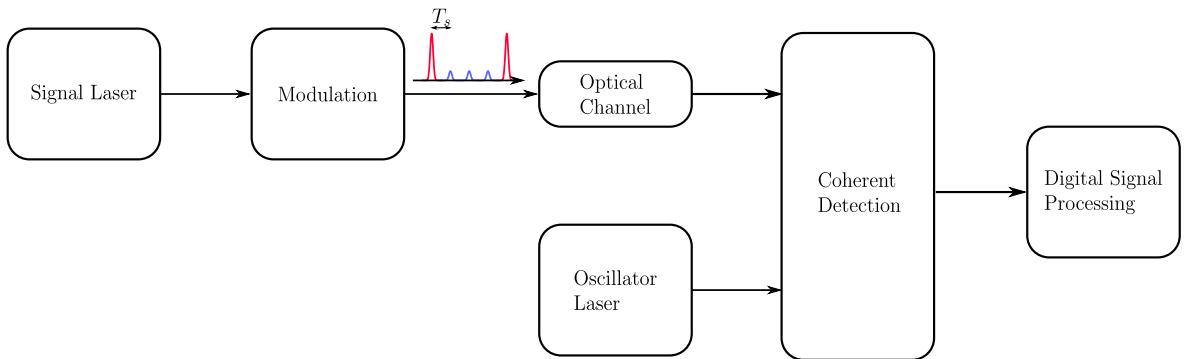


Figure 6.414: Simplified diagram of a LLO-sequential CV-QC scheme. First proposed in [18, 19]

Recalling the frequency mismatch compensated constellations (6.288), the phase mismatch of the  $n$ -th quantum signal constellation point is estimated from an average of the instantaneous phase of the previous and the following pilot signal pulses, yielding

$$\Theta_{\text{est}} = -n\pi k P^{-1} + \phi_0 - \varphi_0 + \frac{d[\phi((n+P-d)T_s) - \varphi((n+P-d)T_s)] + (P-d)[\phi((n-d)T_s) - \varphi((n-d)T_s)]}{P}, \quad (6.290)$$

and then removing this phase from the quantum signal constellation. The final recovered constellation is given by

$$\begin{aligned} \tilde{x}(n) &= \bar{x}_q(n)e^{-i\Theta_{\text{est}}} = \\ &= |x_q|e^{i\left(\frac{d[\phi(nT_s) - \varphi(nT_s) - \phi((n+P-d)T_s) + \varphi((n+P-d)T_s)] + (P-d)[\phi(nT_s) - \varphi(nT_s) - \phi((n-d)T_s) + \varphi((n-d)T_s)]}{P} + \theta(nT_s)\right)} \\ &= |x_q|e^{i(\Delta\Theta(nT_s) + \theta(nT_s))}, \quad n \neq jP, \quad n, j \in \mathbb{N}, \end{aligned} \quad (6.291)$$

where  $d$  is defined as the time distance between the quantum signal pulse and the previous pilot signal pulse, measured in number of symbol periods, and  $\Delta\Theta$  is a randomly distributed Gaussian variable with null mean and variance given by

$$\text{Var}[\Delta\Theta(nT_s)] = 2\pi T_s (\nu_T + \nu_R) \frac{d(P-d)}{P}, \quad (6.292)$$

where  $\Delta\nu_T$  and  $\Delta\nu_R$  are the linewidth of the transmission and reception local oscillators, respectively. From this result we see that the remaining variance will depend on the relation

between the distance to the previous pilot pulse  $d$  and the pilot rate  $P$  and that different values of  $d$  will have different remaining variances. In the LLO-sequential CV-QC scheme the uncertainty introduced by the non-injectiveness of the arg function has no effect in the recovered constellation.

#### 6.19.6.3 LLO-delayline

Building on the LLO-sequential method proposed in [18, 19], a scheme was proposed in which the pilot and quantum signal pulses are obtained from the same wavefront, employing a delayline scheme to accomplish the necessary time multiplexing. In the reception stage a similar method is used to obtain the LO reference pulses used in the detection of corresponding pilot and quantum signal pulses from the same wavefront. In this scheme the pilot rate is set to 2, so the frequency mismatch compensated

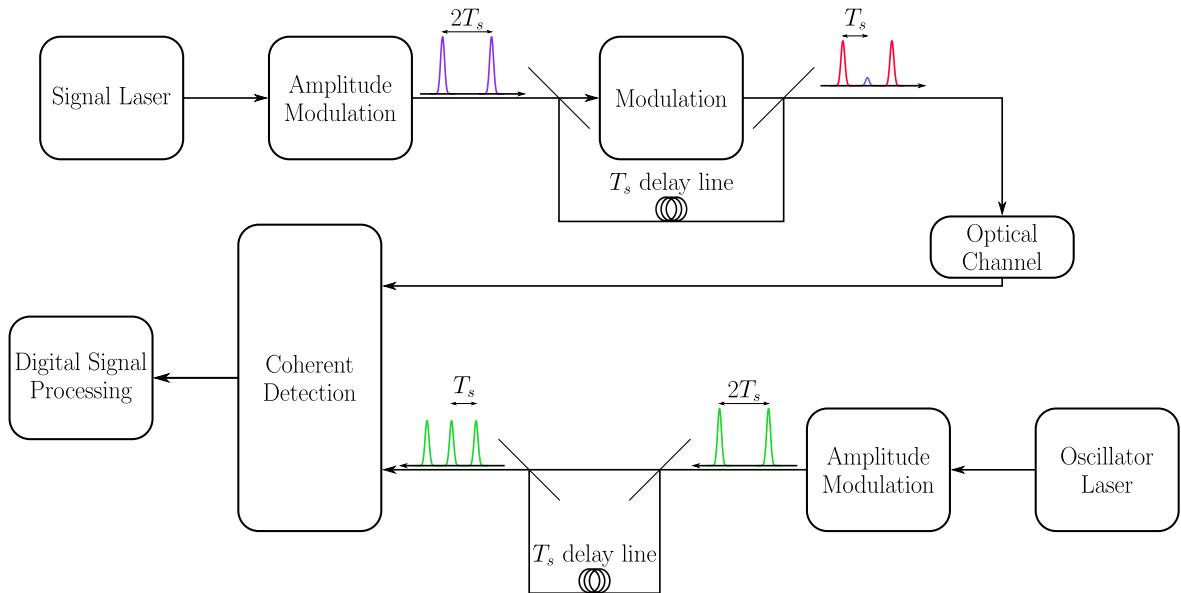


Figure 6.415: Simplified diagram of a partial LLO-delayline CV-QC scheme.

constellations (6.288) simplify to

$$\bar{x}(n) = \begin{cases} \bar{x}_q(n) = |x_q| e^{i(-nk\pi 2^{-1} + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0 + \theta(nT_s))}, & n \neq 2j \\ \bar{x}_p(n) = |x_p| e^{i(-nk\pi 2^{-1} + \phi((n+1)T_s) + \phi_0 - \varphi((n+1)T_s) - \varphi_0)}, & n = 2j \end{cases},$$

$$n, j \in \mathbb{N} \quad -\frac{\omega_I T_s}{\pi} \leq k \leq -\frac{\omega_I T_s}{\pi} + 1. \quad (6.293)$$

The phase mismatch of the  $n$ -th quantum signal constellation point is estimated from the previous pilot signal pulse, which yields

$$\Theta_{\text{est}} = -(n-1)\pi k 2^{-1} + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0. \quad (6.294)$$

The final recovered constellation is then given by

$$\begin{aligned}\tilde{x}(n) &= \bar{x}_q(n)e^{-i\Theta_{\text{est}}} \\ &= |x_q|e^{i(\theta(nT_s)-k\pi 2^{-1})}, \quad n \neq jP, \quad n, \quad j \in \mathbb{N} \quad -\frac{\omega_I T_s}{\pi} \leq k \leq -\frac{\omega_I T_s}{\pi} + 1.\end{aligned}\quad (6.295)$$

This scheme entirely removes the phase noise introduced by both lasers, apart from phase fluctuations that might occur in the delay lines. In the LLO-delayline CV-QC scheme the uncertainty introduced by the non-injectiveness of the arg function will add a phase of either  $\frac{\pi}{2}$ ,  $\pi$ ,  $\frac{3\pi}{2}$  or  $2\pi$ . To allow for this, the proceeding operations must be repeated in order to test all 4 options.

#### 6.19.6.4 Partial LLO-delayline

An alternate scheme is presented in Figure 6.416, being a combination of the transmission setup of the LLO-delayline with the reception setup of the LLO-sequential. This system removes the phase noise from the transmission laser, while accepting the phase noise of the reception laser. In this scheme the pilot rate is set to 2, so the frequency mismatch

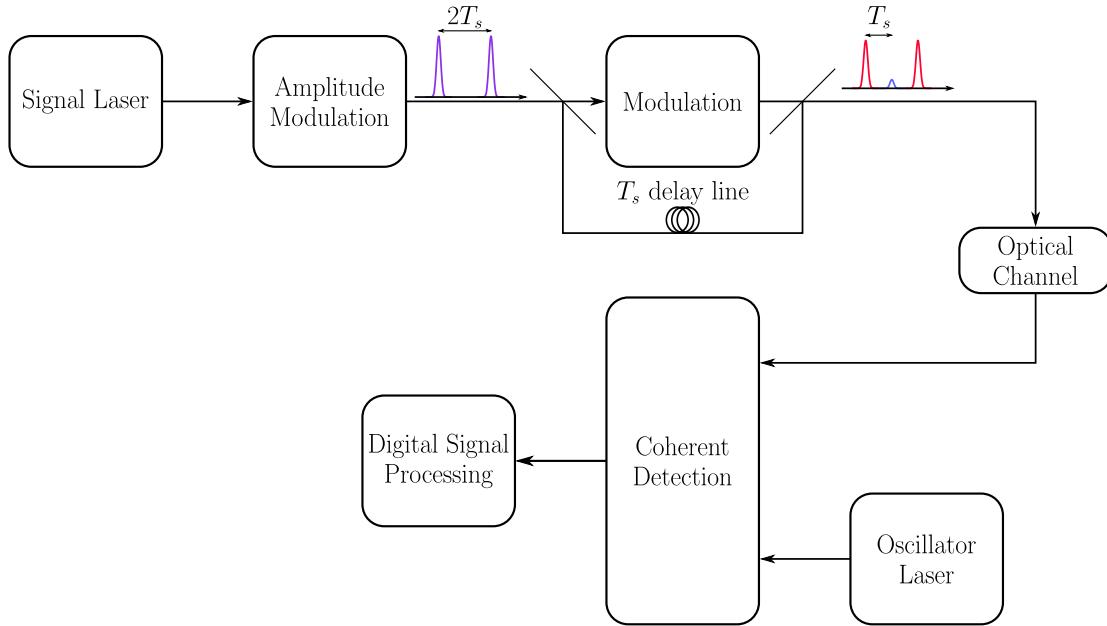


Figure 6.416: Simplified diagram of a partial LLO-delayline CV-QC scheme.

compensated constellations (6.288) simplify to

$$\begin{aligned}\bar{x}(n) &= \begin{cases} \bar{x}_q(n) = |x_q|e^{i(-nk\pi 2^{-1} + \phi(nT_s) + \phi_0 - \varphi(nT_s) - \varphi_0 + \theta(nT_s))}, & n \neq 2j \\ \bar{x}_p(n) = |x_p|e^{i(-nk\pi 2^{-1} + \phi((n+1)T_s) + \phi_0 - \varphi(nT_s) - \varphi_0)}, & n = 2j \end{cases}, \\ &n, \quad j \in \mathbb{N} \quad -\frac{\omega_I T_s}{\pi} \leq k \leq -\frac{\omega_I T_s}{\pi} + 1.\end{aligned}\quad (6.296)$$

The phase mismatch of the  $n$ -th quantum signal constellation point is estimated from the previous pilot signal pulse, which yields

$$\Theta_{\text{est}} = -(n-1)\pi k 2^{-1} + \phi(nT_s) + \phi_0 - \varphi((n-1)T_s) - \varphi_0. \quad (6.297)$$

The final recovered constellation is then given by

$$\begin{aligned} \tilde{x}(n) &= \bar{x}_q(n)e^{-i\Theta_{\text{est}}} \\ &= |x_q|e^{i(\varphi((n-1)T_s)-\varphi(nT_s)+\theta(nT_s)-k\pi 2^{-1})}, \quad n \neq jP, \quad n, \quad j \in \mathbb{N} \end{aligned} \quad -\frac{\omega_I T_s}{\pi} \leq k \leq -\frac{\omega_I T_s}{\pi} + 1, \quad (6.298)$$

The phase noise introduced by the reception laser will have a variance of

$$\sigma^2 = 2\pi\Delta\nu_R T_s, \quad (6.299)$$

where  $\Delta\nu_R$  is the linewidth of the reception local oscillator and  $T_s$  is the symbol period. As in the LLO-delayline CV-QC scheme, the uncertainty introduced by the non-injectiveness of the arg function will add a phase of either  $\frac{\pi}{2}$ ,  $\pi$ ,  $\frac{3\pi}{2}$  or  $2\pi$ . To allow for this, the proceeding operations must be repeated in order to test all 4 options.

### 6.19.7 Error Vector Magnitude

The figure of merit employed for the comparisons presented in this document will be the Error Vector Magnitude (EVM), where the error is evaluated as the relation between the magnitude of the error vector  $e_V$  and the magnitude of the vector of the ideal symbol position  $\text{ref}_V$

$$EVM(\%) = 100 \sqrt{\frac{|e_V|}{|\text{ref}_V|}} = 100 \sqrt{\frac{|\text{m}_V - \text{ref}_V|}{|\text{ref}_V|}} \quad (6.300)$$

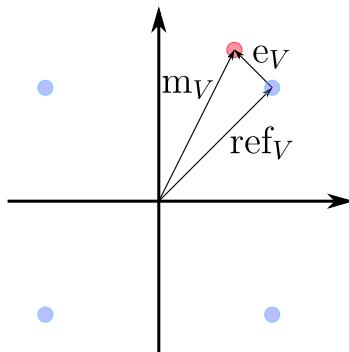


Figure 6.417: Visual representation of the EVM method in a QPSK constellation. The ideal constellation points are presented in blue, while an example for an actual measured symbol is presented in red.

### 6.19.8 Comparative analysis of the frequency ranging techniques

Three of the frequency mismatch compensation techniques described in Section 6.19.1 were applied on simulated signal and reference constellations, similar to the ones presented in (6.283). The blind frequency search method was applied to the pilot signal following the method described in [3], utilizing a coarse step size of 10 MHz followed by a fine step size of 1 MHz. The spectral analysis method was applied via evaluation of the spectrum of the reference constellation. The alternative blind method was applied as described in Section 6.19.1.

The EVM of the recovered constellation (6.288) was computed in function of the symbol frequency. The results for this study are presented in Figure 6.418. Only our novel method

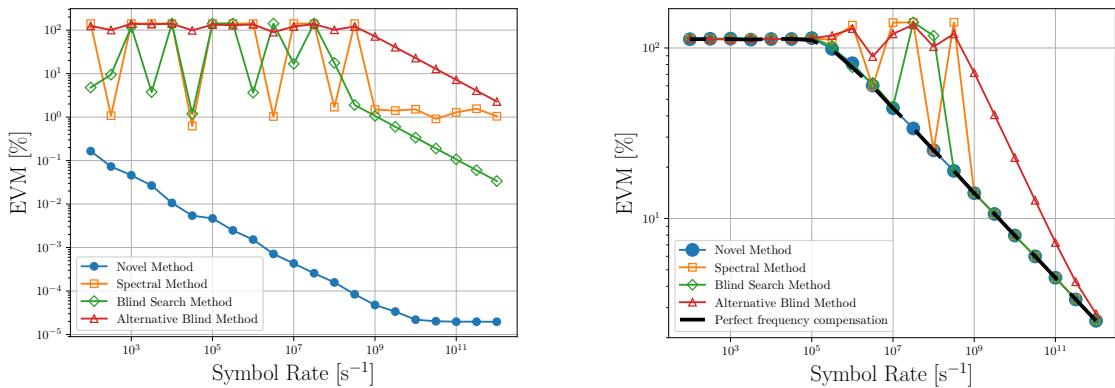


Figure 6.418: EVM in function of the sampling frequency for 4 different frequency ranging techniques without (left) and with (right) phase noise, in the latter case a laser linewidth of 100 kHz was assumed and the partial LLO-delayline phase recovery scheme was applied in order to allow for an easier EVM comparison. In the comparison figure with phase noise, the EVM observed without any frequency mismatch is included as black dashes to indicate the best case scenario. A frequency mismatch of 519213710 Hz (value chosen at random) was assumed.

was able to compensate for the frequency mismatch at the lower sampling frequencies attempted (note the swift increase in EVM for sampling frequencies below  $10^9$  Hz for the other methods in Figure 6.418-left). In the study without phase noise, for all the tested sampling frequencies, our novel method returns consistently EVM values 4 orders of magnitude below the classical alternatives, thus delivering results of much greater precision. From the study with phase noise we see that our novel method is very resistant to high levels of phase noise, performing very close to perfect frequency mismatch compensation for almost all the symbol rates tested.

A comparison of the EVM performance, in function of the frequency mismatch between the two employed lasers, between our novel frequency compensation scheme and the alternative blind frequency search scheme is presented in Figure 6.419. The alternative blind frequency search method was chosen for this comparison because it allows for a cleaner,

more readable figure. In both situations (with and without phase noise), our novel method

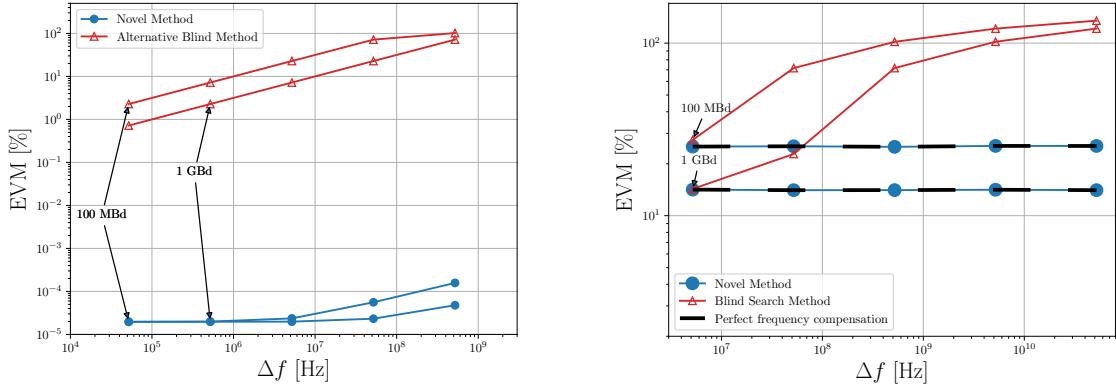


Figure 6.419: EVM in function of the frequency mismatch for 2 different frequency ranging techniques (stars correspond to our novel frequency ranging method and triangles to the alternative blind frequency search technique) and 2 different symbol rates (identified in both figures), without (left) and with (right) phase noise, in the later case a laser linewidth of 100 kHz was assumed and the partial LLO-delayline phase recovery scheme was applied in order to allow for an easier EVM comparison. In the comparison figure with phase noise, the EVM observed without any frequency mismatch is included as black dashes to indicate the best case scenario.

returns results orders of magnitude more precise than the alternative. As observed before, in the situation with phase noise our method performs very close to perfect frequency mismatch compensation, even for very high frequency mismatches.

Our novel technique has a computational complexity of  $O(n)$ . In comparison, the blind frequency estimation technique requires carrier phase recovery, bit decision and computing of the least mean square error for every test frequency, even assuming a frequency and phase recovery and decision algorithm with a total computational complexity of  $O(n)$ , the added least mean square error step will push the computational complexity above the one observed in our novel technique. The spectrum aided technique requires a Fourier Transform to be performed, which in itself has a computational complexity of  $O(n \log(n))$ , this is above the computational complexity of our novel technique for any  $n$  greater than  $e$ .

### 6.19.9 Simulation Analysis

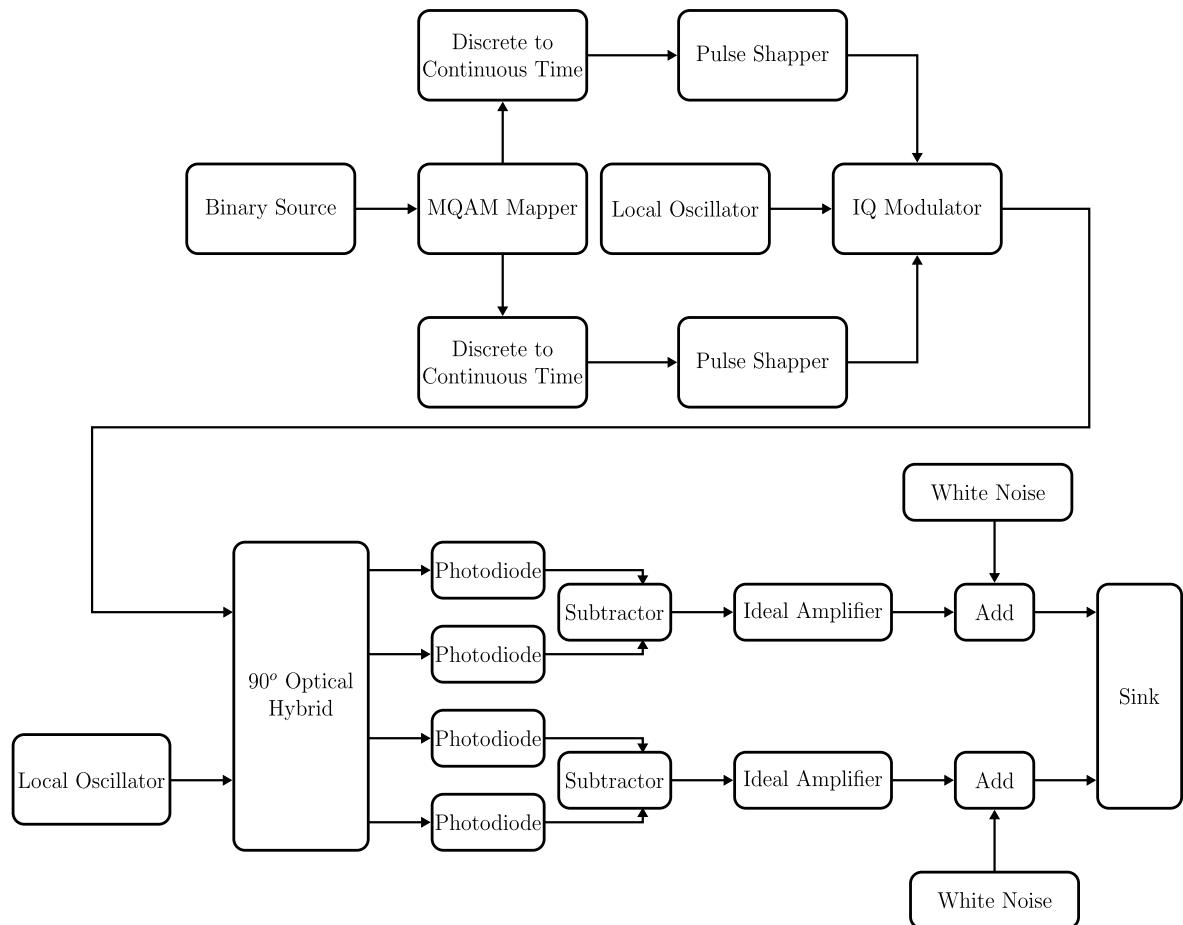


Figure 6.420: Simulation setup for the LLO-sequential CV-QC transmission system employed in testing our novel frequency mismatch compensation system.

| Simulation Blocks           |           |
|-----------------------------|-----------|
| Necessary                   | Available |
| Binary Source               | ✓         |
| MQAM Mapper                 | ✓         |
| Pulse Shaper                | ✓         |
| Discrete to Continuous Time | ✓         |
| Local Oscillator            | ✓         |
| IQ Modulator                | ✓         |
| Attenuator                  |           |
| 90° Optical Hybrid          | ✓         |
| Photodiode                  | ✓         |
| Subtractor                  |           |
| Ideal Amplifier             | ✓         |
| Add                         | ✓         |
| White Noise                 | ✓         |
| Sink                        | ✓         |

### 6.19.10 Simulation Results

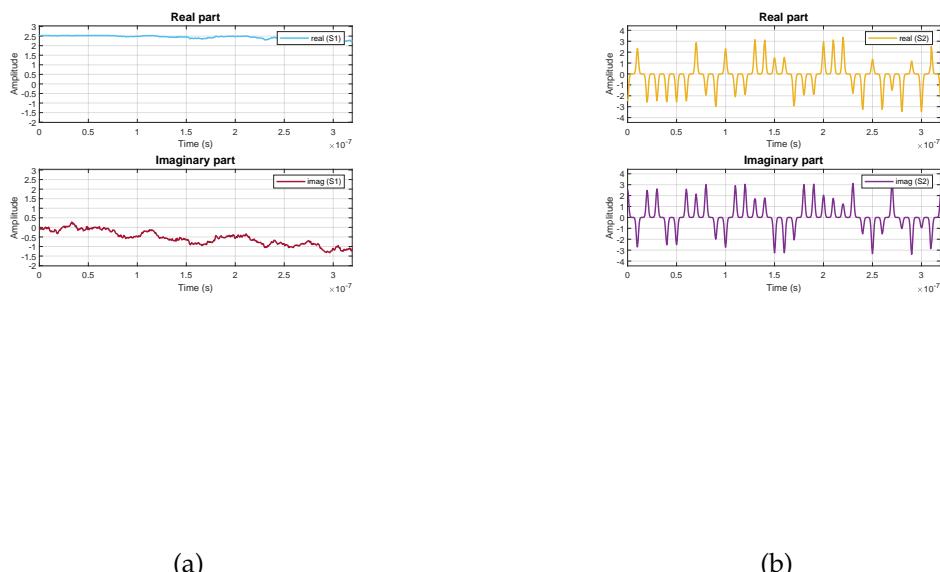


Figure 6.421

### 6.19.11 New study

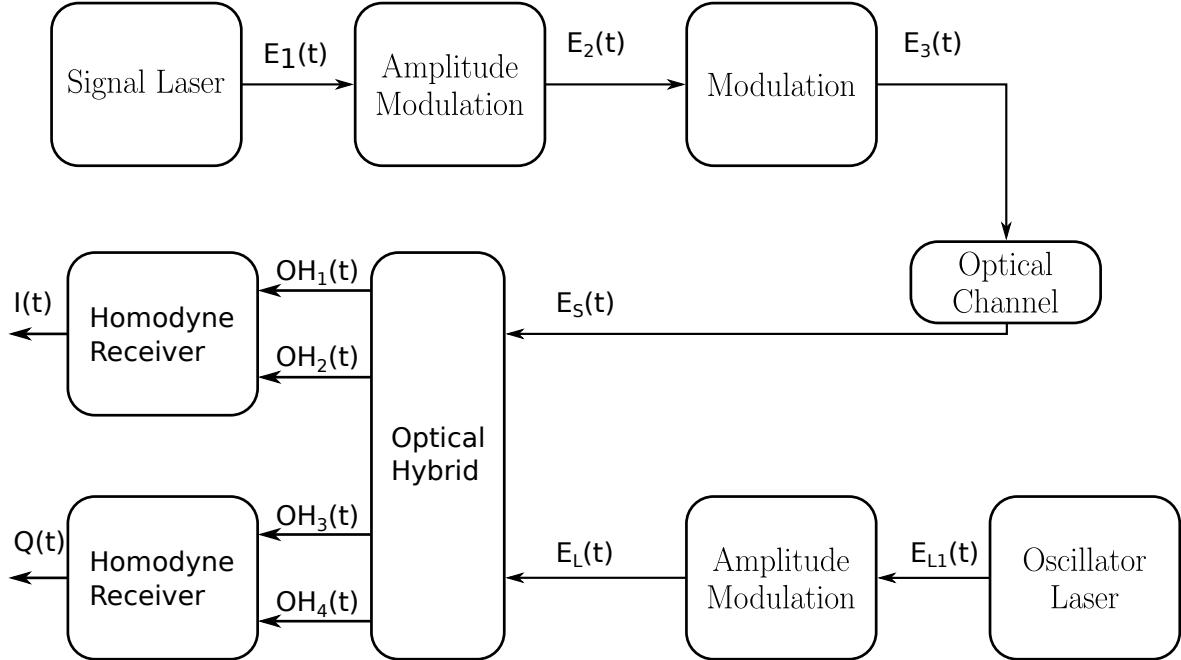


Figure 6.422: Sequential block diagram

Assuming a frequency mismatch between the transmitter and receiver local oscillators of  $\Delta\omega$  and a frequency mismatch of  $\Delta\epsilon$ , this leads to the following sampled signal

$$y_r(t_n) = \begin{cases} x_q(t_n)e^{i[\Delta\omega t_n + \theta_q(t_n) + \Delta\epsilon(t_n)]}, & n \neq mR_P \\ x_p(t_n)e^{i[\Delta\omega t_n + \Delta\epsilon(t_n)]}, & n = mR_P \end{cases}, \quad (6.301)$$

where  $m$  and  $n \in \mathbb{N}$ ,  $x_{q/p}(t_n)$  and  $\theta_{q/p}(t_n)$  are, respectively, the amplitude and phase of the quantum/pilot signal at the instant  $t_n$ .

The phase mismatch of the  $n$ th quantum symbol is compensated by taking a weighted average of the previous and ensuing pilots. For an example, let's consider  $R_P = 2$  and that only the two closest pilots are used. The weight for each pilot is the same and the phase estimate is given by

$$\begin{aligned} \hat{\Theta} &= \frac{\Delta\omega(t_n - T_s) + \Delta\epsilon(t_n - T_s) + \Delta\omega(t_n + T_s) + \Delta\epsilon(t_n + T_s)}{2} \\ &= \Delta\omega t_n + \frac{\Delta\epsilon(t_n - T_s) + \Delta\epsilon(t_n + T_s)}{2} \end{aligned} \quad (6.302)$$

The recovered constellation is given by

$$\begin{aligned}\bar{y}_r(t_n) &= x_q(t_n)e^{i[\Delta\omega t_n + \theta_q(t_n) + \Delta\epsilon(t_n)]}e^{-i\Theta} \\ &= x_q(t_n)e^{i[\Delta\omega t_n + \theta_q(t_n) + \Delta\epsilon(t_n) - \Delta\omega t_n - \frac{\Delta\epsilon(t_n - T_s) + \Delta\epsilon(t_n + T_s)}{2}]} \\ &= x_q(t_n)e^{i[\theta_q(t_n) + \frac{\Delta\epsilon(t_n) - \Delta\epsilon(t_n - T_s)}{2} + \frac{\Delta\epsilon(t_n) - \Delta\epsilon(t_n + T_s)}{2}]}\end{aligned}\quad (6.303)$$

Frequency mismatch is fully compensated, some phase mismatch remains.

#### 6.19.11.1 Delay Line

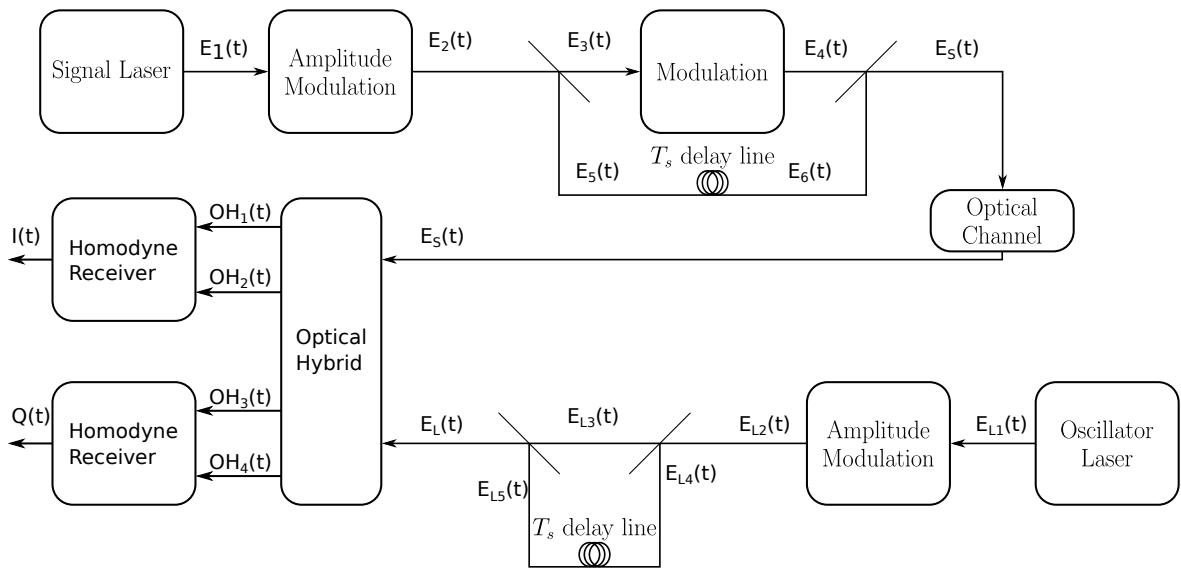


Figure 6.423: Delay line block diagram

The laser output can be described by:

$$E_1(t) = |E_s|e^{i(\omega_{st}t + \epsilon_s(t))}. \quad (6.304)$$

This signal is then pulsed by the following function

$$P(t) = \begin{cases} 1, & \text{for } t = 0, 2T_s, 4T_s, 6T_s, \dots \\ 0, & \text{for } t = T, 3T_s, 5T_s, 7T_s, \dots \end{cases}, \quad (6.305)$$

the specific shape of the pulses is not very relevant here, only the values indicated are of interest.

$$E_2(t) = P(t)|E_s|e^{i(\omega_{st}t + \epsilon_s(t))}. \quad (6.306)$$

The pulsed signal is split at a beamsplitter, resulting in

$$\begin{bmatrix} E_3(t) \\ E_5(t) \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} E_2(t) \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} P(t)|E_s|e^{i(\omega_{st}t + \epsilon_s(t))} \\ P(t)|E_s|e^{i(\omega_{st}t + \epsilon_s(t))} \end{bmatrix} \quad (6.307)$$

Signal  $E_3(t)$  is then phase modulated, resulting in

$$E_4(t) = E_3(t)e^{i\theta(t)} = P(t)|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} \quad (6.308)$$

Signal  $E_5(t)$  is delayed by  $T_s$ , resulting in

$$E_6(t) = E_5(t + T_s) = P(t + T_s)|E_s|e^{i(\omega_S(t + T_s) + \epsilon_S(t + T_s))} \quad (6.309)$$

Signals  $E_4(t)$  and  $E_6(t)$  are combined in a beamsplitter, only one output is monitored, the other is set to  $SINK(t)$ .

$$\begin{aligned} \begin{bmatrix} E_S(t) \\ SINK(t) \end{bmatrix} &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} E_4(t) \\ E_6(t) \end{bmatrix} \\ \iff E_S(t) &= \frac{1}{2} \left[ P(t)|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} + P(t + T_s)|E_s|e^{i(\omega_S(t + T_s) + \epsilon_S(t + T_s))} \right] \end{aligned} \quad (6.310)$$

The local oscillator laser output can be described by

$$E_{L1}(t) = |E_L|e^{i(\omega_L t + \epsilon_L(t))}. \quad (6.311)$$

As previously, this signal is then pulsed

$$E_{L2}(t) = P(t)|E_L|e^{i(\omega_L t + \epsilon_L(t))}, \quad (6.312)$$

the pulsed signal is passed through a beam splitter

$$\begin{bmatrix} E_{L3}(t) \\ E_{L4}(t) \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} E_{L2}(t) \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} P(t)|E_L|e^{i(\omega_L t + \epsilon_L(t))} \\ P(t)|E_L|e^{i(\omega_L t + \epsilon_L(t))} \end{bmatrix}. \quad (6.313)$$

Signal  $E_{L4}(t)$  is then delayed by  $T_s$ , resulting in

$$E_{L5}(t) = E_{L4}(t + T_s) = P(t + T_s)|E_L|e^{i(\omega_L(t + T_s) + \epsilon_L(t + T_s))}, \quad (6.314)$$

signals  $E_{L5}(t)$  and  $E_{L3}(t)$  are then combined, only one output is monitored, the other is set to  $SINK(t)$ .

$$\begin{aligned} \begin{bmatrix} E_L(t) \\ SINK(t) \end{bmatrix} &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} E_{L3}(t) \\ E_{L5}(t) \end{bmatrix} \\ \iff E_L(t) &= \frac{1}{2} \left[ P(t)|E_L|e^{i(\omega_L t + \epsilon_L(t))} + P(t + T_s)|E_L|e^{i(\omega_L(t + T_s) + \epsilon_L(t + T_s))} \right] \end{aligned} \quad (6.315)$$

Signals  $E_S(t)$  and  $E_L(t)$  are then combined in an optical hybrid, yielding

$$\begin{aligned} \begin{bmatrix} OH_1(t) \\ OH_2(t) \\ OH_3(t) \\ OH_4(t) \end{bmatrix} &= \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & i \\ 1 & -i \end{bmatrix} \begin{bmatrix} E_S(t) \\ E_L(t) \end{bmatrix} \\ &= \begin{bmatrix} P(t) [|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} + |E_L|e^{i(\omega_L t + \epsilon_L(t))}] \\ + P(t + T_s) [|E_s|e^{i(\omega_S(t+T_s) + \epsilon_S(t+T_s))} + |E_L|e^{i(\omega_L(t+T_s) + \epsilon_L(t+T_s))}] \\ P(t) [|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} - |E_L|e^{i(\omega_L t + \epsilon_L(t))}] \\ + P(t + T_s) [|E_s|e^{i(\omega_S(t+T_s) + \epsilon_S(t+T_s))} - |E_L|e^{i(\omega_L(t+T_s) + \epsilon_L(t+T_s))}] \\ P(t) [|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} + |E_L|e^{i(\omega_L t + \epsilon_L(t) + \frac{\pi}{2})}] \\ + P(t + T_s) [|E_s|e^{i(\omega_S(t+T_s) + \epsilon_S(t+T_s))} + |E_L|e^{i(\omega_L(t+T_s) + \epsilon_L(t+T_s) + \frac{\pi}{2})}] \\ P(t) [|E_s|e^{i(\omega_S t + \epsilon_S(t) + \theta(t))} - |E_L|e^{i(\omega_L t + \epsilon_L(t) + \frac{\pi}{2})}] \\ + P(t + T_s) [|E_s|e^{i(\omega_S(t+T_s) + \epsilon_S(t+T_s))} - |E_L|e^{i(\omega_L(t+T_s) + \epsilon_L(t+T_s) + \frac{\pi}{2})}] \end{bmatrix} \quad (6.316) \end{aligned}$$

Finally, these signals are evaluated in balanced homodyne receivers, yielding

$$\begin{aligned} I(t) &= OH_1(t)OH_1^*(t) - OH_2(t)OH_2^*(t) \\ &= \frac{P(t)|E_s||E_L|}{4} \cos(\Delta\omega t + \Delta\epsilon(t) + \theta(t)) + \frac{P(t+T_s)|E_s||E_L|}{4} \cos(\Delta\omega(t+T_s) + \Delta\epsilon(t+T_s)) \\ Q(t) &= OH_3(t)OH_3^*(t) - OH_4(t)OH_4^*(t) \\ &= \frac{P(t)|E_s||E_L|}{4} \sin(\Delta\omega t + \Delta\epsilon(t) + \theta(t)) + \frac{P(t+T_s)|E_s||E_L|}{4} \sin(\Delta\omega(t+T_s) + \Delta\epsilon(t+T_s)), \end{aligned} \quad (6.317)$$

where  $\Delta\omega = \omega_S - \omega_L$  and  $\Delta\epsilon(t) = \epsilon_S(t) - \epsilon_L(t)$ .

$$IQ(t) = \frac{P(t)|E_s||E_L|}{4} e^{i(\Delta\omega t + \Delta\epsilon(t) + \theta(t))} + \frac{P(t+T_s)|E_s||E_L|}{4} e^{i(\Delta\omega(t+T_s) + \Delta\epsilon(t+T_s))} \quad (6.318)$$

Signal

$$IQ(t = 2T, 4T, 6t, \dots) = \frac{|E_s||E_L|}{4} e^{i(\Delta\omega t + \Delta\epsilon(t) + \theta(t))} \quad (6.319)$$

Pilot

$$IQ(t = T, 3T, 5t, \dots) = \frac{|E_s||E_L|}{4} e^{i(\Delta\omega(t+T_s) + \Delta\epsilon(t+T_s))} \quad (6.320)$$

Each pilot is used to remove the phase noise of the next signal pulse, for the signal pulse at  $t = 2T$  we have, for example

$$\begin{aligned} \widehat{IQ}(2T) &= IQ(2T)IQ^*(T) \\ &= \left( \frac{|E_s||E_L|}{4} \right)^2 e^{i(\Delta\omega 2T_s + \Delta\epsilon(2T_s) + \theta(2T_s) - \Delta\omega(T_s + T_s) - \Delta\epsilon(T_s + T_s))} \\ &= \left( \frac{|E_s||E_L|}{4} \right)^2 e^{i\theta(2T_s)}, \end{aligned} \quad (6.321)$$

Frequency and phase mismatch are fully compensated.

### 6.19.12 Future work

#### 6.19.12.1 Comparative analysis of the phase mismatch compensation techniques

The EVM of the recovered constellations (6.291) and (6.298) was evaluated in function of the symbol rate and for multiple laser linewidths and is presented in Figure 6.424. To allow for a better comparison, the pilot rate for the LLO-sequential technique was assumed to take the value  $P = 2$ , from this choice the remaining variance (6.292) simplifies to

$$\text{Var}[\Delta\Theta(nT_s)] = \pi T_s(\nu_T + \nu_R). \quad (6.322)$$

The EVM was computed by generating the constellations (6.291) and (6.298) with the respective remaining phase noise (6.322) and (6.299) and computing the EVM from these simulated results.

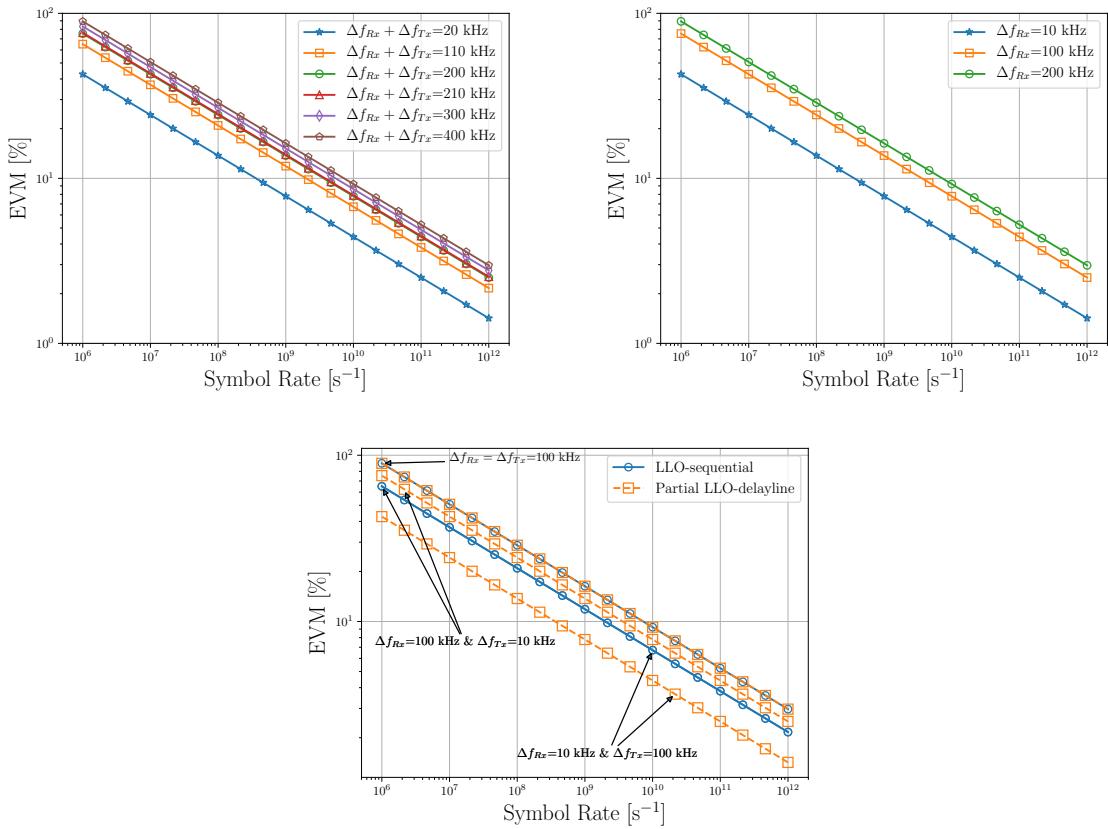


Figure 6.424: EVM in function of the sampling frequency for the LLO-sequential (top left) and partial LLO-delayline (top right) phase mismatch compensation schemes. The bottom figure compares the EVM of the LLO-sequential and the partial LLO-delayline for 3 different combinations of transmission and receptor laser linewidths.

The performance of the LLO-sequential technique does not depend on the individual values of the linewidths, but rather on the value of their sum, i.e. the EVM of a system with  $\Delta f_{T_x} = 10$  kHz and  $\Delta f_{R_x} = 100$  kHz will be equal to that of a system with  $\Delta f_{T_x} = 100$  kHz and  $\Delta f_{R_x} = 10$  kHz (as is visible in Figure 6.424-bottom).

The performance of the partial LLO-delayline scheme, in relation to that of the LLO-sequential scheme will be

- equal, if the transmission and reception lasers have the same linewidths;
- worse, if the transmission laser has a narrower linewidth than the reception laser;
- better, if the transmission laser has a wider linewidth than the reception laser.

All of these results are visible in Figure 6.424-bottom

#### 6.19.12.2 Memory aided sequential referenced LLO CV-QC

We here study the advantages of employing a memory aided sequentially-referenced LLO CV-QC system. Figure 6.425 is included to facilitate this study.

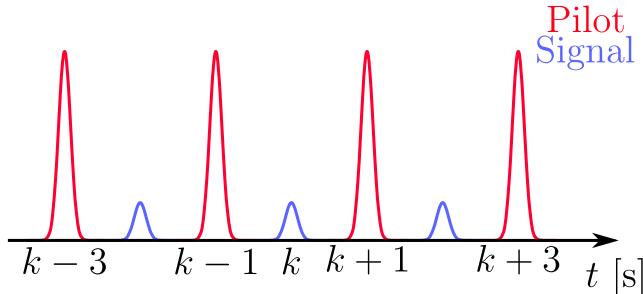


Figure 6.425: Time dependence of a sequential LLO scheme. Indexing in the figure should be considered as relative to the central signal pulse.

Assuming this time dependence to be sampled at the maximum of each pulse, this yields signal and pilot constellations given by

$$x_s(k) = |x_s| e^{i(\theta(t_k) + \phi(t_k) + \phi_0 - \varphi(t_k) - \varphi_0)}, \quad k \text{ is odd}; \quad (6.323)$$

$$x_p(k) = |x_p| e^{i(\phi(t_k) + \phi_0 - \varphi(t_k) - \varphi_0)}, \quad k \text{ is even}; \quad (6.324)$$

where  $\theta(t_k)$  is the phase encoded in the  $k$ th signal symbol,  $\phi(t_k)/\varphi(t_k)$  is the instantaneous phase of the transmission/reception laser at the instant  $t_k$  and  $\phi_0/\varphi_0$  is the initial phase of the transmission/reception laser.

The phase difference between the transmission and reception lasers for the  $k$ th signal symbol can be estimated by using a weighted average of the preceding and ensuing pilot symbols. The originally proposed sequentially-referenced LLO utilized only the pilot

symbols immediately before and after, corresponding in the Figure 6.425 to utilizing the pilot pulses  $k \pm 1$  to compensate the phase difference in the  $k$ th signal symbol. In this case it is easy to understand that both pilot symbols should have the same weight in the estimation, since they both have the same time distance from the point being estimated. This method yields the recovered constellation

$$\begin{aligned} x(k) &= |x_s| e^{i(\theta(t_k) + \phi(t_k) + \phi_0 - \varphi(t_k) - \varphi_0)} e^{-i\left(\frac{\phi(t_{k-1}) + \phi_0 - \varphi(t_{k-1}) - \varphi_0}{2} + \frac{\phi(t_{k+1}) + \phi_0 - \varphi(t_{k+1}) - \varphi_0}{2}\right)} \\ &= |x_s| e^{i\left(\theta(t_k) + \frac{\phi(t_k) - \phi(t_{k-1}) + \phi(t_k) - \phi(t_{k+1})}{2} - \frac{\varphi(t_k) - \varphi(t_{k-1}) + \varphi(t_k) - \varphi(t_{k+1})}{2}\right)}, \quad k \text{ is odd.} \end{aligned} \quad (6.325)$$

The phase variance of this recovered constellation is given by

$$\begin{aligned} \text{Var} [\arg(x(k))] &= \frac{1}{2} \text{Var} [\phi(t_k) - \phi(t_{k-1})] + \frac{1}{2} \text{Var} [\varphi(t_k) - \varphi(t_{k-1})] \\ &= \pi(\Delta\nu_T + \Delta\nu_R)T_s, \end{aligned} \quad (6.326)$$

where  $\Delta\nu_T / \Delta\nu_R$  is the transmission/reception laser's linewidth and  $T_s$  is the symbol period (note that  $t_k - t_{k-1} = T_s$ ).

Lets now consider that four pilot pulses are used to estimate the phase difference in the  $k$ th signal symbol (pulses  $k \pm 1$  and  $k \pm 3$  in Figure 6.425). The averaging weight of each pulse is not immediately clear, so for now they will be termed  $\mu_1$  and  $\mu_2$  for the pulses  $k \pm 1$  and  $k \pm 3$  respectively. The remaining phase variance in this case is given by

$$\begin{aligned} \text{Var} [\arg(x(k))] &= 2\mu_1^2 \text{Var} [\phi(t_k) - \phi(t_{k-1})] + 2\mu_1^2 \text{Var} [\varphi(t_k) - \varphi(t_{k-1})] \\ &\quad + 2\mu_2^2 \text{Var} [\phi(t_k) - \phi(t_{k-3})] + 2\mu_2^2 \text{Var} [\varphi(t_k) - \varphi(t_{k-3})] \\ &= \pi(\Delta\nu_T + \Delta\nu_R)T_s 4(\mu_1^2 + 3\mu_2^2), \quad 2\mu_1 + 2\mu_2 = 1. \end{aligned} \quad (6.327)$$

The values for  $\mu_1$  and  $\mu_2$  can be optimized within their constrains in order to minimize the value of  $\text{Var} [\arg(x(k))]$ . The solution for this optimization yields the values  $\mu_1 = \frac{3}{8}$  and  $\mu_2 = \frac{3}{8}$ , which corresponds to a remaining variance of

$$\text{Var} [\arg(x(k))] = \frac{3}{4}\pi(\Delta\nu_T + \Delta\nu_R)T_s, \quad (6.328)$$

which is lower than the remaining variance observed when using only two pilot pulses.

Performing a couple more iterations quickly reveals that, for an arbitrary number of pilot pulses  $2n$  ( $n$  before and  $n$  after the signal symbol) employed in estimating the phase difference in the  $k$ th signal symbol yields a remaining phase variance given by

$$\text{Var} [\arg(x(k))] = \pi(\Delta\nu_T + \Delta\nu_R)T_s 4 \sum_{i=1}^n (2i-1)\mu_i^2, \quad 2 \sum_{i=1}^n \mu_i = 1. \quad (6.329)$$

Assuming the properties of the optical system remain the same (i.e. the value of  $\pi(\Delta\nu_T + \Delta\nu_R)T_s$  is constant), we have already seen we can reduce the remaining variance by utilizing four pilot pulses instead of only two. We can now prove that we can reduce the remaining

phase variance by an arbitrary amount by employing an appropriate number of pilot pulses  $n$ . We can prove this by computing

$$\lim_{n \rightarrow \infty} \min_{2 \sum_{i=1}^n \mu_i = 1} 4 \sum_{i=1}^n (2i-1) \mu_i^2 = \lim_{n \rightarrow \infty} \min_{g(\mu_i)=0} f(\mu_i) \quad (6.330)$$

This can be accomplished by using the Lagrangian multiplier method, which can maximize or minimize a function  $f(x_i)$  subject to a constraint  $g(x_i)$ , where  $x_i$  is any group of variables, by studying the roots of the Lagrangian

$$\mathcal{L} = f(x_i) \mp \lambda g(x_i), \quad (6.331)$$

where  $\lambda$  is dubbed the Lagrangian multiplier and serves only as an auxiliary variable. The roots of this Lagrangian are evaluated in relation to both its original variables  $x_i$  and to the Lagrangian multiplier  $\lambda$  by application of the gradient operator

$$\nabla_{x_i, \lambda} \mathcal{L} = 0 \iff \begin{cases} \frac{\partial \mathcal{L}}{\partial x_i} = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda} = 0 \end{cases}. \quad (6.332)$$

Applying this to our optimization problem yields

$$\begin{cases} \frac{\partial(4 \sum_{i=1}^n (2i-1) \mu_i^2 + \lambda(2 \sum_{i=1}^n \mu_i - 1))}{\partial \mu_i} = 0 \\ \frac{\partial(4 \sum_{i=1}^n (2i-1) \mu_i^2 + \lambda(2 \sum_{i=1}^n \mu_i - 1))}{\partial \lambda} = 0 \end{cases} \quad \begin{cases} 4 \sum_{i=1}^n (2i-1) \mu_i + \lambda \sum_{i=1}^n 1 = 0 \\ 2 \sum_{i=1}^n \mu_i = 1 \end{cases} \\ \begin{cases} 4 \sum_{i=1}^n (2i-1) \mu_i = - \sum_{i=1}^n \lambda \\ - \end{cases} \quad \begin{cases} 4(2i-1) \mu_i = -\lambda \\ - \end{cases} \quad \begin{cases} \mu_i = -\frac{\lambda}{4(2i-1)} \\ - \end{cases} \\ \begin{cases} - \\ \sum_{i=1}^n \frac{-\lambda}{2(2i-1)} = 1 \end{cases} \quad \begin{cases} - \\ \lambda = -\frac{2}{\sum_{i=1}^n \frac{1}{2i-1}} = -\frac{2}{s} \end{cases} \quad \begin{cases} \mu_i = -\frac{\lambda}{2s(2i-1)} \\ - \end{cases} \quad . \quad (6.333)$$

Substituting this result into  $f(\mu_i)$  yields

$$4 \sum_{i=1}^n (2i-1) \frac{1}{4s^2(2i-1)^2} = \frac{1}{\left(\sum_{i=1}^n \frac{1}{2i-1}\right)^2} \sum_{i=1}^n \frac{1}{2i-1} = \left(\sum_{i=1}^n \frac{1}{2i-1}\right)^{-1}, \quad (6.334)$$

this is the solution to the constrained minimization problem. This result is plotted function of the memory size in Figure 6.426. Finally, inputting this result into (6.330) yields

$$\lim_{n \rightarrow \infty} \left(\sum_{i=1}^n \frac{1}{2i-1}\right)^{-1} = 0, \quad (6.335)$$

which is the result we set out to obtain.

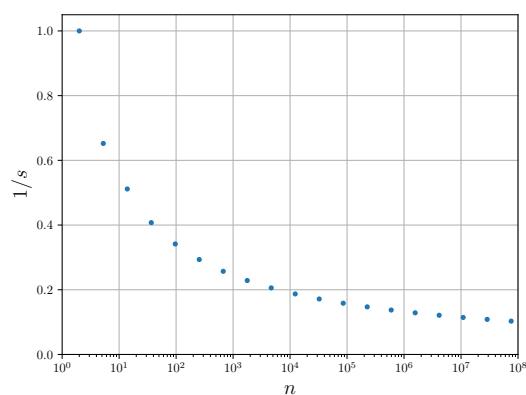


Figure 6.426: Reduction of the observed remaining variance in function of the size of the memory employed.

## References

- [1] Ezra Ip and Joseph M Kahn. "Feedforward carrier recovery for coherent optical communications". In: *Journal of Lightwave Technology* 25.9 (2007), pp. 2675–2692.
- [2] Md Saifuddin Faruk and Seb J Savory. "Digital signal processing for coherent transceivers employing multilevel formats". In: *Journal of Lightwave Technology* 35.5 (2017), pp. 1125–1141.
- [3] Xiang Zhou et al. "64-Tb/s, 8 b/s/Hz, PDM-36QAM transmission over 320 km using both pre-and post-transmission digital signal processing". In: *Journal of Lightwave Technology* 29.4 (2011), pp. 571–577.
- [4] Mehrez Selmi, Yves Jaouën, and Philippe Ciblat. "Accurate digital frequency offset estimator for coherent PolMux QAM transmission systems". In: *Optical Communication, 2009. ECOC'09. 35th European Conference on*. IEEE. 2009, pp. 1–2.
- [5] Xian Zhou, Xue Chen, and Keping Long. "Wide-range frequency offset estimation algorithm for optical coherent systems using training sequence". In: *IEEE Photonics Technology Letters* 24.1 (2012), pp. 82–84.
- [6] Michael G Taylor. "Phase estimation methods for optical coherent detection using digital signal processing". In: *Journal of Lightwave Technology* 27.7 (2009), pp. 901–914.
- [7] Maurizio Magarini et al. "Pilot-symbols-aided carrier-phase recovery for 100-G PM-QPSK digital coherent receivers". In: *IEEE Photonics Technology Letters* 24.9 (2012), p. 739.
- [8] Ricardo M Ferreira et al. "Optimized Carrier Frequency and Phase Recovery Based on Blind  $M$  th Power Schemes". In: *IEEE Photonics Technology Letters* 28.21 (2016), pp. 2439–2442.
- [9] MB Costa Silva et al. "Homodyne detection for quantum key distribution: an alternative to photon counting in BB84 protocol". In: *Photonics North 2006*. Vol. 6343. International Society for Optics and Photonics. 2006, 63431R.
- [10] John Bertrand Johnson. "Thermal agitation of electricity in conductors". In: *Physical review* 32.1 (1928), p. 97.
- [11] Harry Nyquist. "Thermal agitation of electric charge in conductors". In: *Physical review* 32.1 (1928), p. 110.
- [12] T. C. Ralph. "Continuous variable quantum cryptography". In: *Phys. Rev. A* 61 (1 Dec. 1999), p. 010303. DOI: [10.1103/PhysRevA.61.010303](https://doi.org/10.1103/PhysRevA.61.010303). URL: <https://link.aps.org/doi/10.1103/PhysRevA.61.010303>.
- [13] Mark Hillery. "Quantum cryptography with squeezed states". In: *Physical Review A* 61.2 (2000), p. 022309.
- [14] Xiang-Chun Ma et al. "Wavelength attack on practical continuous-variable quantum-key-distribution system with a heterodyne protocol". In: *Physical Review A* 87.5 (2013), p. 052309.

- [15] Jing-Zheng Huang et al. "Quantum hacking on quantum key distribution using homodyne detection". In: *Physical Review A* 89.3 (2014), p. 032304.
- [16] Paul Jouguet, Sébastien Kunz-Jacques, and Eleni Diamanti. "Preventing calibration attacks on the local oscillator in continuous-variable quantum key distribution". In: *Physical Review A* 87.6 (2013), p. 062313.
- [17] Jing-Zheng Huang et al. "Quantum hacking of a continuous-variable quantum-key-distribution system using a wavelength attack". In: *Physical Review A* 87.6 (2013), p. 062329.
- [18] Daniel BS Soh et al. "Self-referenced continuous-variable quantum key distribution protocol". In: *Physical Review X* 5.4 (2015), p. 041010.
- [19] Bing Qi et al. "Generating the local oscillator "locally" in continuous-variable quantum key distribution based on coherent detection". In: *Physical Review X* 5.4 (2015), p. 041009.
- [20] Adrien Marie and Romain Alléaume. "Self-coherent phase reference sharing for continuous-variable quantum key distribution". In: *Physical Review A* 95.1 (2017), p. 012316.
- [21] S Kleis and CG Schaeffer. "Comparison of frequency estimation methods for heterodyne quantum communications". In: *Photonic Networks; 18. ITG-Symposium; Proceedings of*. VDE. 2017, pp. 1–3.
- [22] Fabian Laudenbach et al. "Pilot-assisted intradyne reception for high-speed continuous-variable quantum key distribution with true local oscillator". In: *arXiv preprint arXiv:1712.10242* (2017).
- [23] *PDB48xC-AC Operation Manual*. Thorlabs. 2018.

## 6.20 Quantum Key Distribution Without Basis Switching

|                     |   |  |
|---------------------|---|--|
| <b>Student Name</b> | : | Goncalo Nunes (2018/05/20- )   |
| <b>Goal</b>         | : | Provide practical evidence of the employability of a coherent state quantum key distribution protocol based on simultaneous quadrature measurements. |
| <b>Directory</b>    | : | sdf/qkd_with_cv_without_base_switching   |

Quantum key distribution (QKD) is a method to generate a cryptographic key between two distant parties, Alice and Bob, based on transmission of quantum states. After said transmission and respective measurement, Alice and Bob can then exchange classical messages through an insecure channel and, using the keys generated, perform post-processing to recover the messages.

In the first quantum key distribution schemes, single photons acted as information carriers (discrete variable regime). The practical implementation of said schemes was limited by the single photon generation and detection techniques. Hence the current interest in continuous variable (CV) quantum cryptography, which allows for higher key rates. The security of a CV-QKD relies on randomly switching the measurement basis (quadratures). In practice this implies a change of phase of a local oscillator beam which is difficult to achieve and constitutes a serious technical difficulty for the implementation of this type of cryptosystem, compromising its bandwidth.

Hence the pertinence of a QKD scheme that doesn't require for the change in measurement basis. Such protocol was analysed in [1] and is called the simultaneous quadrature measurement or SQM Protocol, where both bases are measured simultaneously through double homodyne detection. It utilizes the quantum channel more effectively and achieves both higher secret key rates and bandwidths compared to orthodox CV-QKD protocols.

### 6.20.1 Theoretical Analysis

The scheme is similar to an ordinary continuous variable coherent state quantum cryptography protocol. Alice prepares a state by displacing the amplitude and phase quadratures of a vacuum state according to a random Gaussian variable. The quadrature operators of Alice's state are given by:

$$\hat{X}_A^{\pm} = S^{\pm} + \hat{X}^{\pm} \quad (6.336)$$

Where  $S^{\pm}$  are the quadrature operators of the initial vacuum state [1]. In practice this is done through the modulation of previously prepared squared laser pulse using phase and amplitude modulation, changing the quadratures  $\hat{X}^-$  and  $\hat{X}^+$  respectively. Afterwards Alice sends a multiplexed signal: one polarization containing the modulated signal and the orthogonal one with a reference, commonly named local oscillator. This multiplexed signal is mixed using a polarization beam splitter, PBS. Unwanted feedback in the circuit can be prevented by implementing optical isolators, as displayed in the scheme.

The multiplexed signal is sent through a quantum channel, an optical fiber, to Bob with efficiency  $\eta$  and couples in channel noise  $N^\pm$ . The signal is demultiplex using again a PBS, splitting the modulated state and the reference. Bob simultaneously measures the amplitude and phase quadratures of the state using a 50/50 beam splitter. Part of the split local oscillator's phase is shifted  $\pi/2$  for the measurement of  $\hat{X}^+$  and is left unchanged for  $\hat{X}^-$ . The quadratures are measured through the comparison of the modulated signal with the local oscillator, this is homodyne detection.

The state Bob received is given by [1],

$$\hat{X}_B^\pm = \frac{1}{\sqrt{2}} \left( \sqrt{\eta} \hat{X}_A^\pm + \sqrt{1-\eta} \hat{X}_N^\pm + \hat{N}_B^\pm \right) \quad (6.337)$$

Alice and Bob then test the channel transmission and bit error rate of their key by using a randomly chosen subset of their key to check for errors. If the errors are within some tolerated limit, they then use secret key distillation with the objective to produce a key between Alice and Bob which has negligible errors. The rate at which the key is generated has a lower bound given by [1]:

$$\text{Lower Bound} = \log_2 \left( \frac{\left( \frac{\eta}{V_A} + (1-\eta)V_N \right)^{-1} + 1}{\eta + (1-\eta)V_N + 1} \right) \quad (6.338)$$

where  $V_A$  is the variance of Alice's state and symmetry between the amplitude and phase quadratures is assumed. We see that, so long as the channel noise  $V_N$  is not excessive, a secret key can be successfully generated between Alice and Bob, even in the limit of very small channel efficiency  $\eta$ .

## References

- [1] Christian Weedbrook et al. "Quantum Cryptography Without Switching". In: *Physical Review Letters* 93.17 (Oct. 2004). DOI: [10.1103/physrevlett.93.170504](https://doi.org/10.1103/physrevlett.93.170504). URL: <https://doi.org/10.1103/physrevlett.93.170504>.

## 6.21 Intradyne Continuous Variables QKD Transmission System

|                     |   |  |
|---------------------|---|--|
| <b>Student Name</b> | : | Nuno Silva (11/01/2018 - )   |
| <b>Goal</b>         | : | Simulation and experimental validation of a CV-QKD transmission system with a real local oscillator. |
| <b>Directory</b>    | : | sdf/cv_intradyne_system  |

The aim of Continuous Variable Quantum Key Distribution (CV-QKD) is to encode information in observables whose measurements take continuous values, such as the phase and amplitude of a electromagnetic wave, or the stokes parameters. In this work we are going to analyze a CV-QKD system which uses the phase and amplitude degrees of freedom of a coherent light field to generate a secret key. Two protocols are going to be analyzed for implement a CV-QKD system: (1) using a continuous Gaussian-modulated coherent state (continuous constellation) at transmitter side; (2) using a discrete-modulated coherent state (M-ary PSK) at transmitter side. In both protocols, the receiver will implemented a double homodyne detection scheme with an optical hybrid, and locally generated local oscillator, i.e. intradyne detection. Due to phase/frequency uncorrelation between quantum signals and local oscillator, the transmitter will send a strong (classical) reference signal for phase/frequency reference.

### 6.21.1 Theoretical Analysis

In this section we are going to describe detailed the CV-QKD setup, using continuous- and discrete-modulated coherent states. The continuous Gaussian-modulation of a coherent state in principle leads to a higher achievable information rate between the two legitimate users of a CV-QKD system. However, in practice it is preferable to use the discrete-modulation of a coherent state (such as the QPSK constellation) since for this case there exist high efficient error correction codes even at very low SNR values. These two approaches to the implementation of a CV-QKD system will be analyzed in the next two sections.

#### 6.21.1.1 Gaussian-Modulated Coherent States

In the Gaussian-modulated coherent state CV-QKD protocol, Alice randomly generates a coherent state  $|\alpha\rangle$  with  $\alpha = |\alpha|e^{i\theta}$  where  $|\alpha|$  is the amplitude of the coherent-state and  $\theta$  is its phase. In this notation the coherent state can be written

$$|\alpha\rangle = \exp\left(-\frac{1}{2}|\alpha|^2\right) \sum_{n=0}^{+\infty} \frac{\alpha^n}{(n!)^{\frac{1}{2}}} |n\rangle \quad (6.339)$$

$$= \exp\left(-\frac{1}{2}|\alpha|^2\right) \sum_{n=0}^{+\infty} \frac{(\alpha\hat{a}^\dagger)^n}{(n!)} |0\rangle, \quad (6.340)$$

where  $|n\rangle$  represent the number state, and  $\hat{a}^\dagger$  ( $\hat{a}$ ) represents the creation (annihilation) operator, respectively, which obey to the commutation relation

$$[\hat{a}, \hat{a}^\dagger] = \hat{a}\hat{a}^\dagger - \hat{a}^\dagger\hat{a} = 1. \quad (6.341)$$

The coherent state obeys to the following properties

$$\hat{a}|\alpha\rangle = \alpha|\alpha\rangle, \quad (6.342)$$

$$\langle\alpha|\hat{a}^\dagger = \langle\alpha|\alpha^*, \quad (6.343)$$

$$\langle\alpha|\alpha\rangle = 1, \quad (6.344)$$

$$\langle\alpha|\beta\rangle = \exp\left(-\frac{1}{2}|\alpha|^2 - \frac{1}{2}|\beta|^2 + \alpha^*\beta\right), \quad (6.345)$$

$$|\langle\alpha|\beta\rangle|^2 = \exp(-|\alpha - \beta|^2). \quad (6.346)$$

$$(6.347)$$

Note that the coherent states  $|\alpha\rangle$  and  $|\beta\rangle$  become orthogonal when  $|\alpha - \beta|^2 \gg 1$ .

The coherent-state expectation value for the number operator and its variance are given by

$$\bar{n} = \langle\alpha|\hat{n}|\alpha\rangle = |\alpha|^2 \quad (6.348)$$

$$(\Delta n)^2 = \langle\alpha|(\hat{n})^2|\alpha\rangle = \langle\alpha|\hat{a}^\dagger\hat{a}^\dagger\hat{a}\hat{a}|\alpha\rangle + \langle\alpha|\hat{n}|\alpha\rangle - (\langle\alpha|\hat{n}|\alpha\rangle)^2 = |\alpha|^2 \quad (6.349)$$

where  $\hat{n} = \hat{a}^\dagger\hat{a}$ . The probability of finding  $n$  photons in the coherent-state  $|\alpha\rangle$  is

$$P(n) = |\langle n|\alpha\rangle|^2 = \exp(-|\alpha|^2) \frac{|\alpha|^{2n}}{n!} = e^{-\bar{n}} \frac{\bar{n}^n}{n!}, \quad (6.350)$$

which is a Poisson probability distribution. For large values of  $\bar{n}$ , the Poisson distribution approaches a Gaussian distribution, by using  $\bar{n}^n = \exp(n \ln(\bar{n}))$

$$P(n) \approx \frac{1}{\sqrt{2\pi\bar{n}}} \exp\left(-\frac{(n-\bar{n})^2}{2\bar{n}}\right) \quad (6.351)$$

For a coherent-state  $|\alpha\rangle$  with  $\alpha = |\alpha_A|e^{i\theta}$ , the corresponding phase distribution is

$$P(\varphi) = |\langle\varphi|\alpha\rangle|^2 = \frac{1}{2\pi} e^{-\frac{1}{2}|\alpha|^2} \left| \sum_{n=0}^{+\infty} e^{in(\varphi-\theta)} \frac{|\alpha|^n}{\sqrt{n!}} \right|^2, \quad (6.352)$$

with  $\bar{\varphi} = \theta$ . For  $|\alpha|^2 \gg 1$ ,  $P(\varphi)$  tends to a Gaussian distribution

$$P(\varphi) \approx \left( \frac{2|\alpha|^2}{\pi} \right)^{\frac{1}{2}} \exp(-2\bar{n}(\varphi-\theta)^2). \quad (6.353)$$

In this case  $(\Delta\varphi)^2 = 1/(4\bar{n})$ . The number-phase uncertainty is therefore

$$(\Delta n)^2 (\Delta\varphi)^2 = 1/4. \quad (6.354)$$

An important point to analyze now is the quantization of the electromagnetic field. The electromagnetic field is quantized by the association of a quantum-mechanical harmonic-oscillator with each mode of radiation. We assume a one-dimensional harmonic oscillator for this work. In that case the Hamiltonian is

$$\hat{H} = \frac{1}{2} (\hat{p}^2 + \omega^2 \hat{q}^2), \quad (6.355)$$

where  $\omega$  is the frequency of the mode,  $\hat{p}$  and  $\hat{q}$  are Hermitian operators and therefore correspond to observable quantities, with  $\hat{p}$  being the momentum operator and  $\hat{q}$  the position operator which obey to the commutation relation

$$[\hat{q}, \hat{p}] = i\hbar. \quad (6.356)$$

It is convenient non-Hermitian (and therefore non-observable) annihilation and creation operators

$$\hat{a} = (2\hbar\omega)^{-1/2} (\omega\hat{q} + i\hat{p}), \quad (6.357)$$

$$\hat{a}^\dagger = (2\hbar\omega)^{-1/2} (\omega\hat{q} - i\hat{p}). \quad (6.358)$$

Then the electric and the magnetic field operators can be written as

$$\hat{E}_x(z, t) = \hat{E}_x^+(z, t) + \hat{E}_x^-(z, t) \quad (6.359)$$

$$= \sqrt{\frac{\hbar\omega}{2\epsilon_0 V}} (\hat{a}e^{-i\chi} + \hat{a}^\dagger e^{i\chi}), \quad (6.360)$$

$$\hat{B}_y(z, t) = \hat{B}_y^+(z, t) + \hat{B}_y^-(z, t) \quad (6.361)$$

$$= -i \frac{\epsilon_0 \mu_0}{k} \sqrt{\frac{\hbar\omega}{2\epsilon_0 V}} (\hat{a}e^{-i\chi} + \hat{a}^\dagger e^{i\chi}), \quad (6.362)$$

where  $\chi = \omega t - kz - \pi/2$  is the phase of the electromagnetic field, with  $k = \omega/c$  is the wave number,  $V$  is the efective volume of the cavity used for quantization,  $\epsilon_0$  is the vacuum permitivity,  $\mu_0$  is the vacuum permaability,  $c$  is the speed of light in vacuum, and the plus and minus symbols are known as the positive and negative frequency parts of the electromagnetic field. From the electromagnetic field operators we can right the radiation Hamiltonian

$$\hat{H}_R = \frac{1}{2} \int dV \left( \epsilon_0 \hat{E}_x(z, t) \hat{E}_x(z, t) + \frac{1}{\mu_0} \hat{B}_y(z, t) \hat{B}_y(z, t) \right) \quad (6.363)$$

$$= \frac{1}{2} (\hat{p}^2 + \omega^2 \hat{q}^2). \quad (6.364)$$

We give particular attention to the electric field operator because of its important role in the the definitions of degrees of coherence. We now introduce the so-called quadrature operators

$$\hat{X} = \frac{1}{2} (\hat{a}^\dagger + \hat{a}), \quad (6.365)$$

$$\hat{Y} = \frac{i}{2} (\hat{a}^\dagger - \hat{a}), \quad (6.366)$$

with

$$[\hat{X}, \hat{Y}] = i/2. \quad (6.367)$$

The electric field can be written using these quadrature operators

$$\hat{E}_x(\chi) = \sqrt{\frac{2\hbar\omega}{\epsilon_0 V}} (\hat{X} \cos(\chi) + \hat{Y} \sin(\chi)) \quad (6.368)$$

The electric field operator obey to the commutation relation

$$[\hat{E}_x(\chi_1), \hat{E}_x(\chi_2)] = -\frac{i}{2} \sin(\chi_1 - \chi_2). \quad (6.369)$$

Let us now consider the expectation values for those operators considering the coherent-state

$$\langle \alpha | \hat{n} | \alpha \rangle = \frac{1}{2\hbar\omega} (\omega^2 \langle \alpha | \hat{q}^2 | \alpha \rangle + \langle \alpha | \hat{p}^2 | \alpha \rangle - \hbar\omega) = |\alpha_A|^2 \quad (6.370)$$

with

$$\langle \alpha | \hat{q}^2 | \alpha \rangle = \frac{\hbar}{2\omega} (2|\alpha|^2 + 1 + \alpha^2 + (\alpha^*)^2), \quad (6.371)$$

$$\langle \alpha | \hat{p}^2 | \alpha \rangle = \frac{\hbar\omega}{2} (2|\alpha|^2 + 1 - \alpha^2 - (\alpha^*)^2). \quad (6.372)$$

The variance of the momentum and position operators is

$$(\Delta q)^2 = \langle \alpha | \hat{q}^2 | \alpha \rangle - (\langle \alpha | \hat{q} | \alpha \rangle)^2 = \frac{\hbar}{2\omega}, \quad (6.373)$$

$$(\Delta p)^2 = \langle \alpha | \hat{p}^2 | \alpha \rangle - (\langle \alpha | \hat{p} | \alpha \rangle)^2 = \frac{\hbar\omega}{2}, \quad (6.374)$$

with

$$\langle \alpha | \hat{q} | \alpha \rangle = \sqrt{\frac{\hbar}{2\omega}} (\alpha + \alpha^*), \quad (6.375)$$

$$\langle \alpha | \hat{p} | \alpha \rangle = i\sqrt{\frac{\hbar\omega}{2}} (\alpha^* - \alpha). \quad (6.376)$$

In terms of quadrature operators we have

$$\langle \alpha | \hat{X} | \alpha \rangle = \frac{1}{2} (\alpha + \alpha^*) = |\alpha| \cos(\theta), \quad (6.377)$$

$$\langle \alpha | \hat{Y} | \alpha \rangle = \frac{i}{2} (\alpha^* - \alpha) = |\alpha| \sin(\theta), \quad (6.378)$$

$$\langle \alpha | \hat{X}^2 | \alpha \rangle = \frac{1}{4} (2|\alpha|^2 + \alpha^2 + 1 + (\alpha^*)^2) = \frac{1}{4} + |\alpha|^2 \cos^2(\theta), \quad (6.379)$$

$$\langle \alpha | \hat{Y}^2 | \alpha \rangle = \frac{1}{4} (2|\alpha|^2 + 1 - \alpha^2 - (\alpha^*)^2) = \frac{1}{4} + |\alpha|^2 \sin^2(\theta). \quad (6.380)$$

The variance of the quadrature operators is

$$(\Delta X)^2 = \langle \alpha | \hat{X}^2 | \alpha \rangle - \left( \langle \alpha | \hat{X} | \alpha \rangle \right)^2 = \frac{1}{4}, \quad (6.381)$$

$$(\Delta Y)^2 = \langle \alpha | \hat{Y}^2 | \alpha \rangle - \left( \langle \alpha | \hat{Y} | \alpha \rangle \right)^2 = \frac{1}{4}, \quad (6.382)$$

thus the coherent-state is a quadrature minimum uncertainty state for all mean photon numbers  $|\alpha_A|^2$ .

The expectation values for the electric field operator is given by

$$\langle \alpha | \hat{E}_x(\chi) | \alpha \rangle = |\alpha| \sqrt{\frac{2\hbar\omega}{\epsilon_0 V}} \cos(\chi - \theta), \quad (6.383)$$

$$\left\langle \alpha \left| \left( \hat{E}_x(\chi) \right)^2 \right| \alpha \right\rangle = \frac{2\hbar\omega}{\epsilon_0 V} \left( \frac{1}{4} + \frac{|\alpha|^2}{2} \{ 1 + \cos(2\theta) \cos(2\chi) + 2 \sin(2\theta) \sin(\chi) \cos(\chi) \} \right). \quad (6.384)$$

The variance of the electric field is given by

$$(\Delta E_x(\chi))^2 = \left\langle \alpha \left| \left( \hat{E}_x(\chi) \right)^2 \right| \alpha \right\rangle - \left( \langle \alpha | \hat{E}_x(\chi) | \alpha \rangle \right)^2 = \frac{1}{4} \left( \frac{2\hbar\omega}{\epsilon_0 V} \right), \quad (6.385)$$

The coherent-states  $|\alpha\rangle$  are quantum states very close to classical states because: (1) the expectation value of the electric field has the form of the classical expression; (2) the fluctuations in the electric field variables are the same as for a vacuum; (3) the fluctuation in the fractional uncertainty for the photon number decrease with the increasing average photon number; (4) the states become well localized in phase with the increasing average photon number. Moreover, the coherent state may be generated by a classical oscillating current, such as laser radiation which is produced in a resonant cavity where the resonant frequency of the cavity is the same as the frequency associated with the atomic electron transitions providing energy flow into the field.

In the Gaussian-modulated coherent state CV-QKD protocol, Alice randomly generates a coherent state  $|\alpha\rangle$  with  $\alpha = |\alpha_A| e^{i\theta_A} = x_A + i p_A$ . In this protocol for CV-QKD,  $x_A$  and  $p_A$  represents two independent Gaussian variables. In this protocol Alice prepares displaced coherent states with quadrature components  $x_A = |\alpha_A| \cos(\theta)$  and  $p_A = \sin(\theta)$  which are realizations of two independent and identically distributed (i.i.d.) random variables  $Q$  and  $P$ . The random variables  $Q$  and  $P$  obey to zero-centered normal distribution

$$Q \sim N(0, V_{\text{mod}_q}), \quad (6.386)$$

$$P \sim N(0, V_{\text{mod}_p}), \quad (6.387)$$

where  $V_{\text{mod}_q}$  and  $V_{\text{mod}_p}$  represents the modulation variance of  $x_A$  and  $p_A$ . This means that

$$\langle \alpha | \hat{X} | \alpha \rangle = E \{ x_A \} = 0, \quad (6.388)$$

$$\langle \alpha | \hat{Y} | \alpha \rangle = E \{ p_A \} = 0, \quad (6.389)$$

$$\langle \alpha | \hat{X}^2 | \alpha \rangle = \frac{1}{4} + E \{ x_A^2 \}, \quad (6.390)$$

$$\langle \alpha | \hat{Y}^2 | \alpha \rangle = \frac{1}{4} + E \{ p_A^2 \}, \quad (6.391)$$

$$\bar{n} = \langle \alpha | \hat{X}^2 | \alpha \rangle + \langle \alpha | \hat{Y}^2 | \alpha \rangle - \frac{1}{2} = E \{ x_A^2 \} + E \{ p_A^2 \} = V_{\text{mod}_q} + V_{\text{mod}_p}, \quad (6.392)$$

where  $E \{ \cdot \}$  represents the expectation value. Note that, the variance of a coherent state is equal to  $\bar{n}$ . This leads to

$$(\Delta X)^2 = \langle \alpha | \hat{X}^2 | \alpha \rangle = \frac{1}{4} + V_{\text{mod}_q}, \quad (6.393)$$

$$(\Delta Y)^2 = \langle \alpha | \hat{Y}^2 | \alpha \rangle = \frac{1}{4} + V_{\text{mod}_p}. \quad (6.394)$$

If we define  $N_0 = 1/4$  which represents the shot-noise variance of a single mode coherent state (identical to the fluctuation of a vacuum state), we have

$$(\Delta X)^2 = \langle \alpha | \hat{X}^2 | \alpha \rangle = N_0 \left( 1 + \frac{V_{\text{mod}_q}}{N_0} \right), \quad (6.395)$$

$$(\Delta Y)^2 = \langle \alpha | \hat{Y}^2 | \alpha \rangle = N_0 \left( 1 + \frac{V_{\text{mod}_p}}{N_0} \right). \quad (6.396)$$

Assuming that  $V_{\text{mod}_q}/N_0 = V_{\text{mod}_p}/N_0 = V_A$ , which leads to

$$Q \sim P \sim N(0, V_A), \quad (6.397)$$

$$\bar{n} = 2V_A. \quad (6.398)$$

The variance of Alice's field quadratures is

$$V = \langle x_A^2 \rangle = \langle p_A^2 \rangle = (V_A + 1)N_0 \quad (6.399)$$

The  $x_A$  and  $p_A$  components can be generated experimentally by using an amplitude modulator followed by a phase modulator. We are going now relate  $x_A$  and  $p_A$  with the transfer functions of amplitude and phase modulators. According to the Box-Muller transform,  $x_A$  and  $p_A$  can be generated from a pair of uniformly distributed random numbers ( $U_1$  and  $U_2$ ) over the interval  $[0, 1]$

$$x_A = \sqrt{-2V_A \ln(U_1)} \cos(2\pi U_2), \quad (6.400)$$

$$p_A = \sqrt{-2V_A \ln(U_1)} \sin(2\pi U_2), \quad (6.401)$$

with

$$|\alpha_A| = \sqrt{x_A^2 + p_A^2} = \sqrt{-2V_A \ln(U_1)}, \quad (6.402)$$

$$\theta_A = \arccos \left( \frac{x_A}{|\alpha_A|} \right) = 2\pi U_2, \quad (6.403)$$

where  $\theta_A$  has an uniform distribution on  $[0, 2\pi]$ , and  $|\alpha_A|$  obeys the Rayleigh distribution

$$P(|\alpha_A|) = \frac{|\alpha_A|}{V_A} e^{-\frac{|\alpha_A|^2}{2V_A}}. \quad (6.404)$$

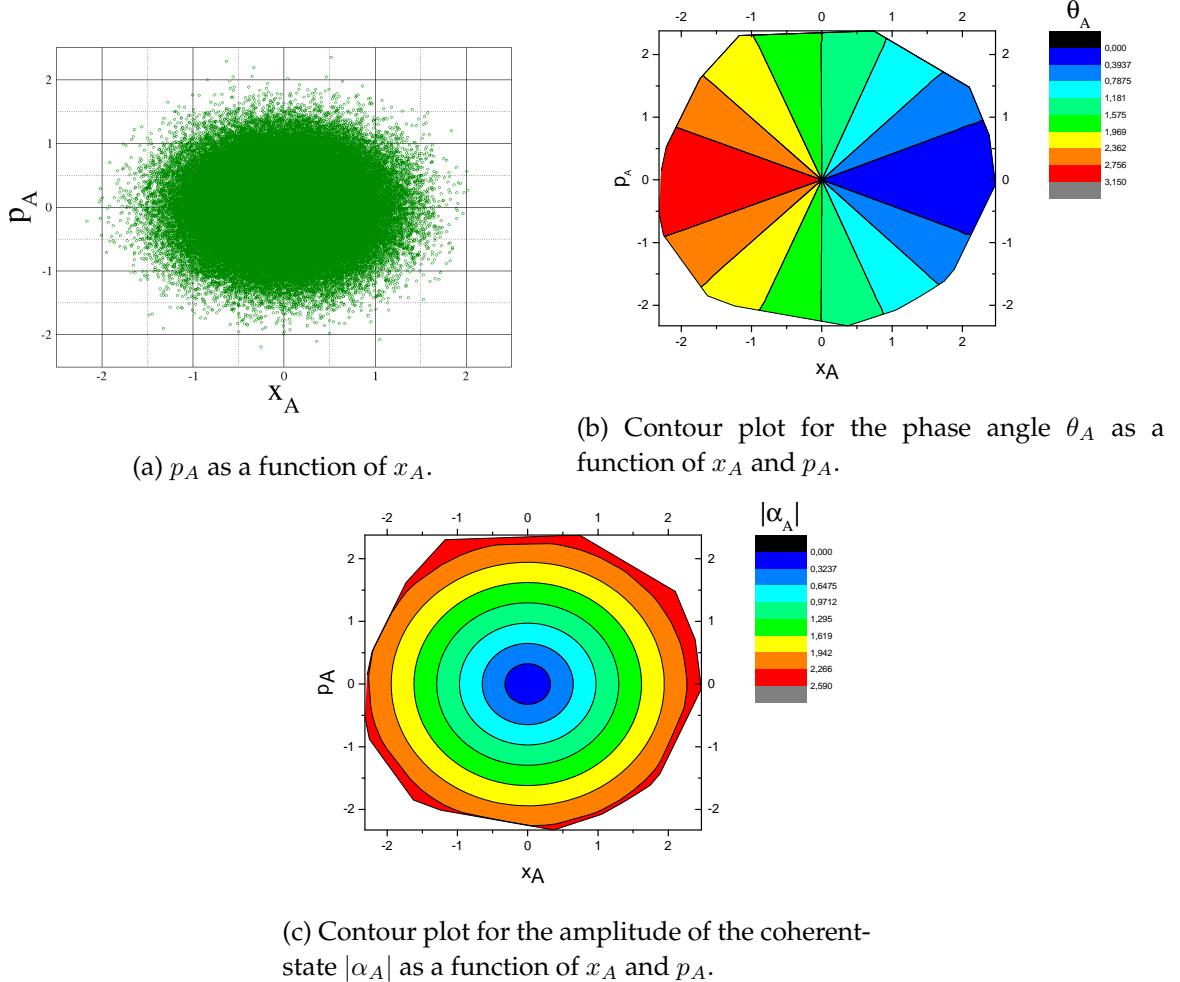


Figure 6.427: (a): Quadrature values in the phase space for a bivariate Gaussian distribution obtained using (6.400); (b) and (c): Contour plots for the phase and amplitude of the coherent-state given by (6.402). In the figure we have used  $V_A = \bar{n}/2 = 0.25$ , and we have used  $10^6$  points for generate each of the uniform distributions in (6.400).

Therefore, an amplitude modulator and a phase modulator are sufficient to achieve above bivariate Gaussian distribution.

The transfer function of a Mach-Zhender (MZ) intensity modulator followed by a phase modulator (PM) can be written as

$$T = t_{MZ}t_{PM} \sin \left( \frac{\pi}{2} \frac{V_{\min} - V_{MZ}}{V_{\pi}^{MZ}} \right) e^{i\pi V_{PM}/V_{\pi}^{PM}}, \quad (6.405)$$

where  $(1 - t_{MZ}^2)$  is the insertion loss coefficient of the MZ,  $(1 - t_{PM}^2)$  is the insertion loss coefficient of the PM,  $V_{MZ}$  is the modulation signal voltage impinged in the MZ,  $V_{\min}$  is the voltage in the MZ which corresponds to a minimum transmission of the modulator,  $V_{\pi}^{MZ}$  is the half-wave voltage. Moreover,  $V_{PM}$  is the modulation voltage impinged in the phase modulator, and  $V_{\pi}^{PM}$  is the half-voltage of the PM. From (6.400) and (6.405) the voltages impinged in the MZ and phase modulators can be written as

$$V_{PM} = 2U_2 V_{\pi}^{PM}, \quad (6.406)$$

$$V_{MZ} = V_{\min} - \frac{2}{\pi} V_{\pi}^{MZ} \arcsin \left( \frac{\sqrt{-2V_A \ln(U_1)}}{t_{MZ} t_{PM} |\alpha_{in}|} \right), \quad (6.407)$$

where  $\alpha_{in}$  is the amplitude of the coherent-state at the modulators input, with  $|\alpha_A|$  its output amplitude. The implementation of the Gaussian modulated quadrature transmitter (Alice) can be seen in Fig. 6.428

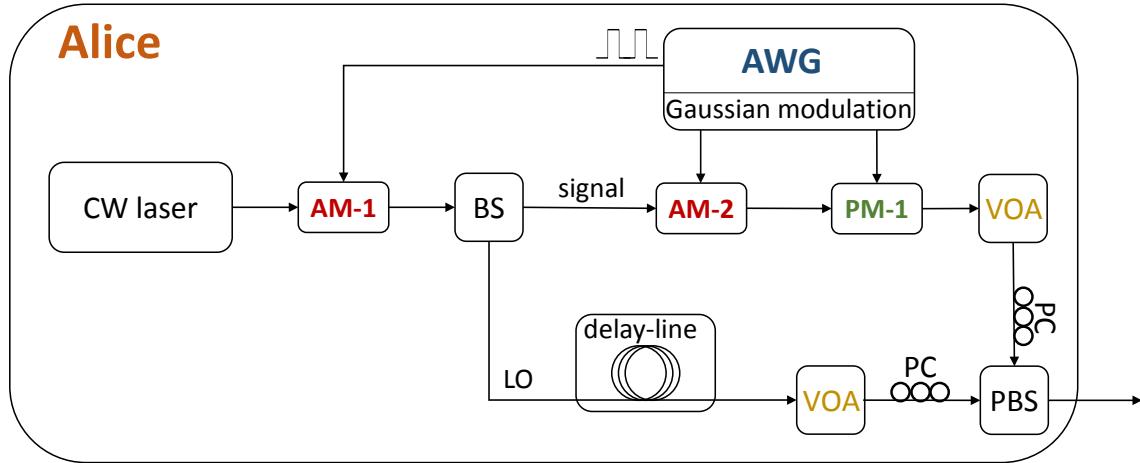


Figure 6.428: Scheme for implement a Gaussian modulated transmitter for being used in CV-QKD systems. In the figure: AM, amplitude modulator; BS, beam-splitter; PM, phase modulator; VOA, variable optical attenuator; PC, polarization controller; PBS, polarization beam-splitter; AWG, arbitrary waveform generator.

Figure 6.428 depicts the scheme for implement a Gaussian modulated coherent state transmitter. Alice uses an AM-1 with a continuous-wave laser to produce a coherent state pulse. Then she uses a beam splitter (BS) to split the coherent state into two portions, where one serves as signal and the other as local oscillator. A combination of MZ amplitude modulator (AM-2) and phase modulator (PM-1) is used to archived a bivariate Gaussian modulation, such as in 6.427. Then the Gaussian modulated signal is adjusted to an optimized variance of  $V_A$  by tuning a variable optical attenuator (VOA). The modulated signal is then combined with the local oscillator by means of a polarization beam-splitter (PBS), and both are sent to the quantum channel. Note that, by using the polarization-multiplexing and time-multiplexing techniques, the signal together with LO are sent to Bob avoiding leakage of photons from LO to the quantum signal

After propagation in the quantum channel channel the Gaussian modulated signal and local oscillator reaches the Bob receiver. Bob can perform homodyne or intradyne detection to extract information from the incoming signal and local oscillator. We are going to start with the Homodyne detection, see Fig. 6.429.

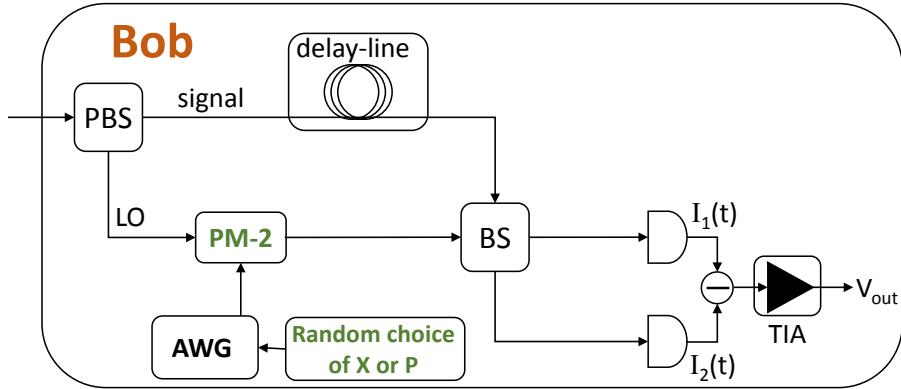


Figure 6.429: Scheme for implementing a Homodyne detection scheme for being used in CV-QKD systems. In the figure: PM, amplitude modulator; BS, beam-splitter; PM, phase modulator; VOA, variable optical attenuator; PBS, polarization beam-splitter; TIA, transimpedance amplifier.

Figure 6.429 shows a Homodyne receiver that can be used to extract the information sent by Alice. At Bob's side, the LO and the signal are polarization and time demultiplexed by using a PBS and a delay line, respectively. The LO passes through a phase modulator (PM-2) which is used to measure either  $\hat{X}$  or  $\hat{Y}$  quadratures generated by Alice. The signal and the LO will further be launched in a beam-splitter where they will interfere and after measured by two balanced photo-diodes. The photo-diodes currents  $I_1(t)$  and  $I_2(t)$  will be subtracted and amplified by means of a transimpedance amplifier, where the current intensity is amplified and converted to voltage  $V_{\text{out}}$ . We consider that the LO is a high intensity classical optical signal

$$E_{\text{LO}}(t) = \frac{1}{2} \left( A_{\text{LO}}(t) e^{i(\omega_{\text{LO}} t + \phi_{\text{LO}}(t))} + A_{\text{LO}}^*(t) e^{-i(\omega_{\text{LO}} t + \phi_{\text{LO}}(t))} \right), \quad (6.408)$$

where  $A_{\text{LO}}(t)$  is the amplitude of the LO electric field,  $\omega_{\text{LO}}$  is the optical frequency of the LO, and  $\phi_{\text{LO}}(t)$ , represents the phase of the LO. After the phase modulator PM-2 in Fig. 6.429, the LO electric field is given by

$$E_{\text{LO}}^{\text{out}}(t) = E_{\text{LO}}(t) t_{\text{PM-2}} e^{i\pi V_{\text{PM-2}}/V_{\pi}^{\text{PM-2}}} \quad (6.409)$$

### 6.21.1.2 Pilot-aided locally generated LO

The detection of the Gaussian modulated quadrature can be performed using an intradyne coherent receiver, assuming that the local oscillator (LO) is generated in the coherent receiver by using an independent laser source at the receiver's end. In that case, the main challenge

is how to effectively establish a reliable phase reference between Alice and Bob, since the quantum signal contains only few photons. While various techniques, such as feedforward carrier recovery, optical phase-locked loops, and optical injection phase-locked loops, have been developed in classical coherent communication, these techniques are not suitable in QKD where the quantum signal is extremely weak and the tolerable phase noise is low. In that scenario, we can solve the above problem by implement a pilot-aided feedforward data recovery scheme, see Fig. 6.430.

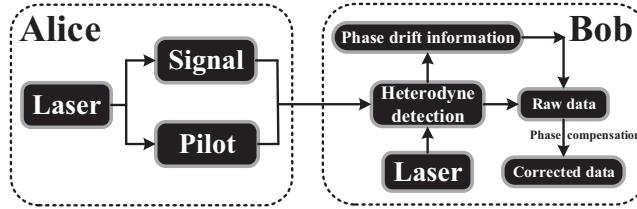


Figure 6.430: Pilot-aided locally generated LO for CV-QKD. Alice first generates the signal and the pilot, both of which are then transmitted to Bob. On the other side, Bob produces the local oscillator to interfere with the signal and the pilot. After the heterodyne (i.e. intradyne) detection, Bob obtains the raw data as well as the phase drift information, which is then utilized to execute the phase compensation yielding the corrected data.

In Fig. 6.430 it is presented the basic idea beyond the pilot-aided feedforward data recovery scheme to be used in CV-QKD systems. Nevertheless, in those systems two conditions need to be met: (1) the LO must be generated at the reception; (2) a large number of pilot pulses carrying the phase information need to be transmitted. However, the production, transmission and detection ways of the pilot pulses are optional and a different arrangement may finally affect the system performance. Three different schemes were proposed to implement a pilot-aided CV-QKD setup.

The first one is the pilot-sequential scheme, see Fig. 6.431.

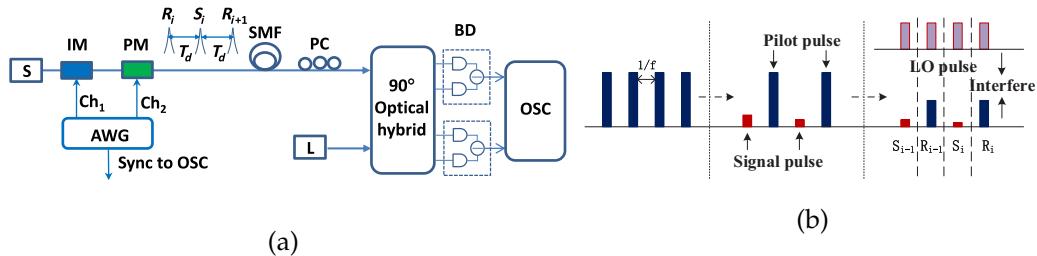


Figure 6.431: The scheme and procedure to implement a pilot-sequential setup.

To begin with, Alice produces a coherent state pulse chain, whose pulse interval is  $1/f$ . Then the odd sequence number pulses are taken as the signal and modulated according to a centered Gaussian distribution, while the even sequence number pulses are regarded as the pilot and passed without any change. At the reception, Bob produces a LO pulse

chain, whose pulse interval is also  $1/f$ . Through adjusting the optical length, the signal and pilot pulses are accurately aligned with the LO pulses. After the heterodyne (i.e. intradyne) detection, the Gaussian key information is obtained from the interference measurement of the signal pulse, while the phase drift information is contained in the interference measurement of the pilot pulse. After the data acquisition of the interference measurement, one can use the adjacent pilot phase to estimate the signal's phase drift. Since the signal pulse  $S_i$  is in the middle of two pilot pulses  $R_{i-1}$  and  $R_i$ , the relative phase drift can be estimated from the phase on  $R_{i-1}$  and  $R_i$ . However, due to the rapid phase fluctuation during the time interval, the phase estimator has an intrinsic estimation error yielding a constant phase noise.

The second one is the pilot-delay-line scheme, see Fig. 6.432.

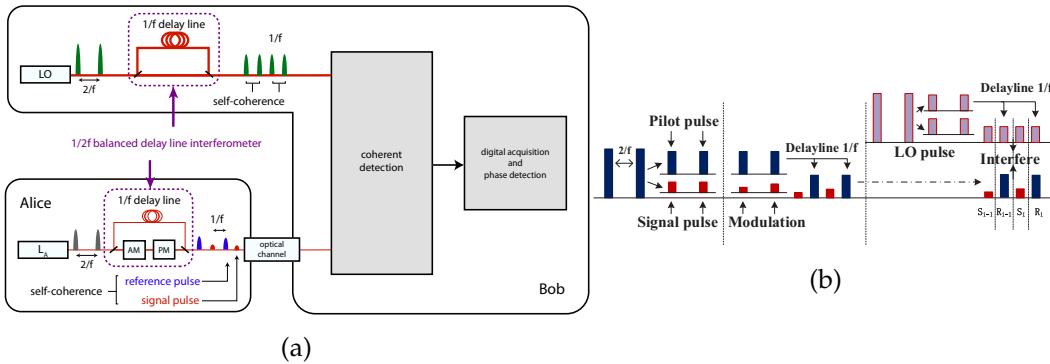


Figure 6.432: The scheme and procedure to implement a pilot-delay-line setup.

The pilot-delayline scheme essential idea is generating the signal pulse and the pilot pulse simultaneously to eliminate the phase fluctuation error. The procedure of the pilot-delayline scheme is illustrated in Fig. 6.432. First, Alice generates a coherent state pulse sequence with a  $2/f$  interval. Then she splits the coherent state into the signal pulse and the pilot pulse. The signal pulse is modulated according to a centered Gaussian distribution, while the pilot pulse is delayed for  $1/f$ , inserted into the signal pulse sequence, and transmitted to Bob. At the reception, the LO pulse sequence is also split into two identical sequences. One is delayed with a time  $1/f$  and then inserted into the other. Through aligning the LO pulses with the signal as well as the pilot pulses, the Gaussian key information and the corresponding phase information can be obtained from the interference measurement of the signal and the pilot, respectively. In the pilot-delayline scheme, since the pilot pulse is generated simultaneously with the corresponding signal pulse, both of them have the same optical phase. Therefore, one can utilize this pilot pulse as a phase monitor to execute an accurate compensation. However, due to the inherent unbalanced interferometric setup, such design requires a stable interferometric setup. Unfortunately, in the practical scenario, the transmitted optical length may fluctuate with a relatively slow rate, making this desired stability hard to maintain.

Recently, a new scheme was proposed to implement the pilot-aided CV-QKD system that solves the problems in the pilot-sequential and pilot-delay-line setups. The pilot-

multiplexed scheme introduces time- and polarization multiplexing techniques and uses two heterodyne (i.e. intradyne) receivers to detect the signal and pilot separately, see Fig. 6.433.

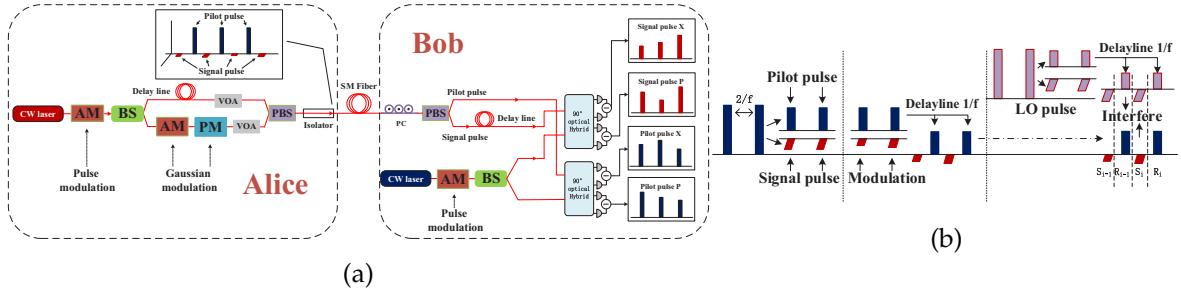


Figure 6.433: The scheme and procedure to implement a pilot-multiplexed setup.

The procedure of the pilot-multiplexed scheme is illustrated in Fig. 6.433. Specifically, the signal pulse and the pilot pulse are also derived from a single pulse to make their optical phases identical. After the signal modulation, Alice delays the reference path for  $1/f$  to realize the time-multiplexing. Besides, orthogonal polarization channels are occupied to transmit the signal pulse and the pilot pulse realizing the polarization multiplexing. At the reception, these polarization-multiplexing pulses are divided and sent to different heterodyne (i.e. intradyne) detectors. Meanwhile, the LO pulse is split into two identical pulses, aligned with its corresponding signal pulse and pilot pulse, and interfered with them. The interference measurements of the signal as well as the pilot contain the key information and the phase drift information, respectively. In this scheme, we can divide the phase drift into the fast-drift part and slow-drift part, thus one can execute two remaps to compensate them separately.

The pilot-multiplexed scheme illustrated in Fig. 6.433 can also be used to implement a CV-QKD system with homodyne detection instead of intradyne, see Fig. 6.434

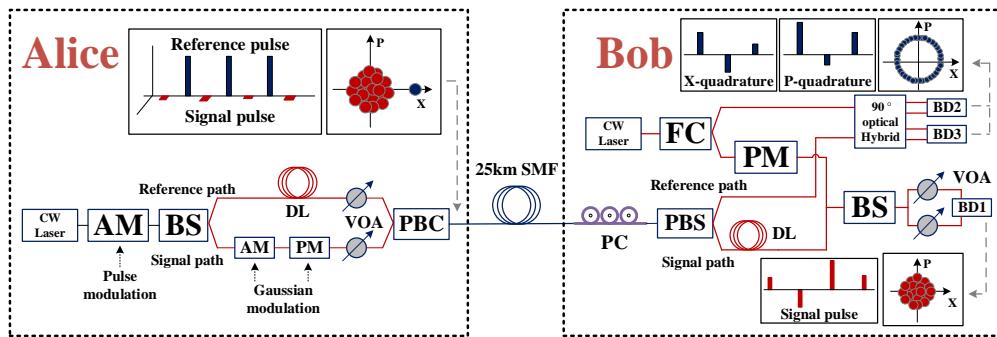


Figure 6.434: The scheme to implement a pilot-multiplexed setup using homodyne detection for the quantum signal.

When compared with 6.433, the scheme in Fig. 6.434 uses a beam-splitter and a pair of photodiodes instead an optical hybrid and four photodiodes. Nevertheless, to random basis

switch scheme in Fig. 6.434 uses a phase modulator in the LO arm.

We are going now to analyze the performance and the security of the pilot-aided CV-QKD system for the Gaussian modulated protocol. After modulation, Alice transmit to Bob the coherent state  $|\alpha_A\rangle = |x_A + ip_A\rangle$  to Bob through a quantum channel with transmission efficiency

$$T_{\text{ch}} = 10^{-\frac{\alpha_{\text{ch}} L_f}{10}}, \quad (6.410)$$

with  $\alpha_{\text{ch}}$  being the channel loss coefficient(in dB/km), and  $L_f$  the channel length (in km). The channel loss gives rise to a quantum noise coefficient  $(1 - T_{\text{ch}})/T_{\text{ch}}$ . Moreover, after propagation in the quantum channel the field quadratures sent by Alice can contain other noise sources, for instance from other classical or quantum signals in the same fiber, or from imperfections on Alice side (modulators and laser source). All these noise source are grouped in the same noise parameter  $\xi_A$  (typically known as excess of noise), such that at the end of the transmission channel the total added noise added by the channel is given by

$$\chi_{\text{ch}} = \frac{1 - T_{\text{ch}}}{T_{\text{ch}}} + \xi_A. \quad (6.411)$$

In addition to the obligatory shot-noise and the propagation noise induced by the quantum channel, various experimental imperfections will contribute to the total noise and undermine the system's performance. The constituents of  $\xi_A$  may originate from Alice modulation imperfections, from nonlinear out-of-band linear and nonlinear effects that generate noise at signal band (such ASE and Raman noise), from phase fluctuations due to the use of two laser sources, from quantisation at the ADCs, from detection noise (referred typically as receiver shot-noise which in this case represents the current fluctuations due to the discreteness of the electrical charge), from the imperfections on the balancing the photodiodes (known as common-mode rejection rate), from imperfections on the optical hybrid, among others. Assuming that all of these noise sources to be stochastically independent which makes the variances they cause to the quadratures additive

$$\xi_A = \xi_{\text{Mod}} + \xi_{\text{ON}} + \xi_{\text{Ph}} + \xi_{\text{ADC}} + \xi_{\text{det}} + \xi_{\text{CMMR}} + \xi_{\text{OH}} + \dots \quad (6.412)$$

Note that, the subscript  $A$  in  $\xi_A$  is referring to the channel input. We refer to this additional noise as excess noise  $\chi_{\text{det}}$ .

Finally, the homodyne or intradyne receiver it self introduces noise due to the fact that is not ideal due to the non-zero temperature, thermal fluctuations are an unavoidable source of noise

$$\chi_{\text{Hom}} = \frac{1 - \eta_{\text{det}}}{\eta_{\text{det}}} + \frac{v_{\text{el}}}{\eta_{\text{det}}}, \quad (6.413)$$

$$\chi_{\text{Het}} = \frac{2 - \eta_{\text{det}}}{\eta_{\text{det}}} + \frac{2v_{\text{el}}}{\eta_{\text{det}}}, \quad (6.414)$$

where  $\eta_{\text{det}}$  represents the homodyne or intradyne detector efficiency, and  $v_{\text{el}}$  represents the receiver electronic noise. Note that  $\chi_{\text{Hom}}$  or  $\chi_{\text{Het}}$  are referring to the channel output. The

total noise is given by

$$\chi = \chi_{\text{ch}} + \frac{\chi_{\text{det}}}{T_{\text{ch}}}, \quad (6.415)$$

with  $\chi_{\text{det}} = \chi_{\text{Hom}}$  for homodyne detection, and  $\chi_{\text{det}} = \chi_{\text{Het}}$  for heterodyne (intradyne) detection. Note that  $\chi_{\text{det}}$  is divided by  $T_{\text{ch}}$  in order to refer to the channel input.

After transmission over the loss and noisy channel, Bob will measure a quadrature variance

$$V_B = T_{\text{ch}}\eta_{\text{det}}(V + \chi N_0), \quad (6.416)$$

where  $V = V_A + 1$  as previously defined. This means that Bob will measure an attenuated and noise low intensity quantum signal sent by Alice.

In order to access to the performance of the CV-QKD system, we first calculate the secure key rate per pulse under the finite-size analysis

$$K_p = \left(1 - \frac{m}{N}\right) [\beta I_{\text{BA}} - I_{\text{BE}} - \Delta(n)], \quad (6.417)$$

where  $N$  is total number of signals exchanged to establish the key, and  $m$  are the number of signals used to estimate  $T_{\text{ch}}$  and  $\xi_A$ . This indicates that  $n = N - m$  represents are the number of signals used for establishment of the key. In (6.417),  $\beta$  is the efficiency of the reverse reconciliation protocol used for Alice and Bob extract their mutual information (including error correction). Moreover,  $I_{\text{BA}}$  represents the Shannon mutual information between Bob and Alice, and  $I_{\text{BE}}$  is the Shannon mutual information between Bob and Eve. Finally in (6.417),  $\Delta(n)$  is related with the security of the privacy amplification procedure. The mutual information in (6.417) depends on the kind of receiver that Bob uses, i.e.  $I_{\text{BA}} = I_{\text{BA}}^{\text{Hom}}$  if he uses a homodyne receiver such as in Fig. 6.434, and  $I_{\text{BA}} = I_{\text{BA}}^{\text{Het}}$  if Bob uses an intradyne detector such as in Fig. 6.433. The mutual information for those detection schemes between Bob and Alice are given by

$$I_{\text{BA}}^{\text{Hom}} = \frac{1}{2} \log_2 \left( \frac{V_B^\pm}{V_{A|B}^\pm} \right), \quad (6.418)$$

$$I_{\text{BA}}^{\text{Het}} = \frac{1}{2} \log_2 \left( \frac{V_B^+}{V_{A|B}^+} \right) + \frac{1}{2} \log_2 \left( \frac{V_B^-}{V_{A|B}^-} \right), \quad (6.419)$$

where we assume  $V_B^+$  represents the Bob variance associate to the  $x_A$  quadrature component sent by Alice,  $V_B^-$  represents the Bob variance associate to the  $p_A$  quadrature component sent by Alice. Moreover,  $V_{A|B}^\pm$  is the Alice conditional variance of Bob measurement, given by

$$V_{A|B}^\pm = V_B^\pm - \frac{|\langle S_A^\pm X_B^\pm \rangle|^2}{V_S^\pm}, \quad (6.420)$$

where  $V_S^\pm$  is the quadrature variance generate by Alice for the  $x_A$  and  $p_A$  quadrature components, and  $\langle S_A^\pm X_B^\pm \rangle$  is correlation coefficient between the Alice prepared state and the Bob measurement state. The coefficients in (6.420) are slightly different in the case of homodyne or intradyne detection.

### 1. Homodyne detection

$$S_A^+ = x_A = \left( \hat{X} - \hat{X}_v^+ \right), \quad (6.421)$$

$$S_A^- = p_A = \left( \hat{Y} - \hat{X}_v^- \right), \quad (6.422)$$

$$V_S^\pm = \langle (S_A^\pm)^2 \rangle = V_A N_0, \quad (6.423)$$

$$X_B^+ = \sqrt{T_{\text{ch}} \eta_{\text{det}}} (x_A + X_N^+), \quad (6.424)$$

$$X_B^- = \sqrt{T_{\text{ch}} \eta_{\text{det}}} (p_A + X_N^-), \quad (6.425)$$

$$V_B^\pm = \langle (X_B^\pm)^2 \rangle = T_{\text{ch}} \eta_{\text{det}} (V + \chi N_0), \quad (6.426)$$

$$\langle x_A^2 \rangle = \langle p_A^2 \rangle = (V_A + 1) N_0, \quad (6.427)$$

$$\langle (X_N^\pm)^2 \rangle = \chi N_0, \quad (6.428)$$

$$\langle X_A^\pm X_B^\pm \rangle = \sqrt{T_{\text{ch}} \eta_{\text{det}}} V_A N_0, \quad (6.429)$$

where  $X_v^\pm$  represents the quadrature operators for the vacuum state, and  $X_N^\pm$  represents noise added to the channel. In this case, the mutual information between Bob and Alice is

$$I_{\text{BA}}^{\text{Hom}} = \frac{1}{2} \log_2 \left( \frac{T_{\text{ch}} \eta_{\text{det}} (V_A + 1 + \chi) N_0}{T_{\text{ch}} \eta_{\text{det}} (1 + \chi) N_0} \right) \quad (6.430)$$

$$= \frac{1}{2} \log_2 \left( 1 + \frac{V_A}{1 + \chi} \right), \quad (6.431)$$

with  $\chi = \chi_{\text{ch}} + \chi_{\text{Hom}} / T_{\text{ch}}$ , and  $\chi_{\text{Hom}} = (1 - \eta_{\text{det}}) / \eta_{\text{det}} + v_{\text{el}} / \eta_{\text{det}}$ .

2. Intradyne detection. In this case, the quantum signal entering at Bob detection system, passes through an extra beam-splitter before being measured by two homodyne detectors. This extra beam-splitter leads to a decrease in the detection efficiency of  $1/\sqrt{2}$  and an extra noise factor  $\hat{N}_B^\pm$ . This leads to

$$S_A^+ = x_A = \left( \hat{X} - \hat{X}_v^+ \right), \quad (6.432)$$

$$S_A^- = p_A = \left( \hat{Y} - \hat{X}_v^- \right), \quad (6.433)$$

$$V_S^\pm = \langle (S_A^\pm)^2 \rangle = V_A N_0, \quad (6.434)$$

$$X_B^+ = \sqrt{\frac{T_{\text{ch}} \eta_{\text{det}}}{2}} (x_A + X_N^+ + \hat{N}_B^+), \quad (6.435)$$

$$X_B^- = \sqrt{\frac{T_{\text{ch}} \eta_{\text{det}}}{2}} (p_A + X_N^- + \hat{N}_B^-), \quad (6.436)$$

$$V_B^\pm = \langle (X_B^\pm)^2 \rangle = \frac{T_{\text{ch}} \eta_{\text{det}}}{2} (V + \chi N_0), \quad (6.437)$$

$$\langle x_A^2 \rangle = \langle p_A^2 \rangle = (V_A + 1) N_0, \quad (6.438)$$

$$\langle (X_N^\pm + \hat{N}_B^\pm)^2 \rangle = \chi N_0, \quad (6.439)$$

$$\langle X_A^\pm X_B^\pm \rangle = \sqrt{\frac{T_{\text{ch}} \eta_{\text{det}}}{2}} V_A N_0. \quad (6.440)$$

The mutual information between Bob and Alice for the intradyne detection is

$$I_{BA}^{\text{Het}} = \log_2 \left( \frac{T_{\text{ch}}\eta_{\text{det}}(V_A + 1 + \chi)N_0}{T_{\text{ch}}\eta_{\text{det}}(1 + \chi)N_0} \right) \quad (6.441)$$

$$= \log_2 \left( 1 + \frac{V_A}{1 + \chi} \right), \quad (6.442)$$

with  $\chi = \chi_{\text{ch}} + \chi_{\text{det}}/T_{\text{ch}}$  with  $\chi_{\text{Het}} = (2 - \eta_{\text{det}})/\eta_{\text{det}} + 2v_{\text{el}}/\eta_{\text{det}}$ .

For the mutual information's between Bob and Eve the conditional variances are much more complex to obtain, and we will not derive the result here, we will present only the final result.

### 1. Homodyne detection

$$V_{B|E}^{\pm} = \eta_{\text{det}} \left[ \frac{1}{T_{\text{ch}}((V_A + 1)^{-1} + \chi_{\text{ch}})} + \chi_{\text{Hom}} \right] N_0. \quad (6.443)$$

This leads to mutual information between Bob and Eve equal to

$$I_{BE}^{\text{Hom}} = \frac{1}{2} \log_2 \left( \frac{V_B^{\pm}}{V_{B|E}^{\pm}} \right) \quad (6.444)$$

$$= \frac{1}{2} \log_2 \left( \frac{T_{\text{ch}}^2(V_A + 1 + \chi_{\text{ch}} + \chi_{\text{Hom}}/T_{\text{ch}})((V_A + 1)^{-1} + \chi_{\text{ch}})}{1 + T_{\text{ch}}\chi_{\text{Hom}}((V_A + 1)^{-1} + \chi_{\text{ch}})} \right) \quad (6.445)$$

### 2. Intradyne detection

$$V_{B|E}^{\pm} = \frac{\eta_{\text{det}}}{2} \left[ \frac{(V_A + 1)\chi_E + 1}{V_A + 1 + \chi_E} + \chi_{\text{Het}} \right] N_0, \quad (6.446)$$

where

$$\chi_E = \frac{T_{\text{ch}}(2 - \xi_A)^2}{(\sqrt{2 - 2T_{\text{ch}}} + T_{\text{ch}}\xi_A + \sqrt{\xi_A})^2} + 1. \quad (6.447)$$

This leads to an mutual information between Bob and Eve equal to

$$I_{BE}^{\text{Het}} = \log_2 \left( \frac{V_B^{\pm}}{V_{B|E}^{\pm}} \right) \quad (6.448)$$

$$= \log_2 \left( \frac{T_{\text{ch}}(V_A + 1 + \chi_{\text{ch}} + \chi_{\text{Het}}/T_{\text{ch}})(V_A + 1 + \chi_E)}{(V_A + 1)\chi_E + 1 + \chi_{\text{Het}}(V_A + 1 + \chi_E)} \right). \quad (6.449)$$

### 6.21.1.3 Discrete-Modulated Coherent States

### 6.21.2 Simulation Analysis

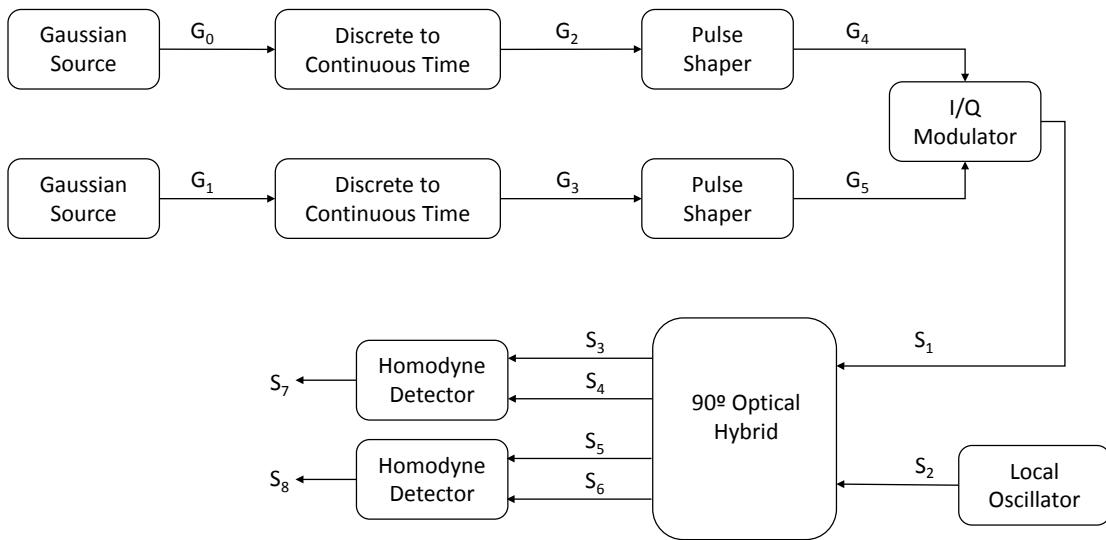


Figure 6.435: Overview of the simulated continuous variable intradyne QKD system.

### 6.21.3 Experimental Analysis

### 6.21.4 Comparative Analysis

## 6.22 Classical Multi-Party Computation

### 6.22.1 Introduction

Multi-Party Computation (MPC), also known as Secure Function Evaluation, allows two or more parties to correctly compute a function of their private inputs without exposure, i.e., without the input of one party being revealed to the other parties. In other terms, a generic function  $f$  receives as input a set  $\{a_1, a_2, \dots, a_n\}$  of arguments, where  $a_i$  is the input of the  $i$ -th party, and  $1 \leq i \leq n$ , and outputs a value  $c$ , which represents the result of the joint computation of  $f$ , as shown in Figure 6.436. The output of  $f$  is given by the following expression.

$$c = f(a_1, a_2, \dots, a_n) \quad (6.450)$$

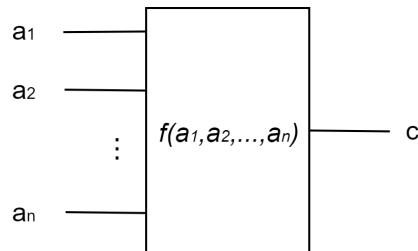


Figure 6.436: Multi-Party Computation Diagram

There are two main models in the literature for the analysis of the Multi-Party Computation problems. The ideal model, which consists in using a Trusted Third Party (TTP). The parties provide their input to the TTP, who will perform the computation of the function. The result is then sent to all the parties. This paradigm relies on the trustworthiness of the TTP because if it turns corrupt, it can supply the private input of one party to the others. This model is extensively used due to its easy implementation and protocols available which prevent the TTP from acting maliciously. The real model of MPC does not use a TTP. In this model, the parties agree on some protocol which will allow them to jointly compute the function. Different protocols are needed for different MPC models and different party behavior models. In 6.22.4 we will be analyzing different solutions to known MPC problems.

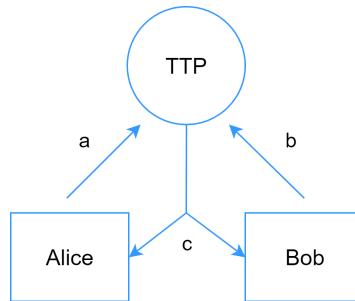


Figure 6.437: MPC using a Third Trusted Party

### 6.22.2 Two-Party Computation

Two-Party Computation (2PC) is a specific case of MPC, where a generic function  $f$  receives as input a set  $\{a, b\}$  of arguments, where  $a$  is the input from the first party and  $b$  is the input from the second, and outputs a value  $c$ , as shown in Figure 6.438. The output of  $f$  is given by the following expression.

$$c = f(a, b) \quad (6.451)$$

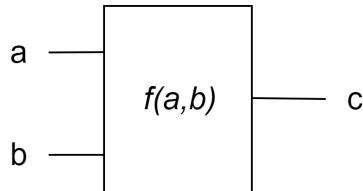


Figure 6.438: Two-Party Computation Diagram

### 6.22.3 Party Behavior Models

There are three main models to describe the behavior of a party. The honest model, where the party follows the protocol and respects the privacy of other parties. A semi honest model, where the party follows the protocol but also tries to gain additional information other than the result. The corrupt model, where the party neither follows the protocol nor respects the privacy of other parties.

### 6.22.4 MPC Problems and Solutions

In this subsection, we will be analyzing different solutions to known MPC problems without the presence of a TTP. We will present a problem and then analyze various solution.

#### 6.22.4.1 The Millionaires' Problem

Consider 2 parties, Alice and Bob, with inputs  $a$  and  $b$  respectively. The output of this function is 1 when  $a < b$  and 0 when  $a \geq b$ . In other terms,

$$f(a, b) = \begin{cases} 0, & \text{if } a \geq b \\ 1, & \text{if } a < b \end{cases}$$

**Yao's Solution** This method was proposed by Andrew C. Yao in 1982 [1]. Let  $E_a$  be Alice's public key,  $E_a(x)$  the process of encrypting input  $x$  by performing a bitwise XOR between  $x$  and Alice's public key and  $D_a(x)$  the process of decrypting input  $x$ . For this example, we will assume the following values:  $a = 7$ ,  $b = 3$ ,  $N = 16$ ,  $x = 39226$ ,  $p = 211$  and  $E_a = 24698$ .

1. Bob picks a random N-bit integer,  $x$ , and computes privately the value of  $E_a(x)$ ; calls the result  $k$ .

$$k = E_a(x) = x \oplus E_a = 63808 \quad (6.452)$$

2. Bob sends Alice the number  $k - b + 1$

$$k - b + 1 = 63806 \quad (6.453)$$

3. Alice computes privately the values of  $Y_u = D_a(k - b + u)$  for  $u = 1, 2, \dots, 10$

$$Y_u = [39236, 39237, 39226, 39227, 39224, 39225, 39230, 39231, 39228, 39229]$$

4. Alice generates a random prime  $p$  of  $N/2$  bits, and computes  $Z_u = Y_u \bmod p$

$$Z_u = [201, 202, 191, 192, 189, 190, 195, 196, 193, 194]$$

5. Alice sends the prime  $p$  and the following 10 numbers to Bob:  $Z_1, Z_2, \dots, Z_a$  followed by  $Z_{a+1} + 1, \dots, Z_{10} + 1$

$$M = [201, 202, 191, 192, 189, 190, 195, 197, 194, 195]$$

Note that only the elements of  $Z_u$  and  $M$  with indexes 7, 8, 9 and 10 are different, since  $a = 7$

6. Bob looks at the  $b$ -th number sent by Alice, and decides that  $a \geq b$  if it is equal to  $x \bmod p$ , and  $a < b$  otherwise. In other terms,

$$f(a, b) = \begin{cases} 0, & \text{if } M_b = x \bmod p \\ 1, & \text{if } M_b \neq x \bmod p \end{cases}$$

Since  $M_b = x \bmod p = 191$ , we have  $f(a, b) = 0$ , which means that  $a \geq b$ .

#### 6.22.4.2 1-2 Oblivious Transfer

Consider 2 parties, Alice and Bob. Alice has a set of messages  $A = \{m_0, m_1, m_2, \dots, m_{n-1}\}$  and Bob has an index  $b$ . The output of this function should be  $m_b$ , i.e., the message from Alice's set of index  $b$ . Bob should not gain any more information other than  $m_b$  and Alice should not know the value of  $b$ .

$$f(A, b) = A_b \quad (6.454)$$

<<<< HEAD

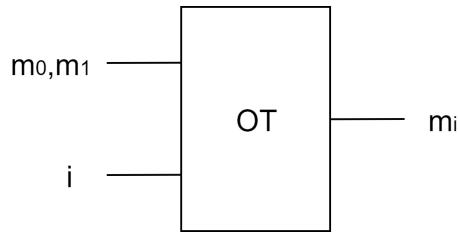


Figure 6.439: Oblivious Transfer Diagram

#### 6.22.4.3 Average Value

Consider 2 parties, Alice and Bob, with inputs  $a$  and  $b$  respectively. The output of this function should be the average value of  $a$  and  $b$ . In other terms,

$$f(a, b) = \frac{a + b}{2} \quad (6.455)$$

#### 6.22.5 Garbled Circuit Protocol

Introduced in 1986 by Andrew Yao, the Garbled Circuit protocol (GC) addresses the case of Two-Party Computation (2PC), without the presence of a trusted third party. GC allows a secure evaluation of a function given as a Boolean circuit that is represented as a series of logic gates. The circuit is known to both parties.

#### 6.22.6 Hardware Description Languages

Contrary to Programming Languages such as C or C++, which are used to specify a set of instructions to a computer, Hardware Description Languages (HDL) are computer languages used to describe the structure and behavior of digital logic circuits. They allow for the synthesis of HDL description code into a netlist (specification of physical electronic components, such as AND gates or NOT gates, and how they are connected together).

#### 6.22.7 TinyGarble

TinyGarble is a GC framework that takes advantage of powerful logic synthesis techniques, provided by both HDL synthesis tools and TinyGarble's custom libraries, in order to improve

the overall efficiency of the GC protocol. It is possible to describe the circuit using High-Level Programming Languages (HLPL) such as C, although High-Level Synthesis (HLS) is required. HLS is performed by High-Level Synthesis tools, such as SPARK for the C language.

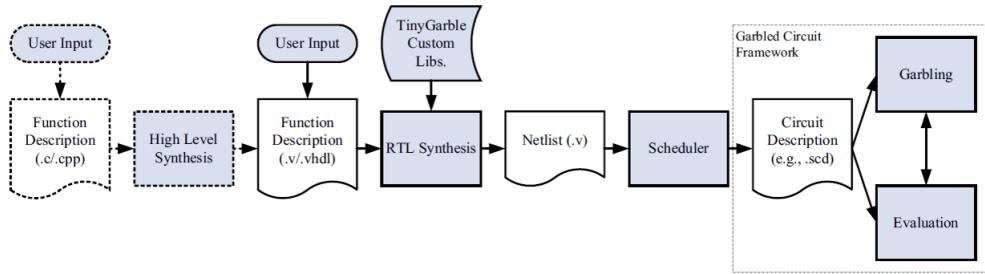


Figure 6.440: TinyGarble Flow Diagram

### 6.22.8 ARM2GC

Although circuit description in HLPL is possible, it is not very efficient when compared to HDL circuit description. ARM2GC addresses this problem, significantly improving the performance of garbled circuits described in HLPL.

In the case of 2PC, ARM2GC's approach to GC is based on the ARM processor architecture and consists in providing a public parameter to function  $f$ , so that its output would be given by the following expression.

$$z = f(a, b, p) \quad (6.456)$$

, where  $p$  represents a public parameter, known to both parties.

In ARM2GC the Boolean circuit required to perform GC is that of a processor to which the compiled binary of the function is given as a public input ( $p = \text{compiled binary of the function}$ ). This optimization is performed by the SkipGate algorithm.

## References

- [1] Andrew C. Yao. "Protocols for Secure Computations". In: - (1982).

## 6.23 Quantum Multi-Party Computation

### 6.23.1 Our Approach

#### 6.23.1.1 1 out of 2 Oblivious Transfer

This section will provide the description of a 1-2 Oblivious Transfer (OT) based on the appliance of a Quantum Oblivious Key Distribution Protocol (QOKD). The 1-2 OT consists in a two party, Alice and Bob, communication protocol. Supposing that Alice has two messages  $\{m_1, m_0\}$  length  $s$ , Bob wants to know one of those in such a way that:

- Alice doesn't know Bob's choice, i.e. the protocol is oblivious;
- Bob doesn't get any information on the message he didn't choose, i.e. the protocol is concealing.

Considering the notation of a canonical quantum oblivious transfer protocol, let  $U = \{+, \times\}^n \times \{0, 1\}^n$ , where  $+, \times$  stand for the rectilinear and diagonal bases, with a correspondence previously agreed by both Bob and Alice. Physically this corresponds to the general algorithm of the protocol can be described as:

- Step 1:  
Alice picks a random uniformly chosen  $(a, g) \in U$ , and sends Bob photons  $i, 1 \leq i \leq n$  with polarizations given by the bases  $a[i]$  and states  $g[i]$ .
- Step 2:  
Bob picks a random uniformly chosen  $b \in \{+, \times\}^n$ , measures photons  $i$  in basis  $b[i]$  and records the results, if a photon is detected, as  $h[i] \in \{0, 1\}$ . Bob then makes a bit commitment of all  $n$  pairs  $(b[i], h[i])$  to Alice.
- Step 3:  
Alice picks a random uniformly chosen subset  $R \subset \{1, 2, \dots, n\}$  and tests the commitment made by Bob at positions in  $R$ . If more  $\delta n$  (acceptance threshold) positions  $i \in R$  reveal  $a[i] = b[i]$  and  $g[i] \neq h[i]$  then Alice stops protocol; otherwise, the test result is accepted.
- Step 4:  
Alice announces the base  $a$ . Let  $T_0$  be the set of  $1 \leq i \leq n$  such that  $a[i] = b[i]$  and let  $T_1$  be the set of all  $1 \leq i \leq n$  such that  $a[i] \neq b[i]$ . Bob chooses  $I_0, I_1 \subset T_0 - R, T_1 - R$  and sends  $S_i = \{I_{1-i}, I_i\}$ , wishing to know  $m_i, i \in \{0, 1\}$ .
- Step 5:  
Alice defines two encryption keys  $K_0, K_1$  in such a way that  $K_i = g[I_i]$  for  $i = 0 \vee i = 1$ . Alice then cyphers both messages:  $m_{\text{coded}} = \{m_0 \oplus K_0, m_1 \oplus K_1\}$  and sends the result  $m$  to Bob.
- Step 6:  
Bob will then decode  $m$  using the values of his initially chosen basis:  $b[S_i]$  with  $i \in$

$\{0, 1\}$ , according to his preference.  $m_{\text{decoded}} = m_{\text{coded}} \oplus b [S_i]$ . The output of this process will be  $m_{\text{decoded}}$  that will have the correct message in the first or last  $s^{\text{th}}$  positions if he chose  $m_0$  or  $m_1$ , respectively.

It is intuitively clear that the above protocol performs correctly if both parties are honest [1]. The security of protocol depends, though on the honesty of both parties (and a potential eavesdropper) involved. The security of the protocol can be evaluated in terms of the amount of information received by any given participant. In order to formalize and proof the security of such a system for any case though one has to think of the proceedings of a hypothetically dishonest Bob and an eventual eavesdropper Eve.

- Step 1:  
Dishonest Bob has no advantage in being dishonest at this point.
- Step 2:  
Dishonest Eve transfers some information from this pulse into her quantum system and she uses that information to modify the residual state of the pulse which is sent to Bob.  
Dishonest Bob executes a coherent measurement on the pulse received in order to determine: whether or not he declares this pulse as detected and the bit that he commits to Alice.
- Step 4:  
Having learnt Alice's string of basis  $a$  dishonest Bob executes a first post-measurement of his choice and uses the outcome to compute the ordered pair  $S$ .
- Step 5:  
Using the information obtained in the previous step dishonest Bob makes a second post-test measurement and obtains the outcome  $\mathcal{J}_{Bob}$ . Eve measures her system and obtains the outcome  $\mathcal{J}_{Eve}$  [2].

From [2] one can concluded that a dishonest Bob following these proceedings learns nothing about  $m$ , the set of both original messages concatenated, in its full extended, either he passes or fails Alice's original verification. This protocol also compensates the errors in the quantum channel. It is also stated that security against Bob and tolerance against errors implies the security of the protocol against Eve [2].

#### 6.23.1.2 Comparison Protocol

The millionaire problem was originally a two-party secure computation problem, in which two millionaires, Alice and Bob, want to know which of them is richer without revealing their actual wealth. It is analogous to a more general problem whose goal is to compare two numbers  $a$  and  $b$ , without revealing any extra information on their values other than what can be inferred from the comparison result. Boudot proposed a protocol to solve said. However, Lo pointed out that the equality function cannot be securely evaluated with a two-party scenario. Therefore, some additional assumptions should be considered to reach the goal of private comparison, a quantum comparison protocol (QPC) always needs:

- An at least semi-honest Third Party (TP) is required to help the two parties (Alice and Bob) accomplish the comparison. A semi-honest TP is a party who always follows the procedure of the protocol recording all of intermediate computations and despite not being corrupted by an outside eavesdropper, TP might try to steal the information from the record. TP will know the positions of different bit value in the compared information, but it will not be able to know the actual bit value of the information.
- All outsiders and the two players should only know the result of the comparison, but not the different positions of the information.
- To guarantee the security of private information, it is better to compare several bits instead of one bit at a time.

Since Yao's initial proposal, several QKD protocols using Einsteinâ€“Podolskyâ€“Rosen (EPR) pairs have been proposed in previous work to achieve secret communication for two communicants. These QKD protocols attempt to use the correlation of EPR pairs to distribute a common shared key for two mutually trusted users. However, EPR pairs in the proposed QPC protocol are used to create two individual keys for each of two mutually suspicious users and at the same time allow a semi-honest third party to perform the comparison without knowing the secret content of their information. Therefore, a QKD protocol may not be able to directly solve the QPC problem.

## References

- [1] Andrew Chi-Chih Yao. "Security of quantum protocols against coherent measurements". In: *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing - STOC '95*. ACM Press, 1995. DOI: [10.1145/225058.225085](https://doi.org/10.1145/225058.225085). URL: <https://doi.org/10.1145/225058.225085>.
- [2] Dominic Mayers. "On the Security of the Quantum Oblivious Transfer and Key Distribution Protocols". In: *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology. CRYPTO '95*. London, UK, UK: Springer-Verlag, 1995, pp. 124–135. ISBN: 3-540-60221-6. URL: <http://dl.acm.org/citation.cfm?id=646760.705993>.

## 6.24 Secure Multi-Party Computation

|                      |   |   |
|----------------------|---|---|
| <b>Students Name</b> | : | Armando Pinto (01/06/2018 - )<br>: Goncalo Vitor (15/06/2018 - )<br>: Mariana Ramos (02/05/2018 - 21/05/2018) |
| <b>Goal</b>          | : | Description of the problem of Secure Multi-Party Computation.   |
| <b>Directory</b>     | : |   |

### 6.24.1 Problem definition

In *Secure Multi-Party Computation* (SMC) the function to be computed is defined as  $(y_1, y_2, y_3, y_4, \dots, y_N) = f(x_1, x_2, x_3, x_4, \dots, x_N)$  and each party  $i$  must only know (before and after the computation) its  $x_i$  and  $y_i$  being unknown any information about the input or output of the other parties. [1].

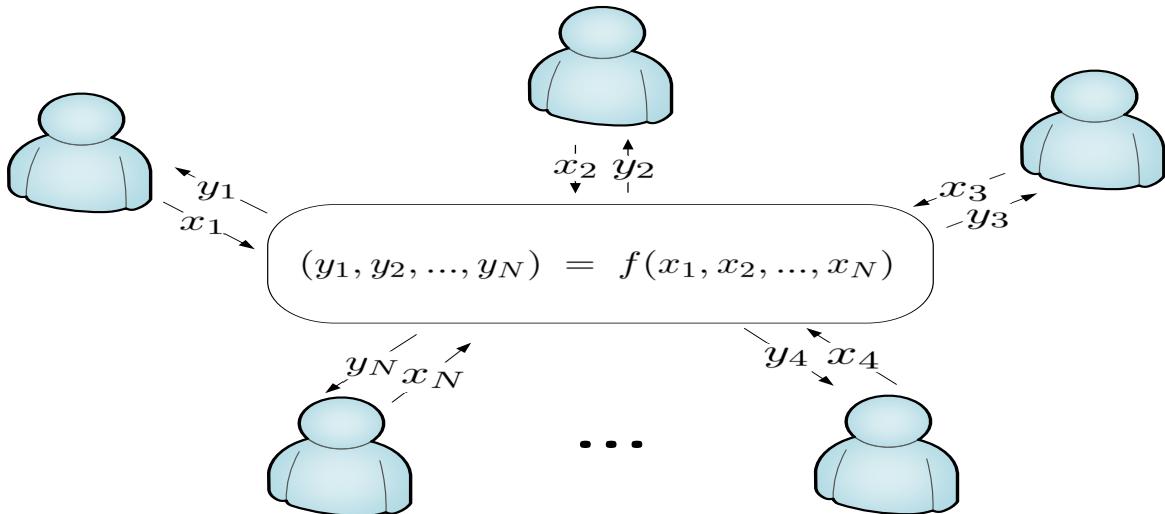


Figure 6.441:  $N$  entities are able to jointly compute  $(y_1, y_2, y_3, y_4, \dots, y_N) = f(x_1, x_2, x_3, x_4, \dots, x_N)$  making sure that each entity only have access its input/output pair, i.e.  $(x_i, y_i)$ .

fig:smcN

Thus, there are two ways of solving the problem of SMC:

1. The result can be computed using a trusted party involved who has the trust from all the parties. This is a simple solution for the current problem, however, it requires a trusted party.
2. The result can be computed without a trusted party involved using a SMC protocol.

We are going to consider the *Garbled Circuit Protocol* (GCP), which will have our best attention in this chapter.

We will only focus on a two-party computation ( $N = 2$ ), with a single output, i.e.  $y = f(x_1, x_2)$ .

### 6.24.2 Secure Two-Party Computation

The *Garbled Circuit Protocol* (GCP) starts with a transformation of the  $f(x_1, x_2, \dots, x_N)$  function into a boolean circuit. The logical function  $f_c$  is the output of the logical boolean circuit generator which implements  $f(x_1, x_2, \dots, x_N)$ . From the logical implementation,  $f_c()$ , Alice will generate a garbled version of the logical function,  $F_g^f$ , as well as an encryption key  $e_g^f$  and a decryption key  $d_g^f$ . After that, Alice's input parameter ( $x_1$ ) and Bob's input parameter ( $x_2$ ) must both be encrypted using the key  $e_g^f$ . Note that the encryption must be implemented with OT because Alice cannot know the Bob's input  $x_2$  and Bob cannot know the encryption key,  $e_g^f$ , generated by Alice [1]. Therefore after encryption Bob only gets  $X_2$ , which is its garbled input parameter. Next, Alice sends to Bob the garbled circuit,  $F_g^f$ , and its garbled input,  $X_1$ . With  $F_g^f$ ,  $X_1$  and  $X_2$  Bob computes, the garbled output  $Y_g^f$ . With  $Y_g^f$  and the decryption key  $d_g^f$ , Bob obtains  $y = f(x_1, x_2)$ . To conclude the protocol Bob will send to Alice the result  $y$ . Figure 6.446 shows the diagram of *garbled circuit protocol*.

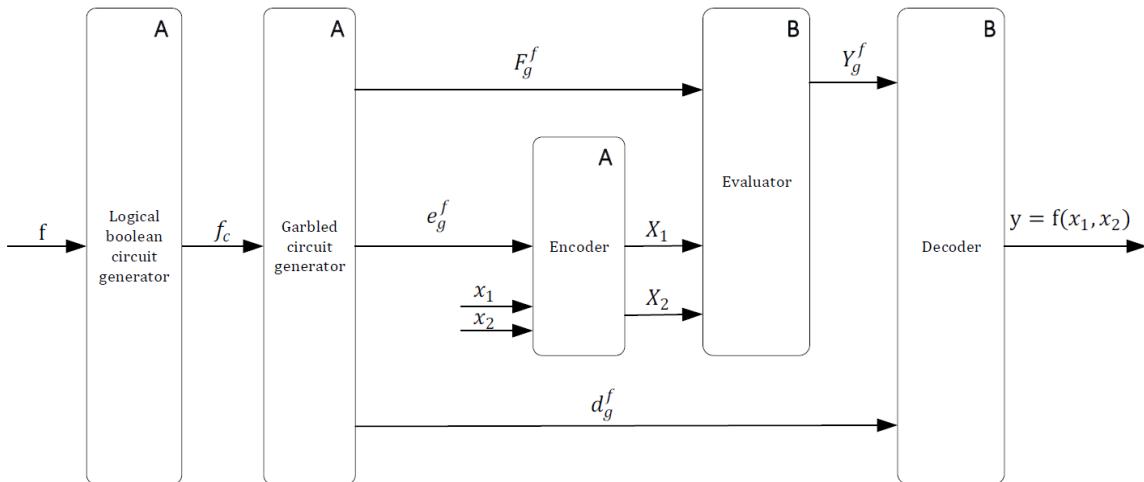


Figure 6.442: Block diagram of the garbled circuit protocol. Function to be computed  $y = f(x_1, x_2)$ . The blocks with label A are implemented by Alice and the blocks with label B are implemented by Bob.

|         |                                 |
|---------|---------------------------------|
| $x_1$   | Alice input parameter           |
| $x_2$   | Bob input parameter             |
| $f$     | Function to compute             |
| $f_c$   | Logical function to be computed |
| $F_g^f$ | Garbled circuit of $f$          |
| $X_1$   | Alice garbled input             |
| $X_2$   | Bob garbled input               |
| $e_g^f$ | Encoding key                    |
| $d_g^f$ | Decoding key                    |
| $Y_g^f$ | Garbled output                  |

Table 6.58: Description of parameters of figure 6.446.

#### 6.24.2.1 Example of a two-party computation with a single output

Lets assume:

$$y = f(x_1, x_2), \quad (6.457)$$

with,

$$y = f_c(x_1, x_2) = x_1 \wedge x_2, \quad (6.458)$$

where  $x_1$  and  $x_2$  are single bit values. Note that even if the input parameters of  $f$  and  $f_c$  are represented by the same symbols they are different. The input parameters of  $f_c$  are logical representations of the input parameters of  $f$ .

Figure 6.443: Gate level implementation of  $f_c$ .

Figure 6.447 shows the logical boolean circuit representation of the function to be computed. In this particular case there are two parties, each with an input parameter  $x_i$ , where  $i$  can be 1 (Alice) or 2 (Bob), and  $x_i$  can take values 0 or 1. Alice will play the role of *garbled circuit generator* so she will generate a garbled AND gate and send it to Bob. On the other hand, Bob will play the role of *garbled circuit evaluator* [2].

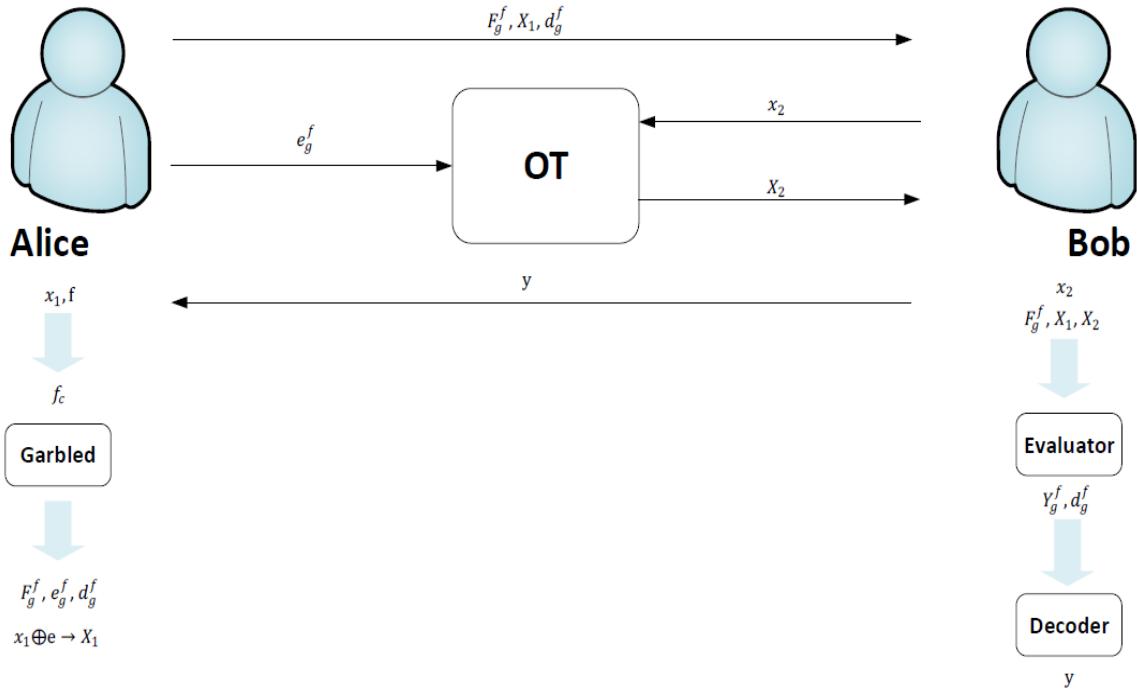


Figure 6.444: Garbled Circuit Protocol with OT.

| $x_1$ | $x_2$ | $y = x_1 \wedge x_2$ |
|-------|-------|----------------------|
| 0     | 0     | 0                    |
| 0     | 1     | 0                    |
| 1     | 0     | 0                    |
| 1     | 1     | 1                    |

Table 6.59: Truth table of  $f_c(x_1, x_2)$ .

Table 6.69 shows the truth table of  $f_c$ . Initially, Alice generates an encryption key with length equals to the sum of the length of both input parameters, two bit length in this case. Then, she encrypts her two possible input parameters,  $X_1 = x_1 \oplus e_g^f[1]$ , as well as Bob's two possible input parameters,  $X_2 = x_2 \oplus e_g^f[2]$ . Lets assume  $e_g^f = 01$  the encryption key used by Alice. In this case, the encryption key has two bits and Alice will use the first bit to encrypt her input parameter  $x_1$  and the second bit to encrypt Bob's input parameter  $x_2$ . We are defining the Alice encrypted input value with the label  $W_1^0$  when it before encryption assumes the value  $x_1 = 0$ , and  $W_1^1$  when it before encryption assumes the value  $x_1 = 1$ . The same happens for Bob, please see table 6.60 and 6.61.

|       |                                   |
|-------|-----------------------------------|
| $x_1$ | $X_1 = W_1^{x_1}$                 |
| 0     | $W_1^0 = x_1 \oplus e_g^f[1] = 0$ |
| 1     | $W_1^1 = x_1 \oplus e_g^f[1] = 1$ |

Table 6.60: Encryption results for Alice's input parameter  $x_1$  using the first bit of encryption key  $e_g^f[1] = 0$ .

|       |                                   |
|-------|-----------------------------------|
| $x_2$ | $X_2 = W_2^{x_2}$                 |
| 0     | $W_2^0 = x_2 \oplus e_g^f[2] = 1$ |
| 1     | $W_2^1 = x_2 \oplus e_g^f[2] = 0$ |

Table 6.61: Encryption results for Bob's input parameter  $x_1$  using the second bit of encryption key  $e_g^f[2] = 1$ .

Now, Alice computes an hash value for each combination  $W_1^i$ ,  $W_2^i$ , and  $y = f_c(x_1, x_2)$ . Lets assume she computed the Hash Functions and the results are the following:

| $x_1$ | $W_1^i$ | $x_2$ | $W_2^i$ | $y = f_c(x_1, x_2)$ | $H(W_1^i, W_2^i, y)$ | Output    |
|-------|---------|-------|---------|---------------------|----------------------|-----------|
| 0     | 0       | 0     | 1       | 0                   | H(0,1,0)             | 0 0 0 0 0 |
| 0     | 0       | 1     | 0       | 0                   | H(0,0,0)             | 0 1 0 0 0 |
| 1     | 1       | 0     | 1       | 0                   | H(1,0,0)             | 1 0 0 0 0 |
| 1     | 1       | 1     | 0       | 1                   | H(1,0,1)             | 1 1 1 0 0 |

Table 6.62: Hash function computed and its result.

Before forwarding these values to Bob, Alice also performs an encryption over these results using a decryption key with size equal to the length of the hash value,  $d_g^f = 11100$ ,  $\text{Enc}(H(W_1^i, W_2^i, f_c)) = H(W_1^i, W_2^i, f_c) \oplus d_g^f$ .

|                                  |           |
|----------------------------------|-----------|
| $\text{Enc}(H(W_1^0, W_2^0, 0))$ | 1 1 1 0 0 |
| $\text{Enc}(H(W_1^0, W_2^1, 0))$ | 1 0 1 0 0 |
| $\text{Enc}(H(W_1^1, W_2^0, 0))$ | 0 1 1 0 0 |
| $\text{Enc}(H(W_1^1, W_2^1, 1))$ | 0 0 0 0 0 |

Table 6.63: Encrypted hash results.

Now the rows of table 6.63 must be permuted randomly.

|                                  |           |
|----------------------------------|-----------|
| $\text{Enc}(H(W_1^0, W_2^1, 0))$ | 1 0 1 0 0 |
| $\text{Enc}(H(W_1^1, W_2^1, 1))$ | 0 0 0 0 0 |
| $\text{Enc}(H(W_1^0, W_2^0, 0))$ | 1 1 1 0 0 |
| $\text{Enc}(H(W_1^1, W_2^0, 0))$ | 0 1 1 0 0 |

Table 6.64: Permutated Garbled Circuit Output

| $W_1^i$ | $W_2^i$ | $\text{Enc}(H(W_1^i, W_2^i, f))$ |
|---------|---------|----------------------------------|
| 0       | 0       | 1 0 1 0 0                        |
| 1       | 0       | 0 0 0 0 0                        |
| 0       | 1       | 1 1 1 0 0                        |
| 1       | 1       | 0 1 1 0 0                        |

Table 6.65: Permutated Garbled Circuit.

Alice is going to send to Bob the permuted Garbled Circuit to Bob, i.e. table 6.65 or a circuit that implement that table. Alice is also going to send to Bob the decryption key. Using  $X_1$ , that Alice also sends to Bob, and  $X_2$ , that Bob obtains from OT, he obtains the circuit output for that input values  $(X_1, X_2)$ . He decrypts this obtaining  $H = Y \oplus d_g^f$ , i.e. a valid hash value. After, he tries all possible outputs, doing  $H(W_1^i, W_2^i, \text{guess } y)$ . The triplet  $X_1, X_2, \text{guess } y$  that generates the correct hash value  $H$  corresponds to the right  $y$ . After having  $y$  he send this value to Alice. Note that at the end Bob only knows  $x_2$  and  $y$ , and Alice only knows  $x_1$  and  $y$ .

### 6.24.3 ANP - Problem definition

This is a rewritten version of the above sections. In *Secure Multi-Party Computation* (SMC) a function  $(y_1, y_2, y_3, y_4, \dots, y_N) = f(x_1, x_2, x_3, x_4, \dots, x_N)$  is computed, assuring that each party  $i$  only know its  $(x_i, y_i)$  pair, the input/output pair of all the other parties must remain unknown. [1].

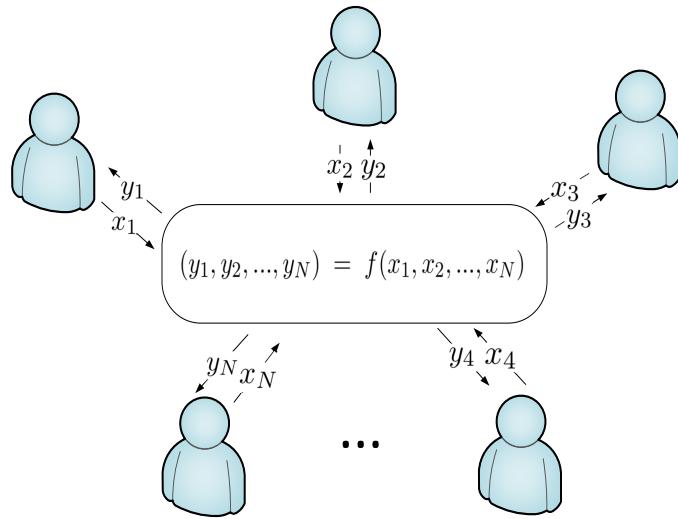


Figure 6.445:  $N$  entities are able to jointly compute  $(y_1, y_2, y_3, y_4, \dots, y_N) = f(x_1, x_2, x_3, x_4, \dots, x_N)$  making sure that each entity only have access its input/output pair, i.e.  $(x_i, y_i)$ .

There are, at least, two different ways of computing this function:

1. Using a trusted party, who has the trust of all the parties. This trusted party receives the all inputs, calculates the outputs and gives the the information to each party without any leak of information.
2. Using a SMC protocol.

Here, we are only interest in the use of a SMC protocol and we are going to restrict our work to two-party computation, with a single output, i.e.  $y = f(x_1, x_2)$ . For this case there is a well known protocol named *Yao's Garbled Circuit Protocol* (GCP), which we are going to describe.

### 6.24.4 ANP - Secure Two-Party Computation

The GCP starts with the implementation of the function  $f(x_1, x_2)$  using a logical circuit, i.e. using logical gates. We are going to assume that the logical circuit is implemented using NAND gates. Note that a NAND gate is a universal gate, i.e. any logical circuit can be implemented using only NAND gates. From the logical implementation,  $f_c$ , Alice will generate a garbled circuit,  $F_g^f$ , as well as an encryption key  $e_g^f$  and a decryption key

$d_g^f$ . Because we are going to assume AES-128 as the encrypting and decrypting, which is a symmetric algorithm, both keys are identical and correspond to 128 bits. Alice's input parameter ( $x_1$ ) and Bob's input parameter ( $x_2$ ) must both be encrypted using the key  $e_g^f$ . Note that the encryption of Bob's input,  $x_2$ , must be implemented with OT because Alice cannot know the Bob's input,  $x_2$ , and Bob cannot know the encryption key,  $e_g^f$ , used by Alice [1]. Therefore after encryption Bob only gets  $X_2$ , which is its encrypted input. Next, Alice sends to Bob the garbled circuit,  $F_g^f$ , and its encrypted input,  $X_1$ . With  $F_g^f$ ,  $X_1$  and  $X_2$  Bob computes, the garbled output  $Y_g^f$ . At this stage Alice is going to send to Bob enough information so that from  $Y_g^f$  he can obtain  $y$ . To conclude the protocol Bob will send to Alice the result  $y$ . Figure 6.446 shows the diagram of *garbled circuit protocol*.

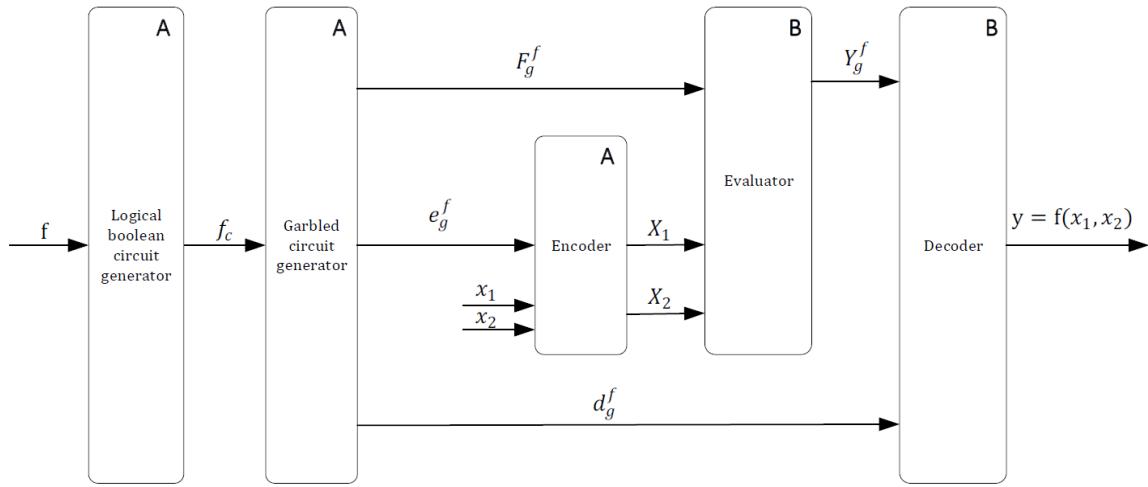


Figure 6.446: Block diagram of the garbled circuit protocol. Function to be computed  $y = f(x_1, x_2)$ . The blocks with label A are implemented by Alice and the blocks with label B are implemented by Bob.

|         |                         |
|---------|-------------------------|
| $x_1$   | Alice input parameter   |
| $x_2$   | Bob input parameter     |
| $f$     | Function to be computed |
| $f_c$   | Logical implementation  |
| $F_g^f$ | Garbled circuit         |
| $X_1$   | Alice encrypted input   |
| $X_2$   | Bob encrypted input     |
| $e_g^f$ | Encoding key            |
| $d_g^f$ | Decoding key            |
| $Y_g^f$ | Garbled output          |

Table 6.66: Description of parameters of figure 6.446.

Note that to implement the GCP, it is necessary an encryption protocol, usually AES, and an OT protocol, usually based on the RSA.

#### 6.24.4.1 Lets consider the simplest possible circuit - a NAND gate

Lets assume:

$$y = f(x_1, x_2), \quad (6.459)$$

with,

$$y = f_c(x_1, x_2) = \sim(x_1 \wedge x_2), \quad (6.460)$$

where  $x_1$  and  $x_2$  are single bit values. Note that even if the input parameters of  $f$  and  $f_c$  are represented by the same symbols they are different. The input parameters of  $f_c$  are logical representations of the input parameters of  $f$ .

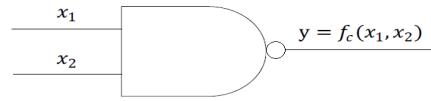


Figure 6.447: Gate level implementation of  $f_c$ .

Figure 6.447 shows the logical boolean circuit representation of the function to be computed. In this particular case there are two parties, each with an input parameter  $x_i$ , where  $i$  can be 1 (Alice) or 2 (Bob), and  $x_i$  can take the logical value 0 or 1. Alice will play the role of *garbled circuit generator* so she will generate a garbled NAND gate and send it to Bob. On the other hand, Bob will play the role of *garbled circuit evaluator* [2].

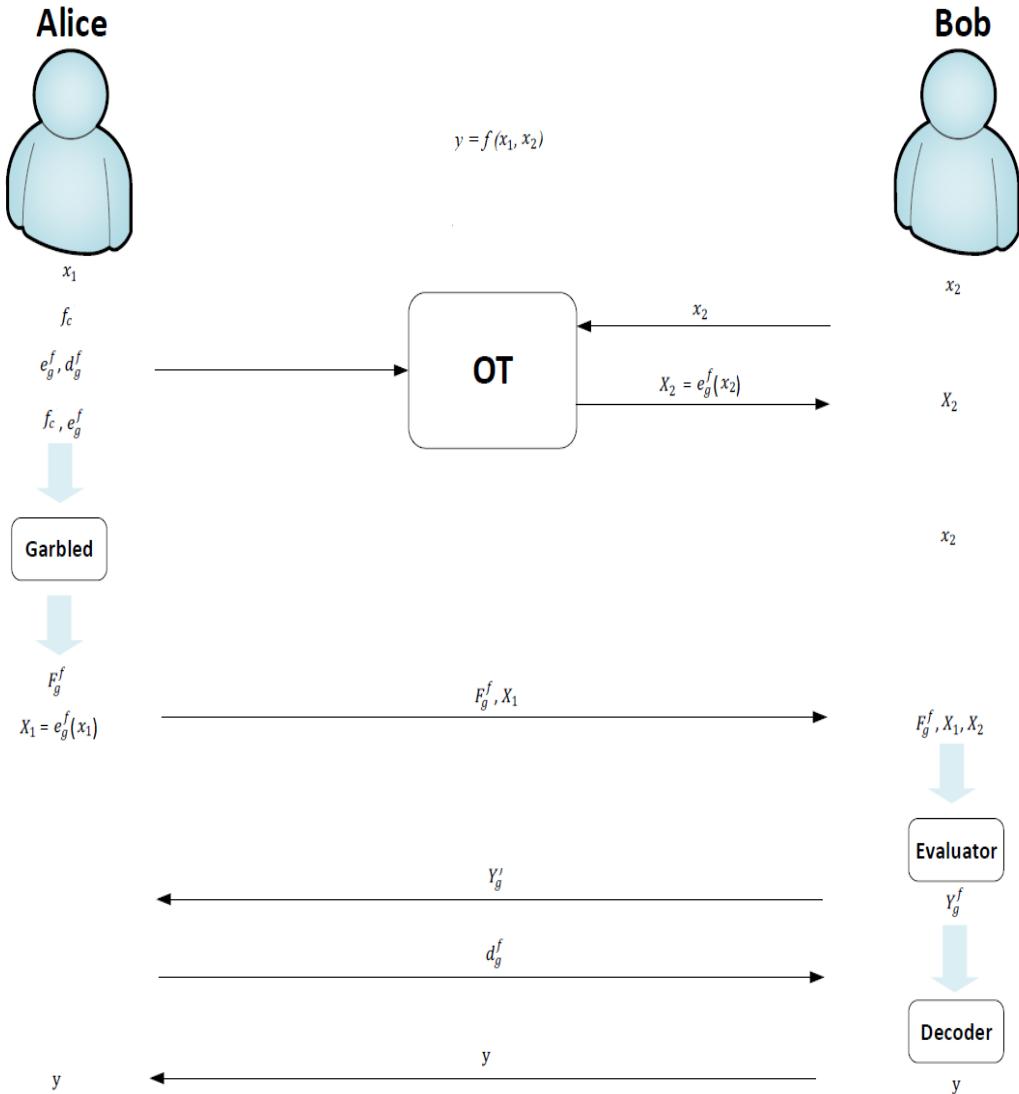


Figure 6.448: Garbled Circuit Protocol with OT.

| $x_1$ | $x_2$ | $y = \sim (x_1 \wedge x_2)$ |
|-------|-------|-----------------------------|
| 0     | 0     | 1                           |
| 0     | 1     | 1                           |
| 1     | 0     | 1                           |
| 1     | 1     | 0                           |

Table 6.67: Truth table of  $f_c(x_1, x_2)$ .

Table 6.69 shows the truth table of  $f_c$ .

Alice is going to use six random strings,  $W_0^A, W_1^A, W_0^B, W_1^B, W_0^C, W_1^C$  usually named labels.  $W_0^A$ , represents a logical zero for Alice,  $W_1^A$  represents a logical one for Alice,  $W_0^B$ ,

represents a logical zero for Bob,  $W_1^B$  represents a logical one for Bob. We are going to assume that the  $W_0^A, W_1^A, W_0^B, W_1^B$  labels are 64 bits. By concatenating two 64-bit labels Alice is going to obtain four different keys, that we are going to assume to be 128-bits AES keys,  $K00 = [W_0^A W_0^B]$ ,  $K01 = [W_0^A W_1^B]$ ,  $K10 = [W_1^A W_0^B]$  and  $K11 = [W_1^A W_1^B]$ . The labels  $W_0^C, W_1^C$  are 128 labels and are used to represent the outputs 0 and 1. Alice is going to obtain the following table:

| $X_1$   | $X_2$   | $Y$                |
|---------|---------|--------------------|
| $W_0^A$ | $W_0^B$ | $enc_{K00}(W_1^C)$ |
| $W_0^A$ | $W_1^B$ | $enc_{K01}(W_1^C)$ |
| $W_1^A$ | $W_0^B$ | $enc_{K10}(W_1^C)$ |
| $W_1^A$ | $W_1^B$ | $enc_{K11}(W_0^C)$ |

Table 6.68: Truth table of  $f_c(x_1, x_2)$ .

Now she apply a random permutation to the table and send the table to Bob.

| $X_1$   | $X_2$   | $Y$                |
|---------|---------|--------------------|
| $W_0^A$ | $W_0^B$ | $enc_{K00}(W_1^C)$ |
| $W_1^A$ | $W_1^B$ | $enc_{K11}(W_0^C)$ |
| $W_0^A$ | $W_1^B$ | $enc_{K01}(W_1^C)$ |
| $W_1^A$ | $W_0^B$ | $enc_{K10}(W_1^C)$ |

Table 6.69: Truth table of  $f_c(x_1, x_2)$ .

She is also sending to Bob the output labels,  $W_0^C$  and  $W_1^C$ .

Finally, Alice sends to Bob her encrypted input,  $W_{x_1}^A$ . Note that from the label Bob cannot know nothing about  $x_1$ .

Bob also knows its label  $W_{x_2}^B$  that he obtained through OT, therefore he can build the right key  $K_{x_1 x_2} = [W_{x_1}^A W_{x_2}^B]$ .

He is going to apply the key to all the rows of the permuted column and he is going to obtain 4 decrypted words. Because only one of the four decrypted words equals  $W_0^C$  or  $W_1^C$  Bob knows which is the output of this gate,  $W_0^C$  or  $W_1^C$ . Bob sends this label to Alice and Alice will tell him which was the result of the computation, 0 or 1.

Note that this is scalable for any number of gates.

### 6.24.5 TinyGarble

This subsection will cover the installation and usage of TinyGarble, a GC framework that allows two parties to safely compute any function with their private inputs through Yao's Garbled Circuit Protocol. It accepts HDL and C/C++ as user inputs, although HDL will be the primary focus.

#### 6.24.5.1 Installation

Since TinyGarble and most of the tools mentioned below have been mostly tested on Linux, it will be assumed that the user is running Linux on a virtual machine (VM), or directly from a partition. To do the former one can use Oracle's VirtualBox, which can be downloaded at <https://www.virtualbox.org/wiki/Downloads>. A copy of Linux is also needed, for this section Ubuntu 18.04 was used, found at <https://www.ubuntu.com/download/desktop>.

After installing and opening VirtualBox, select 'New', a Window should pop up asking for a name for the new VM, it doesn't matter, and for type and version, which should be Linux and Ubuntu respectively. The next step is choosing the amount of RAM that is going to be dedicated to the VM, it depends on the system, but anything between 2048 and 4096 MB should be fine, 1024 MB or lower if the host PC has small amount of memory. Next create a virtual hard disk of VDI type, and choose dynamic or fixed size disk, it comes down to user preference. For the size of the disk, 10GB was enough to install and use TinyGarble properly, but slightly more is advisable. After creating the virtual machine, it should ask for a start-up disk when launching it for the first time, select the Linux ISO file that was downloaded before. After that the installation of linux is pretty straight forward.

For TinyGarble to work, some dependencies have to be installed first, on Ubuntu it can be done with the following lines:

```
$ sudo apt-get install build-essential
$ sudo apt-get install libssl1.0-dev
$ sudo apt-get install libboost-all-dev
$ sudo apt-get install cmake
```

Listing 6.1: Installation of TinyGarble's dependencies

After this step it's necessary to clone TinyGarble's repository, at <https://github.com/esonghori/TinyGarble>, and build it.

```
$ git clone https://github.com/esonghori/TinyGarble.git
$ cd TinyGarble/
$ ./configure
$ cd bin/
$ make
```

Listing 6.2: Configuration and compilation of TinyGarble

To be able to use TinyGarble, a RTL Synthesis tool is also needed, to translate Verilog files into Netlist ones. In the example showed later, Yosys-abc is used. Which can be installed on Ubuntu 15.04 or higher with:

```
$ sudo apt-get install yosys
```

Listing 6.3: Installation of Yosys-abc for Ubuntu 15.04>

If on Ubuntu 14.04 or lower, a repository needs to be added first:

```
$ sudo add-apt-repository ppa:saltmakrell/ppa  
$ sudo apt-get update  
$ sudo apt-get install yosys
```

Listing 6.4: Installation of Yosys-abc for Ubuntu 14.04<

This next tool, vhd2vl, will also be needed if the desirable input is VHDL, to translate it into Verilog, which is the input supported by the RTL Synthesis Tool mentioned above. Before cloning vhd2vl at <https://github.com/ldoolitt/vhd2vl> and building it, some dependencies have to be installed:

```
$ sudo apt-get install flex  
$ sudo apt-get install bison  
$ git clone https://github.com/ldoolitt/vhd2vl  
$ cd vhd2vl/src  
$ make  
$ sudo cp vhd2vl /usr/local/bin
```

Listing 6.5: Installation of VHD2VL

### 6.24.5.2 Usage

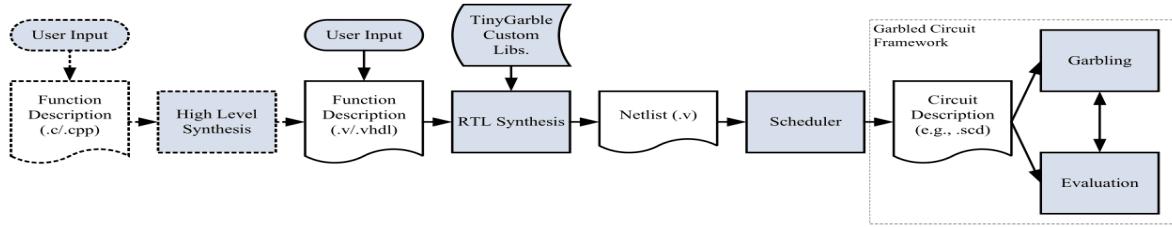


Figure 6.449: TinyGarble Global Flow[3]

Following TinyGarble's global flow showed on the image above, it's possible to see that a C/C++ or Hardware Design Language (HDL) input is necessary. The former is slightly worse in terms of performance, since it needs an extra step of synthesis, with the help of a High Level Synthesis (HLS) tool, like Spart or Xilinx Vivado for C, to compile the C/C++ file into HDL, causing the end SCD file to have more logical gates than when using HDL input directly. This section mostly covers Verilog, since it's the most direct input, if VHDL input is needed, check ??.

When writing the input file, some attention to the circuit's ports is needed, since TinyGarble's V2SCD (Netlist Verilog to SCD converter) accepts netlist files with a special format. Below are the ports that can be used and what they represent.

- clk: clock cycle
- rst: active high reset
- g\_init: garbler's initial values (read at the first clock)
- e\_init: evaluator's initial values (read at the first clock)
- g\_input: garbler's input (read at every clock)
- e\_input: evaluator's input (read at every clock)
- o: output

Below it's an example of an and\_gate written in verilog with the format above.

```

module and_gate (input g_input, e_input, output o);
  assign o = g_input & e_input;
endmodule
  
```

Listing 6.6: and\_gate.v

If the file is written in VHDL instead, it has to be translated to Verilog with the tool mentioned before, vhdl2v1, since the next step needs a Verilog file.

```
$ vhdl2vlg and_gate.vhd and_gate.v
```

Listing 6.7: Translation of VHDL file into Verilog

This next step requires the RTL Synthesis tool mentioned before, Yosys, in order to compile the RTL Verilog file into a Netlist Verilog File. Using TinyGarble's custom library, "asic\_cell\_extended.lib", located at TinyGarble/circuit\_synthesis/lib/, and assuming that the Verilog file is located at a folder named "circuits", the compilation can be done with the following instructions:

```
$ ./yosys
yosys> read_verilog circuits/and_gate.v
yosys> hierarchy -check -top and_gate
yosys> proc; opt; memory; opt; fsm; opt; techmap; opt;
yosys> abc -liberty TinyGarble/circuit_synthesis/lib/
asic_cell_extended.lib
yosys> clean
yosys> write_verilog circuits/and_gate_netlist.v
yosys> exit
```

Listing 6.8: Yosys instructions to compile the HDL file to a Netlist file

Following these, the Netlist file below should be generated.

```
(* top = 1 *)
(* src = "and.v:1" *)
module and_gate( g_input, e_input, o);
(* src = "and.v:2" *)
  input g_input;
(* src = "and.v:2" *)
  input e_input;
(* src = "and.v:3" *)
  output o;
AND _0_ (
  .A(e_input),
  .B(g_input),
  .Z(o)
);
endmodule
```

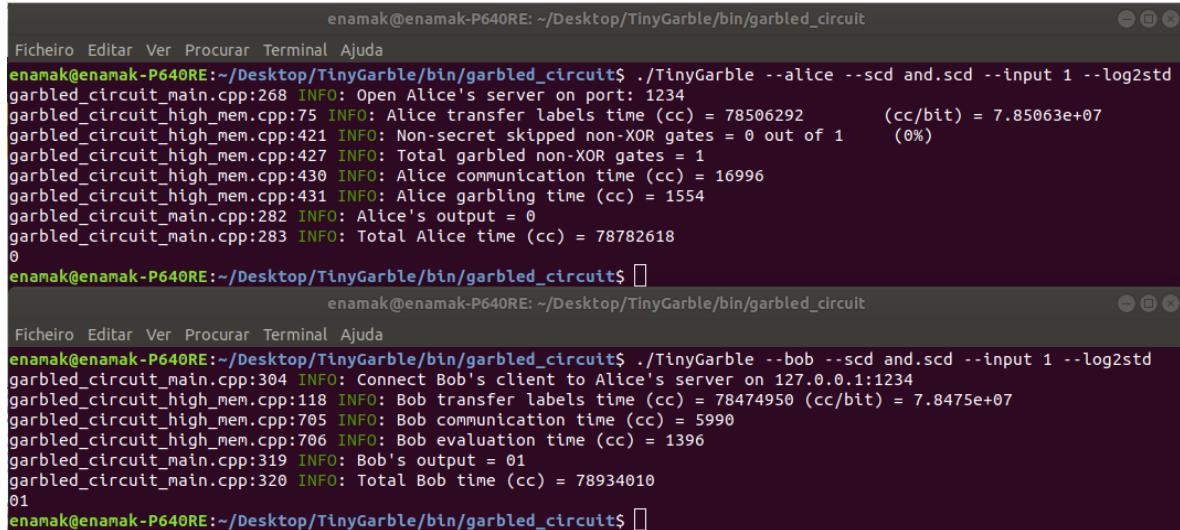
Listing 6.9: and\_gate\_netlist.v

For some unkown reason, the output file has some noticeably weird lines, starting with "(\*", that will cause Segmentation Fault error in the next step if not removed. After removing those lines, the Netlist file can be converted into a SCD file with the instruction below:

```
$ ./TinyGarble/bin/scd/V2SCD_Main -i circuits/and_gate_netlist.v
-o circuits/and_gate_scd.scd
```

Listing 6.10: Installation of Yosys-abc

The SCD file can now be executed by both parties, "Alice" and "Bob", as shown on the upper terminal and lower terminal of the following images, respectively. For this examples the SCD file was located at TinyGarble/bin/garbled\_circuits, same directory where the instructions were executed.



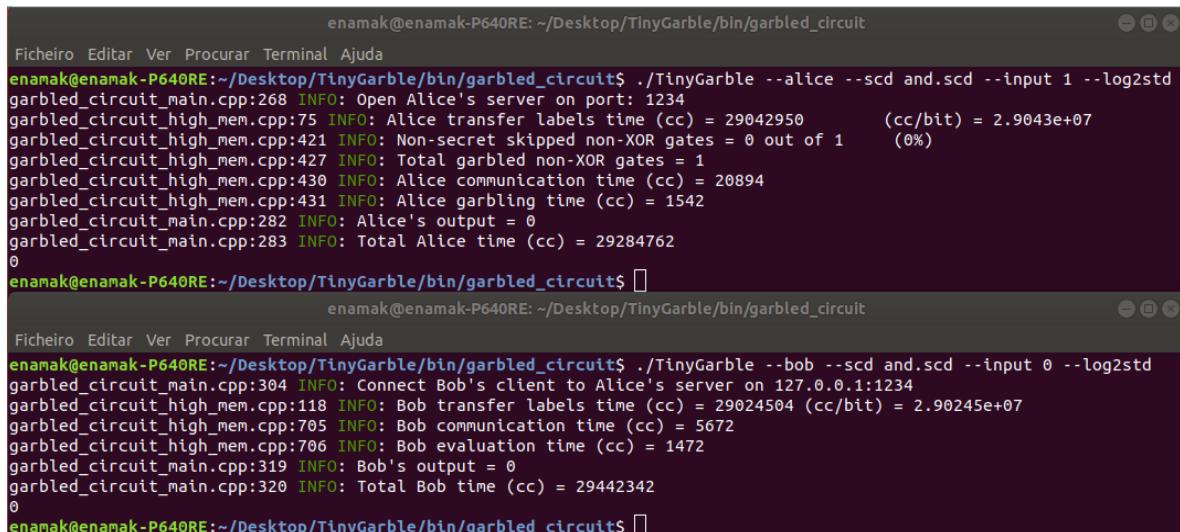
```

enamak@enamak-P640RE: ~/Desktop/TinyGarble/bin/garbled_circuit$ ./TinyGarble --alice --scd and.scd --input 1 --log2std
garbled_circuit_main.cpp:268 INFO: Open Alice's server on port: 1234
garbled_circuit_high_mem.cpp:75 INFO: Alice transfer labels time (cc) = 78506292 (cc/bit) = 7.85063e+07
garbled_circuit_high_mem.cpp:421 INFO: Non-secret skipped non-XOR gates = 0 out of 1 (0%)
garbled_circuit_high_mem.cpp:427 INFO: Total garbled non-XOR gates = 1
garbled_circuit_high_mem.cpp:430 INFO: Alice communication time (cc) = 16996
garbled_circuit_high_mem.cpp:431 INFO: Alice garbling time (cc) = 1554
garbled_circuit_main.cpp:282 INFO: Alice's output = 0
garbled_circuit_main.cpp:283 INFO: Total Alice time (cc) = 78782618
0
enamak@enamak-P640RE:~/Desktop/TinyGarble/bin/garbled_circuit$ []

Ficheiro Editar Ver Procurar Terminal Ajuda
enamak@enamak-P640RE:~/Desktop/TinyGarble/bin/garbled_circuit$ ./TinyGarble --bob --scd and.scd --input 1 --log2std
garbled_circuit_main.cpp:304 INFO: Connect Bob's client to Alice's server on 127.0.0.1:1234
garbled_circuit_high_mem.cpp:118 INFO: Bob transfer labels time (cc) = 78474950 (cc/bit) = 7.8475e+07
garbled_circuit_high_mem.cpp:705 INFO: Bob communication time (cc) = 5990
garbled_circuit_high_mem.cpp:706 INFO: Bob evaluation time (cc) = 1396
garbled_circuit_main.cpp:319 INFO: Bob's output = 01
garbled_circuit_main.cpp:320 INFO: Total Bob time (cc) = 78934010
01
enamak@enamak-P640RE:~/Desktop/TinyGarble/bin/garbled_circuit$ []

```

Figure 6.450: "And" function executed by Alice and Bob, with inputs 1 and 1, respectively



```

enamak@enamak-P640RE: ~/Desktop/TinyGarble/bin/garbled_circuit$ ./TinyGarble --alice --scd and.scd --input 1 --log2std
garbled_circuit_main.cpp:268 INFO: Open Alice's server on port: 1234
garbled_circuit_high_mem.cpp:75 INFO: Alice transfer labels time (cc) = 29042950 (cc/bit) = 2.9043e+07
garbled_circuit_high_mem.cpp:421 INFO: Non-secret skipped non-XOR gates = 0 out of 1 (0%)
garbled_circuit_high_mem.cpp:427 INFO: Total garbled non-XOR gates = 1
garbled_circuit_high_mem.cpp:430 INFO: Alice communication time (cc) = 20894
garbled_circuit_high_mem.cpp:431 INFO: Alice garbling time (cc) = 1542
garbled_circuit_main.cpp:282 INFO: Alice's output = 0
garbled_circuit_main.cpp:283 INFO: Total Alice time (cc) = 29284762
0
enamak@enamak-P640RE:~/Desktop/TinyGarble/bin/garbled_circuit$ []

Ficheiro Editar Ver Procurar Terminal Ajuda
enamak@enamak-P640RE:~/Desktop/TinyGarble/bin/garbled_circuit$ ./TinyGarble --bob --scd and.scd --input 0 --log2std
garbled_circuit_main.cpp:304 INFO: Connect Bob's client to Alice's server on 127.0.0.1:1234
garbled_circuit_high_mem.cpp:118 INFO: Bob transfer labels time (cc) = 29024504 (cc/bit) = 2.90245e+07
garbled_circuit_high_mem.cpp:705 INFO: Bob communication time (cc) = 5672
garbled_circuit_high_mem.cpp:706 INFO: Bob evaluation time (cc) = 1472
garbled_circuit_main.cpp:319 INFO: Bob's output = 0
garbled_circuit_main.cpp:320 INFO: Total Bob time (cc) = 29442342
0
enamak@enamak-P640RE:~/Desktop/TinyGarble/bin/garbled_circuit$ []

```

Figure 6.451: "And" function executed by Alice and Bob, with inputs 1 and 0, respectively

The input is written in hexadecimal, and the log2std option is to show information of the computation on the terminal instead of creating a log file.

The figure consists of two screenshots of a terminal window. Both windows have a dark background and light-colored text. The top window is titled 'enamak@Desktop: ~/TinyGarble/bin/garbled\_circuit'. It shows the command: ./TinyGarble -a --scd\_file xor\_4bits.scd --input 0 --log2std. The output includes log messages from 'garbled\_circuit\_main.cpp' and 'garbled\_circuit\_high\_mem.cpp' related to Alice's server setup and communication times. The bottom window is also titled 'enamak@Desktop: ~/TinyGarble/bin/garbled\_circuit'. It shows the command: ./TinyGarble -b --scd\_file xor\_4bits.scd --input F --log2std. The output includes log messages from 'garbled\_circuit\_main.cpp' and 'garbled\_circuit\_high\_mem.cpp' related to Bob's client connection and evaluation time.

```

enamak@Desktop:~/TinyGarble/bin/garbled_circuit
Ficheiro Editar Ver Procurar Terminal Ajuda
enamak@Desktop:~/TinyGarble/bin/garbled_circuit$ ./TinyGarble -a --scd_file xor_4bits.scd --input 0 --log2std
garbled_circuit_main.cpp:268 INFO: Open Alice's server on port: 1234
garbled_circuit_high_mem.cpp:75 INFO: Alice transfer labels time (cc) = 86117617 (cc/bit) = 2.15294e+07
garbled_circuit_high_mem.cpp:421 INFO: Non-secret skipped non-XOR gates = 0 out of 0 (-nan%)
garbled_circuit_high_mem.cpp:427 INFO: Total garbled non-XOR gates = 0
garbled_circuit_high_mem.cpp:430 INFO: Alice communication time (cc) = 12608
garbled_circuit_high_mem.cpp:431 INFO: Alice garbling time (cc) = 2575
garbled_circuit_main.cpp:282 INFO: Alice's output = 0
garbled_circuit_main.cpp:283 INFO: Total Alice time (cc) = 86729284
0

enamak@Desktop:~/TinyGarble/bin/garbled_circuit
Ficheiro Editar Ver Procurar Terminal Ajuda
enamak@Desktop:~/TinyGarble/bin/garbled_circuit$ ./TinyGarble -b --scd_file xor_4bits.scd --input F --log2std
garbled_circuit_main.cpp:304 INFO: Connect Bob's client to Alice's server on 127.0.0.1:1234
garbled_circuit_high_mem.cpp:118 INFO: Bob transfer labels time (cc) = 86218157 (cc/bit) = 2.15545e+07
garbled_circuit_high_mem.cpp:705 INFO: Bob communication time (cc) = 4248
garbled_circuit_high_mem.cpp:706 INFO: Bob evaluation time (cc) = 2906
garbled_circuit_main.cpp:319 INFO: Bob's output = 0F
garbled_circuit_main.cpp:320 INFO: Total Bob time (cc) = 87006402
0F

```

Figure 6.452: "Xor" function with 4 bits, executed by Alice and Bob, with inputs 0 and F, respectively

#### 6.24.5.3 Using VHDL

ssec:usingVHDL

If the desirable input is VHDL, this next tool, V2VHDL, will be necessary to translate it into Verilog, which is the input supported by the RTL Synthesis Tool mentioned before. Before cloning vhd2vl at <https://github.com/ldoolitt/vhd2vl> and building it, some dependencies have to be installed:

```
$ sudo apt install flex
$ sudo apt install bison
$ git clone https://github.com/ldoolitt/vhd2vl
$ cd vhd2vl/src
$ make
$ sudo cp vhd2vl /usr/local/bin
```

Listing 6.11: Installation of VHD2VL

To use it simply run:

```
$ vhd2vl and_gate.vhd and_gate.v
```

Listing 6.12: Translation of VHDL file into Verilog

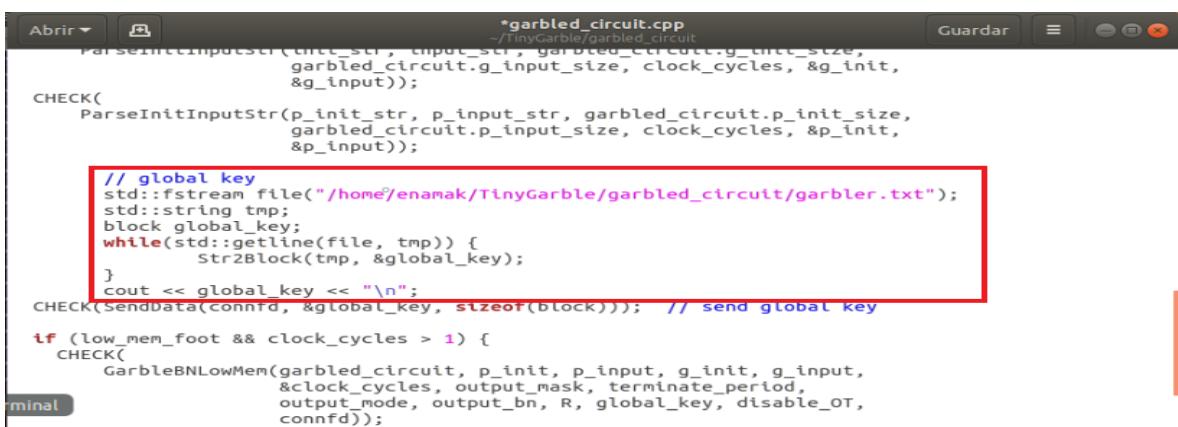
#### 6.24.5.4 Programming

#### 6.24.5.5 File Architecture

1. a23: Algorithms wrote in C.
2. bin: Created when building TinyGarble. Holds executables and examples of netlist and scd files.
3. circuit\_synthesis: Holds netlist and RTL Verilog files and the custom libs to use with Yosys.
4. crypto: Holds C++ files for the Oblivious Transfer and their data structure, 'block'.
5. garbled\_circuit: Holds C++ files to deal with the garbling and evaluation of a circuit.
6. scd: Holds C++ files of the SCD Converter and Evaluator mentioned before, and of the netlist Parser.
7. tcip: Holds C++ files that deal with server connection.
8. util: Holds C++ support files.

#### 6.24.5.6 Keys

Since TinyGarble is based on JustGarble, it's garbling scheme will be based on fixed-key AES. If the user wants to use his own encryption and decryption keys, the "garbled\_circuit.cpp" file located at "TinyGarble/garbled\_circuit" has to be slightly changed. For this example, both the garbler and evaluator get their keys from their respective file, to accomplish this, on GarbleStr and EvaluateStr functions, change the line "block global\_key = RandomBlock();" with the code below:



```
*garbled_circuit.cpp
~/TinyGarble/garbled_circuit
ParseInitInputStr(p_init_str, p_input_str, garbled_circuit.g_init_size,
                  garbled_circuit.g_input_size, clock_cycles, &g_init,
                  &g_input));
CHECK(
    ParseInitInputStr(p_init_str, p_input_str, garbled_circuit.p_init_size,
                      garbled_circuit.p_input_size, clock_cycles, &p_init,
                      &p_input));
// global key
std::ifstream file("/home/enamak/TinyGarble/garbled_circuit/garbler.txt");
std::string tmp;
block global_key;
while(std::getline(file, tmp)) {
    Str2Block(tmp, &global_key);
}
cout << global_key << "\n";
CHECK(SendData(connfd, &global_key, sizeof(block))); // send global key
if (low_mem_foot && clock_cycles > 1) {
    CHECK(
        GarbleBNLowMem(garbled_circuit, p_init, p_input, g_init, g_input,
                        &clock_cycles, output_mask, terminate_period,
                        output_mode, output.bn, R, global_key, disable_OT,
                        connfd));
}
```

Figure 6.453: Reading Garbler's key from a file.

The figure above is showing a piece of the modified GarbleStr function, the same would have to be done at the EvaluateStr function with the respective path for the evaluator

keys. The absolute(starting from the root) path has to be given. "#Include <fstream>" is also needed at the top of the file.

To apply any changes made to the file, go to TinyGarble/bin and run the "make" command.

#### 6.24.5.7 Moda function

Since the input of the function both for Alice and Bob is a series of numbers, and since TinyGarble's iinput is in hexadecimal, we can read each digit as bcd, and easily get the total by multiplying the current number by 10 and summing the digit. Instead of using a ',' any value between a and f can be used.

The following function was written in VHDL:

```

module moda(clk, rst, g_input, e_input, o);

    input clk, rst;
    input [1023:0] g_input, e_input; // Up to 256 hexa chars, sent from the testbench
    output [8:0] o; // With 256 hexa chars the max of different numbers is 512 (256 / 2 (commas) * 4)

    wire clk, rst;
    reg [8:0] o;

    reg [9:0] g_iterator, e_iterator; // Represent up to the number 1023, to iterate the inputs
    wire [3:0] g_tmp, e_tmp; // To hold the current hexa char being checked
    reg [29:0] g_number, e_number; // Maximum of around 1 trillion, (10 hexa digits, because in bcd)
    reg g_end, e_end, g_start, e_start, g_blocked, e_blocked; // Control registers

    reg [30:0] bigger; // Hold the bigger number, one bit bigger than the biggest number for the inputs
    reg [8:0] biggerKey, counter; // Same as output

    reg [2:0] State;

    // Wires, constantly receiving the current value of their corresponding input with the current
    // iterator
    assign g_tmp = g_input[g_iterator -: 4]; // -: notation SystemVerilog, ver se funcioa com yosys
    assign e_tmp = e_input[e_iterator -: 4];

    always @(posedge clk) begin

        if (rst) begin
            State <= 0;
            o = 0;
        end else begin
            case(State)
                0 : begin
                    g_iterator <= 1023;
                    e_iterator <= 1023;
                    g_number <= 0;
                    e_number <= 0;
                    g_blocked <= 0;
                    e_blocked <= 0;
                end
            endcase
        end
    end

```

```

g_end <= 0;
e_end <= 0;

bigger <= 0;
biggerKey <= 0;
counter <= 0;

State <= 1;

end
1 : begin
  // Se ainda nao tiver chegado ao ultimo digito it ao proximo
  if(g_blocked == 0) begin
if(g_iterator < 4)
  g_end <= 1;
else
  g_iterator <= g_iterator - 4;
end
  if(e_blocked == 0) begin
if(e_iterator < 4)
  e_end <= 1;
else begin
  e_iterator <= e_iterator - 4;
end
  end
  State <= 2;
end
end
2 : begin
  if(g_blocked == 1 & e_blocked == 1)
State <= 3;
  else begin
if(g_tmp == 10 | g_end == 1)
  g_blocked <= 1;
else
  g_number <= (g_number * 10) + g_tmp;
if(e_tmp == 10 | e_end == 1)
  e_blocked <= 1;
else
  e_number <= (e_number * 10) + e_tmp;
State <= 1;
  end;
end
end
3 : begin
  if(g_number + e_number > bigger) begin
$display("Found bigger");
bigger <= g_number + e_number;
biggerKey <= counter;
  end
  counter <= counter + 1;
  g_number <= 0;
  g_blocked <= 0;
  e_number <= 0;
  e_blocked <= 0;
  if(g_end == 1 & e_end == 1) begin
State <= 4;
  end else
State <= 1;
end
end
4 : begin
  o <= biggerKey + 1;
  State <= 4;
end
default : begin
  State <= 0;

```

```

end
endcase
end
end

endmodule

```

Listing 6.13: moda.vhd

When running a testbench of the above with the g\_input = 80'h421a425a9a23a412a432, and e\_input = 80'h94a420a83a902a74a821, the result is the below:

```

enamak@P640RE: ~/circuits
Ficheiro Editar Ver Procurar Terminal Separadores Ajuda
enamak@P640RE:~/Tiny... x enamak@P640RE:~/circuits x enamak@P640RE:~/Tiny... x
enamak@P640RE:~/circuits$ iverilog moda_tb.v moda.v
enamak@P640RE:~/circuits$ vvp a.out
At time 0, value = xxx (x)
Number 1 is -> 515
Found bigger
Number 2 is -> 845
Found bigger
Number 3 is -> 92
Number 4 is -> 925
Found bigger
Number 5 is -> 486
Number 6 is -> 1253
Found bigger
At time 5365, value = 006 (6)
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 6010 ticks.
> []

```

Figure 6.454: Moda's testbench execution

Converting the file to Netlist Verilog wasnt a problem besides having to delete all those weird lines mentioned above. When trying to converto to Scd it gives the error:

```

enamak@P640RE: ~/TinyGarble/bin/scd
Ficheiro Editar Ver Procurar Terminal Separadores Ajuda
enamak@P640RE:~/Tiny... x enamak@P640RE:~/circuits x enamak@P640RE:~/Tiny... x
enamak@P640RE:~/TinyGarble/bin/scd$ cat V2SCD_Main-1537499093-error.log
parse_netlist.cpp:573 ERROR: wire_name_table.count(read_circuit_string.gate_list
_string[i].input[0]) != 0 failed: "e_iterator[1] _01398"
v_2_scd.cpp:38 ERROR: id assignment to netlist components failed.
v_2_scd_main.cpp:83 ERROR: Verilog to SCD failed.
enamak@P640RE:~/TinyGarble/bin/scd$ []

```

Figure 6.455: Model SCD conversion fail

## References

- [1] Frederic Naumann. "Garbled Circuits". In: *Network* 71 (2016).
- [2] Sophia Yakoubov. "A Gentle Introduction to Yaos Garbled Circuits". In: (2017).
- [3] Ebrahim M. Songhori et al. "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits". In: (2015).

## 6.25 Tiny Garble

|                      |   |  |
|----------------------|---|--|
| <b>Students Name</b> | : | Goncalo Vitor (15/06/2018 - )  |
| <b>Goal</b>          | : | Implementation of secure multi-party computation based on the Tiny Garble framework. |
| <b>Directory</b>     | : |  |

This subsection will cover the installation and usage of TinyGarble, a GC framework that allows two parties to safely compute any function with their private inputs through Yao's Garbled Circuit Protocol. It accepts HDL and C/C++ as user inputs, although HDL will be the primary focus.

### 6.25.1 Installation

Since TinyGarble and most of the tools mentioned below have been mostly tested on Linux, it will be assumed that the user is running Linux on a virtual machine (VM), or directly from a partition. To do the former one can use Oracle's VirtualBox, which can be downloaded at <https://www.virtualbox.org/wiki/Downloads>. A copy of Linux is also needed, for this section Ubuntu 18.04 was used, found at <https://www.ubuntu.com/download/desktop>.

After installing and opening VirtualBox, select 'New', a Window should pop up asking for a name for the new VM, it doesn't matter, and for type and version, which should be Linux and Ubuntu respectively. The next step is choosing the amount of RAM that is going to be dedicated to the VM, it depends on the system, but anything between 2048 and 4096 MB should be fine, 1024 MB or lower if the host PC has small amount of memory. Next create a virtual hard disk of VDI type, and choose dynamic or fixed size disk, it comes down to user preference. For the size of the disk, 10GB was enough to install and use TinyGarble properly, but slightly more is advisable. After creating the virtual machine, it should ask for a start-up disk when launching it for the first time, select the Linux ISO file that was downloaded before. After that the installation of linux is pretty straight forward.

For TinyGarble to work, some dependencies have to be installed first, on Ubuntu it can be done with the following lines:

```
$ sudo apt install git
$ sudo apt install build-essential
$ sudo apt install libssl1.0-dev
$ sudo apt install libboost-all-dev
$ sudo apt install cmake
```

Listing 6.14: Installation of TinyGarble's dependencies

After this step it's necessary to clone TinyGarble's repository, at <https://github.com/esonghori/TinyGarble>, and build it. For this section it was cloned at the home directory ''.

```
$ git clone https://github.com/esonghori/TinyGarble.git
$ cd TinyGarble/
$ ./configure
$ cd bin/
$ make
```

**Listing 6.15:** Configuration and compilation of TinyGarble. 'Sudo ./configure' and 'sudo make' may be needed.

To be able to use TinyGarble, a RTL Synthesis tool is also needed, to translate Verilog files into Netlist ones. In the example showed later, Yosys-abc is used. Which can be installed on Ubuntu 15.04 or higher with:

```
$ sudo apt-get install yosys
```

**Listing 6.16:** Installation of Yosys-abc for Ubuntu 15.04>

If on Ubuntu 14.04 or lower, a repository needs to be added first:

```
$ sudo add-apt-repository ppa:saltmakrell/ppa
$ sudo apt update
$ sudo apt install yosys
```

**Listing 6.17:** Installation of Yosys-abc for Ubuntu 14.04<

### 6.25.2 Usage

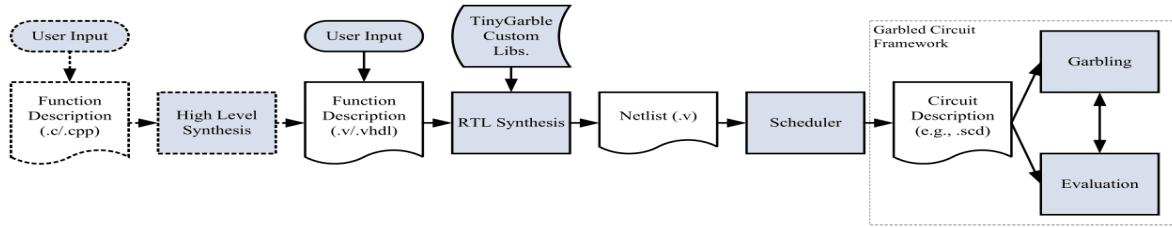


Figure 6.456: TinyGarble Global Flow[1]

Following TinyGarble's global flow showed on the image above, it's possible to see that a C/C++ or Hardware Design Language (HDL) input is necessary. The former is slightly worse in terms of performance, since it needs an extra step of synthesis, with the help of a High Level Synthesis (HLS) tool, like Spart or Xilinx Vivado for C, to compile the C/C++ file into HDL, causing the end SCD file to have more logical gates than when using HDL input directly. This section mostly covers Verilog, since it's the most direct input, if VHDL input is needed, check the subsection UsingVHDL.

When writing the input file, some attention to the circuit's ports is needed, since TinyGarble's V2SCD (Netlist Verilog to SCD converter) accepts netlist files with a special format. Below are the ports that can be used and what they represent.

- clk: clock cycle
- rst: active high reset
- g\_init: garbler's initial values, it's read only at the first clock, so it has to be saved on some register at the first rising edge
- e\_init: evaluator's initial values, same as above
- g\_input: garbler's input, it's read at every clock, so if the input was defined at N bits, and the circuit is executed X times, the total size of the input will be N \* X bits, and each clock will shift the g\_input to the left N bits;
- e\_input: evaluator's input, same as above
- o: output, same as above, but TinyGarble executable has the option to show all the outputs of each clock concatenated, separated, or only show the last clock output

Below it's an example of an and\_gate written in verilog with the format above.

```
module and_gate (input g_input, e_input, output o);
    assign o = g_input & e_input;
endmodule
```

Listing 6.18: and\_gate.v

Using TinyGarble's custom library, "asic\_cell\_yosys\_extended.lib", located at TinyGarble/circuit\_synthesis/lib/, and assuming that the Verilog file is in a folder named "circuits" located at the home directory ' ', the compilation for the circuit can be done with the following instructions:

```
host@ubuntu:~$ yosys
yosys> read_verilog TinyGarble/circuit_synthesis/syn_lib/*.v
yosys> read_verilog circuits/and_gate.v
yosys> hierarchy -check -top and_gate
yosys> proc; fsm; flatten; opt;
yosys> techmap; opt;
yosys> dfplibmap
-liberty TinyGarble/circuit_synthesis/lib/asic_cell_yosys_extended.lib
yosys> abc
-liberty TinyGarble/circuit_synthesis/lib/asic_cell_yosys_extended.lib
-script TinyGarble/circuit_synthesis/lib/script.abc;
yosys> opt; clean;
yosys> opt_clean -purge
yosys> stat
-liberty TinyGarble/circuit_synthesis/lib/asic_cell_yosys_extended.lib
yosys> write_verilog -noattr -noexpr sum_syn_yos.v
```

Listing 6.19: Yosys instructions to compile the combinational circuit Verilog file into a Netlist File. 'Sudo yosys' may be needed.

The first 'read\_verilog' command is optional, it loads some circuits described in Verilog modules, to see which modules are available check 'TinyGarble/circuit\_synthesis/syn\_lib'. For this case in particular no exterior module was needed, so the command won't really do anything. The 'stat' command is only used to show statistics of the final circuit configuration, optional as well. These instructions should work for any combinational or sequential circuit.

The inscructions can be saved in a file ending with '.yos', and be used as a script in yosys, as the following shows:

```
host@ubuntu:~$ yosys
yosys> script file.yos
```

Listing 6.20: Running yosys instructions contained in a script file.

Following the instructions the Netlist file below should be generated.

```
module and_gate( g_input, e_input, o);
  input g_input;
  input e_input;
  output o;
  AND _0_ (
    .A(e_input),
    .B(g_input),
    .Z(o)
  );
endmodule
```

Listing 6.21: and\_gate\_netlist.v

The Netlist file can then be converted into a SCD file with the instruction below:

```
host@ubuntu~$ ./TinyGarble/bin/scd/V2SCD_Main
-i circuits/and_gate_netlist.v-o circuits/and_gate.scd
```

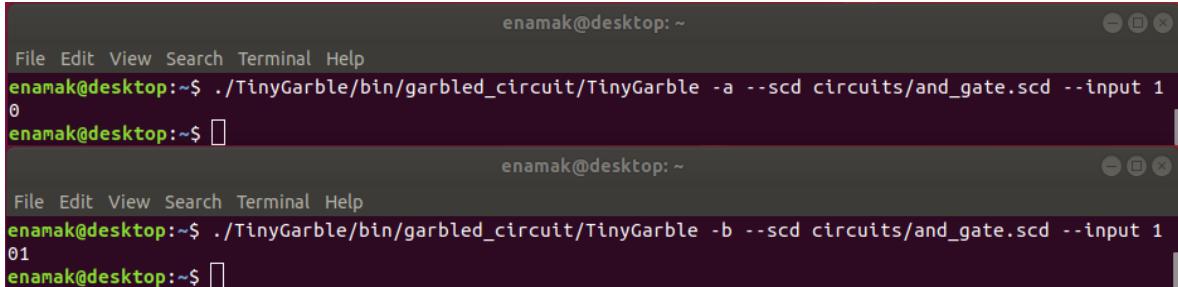
Listing 6.22: Installation of Yosys-abc

Before executing the SCD file, it can be tested with TinyGarble's SCD\_Evaluator\_Main located at TinyGarble/bin/scd. The '-c' option is the number of cycles, and the output\_mode differs the way the output is shown, 0 for concatenated(from all clocks), 1 for separated(for each clock), and 2 to show only the last clock output.

```
host@ubuntu~$ ./TinyGarble/bin/scd/SCD_Evaluator_Main
-i circuits/and_gate -c 1 --g_input 1 --e_input 0 --output_mode 0
```

Listing 6.23: Testing a SCD file

The SCD file can now be executed by both parties, "Alice" and "Bob", as shown on the upper terminal and lower terminal of the following images, respectively. For this examples the SCD file was located at TinyGarble/bin/garbled\_circuits, same directory where the instructions were executed.



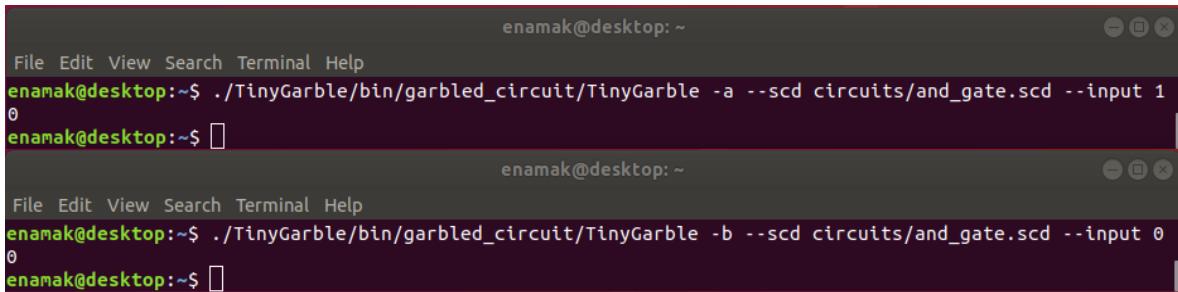
```

enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a --scd circuits/and_gate.scd --input 1
0
enamak@desktop:~$ 

enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b --scd circuits/and_gate.scd --input 1
01
enamak@desktop:~$ 

```

Figure 6.457: "And" function executed by Alice and Bob, with inputs 1 and 1, respectively.



```

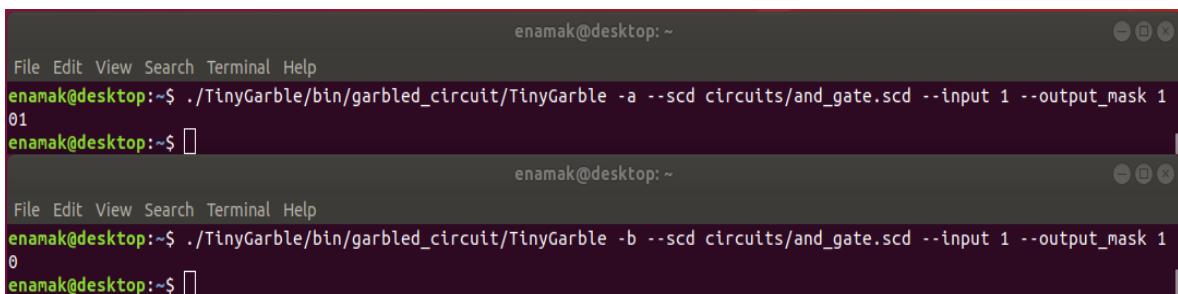
enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a --scd circuits/and_gate.scd --input 1
0
enamak@desktop:~$ 

enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b --scd circuits/and_gate.scd --input 0
0
enamak@desktop:~$ 

```

Figure 6.458: "And" function executed by Alice and Bob, with inputs 1 and 0, respectively.

The option '`output_mask`' allows to choose if the output will appear on Alice or Bob, 0 and 1, respectively, beeing the default value 0.



```

enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a --scd circuits/and_gate.scd --input 1 --output_mask 1
01
enamak@desktop:~$ 

enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b --scd circuits/and_gate.scd --input 1 --output_mask 1
0
enamak@desktop:~$ 

```

Figure 6.459: "And" function executed by Alice and Bob, with inputs 1 and 1, respectively. Example of `output_mask` option.

By default the information about the computation is stored in a log file, in the same folder where the TinyGarble executable is located (also works for the V2SCD\_Main and SCD\_Evaluator\_Main programs) . The option 'log2std' allows to print the information in the terminal instead of creating the log file.

```

enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a --scd circuits/xor_4bits.scd --input 0 --log2std
garbled_circuit_main.cpp:268 INFO: Open Alice's server on port: 1234
garbled_circuit_high_mem.cpp:75 INFO: Alice transfer labels time (cc) = 114640317 (cc/bit) = 2.86601e+07
garbled_circuit_high_mem.cpp:421 INFO: Non-secret skipped non-XOR gates = 0 out of 0 (-nan%)
garbled_circuit_high_mem.cpp:427 INFO: Total garbled non-XOR gates = 0
garbled_circuit_high_mem.cpp:430 INFO: Alice communication time (cc) = 18523
garbled_circuit_high_mem.cpp:431 INFO: Alice garbling time (cc) = 3811
garbled_circuit_main.cpp:282 INFO: Alice's output = 0
garbled_circuit_main.cpp:283 INFO: Total Alice time (cc) = 115303793
0
enamak@desktop:~$ 

enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b --scd circuits/xor_4bits.scd --input F --log2std
garbled_circuit_main.cpp:304 INFO: Connect Bob's client to Alice's server on 127.0.0.1:1234
garbled_circuit_high_mem.cpp:118 INFO: Bob transfer labels time (cc) = 114498193 (cc/bit) = 2.86245e+07
garbled_circuit_high_mem.cpp:705 INFO: Bob communication time (cc) = 4222
garbled_circuit_high_mem.cpp:706 INFO: Bob evaluation time (cc) = 2457
garbled_circuit_main.cpp:319 INFO: Bob's output = 0F
garbled_circuit_main.cpp:320 INFO: Total Bob time (cc) = 115510972
OF
enamak@desktop:~$ 

```

Figure 6.460: "Xor" function with 4 bits executed by Alice and Bob, with inputs 0 and F, respectively. Example of option 'log2std'.

### 6.25.2.1 Using VHDL

If the desirable input is VHDL, this next tool, V2VHDL, will be necessary to translate it into Verilog, which is the input supported by the RTL Synthesis Tool mentioned before. Before cloning vhd2vl at <https://github.com/ldoolitt/vhd2vl> and building it, some dependencies have to be installed:

```

$ sudo apt install flex
$ sudo apt install bison
$ git clone https://github.com/ldoolitt/vhd2vl
$ cd vhd2vl/src
$ make
$ sudo cp vhd2vl /usr/local/bin

```

Listing 6.24: Installation of VHD2VL

To use it simply run:

```
$ vhd2vl and_gate.vhd and_gate.v
```

Listing 6.25: Translation of VHDL file into Verilog

### 6.25.3 Programming

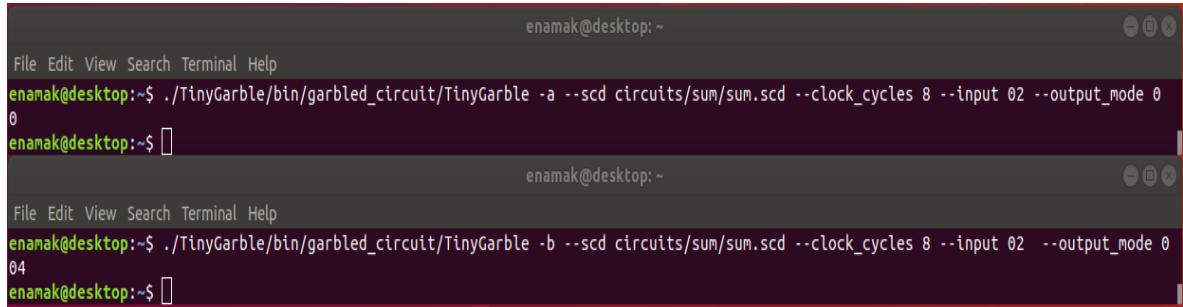
#### 6.25.3.1 Folder Layout

1. a23: Low level functions wrote in C.
2. bin: Created when building TinyGarble. Holds executables and examples of netlist and scd files.
3. circuit\_synthesis: Holds netlist and RTL Verilog files and the custom libs to use with Yosys. Examples below.
4. crypto: Holds C++ files for the Oblivious Transfer and their data structure, 'block'.
5. garbled\_circuit: Holds C++ files to deal with the garbling and evaluation of a circuit.
6. scd: Holds C++ files of the SCD Converter and Evaluator mentioned before, and of the netlist Parser.
7. tcip: Holds C++ files that deal with server connection.
8. util: Holds C++ support files.

This subsection will show an example of a function described in a sequential circuit, more specifically the sum function located at TinyGarble/circuit\_synthesis/sum. For this example the CC parameter was set to the same as N, otherwise if kept at 1 it would be a combinational circuit. If CC is set to the same as N, each clock cycle will read one bit from the input, and also output 1 bit. So in this case if the 'clock\_cycle' option of TinyGarble is set to 8, it would read 8 bits total from 'g\_input' and 'e\_input', and would write 8 bits to 'o'. If CC was set to 2 for example, each cycle would read and write N/2 bits, if set to N/2, each cycle would read and write 2 bits. If the input given is smaller than expected size, it will be read as 0 on the clock cycles were the input is non-existent.

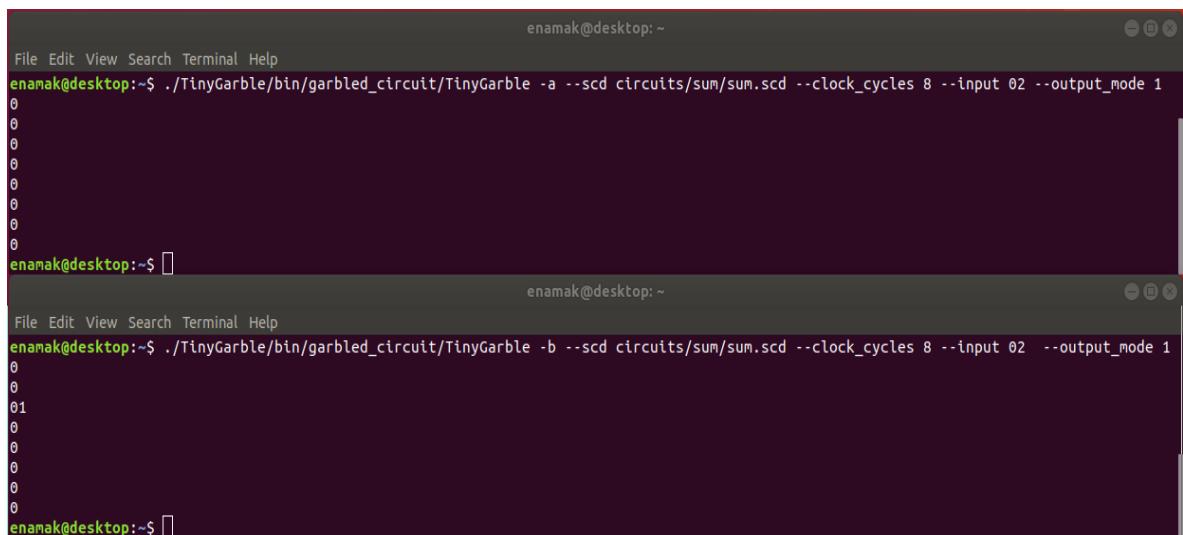
After doing the steps explained at 'Using' section, (with the respective names for the files), the sum.scd file can be executed as showed before with the and\_gate.scd.

When working with sequential circuits, we may want the output of each clock to be printed concatenated, separated (starting from the smallest bit), or print only the last clock output, and for that the option 'output\_mode' can be used, with values 0, 1 and 2, respectively, being the default 0 (it also works with SCD\_Evaluator\_Main program). The output is printed in hexadecimal format.



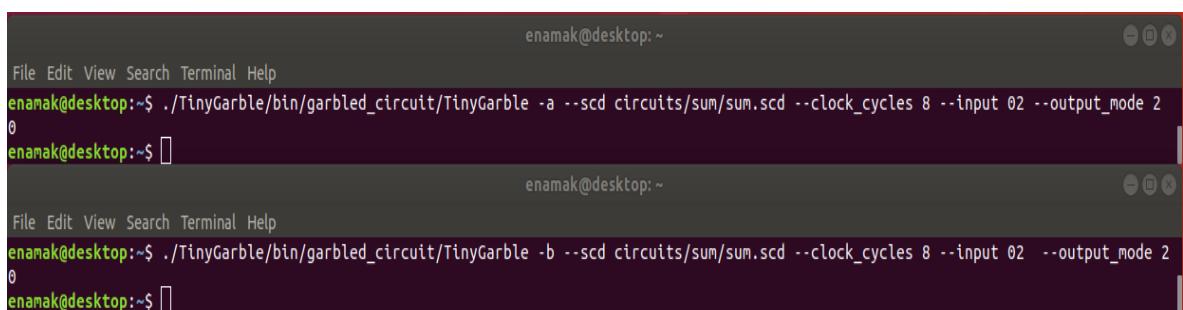
```
enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a --scd circuits/sum/sum.scd --clock_cycles 8 --input 02 --output_mode 0
0
enamak@desktop:~$ 
enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b --scd circuits/sum/sum.scd --clock_cycles 8 --input 02 --output_mode 0
04
enamak@desktop:~$ 
```

Figure 6.461: "Sum" function with 8 bits/clocks executed by Alice and Bob, with inputs 02 and 02, respectively. Printing output concatenated(8 bits).



```
enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a --scd circuits/sum/sum.scd --clock_cycles 8 --input 02 --output_mode 1
0
0
0
0
0
0
0
0
enamak@desktop:~$ 
enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b --scd circuits/sum/sum.scd --clock_cycles 8 --input 02 --output_mode 1
0
0
01
0
0
0
0
0
enamak@desktop:~$ 
```

Figure 6.462: "Sum" function with 8 bits/clocks executed by Alice and Bob, with inputs 02 and 02, respectively. Printing output separated(1 bit each cycle).



```
enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a --scd circuits/sum/sum.scd --clock_cycles 8 --input 02 --output_mode 2
0
0
enamak@desktop:~$ 
enamak@desktop:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b --scd circuits/sum/sum.scd --clock_cycles 8 --input 02 --output_mode 2
0
enamak@desktop:~$ 
```

Figure 6.463: "Sum" function with 8 bits/clocks executed by Alice and Bob, with inputs 02 and 02, respectively. Printing the last clock cycle output (last bit of output).

### 6.25.3.2 Running on seperate Hosts

To run Bob and Alices in different hosts, the host running Bob has to give an extra option '-s <Alice IP adress>'. To find the ip adress run 'hostname -I' or 'ifconfig' in Alice's terminal. The default port is 1234, to change it add the optionn -p <port> on both Hosts. The example below assumes Alice's IP adress is 192.168.1.3. The '-output\_mask arg <0 or 1>' can also be used, to determine which host shows the output, the default is 0 (Bob), and can be changed to 1 (Alice).

```
alice@ubuntu:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a
-i circuits/and.scd -p 3000 --input 1
bob@ubuntu:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b
-i circuits/and.scd -s 192.168.1.3 --p 3000 --input 1
```

Listing 6.26: Installation of Yosys-abc for Ubuntu 15.04>

### 6.25.3.3 Keys

Since TinyGarble is based on JustGarble, it's garbling scheme will be based on fixed-key AES. Alice generates a random key, 'with the randomBlock() function', that will then be expanded 10 times (128bit AES), thus if a custom key is wanted, 128\*11 bits are needed.

Below is the struct used to represent the AES\_KEY, present at TinyGarble/crypto/aes.h:

```
typedef struct {
    __m128i rd_key[15];
    int rounds;
} AES_KEY;
```

Listing 6.27: AES\_KEY struct

The \_\_m128i variable represents a 128bit register, the rd\_key[0] holds the global\_key, and rd\_key[10:1] will have the expanded keys. rd\_key[15:11] are only used for 192 or 256 bit AES. Rounds represent the number of times the key was expanded (10, in case of 128bit AES).

The following function "AES128KeyExpansion" located at the same file, expands the 128 bit global\_key into more ten 128 bit keys, that are stored in rd\_key[10:1], as mentioned before.

```
inline void AES128KeyExpansion(const unsigned char *userkey, void *key) {
    __m128i x0, x1, x2;
    __m128i *kp = (__m128i *) key;
    kp[0] = x0 = _mm_loadu_si128((__m128i *) userkey);
    x2 = _mm_setzero_si128();
    EXPAND_ASSIST(x0, x1, x2, x0, 255, 1);
    kp[1] = x0;
    EXPAND_ASSIST(x0, x1, x2, x0, 255, 2);
    kp[2] = x0;
    EXPAND_ASSIST(x0, x1, x2, x0, 255, 4);
    kp[3] = x0;
    EXPAND_ASSIST(x0, x1, x2, x0, 255, 8);
    kp[4] = x0;
    EXPAND_ASSIST(x0, x1, x2, x0, 255, 16);
    kp[5] = x0;
    EXPAND_ASSIST(x0, x1, x2, x0, 255, 32);
```

```

kp[6] = x0;
EXPAND_ASSIST(x0, x1, x2, x0, 255, 64);
kp[7] = x0;
EXPAND_ASSIST(x0, x1, x2, x0, 255, 128);
kp[8] = x0;
EXPAND_ASSIST(x0, x1, x2, x0, 255, 27);
kp[9] = x0;
EXPAND_ASSIST(x0, x1, x2, x0, 255, 54);
kp[10] = x0;
}

```

Listing 6.28: AES Expansion function

To use a non AES Key, instead of giving a global key and expand it, all 11 different keys have to be provided, and stored on rd\_key[10:0]. To do that, the intrinsic function '`_mm_loadu_si128`' can be used, as showed in the example below.

```

block keyBlock;
Str2Block("50607080 10203040 0fedcba9 87654391", &keyBlock);
__m128i teste = _mm_loadu_si128((__m128i *)((unsigned char *)&keyBlock));
int32_t *tmp1 = (int32_t *) &teste;
printf("%.8x %.8x %.8x %.8x\n", tmp[3], tmp[2], tmp[1], tmp[0]);

```

Listing 6.29: Storing four ints into a 128bit register. It should ouput "50607080 10203040 0fedcba9 87654321".

The same process has to be repeated for the remaining 10 keys.

#### 6.25.3.4 Moda function

The purpose of this function is to take two inputs from each host, and return 0 if the first inputs of each host summed are greater than the sum of the second inputs, and return 1 otherwise. It outputs 2 if both sums are equal. Examples:

1. a:

```
Alice -> 5,10 Bob -> 9, 1 Output -> 0
```

Listing 6.30: Example of moda inputs and output

2. b:

```
Alice -> 1,3 Bob -> 6, 9 Output -> 1
```

Listing 6.31: Example of moda inputs and output

This function can be both implemented with a combinational and sequential circuit, although with the combinational the maximum size of the inputs has to be set, and cannot be changed between executions. The module 'ADD', which represents a full adder, located at 'TinyGarble/circuit\_synthesis/syn\_lib' (folder mentioned before) was used, so the command `read_verilog TinyGarble/circuit_synthesis/syn_lib/*.v` has to be used.

The following verilog module shows the combinationl circuit:

```

module moda(g_input, e_input, o);

input[31:0] g_input, e_input;
wire[16:0] tmp_0, tmp_1;
output [1:0] o;

ADD
#(
    .N(16)
)
input_0
(
    .A(g_input[31:16]),
    .B(e_input[31:16]),
    .CI(),
    .S(tmp_0[15:0]),
    .CO(tmp_0[16:16])
);
ADD
#(
    .N(16)
)
input_1
(
    .A(g_input[15:0]),
    .B(e_input[15:0]),
    .CI(),
    .S(tmp_1[15:0]),
    .CO(tmp_1[16:16])
);

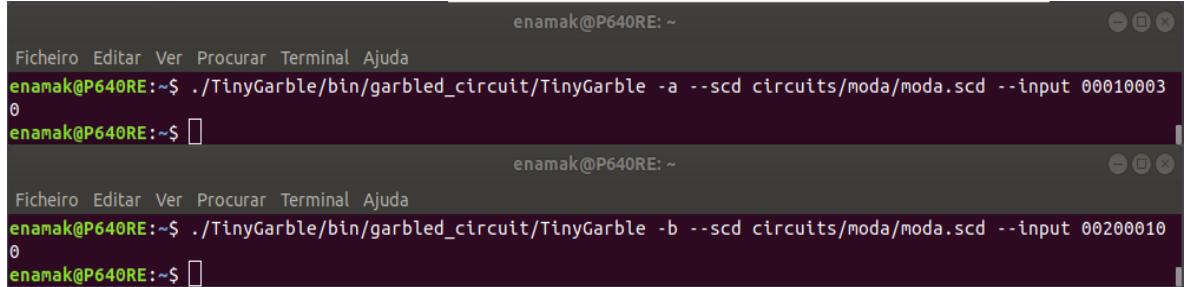
always@ *
begin
    if(tmp_0 > tmp_1)
        o <= 0;
    else if(tmp_0 < tmp_1)
        o <= 1;
    else
        o <= 2;
end
endmodule

```

Listing 6.32: moda.v

Since the garblers and evaluator inputs have 32 bits, values between 0 and 9999 are accepted. The sums will be made in hexadecimal even if the input is intended to be decimal, but the output should not be affected.

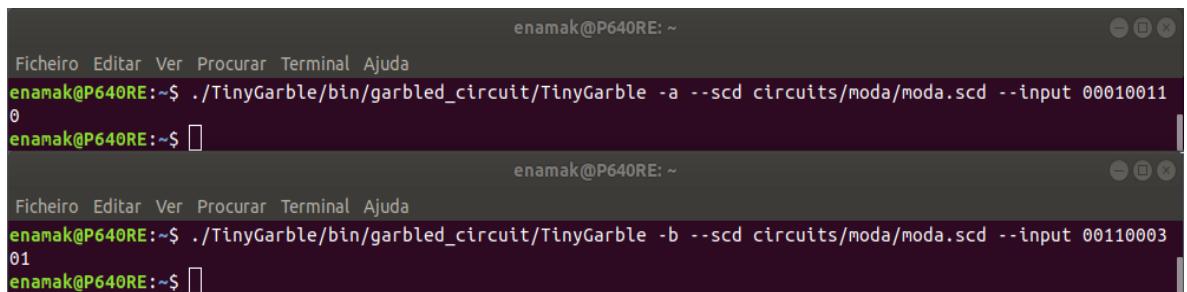
Examples:



```
enamak@P640RE: ~
Ficheiro Editar Ver Procurar Terminal Ajuda
enamak@P640RE:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a --scd circuits/moda/moda.scd --input 00010003
0
enamak@P640RE:~$ [REDACTED]

enamak@P640RE: ~
Ficheiro Editar Ver Procurar Terminal Ajuda
enamak@P640RE:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b --scd circuits/moda/moda.scd --input 00200010
0
enamak@P640RE:~$ [REDACTED]
```

Figure 6.464: "Moda" function executed by Alice and Bob, with inputs 00010003(1,3) and 00200010(20,10), respectively.



```
enamak@P640RE: ~
Ficheiro Editar Ver Procurar Terminal Ajuda
enamak@P640RE:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a --scd circuits/moda/moda.scd --input 00010011
0
enamak@P640RE:~$ [REDACTED]

enamak@P640RE: ~
Ficheiro Editar Ver Procurar Terminal Ajuda
enamak@P640RE:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b --scd circuits/moda/moda.scd --input 00110003
01
enamak@P640RE:~$ [REDACTED]
```

Figure 6.465: "Moda" function executed by Alice and Bob, with inputs 00010011(1,11) and 00110003(11,3), respectively.



```
enamak@P640RE: ~
Ficheiro Editar Ver Procurar Terminal Ajuda
enamak@P640RE:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -a --scd circuits/moda/moda.scd --input 00010011
0
enamak@P640RE:~$ [REDACTED]

enamak@P640RE: ~
Ficheiro Editar Ver Procurar Terminal Ajuda
enamak@P640RE:~$ ./TinyGarble/bin/garbled_circuit/TinyGarble -b --scd circuits/moda/moda.scd --input 00200010
02
enamak@P640RE:~$ [REDACTED]
```

Figure 6.466: "Moda" function executed by Alice and Bob, with inputs 00010011(1,11) and 00200010(20,10), respectively.

### 6.25.3.5 TinyGarble Interface

To ease the computation of the moda function between Alice and Bob, the interface 'STPC' was made. It can be found at [sdf/tiny\\_garble/STPC](#) zip, which contains a folder with the src files, and the folder with the built project. The interface was made with QT Creator, which contains the libraries required for the interface and sockets, so in case any changes have to be made to the interface itself or to the c++ files, it has to be installed. QT Creator 5.11.3 was used, and installed with the '.run' file downloaded at QT's website. After opening the project with the '.pro' file located at the work folder, this one and the build folder can be configured in 'Projects', which are in this case STCP-src and STPC-gui, respectively.

In order to make the app usable in any linux system, <https://github.com/probonopd/linuxdeployqt> was used, more specifically the linuxdeployqt-5-x86\_64.AppImage file, that can be found at the Git's Releases tab. It essentially creates a bundle with all the libraries and other necessary files to execute the app.

```
$ export PATH=/[pathToQt]/[version]/gcc/bin:$PATH
$ chmod a+x linuxdeployqt-5-x86_64.AppImage-x86_64.AppImage
$ ./linuxdeployqt-5-x86_64.AppImage-x86_64.AppImage [PathToExecutable]
```

Listing 6.33: Deploying QT application. Executable in this case is at STPC-gui/STPC

If no error occurred, one should be able to go to the folder where the built project is, and run the executable, in this case './STPC'. When opening the interface, the user has to insert the path to TinyGarble, and the path to the folder containing the 'moda.scd' file, before choosing which host he will be (Alice or Bob).

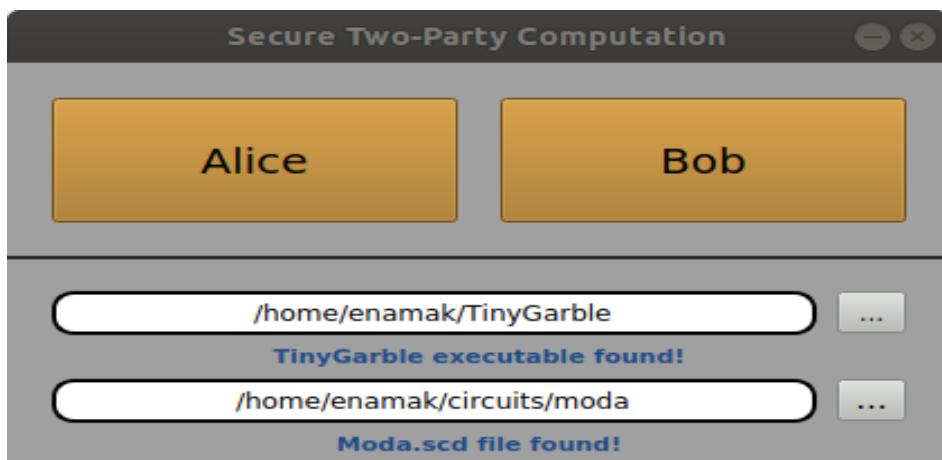


Figure 6.467: Starting window of STPC interface.

If Alice was selected, the user will have to insert the PORT for TinyGarble's connection, and the IP Address of the Key provider and the respective PORT for the connection.

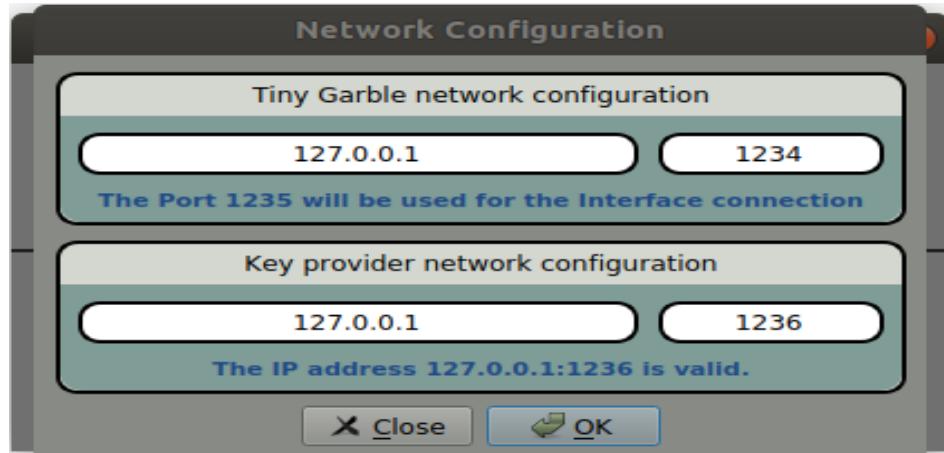


Figure 6.468: Alice network configuration.

For Bob, only the IP Address of Alice and the respective PORT are needed.

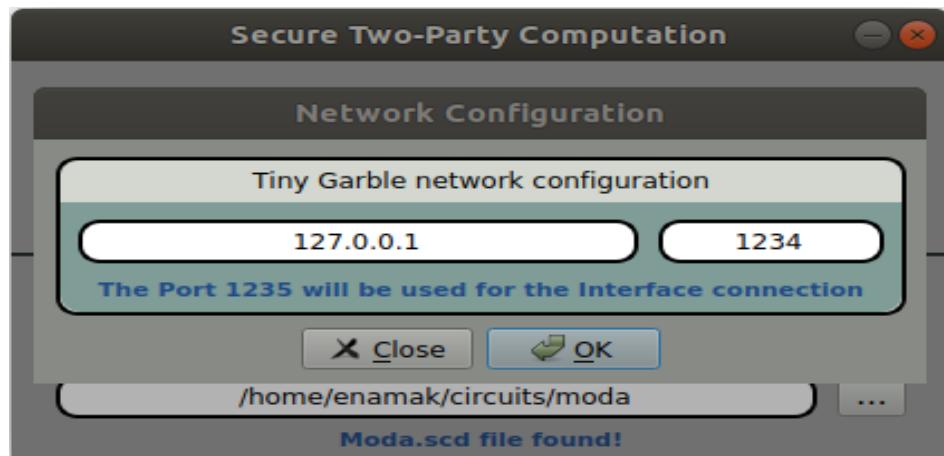


Figure 6.469: Bob network configuration.

After all configuration is done, the final window will appear. In this window the user can request the computation of the moda function, or accept it in case the other party requested it first. The blue frame shows warnings and information about the connection and current state of the application.

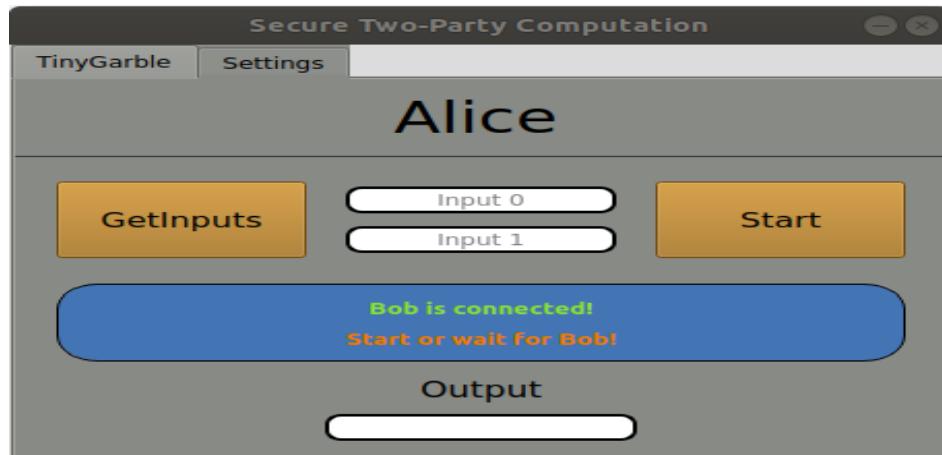


Figure 6.470: STPC main window.

#### 6.25.3.6 Server to provide Keys

This next program, 'key\_server', is a TCP server that provides a random number in binary with N bits, when receiving an 'N' from a client. It will respond with an error message if N is not a number. It is used in the Interface showed above, to retrieve the Key for Alice. Each time a computation is accepted by any of the hosts, Alice will connect to the key provider, send a message 'N', and receive the random number that can then be used as the Key to generate the Garbled Circuit. Qt network libraries were used, so Qt will need to be installed, and the app can be built and deployed the same way the interface app is.

```
$ ./key_server [pathToKeysTxt] [PORT]
```

Listing 6.34: Deploying QT application

#### 6.25.4 Open Issues

1. No free High Level Synthesis was yet found, so impossible to run examples with C or C++ as input.

## References

- [1] Ebrahim M. Songhori et al. "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits". In: (2015).



## 7.1 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 7.1: ADC input parameters

### Methods

```

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

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

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

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

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

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

void setQuantizer.MaxValue(double maxvalue) B03.setMaxValue(maxvalue);
B04.setMaxValue(maxvalue);

void setQuantizer.MinValue(double minvalue) B03.setMinValue(minvalue);
B04.setMinValue(minvalue);

```

### Functional description

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

**Input Signals**

**Number:** 2

**Output Signals**

**Number:** 2

**Type:** Electrical complex signal

**Examples**

**Sugestions for future improvement**

## 7.2 Add

|                    |   |          |
|--------------------|---|----------|
| <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)

### 7.3 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 7.2: Arithmetic encoder block input parameters.

#### Methods

```
bool runBlock(void)

void initialize(void);

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

#### Functional description

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

**Input Signals**

**Number:** 1

**Output Signals**

**Number:** 1

**Type:** binary

**Examples**

**Sugestions for future improvement**

## 7.4 Arithmetic Decoder

|                    |   |                         |
|--------------------|---|-------------------------|
| <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 7.3: Arithmetic decoding block input parameters.

### Methods

```
bool runBlock(void)

void initialize(void);

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

### Functional description

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

**Input Signals**

**Number:** 2

**Output Signals**

**Number:** 2

**Type:** Electrical complex signal

**Examples**

**Sugestions for future improvement**

## 7.5 Alice QKD

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

### Input Parameters

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

### Methods

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

### Functional description

This block receives a sequence of binary numbers (1's or 0's) and a clock signal which will set the rate of the signals produced to generate single polarized photons. The real discrete time signal **SA\_1** is generated based on the clock signal and the real discrete time signal **SA\_2** is generated based on the random sequence of bits received through the signal **NUM\_A**. This last sequence is analysed by the polarizer in pairs of bits in which each pair has a bit for basis choice and other for direction choice.

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

### Input Signals

**Number** : 3

**Type** : Binary, Real Continuous Time and Messages signals.

### Output Signals

**Number** : 3

**Type** : Binary, Real Discrete Time and Messages signals.

**Examples**

**Sugestions for future improvement**

## 7.6 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 7.4: Binary source input parameters

### Methods

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

```
void initialize(void);
bool runBlock(void);
void setMode(AsciiSourceMode m);
AsciiSourceMode const getMode(void);
void setAsciiString(string s);
string getAsciiFilePath();
void setNumberOfCharacters(int n);
int getNumberOfCharacters();
```

### Functional description

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

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

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

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

### Input Signals

**Number:** 0

### Output Signals

**Number:** 1 or more

**Type:** Ascii

## 7.7 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

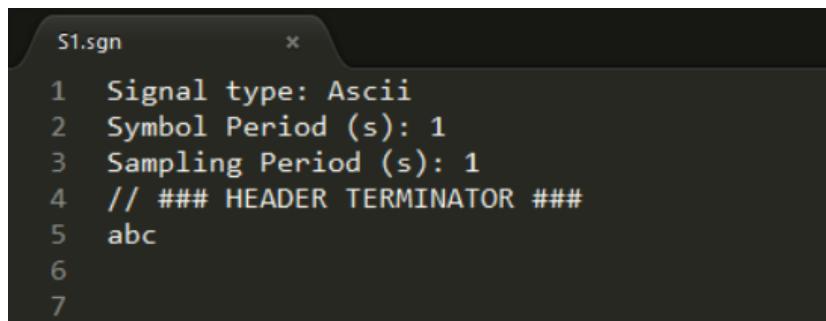
```
AsciiToBinary(vector<Signal *> &InputSig, vector<Signal *> &OutputSig) :Block(InputSig,
OutputSig){};
```

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

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

### Functional description

Figure 7.6 shows an example of an input signal that can be passed as argument. This signal contains three characters: a, b and c. Each character can be represented by 8 bits, according to the Ascii Table. The values to these three characters are respectively: 01100001, 1100010 and 1100011. The block AsciiToBinary will convert the characters a, b and c to their respective binary codes. The resulting output signal will be of type Binary. The output signal to this example, shown in figure 7.5, is the concatenation of the previous binary codes. The full binary sequence is 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 7.1: Ascii signal passed as input to the AsciiToBinary block

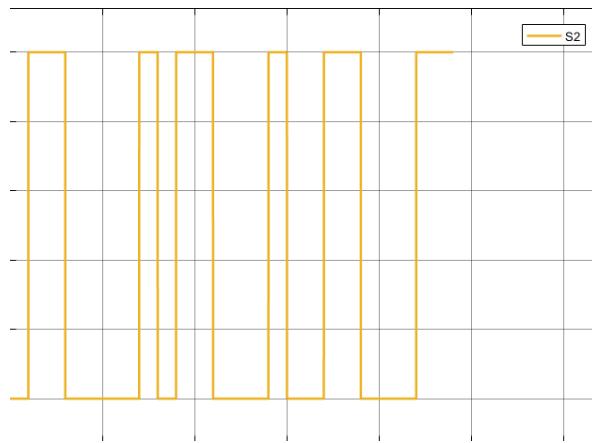


Figure 7.2: Resulting Binary signal from the output of the AsciiToBinary block

### Input Signals

**Number:** 1

**Type:** Ascii

### Output Signals

**Number:** 1

**Type:** Ascii

## 7.8 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> | $\left\{ \left\{ \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}} \right\} \right\}$ |
| 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,

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

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = M_{BS} \begin{bmatrix} A \\ B \end{bmatrix} \quad (7.2)$$

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

### Input Signals

**Number:** 1 or 2

**Type:** Complex

### Output Signals

**Number:** 2

**Type:** Complex

## 7.9 Bit Error Rate

|                    |   |                           |
|--------------------|---|---------------------------|
| <b>Header File</b> | : | bit_error_rate_*.h        |
| <b>Source File</b> | : | bit_error_rate_*.cpp      |
| <b>Version</b>     | : | 20171810 (Daniel Pereira) |
|                    | : | 20181424 (Mariana Ramos)  |

### Input Parameters

| Name      | Type    | Default Value       |
|-----------|---------|---------------------|
| alpha     | double  | 0.05                |
| m         | integer | 0                   |
| lMinorant | double  | $1 \times 10^{-10}$ |

### Methods

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

### Input Signals

**Number:** 2

**Type:** Binary (DiscreteTimeDiscreteAmplitude)

### Output Signals

**Number:** 1

**Type:** Binary (DiscreteTimeDiscreteAmplitude)

### Functional Description

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

for a given probability  $\alpha$ . In version 20181113 instead of the previous binary output string, this block outputs a 0 if the two input samples are equal and 1 if not.

The block allows for mid-reports to be generated, the number of bits between reports is customizable, if it is set to 0 then the block will only output the final report. In version 20180424 this block can operate mid-reports using a CUMULATIVE mode, in which the BER is calculated in a cumulative way taking into account all received bits, coincidences and errors, or in a RESET mode, in which at each  $m$  bits the number of received bits and coincidence bits is reset for the BER calculation.

### Theoretical Description

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

$$\widehat{\text{BER}} = 1 - \frac{K}{N_T}. \quad (7.3)$$

The upper and lower bounds,  $\text{BER}_{\text{UB}}$  and  $\text{BER}_{\text{LB}}$  respectively, are calculated using the Clopper-Pearson confidence interval, which returns the following simplified expression for  $N_T > 40$  [1]:

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

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

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

### Version 20181424

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

- **Input Parameters**

| Name                    | Type                       | Default Value           |
|-------------------------|----------------------------|-------------------------|
| <code>midRepType</code> | <code>MidReportType</code> | <code>Cumulative</code> |

- **Methods**

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

## References

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

## 7.10 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 7.5: 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 7.6: 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 7.3 numbered in this order). Some of these require wrap.

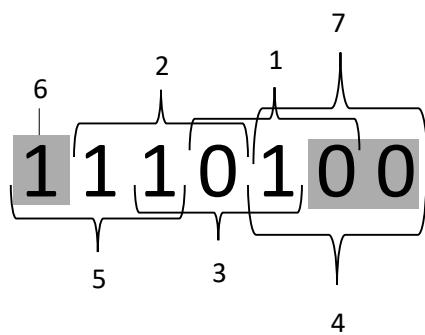


Figure 7.3: Example of a pseudorandom sequence with a pattern length equal to 3.

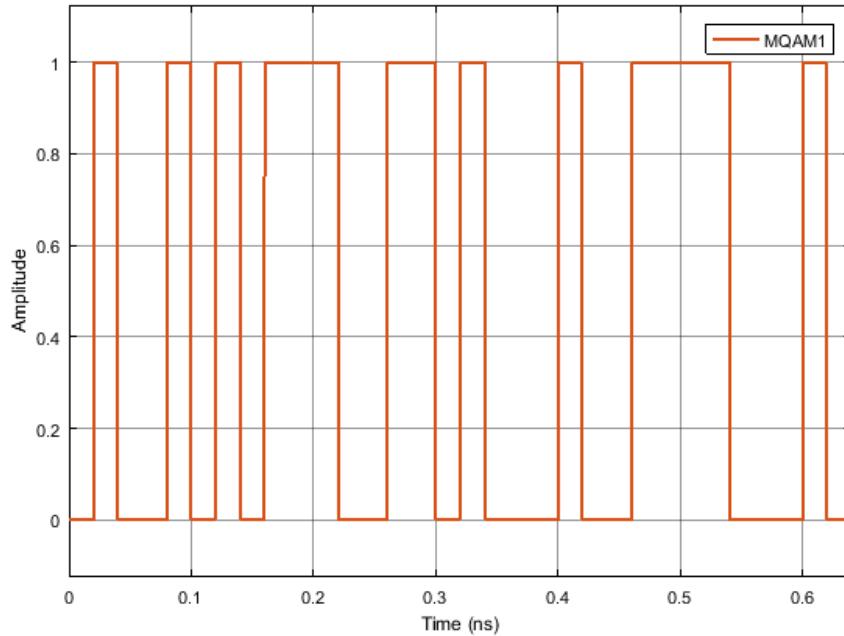


Figure 7.4: Binary signal generated by the block operating in the *Deterministic Append Zeros* mode with a binary sequence 01000...

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

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

### Suggestions for future improvement

Implement an input signal that can work as trigger.

## 7.11 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 7.5 shows an example of an input signal that can be passed as argument. This signal contains the binary representation of three characters: a, b and c. Each character can be represented by 8 bits, according to the Ascii Table. This signal contains the following stream of bits: 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 7.6, contains the characters that can be represented with the previous binary codes.

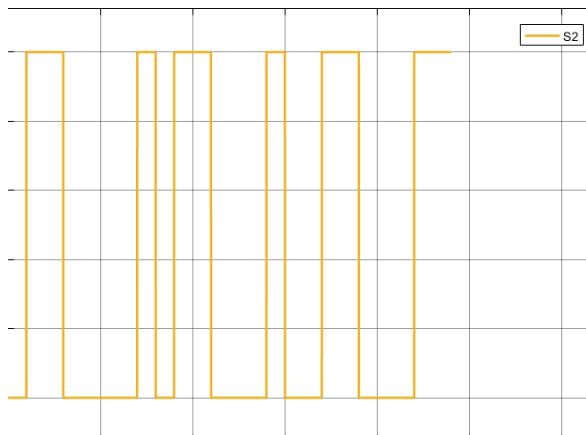
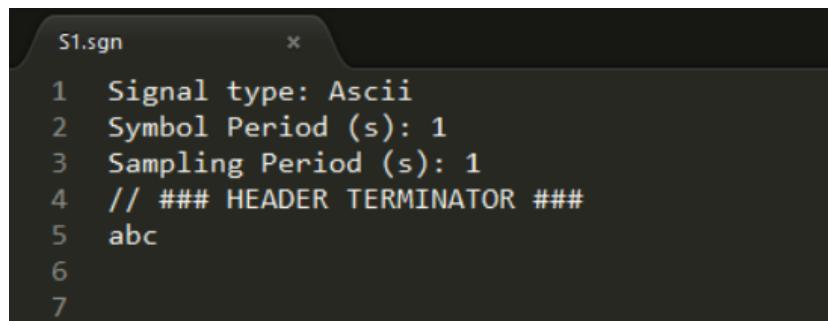


Figure 7.5: Binary signal passed as input to the BinaryToAscii block



The screenshot shows a terminal window titled "S1.sgn" with the following text content:

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

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

### Input Signals

**Number:** 1

**Type:** Ascii

### Output Signals

**Number:** 1

**Type:** Ascii

## 7.12 Bob QKD

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

1.

2.

### **Input Parameters**

- 
- 

### **Methods**

### **Functional description**

### **Input Signals**

### **Examples**

### **Sugestions for future improvement**

### 7.13 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)

## 7.14 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 7.7: 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****Sugestions for future improvement**

## 7.15 Clock\_20171219

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

### Input Parameters

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

### Methods

Clock()

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

```
void initialize(void)
```

```
bool runBlock(void)
```

```
void setClockPeriod(double per) double getClockPeriod()
```

```
void setClockPhase(double pha) double getClockPhase()
```

```
void setSamplingPeriod(double sPeriod) double getSamplingPeriod()
```

### Functional description

#### Input Signals

**Number:** 0

#### Output Signals

**Number:** 1

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

#### Examples

#### Sugestions for future improvement

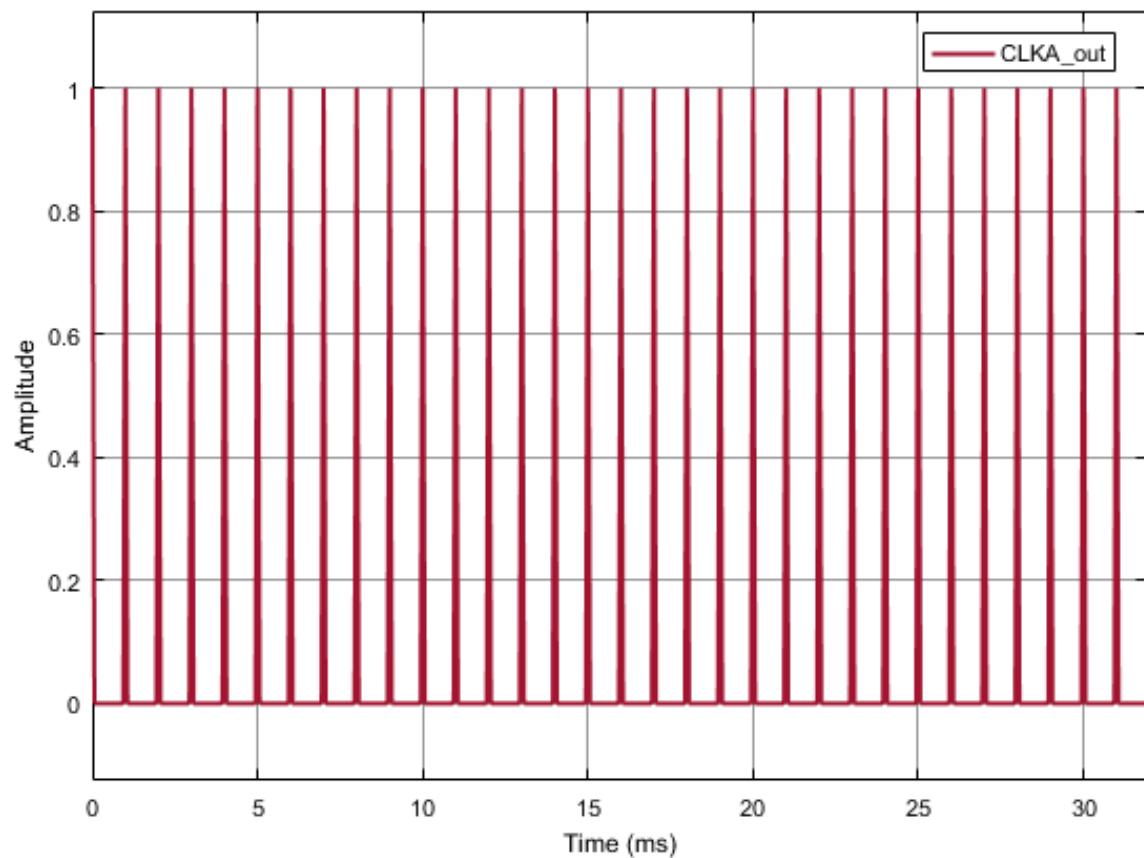


Figure 7.7: Example of the output signal of the clock without phase shift.

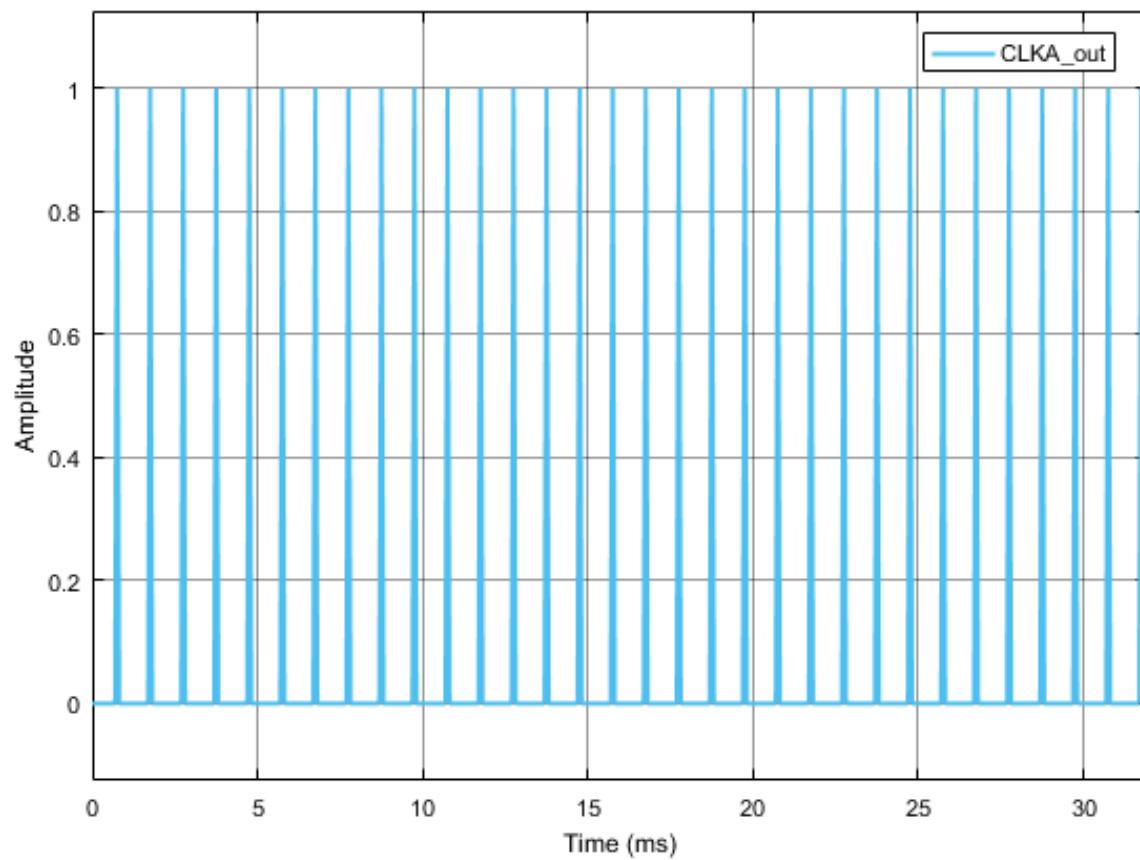


Figure 7.8: Example of the output signal of the clock with phase shift.

## 7.16 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**

**Sugestions for future improvement**

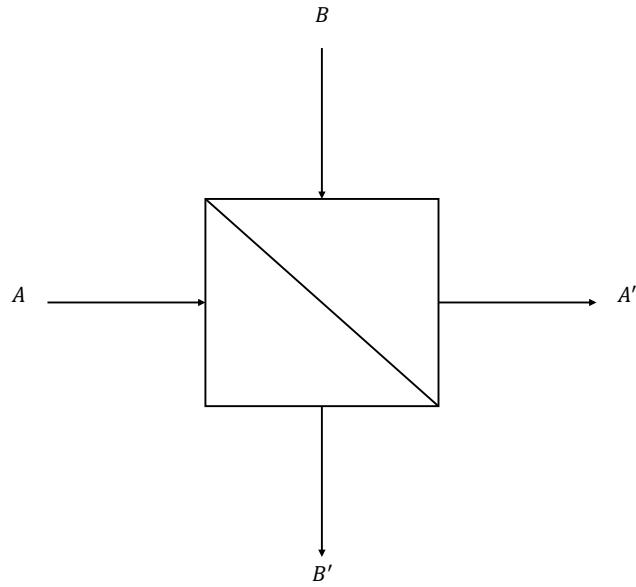


Figure 7.9: 2x2 coupler

### 7.17 Coupler 2 by 2

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

$$\begin{bmatrix} A' \\ B' \end{bmatrix} = \begin{bmatrix} T & iR \\ iR & T \end{bmatrix} \cdot \begin{bmatrix} A \\ B \end{bmatrix} \quad (7.6)$$

Where, A and B represent inputs to the 2x2 coupler and A' and B' represent output of the 2x2 coupler. Parameters T and R represent transmitted and reflected part respectively which can be quantified in the following form,

$$T = \sqrt{1 - \eta_R} \quad (7.7)$$

$$R = \sqrt{\eta_R} \quad (7.8)$$

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

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

## 7.18 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 7.8: 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 7.9: 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.

#### 7.18.0.1 Viterbi-Viterbi Algorithm

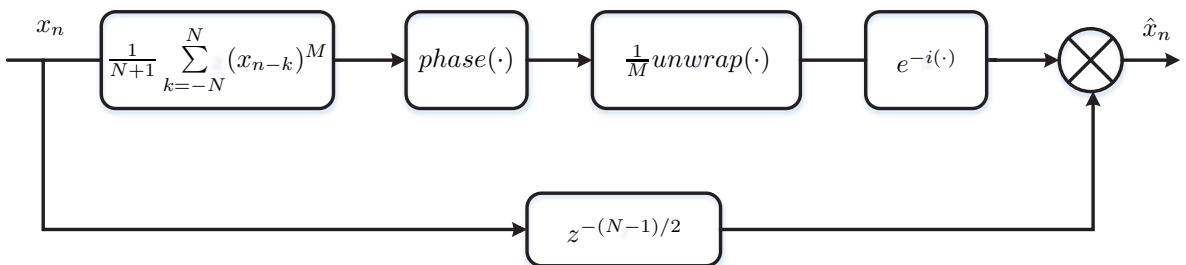


Figure 7.10: Block diagram of Viterbi-Viterbi algorithm for carrier phase recovery.

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

phase is removed by employing the n-th power operation on the received symbols. The algorithm implementation diagram is shown in Figure 7.10, 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.

#### 7.18.0.2 Blind Phase Search Algorithm

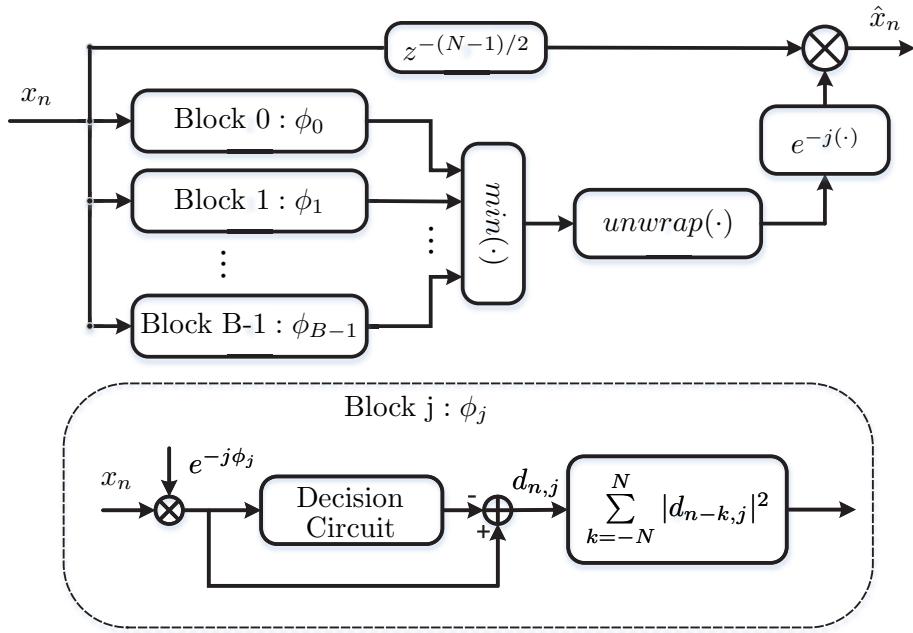


Figure 7.11: Block diagram of blind phase search algorithm for carrier phase recovery.

An alternative to the VV phase noise estimator is the so-called BPS algorithm, in which the operation principle is shown in the Figure 7.11. Firstly, a block of  $2N + 1$  consecutive received symbols is rotated by a number of  $B$  uniformly distributed test phases defined as,

$$\phi_b = \frac{b}{B} \frac{\pi}{2}, b \in \{0, 1, \dots, B - 1\}. \quad (7.9)$$

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

**Input Signals**

**Number:** 1

**Output Signals**

**Number:** 1

**Type:** Electrical complex signal

**Examples**

**Sugestions for future improvement**

## 7.19 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 7.10: 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**

**Sugestions for future improvement**

Extend the decision to higher order modulation format.

## 7.20 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 7.11: 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.

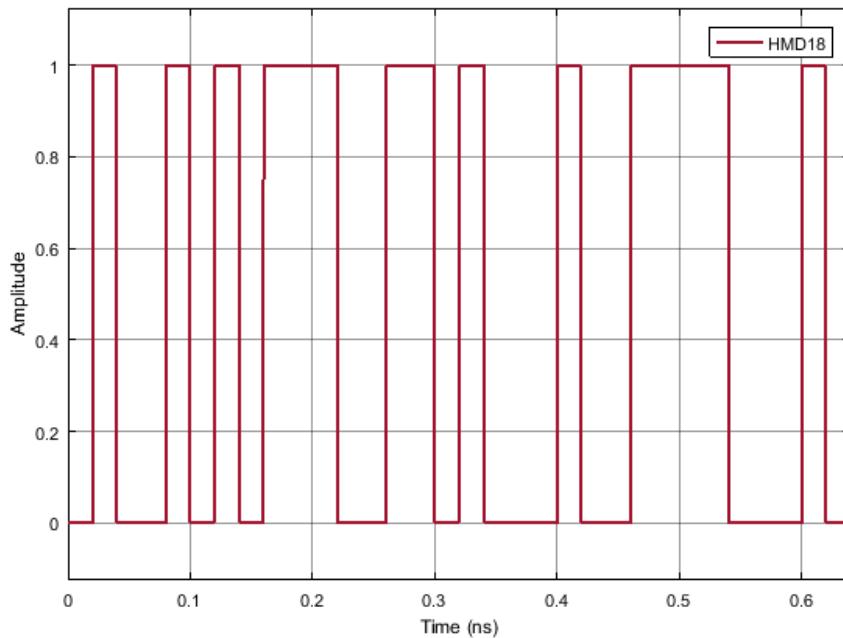


Figure 7.12: Example of the output signal of the decoder for a binary sequence 01. As expected it reproduces the initial bit stream

## Sugestions for future improvement

## 7.21 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 7.12: 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**

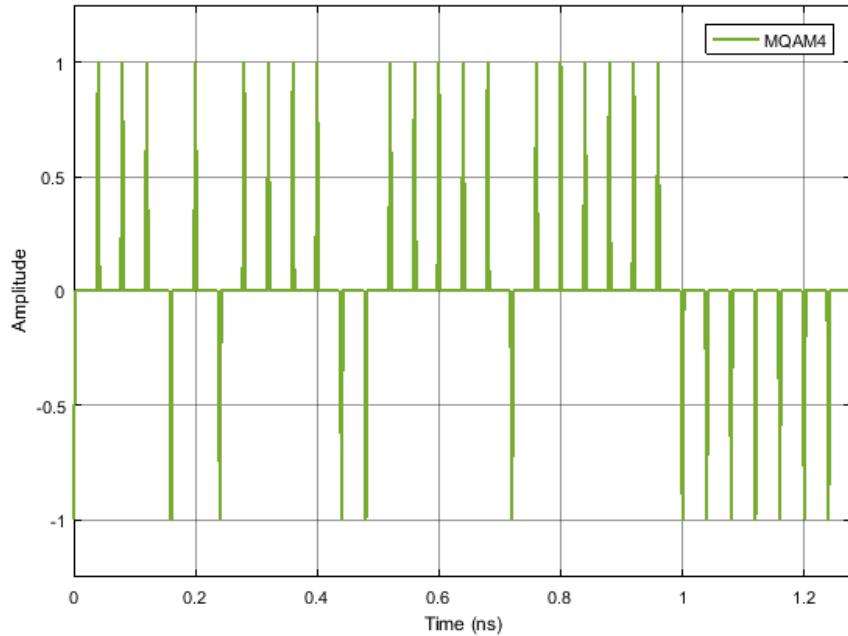


Figure 7.13: Example of the type of signal generated by this block for a binary sequence 0100...

## 7.22 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 7.13: 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**

**Sugestions for future improvement**

## 7.23 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 7.14: 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**

**Sugestions for future improvement**

## 7.24 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.

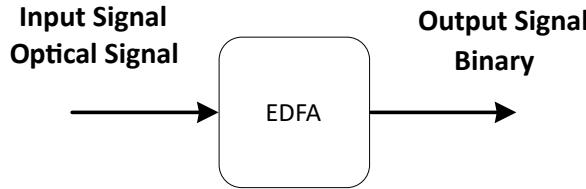


Figure 7.14: Basic configuration of the MQAM receiver

### Functional description

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

This block is made of smaller blocks, and its internal constitution is shown in Figure 7.15.

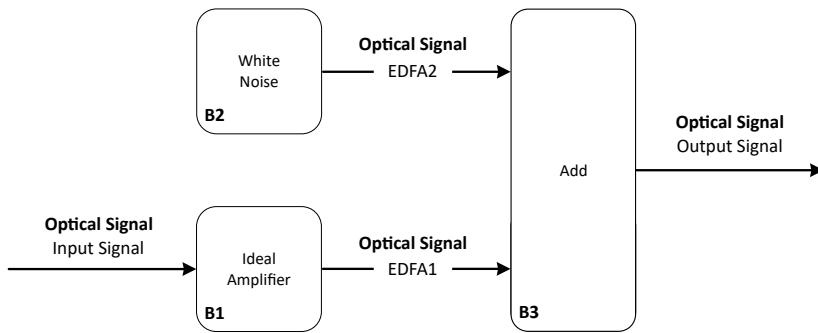


Figure 7.15: Schematic representation of the block homodyne receiver.

### Input parameters

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

Table 7.15: 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 getWavelength(void)

void setDirName(string newDirName);

string getDirName(void)

```

## Input Signals

**Number:** 1

**Type:** Optical signal

## Output Signals

**Number:** 1

**Type:** Optical signal

**Example**

**Sugestions for future improvement**

## 7.25 Electrical Signal Generator

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

### 7.25.1 ContinuousWave

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

#### Input Parameters

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

#### Methods

```
ElectricalSignalGenerator() {};
void initialize(void);
bool runBlock(void);
void setFunction(ElectricalSignalFunction fun) ElectricalSignalFunction getFunction()
void setSamplingPeriod(double speriod) double getSamplingPeriod()
void setSymbolPeriod(double speriod) double getSymbolPeriod()
void setGain(double gvalue) double getGain()
```

#### Functional description

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

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

**Input Signals**

**Number:** 0

**Type:** No type

**Output Signals**

**Number:** 1

**Type:** TimeContinuousAmplitudeContinuous

**Examples****Sugestions for future improvement**

Implement other functions according to the needs.

## 7.26 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)

## 7.27 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

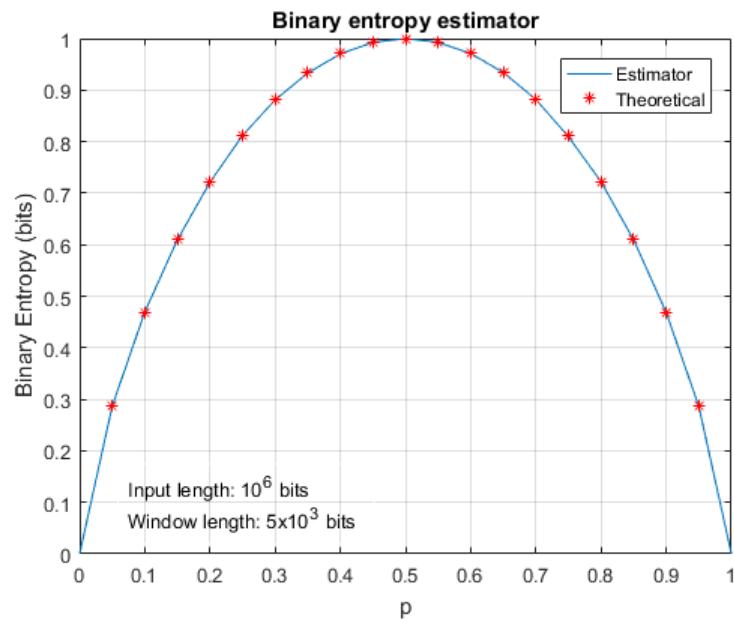
**Type:** Binary Stream

### 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



## 7.28 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.

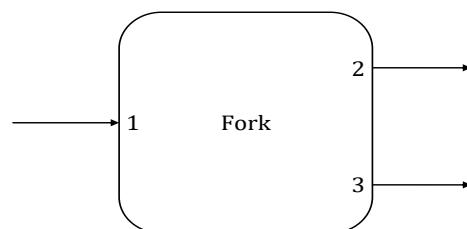


Figure 7.16: Fork

## 7.29 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 7.16: 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**

**Sugestions for future improvement**

## 7.30 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 ( $n$ Bits and  $k$ Bits) that are integers. These variables define the Hamming Algorithm used according to the table below. The values of each valid pair  $(n, k)$  are present in columns  $n$  and  $k$ .

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

### Functional Description

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

### Input Signals

**Number:** 1

**Type:** Binary Signal

### Output Signals

**Number:** 1

**Type:** Binary Signal

### 7.31 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 ( $n$ Bits and  $k$ Bits) that are integers. These variables define the Hamming Algorithm used according to the table below. The values of each valid pair  $(n, k)$  are present in columns  $n$  and  $k$ .

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

#### Functional Description

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

#### Input Signals

**Number:** 1

**Type:** Binary Signal

#### Output Signals

**Number:** 1

**Type:** Binary Signal

## 7.32 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) outputing a binary signal. A simplified schematic representation of this block is shown in figure 7.17.

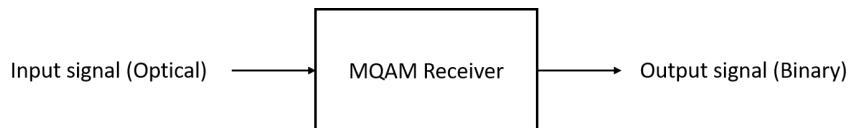


Figure 7.17: Basic configuration of the MQAM receiver

### Functional description

This block accepts one optical input signal and outputs one binary signal that corresponds to the M-QAM demodulation of the input signal. It is a complex block (as it can be seen from figure 7.18) of code made up of several simpler blocks whose description can be found in the *lib* repository.

It can also be seen from figure 7.18 that there's an extra internal (generated inside the homodyne receiver block) input signal generated by the *Clock*. This block is used to provide the sampling frequency to the *Sampler*.

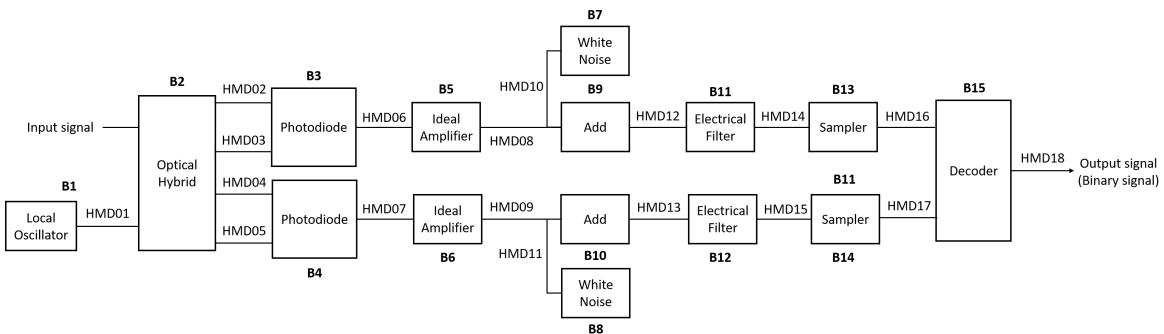


Figure 7.18: Schematic representation of the block homodyne receiver.

## 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 7.33) the input parameters and corresponding functions are summarized.

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

Table 7.17: 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**

**Sugestions for future improvement**

### 7.33 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 7.18: 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

## 7.34 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 7.19: 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

## 7.35 Ideal Amplifier

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

### Input Parameters

| Parameter | Type   | Values | Default         |
|-----------|--------|--------|-----------------|
| gain      | double | any    | $1 \times 10^4$ |

Table 7.20: 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**

**Sugestions for future improvement**

## 7.36 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 blocks 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 7.21: 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

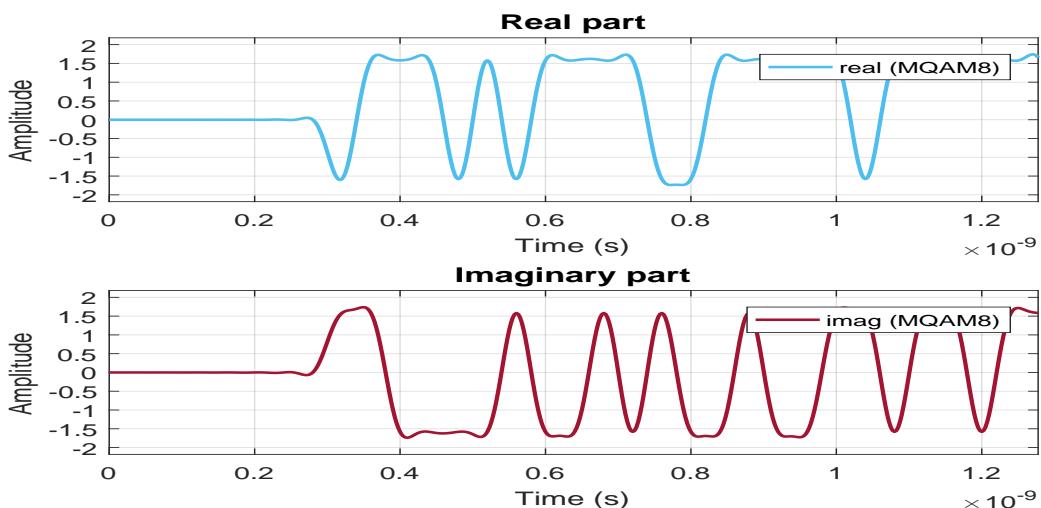


Figure 7.19: Example of a signal generated by this block for the initial binary signal 0100...

**Version 20180828**

**Input Parameters:**

—NA—

**Input Signals:**

**Number:** 1, 2, 3

**Type:** RealValue

**Output Signals:**

**Number:** 4

**Type:** RealValue

**Functional Description**

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

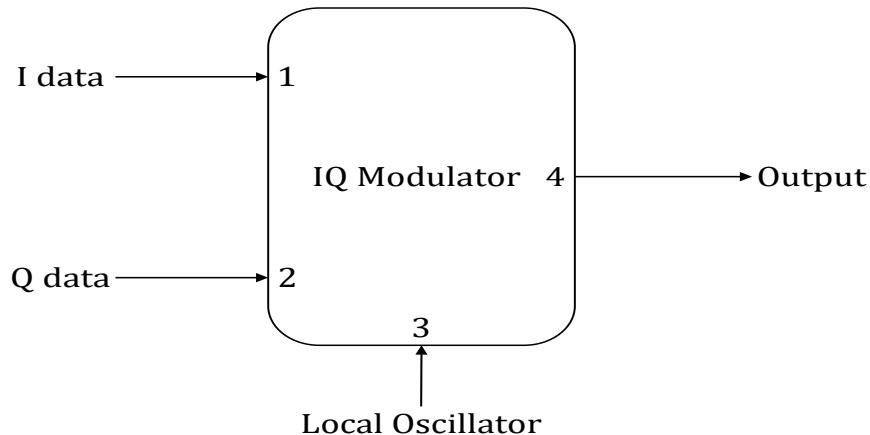


Figure 7.20: IQ Modulator block

**IQ MZM Description**

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

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

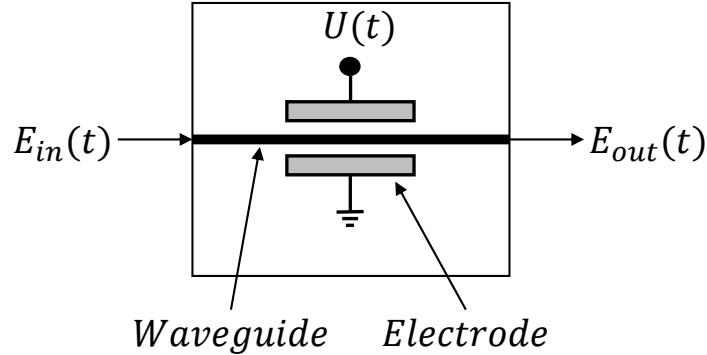


Figure 7.21: Phase Modulator

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,

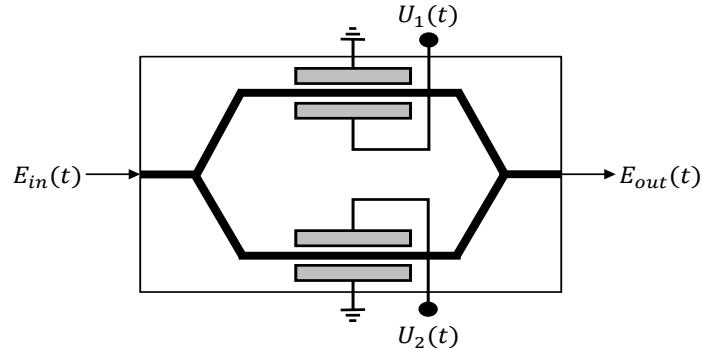


Figure 7.22: Mach-Zehnder Modulator

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

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

The structure of the IQ MZM can be represented shown in Figure ?? where the incoming source light spitted into two portions. The first portion will drive the MZM of the I-channel and other portion will drive MZM Q-channel data. In the Q-channel, before feeding it to the MZM, it passed though the phase modulator to provide a  $\pi/2$  phase shift to the carrier. The output of the MZM combined to form the electrical field  $E_{out}(t)$  [1]. The transfer function of

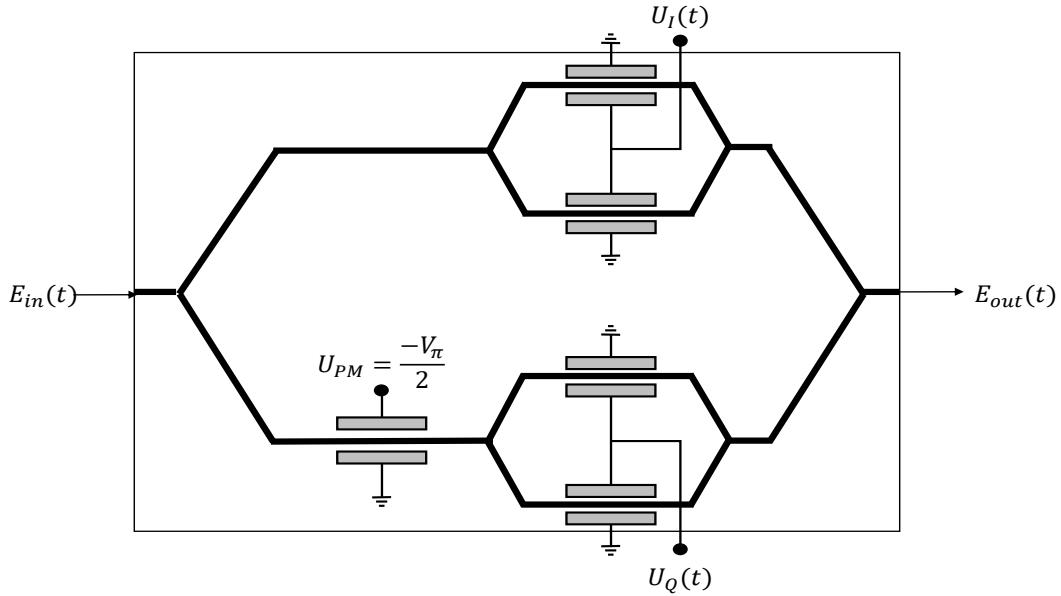


Figure 7.23: IQ Mach-Zehnder Modulator

the IQ MZM can be written as,

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

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

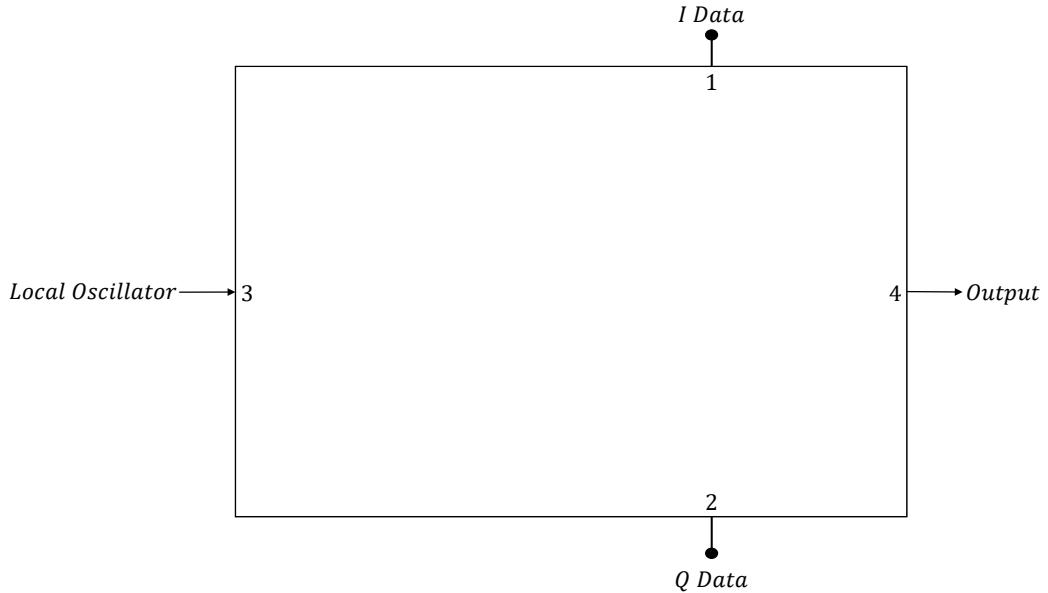


Figure 7.24: Simulation model of the IQ Mach-Zehnder Modulator

## References

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

## 7.37 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 [1].

### Theoretical Description

### Known Issues

## References

- [1] Michel C. Jeruchim, Philip Balaban, and K. Sam Shanmugan. "Simulation of communication systems: modeling, methodology and techniques". In: Springer Science & Business Media, 2006. Chap. 10, pp. 645–654.

## 7.38 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                      | SPEED_OF_LIGHT / 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 7.22: 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 wlength);  
void setFrequency(double freq);  
void setPhase(double lOscillatorPhase);  
void setSignaltoNoiseRatio(double sNoiseRatio);  
void setLaserLinewidth(double laserLinewidth);  
void setLaserRIN(double laserRIN);
```

### Functional description

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

### Input Signals

**Number:** 0

### Output Signals

**Number:** 1

**Type:** Optical signal

**Version 20180828**

## 7.39 Local Oscillator

|                    |   |                      |
|--------------------|---|----------------------|
| <b>Header File</b> | : | local_oscillator.h   |
| <b>Source File</b> | : | local_oscillator.cpp |

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

### Input Parameters

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

Table 7.23: 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 wlenght);

void setPhase(double lOscillatorPhase);

void setLaserLinewidth(double laserLinewidth);

```

```
double getLaserLinewidth();  
  
void setLaserRIN(double LOlaserRIN);  
  
double getLaserRIN();
```

### Functional description

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

**Input Signals**

**Number:** 0

**Output Signals**

**Number:** 1

**Type:** Optical signal

**Examples**

**Sugestions for future improvement**

## 7.40 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:

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

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

This block uses the property 1.9.3 in [1] of Mutual information which tells that it can be determined using the formula:

$$I(X; Y) = H(Y) - H(Y|X). \quad (7.13)$$

Nevertheless, this block estimates the mutual information of binary signals, which means that it estimates the probability of the input bit is 0  $P(X = 0)$  and assumes that the complementary of this probability corresponds to the probability of the input bit is 1  $P(X = 1)$ .  $P(X = 0)$  corresponds to the  $\alpha$  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 7.31 we should calculate the conditional entropy  $H(Y|X)$  and the entropy of the channel outputs  $H(Y)$ . First, lets calculate  $H(Y|X)$ :

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

which means that the conditional entropy depends only on the channel properties and it does not depend 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  $P(Y = 1)$ .

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

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

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

As a result:

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

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

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

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 7.9 for bit error rate calculation. This way, it returns the simplified expression:

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

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

where  $z_{\alpha/2}$  is the  $100(1 - \frac{\alpha}{2})$ 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.

## References

- [1] Krzysztof Wesolowski. *Introduction to digital communication systems*. John Wiley & Sons, 2009.

## 7.41 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 7.24: 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 7.25.

### Input Signals

**Number** : 1

**Type** : Binary (DiscreteTimeDiscreteAmplitude)

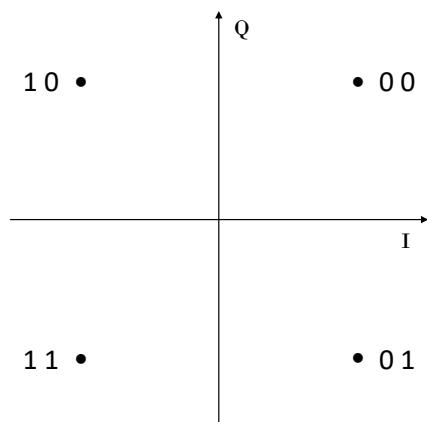


Figure 7.25: Constellation used to map the signal for  $m=4$

### Output Signals

**Number** : 2

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

### Example

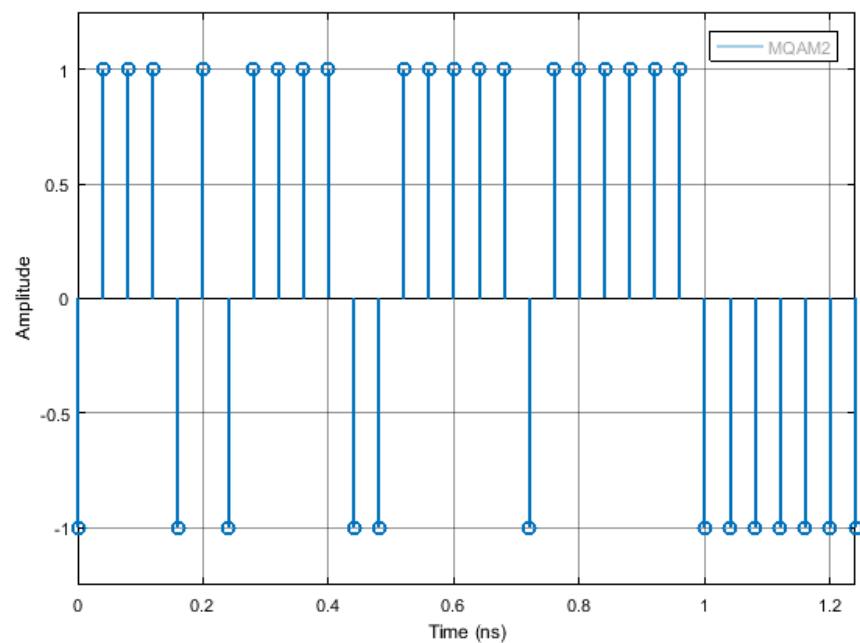


Figure 7.26: Example of the type of signal generated by this block for the initial binary signal 0100...

## 7.42 MQAM Transmitter

|                    |   |                       |
|--------------------|---|-----------------------|
| <b>Header File</b> | : | m_qam_transmitter.h   |
| <b>Source File</b> | : | m_qam_transmitter.cpp |

This block generates a MQAM optical signal. It can also output the binary sequence. A schematic representation of this block is shown in figure 7.27.

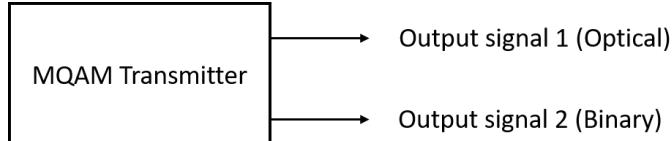


Figure 7.27: Basic configuration of the MQAM transmitter

### Functional description

This block generates an optical signal (output signal 1 in figure 7.28). The binary signal generated in the internal block Binary Source (block B1 in figure 7.28) can be used to perform a Bit Error Rate (BER) measurement and in that sense it works as an extra output signal (output signal 2 in figure 7.28).

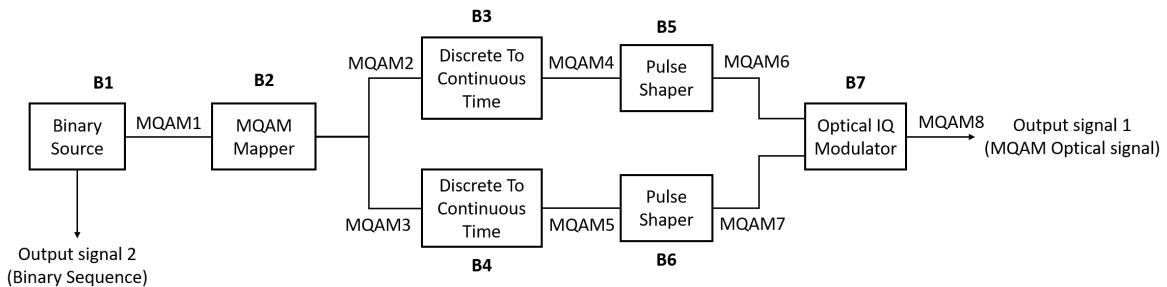


Figure 7.28: Schematic representation of the block MQAM transmitter.

### Input parameters

This block has a special set of functions that allow the user to change the basic configuration of the transmitter. The list of input parameters, functions used to change them and the values that each one can take are summarized in table 7.33.

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

Table 7.25: List of input parameters of the block MQAM transmitter

## Methods

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

void set(int opt);

void setMode(BinarySourceMode m)

BinarySourceMode const getMode(void)

void setProbabilityOfZero(double pZero)

double const getProbabilityOfZero(void)

void setBitStream(string bStream)

```
string const getBitStream(void)

void setNumberOfBits(long int nOfBits)

long int const getNumberOfBits(void)

void setPatternLength(int pLength)

int const getPatternLength(void)

void setBitPeriod(double bPeriod)

double const getBitPeriod(void)

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

void setIqAmplitudes(vector<t_iqValues> iqAmplitudesValues)

vector<t_iqValues> const getIqAmplitudes(void)

void setNumberOfSamplesPerSymbol(int n)

int const getNumberOfSamplesPerSymbol(void)

void setRollOffFactor(double rOffFactor)

double const getRollOffFactor(void)

void setSeeBeginningOfImpulseResponse(bool sBeginningOfImpulseResponse)

double const getSeeBeginningOfImpulseResponse(void)

void setOutputOpticalPower(t_real outOpticalPower)

t_real const getOutputOpticalPower(void)

void setOutputOpticalPower_dBm(t_real outOpticalPower_dBm)

t_real const getOutputOpticalPower_dBm(void)
```

## Output Signals

**Number:** 1 optical and 1 binary (optional)

**Type:** Optical signal

## Example

### Sugestions for future improvement

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

## 7.43 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           |                            |

### 7.43.1 Version 20180118

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

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

### 7.43.2 Version 20180418

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

### Sugestions for future improvement

## 7.44 Alice QKD

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

### Input Parameters

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

### Methods

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

### Functional description

This block receives a sequence of binary numbers (1's or 0's) and a clock signal which will set the rate of the signals produced to generate single polarized photons. The real discrete time signal **SA\_1** is generated based on the clock signal and the real discrete time signal **SA\_2** is generated based on the random sequence of bits received through the signal **NUM\_A**. This last sequence is analysed by the polarizer in pairs of bits in which each pair has a bit for basis choice and other for direction choice.

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

### Input Signals

**Number** : 3

**Type** : Binary, Real Continuous Time and Messages signals.

### Output Signals

**Number** : 3

**Type** : Binary, Real Discrete Time and Messages signals.

**Examples**

**Sugestions for future improvement**

## 7.45 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

## 7.46 Probability Estimator

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

In statistics theory, considering the results space  $\Omega$  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

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

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

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

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

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  $np(1 - p)$ .

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

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

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

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

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:

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

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

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

$$ME = z_{\text{score}} \times \sqrt{\frac{\hat{p}(1 - \hat{p})}{N}} \quad (7.28)$$

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

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

## Input Signals

**Number:** 1

**Type:** Binary

## Output Signals

**Number:** 2

**Type:** Binary

**Type:** txt file

## Examples

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

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

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

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

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

## Sugestions for future improvement

## 7.47 Bob QKD

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

1.

2.

### **Input Parameters**

- 
- 

### **Methods**

### **Functional description**

### **Input Signals**

### **Examples**

### **Sugestions for future improvement**

## 7.48 Eve QKD

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

1.

2.

### **Input Parameters**

- 
- 
- 

### **Methods**

### **Functional description**

### **Input Signals**

### **Examples**

### **Sugestions for future improvement**

## 7.49 Rotator Linear Polarizer

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

### Input Parameters

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

### Methods

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

### Functional description

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

### Input Signals

**Number** : 2

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

### Output Signals

**Number** : 1

**Type** : Photon Stream

**Examples**

**Sugestions for future improvement**

## 7.50 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:

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

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

This block uses the property 1.9.3 in [1] of Mutual information which tells that it can be determined using the formula:

$$I(X; Y) = H(Y) - H(Y|X). \quad (7.31)$$

Nevertheless, this block estimates the mutual information of binary signals, which means that it estimates the probability of the input bit is 0  $P(X = 0)$  and assumes that the complementary of this probability corresponds to the probability of the input bit is 1  $P(X = 1)$ .  $P(X = 0)$  corresponds to the  $\alpha$  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 7.31 we should calculate the conditional entropy  $H(Y|X)$  and the entropy of the channel outputs  $H(Y)$ . First, lets calculate  $H(Y|X)$ :

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

which means that the conditional entropy depends only on the channel properties and it does not depend 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  $P(Y = 1)$ .

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

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

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

As a result:

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

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

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

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 7.9 for bit error rate calculation. This way, it returns the simplified expression:

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

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

where  $z_{\alpha/2}$  is the  $100(1 - \frac{\alpha}{2})$ 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.

## References

- [1] Krzysztof Wesolowski. *Introduction to digital communication systems*. John Wiley & Sons, 2009.

## 7.51 Optical Switch

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

### Input Parameters

No input parameters.

### Methods

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

### Functional description

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

### Input Signals

**Number** : 1

**Type** : Photon Stream

### Output Signals

**Number** : 2

**Type** : Photon Stream

### Examples

### Sugestions for future improvement

## 7.52 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 7.29 shows a schematic representation of this block.

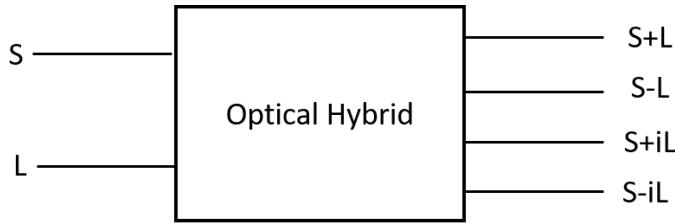


Figure 7.29: Schematic representation of an optical hybrid.

### Input Parameters

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

Table 7.26: Optical hybrid input parameters

### Methods

OpticalHybrid()

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

```
void initialize(void)
```

```
bool runBlock(void)
```

```
void setOutputOpticalPower(double outOpticalPower)
```

```
void setOutputOpticalPower_dBm(double outOpticalPower_dBm)
```

```
void setOutputOpticalWavelength(double outOpticalWavelength)  
void setOutputOpticalFrequency(double outOpticalFrequency)  
void setPowerFactor(double pFactor)
```

### Functional description

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

### Input Signals

**Number:** 2

**Type:** Optical (OpticalSignal)

### Output Signals

**Number:** 4

**Type:** Optical (OpticalSignal)

### Examples

### Sugestions for future improvement

### 7.53 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 7.30. It accepts two optical input signals and outputs one electrical signal. Each photodiode converts an optical signal to an electrical signal. The two electrical signals are then subtracted and the resulting signals corresponds to the output signal of the block.

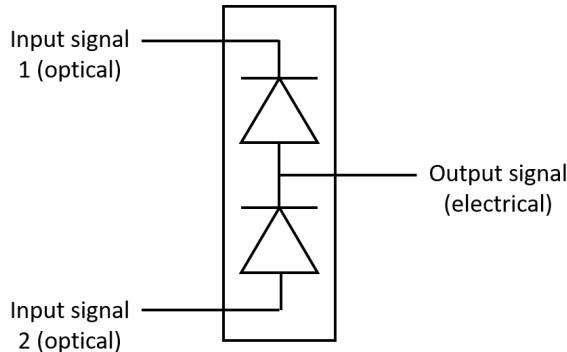


Figure 7.30: Schematic representation of the physical equivalent of the photodiode code block.

#### Input Parameters

- responsivity{1}
- outputOpticalWavelength{ 1550e-9 }
- outputOpticalFrequency{ SPEED\_OF\_LIGHT / wavelength }

#### Methods

Photodiode()

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

```
void initialize(void)
```

```
bool runBlock(void)
```

```
void setResponsivity(t_real Responsivity)
```

### Functional description

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

### Input Signals

**Number:** 2

**Type:** Optical (OpticalSignal)

### Output Signals

**Number:** 1

**Type:** Electrical (TimeContinuousAmplitudeContinuousReal)

### Examples

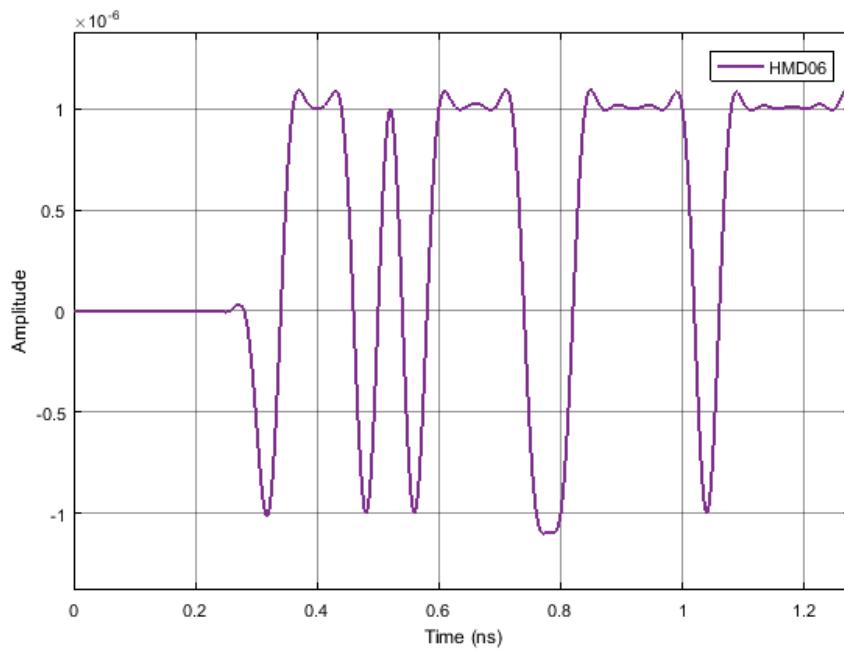


Figure 7.31: Example of the output singal of the photodiode block for a bunary sequence 01

**Sugestions for future improvement**

## 7.54 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.

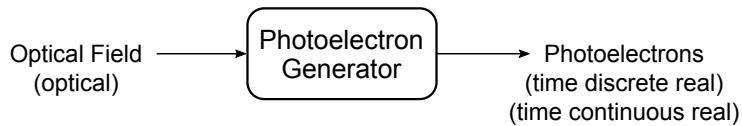


Figure 7.32: Schematic representation of the photoelectron generator code block

### 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 [1]

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

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

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

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

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

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

For each incident photon, there is a probability  $\eta$  of generating a phototransistor. 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 [1]

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

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

## Functional description

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

If the shot-noise is negleted, then the output number of photoelectrons,  $n_e(t)$  in the interval, will be equal to

$$m(t) = \eta \bar{n}(t) \quad (7.43)$$

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

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

## Input Parameters

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

## Methods

PhotoelectronGenerator()

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

```
void initialize(void)
```

```
bool runBlock(void)
```

```
void setEfficiency(t_real efficiency)
```

```
void getEfficiency()
```

```
void setShotNoise(bool shotNoise)
```

```
void getShotNoise()
```

## Input Signals

**Number:** 1

**Type:** Optical (OpticalSignal)

## Output Signals

**Number:** 1

**Type:** Electrical (TimeDiscreteAmplitudeContinuousReal  
TimeContinuousAmplitudeContinuousReal) or

## Examples

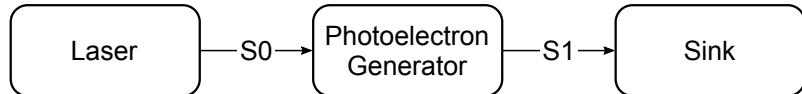


Figure 7.33: Constant power simulation setup

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

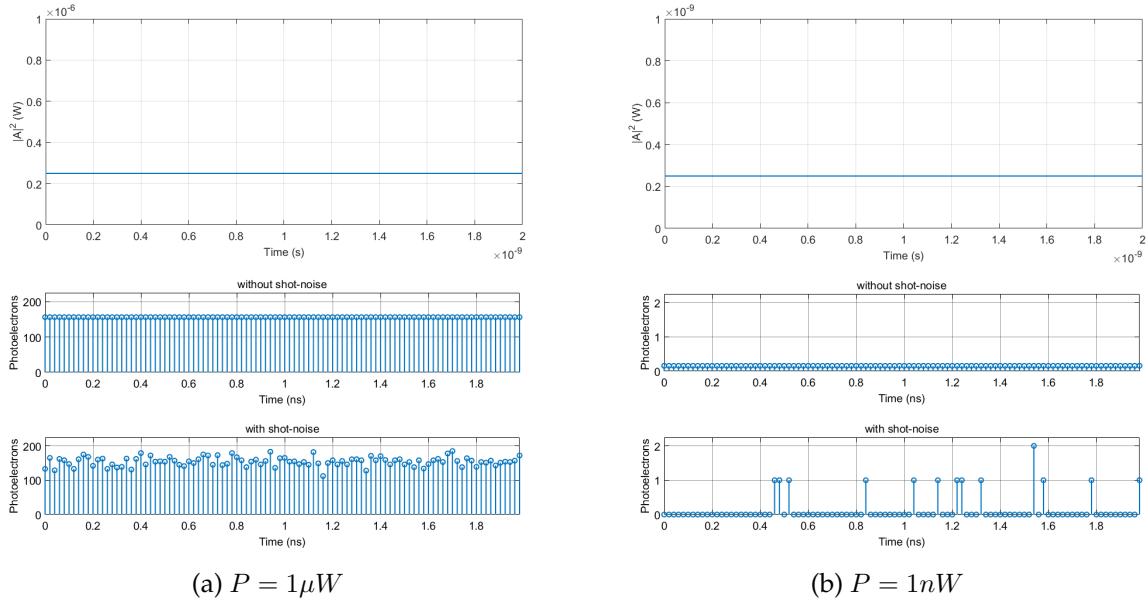


Figure 7.34: Upper plots: input optical squared amplitude on the photoelectron generator; Middle plots: number of output photoelectrons per sample neglecting quantum noise; Bottom plots: number of output photoelectrons per sample considering quantum noise.

Figure 7.34 shows clearly that turning the quantum noise on or off will produce a signal with or without variance, as predicted. If we compare this result with plot (a) in [1], in particular  $P = 1nW$ , we see that they are in conformance, with a slight difference, where a sample has more than one photoelectron. In 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 it's input power, multiple photoelectron generation events will occur during the sample time window. Therefore, to recreate the reference result, we just need to reduce the sample period until the probability of generating more than 1 photoelectron per sample goes to 0.

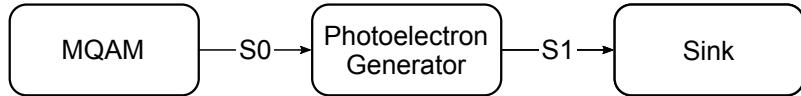


Figure 7.35: Variable power simulation setup

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

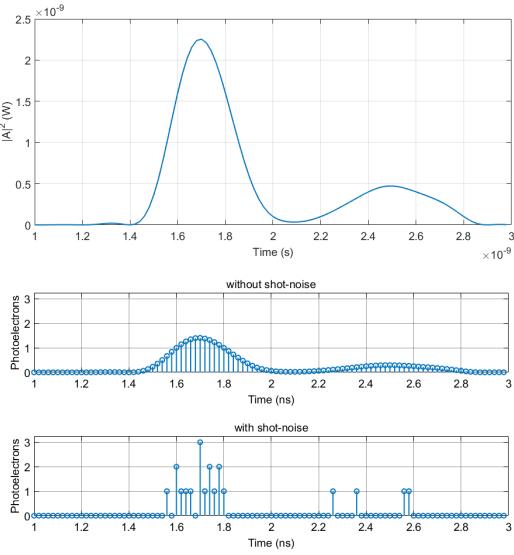


Figure 7.36: Upper plots: input optical squared amplitude on the photoelectron generator; Middle plots: number of output photoelectrons per sample neglecting quantum noise; Bottom plots: number of output photoelectrons per sample considering quantum noise.

Figure 7.36 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 [1], because has already discussed previously, in the high power input samples, we have a great probability of generating many photoelectrons.

### Sugestions for future improvement

## References

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

## 7.55 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

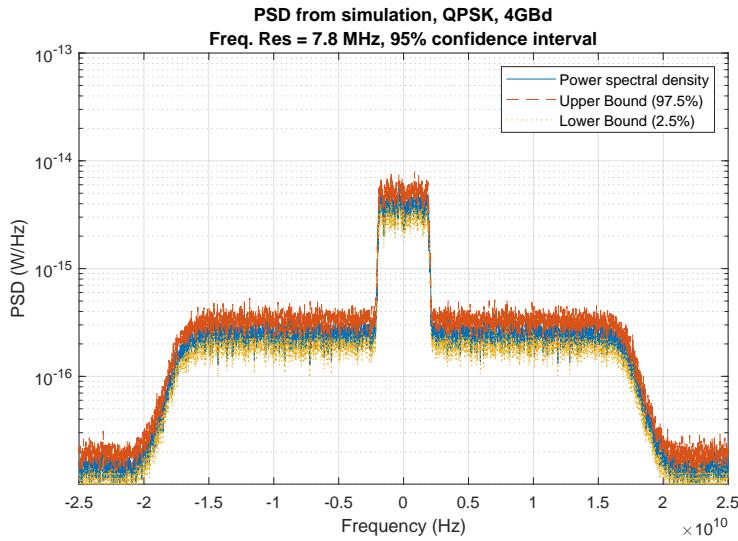


Figure 7.37: Estimated power spectral density for a QPSK signal.

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 [1]:

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

with

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

and

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

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

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

with

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

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

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

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

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

### Confidence intervals

The periodogram is Chi-Square distributed [1]. Therefore, the confidence interval must be calculated accordingly. The confidence interval for a an estimated Power Spectral Density value  $\hat{S}(f)$  is defined by [2]

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

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

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

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

$\chi^2_{\alpha/2}$  can be obtained by finding the values at which the chi-squared cumulative distribution function, for  $\nu$  degrees of freedom, equals the desired probabilities. This function is calculated as [3]

$$\text{CDF}(\chi^2) = \frac{\gamma(\nu/2, \chi^2/2)}{\Gamma(\nu/2)} \quad (7.50)$$

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 7.50 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 7.55 and 7.55.

### 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 [1].

## References

- [1] Michel C. Jeruchim, Philip Balaban, and K. Sam Shanmugan. "Simulation of communication systems: modeling, methodology and techniques". In: Springer Science & Business Media, 2006. Chap. 10, pp. 645–654.
- [2] "Power Spectrum Estimation". In: *National Semiconductor, Application Note AN-255* (Nov. 1980).
- [3] Eric W. Weisstein. *Chi-Squared Distribution. From MathWorld—A Wolfram Web Resource*. Visited on 16/06/18. URL: <http://mathworld.wolfram.com/Chi-SquaredDistribution.html>.

## 7.56 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 7.27: 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

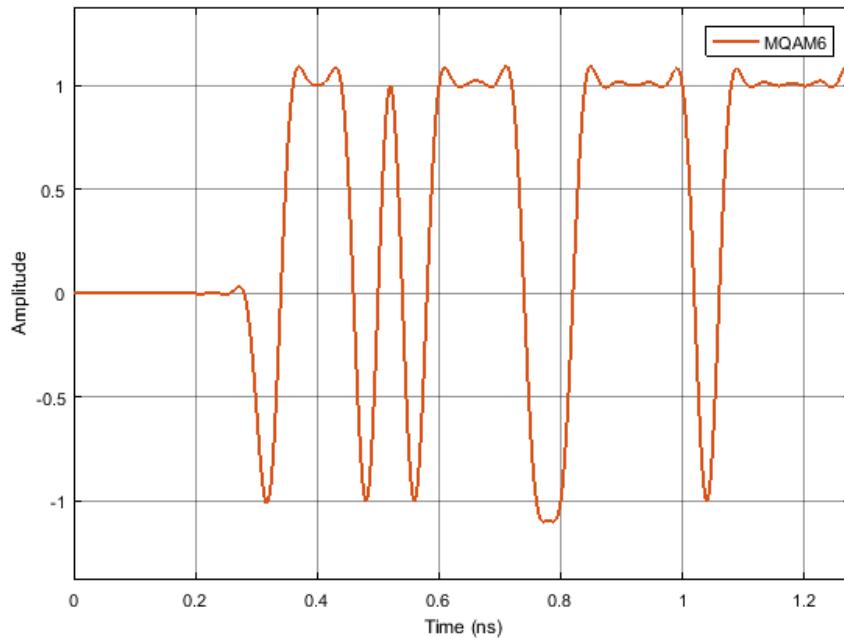


Figure 7.38: Example of a signal generated by this block for the initial binary signal 0100...

## Sugestions for future improvement

Include other types of filters.

## 7.57 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 7.28: 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 nbites) resolution = nbites;

    double getResolution() return resolution;

    void setMinValue(double maxvalue) maxValue = maxvalue;

    double getMinValue() return maxValue;

    void setMaxValue(double maxvalue) maxValue = maxvalue;

    double getMaxValue() return maxValue;

```

### Functional description

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

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

**Input Signals**

**Number:** 1

**Output Signals**

**Number:** 1

**Type:** Electrical complex signal

**Examples**

**Sugestions for future improvement**

## 7.58 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 sampling rate, 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 7.29: Resample input parameters

### Methods

```
Resample() ; Resample(vector<Signal *> &InputSig, vector<Signal *> &OutputSig)
:Block(InputSig, OutputSig){};
void initialize(void); bool runBlock(void);
void setSamplingPeriod(double sPeriod)    samplingPeriod = sPeriod;      void
setSymbolPeriod(double sPeriod) symbolPeriod = sPeriod;
void setOutRateFactor(double OUTsRate)    rFactor = OUTsRate;      double
getOutRateFactor() return rFactor;
```

### Functional description

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

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

**Input Signals**

**Number:** 1

**Output Signals**

**Number:** 1

**Type:** Electrical complex signal

**Examples**

**Sugestions for future improvement**

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

## Theoretical Description

This block can use 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 [1, 2, 3];
- **m2m4** - Moments method [4];
- **ren** - Modified moments method [5];

### 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 though these informations are not available, at the cost of being less efficient [1, 2, 3].

The power spectrum is estimated through Welch's periodogram over a predefined interval size [6]. 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 [6]:

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

By default, the noise is assumed to be AWGN with a given constant spectral density. Therefore, by estimating the noise spectral density, an estimation can be made of the noise's full power. The noise power spectral density is therefore estimated from the 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.

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

This value is saved and the process is repeated for every sequence of samples. The confidence interval is calculated based on the all the obtained SNR values [7]. The SNR value and confidence interval are then saved to a text file.

### Moments method

If the sampling time is known, or it the signal has already been sampled, it is possible to resort to higher order statistics to get an SNR estimation. One popular method, due to its simplicity and efficiency, uses the second and fourth order moments of the signal to estimate the SNR [4].

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.

$$M_2 = E\{|S_k|^2\} = E\{S_{I_k}^2 + S_{Q_k}^2\} = P_S + P_N \quad (7.53)$$

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

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

$$P_S = \sqrt{2M_2^2 - M_4} \quad (7.55)$$

$$P_N = M_2 - \sqrt{2M_2^2 - M_4} \quad (7.56)$$

and therefore

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

### Modified Moments method

This method works like the previous one, but tries to decrease bias and mean square error by using a different fourth moment of the received signal [5].

$$M'_4 = E\{(S_{I_k}^2 - S_{Q_k}^2)^2\} = 2P_S P_N + P_N^2 \quad (7.58)$$

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

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

$$P'_N = M_2 \sqrt{M_2^2 - M_4'} \quad (7.60)$$

$$\text{SNR} = \frac{\sqrt{M_2^2 - M_4'}}{M_2 - \sqrt{M_2^2 - M_4'}} \quad (7.61)$$

### Known Issues

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

## References

- [1] Fred Harris and Chris Dick. "SNR estimation techniques for low SNR signals". In: *Wireless Personal Multimedia Communications (WPMC), 2012 15th International Symposium on*. IEEE. 2012, pp. 276–280.
- [2] Haifeng Xiao et al. "An investigation of non-data-aided snr estimation techniques for analog modulation signals". In: *Sarnoff Symposium, 2010 IEEE*. IEEE. 2010, pp. 1–5.
- [3] Hamide Kashefi, Sahar Karimkeshteh, and H Khoshbin Ghomash. "Spectrum-based SNR estimator for analog and digital bandpass signals". In: *Telecommunications (IST), 2012 Sixth International Symposium on*. IEEE. 2012, pp. 373–377.
- [4] Rolf Matzner. "An SNR estimation algorithm for complex baseband signals using higher order statistics". In: *Facta Universitatis (Nis) 6.1* (1993), pp. 41–52.
- [5] Guangliang Ren, Yilin Chang, and Hui Zhang. "A new SNR's estimator for QPSK modulations in an AWGN channel". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 52.6 (2005), pp. 336–338.
- [6] John G.. Proakis and Dimitris G.. Manolakis. *Digital signal processing: principles, algorithms, and applications*. Pearson Prentice Hall, 2007.
- [7] William H Tranter et al. *Principles of communication systems simulation with wireless applications*. Prentice Hall New Jersey, 2004.

## 7.60 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 7.30: 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

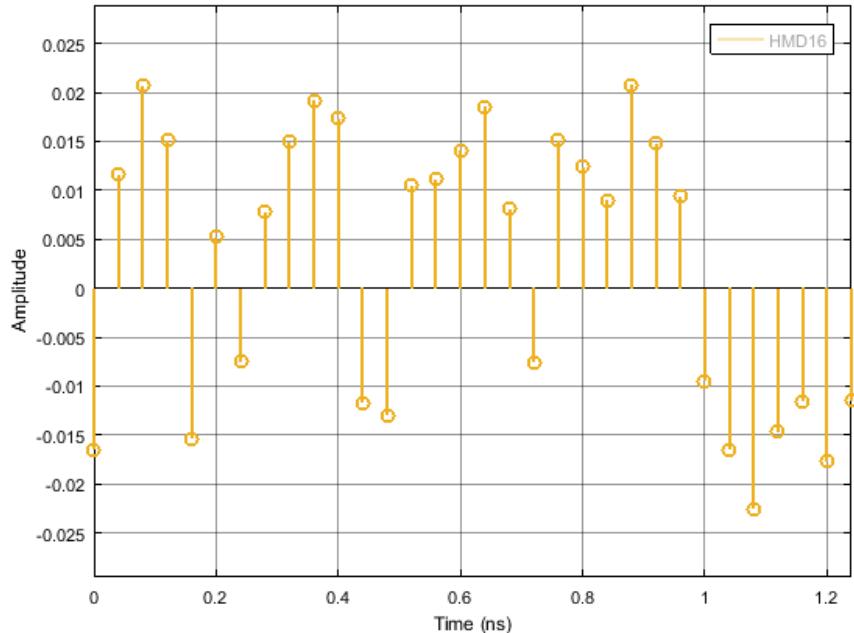


Figure 7.39: Example of the output signal of the sampler

## Sugestions for future improvement

## 7.61 SNR of the Photoelectron Generator

|                    |   |                                   |
|--------------------|---|-----------------------------------|
| <b>Header File</b> | : | srn_photoelectron_generator_*.h   |
| <b>Source File</b> | : | snr_photoelectron_generator_*.cpp |
| <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.

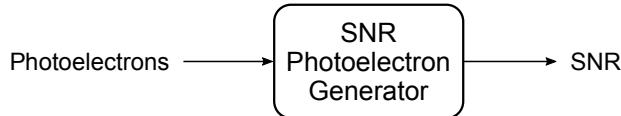


Figure 7.40: Schematic representation of the SNR of the Photoelectron Generator code block

### 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 [1]

$$\text{SNR} = \frac{\bar{n}^2}{\sigma_n^2} \quad (7.62)$$

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,  $s_k^2$  [2]

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

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

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

in which

$$j_{min}(k) = \lceil t_k / \Delta t \rceil \quad (7.65)$$

$$j_{max}(k) = \lceil t_{k+1} / \Delta t \rceil - 1 \quad (7.66)$$

$$N_k = j_{max}(k) - j_{min}(k) + 1 \quad (7.67)$$

$$t_k = kT \quad (7.68)$$

where  $\lceil x \rceil$  is the ceiling function.

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

$$S_1(k) = N_k \langle y \rangle_k \quad (7.69)$$

$$S_2(k) = N_k \langle y^2 \rangle_k \quad (7.70)$$

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

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

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

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

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

$$\text{SNR} = \eta \bar{n} \quad (7.73)$$

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  $t_{\text{Window}} = T$ . For each time window, the general process consists in accumulating the input sample values and the square of the input sample values, and calculating the SNR of the time window based on these two variables.

To process this accumulation, the block uses two state variables, `aux_sum1` and `aux_sum2`, which hold the accumulation of the sample values and accumulation of the square of sample values, respectively.

The block starts by calculating the number of samples it has to process for the current time window, using equations 7.66, 7.67 and 7.68. If the duration of  $t_{\text{Window}}$  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 7.72.

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      |
|------------|---------------|------------------|
| windowTime | 0             | SNR time window. |

## Methods

`SnrPhotoelectronGenerator()`

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

`void initialize(void)`

`bool runBlock(void)`

`void setTimeWindow(t_real timeWindow)`

## Input Signals

**Number:** 1

**Type:** Electrical (TimeDiscreteAmplitudeContinuousReal)

## Output Signals

**Number:** 1

**Type:** Electrical (TimeDiscreteAmplitudeContinuousReal)

## Examples

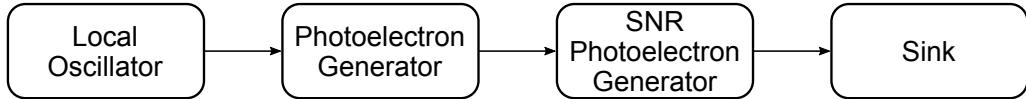


Figure 7.41: Simulation setup

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  $\Delta t = 10^{-10}s$ .

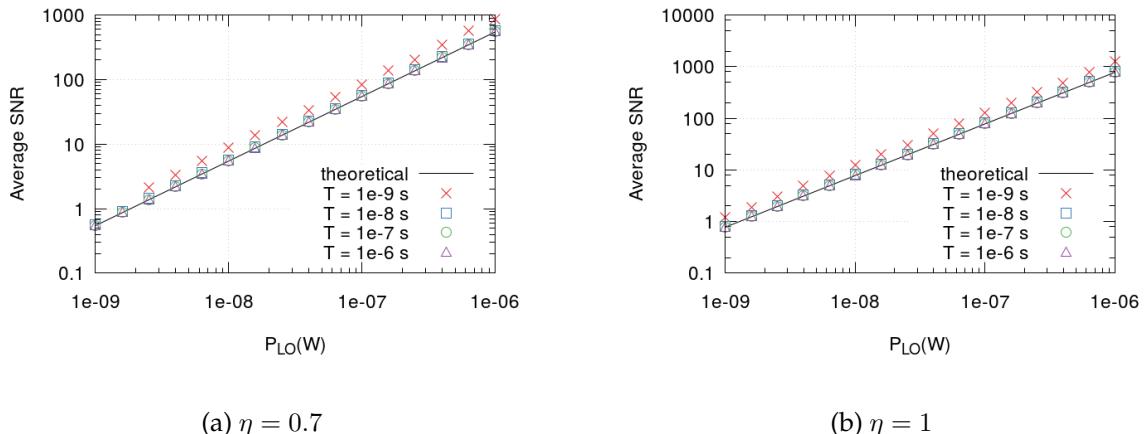


Figure 7.42: Theoretical and simulated results of the average SNR, for two photodiode efficiencies.

The plots in 7.42 show the comparison between the theoretical result 7.73 and the simulation results. We see that for low values of  $T$ , the average SNR shows a systematic deviation 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 7.42a). This is because in those conditions, the probability of having a time windows with no photoelectrons, creating an 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 its 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

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

## Block problems

### Future work

The block could also output a confidence interval for the calculated SNR. Given that the output of the Photoelectron Generator follows a Poissonian 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 [3], could be used as a reference to implement such feature.

## References

- [1] Bahaa E. A. Saleh and Malvin Carl Teich. *Fundamentals of photonics*. Wiley series in pure and applied optics. New York (NY): John Wiley & Sons, 1991.
- [2] S. W. Smith et al. *The scientist and engineer's guide to digital signal processing*. California Technical Pub. San Diego, 1997.
- [3] G. Florence and K. B. Golam. "Confidence intervals for signal to noise ratio of a Poisson distribution". In: *American Journal of Biostatistics* 2.2 (2011), p. 44.

## 7.62 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 7.31: 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 setNumberOfBytesToFile(long int newValue)

long int getNumberOfBytesToFile()

void setNumberOfSamples(long int nOfSamples)

long int getNumberOfSamples const()

void setDisplayNumberOfSamples(bool opt)

bool getDisplayNumberOfSamples const()

```

## Functional Description

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

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

## Theoretical Description

This block can use 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 [1, 2, 3];
- **m2m4** - Moments method [4];
- **ren** - Modified moments method [5];

### 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 [1, 2, 3].

The power spectrum is estimated through Welch's periodogram over a predefined interval size [6]. 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 [6]:

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

By default, the noise is assumed to be AWGN with a given constant spectral density. Therefore, by estimating the noise spectral density, an estimation can be made of the noise's full power. The noise power spectral density is therefore estimated from the 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.

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

This value is saved and the process is repeated for every sequence of samples. The confidence interval is calculated based on the all the obtained SNR values [7]. The SNR value and confidence interval are then saved to a text file.

### Moments method

If the sampling time is known, or it the signal has already been sampled, it is possible to resort to higher order statistics to get an SNR estimation. One popular method, due to its simplicity and efficiency, uses the second and fourth order moments of the signal to estimate the SNR [4].

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.

$$M_2 = E\{|S_k|^2\} = E\{S_{I_k}^2 + S_{Q_k}^2\} = P_S + P_N \quad (7.77)$$

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

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

$$P_S = \sqrt{2M_2^2 - M_4} \quad (7.79)$$

$$P_N = M_2 - \sqrt{2M_2^2 - M_4} \quad (7.80)$$

and therefore

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

### Modified Moments method

This method works like the previous one, but tries to decrease bias and mean square error by using a different fourth moment of the received signal [5].

$$M'_4 = E\{(S_{I_k}^2 - S_{Q_k}^2)^2\} = 2P_S P_N + P_N^2 \quad (7.82)$$

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

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

$$P'_N = M_2 \sqrt{M_2^2 - M_4'} \quad (7.84)$$

$$\text{SNR} = \frac{\sqrt{M_2^2 - M_4'}}{M_2 - \sqrt{M_2^2 - M_4'}} \quad (7.85)$$

### Known Issues

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

## References

- [1] Fred Harris and Chris Dick. "SNR estimation techniques for low SNR signals". In: *Wireless Personal Multimedia Communications (WPMC), 2012 15th International Symposium on*. IEEE. 2012, pp. 276–280.
- [2] Haifeng Xiao et al. "An investigation of non-data-aided snr estimation techniques for analog modulation signals". In: *Sarnoff Symposium, 2010 IEEE*. IEEE. 2010, pp. 1–5.
- [3] Hamide Kashefi, Sahar Karimkeshteh, and H Khoshbin Ghomash. "Spectrum-based SNR estimator for analog and digital bandpass signals". In: *Telecommunications (IST), 2012 Sixth International Symposium on*. IEEE. 2012, pp. 373–377.
- [4] Rolf Matzner. "An SNR estimation algorithm for complex baseband signals using higher order statistics". In: *Facta Universitatis (Nis) 6.1* (1993), pp. 41–52.
- [5] Guangliang Ren, Yilin Chang, and Hui Zhang. "A new SNR's estimator for QPSK modulations in an AWGN channel". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 52.6 (2005), pp. 336–338.
- [6] John G.. Proakis and Dimitris G.. Manolakis. *Digital signal processing: principles, algorithms, and applications*. Pearson Prentice Hall, 2007.
- [7] William H Tranter et al. *Principles of communication systems simulation with wireless applications*. Prentice Hall New Jersey, 2004.

## 7.64 Single Photon Receiver

This block of code simulates the reception of two time continuous signals which are the outputs of single photon detectors and decode them in measurements results. A simplified schematic representation of this block is shown in figure 7.43.

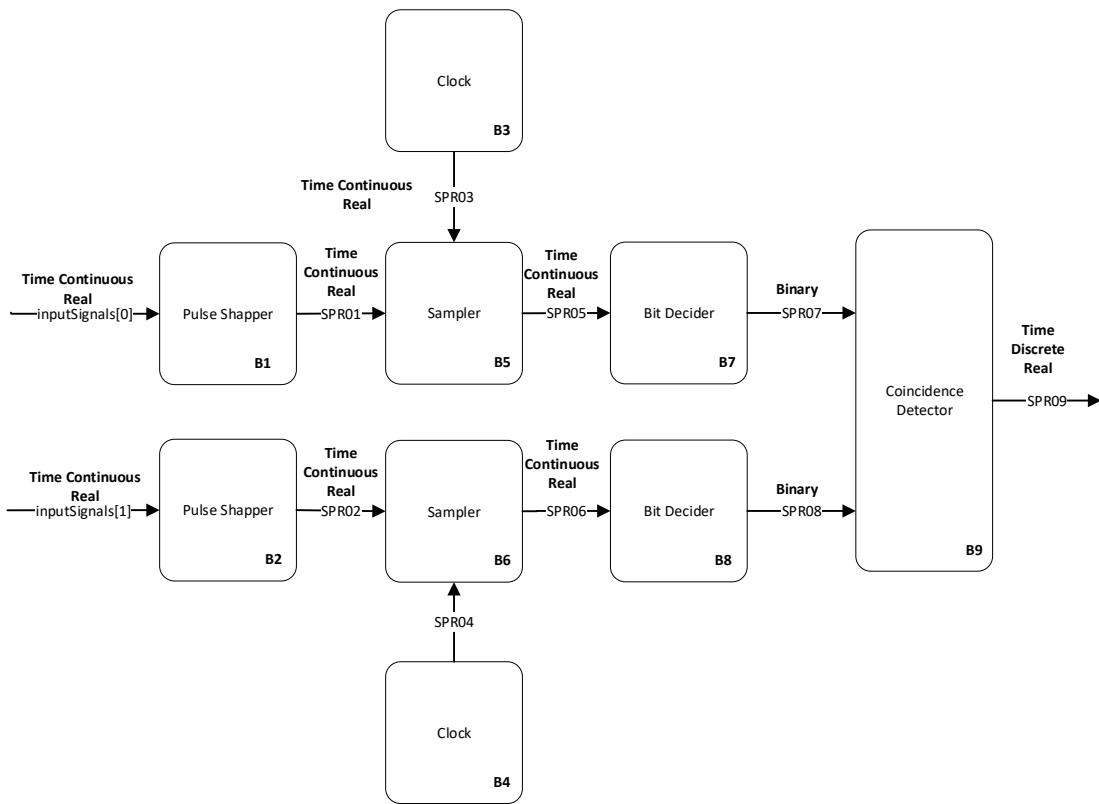


Figure 7.43: Basic configuration of the SPR receiver

### Functional description

This block accepts two time continuous input signals and outputs one time discrete signal that corresponds to the single photon detection measurements demodulation of the input signal. It is a complex block (as it can be seen from figure 7.43 of code made up of several simpler blocks whose description can be found in the *lib* repository).

It can also be seen from figure 7.43 that there are two extra internal input signals generated by the *Clock* in order to keep all blocks synchronized. This block is used to provide the sampling frequency to the *Sampler* blocks.

### Input parameters

This block has some input parameters that can be manipulated by the user in order to change the basic configuration of the receiver. Each parameter has associated a function that allows for its change. In the following table (table 7.33) the input parameters and corresponding functions are summarized.

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

Table 7.32: List of input parameters of the block SP receiver

**Methods**

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

```
void setPulseShaperFilter(PulseShaperFilter fType)
```

```
void setPulseShaperWidth(double pulseW)
```

```
void setClockBitPeriod(double period)
```

```
void setClockPhase(double phase)
```

```
void setClockSamplingPeriod(double sPeriod)
```

```
void setThreshold(double threshold)
```

**Input Signals**

**Number:** 2

**Type:** Time Continuous Amplitude Continuous Real

**Output Signals**

**Number:** 1

**Type:** Time Discrete Amplitude Discrete Real

**Example**

**Sugestions for future improvement**

## 7.65 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 [1] was implemented.

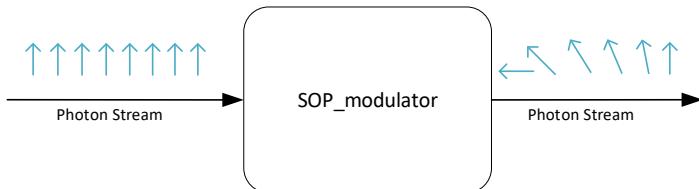


Figure 7.44: Diagram block of SOP block input and outputs.

The block in figure 7.44 was implemented based on a continuous matrix calculation. Lets assume that the input photon stream can be represented by the matrix:

$$S_{in} = \begin{bmatrix} A_x \\ A_y \end{bmatrix}, \quad (7.86)$$

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

$$S_{out} = S_{in}J_k, \quad (7.87)$$

where,

$$J_k = J(\dot{\alpha})J_{k-1}. \quad (7.88)$$

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

$$\begin{aligned} J(\alpha) &= e^{-i\alpha \cdot \vec{\sigma}} \\ &= \mathbf{I}_2 \cos(\theta) - i\mathbf{a} \cdot \vec{\sigma} \sin(\theta), \end{aligned} \quad (7.89)$$

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

$$\sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}; \sigma_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \sigma_3 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}. \quad (7.90)$$

$\mathbf{I}_2$  is the  $2 \times 2$  identity matrix. In addition,  $\alpha = \theta\mathbf{a}$ , having the length  $\theta = \|\alpha\|$  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  $\alpha$  parameters:

$$\dot{\alpha} \sim \mathcal{N}(0, \sigma_p^2 \mathbf{I}_3), \quad (7.91)$$

where  $\sigma_p^2 = 2\pi\Delta p T$ , 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 7.33) 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 7.33: List of input parameters of the block sop modulator.

### Methods

```
SOPModulator(vector <Signal*> &inputSignals, vector <Signal*>
&outputSignals)(constructor)
```

```
void initialize(void)  
bool runBlock(void)  
void setSOPType(SOPType sType)  
void setRotationAngle(double angle)  
void setElevationAngle(double angle)  
void setDeltaP(long double deltaP)  
long double getDeltaP()
```

**Input Signals**

**Number:** 1

**Type:** PhotonStreamXY

**Output Signals**

**Number:** 1

**Type:** PhotonStreamXY

**Example**

**Sugestions for future improvement**

## References

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

## 7.66 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 sourceOrde.

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

Table 7.34: Source Code Efficiency input parameters

### Functional Description

This block estimates the efficiency of the message, by calculating the entropy and the length of the message.

### Input Signals

**Number:** 1

**Type:** Binary

### Output Signals

**Number:** 1

**Type:** Real (TimeContinuousAmplitudeContinuousReal)

## 7.67 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 7.35: 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  $T$ .

$$N = n_0 B = n_0 \frac{2}{T} \quad (7.92)$$

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

### Input Signals

**Number:** 0

### Output Signals

**Number:** 1 or more

**Type:** RealValue, ComplexValue or ComplexValueXY

**Examples**

**Random Mode**

**Suggestions for future improvement**

## 7.68 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 7.36: Ideal Amplifier input parameters

### Methods

IdealAmplifier()

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

void initialize(void);

bool runBlock(void);

void setGain(double ga) gain = ga;

double getGain() return gain;
```

### Functional description

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

**Input Signals**

**Number:** 1

**Type:** Electrical (TimeContinuousAmplitudeContinuousReal)

**Output Signals**

**Number:** 1

**Type:** Electrical (TimeContinuousAmplitudeContinuousReal)

**Examples**

**Sugestions for future improvement**

## 7.69 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 [1, 2]. A representation of the output of the modulation stage of a pilot-assisted LLO CV-QC scheme is presented in Figure 7.45. An unmodulated pilot signal

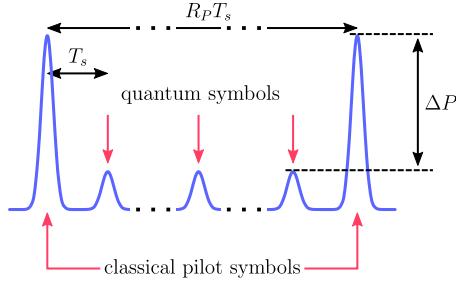


Figure 7.45: Representative time dependence of a pilot assisted phase mismatch compensation signal.

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

$$y_r(t_n) = \begin{cases} x_q(t_n)e^{i[\theta_q(t_n)+\epsilon(t_n)]}, & n \neq mR_P \\ x_p(t_n)e^{i[\epsilon(t_n)]}, & n = mR_P \end{cases}, \quad n, m \in \mathbb{N}. \quad (7.93)$$

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

$$\hat{\epsilon}(t_n) = \frac{\epsilon(t_{(m+1)R_P}) + \epsilon(t_{mR_P})}{2}, \quad mR_P < n < (m+1)R_P, \quad (7.94)$$

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

$$x_q(t_n)e^{i[\theta_q(t_n)+\epsilon(t_n)-\hat{\epsilon}(t_n)]}. \quad (7.95)$$

## References

- [1] Daniel BS Soh et al. "Self-referenced continuous-variable quantum key distribution protocol". In: *Physical Review X* 5.4 (2015), p. 041010.
- [2] Bing Qi et al. "Generating the local oscillator "locally" in continuous-variable quantum key distribution based on coherent detection". In: *Physical Review X* 5.4 (2015), p. 041009.

## 7.70 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

$$y(t) = |y(t)|e^{i[\Delta\omega t + \theta(t) + \epsilon(t)]}, \quad (7.96)$$

where  $\Delta\omega$  is the frequency mismatch,  $\theta(t_n)$  is the phase encoded in the signal and  $\epsilon(t_n)$  is the phase noise contribution. Frequency mismatch compensation is accomplished by first estimating the value of  $\Delta\omega$  and then removing via

$$y(t)' = y(t) * e^{-i\Delta\hat{\omega}}. \quad (7.97)$$

Three methods for estimating  $\Delta\omega$  are presented here.

#### Pilot aided frequency mismatch compensation

This method uses a pilot signal similar to the ones employed in pilot assisted phase mismatch compensation techniques. In a pilot aided technique a reference signal (the pilot), composed of pre-agreed on symbols, is inserted, time multiplexed, with the data payload at a pre-agreed on rate. A visual representation of the output of the modulation stage of a pilot-assisted scheme is presented in Figure 7.46. At the receiver stage, after coherent detection,

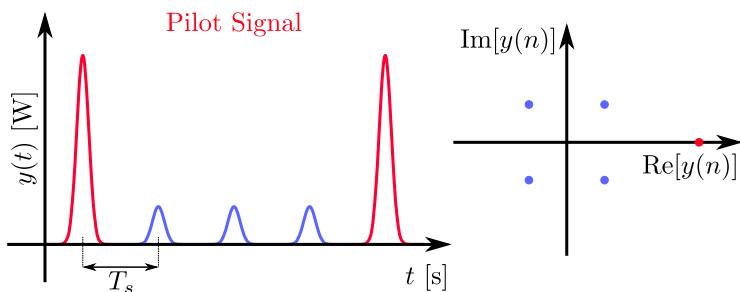


Figure 7.46: Time dependence (left) and constellation (right) at the output of the modulation stage of a pilot-assisted scheme. Pilot pulses/points identified by color. Pilot rate in the time dependence image is meant only as illustrative, actual pilot rate may be higher or lower.

the signal is described by

$$y(t_n) = \begin{cases} x_s(t_n)e^{i[\Delta\omega t_n + \theta_q(t_n) + \epsilon(t_n)]}, & n \neq mR_P \\ x_p(t_n)e^{i[\Delta\omega t_n + \epsilon(t_n)]}, & n = mR_P \end{cases}, \quad (7.98)$$

where  $x_s(t)/x_p(t)$  represents the signal/pilot amplitude,  $R_P$  is the pilot rate,  $\Delta\omega$  is the offset frequency,  $\epsilon(t)$  is the instantaneous phase noise contribution and  $\theta(t)$  is the phase

modulation at instant  $t$ . The first step in this technique is to obtain the auxiliary signal  $x_f(m)$ , which is defined as

$$\begin{aligned} x_f(m) &= y_r^*(t_{mR_p})y_r(t_{(m+1)R_p}) \\ &= x_p(t_{mR_p})x_p(t_{(m+1)R_p})e^{i(\Delta\omega R_p T_s + \Delta\epsilon(m))}, \end{aligned} \quad (7.99)$$

where

$$\Delta\epsilon(m) = \epsilon(t_{mR_p}) - \epsilon(t_{(m+1)R_p}). \quad (7.100)$$

The frequency estimation is accomplished by taking the expected value of the complex argument of (7.99) and dividing it by  $R_p T_s$ . This returns

$$\begin{aligned} \hat{\Delta\omega} &= E\left[\frac{1}{R_p T_s} \arg(x_f(m))\right] \\ &= E\left[\Delta\omega + 2k\pi(R_p T_s)^{-1} + \frac{\Delta\epsilon(m)}{R_p T_s}\right] \\ &= \Delta\omega + 2k\pi(R_p T_s)^{-1}. \end{aligned} \quad (7.101)$$

### **Blind estimation frequency mismatch compensation**

In this method the frequency is scanned over a predetermined range, symbol decisions are made and the minimum square error used as the frequency-selection criteria. Scanning is first done with a large step to find a rough value for  $\Delta\omega$ , this is then repeated with a smaller step to find a more exact estimate [1].

### **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 [2], which exhibits a peak at the frequency  $m\Delta\Omega$ , this value is used as the estimate.

## References

- [1] Xiang Zhou et al. "64-Tb/s, 8 b/s/Hz, PDM-36QAM transmission over 320 km using both pre-and post-transmission digital signal processing". In: *Journal of Lightwave Technology* 29.4 (2011), pp. 571–577.
- [2] Mehrez Selmi, Yves Jaouën, and Philippe Ciblat. "Accurate digital frequency offset estimator for coherent PolMux QAM transmission systems". In: *Optical Communication, 2009. ECOC'09. 35th European Conference on*. IEEE. 2009, pp. 1–2.

## 7.71 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

## 7.72 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  $\vec{ref}_v$ . This process is presented visually in Figure 7.47 and is described by

$$\widehat{EVM} = 100 \sqrt{\frac{|\vec{e}_v|}{|\vec{ref}_v|}} = 100 \sqrt{\frac{|\vec{m}_v - \vec{ref}_v|}{|\vec{ref}_v|}}. \quad (7.102)$$

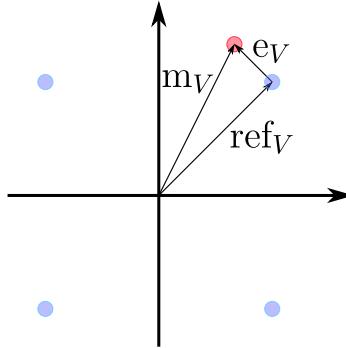


Figure 7.47: Visual representation of the EVM method in a QPSK constellation. The ideal constellation points are presented in blue, while an example for the actual measured symbol is presented in red.

## 7.73 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    | CommaSepreatedValues,<br>ConcatenatedValues             | CommaSepreatedValues |
| dataType       | signal_value_type | BinaryValue, RealValue,<br>ComplexValue, ComplexValueXY | BinaryValue          |

Table 7.37: Load ascii input parameters

### Methods

LoadAscii()

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

void initialize(void);

bool runBlock(void);

void setSamplingPeriod(double sPeriod);

void setSymbolPeriod(double sPeriod);

void setDataType(signal_value_type dType);

void setAsciiFileName(string aFileName);
```

### Functional description

This block loads signals from a .txt file and generates a signal with a specified data type given by the input parameter *dataType* and with a specified delimiter type given by the input parameter *delimiterType*.

**Input Signals**

**Number:** 0

**Output Signals**

**Number:** 1

**Type:** Optical signal, Electrical signal, Complex signal, Binary signal (user defined)

## 7.74 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 7.38: 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

## **Chapter 8**

---

### **Mathlab Tools**

## 8.1 Generation of AWG Compatible Signals

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

This section shows how to convert a simulation signal into an AWG compatible waveform file through the use of a matlab function called sgnToWfm. This allows the application of simulated signals into real world systems.

### 8.1.1 sgnToWfm.m

#### Structure of a function

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

#### Inputs

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

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

**fname\_wfm:** Name that will be given to the waveform file.

#### Outputs

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

**data:** A vector with the signal data.

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

**samplingPeriod:** Sampling period of the signal.

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

**numberOfSymbols:** Number of symbols retrieved from the signal.

**samplingRate:** Sampling rate of the signal.

## Functional Description

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

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

Using one argument:

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

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

Using two arguments:

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

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

Using three arguments:

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

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

### **8.1.2 sgnToWfm\_20171121.m**

#### **Structure of a function**

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

#### **Inputs**

Same as discussed above in the file sgnToWfm.m.

## Outputs

The output of the function sgnToWfm\_20171121.m contains eight different parameters. Among those eight different parameters, six output parameters are the same as discussed above in the function sgnToWfm.m and remaining two parameters are the following:

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

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

«««< HEAD

## Functional Description

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

## Outputs

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

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

### Decimation factor calculation

The flowchart for calculating the decimation factor is as follows:

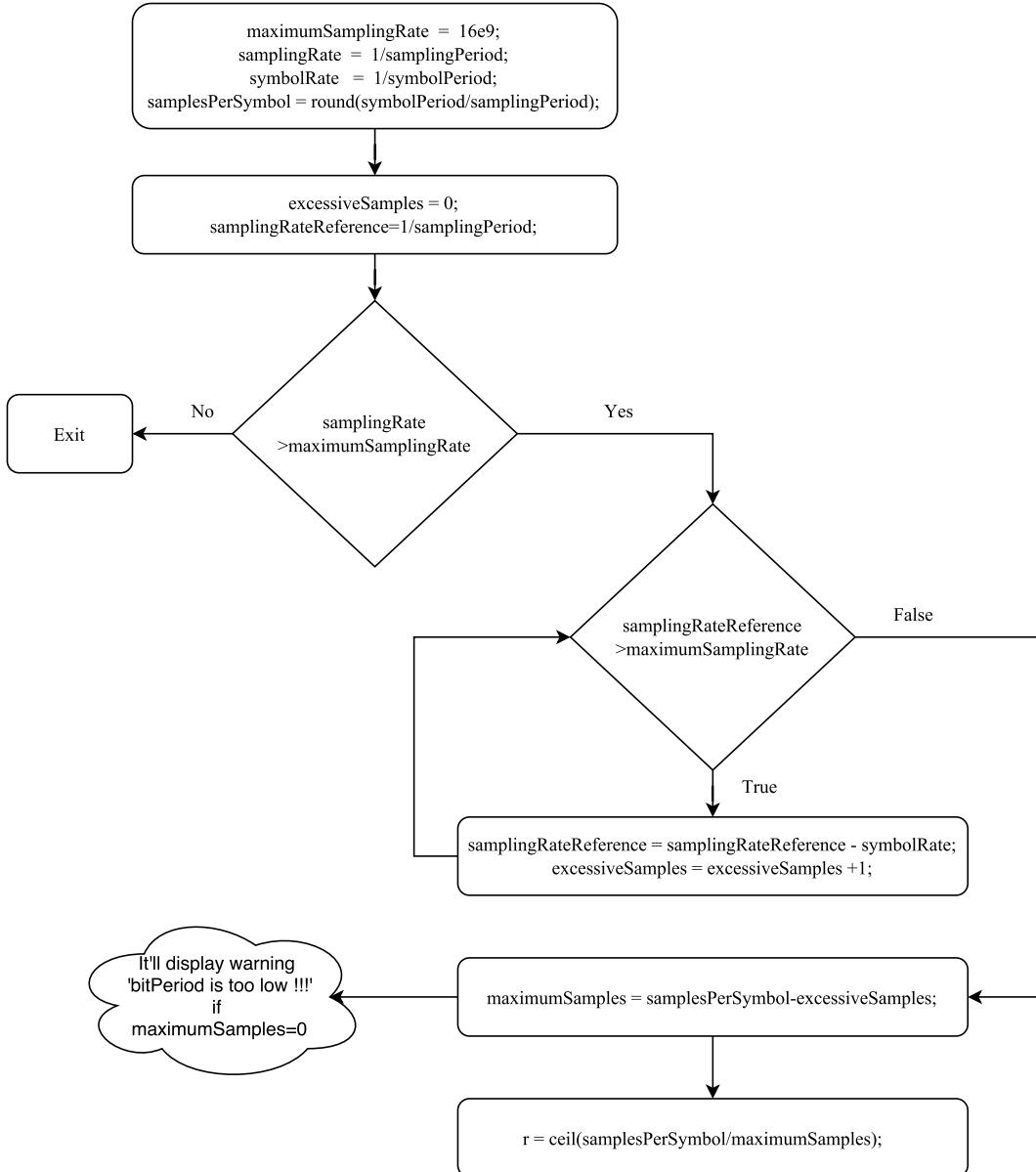


Figure 8.1: Flowchart to calculate decimation factor

»»»» Develop.Romil

#### 8.1.3 Loading a signal to the Tektronix AWG70002A

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

**Sampling rate up to 16 GS/s:** This is the most important characteristic because it determines the maximum sampling rate that your signal can have. It must not be over 16 GS/s or else the AWG will not be able to recreate it appropriately.

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

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

**1. Using the function sgnToWfm:** Start up Matlab and change your current folder to mtools and add the signals folder that you want to convert to the Matlab search path. Use the function accordingly, putting as the input parameter the signal file name you want to convert.

**2. AWG sampling rate:** After calling the function there should be waveform file in the mtools folder, as well as a variable called samplingRate in the Matlab workspace. Make sure this is equal or bellow the maximum sampling frequency of the AWG (16 GS/s), or else the waveform can not be equal to the original signal. If it is higher you have to adjust the parameters in the simulation in order to decrease the sampling frequency of the signal(i.e. decreasing the bit period or reducing the samples per symbol).

**3. Loading the waveform file to the AWG:** Copy the waveform file to your pen drive and connect it to the AWG. With the software of the awg open, go to browse for waveform on the channel you want to use, and select the waveform file you created (Figure 7.1).

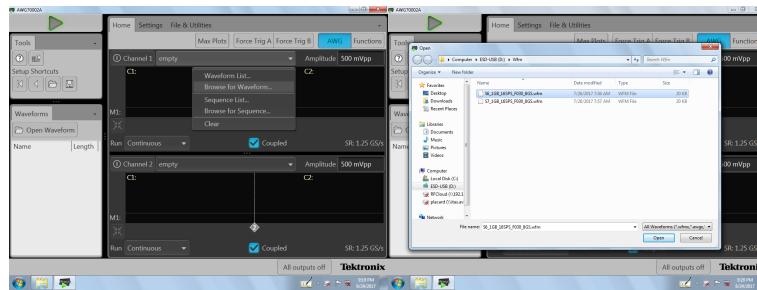


Figure 8.2: Selecting your waveform in the AWG

Now you should have the waveform displayed on the screen. Although it has the same shape, the waveform might not match the signal timing wise due to an incorrect sampling rate configured in the AWG. In this example (Figure 7.2), the original signal has a sample rate of 8 GS/s and the AWG is configured to 1.25 GS/s. Therefore it must be changed to the correct value. To do this go to the settings tab, clock settings, and change the sampling rate to be equal to the one of the original signal, 8 GS/s (Figure 7.3). Compare the waveform in the AWG with the original signal, they should be identical (Figure 7.4).

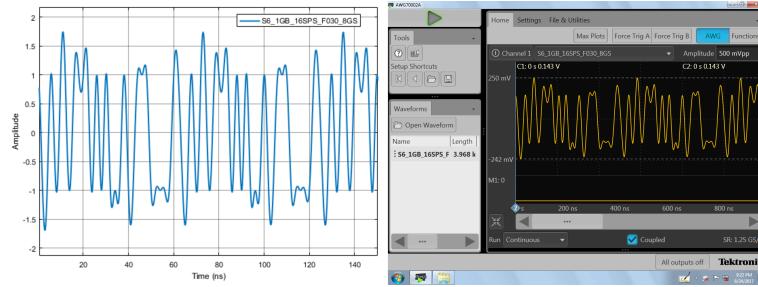


Figure 8.3: Comparison between the waveform in the AWG and the original signal before configuring the sampling rate

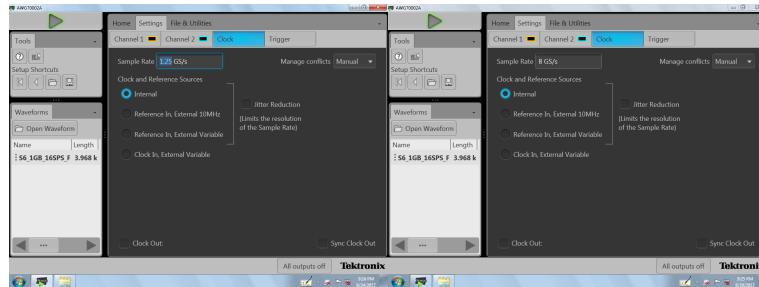


Figure 8.4: Configuring the right sampling rate

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

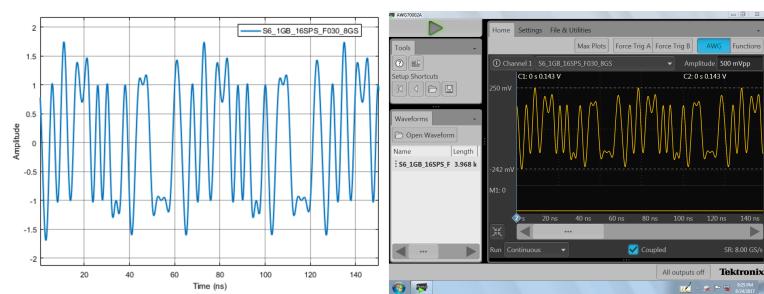


Figure 8.5: Comparison between the waveform in the AWG and the original signal after configuring the sampling rate

## 8.2 Polarization Analysis Signals

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

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

### 8.2.1 jonesToStokes.m

#### Structure of a function

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

#### Inputs

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

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

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

**NumberOfSamples:** Number of Samples to plot in Poincare sphere.

#### Outputs

Two graphics are plotted from the photon stream input signal:

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

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

## Functional Description

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

|                      |   |                |
|----------------------|---|----------------|
| <b>Students Name</b> | : | Mariana Ramos  |
| <b>Version</b>       | : | ACF_20180614.m |

### 8.2.2 ACF.m

#### Structure of a function

[] = ACF(fname,deltaP,N)

#### Inputs

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

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

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

#### Outputs

This matlab function has two outputs:

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

**ACF plot:** A plot with numerical ACF calculated from simulation data and with the theoretical ACF calculated based on the value of  $\Delta p$  inserted as an input of the function.

## Functional Description

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

|                      |   |                             |
|----------------------|---|-----------------------------|
| <b>Students Name</b> | : | Mariana Ramos               |
| <b>Version</b>       | : | plotPhotonStream_20180102.m |

### 8.2.3 plotPhotonStream\_20180102.m

#### Structure of a function

[mag\_x, mag\_y, phase\_dif, h] = plotPhotonStream\_20180102(fname, opt, h)

#### Inputs

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

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

**h:** Number of the figure to plot.

#### Outputs

This matlab function has four outputs:

**mag\_x:** Magnitude of the complex number X.

**mag\_y:** Magnitude of the complex number Y.

**phase\_dif:** Difference phase between the two complex numbers X and Y.

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

#### Functional Description

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

|                      |   |               |
|----------------------|---|---------------|
| <b>Students Name</b> | : | Mariana Ramos |
| <b>Version</b>       | : | plot_sphere.m |

#### 8.2.4 `plot_sphere.m`

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

## **Chapter 9**

---

## **Algorithms**

## 9.1 Fast Fourier Transform

|                    |   |                        |
|--------------------|---|------------------------|
| <b>Header File</b> | : | fft_*.h                |
| <b>Source File</b> | : | fft_*.cpp              |
| <b>Version</b>     | : | 20180201 (Romil Patel) |

### 9.1.0.1 Algorithm

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

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

Similarly, for IFFT,

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

From equations 9.1 and 9.2, we can write only one script for the implementations of the direct and inverse Discrete Fourier Transfer and manipulate its functionality as a FFT or IFFT by applying an appropriate input arguments. The generalized form for the algorithm can be given as,

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

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

### 9.1.0.2 Function description

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

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

or

$$y = fft(x)$$

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

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

or

$$x = ifft(y)$$

### 9.1.0.3 Flowchart

The figure 9.1 displays top level architecture of the FFT algorithm. If the length of the input signal is  $2^N$ , it'll execute Radix-2 algorithm otherwise it'll execute Bluestein algorithm [1]. The computational complexity of Radix-2 and Bluestein algorithm is  $O(N \log_2 N)$ , however, the computation of Bluestein algorithm involves the circular convolution which increases the number of computations. Therefore, to reduce the computational time it is advisable to work with the vectors of length  $2^N$  [2].

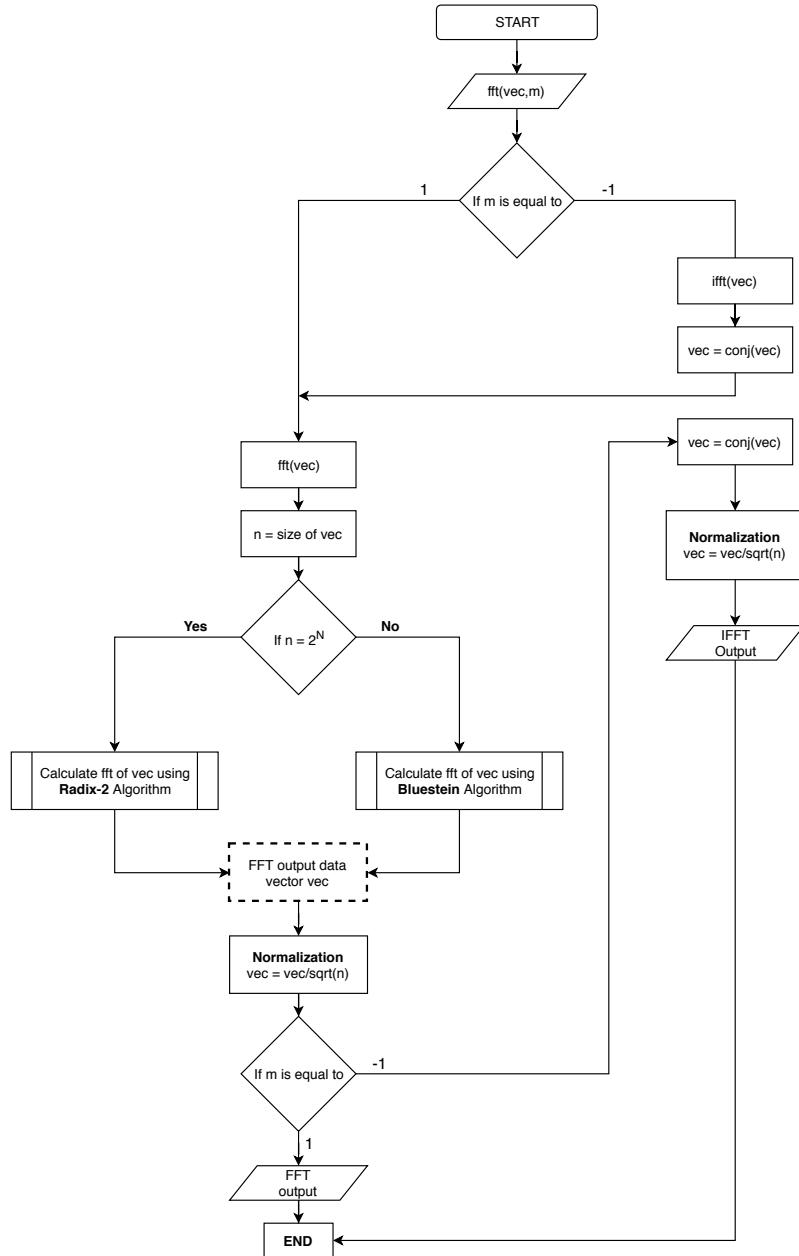


Figure 9.1: Top level architecture of FFT algorithm

#### 9.1.0.4 Radix-2 algorithm

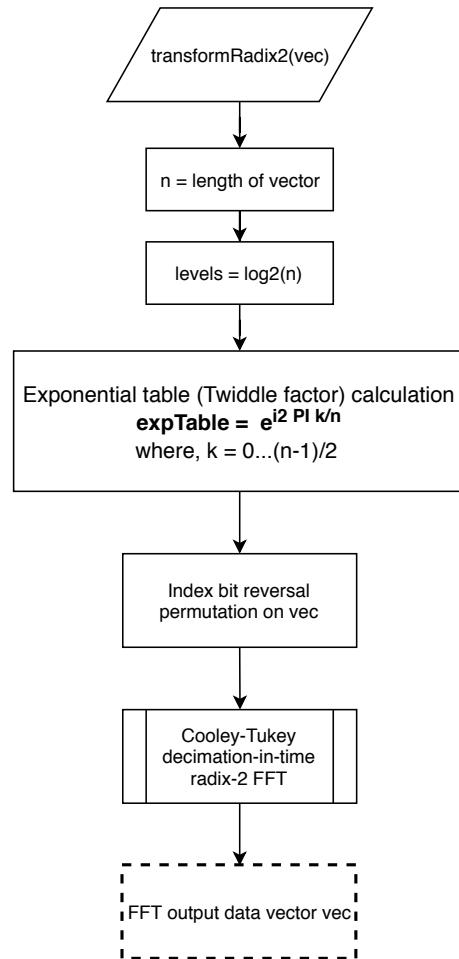


Figure 9.2: Radix-2 algorithm

### 9.1.0.5 Cooley-Tukey algorithm

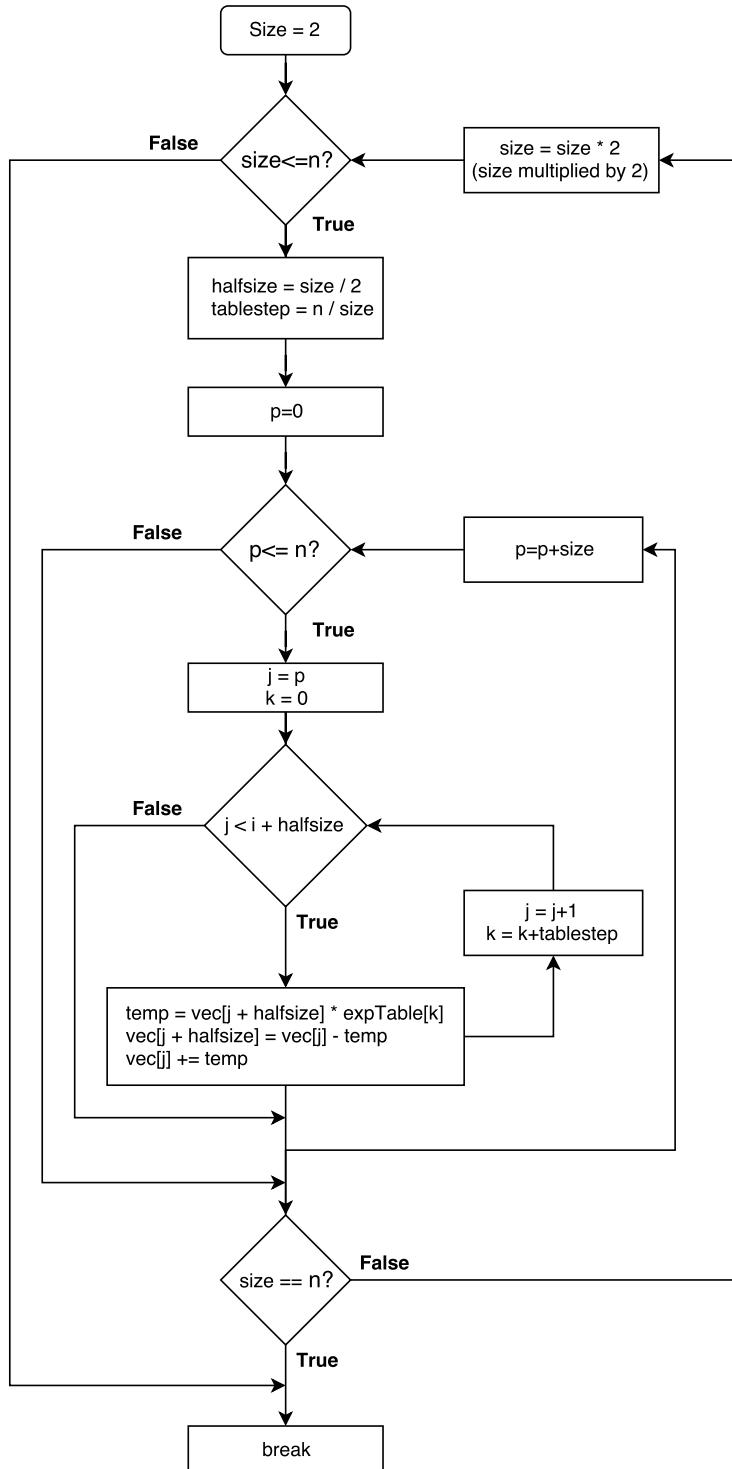


Figure 9.3: Cooley-Tukey algorithm

### 9.1.0.6 Bluestein algorithm

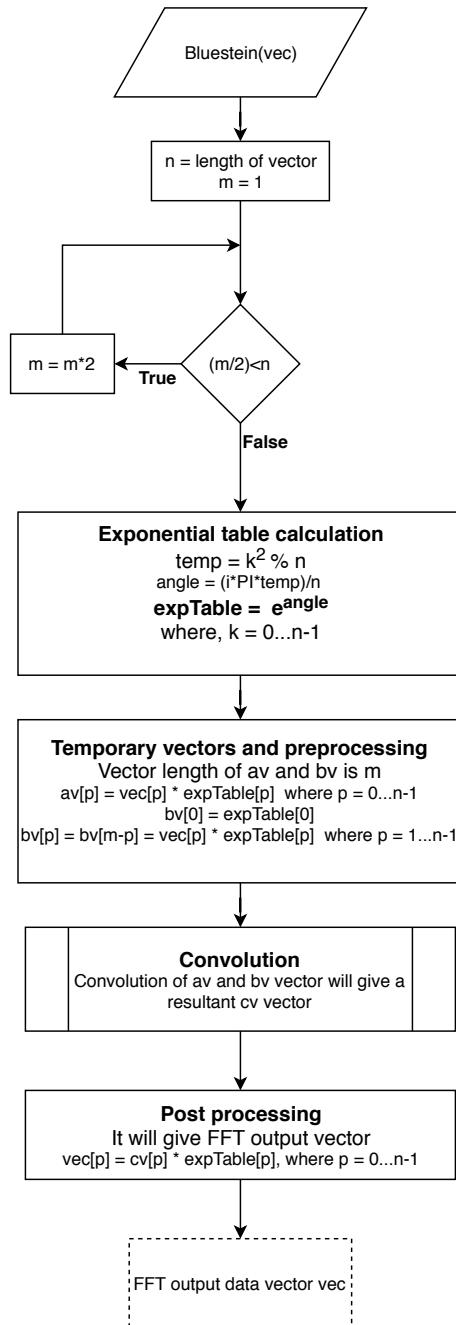


Figure 9.4: Bluestein algorithm

### 9.1.0.7 Convolution algorithm

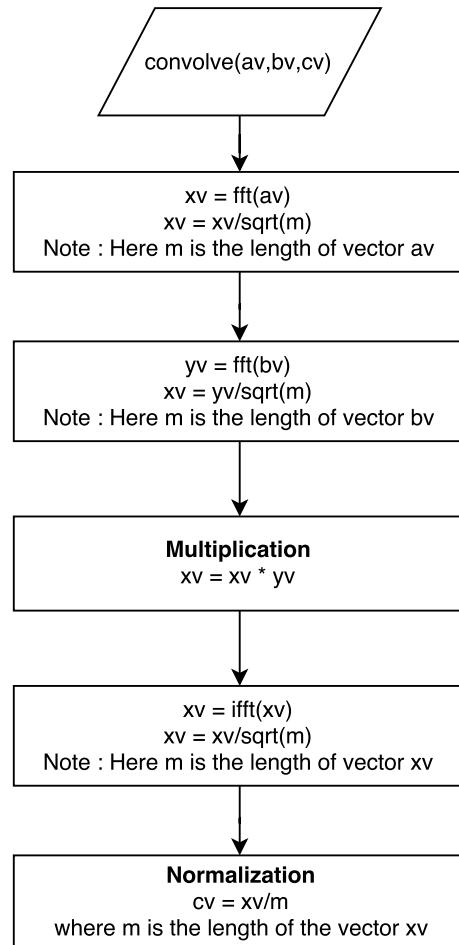


Figure 9.5: Circular convolution algorithm

### 9.1.0.8 Test example

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

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

**Step 2 :** Find the **fft\_test.m** file and open it.

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

```

1 %
2 %
3 %
4 %
5 %
6 %
7 %
8 %
9 %
10 %
11 %
12 %
13 %
14 %
15 %
16 %
17 %
18 %
19 %
20 %
21 %
22 %
23 %
24 %
25 %
26 %
27 %
28 %
29 %
30 %
31 %

%Choose for sig a value between [1, 7]
sig = 7;
switch sig
    case 1
        signal_title = 'Signal with one signusoid and random noise';
        S = 0.7*sin(2*pi*50*t);
        X = S + 2*randn(size(t));
    case 2
        signal_title = 'Sinusoids with Random Noise';
        S = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
        X = S + 2*randn(size(t));
    case 3
        signal_title = 'Single sinusoids';
        X = sin(2*pi*t);
    case 4
        signal_title = 'Summation of two sinusoids';
        X = sin(2*pi*t) + cos(2*pi*t);
    case 5
        signal_title = 'Single Sinusoids with Exponent';

```

```

33 X = sin(2*pi*200*t).*exp(-abs(70*t));
34 case 6
35     signal_title = 'Mixed signal 1';
36     X = sin(2*pi*10*t).*exp(-t)+sin(2*pi*t)+7*sin(2*pi*+5*t)+7*cos(2*pi*+20*t)+5*sin(2*pi*+50*t);
37 case 7
38     signal_title = 'Mixed signal 2';
39     X = 2*sin(2*pi*100*t).*exp(-t)+2.5*sin(2*pi*+250*t)+sin(2*pi*+50*t).*cos(2*pi*+20*t)+1.5*sin(2*pi*+50*t).*sin(2*pi*+150*t);
40 case 8
41     signal_title = 'Sinusoid tone';
42     X = cos(2*pi*100*t);
43 end
44
45 plot(t(1:end),X(1:end))
46 title ( signal_title )
47 axis([ min(t) max(t) 1.1*min(X) 1.1*max(X)]);
48 xlabel('t (s)')
49 ylabel('X(t)')
50 grid on
51
52 % dlmwrite will generate text file which represents the time domain signal.
53 % dlmwrite('time_function.txt', X, 'delimiter','\t');
54 fid=fopen('time_function.txt','w');
55 b=fprintf(fid ,'%0.15f\n',X); % 15-Digit accuracy
56 fclose(fid);
57
58 tic
59 fy = ifft (X)*sqrt(length(X));% According to the definition of "optical fft "
60 % with the (1/sqrt(N)) concept.
61 toc
62 fy = fftshift (fy);
63 figure(2);
64 subplot(2,1,1)
65 plot(f,abs(fy));
66 axis([-Fs/(2*5) Fs/(2*5) 0 1.1*max(abs(fy))]);
67 xlabel('f');
68 ylabel('|Y(f)|');
69 title ('MATLAB program Calculation : Magnitude');
70 grid on
71 subplot(2,1,2)
72 plot(f,phase(fy));
73 xlim([-Fs/(2*5) Fs/(2*5)]);
74 xlabel('f');
75 ylabel('phase(Y(f))');
76 title ('MATLAB program Calculation : Phase');
77 grid on
78
79 %%%
80 %
81 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SECTION 2

```

```

%%%%%
%
83 % Read C++ transformed data file
fullData = load('frequency_function.txt');
85 A=1;
B=A+1;
87 l=1;
Z=zeros(length(fullData)/2,1);
89 while (l<=length(Z))
Z(l) = fullData(A)+fullData(B)*1i;
91 A = A+2;
B = B+2;
93 l=l+1;
end
95
% % Comparsion of the MATLAB and C++ fft calculation.
97 figure;
98 subplot(2,1,1)
99 plot(f,abs(fftshift ( ifft (X)*sqrt(length(X))))) 
hold on
101 %Multiplied by sqrt(n) to verify our C++ code with MATLAB implemenrtation.
102 %plot(f,(sqrt(length(Z))*abs( fftshift (Z))), '---o')
103 plot(f,abs( fftshift (Z)), '---o') % fft from C++
104 axis([-Fs/(2*5) Fs/(2*5) 0 1.1*max(abs(fy))]);
105 xlabel('f (Hz)');
106 title ('Main reference for Magnitude')
107 legend('MATLAB','C++')
grid on
108 subplot(2,1,2)
109 plot(f,phase( fftshift ( ifft (X))))
110 hold on
111 plot(f,phase( fftshift (Z)), '---o')
112 xlim([-Fs/(2*5) Fs/(2*5)])
113 title ('Main reference for Phase')
114 xlabel('f (Hz)');
115 legend('MATLAB','C++')
grid on
116
117 %
118 % % IFFT test comparision Plot
119 % figure; plot(X); hold on; plot(real(Z),'---o');
120

```

Listing 9.1: fft\_test.m code

**Step 3 :** Choose for sig a value between [1, 7] and run the first section namely **section 1** by pressing "ctrl+Enter".

This will generate a *time\_function.txt* file in the same folder which contains the time domain signal data.

**Step 4 :** Now, find the **fft\_test.vcxproj** file in the same folder and open it.

In this project file, find *fft\_test.cpp* and click on it. This file is an example of FFT calculation using C++ program. Basically this *fft\_test.cpp* file consists of four sections:

**Section 1.** Read the input text file (import "time\_function.txt" data file)

**Section 2.** It calculates FFT.

**Section 3.** Save FFT calculated data (export *frequency\_function.txt* data file).

**Section 4.** Displays in the screen the FFT calculated data and length of the data.

```

1 # include "fft_20180208.h"
2 # include <complex>
3 # include <fstream>
4 # include <iostream>
5 # include <math.h>
6 # include <stdio.h>
7 # include <string>
8 # include <strstream>
9 # include <algorithm>
10 # include <vector>
11 #include <iomanip>

13 using namespace std;

15 int main()
{
    //////////////////////////////////////////////////////////////////// Section 1 ///////////////////////////////
    //////////////////////////////////////////////////////////////////// Read the input text file (import "time_function.txt") ///////////////////
    //////////////////////////////////////////////////////////////////// /////////////////////////////////
19 ifstream inFile;
21 inFile.precision(15);
22 double ch;
23 vector <double> inTimeDomain;
24 inFile.open("time_function.txt");
25 // First data (at 0th position) applied to the ch it is similar to the "cin".
26 inFile >> ch;
27 // It' ll count the length of the vector to verify with the MATLAB
28 int count=0;
29 while (!inFile.eof()){
30     // push data one by one into the vector
31     inTimeDomain.push_back(ch);
32     // it' ll increase the position of the data vector by 1 and read full vector.
33     inFile >> ch;
34     count++;
35 }
36 inFile.close(); // It is mandatory to close the file at the end.
37 //////////////////////////////////////////////////////////////////// Section 2 ///////////////////////////////
38 //////////////////////////////////////////////////////////////////// Calculate FFT ///////////////////////////////
39 //////////////////////////////////////////////////////////////////// /////////////////////////////////
41 vector <complex<double>> inTimeDomainComplex(inTimeDomain.size());

```

```

43 vector <complex<double>> fourierTransformed;
44 vector <double> re(inTimeDomain.size());
45 vector <double> im(inTimeDomain.size());

47 for (unsigned int i = 0; i < inTimeDomain.size(); i++)
48 {
49     re[i] = inTimeDomain[i]; // Real data of the signal
50 }
51
52 // Next, Real and Imaginary vector to complex vector conversion
53 inTimeDomainComplex = reImVect2ComplexVector(re, im);

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

61 //////////////////////////////// Section 3 ///////////////////////////////
62 ////////////////// Save FFT calculated data (export "frequency_function.txt" ) //////////////////
63 //////////////////////////////// //////////////////////////////// ////////////////////////////////
64 ofstream outFile;
65 complex<double> outFileData;
66 outFile.open("frequency_function.txt");
67 outFile.precision(15);
68 for (unsigned int i = 0; i < fourierTransformed.size(); i++){
69     outFile << fourierTransformed[i].real() << endl;
70     outFile << fourierTransformed[i].imag() << endl;
71 }
72 outFile.close();

73 //////////////////////////////// Section 4 ///////////////////////////////
74 ////////////////// Display Section ///////////////////////////////
75 //////////////////////////////// //////////////////////////////// ////////////////////////////////
76 for (unsigned int i = 0; i < fourierTransformed.size(); i++){
77     cout << fourierTransformed[i] << endl; // Display all FFT calculated data
78 }
79 cout << "\n\nTime elapsed to calculate FFT : " << elapsed_secs << " seconds" << endl;
80 cout << "\nTotal length of of data :" << count << endl;
81 getchar();
82 return 0;
83 }
```

Listing 9.2: fft\_test.cpp code

**Step 5 :** Run the *fft\_test.cpp* file.

This will generate a *frequency\_function.txt* file in the same folder which contains the Fourier transformed data.

**Step 6 :** Now, go to the *fft\_test.m* and run section 2 in the code by pressing "ctrl+Enter".

The section 2 reads *frequency\_function.txt* and compares both C++ and MATLAB calculation

of Fourier transformed data.

#### 9.1.0.9 Resultant analysis of various test signals

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

##### 9.1.0.10 1. Signal with two sinusoids and random noise

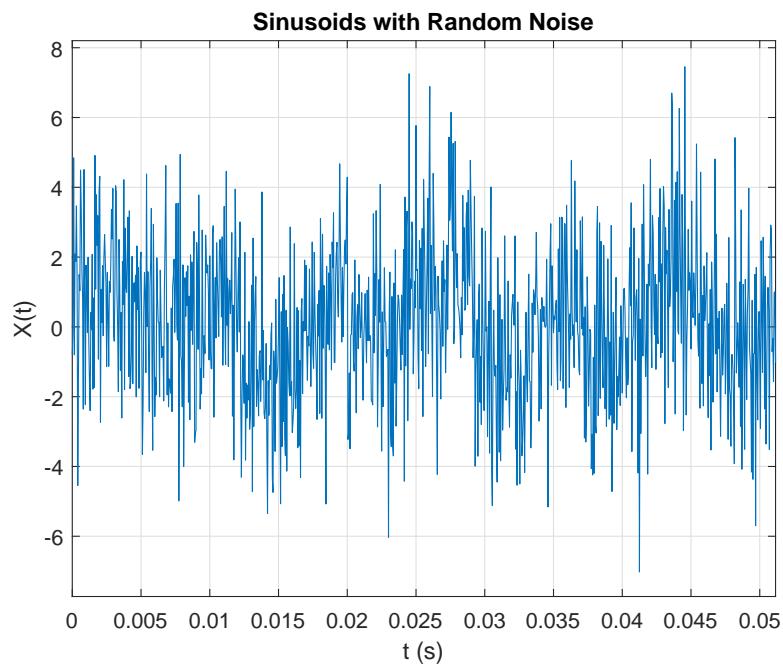


Figure 9.6: Random noise and two sinusoids

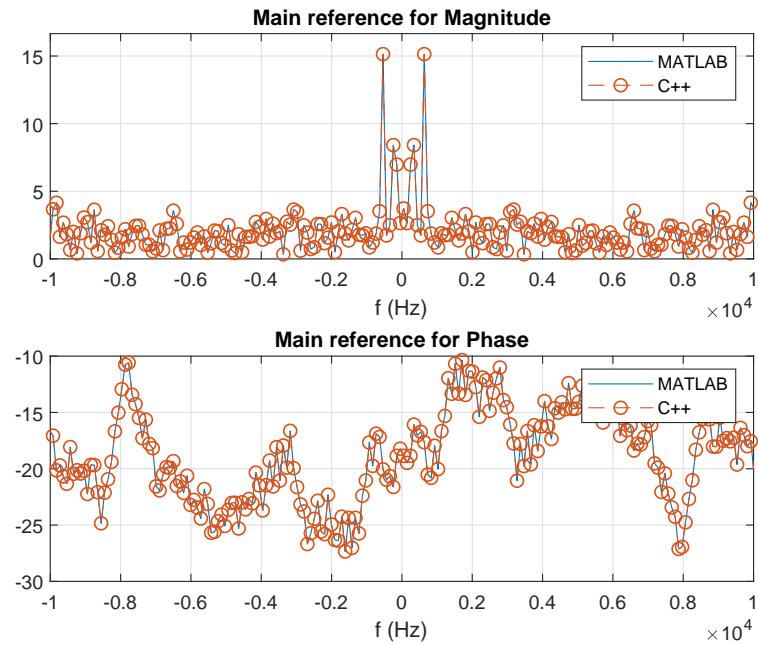


Figure 9.7: MATLAB and C++ comparison

#### 9.1.0.11 2. Sinusoid with an exponent

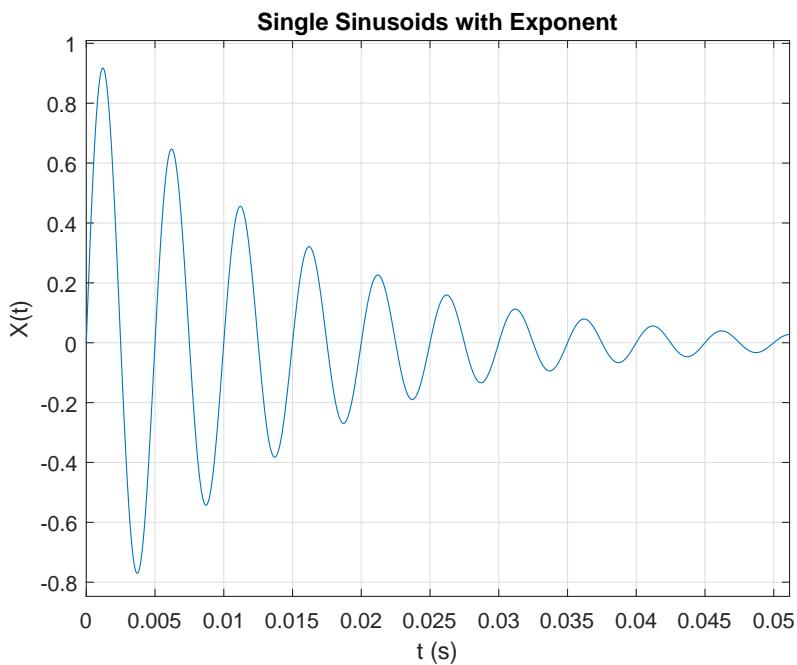


Figure 9.8: Sinusoids with exponent

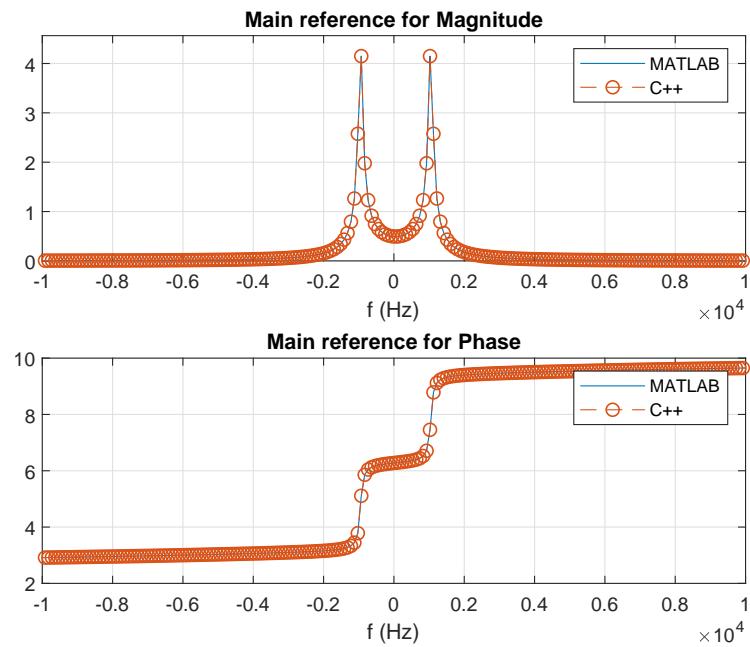


Figure 9.9: MATLAB and C++ comparison

#### 9.1.0.12 3. Mixed signal

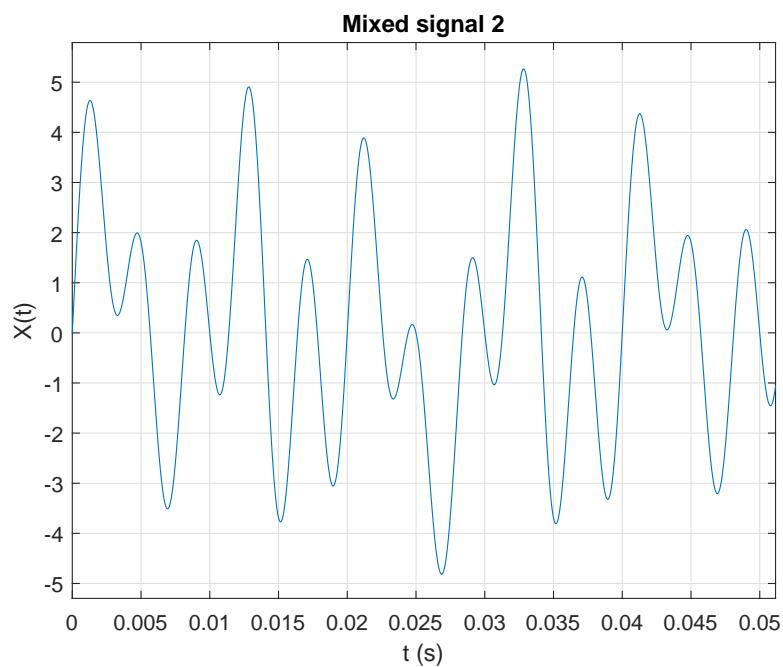


Figure 9.10: mixed signal

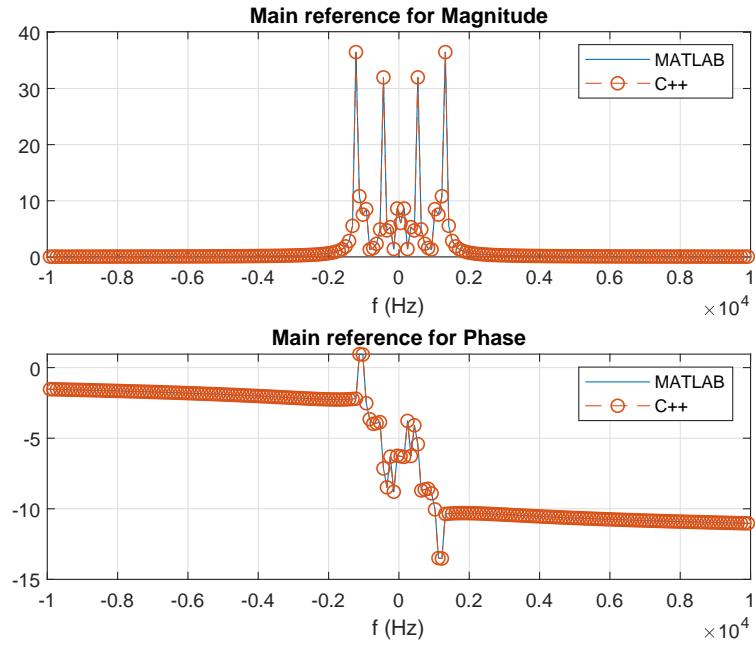


Figure 9.11: MATLAB and C++ comparison

## Remarks

To write the data from the MATLAB in the form of text file, **fprintf** MATLAB function was used with the accuracy of the 15 digits. Similarly; to write the fft calculated data from the C++ in the form of text file, C++ **double** data type with precision of 15 digits applied to the object of **ofstream** class.

## Optimized FFT

### 9.1.0.13 Algorithm

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

$$X_k = \sum_{n=0}^{N-1} x_n e^{m i 2\pi k n / N} \quad 0 \leq k \leq N - 1 \quad (9.4)$$

Similarly, for IFFT,

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{m i 2\pi k n / N} \quad 0 \leq k \leq N - 1 \quad (9.5)$$

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

#### 9.1.0.14 Function description

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

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

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

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

If we manipulate the optimized FFT and IFFT functions as  $y = fft(x, 1, 0)$  and  $x = fft(y, -1, 0)$  then it'll calculate the FFT and IFFT as discussed in equation 9.1 and 9.2 respectively.

#### 9.1.0.15 Comparative analysis

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

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

## References

- [1] K. Ramamohan (Kamisetty Ramamohan) Rao, D. N. Kim, and J. J. Hwang. *Fast Fourier transform : algorithms and applications*. Springer, 2010, p. 423. ISBN: 9781402066290.
- [2] Eleanor Chin-hwa Chu and Alan. George. *Inside the FFT black box : serial and parallel fast Fourier transform algorithms*. CRC Press, 2000, p. 312. ISBN: 9781420049961. URL: <https://www.crcpress.com/Inside-the-FFT-Black-Box-Serial-and-Parallel-Fast-Fourier-Transform-Algorithms/Chu-George/p/book/9780849302701>.

## 9.2 Overlap-Save Method

|                    |   |                        |
|--------------------|---|------------------------|
| <b>Header File</b> | : | overlap_save_*.h       |
| <b>Source File</b> | : | overlap_save_*.cpp     |
| <b>Version</b>     | : | 20180201 (Romil Patel) |

### Overlap-save using impulse response

Overlap-save is an efficient way to evaluate the discrete convolution between a very long signal and a finite impulse response (FIR) filter. The overlap-save procedure cuts the signal into equal length segments with some overlap and then it performs convolution of each segment with the FIR filter. The overlap-save method can be computed in the following steps [1, 2] :

**Step 1** : Determine the length  $M$  of impulse response,  $h(n)$ .

**Step 2** : Define the size of FFT and IFFT operation,  $N$ . The value of  $N$  must greater than  $M$  and it should in the form  $N = 2^k$  for the efficient implementation.

**Step 3** : Determine the length  $L$  to section the input sequence  $x(n)$ , considering that  $N = L + M - 1$ .

**Step 4** : Pad  $L - 1$  zeros at the end of the impulse response  $h(n)$  to obtain the length  $N$ .

**Step 5** : Make the segments of the input sequences of length  $L$ ,  $x_i(n)$ , where index  $i$  correspond to the  $i^{th}$  block. Overlap  $M - 1$  samples of the previous block at the beginning of the segmented block to obtain a block of length  $N$ . In the first block, it is added  $M - 1$  null samples.

**Step 6** : Compute the circular convolution of segmented input sequence  $x_i(n)$  and  $h(n)$  described as,

$$y_i(n) = x_i(n) \circledast h(n). \quad (9.6)$$

This is obtained in the following steps:

1. Compute the FFT of  $x_i$  and  $h$  both with length  $N$ .
2. Compute the multiplication of  $X_i(f)$  and the transfer function  $H(f)$ .
3. Compute the IFFT of the multiplication result to obtain the time-domain block signal,  $y_i$ .

**Step 7** : Discarded  $M - 1$  initial samples from the  $y_i$ , and save only the error-free  $N - M - 1$  samples in the output record.

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

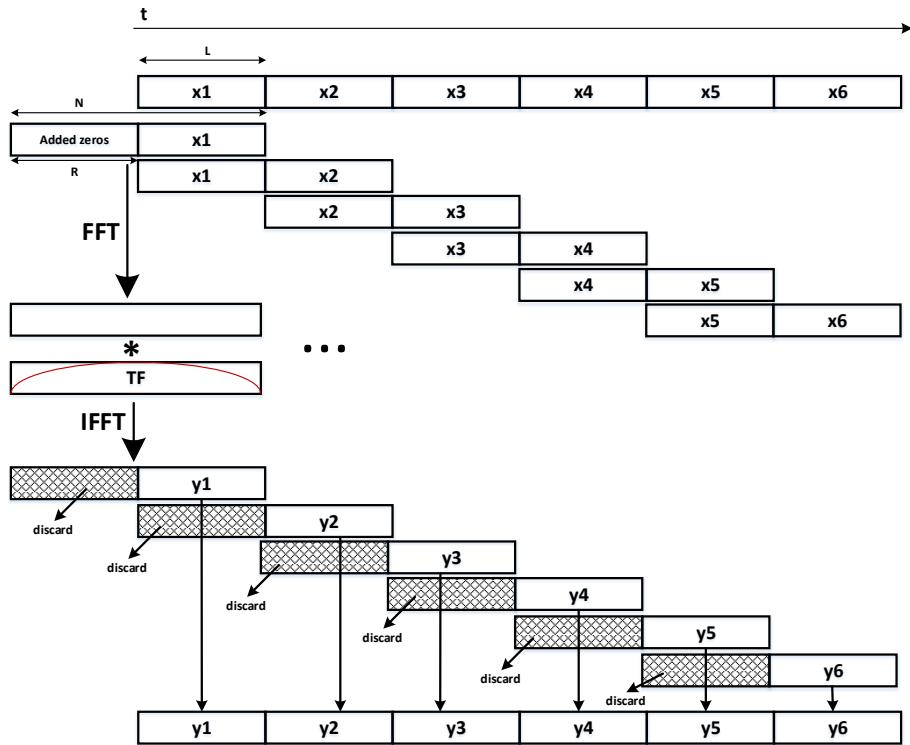


Figure 9.12: Illustration of Overlap-save method.

### Function description

Traditionally, overlap-save method performs the linear convolution between discrete time-domain signal  $x(n)$  and the filter impulse response  $h(n)$  with the help of circular convolution. Here the length of the signal  $x(n)$  is greater than the length of the filter  $h(n)$ . To perform convolution between the time domain signal  $x(n)$  with the filter  $h(n)$ , include the header file `overlap_save_*.h` and then supply input argument to the function as follows,

$$y(n) = \text{overlapSave}(x(n), h(n))$$

Where,  $x(n)$ ,  $h(n)$  and  $y(n)$  are of the C++ type vector< complex<double> > and the length of the signal  $x(n)$  and filter  $h(n)$  could be arbitrary.

The one noticeable thing in the traditional way of implementation of overlap-save is that it cannot work with the real-time system. Therefore, to make it usable in the real-time environment, one more `overlapSave` function with three input parameters was implemented and used along with the traditional overlap-save method. The structure of the new function is as follows,

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

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

### Linear and circular convolution

In the circular convolution, if we determine the length of the signal  $x(n)$  is  $N_1 = 8$  and length of the filter is  $h(n)$  is  $N_2 = 5$ ; then the length of the output signal is determined by  $N = \max(N_1, N_2) = 8$ . Next, the circular convolution can be performed after padding 0 in the filter  $h(n)$  to make it's length equals  $N$ .

In the linear convolution, if we determine the length of the signal  $x(n)$  is  $N_1 = 8$  and length of the filter is  $h(n)$  is  $N_2 = 5$ ; then the length of the output signal is determined by  $N = N_1 + N_2 - 1 = 12$ . Next, the linear convolution using circular convolution can be performed after padding 0 in the signal  $x(n)$  filter  $h(n)$  to make it's length equals  $N$ .

### Flowchart of real-time overlap-save method

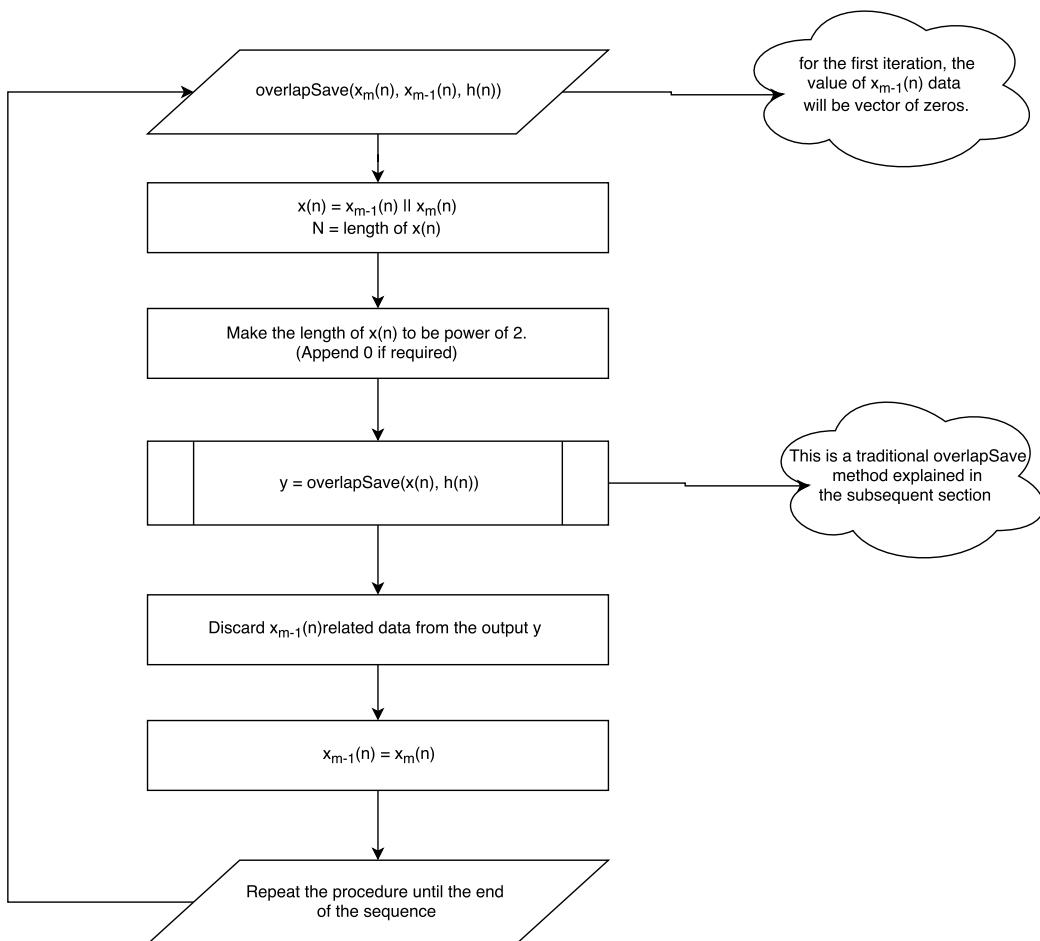


Figure 9.13: Flowchart for real-time overlap-save method

### Flowchart of traditional overlap-save method

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

#### 1. Decide length of FFT, data block and filter

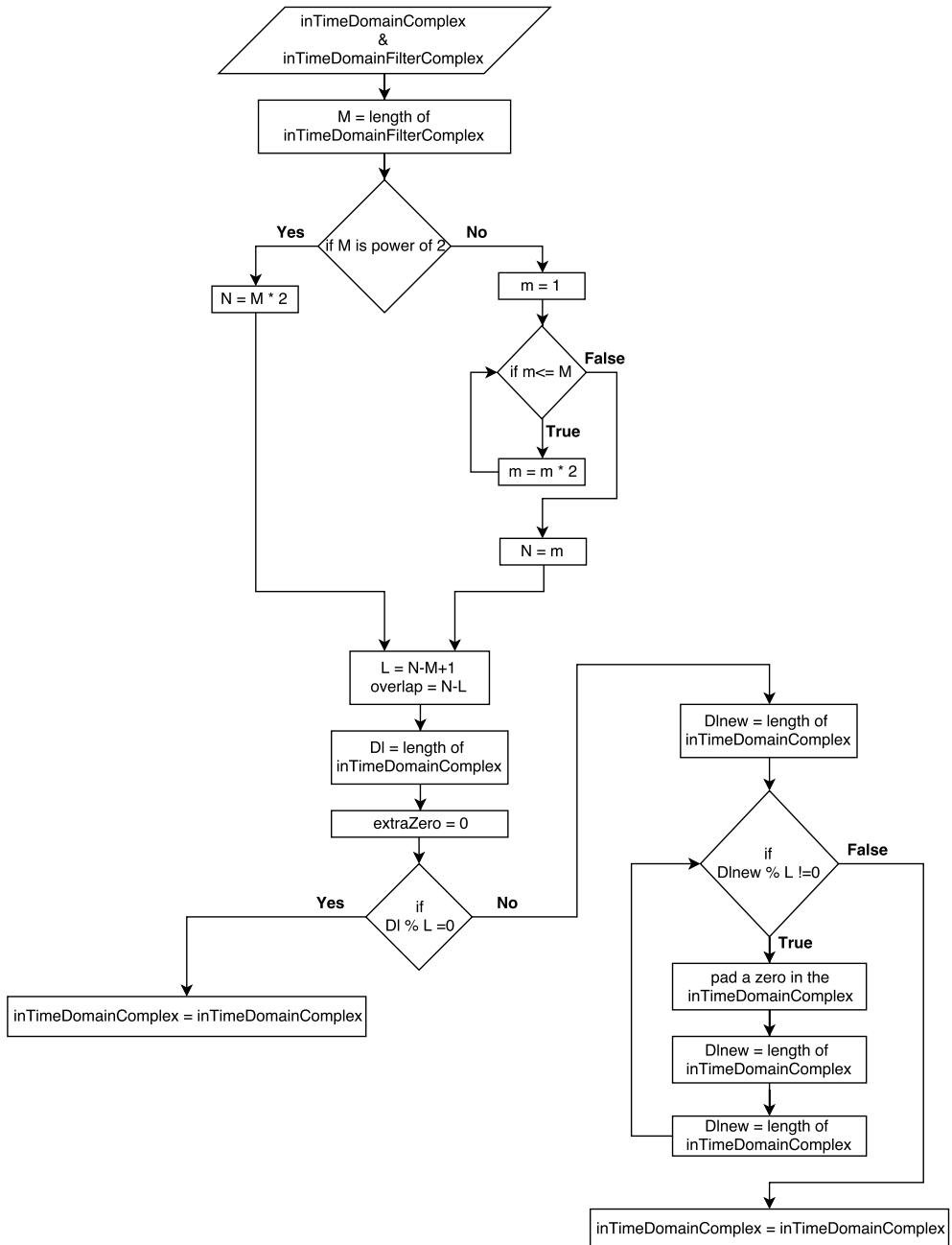


Figure 9.14: Flowchart for calculating length of FFT, data block and filter

## 2. Create matrix with overlap

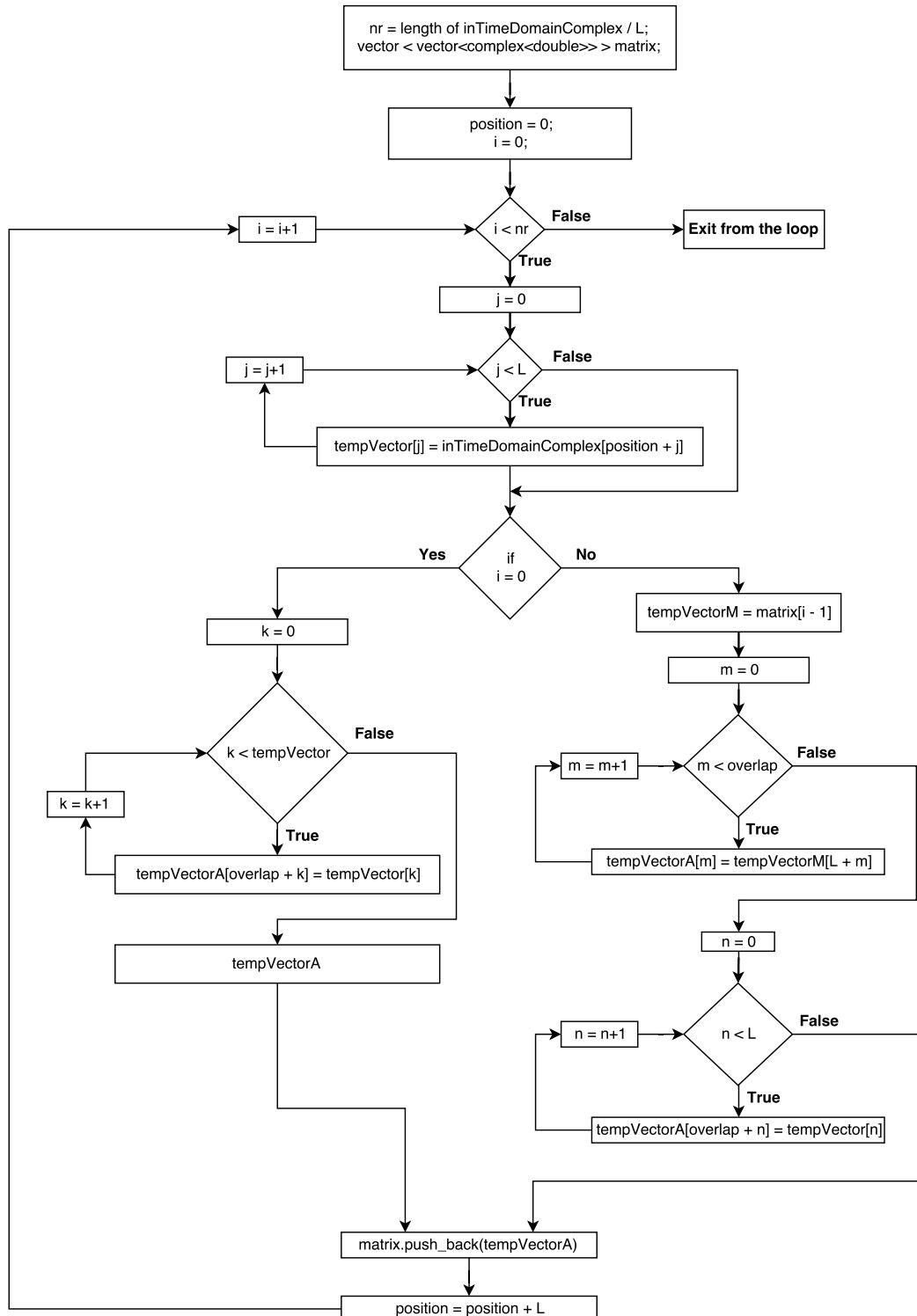


Figure 9.15: Flowchart of creating matrix with overlap

### 3. Convolution between filter and data blocks

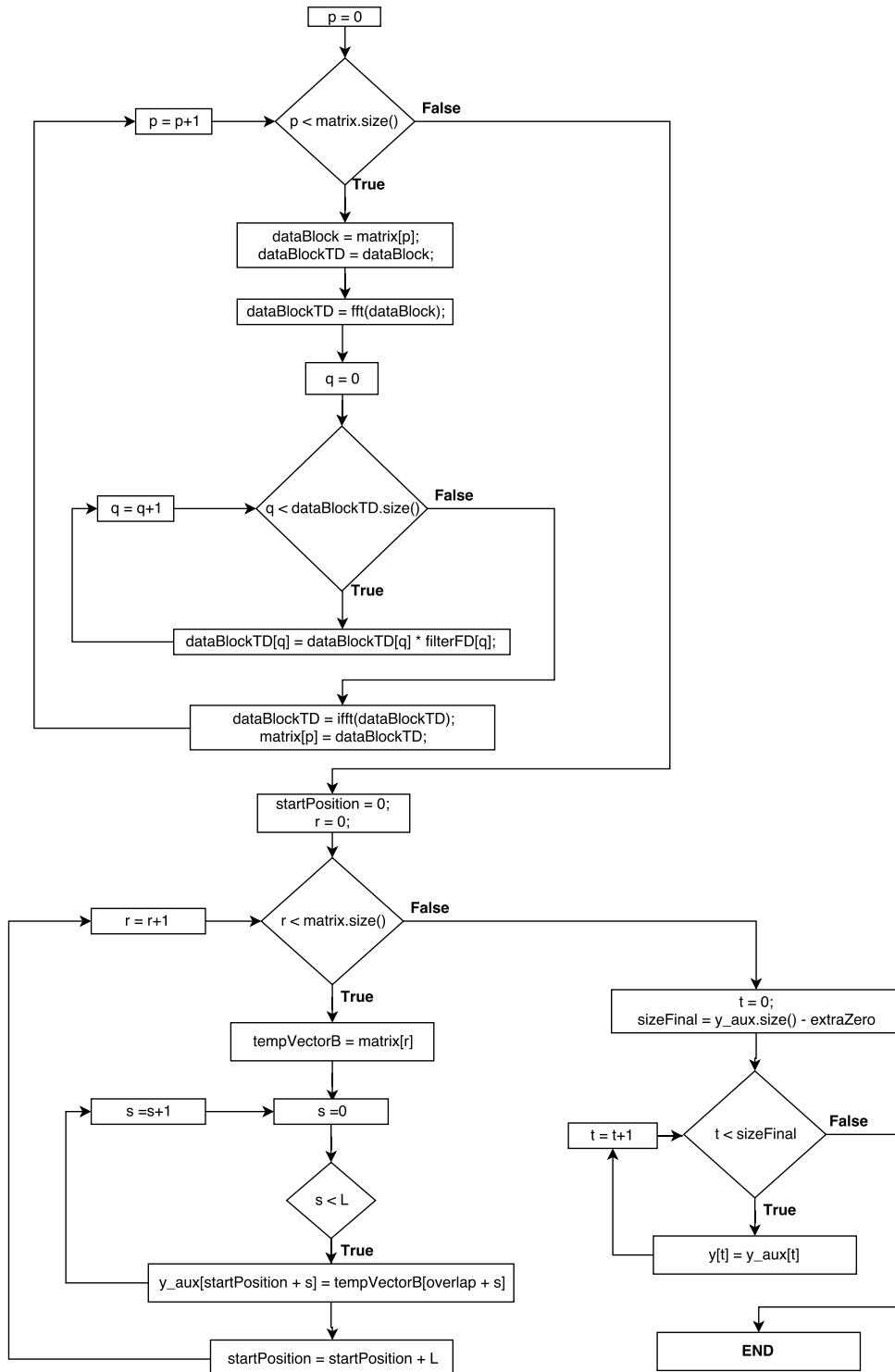


Figure 9.16: Flowchart of the convolution

### Test example of traditional overlap-save function

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

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

**Step 2 :** Find the **overlapSave\_test.m** file and open it.

This overlapSave\_test.m consists of five sections:

**section 1 :** It generates the time domain signal and filter impulse response and save them in the form of the text file with the name of *time\_domain\_data.txt* and *time\_domain\_filter.txt* respectively in the same folder.

**Section 2 :** It calculates the length of FFT, data blocks and filter to perform convolution using overlap-save method.

**Section 3 :** It consists of overlap-save code which first converts the data into the form of matrix with 50% overlap and then performs circular convolution with filter.

**Section 4 :** It read *overlap\_save\_data.txt* data file generated by C++ program and compare with MATLAB implementation.

**Section 5 :** It compares our MATLAB and C++ implementation with the built-in MATLAB function conv().

```

%
%%%%%%%%%%%%%%%
2 %%%%%%%%%%%%%%%% SECTION 1
    %%%%%%%%%%%%%%
%
%
4 % generate signal and filter data and save it as a .txt file .
clc
6 clear all
close all
8
10 Fs = 1e5;           % Sampling frequency
12 T = 1/Fs;          % Sampling period
L = 2^10;             % Length of signal
14 t = (0:L-1)*(5*T); % Time vector
f = linspace(-Fs/2,Fs/2,L);
16
18 %Choose for sig a value between [1, 7]
sig = 7;
switch sig
    case 1
        signal_title = 'Signal with one signusoid and random noise';
        S = 0.7*sin(2*pi*50*t);
20

```

```

X = S + 2*randn(size(t));
22 case 2
    signal_title = 'Sinusoids with Random Noise';
24     S = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
     X = S + 2*randn(size(t));
26 case 3
    signal_title = 'Single sinusoids';
28     X = sin(2*pi*t);
case 4
30     signal_title = 'Summation of two sinusoids';
     X = sin(2*pi*1205*t) + cos(2*pi*1750*t);
32 case 5
    signal_title = 'Single Sinusoids with Exponent';
34     X = sin(2*pi*250*t).*exp(-70*abs(t));
case 6
36     signal_title = 'Mixed signal 1';
     X = sin(2*pi*10*t).*exp(-t)+sin(2*pi*t)+7*sin(2*pi*+5*t)+7*cos(2*pi*+20*t)+5*sin(2*pi*+50*t);
38 case 7
    signal_title = 'Mixed signal 2';
40     X = 2*sin(2*pi*100*t).*exp(-t)+2.5*sin(2*pi*+250*t)+sin(2*pi*+50*t).*cos(2*pi*+20*t)+1.5*sin(2*pi*+50*t).*sin(2*pi*+150*t);
end
42 Xref = X;
44 % dlmwrite will generate text file which represents the time domain signal.
%dlmwrite('time_domain_data.txt',X,'delimiter','\t');
46 fid=fopen('time_domain_data.txt','w');
fprintf(fid,'%.20f\n',X); % 12-Digit accuracy
fclose(fid);

50 % Choose for filt a value between [1, 3]
52 filt = 1;
switch filt
54     case 1
        filter_type = 'Impulse response of rcos filter';
56         h = rcosdesign(0.25,11,6);
    case 2
        filter_type = 'Impulse response of rrcos filter';
58         h = rcosdesign(0.25,11,6,'sqrt');
    case 3
        filter_type = 'Impulse response of Gaussian filter';
60         h = gaussdesign(0.25,11,6);
end
64 %dlmwrite('time_domain_filter.txt',h,'delimiter','\t');
66 fid=fopen('time_domain_filter.txt','w');
fprintf(fid,'%.20f\n',h); % 20-Digit accuracy
fclose(fid);

70 figure;

```

```

72 subplot(211)
73 plot(t,X)
74 grid on
75 title('Signal Plot')
76 axis([min(t) max(t) 1.1*min(X) 1.1*max(X)]);
77 xlabel('t (s)')
78 ylabel('X(t)')

80 subplot(212)
81 plot(h)
82 grid on
83 title('Filter Plot')
84 axis([1 length(h) 1.1*min(h) 1.1*max(h)]);
85 xlabel('Samples')
86 ylabel('h(t)')

88 %%
89 %

%%%%%%%%%%%%%% SECTION 2
90 %

92 % Calculate the length of FFT, data blocks and filter
M = length(h);

94 if (bitand(M,M-1)==0)
95     N = 2 * M; % Where N is the size of the FFT
96 else
97     m = 1;
98     while(m<=M) % Next value of the order of power 2.
99         m = m*2;
100    end
101    N = m;
102 end
103

104 L = N - M + 1; % Size of data block (50% of overlap)
105 overlap = N - L; % size of overlap
Dl = length(X);
107 extraZeros = 0;
108 if (mod(Dl,L) == 0)
109     X = X;
110 else
111     Dlnew = length(X);
112     while (mod(Dlnew,L) ~= 0)
113         X = [X 0];
114         Dlnew = length(X);
115         extraZeros = extraZeros + 1;
116     end
117 end
118

```

```

%%%
%
%%%%% SECTION 3
%
%%% MATLAB approach of overlap-save method (First create matrix with
%%% overlap and then perform convolution)
zerosForFilter = zeros(1,N-M);
h1=[h zerosForFilter];
H1 = fft(h1);

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

tic
for k=1:nr
    Ma(k,:)=x1(((k-1)*L+1):k*L);
    if k==1
        Ma1(k,:)=[zeros(1,overlap) Ma(k,:)];
        % Ma1(k,:)=[Ma(1:overlap) Ma(k,:)];
    else
        tempVectorM = Ma1(k-1,:);
        overlapData = tempVectorM(L+1:end);
        Ma1(k,:)=[overlapData Ma(k,:)];
    end
    auxfft = fft (Ma1(k,:));
    auxMult = auxfft.*H1;
    Ma2(k,:)=ifft (auxMult);
end

Ma3=Ma2(:,N-L+1:end);
y1=Ma3';
y=y1(:)';
y = y(1:end - extraZeros);
toc
%%%
%
%%%%% SECTION 4
%
%%% Read overlap-save data file generated by C++ program and compare with
fullData = load('overlap_save_data.txt');
A=1;
B=A+1;

```

```

l=1;
162 Z=zeros(length(fullData)/2,1);
while (l<=length(Z))
164 Z(l) = fullData(A)+fullData(B)*1i;
A = A+2;
166 B = B+2;
l=l+1;
168 end

170 figure;
plot(t,real(y))
172 hold on
plot(t,real(Z),'o')
174 axis([ min(t) max(t) 1.1*min(y) 1.1*max(y)]);
 xlabel('t (Seconds)')
176 ylabel('y(t)')
 title ('Comparision of overlapSave method of MATLAB and C++ ')
178 legend('MATLAB overlapSave','C++ overlapSave')
grid on
180 %%
181 %
%%%%%%%%%%%%%%%
182 %%%%%%%%%%%%%% SECTION 5
%%%%%%%%%%%%%%%
183 %
%%%%%%%%%%%%%%%
184 % Our MATLAB and C++ implementation test with the built-in conv function of
% MATLAB.
185 tic
P = conv(Xref,h);
187 toc
figure
188 plot(t, P(1:size(Z,1)), 'r')
hold on
189 plot(t, real(Z), 'o')
title ('Comparision of MATLAB function conv() and overlapSave')
190 axis([ min(t) max(t) 1.1*min(real(Z)) 1.1*max(real(Z))]);
 xlabel('t (Seconds)')
192 ylabel('y(t)')
legend('MATLAB function : conv(X,h)','C++ overlapSave')
grid on
193
194
195
196
197
198

```

Listing 9.3: overlapSave\_test.m code

**Step 3 :** Choose for a sig and filt value between [1 7] and [1 3] respectively and run the first three sections namely **section 1**, **section 2** and **section 3**.

This will generate a *time\_domain\_data.txt* and *time\_domain\_filter.txt* file in the same folder which contains the time domain signal and filter data respectively.

**Step 4 :** Find the **overlapSave\_test.vcxproj** file in the same folder and open it.

In this project file, find *overlapSave\_test.cpp* in *SourceFiles* section and click on it. This file is an example of using *overlapSave* function. Basically, *overlapSave\_test.cpp* file consists of four sections:

**Section 1 :** It reads the *time\_domain\_data.txt* and *time\_domain\_filter.txt* files.

**Section 2 :** It converts signal and filter data into complex form.

**Section 3 :** It calls the *overlapSave* function to perform convolution.

**Section 4 :** It saves the result in the text file namely *overlap\_save\_data.txt*.

```

1 # include "overlap_save_20180208.h"
2
3 # include <complex>
4 # include <fstream>
5 # include <iostream>
6 # include <math.h>
7 # include <stdio.h>
8 # include <string>
9 # include <strstream>
10 # include <algorithm>
11 # include <vector>
12
13 using namespace std;
14
15 int main()
16 {
17     //////////////////////////////// Section 1 ///////////////////////////////
18     // Read the time_domain_data.txt and time_domain_filter.txt files //
19     //////////////////////////////// ////////////////////////////////
20     ifstream inFile;
21     double ch;
22     vector <double> inTimeDomain;
23     inFile.open("time_domain_data.txt");
24
25     // First data (at 0th position) applied to the ch it is similar to the "cin".
26     inFile >> ch;
27
28     // It'll count the length of the vector to verify with the MATLAB
29     int count = 0;
30
31     while (!inFile.eof())
32     {
33         // push data one by one into the vector
34         inTimeDomain.push_back(ch);
35
36         // it'll increase the position of the data vector by 1 and read full vector.s
37         inFile >> ch;
38         count++;
39     }
40
41     inFile.close(); // It is mandatory to close the file at the end.

```

```

42    ifstream inFileFilter ;
44    double chFilter;
45    vector <double> inTimeDomainFilter;
46    inFileFilter .open("time_domain_filter.txt");
47    inFileFilter  >> chFilter;
48    int countFilter = 0;

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

56    //////////////////////////////// Section 2 ///////////////////////////////
57    //////////////////////////////// Real to complex conversion ///////////////////////////////
58    //////////////////////////////// For signal data ///////////////////////////////
59
60    vector <complex<double>> inTimeDomainComplex(inTimeDomain.size());
61    vector <complex<double>> fourierTransformed;
62    vector <double> re(inTimeDomain.size());
63    vector <double> im(inTimeDomain.size());

64    for (unsigned int i = 0; i < inTimeDomain.size(); i++)
65    {
66        re[i] = inTimeDomain[i]; // Real data of the signal
67        im[i] = 0;               // Imaginary data of the signal
68    }
69    // Next, Real and Imaginary vector to complex vector conversion
70    inTimeDomainComplex = reImVect2ComplexVector(re, im);

71    //////////////////////////////// For filter data ///////////////////////////////
72    vector <complex<double>> inTimeDomainFilterComplex(inTimeDomainFilter.size());
73    vector <double> reFilter(inTimeDomainFilter.size());
74    vector <double> imFilter(inTimeDomainFilter.size());

75    for (unsigned int i = 0; i < inTimeDomainFilter.size(); i++)
76    {
77        reFilter [i] = inTimeDomainFilter[i];
78        imFilter[i] = 0;
79    }
80    inTimeDomainFilterComplex = reImVect2ComplexVector(reFilter, imFilter);

81    //////////////////////////////// Section 3 ///////////////////////////////
82    //////////////////////////////// Overlap & save ///////////////////////////////
83
84    vector <complex<double>> y;
85    y = overlapSave(inTimeDomainComplex, inTimeDomainFilterComplex);

86    //////////////////////////////// Section 4 ///////////////////////////////

```

```

94 ////////////////////////////////////////////////////////////////// Save data //////////////////////////////////////////////////////////////////
95 ofstream outFile;
96 complex<double> outFileData;
97 outFile.precision(20);
98 outFile.open("overlap_save_data.txt");
99
100 for (unsigned int i = 0; i <y.size(); i++)
101 {
102     outFile << y[i].real() << endl;
103     outFile << y[i].imag() << endl;
104 }
105 outFile.close();
106
107 cout << "Execution finished! Please hit enter to exit." << endl;
108 getchar();
109 return 0;
110 }
```

Listing 9.4: overlapSave\_test.cpp code

**Step 5 :** Now, go to the **overlapSave\_test.m** and run section 4 and 5.

It'll display the graphs of comparative analysis of the MATLAB and C++ implementation of overlapSave program and also compares results with the MATLAB conv() function.

#### 9.2.0.1 Resultant analysis of various test signals

##### 9.2.0.2 1. Signal with two sinusoids and random noise

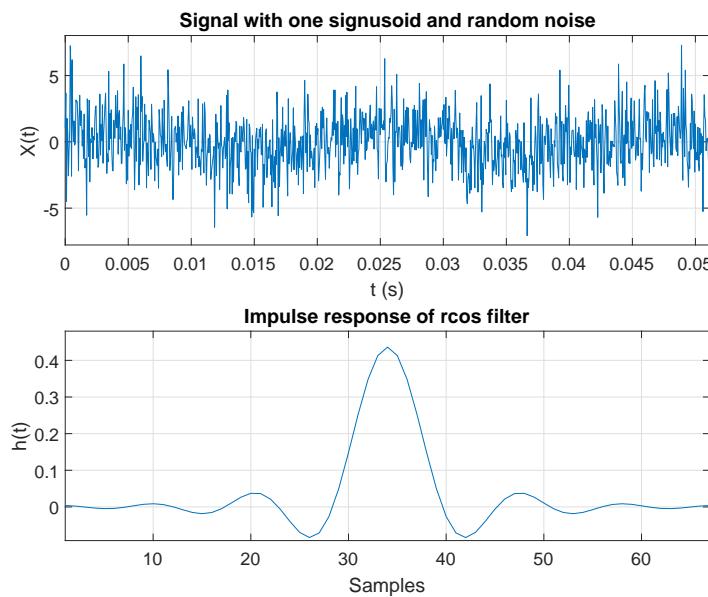


Figure 9.17: Random noise and two sinusoids signal &amp; Impulse response of rcos filter

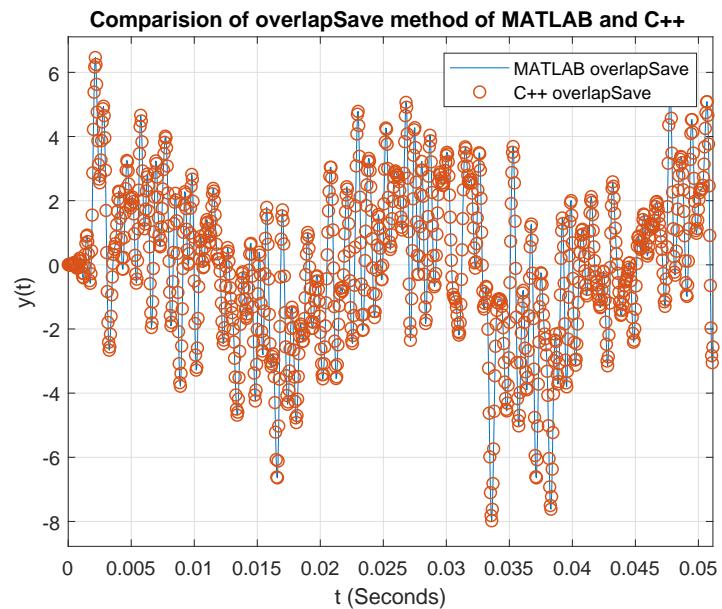


Figure 9.18: MATLAB and C++ comparison

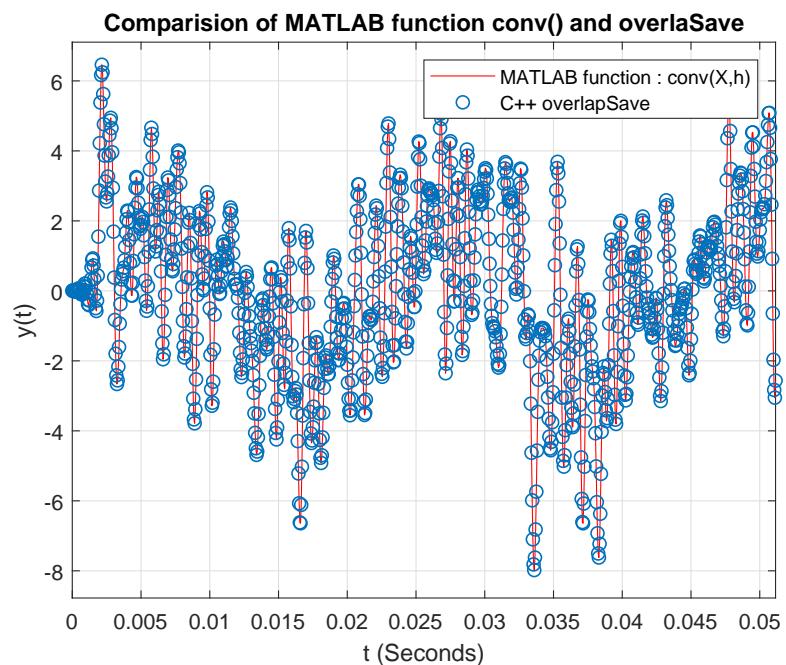


Figure 9.19: MATLAB function conv() and C++ overlapSave comparison

### 9.2.0.3 2. Mixed signal2

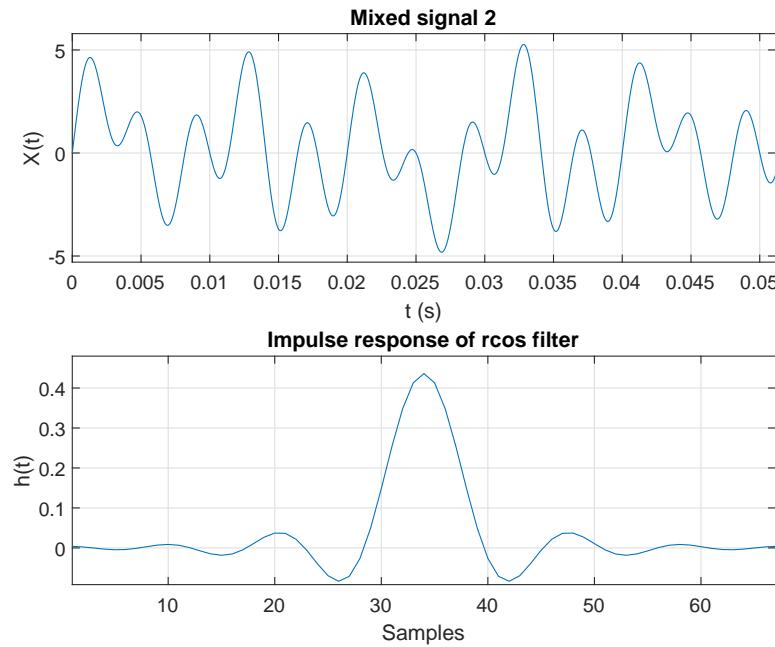


Figure 9.20: Mixed signal2 &amp; Impulse response of rcos filter

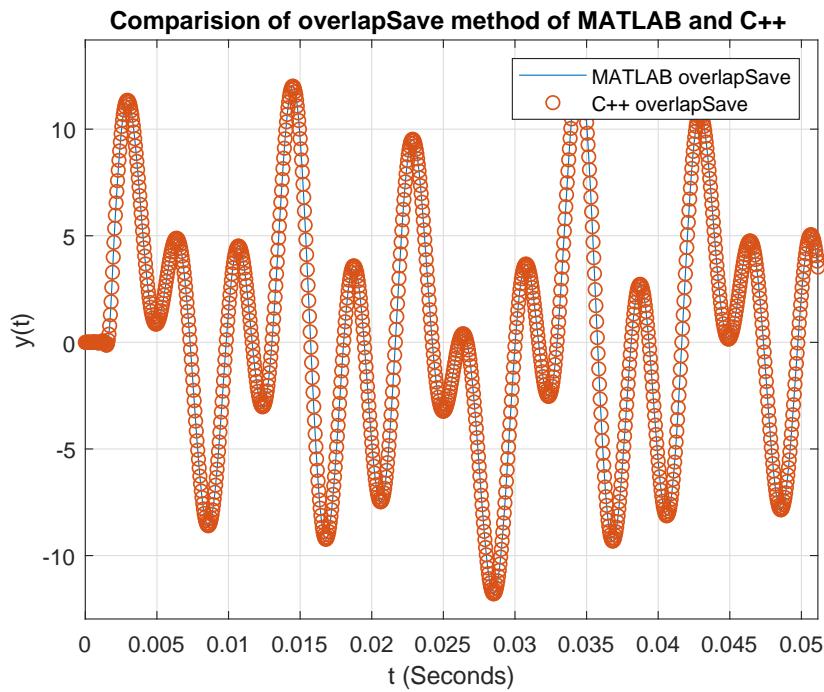


Figure 9.21: MATLAB and C++ comparison

#### 9.2.0.4 3. Sinusoid with exponent

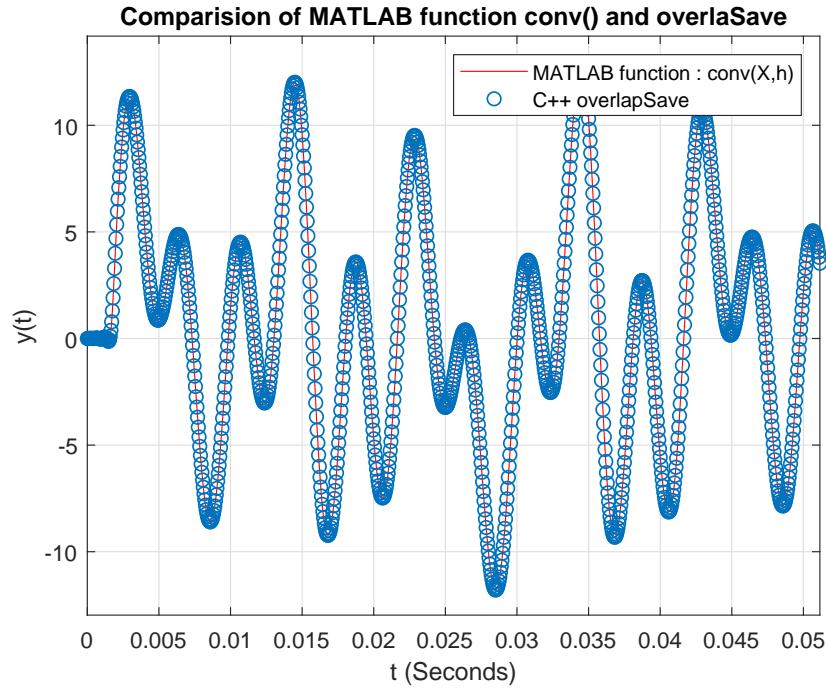


Figure 9.22: MATLAB function conv() and C++ overlapSave comparison

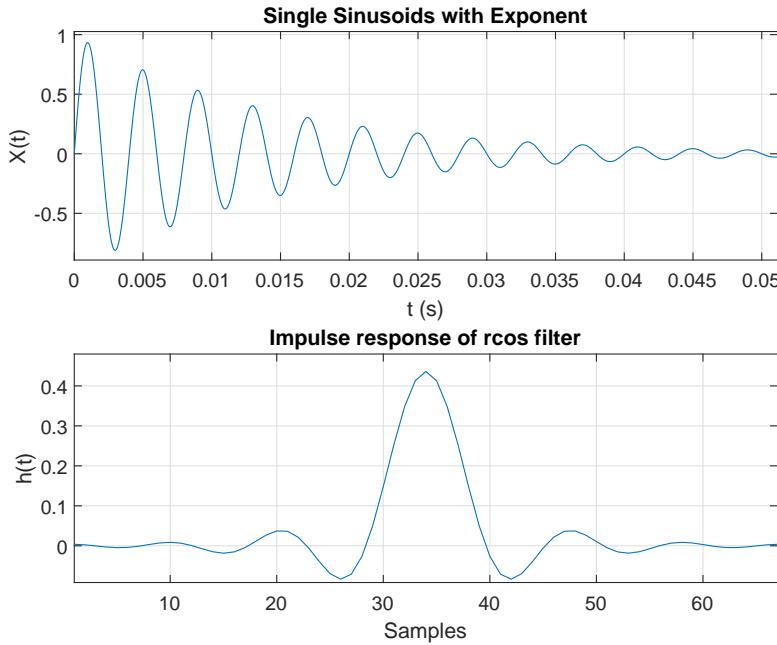


Figure 9.23: Sinusoid with exponent &amp; Impulse response of Gaussian filter

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

This section explains the steps of comparing real-time overlap-save method with the time-domain filtering. The structure of the real-time overlap-save function

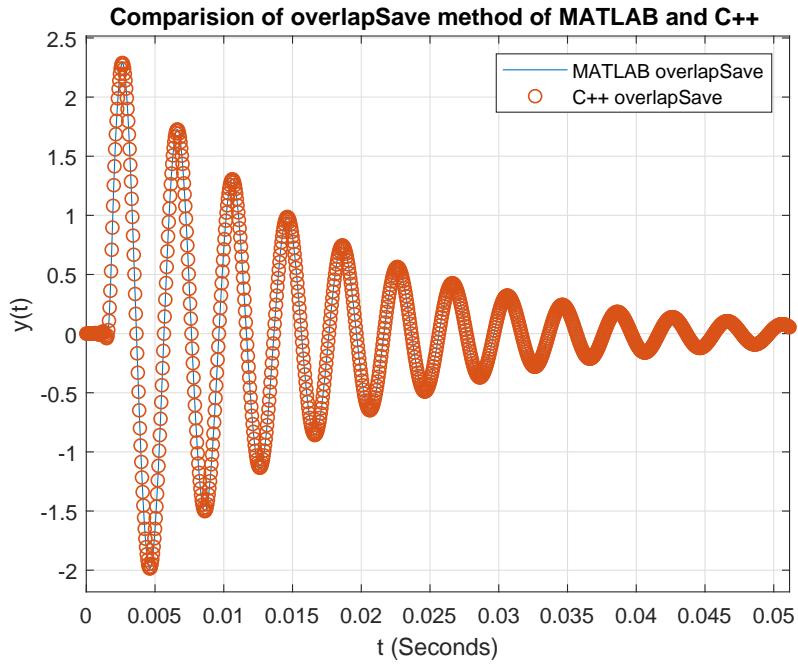


Figure 9.24: MATLAB and C++ comparison

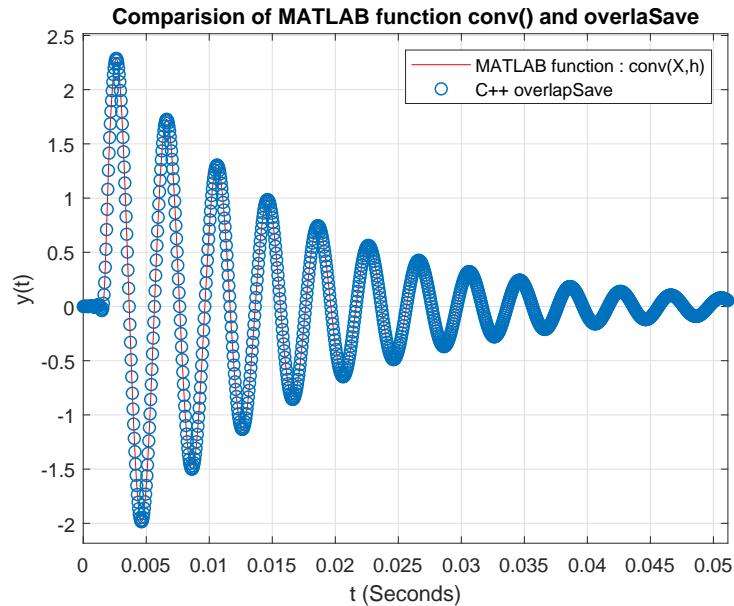


Figure 9.25: MATLAB function conv() and C++ overlapSave comparison

*overlapSave( $x_{m-1}(n), x_m(n), h(n)$ )* requires an impulse response  $h(n)$  of the filter. There are two methods to feed the impulse response to the real-time overlap-save function:

**Method 1.** The impulse response  $h(n)$  of the filter can be fed using the time-domain impulse response formula of the filter.

**Method 2.** Write the transfer function of the filter and convert it into the impulse response using Fourier transform method.

Here, this example uses the method 2 to feed the impulse response of the filter. In order to compare the result, follow the steps given below:

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

**Step 2 :** Find the **overlapSaveRealTime\_test.vcxproj** file and open it.

In this project file, find *filter\_20180306.cpp* in *SourceFiles* section and click on it. This file includes the several definitions of the two different filter class namely **FIR\_Filter** and **FD\_Filter** for filtering in time-domain and frequency-domain respectively. In this file, **FD\_Filter::runBlock** displays the logic of real-time overlap-save method.

```

1 //////////////////////////////////////////////////////////////////
2 ////////////////////////////////////////////////////////////////// FD_Filter //////////////////////////////////////////////////////////////////
3 //////////////////////////////////////////////////////////////////
4 void FD_Filter:: initializeFD_Filter (void)
5 {
6     outputSignals[0] ->symbolPeriod = inputSignals[0] ->symbolPeriod;
7     outputSignals[0] ->samplingPeriod = inputSignals[0] ->samplingPeriod;
8     outputSignals[0] ->samplesPerSymbol = inputSignals[0] ->samplesPerSymbol;
9
10    if (!getSeeBeginningOfImpulseResponse()) {
11        int aux = (int) ((double)impulseResponseLength) / 2) + 1;
12        outputSignals[0] ->setFirstValueToBeSaved(aux);
13    }
14
15    if (saveImpulseResponse) {
16        ofstream fileHandler("./signals/" + impulseResponseFilename, ios::out);
17        fileHandler << " // ### HEADER TERMINATOR ##\n";
18
19        t_real t;
20        double samplingPeriod = inputSignals[0] ->samplingPeriod;
21        for (int i = 0; i < impulseResponseLength; i++) {
22            t = -impulseResponseLength / 2 * samplingPeriod + i * samplingPeriod;
23            fileHandler << t << " " << impulseResponse[i] << "\n";
24        }
25        fileHandler.close();
26    }
27}
28
29 bool FD_Filter :: runBlock(void)
30 {
31    bool alive{ false };
32
33    int ready = inputSignals[0] ->ready();
34    int space = outputSignals[0] ->space();

```



```

////////// void FD_Filter_20181110::initializeFD_Filter_20181110(void)
87 {
88     outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
89     outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
90     outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;

93     if (!getSeeBeginningOfTransferFunction()) {
94         int aux = (int) ((double)transferFunctionLength) / 4 + 1;
95         outputSignals[0]->setFirstValueToBeSaved(aux);
96     }

97     if (saveTransferFunction)
98     {
99         ofstream fileHandler("./signals/" + transferFunctionFilename, ios::out);
100        fileHandler << "// ### HEADER TERMINATOR ###\n";
101

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

107     t_real f;
108     for (int k = 0; k < transferFunction.size(); k++)
109     {
110         f = -fWindow / 2 + k * df;
111         fileHandler << f << " " << transferFunction[k] << "\n";
112     }
113     fileHandler.close();
114 }
115 }

117 bool FD_Filter_20181110::runBlock(void)
118 {
119     bool alive{ false };

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

125     ////////// currentCopy //////////
126     ////////// currentCopy //////////
127     vector<t_complex> currentCopy(process); // Get the Input signal
128     t_real input;
129     for (int i = 0; i < process; i++) {
130         inputSignals[0]->bufferGet(&input);
131         currentCopy.at(i) = { input, 0};
132     }

135     vector<t_complex> pcinitialize(process);
136     if (K == 0)
137     {

```

```

139     if (symmetryTypeTf == "Symmetric")
140         previousCopy = pcinitialize;
141
142     else
143         previousCopy = currentCopy;
144     }
145
146     if (previousCopy.size() != currentCopy.size())
147     {
148         vector<t_complex> currentCopyAux;
149         while (currentCopyAux.size() <= previousCopy.size())
150         {
151             for (int i = 0; i < currentCopy.size(); i++)
152             {
153                 currentCopyAux.push_back(currentCopy[i]);
154             }
155         }
156
157         vector<t_complex> cc(previousCopy.size());
158         for (int i = 0; i < previousCopy.size(); i++)
159         {
160             cc[i] = currentCopyAux[i];
161         }
162
163         currentCopy = cc;
164     }
165
166     vector<t_complex> out;
167     out = overlapSaveTransferFunction(currentCopy, previousCopy, ifftshift(transferFunction));
168
169 // Bufferput
170 for (int i = 0; i < process; i++) {
171     t_real val;
172     val = out[i].real();
173     outputSignals[0]->bufferPut((t_real)(val));
174 }
175
176 K = K + 1;
177 previousCopy = currentCopy;
178
179 return true;
}

```

Listing 9.5: filter\_20180306.cpp code

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

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

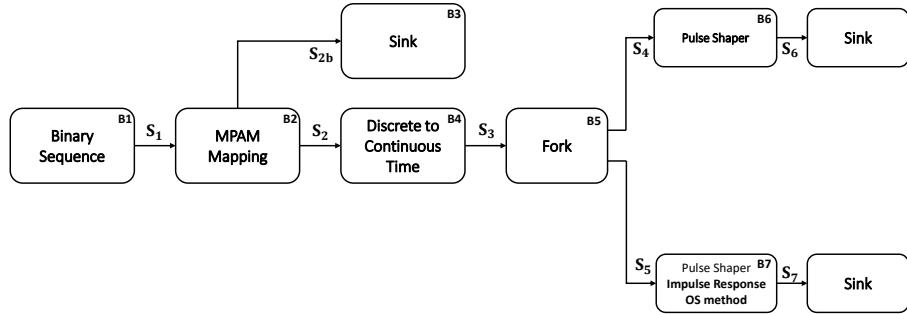


Figure 9.26: Real-time overlap-save example setup

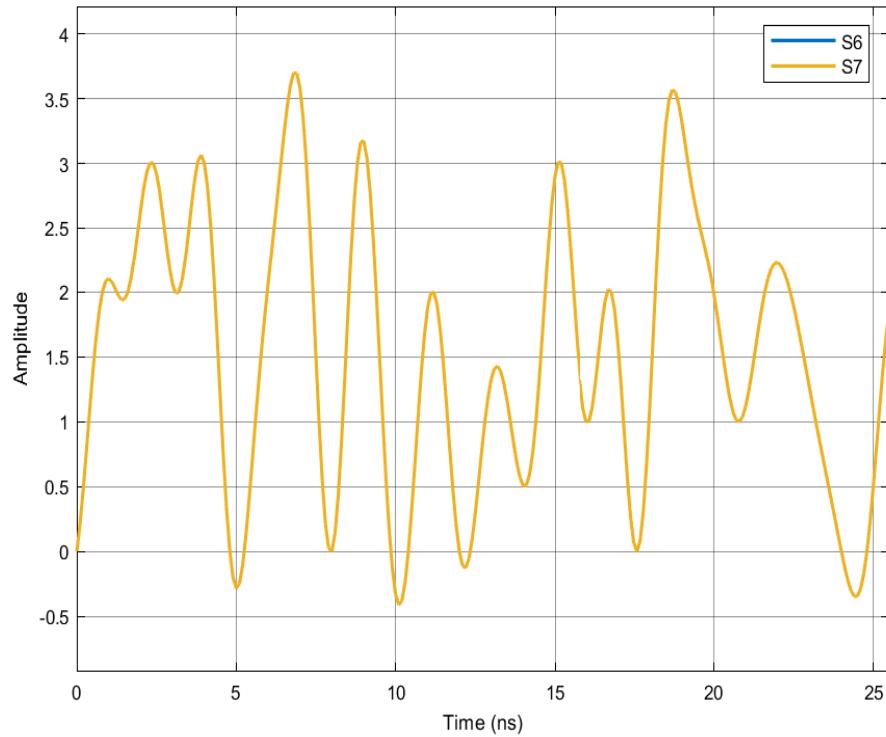


Figure 9.27: Comparison of signal S6 and S7

### Overlap-save using transfer function

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

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

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

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

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

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

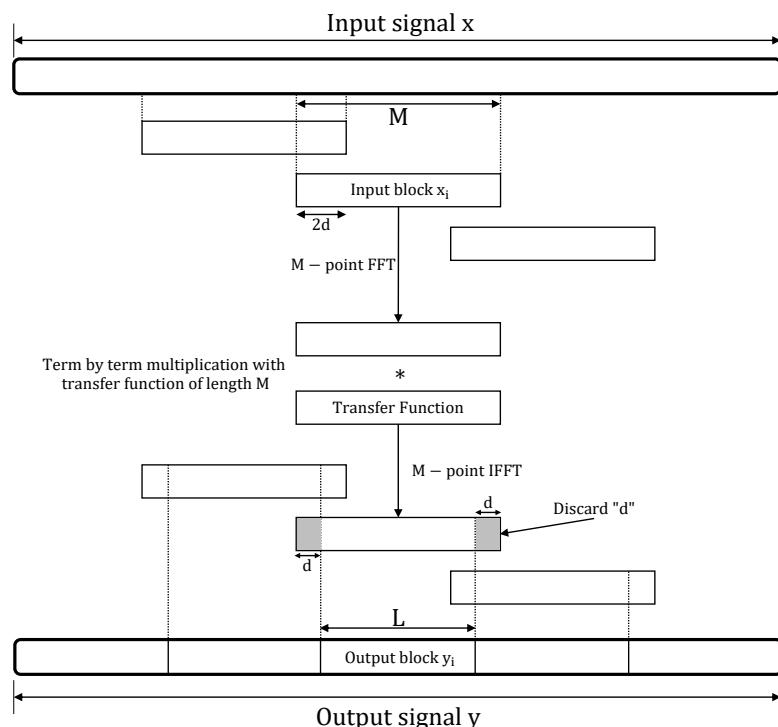


Figure 9.28: Process of the overlap save using transfer function

### Function description

The structure of the new function is as follows,

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

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

must be greater than the length of  $tf(k)$ .

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

This section explains the steps of comparing the result of impulse response and transfer function based overlap-save methods.

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

**Step 2 :** Find the **overlapSaveRealTimeIrTf\_test.vcxproj** file and open it.

In this project file, find *filter\_20180306.cpp* in *SourceFiles* section and click on it. This file includes the definitions of two different filter class namely **FD\_Filter** and **FD\_Filter\_20181110** for applying overlap-save method using impulse response and transfer function, respectively.

```

1 ///////////////////////////////////////////////////////////////////
2 ///////////////////////////////////////////////////////////////////
3 /////////////////////////////////////////////////////////////////// FD_Filter ///////////////////////////////////////////////////////////////////
4 ///////////////////////////////////////////////////////////////////
5 void FD_Filter:: initializeFD_Filter (void)
{
7     outputSignals[0] ->symbolPeriod = inputSignals[0] ->symbolPeriod;
9     outputSignals[0] ->samplingPeriod = inputSignals[0] ->samplingPeriod;
9     outputSignals[0] ->samplesPerSymbol = inputSignals[0] ->samplesPerSymbol;

11    if (!getSeeBeginningOfImpulseResponse()) {
12        int aux = (int) (((double)impulseResponseLength) / 2) + 1;
13        outputSignals[0] ->setFirstValueToBeSaved(aux);
14    }
15
16    if (saveImpulseResponse) {
17        ofstream fileHandler("./signals/" + impulseResponseFilename, ios::out);
18        fileHandler << "// ### HEADER TERMINATOR ###\n";
19
20        t_real t;
21        double samplingPeriod = inputSignals[0] ->samplingPeriod;
22        for (int i = 0; i < impulseResponseLength; i++) {
23            t = -impulseResponseLength / 2 * samplingPeriod + i * samplingPeriod;
24            fileHandler << t << " " << impulseResponse[i] << "\n";
25        }
26        fileHandler.close();
27    }
28}
29
30 bool FD_Filter::runBlock(void)
31 {
32     bool alive{ false };
33
34     int ready = inputSignals[0] ->ready();
35     int space = outputSignals[0] ->space();

```



```

87 //////////////////////////////////////////////////////////////////
88 void FD_Filter_20181110::initializeFD_Filter_20181110(void)
89 {
90     outputSignals[0]->symbolPeriod = inputSignals[0]->symbolPeriod;
91     outputSignals[0]->samplingPeriod = inputSignals[0]->samplingPeriod;
92     outputSignals[0]->samplesPerSymbol = inputSignals[0]->samplesPerSymbol;
93
94     if (!getSeeBeginningOfTransferFunction()) {
95         int aux = (int) ((double)transferFunctionLength) / 4 + 1;
96         outputSignals[0]->setFirstValueToBeSaved(aux);
97     }
98
99     if (saveTransferFunction)
100    {
101        ofstream fileHandler("./signals/" + transferFunctionFilename, ios::out);
102        fileHandler << "// ### HEADER TERMINATOR ###\n";
103
104        double samplingPeriod = inputSignals[0]->samplingPeriod;
105        t_real fWindow = 1 / samplingPeriod;
106        t_real df = fWindow / transferFunction.size();
107
108        t_real f;
109        for (int k = 0; k < transferFunction.size(); k++)
110        {
111            f = -fWindow / 2 + k * df;
112            fileHandler << f << " " << transferFunction[k] << "\n";
113        }
114        fileHandler.close();
115    }
116
117 bool FD_Filter_20181110::runBlock(void)
118 {
119     bool alive{ false };
120
121     int ready = inputSignals[0]->ready();
122     int space = outputSignals[0]->space();
123     int process = min(ready, space);
124     if (process == 0) return false;
125
126 ////////////////////////////////////////////////////////////////// currentCopy //////////////////////////////////////////////////////////////////
127 //////////////////////////////////////////////////////////////////
128 vector<t_complex> currentCopy(process); // Get the Input signal
129 t_real input;
130 for (int i = 0; i < process; i++) {
131     inputSignals[0]->bufferGet(&input);
132     currentCopy.at(i) = { input, 0};
133 }
134
135 vector<t_complex> pcinitialize(process);
136 if (K == 0)
137 {

```

```

139     if (symmetryTypeTf == "Symmetric")
140         previousCopy = pcinitialize;
141
142     else
143         previousCopy = currentCopy;
144     }
145
146     if (previousCopy.size() != currentCopy.size())
147     {
148         vector<t_complex> currentCopyAux;
149         while (currentCopyAux.size() <= previousCopy.size())
150         {
151             for (int i = 0; i < currentCopy.size(); i++)
152             {
153                 currentCopyAux.push_back(currentCopy[i]);
154             }
155         }
156
157         vector<t_complex> cc(previousCopy.size());
158         for (int i = 0; i < previousCopy.size(); i++)
159         {
160             cc[i] = currentCopyAux[i];
161         }
162
163         currentCopy = cc;
164     }
165
166     vector<t_complex> out;
167     out = overlapSaveTransferFunction(currentCopy, previousCopy, ifftshift(transferFunction));
168
169 // Bufferput
170 for (int i = 0; i < process; i++) {
171     t_real val;
172     val = out[i].real();
173     outputSignals[0]->bufferPut((t_real)(val));
174 }
175
176 K = K + 1;
177 previousCopy = currentCopy;
178
179 return true;
180 }
```

Listing 9.6: filter\_20180306.cpp code

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

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

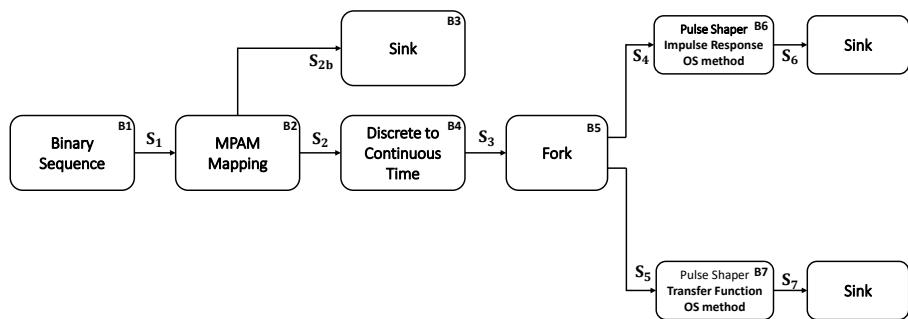


Figure 9.29: Real-time overlap-save example setup

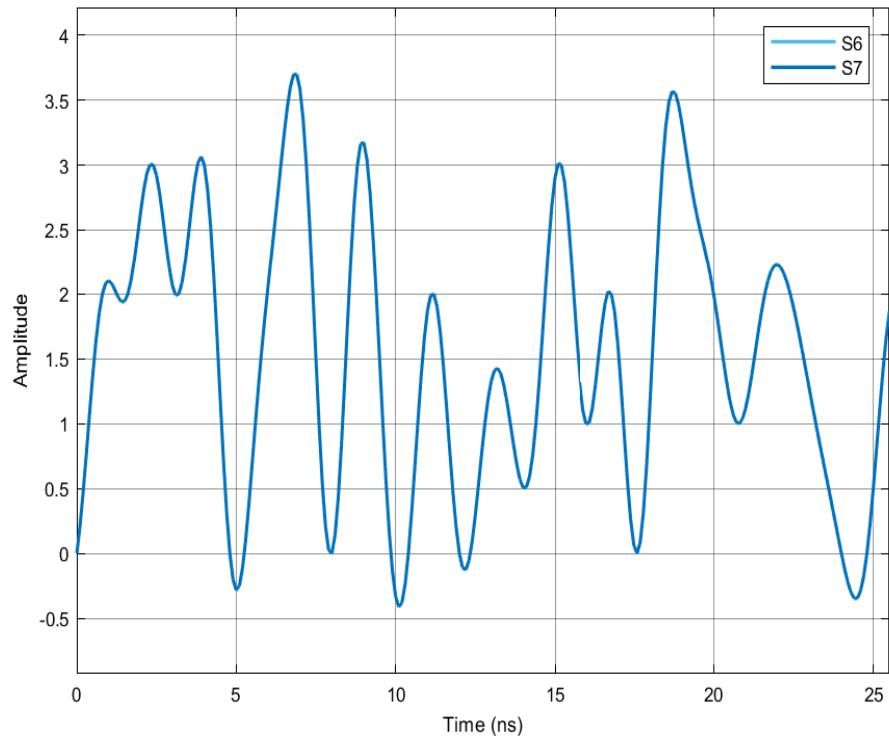


Figure 9.30: Comparison of signal S6 and S7

## References

- [1] Steven W. Smith. *The scientist and engineer's guide to digital signal processing*. California Technical Pub, 1999. ISBN: 0966017676.
- [2] Richard E. Blahut and Richard E. *Fast algorithms for digital signal processing*. Addison-Wesley Pub. Co, 1985, p. 441. ISBN: 0201101556. URL: <https://dl.acm.org/citation.cfm?id=537283>.

### 9.3 Filter

|                    |   |                        |
|--------------------|---|------------------------|
| <b>Header File</b> | : | filter_*.h             |
| <b>Source File</b> | : | filter_*.cpp           |
| <b>Version</b>     | : | 20180201 (Romil Patel) |

In order to filter any signal, a new generalized version of the filter namely *filter\_\*.h* & *filter\_\*.cpp* is programmed which facilitate to filtering in both time and frequency domain. Basically, *filter\_\*.h* file contains the declaration three distinct class namely **FIR\_Filter**, **FD\_Filter**, and **FD\_Filter\_20181110** which facilitate filtering operation in time-domain (using impulse response), overlap-save (using impulse response), and overlap-save (using transfer function), respectively (see Figure 9.31). The *filter\_\*.cpp* file contains the definitions of all the functions declared in the **FIR\_Filter**, **FD\_Filter** and **FD\_Filter\_20181110**.

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

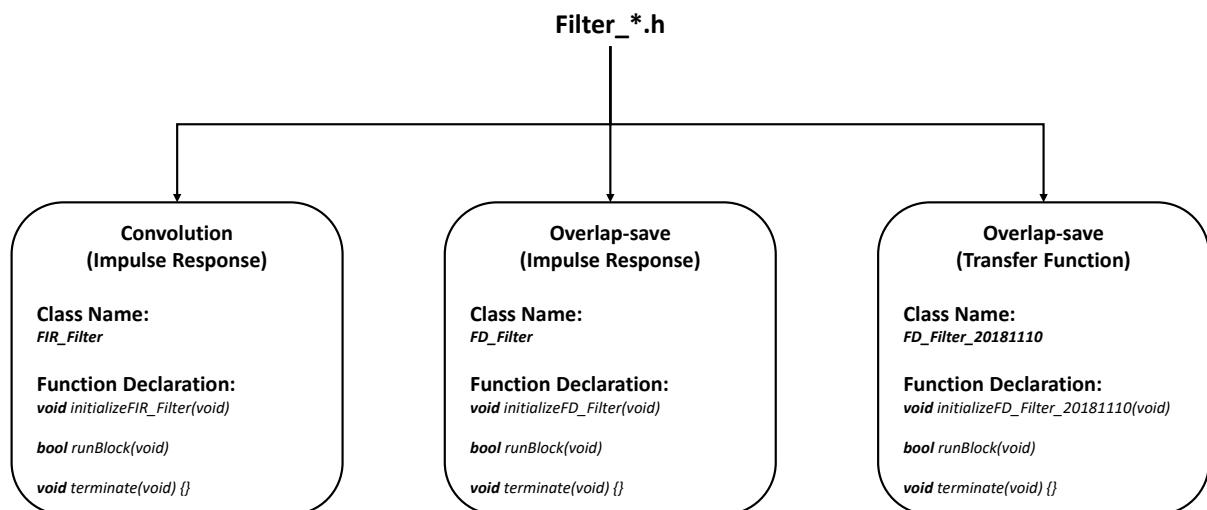


Figure 9.31: Filter class

**FIR\_Filter**, **FD\_Filter** and **FD\_Filter\_20181110** represent the definition of time domain convolution filtering (using impulse response), frequency domain overlap-save filtering (using impulse response), and frequency-domain overlap-save filtering (using transfer function), respectively. [1, 2, 3].

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

```

1 bool FIR_Filter :: runBlock(void) {
2
3     int ready = inputSignals[0] ->ready();
4     int space = outputSignals[0] ->space();
5     int process = min(ready, space);
6     if (process == 0) return false;
7
8     for (int i = 0; i < process; i++) {
9         int in = inputSignals[0] ->get();
10        int out = filter(in);
11        outputSignals[0] ->put(out);
12    }
13
14    return true;
15}

```

```

7   for (int i = 0; i < process; i++) {
9     t_real val;
10    (inputSignals[0])->bufferGet(&val);
11    if (val != 0) {
12      vector<t_real> aux(impulseResponseLength, 0.0);
13      transform(impulseResponse.begin(), impulseResponse.end(), aux.begin(), bind1st(multiplies<t_real>(),
14        val));
15      transform(aux.begin(), aux.end(), delayLine.begin(), delayLine.begin(), plus<t_real>());
16    }
17    outputSignals[0]->bufferPut((t_real)(delayLine[0]));
18    rotate(delayLine.begin(), delayLine.begin() + 1, delayLine.end());
19    delayLine[impulseResponseLength - 1] = 0.0;
20  }
21
22  return true;
23}

```

Listing 9.7: Definition of **bool FIR\_Filter::runBlock(void)**

```

1  bool FD_Filter :: runBlock(void)
2  {
3    bool alive{ false };
4
5    int ready = inputSignals[0]->ready();
6    int space = outputSignals[0]->space();
7    int process = min(ready, space);
8    if (process == 0) return false;
9
10   ////////////////////////////// currentCopy //////////////////////////////
11   ////////////////////////////// currentCopy //////////////////////////////
12   vector<t_complex> currentCopy(process); // Get the Input signal
13   t_real input;
14   for (int i = 0; i < process; i++){
15     inputSignals[0]->bufferGet(&input);
16     currentCopy.at(i) = { input,0 };
17   }
18
19   ////////////////////////////// Impulse response /////////////////////
20   ////////////////////////////// Impulse response /////////////////////
21   vector<t_complex> impulseResponseComplex(impulseResponse.size());
22   vector<t_real> irImag(impulseResponse.size());
23
24   impulseResponseComplex = reImVect2ComplexVector(impulseResponse, irImag);
25
26   ////////////////////////////// OverlapSave function ///////////////////
27   ////////////////////////////// OverlapSave function ///////////////////
28   vector<t_complex> pcinitialize(process);
29   if (K == 0)
30   {
31     if (symmetryTypeIr == "Symmetric")
32       previousCopy = pcinitialize;

```

```

34     else
35         previousCopy = currentCopy;
36     }
37
38     vector<t_complex> OUT = overlapSaveImpulseResponse(currentCopy, previousCopy,
39             impulseResponseComplex);
40
41     previousCopy = currentCopy;
42     K = K + 1;
43
44     // Bufferput
45     for (int i = 0; i < process; i++){
46         t_real val;
47         val = OUT[i].real();
48         outputSignals[0]→bufferPut((t_real)(val));
49     }
50
51     return true;
52 }
```

Listing 9.8: Definition of **bool FD\_Filter::runBlock(void)**

```

1
2     bool alive{ false };
3
4     int ready = inputSignals[0]→ready();
5     int space = outputSignals[0]→space();
6     int process = min(ready, space);
7     if (process == 0) return false;
8
9     /////////////////////////////// currentCopy ///////////////////////////////
10    /////////////////////////////// currentCopy ///////////////////////////////
11    vector<t_complex> currentCopy(process); // Get the Input signal
12    t_real input;
13    for (int i = 0; i < process; i++) {
14        inputSignals[0]→bufferGet(&input);
15        currentCopy.at(i) = { input, 0};
16    }
17
18    vector<t_complex> pcinitialize(process);
19    if (K == 0)
20    {
21        if (symmetryTypeTf == "Symmetric")
22            previousCopy = pcinitialize;
23
24        else
25            previousCopy = currentCopy;
26    }
27
28    if (previousCopy.size() != currentCopy.size())
29    {
```

```

30    vector<t_complex> currentCopyAux;
31    while (currentCopyAux.size() <= previousCopy.size())
32    {
33        for (int i = 0; i < currentCopy.size(); i++)
34        {
35            currentCopyAux.push_back(currentCopy[i]);
36        }
37    }
38
39    vector<t_complex> cc(previousCopy.size());
40    for (int i = 0; i < previousCopy.size(); i++)
41    {
42        cc[i] = currentCopyAux[i];
43    }
44
45    currentCopy = cc;
46}
47
48 vector<t_complex> out;
49 out = overlapSaveTransferFunction(currentCopy,previousCopy,ifftshift(transferFunction));
50
51 // Bufferput
52 for (int i = 0; i < process; i++) {
53     t_real val;
54     val = out[i].real();
55     outputSignals[0]->bufferPut((t_real)(val));
56 }
57
58 K = K + 1;
59 previousCopy = currentCopy;
60
61 return true;
62}

```

Listing 9.9: Definition of **bool FD\_Filter\_20181110::runBlock(void)**

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

### Example of pulse shaping filtering

This section explains how to use **FIR\_Filter**, **FD\_Filter** and **FD\_Filter\_20181110** classes can be used for the filtering operation with the help of impulse response and transfer function of the pulse shaping filter. The impulse response for the both **FIR\_Filter** and **FD\_Filter** classes will be generated by a *pulse\_shaper.cpp* and *pulse\_shaper\_20180306.cpp* files respectively and

applied to the **bool runblock(void)** block as shown in Figure 9.32. The transfer function for the class FD\_Filter\_20181110 will be generated by *pulse\_shaper\_20181110.cpp* and applied to the **bool runblock(void)** as shown in Figure 9.32.

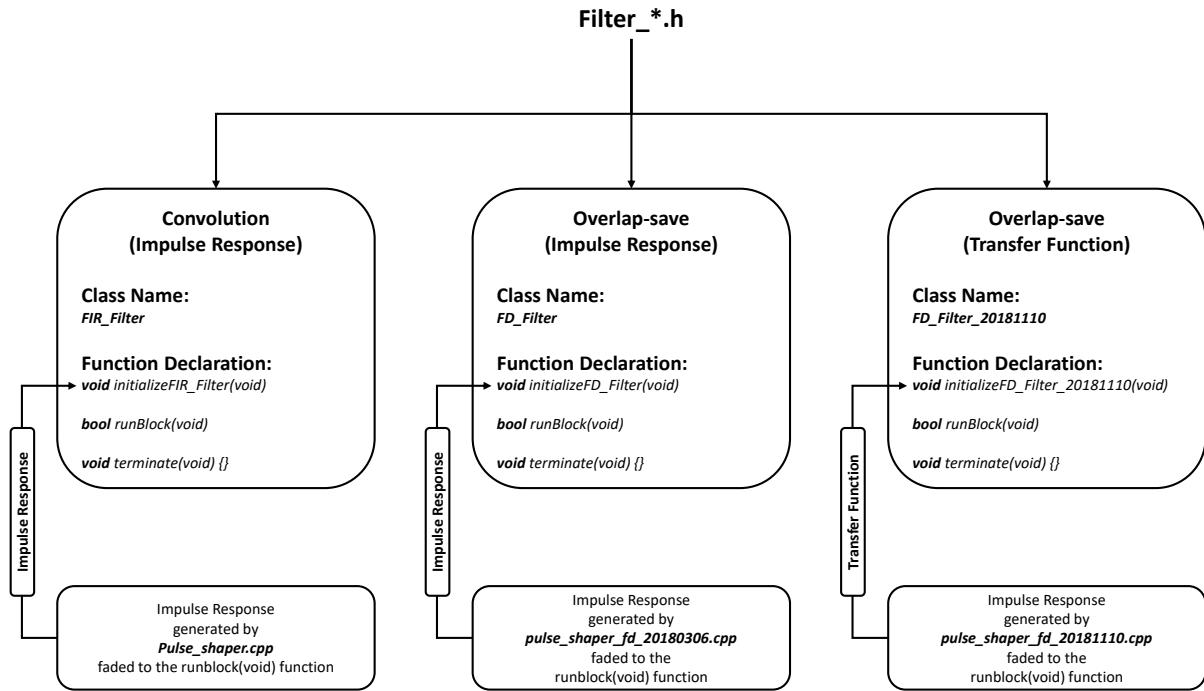


Figure 9.32: Pulse shaping using `filter_*.h`

### Example of pulse shaping filtering : Procedural steps

This section explains the steps of pulse shaping filtering with the help of convolution and overlap-save method using its impulse response. It also displays the comparison between the resultant output generated by both the methods. In order to conduct the experiment, follow the steps given below:

**Step 1 :** In the directory, open the folder namely `filter_test` by following the path `"/algorithms/filter/filter_test"`.

**Step 2 :** Find the `filter_test.vcxproj` file in the same folder and open it.

In this project file, find `filter_test.cpp` in *SourceFiles* section and click on it. This file represents the simulation set-up as shown in Figure 9.33.

**Step 3 :** Check how **PulseShaper** and **PulseShaperFd** blocks are implemented.

Check the appendix for the various types of pulse shaping techniques and what are the different parameters used to adjust the shape of the pulse shaper.

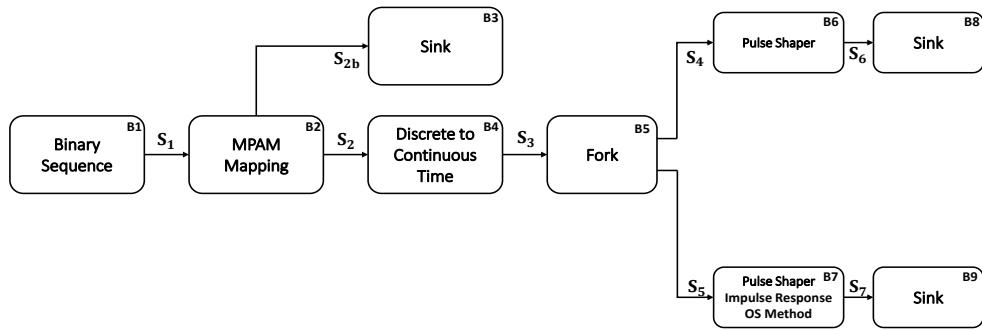


Figure 9.33: Filter test setup

**Step 4 :** Run the *filter\_test.cpp* code and compare the signals **S6.sgn** and **S7.sgn** using visualizer.

Here, we have used three different types of pulse shaping filter namely, raised cosine, root raised cosine and Gaussian pulse shaper. The following Figure 9.34, 9.35 and 9.36 display the comparison of the output signals **S6.sgn** and **S7.sgn** for the raised cosine, root raised cosine and Gaussian pulse shaping filter, respectively. Similarly, we can verify the result of the transfer function based overlap-save method.

### Case 1 : Raised cosine

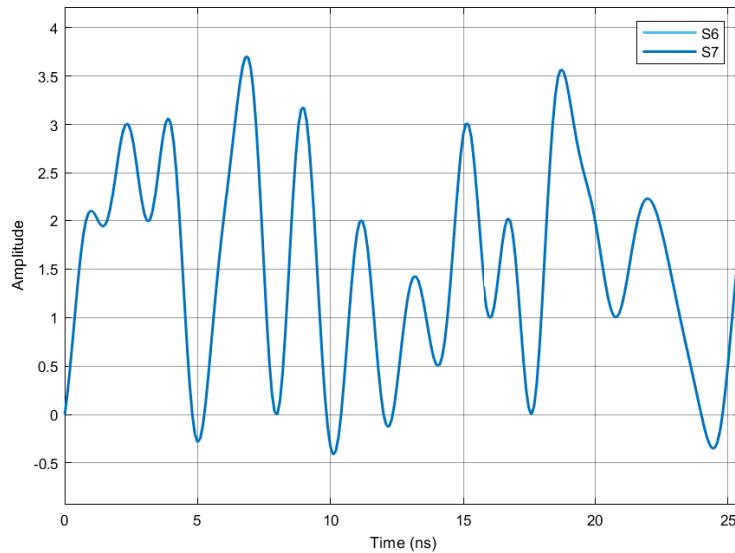


Figure 9.34: Raised cosine pulse shaping results comparison

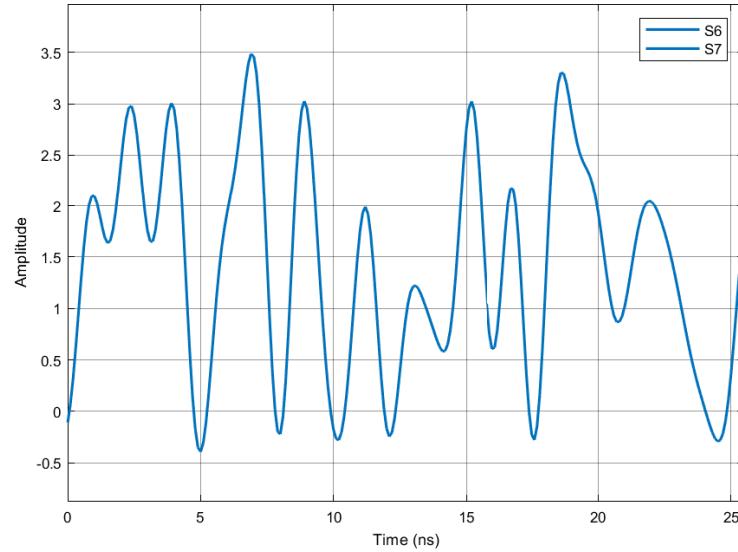
**Case 2 : Root raised cosine**

Figure 9.35: Root raised cosine pulse shaping result

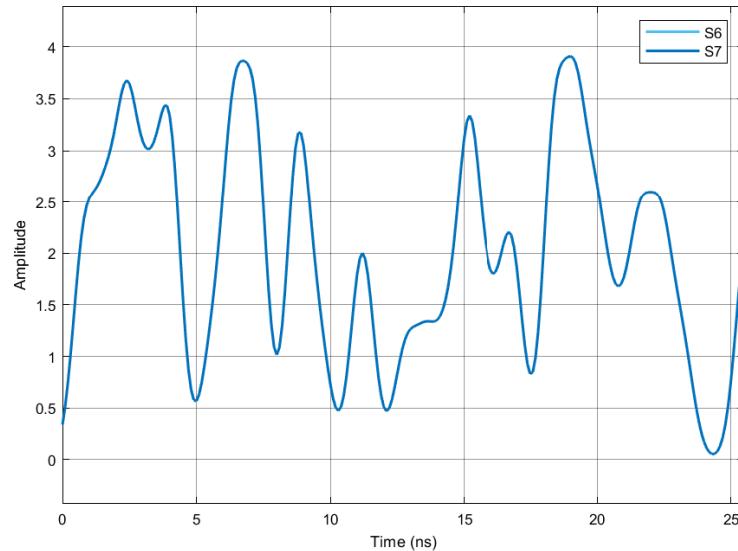
**Case 3 : Gaussian**

Figure 9.36: Gaussian pulse shaping results comparison

## APPENDICES

### A. Raised cosine pulse shaper

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

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

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

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

### B. Root raised cosine pulse shaper

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

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

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

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

### C. Gaussian pulse shaper

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

$$H_G(f) = \exp(-\alpha^2 f^2) \quad (9.11)$$

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

$$\alpha = \frac{\sqrt{\ln 2}}{\sqrt{2}B} = \frac{0.5887}{B} \quad (9.12)$$

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

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

From the equation 9.12, we can also write that,

$$\alpha = \frac{0.5887}{BT_s} T_s \quad (9.14)$$

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

## References

- [1] Sen M. (Sen-Maw) Kuo, Bob H. Lee, and Wenshun. Tian. *Real-time digital signal processing : fundamentals, implementations and applications*. ISBN: 9781118414323. URL: <https://www.wiley.com/en-us/Real+Time+Digital+Signal+Processing%7B%5C%7D3A+Fundamentals%7B%5C%7D2C+Implementations+and+Applications%7B%5C%7D2C+3rd+Edition-p-9781118414323>.
- [2] Theodore S. Rappaport. *Wireless communications : principles and practice*. Prentice Hall PTR, 2002, p. 707. ISBN: 0130422320.
- [3] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time signal processing*. Prentice Hall, 1999, p. 870. ISBN: 0137549202. URL: <https://dl.acm.org/citation.cfm?id=294797>.

## 9.4 Hilbert Transform

|                    |   |                        |
|--------------------|---|------------------------|
| <b>Header File</b> | : | hilbert_filter_*.h     |
| <b>Source File</b> | : | hilbert_filter_*.cpp   |
| <b>Version</b>     | : | 20180306 (Romil Patel) |

### What is the purpose of Hilbert transform?

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

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

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

### Transfer function for the discrete Hilbert transform

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

#### Method 1 :

The discrete time analytical signal  $S_a(t)$  corresponding to  $s(t)$  is defined in the frequency domain as [1] (This method requires MATLAB Hilbert transform definition)

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

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

#### Method 2 :

The discrete time Hilbert transformed signal  $\hat{s}(f)$  corresponding to  $s(f)$  is defined in the frequency domain as [2]

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

which is inverse transformed to obtain a Hilbert transformed signal  $\hat{S}(t)$ . To generate an analytical signal,  $\hat{S}(t)$  is added to the  $S(t)$  to get the equation 9.15.

### Real-time Hilbert transform : Proposed logical flow

To understand the new proposed method, consider that the signal consists of 2048 samples and the **bufferLength** is 512. Therefore, by considering the **bufferLength**, we will process the whole signal in four consecutive blocks namely *A*, *B*, *C* and *D*; each with the length of 512 samples as shown in Figure 9.37. The filtering process will start only after acquiring first

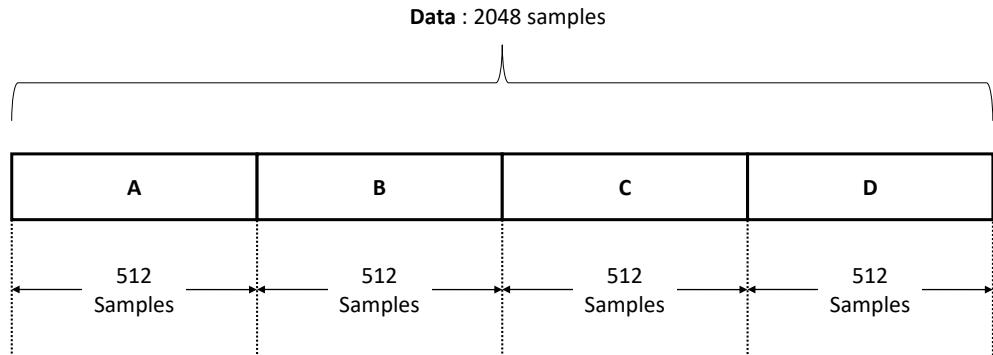


Figure 9.37: Logical flow

two blocks *A* and *B* (see iteration 1 in Figure 9.38), which introduces delay in the system. In the iteration 1,  $x(n)$  consists of 512 front Zeros, block *A* and block *B* which makes the total length of the  $x(n)$  is  $512 \times 3 = 1536$  symbols. After applying filter to the  $x(n)$ , we will capture the data which corresponds to the block *A* only and discard the remaining data from each side of the filtered output.

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

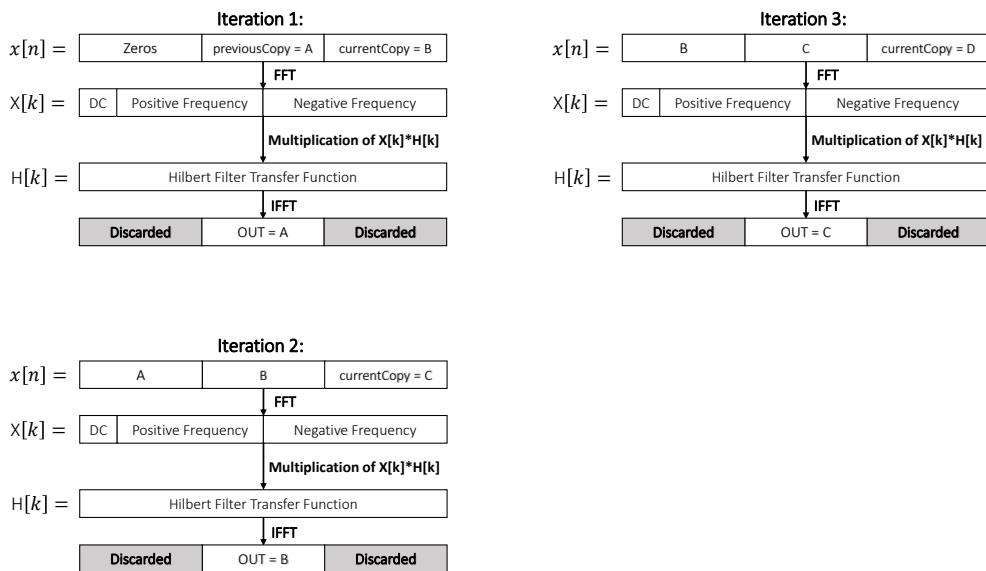


Figure 9.38: Logical flow of real-time Hilbert transform

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

### Real-time Hilbert transform : Test setup

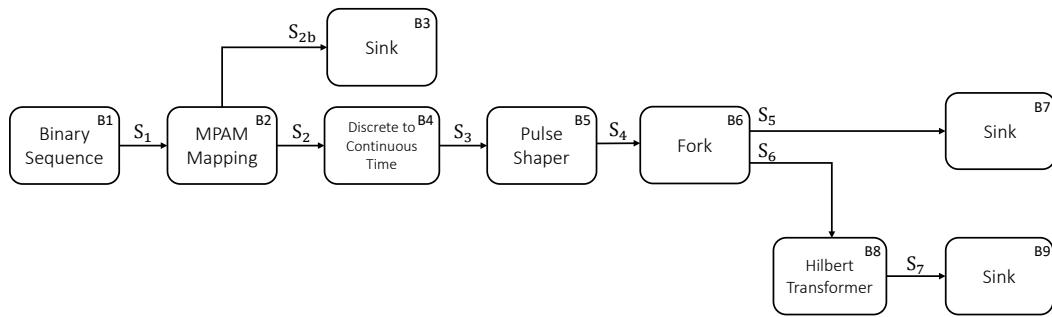


Figure 9.39: Test setup for the real time Hilbert transform

### Real-time Hilbert transform : Results

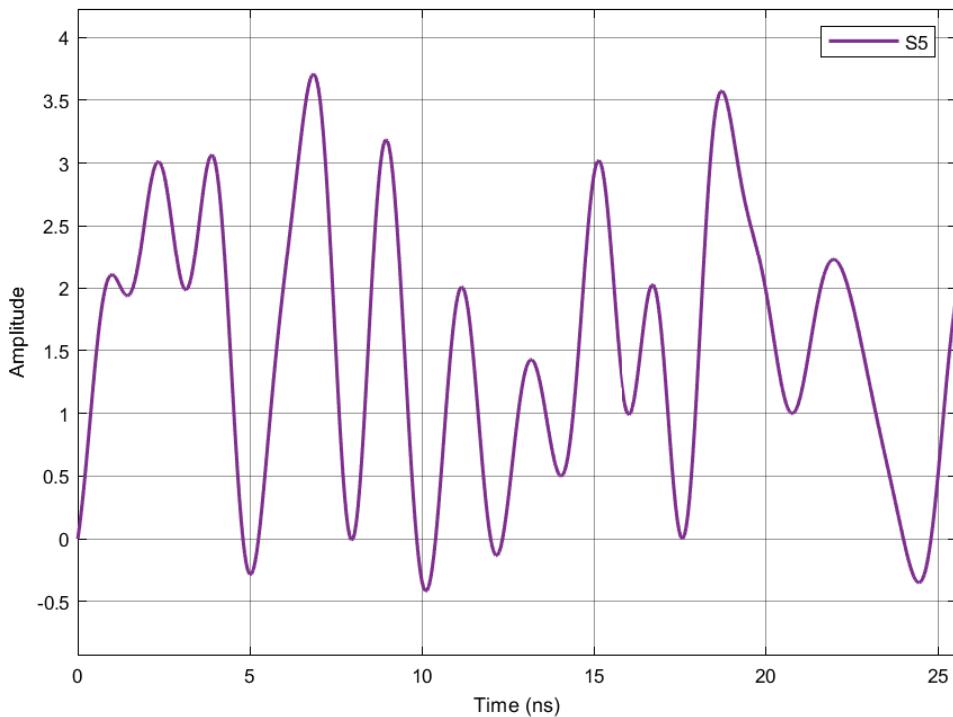


Figure 9.40:  $S_5$  signal

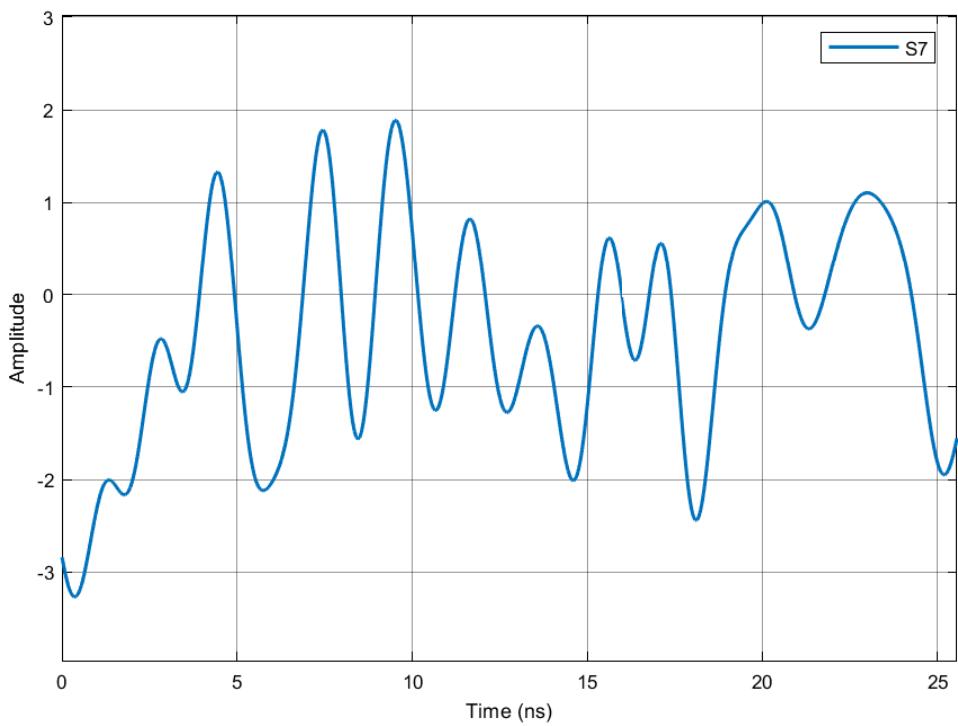


Figure 9.41:  $S7$  signal

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

## References

- [1] S.L. Marple. "Computing the discrete-time 'analytic' signal via FFT". In: *Conference Record of the Thirty-First Asilomar Conference on Signals, Systems and Computers (Cat. No.97CB36136)*. Vol. 2. IEEE Comput. Soc, pp. 1322–1325. ISBN: 0-8186-8316-3. DOI: [10.1109/ACSSC.1997.679118](https://doi.org/10.1109/ACSSC.1997.679118). URL: <http://ieeexplore.ieee.org/document/679118/>.
- [2] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time signal processing*. Prentice Hall, 1999, p. 870. ISBN: 0137549202. URL: <https://dl.acm.org/citation.cfm?id=294797>.

## **Chapter 10**

# **Code Development Guidelines**

[github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md](https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md)

### **10.0.1 Integrated Development Environment**

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

Visual Studio Community 2017 - version 15.7.6

### **10.0.2 Compiler Switches**

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

## Chapter 11

# Building C++ Projects Without Visual Studio

---

This is a guide on how to build C++ projects without having Microsoft Visual Studio installed. All the necessary files will be available in the \msbuild\ folder on this repository.

## 11.1 Installing Microsoft Visual C++ Build Tools

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

## 11.2 Adding Path To System Variables

Please follow this step-by-step tutorial carefully.

1. Open the **Control Panel**.
2. Select the option **System and Security**.
3. Select the option **System**.
4. Select **Advanced System Settings** on the menu on the left side of the window.
5. This should have opened another window. Click on **Environment variables**.
6. Check if there is a variable called **Path** in the **System Variables** (bottom list).
7. If it doesn't exist, create a new variable by pressing **New** in **System Variables** (bottom list). Insert the name **Path** as the name of the variable and enter the following value `C:\Windows\Microsoft.Net\Framework\v4.0.30319`. Jump to step 10.
8. If it exists, click on the variable **Path** and press **Edit**. This should open another window;
9. Click on **New** to add another value to this variable. Enter the following value: `C:\Windows\Microsoft.Net\Framework\v4.0.30319`.
10. Press **Ok** and you're done.

## 11.3 How To Use MSBuild To Build Your Projects

You are now able to build (compile and link) your C++ projects without having Visual Studio installed on your machine. To do this, please follow the instructions below:

1. Open the **Command Line** and navigate to your project folder (where the .vcxproj file is located).
2. Enter the command:  
`msbuild <filename> /tv:14.0 /p:PlatformToolset=v140,TargetPlatformVersion=8.1,OutDir=".\"`, where <filename> is your .vcxproj file.

After building the project, the .exe file should be automatically generated to the current folder.

The signals will be generated into the sub-directory \signals\, which must already exist.

## 11.4 Known Issues

### 11.4.1 Missing ucrtbased.dll

In order to solve this issue, please follow the instructions below:

1. Navigate to **C:\Program Files (x86)\Windows Kits\10\bin\x86\ucrt\**
2. Copy the following file: **ucrtbased.dll**
3. Paste this file in the following folder: **C:\Windows\System32\**
4. Paste this file in the following folder: **C:\Windows\SysWOW64\**

**Attention:**you need to paste the file in BOTH of the folders above.

### 12.1 Starting with Git

Git is a free and open source distributed version control system [1]. Git creates and maintains a database that records all changes that occur in a folder. The Git database is named a repository. It also allows to merge repositories that shared a common state in the past. These can be local repositories, i.e. stored in the same machine, or can be remote repositories, i.e. stored anywhere.

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

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

```
git init
```

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

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

### 12.2 Data Model

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

Git manipulates the following objects:

- commits - text files that store a description of the repository;
- trees - text files that store a description of a folder;
- blobs - the files that exist in your repository;
- tags - text files that store information about commits.

The objects are stored in the folder `.git/objects`. Each stored object is identified by its SHA1 hash value, i.e. 20 bytes which identifies unequivocally the object. The SHA1 is just an algorithm

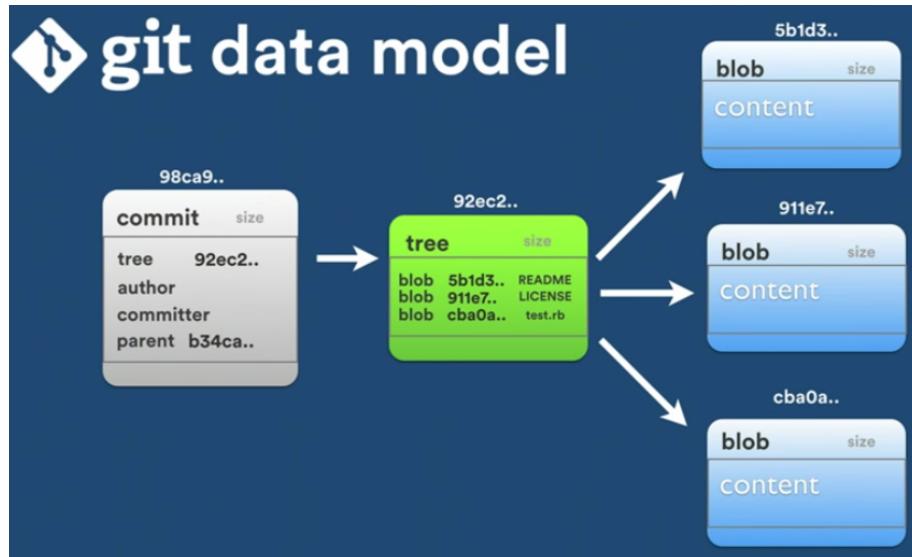


Figure 12.1: Git data model.

that accepts some binary information and generates 20 bytes which are ideally unique for that information. The probability of collisions is extremely low, i.e. the probability that different information generates the same hash value is extremely low. Note that 20 bytes can be represented by a 40 characters hexadecimal string. The identifier of each object is the 40 characters hexadecimal string. Each particular object is stored in a sub-folder inside the `.git/objects`. The name of the sub-folder is the two most significative characters of the SHA1 hash value. The name of the file that is inside the sub-folder is the remanning thirty eight characters of the SHA1 hash value. The Git stores all committed versions of a file. The Git maintains a contend-addressable file systems, i.e. a file system in which the files can be accessed based on its contend. Lets look more carefully in each stored object.

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

```

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

Here goes the commit message.

A **tree** object is identified by a SHA1 hash value, and has a list of blobs and trees that

are inside that tree. A tree object identifies a folder and its contend. Example of a tree file contend:

|        |      |   |                  |
|--------|------|---|------------------|
| 100644 | blob | bdb0cabc87cf50106df6e15097dff816c8c3eb34  | .gitattributes   |
| 100644 | blob | 50492188dc6e12112a42de3e691246dafdad645b  | .gitignore       |
| 100644 | blob | 8f564c4b3e95add1a43e839de8adbfd1ceccf811  | bfg-1.12.16.jar  |
| 040000 | tree | de44b36d96548240d98cb946298f94901b5f5a05  | doc              |
| 040000 | tree | 8b7147dbfdc026c78fee129d9075b0f6b17893be  | garbage          |
| 040000 | tree | bdfcd8ef2786ee5f0f188fc04d9b2c24d00d2e92  | include          |
| 040000 | tree | 040373bd71b8fe2fe08c3a154cada841b3e411fb  | lib              |
| 040000 | tree | 7a5fce17545e55d2faa3fc3ab36e75ed47d7bc02  | msbuild          |
| 040000 | tree | b86efba0767e0fac1a23373aaaf95884a47c495c5 | mtools           |
| 040000 | tree | 1f981ea3a52bccf1cb00d7cb6dfdc687f33242ea  | references       |
| 040000 | tree | 86d462afd7485038cc916b62d7cbfc2a41e8cf47  | sdf              |
| 040000 | tree | 13bfce10b78764b24c1e3dfbd0b10bc6c35f2f7b  | things_to_do     |
| 040000 | tree | 232612b8a5338ea71ab6a583d477d41f17ebae32  | visualizerXPTO   |
| 040000 | tree | 1e5ee96669358032a4a960513d5f5635c7a23a90  | work_in_progress |

A **blob** is identified by a SHA1 hash value, and has the file contend compressed. A git header and tailor is added to each file and the file is compressed using the zlib library. The git header is just the object type, a space character, the file size in bytes and the \NUL caracter, for instance "blob 13\NUL", the tailor is just the \n caracter. The compressed blob (header+file contend+tailer) is stored as a binary file.

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

### 12.2.1 Objects Folder

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

The folder *.git/objects* stores information about all objects (commits, trees, blobs and annotated tags). The objects are stored in files inside folders. The name of the folders are the 2 first characters of the SHA1 40 characters hexadecimal string. The name of the files are the other 38 hexadecimal characters of the SHA1. The information is compressed to save space.

## 12.3 Refs

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

name of the file is the name of the ref and inside the file is a string with the SHA1 hash value. Tags and branches are example of refs. Tags are static references, i.e. tags are never updated, and branches are dynamic references, i.e. branches are always automatically updated.

### 12.3.1 Refs Folder

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

### 12.3.2 Branch

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

### 12.3.3 Heads

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

## 12.4 Git Logical Areas

Git uses several spaces.

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

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

## 12.5 Merge

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

### 12.5.1 Fast-Forward Merge

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

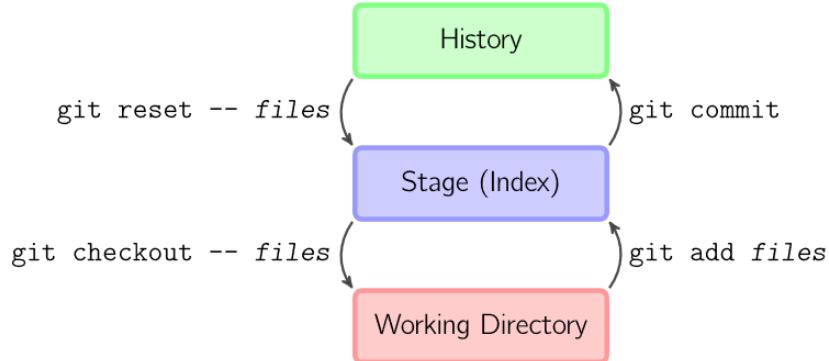


Figure 12.2: Git logical areas. Figure adapted from [2].

### 12.5.2 Three-Way Merge

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

## 12.6 Remotes

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

### 12.6.1 GitHub

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

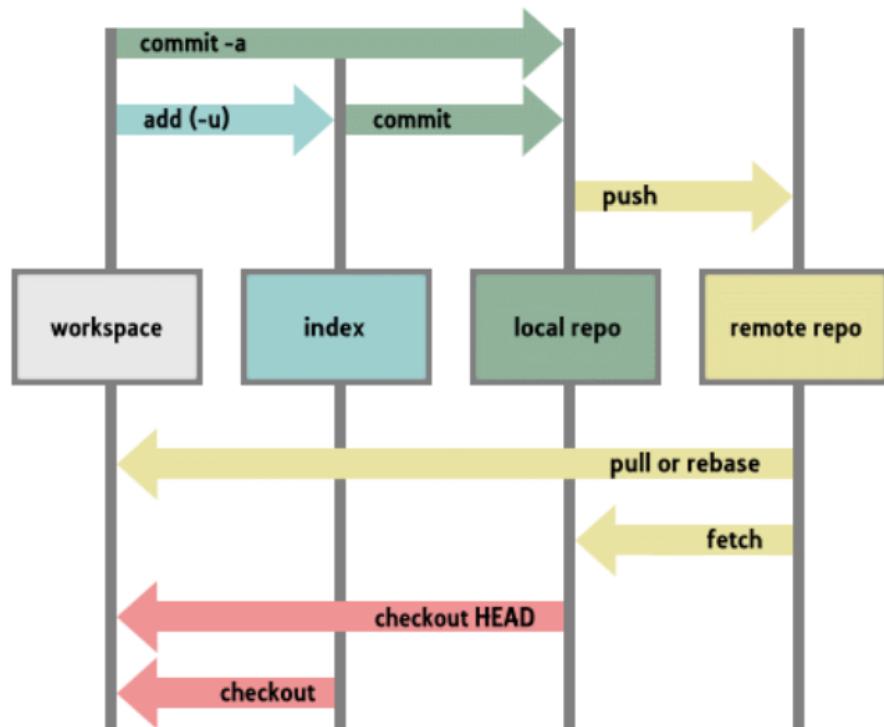


Figure 12.3: Git remotes.

## 12.7 Commands

### 12.7.1 Porcelain Commands

Porcelain commands are high-level commands.

#### git add

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

#### git branch

*git branch*, lists local branches.

*git branch -r*, lists remote branches.

*git branch -a*, lists all branches.

*git branch -u <remote>/<branch>*, links the local branch in which you are in with a remote branch.

*git branch --set-upstream-to=<remote>/<branch>*, longer version of the previous command.

*git branch -u <remote>/<branch> <local\_branch>*, links the local branch with a remote branch.

*git branch --set-upstream-to=<remote>/<branch> <local\_branch>*, longer version of the previous command.

#### git cat-file

*git cat-file -t <hash>*, shows the type of the object identified by the hash value.

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

### **git checkout**

*git checkout -b <new\_branch\_name>*, creates a new branch in the same position as the current branch and move to it.

*git checkout -b <new\_branch\_name> <branch\_name>*, create a new branch in the position of <branch\_name> and move to it.

### **git clean**

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

### **git clone**

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

### **git config**

*git config --global user.name "netxpto"*, sets the user name globally.

*git config --global user.email "netxpto@gmail.com"*, sets the user e-mail globally.

*git config --global user.emmail "netxpto@gmail.com"*, sets the user e-mail globally.

*git config --global alias.<alias name> <commands>*, creates a global alias for the commands.

### **git diff**

*git diff*, shows the changes between the working space and the staging area.

*git diff --name-only*, shows the changes between the working space and the staging area, in the specified files.

*git diff --cached*, shows the changes between the staging area and the current branch history. Note that --cached or --staged are synonymous.

*git diff --cached --name-only*, shows the changes between the staging area and the current branch history, in the specified files.

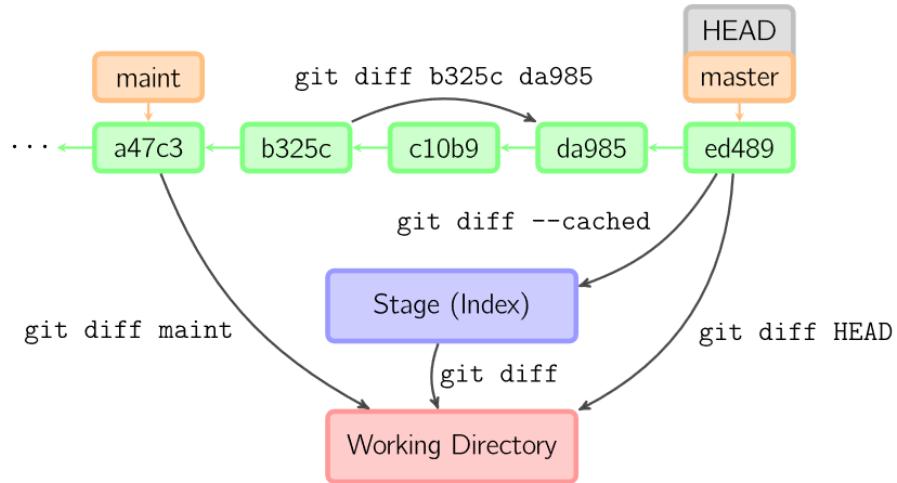


Figure 12.4: Git diff command. Figure adapted from [2].

**git fetch**

*git fetch -all*, downloads all history from all remote repositories.

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

**git init**

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

**git log**

*git log*, shows a list of commits from reverse time order.

*git log --graph --decorate --oneline <commit1>..<commit2>*, shows the history of the repository in a colourful way.

*git log --graph*, shows a graphical representation of the repository commits history.

*git log --stat*, shows the name of the files that were changed in each commit

*git log --follow <file>*, lists version history for a file, including rename.

**git ls-files****git ls-remote****git merge**

*git merge <branch>*, combines the specified branch's history into the current branch.

**git pull**

*git pull*, downloads remote branch history and incorporates changes.

**git push**

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

**git rebase**

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

**git remote**

*git remote*, shows a list of existing remotes.

*git remote -v*, shows the full location of existing remotes.

*git remote add <remote name> <remote repository url>*, adds a remote.

*git remote remove <remote name>*, removes a remote.

**git reset**

*git reset --soft <commit\_hash\_value>*, moves to the commit identified by <commit\_hash\_value> but leaves in the staging area all modified files.

*git reset --hard <commit\_hash\_value>*, moves to the commit identified by <commit\_hash\_value> and cleans all modified tracked files.

*git reset <commit\_hash\_value>*, this is a mix reset (is the default reset), puts all modified files in the working area.

*git reset <file>*, unstages the file, but preserve its contents.

**git reflog**

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

**git rm**

*git rm <file>*, deletes the file from the working directory and stages the deletion.

*git rm --cached <file>*, removes the file from version control but preserves the file locally.

**git show**

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

**git stash**

*git stash*, temporarily stores all modified tracked files.

*git stash -list*, shows what is in the stash.

*git stash pop*, restores the most recently stashed files.

*git stash drop*, discards the most recently stashed changeset.

**git status**

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

### 12.7.2 Pluming Commands

Pluming commands are low-level commands.

**git cat-files**

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

*git cat-files -t <sha1>*, shows the type of a object, i.e. blob, tree or commit.

**git count-object**

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

**git gc**

*git gc*, garbage collector, eliminates all objects that has no reference associated with.

*git gc --prune=all*

**git hash-object**

*git hash-object <file>*, calculates the SHA1 hash value of a file plus a header.

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

**git merge-base**

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

**git update-index**

*git update-index --add <file name>*, creates the hash and adds the <file\_name> to the index.

**git ls-files**

*git ls-files --stage*, shows all files that you are tracking.

**git write-tree****git commit-tree****git rev-parse**

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

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

### **git update-ref**

`git update-ref refs/heads/<branch name> <commit sha1 value>`, creates a branch that points to the `<commit sha1 value>`.

### **git verify-pack**

## 12.8 Navigation Helpers

`<ref>^`, one commit before `<ref>`.

`<ref>^^`, two commits before `<ref>`.

`<ref>~5`, five commits before `<ref>`.

`<ref1>..<ref2>`, between commit `<ref1>` and `<ref2>`.

`<branch>^tree`, identifies the tree pointed by the commit pointed by `<branch>`.

`<commit>:<file_name>`, identifies the version of a file in a given commit.

## 12.9 Configuration Files

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

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

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

### **git config -global -e**

## 12.10 Pack Files

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

## 12.11 Applications

### 12.11.1 Meld

### 12.11.2 GitKraken

## 12.12 Error Messages

### 12.12.1 Large files detected

Clean the repository with the [BFG Repo-Cleaner](#).

Run the Java program:

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

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

```
git push --force.
```

## 12.13 Git with Overleaf

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

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

In this case the repository address is <https://git.overleaf.com/12925162jkwbhrdkwrfm>. The only change was the replacement of **www** by **git**.

Now you can just do

```
git clone https://git.overleaf.com/12925162jkwbhrdkwrfm
```

and clone your repository.

You can also do

```
git push https://git.overleaf.com/12925162jkwbhrdkwrfm
```

---

## Bibliography

- [1] Scott Chacon and Ben Straub. *Pro Git, 2nd Edition*. Apress, 2014.
- [2] Feb. 4, 2019. URL: <https://marklodato.github.io/visual-git-guide>.

Beamer allows the creation of presentations based on the L<sup>A</sup>T<sub>E</sub>Xdocument creation platform. This document contains a short introduction on how to use Beamer, including a tutorial on how to install and use *pympress*.

### 13.1 Intro to Beamer presentations

Beamer presentations use the beamer document class, ie. the document must start with `\documentclass{beamer}`, other than a few other beamer specific options, the preamble is equivalent to any other L<sup>A</sup>T<sub>E</sub>Xdocument.

Slides are declared by the frame environment, all content that is meant to be shown should be declared inside this environment. Slides will appear in the order they are declared.

### 13.2 Adding presenter notes to beamer

Presenter notes are added by the `\note[slide notes]` command. This command is declared outside the frame environment and by default adds notes to the previous frame environment (if necessary this can be altered with the appropriate options). Combined with the itemize option, the note command can be called as

```
\note[itemize]{  
    \item item 1  
    \item item 2  
    ...  
}
```

The notes are not displayed by default, the option `\setbeameroption{show notes on second screen}` should be added in the preamble of the document. The notes will then appear appended to the right of every slide, an example of this can be seen in Figure 13.1. These note slides are flagged, so using an appropriate program (for example *pympress*) it is possible to achieve a presenter view, displaying the notes and slide proper on different windows/screens.

#### 13.2.1 Installing and using *pympress*

Pympress is a multiplatform, python based PDF reader designed specifically for dual-screen presentations, available at <https://github.com/Cimbali/pympress>. Installers for Windows are available at <https://github.com/Cimbali/pympress/releases/>

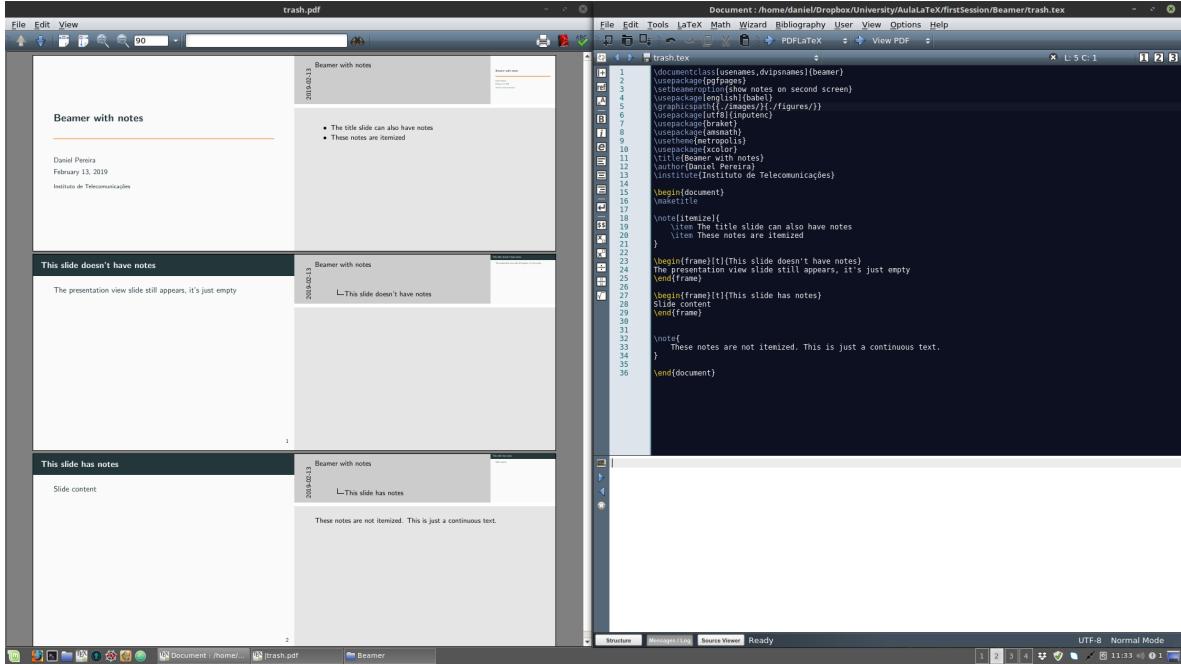


Figure 13.1: Example of a Beamer presentation with presentation notes.

`tag/v1.2.0`, these also deal with all the dependencies and are thus the recommended method for Windows machines.

For non-Windows machines the install process can be more involved and the precise steps may vary from machine to machine. First of all, the following dependencies are required

- Python 2.7 or 3.x, installed from <https://www.python.org> preferably, so as to include the *pip* package manager.
- Poppler
- Gtk+ 3 and the following sub-dependencies
  - Cairo
  - Gdk
- PyGi
- optional: VLC

more information on the installation of the dependencies can be found in <https://github.com/Cimbali/pympress#dependencies>.

After all the the dependencies have been installed, *pympress* is installed through the command

```
pip install pympress
```

On Windows, *pympress.exe* can be used to run the program, from which you can then open the presentation PDF as with a regular PDF reader, this executable should be in `/ProgramFiles/pympress/`. On Linux *pympress* is run from the terminal, either by calling `pympress`

in which case `pympress` is opened without a PDF file, the presentation can then be opened from inside `pympress`. In both of the previous situations, `pympress` opens as in Figure 13.2

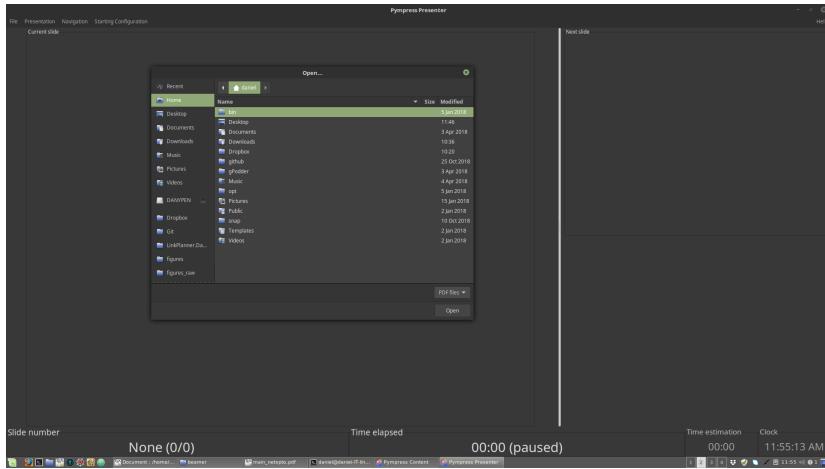


Figure 13.2: Example of *pympress* open without a PDF presentation.

Alternatively, the presentation can be opened directly from the terminal by navigating to the folder with the PDF file and calling

pympress presentationFileName.pdf

The presenter view includes multiple functionalities, available from the menus in the menu bar. An example of *pympress* opened with a presentation can be seen in Figure 13.3.

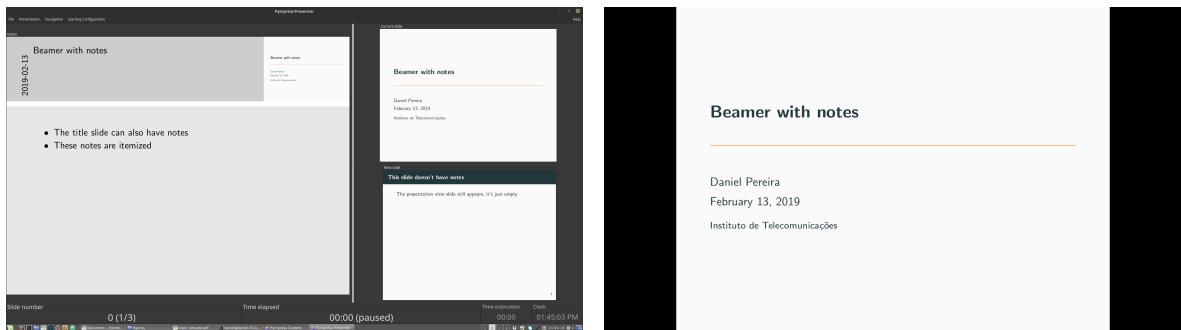


Figure 13.3: Example of *pympress* presenter view (left) and corresponding presentation (right).

### 13.3 IT template

## Chapter 14

# Simulating VHDL Programs with GHDL

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

### 14.1 Adding Path To System Variables

Please follow this step-by-step tutorial:

1. Open the **Control Panel**.
2. Select the option **System and Security**.
3. Select the option **System**.
4. Select **Advanced System Settings** on the menu on the left side of the window.
5. This should have opened another window. Click on **Environment variables**.
6. Check if there is a variable called **Path** in the **System Variables** (bottom list).
7. **If it doesn't exist**, create a new variable by pressing **New** in **System Variables** (bottom list). Insert the name **Path** as the name of the variable and enter your absolute path to the folder `\LinkPlanner\ghdl\bin`.  
Example: `C:\repos\LinkPlanner\ghdl\bin`.  
Jump to step 10.
8. **If it exists**, click on the variable **Path** and press **Edit**. This should open another window;
9. Click on **New** to add another value to this variable. Enter your absolute path to the folder `\LinkPlanner\ghdl\bin`.  
Example: `C:\repos\LinkPlanner\ghdl\bin`.
10. Press **Ok** and you're done.

## 14.2 Using GHDL To Simulate VHDL Programs

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

### 14.2.1 Simulation Input

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

### 14.2.2 Executing Testbench

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

### 14.2.3 Simulation Output

The simulation will produce two files: **sim\_out\_1.sgn** and **sim\_out\_2.sgn** which will contain the output of the CPE BPS module.

