UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Optimization of digital signal processing routines for high speed coherent transmissions

Leonardo Marcon

Relatore: Prof. Andrea Galtarossa
Correlatore: Prof. Sergei Popov
Supervisor: Dr. Xiaodan Pang

Corso di Laurea Magistrale in Ingegneria delle Telecomunicazioni

Anno Accademico 2015/2016

Laureando

Relatore

UNIVERSITÀ DEGLI STUDI DI PADOVA

Abstract

Facoltà di Ingegneria
Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria delle Telecomunicazioni

Optimization of digital signal processing routines for high speed coherent transmissions

di Leonardo Marcon

Un framework di simulazione per sistemi ottici coerenti implementa varie routines per digital signal processors necessarie per correggere distorsioni quali dispersione cromatica, dispersione dei modi di polarizzazione, perdite dipendenti dalla polarizzazione e rumore di fase. In questa tesi vengono studiati alcuni di questi moduli e, basandosi sulla comprensione del relativo principio di funzionamento, un'ottimizzazione viene eseguita per ridurre il loro tempo di esecuzione. Il processo di ottimizzazione prevede di migliorare il codice degli algoritmi, riducendo il numero complessivo di accessi alla memoria e introducendo funzioni realizzate con un linguaggio di programmazione di più basso livello. Dopo la fase di ottimizzazione vengono valutate le prestazioni delle routines per misurare il miglioramento dei tempi di esecuzione e verificare il corretto funzionamento dei nuovi codici.

Acknowledgements

I must express my very profound gratitude to my parents, sister and friends for providing me with unfailing support and continuous encouragements throughout my years of study and through the process of researching and writing this thesis.
I would like to thank first my supervisors Dr. Xiaodan Pang from Swedish ICT Acreo and Prof. Sergei Popov from KTH. The doors of their offices were always open whenever I ran into a trouble spot or had a question about my research or writing. From Acreo I would also like to thank Gunnar, Richard, Oskars, Jaime and Aditya of the Optical Transmissions group for their support.
I would also like to thank Prof. Andrea Galtarossa and Prof. Luca Palmieri from University of Padova. I am gratefully indebted to them for all the very valuable comments on this thesis and for the chance they gave me to do my M.Sc. project abroad.
Finally I would like to thank all the people I have met and managed to know during my months in Stockholm, in particular Alberto, Lorenzo and Matteo, Elena and Mounir, Thomas, Roslan and all the others.

Leonardo Marcon

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Optical Communications

Every year an incredible amount of information is exchanged all around the world. In 2014 slightly more than 10^8 TBytes were exchanged, according to Cisco [1] and the predictions state that the world is finally entering the Zettabyte (10^{21} Byte) era. This enormous flow of data is possible only thanks to the huge bandwidth provided by optical fibers of increasing quality and to decades of researches in telecommunication systems that resulted in efficient transmitters and receivers. However, the continuous increase in the total capacity need, due to the rapid development of bandwidth-hungry applications still spurs research in this field and motivate researcher all around the globe to improve and reach further goals.

The optical communications field is the branch of telecommunications that studies how to properly implement a transmission system based on optical fibers, so it includes how to efficiently generate and modulate optical signals, how to transmit and how to properly receive them in order to recover the transmitted information.

1.1.1 Brief history of optical communications

Research and development related to optical fibers communication systems was initiated in the early 1970s. Such systems worked by implementing an *Intensity Modulation - Direct Detection* (IM·DD) scheme, consisting of an intensity modulated laser detected by a photodiode, which was independent of both randomly variating phase and *state*

of polarization (SOP) of the incoming signal. During that period also another scheme, called coherent receiver, was proposed where a local oscillator on receiver side was interfered with the input signal to extract both intensity and phase information. Coherent receivers can be realized both with homodyne and heterodyne schemes, but in both cases the output signals are highly sensitive to random variations in phase and SOP and so their configurations are much more complicated than the ones of IM·DD systems [2]. Due to this higher complexity, coherent receivers were mostly ignored during the so-called *first era* of optical communications, dated from 1977 to 1997 [3].

During these years the first optical communication link was deployed in Japan by using a multimode fiber as channel and a laser tuned at 850 nm. This system was short-lived due to the advantages in capacity and range provided by singlemode fibers and by the improvements in laser technology that manage to realize reliable sources tuned around 1300 nm. In the late 1980s the firsts *Erbium Doped Fiber Amplifiers* (EDFAs) were developed, opening the way for completely optical long-haul systems and *Wavelength Division Multiplexing* (WDM) techniques. However these advances were not sufficient to realize a huge improvement in optical communication systems due to the very high dispersion of *Standard Single Mode Fibers* (SSMF). Due to chromatic dispersion the maximum reach at that time was around 60 km for a 10 Gb/s bitrate. Around 1990 the first *Dispersion Shifted Fiber* (DSF) was developed in order to solve the issue of chromatic dispersion, allowing 10 Gb/s systems over many thousands kilometers, but DSFs were vulnerable to non-linear effects (in particular *Four Wave Mixing* (FWM)), and so they could not be used to implement WDM. To avoid these issues, around 1993, the Bell labs start engineering the dispersion profile of fibers obtaining fibers with negative dispersion coefficient. By concatenating fibers with opposite sign of dispersion it was possible to compensate the distortions over huge distances without incurring in non-linear effects and so the old systems started to get dismissed. It was around 1997-98 that Pirelli, Ciena, Alcatel and Lucent 40 Gb/s Dense-WDM systems start to become commercially available and a second era, called the "Dispersion managed era" started. The second era last approximately up to 2009 when EDFAs ran out of bandwidth due to a WDM scheme with 80 carrier wavelengths operating at 40 Gb/s. In order to increase system capacity to meet the global need it was then necessary to adopt more advanced modulation formats, like *Quadrature Phase Shift Keying* (QPSK) that could still be received without phase information, or M-ary Quadrature Amplitude Modulation (M-QAM) which instead require a coherent detection. Thanks to coherent receivers the

optical signal can be linearly converted in digital signals containing both phase and amplitude information allowing the use of *Application Specific Integrated Circuits* (ASICs) to correct in an very cheap and effective way a wide variety of impairments, included an arbitrary amount of chromatic dispersion [3].

1.1.2 Optical communication systems

An optical communication system is a communication system that use electromagnetic radiation in the spectrum of visible or infrared light to carry information. It can be separated, in first analysis, in three main blocks:

- **Transmitter:** This block includes the sources of the optical signal, usually lasers with a very narrow linewidth, the pre-compensating *Digital Signal Processor* (DSP) blocks, used to introduce specific distortions that will be compensated during the propagation of the signal, the modulators that modulate the optical carrier in accord with the input data, the pulse shaping module to shape the pulse in a way suitable for transmission and the coding module that performs some coding over the data and implements error correction algorithms (usually *Forward Error Correction* (FEC)).
- **Channel:** The channel consists in the various spans of optical fibers used to realize the link between transmitter and receiver. The single span usually includes also an amplifier (EDFA) and a patch of *Dispersion Compensating Fiber* (DCF) with adequate dispersion coefficient to regenerate as much as possible the optical signal. Lately, thanks to improved DSPs on receiver side, the DCF can be avoided since it is possible to correct, in the electrical domain, an almost arbitrary amount of chromatic dispersion. This allows to reduce the number of amplifiers required along the link thus reducing the correlated *Optical Signal to Noise Ratio* (OSNR) penalty.
- **Receiver:** The scheme implemented by the receiver can be more or less complex, depending on the modulation imposed by the transmitter, but at the moment the principals commercially available alternatives are the direct detection (DD) and the coherent detection. DD is much more easier to realize, it is very cost-efficient, but does not allow to recover the phase information of the optical signal so only

intensity modulations like M-PAM, and with some more complexity QPSK, can be used. Coherent receivers are more complex and power consuming, but allow to linearly map both amplitude and phase components of the optical signal into digital ones, thus allowing the use of more spectrally efficient modulations formats. After the detection, usually, a cascade of DSPs can be found trying to correct a wide variety of impairments like *Chromatic Dispersion* (CD), *Polarization Mode Dispersion* (PMD) and *Polatization Demultiplexing* (PD) (for systems implementing *Polatization Division Multiplexing* (PDM)), timing recovery, carrier phase recovery, and so on. Finally the decoder retrieves the transmitted data and eventually corrects errors.

1.2 Objectives of the thesis

High speed optical fibers transmissions of advanced signal modulations, like PDM-QPSK or PDM-16QAM, are enabled by coherent detection technology. The transmitted data can be recovered with a coherent receiver where impairments induced by fiber transmission are mitigated by specifically developed digital signal processing (DSP) algorithms. These algorithms form a demodulation routine which is normally employed in an offline processing mode in the research environment. However, in order to evaluate the transmission performances in a "quasi-real time" mode, particularly for research activities investigating fast changing effects in the optical fibers transmissions, the DSP routine will require speed optimization, so that fast signal demodulation and evaluation become possible.

In this thesis some MATLAB-based DSP routines in a complete simulation platform for PMD-16QAM coherent transmissions, consisting of CD compensation, adaptive equalizer, carrier phase recovery and *Bit Error Rate* (BER) counter will be analyzed. Then, based on the understanding of the algorithms, DSP routine optimization with different possible ways to speed up the process (algorithm code optimization and code compiling) will be performed. Once the optimization is done, the performance of each step of the routine will be evaluated.

1.3 Thesis outline

The thesis will initially provide a complete theoretical background regarding state-of-the-art coherent receivers structure, with a particular focus on the latest mainstream DSPs algorithms. The decoder module of the receiver will be neglected because it is beyond the scope of this thesis.

The following chapter will focus on the Equalizer module, starting from a more detailed explanation of its purpose and by a complete analysis of the algorithm implemented. The chapter will continue with the proposed changes introduced and a description of the results obtained.

The thesis will continue with the analysis of the Carrier Phase Recovery module, once again by starting from explaining purpose and algorithm implemented and by continuing with the proposed changes and corresponding results.

Finally the last chapter will conclude the thesis and in the appendix it will be possible to find a small guide to optimization in MATLAB and to the use and realization of MEX files.

Chapter 2

State of the art

2.1 Coherent transmitters

To develop methods for meeting the ever-increasing bandwidth demand multi-level modulation optical coherent systems, which implements coherent transmitters and receivers, are required. Such devices were adapted from the radio frequency counterparts and implement *Digital Signal Processors* (DSPs) to realize useful corrections on the signal to help compensating propagation impairments. A general scheme for commercially-available coherent transmitters is shown in Figure (2.1).

The usuals DSPs of a coherent transmitter include functions like Nyquist spectral shaping, signal pre-distortion/compensation, coding and *Forward Error Correction* (FEC) schemes [2], however these arguments are beyond the scope of this thesis and the attention will be focused on coherent receivers.

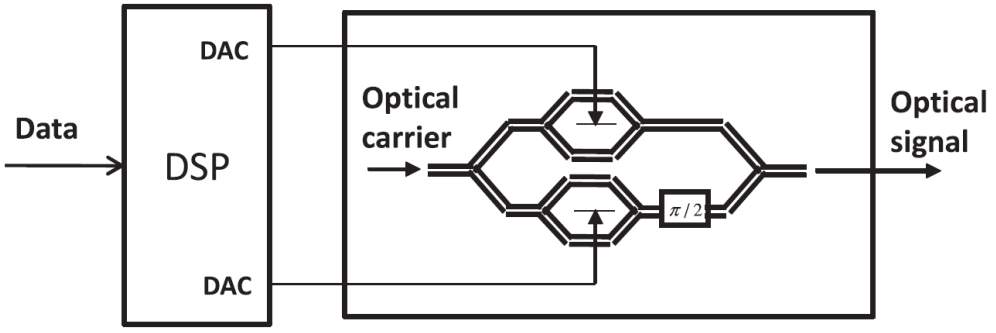

FIGURE 2.1: General scheme of a coherent transmitter.

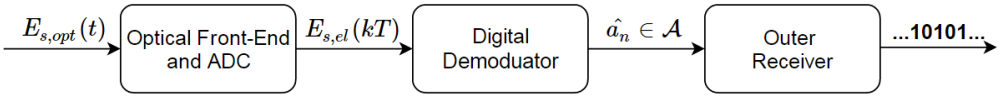

FIGURE 2.2: General subsystems of a coherent receiver. The optical Front-End digitalizes the electric field of the optical signal in input, the Digital Demodulator correct the distortions and convert the digital samples into symbols of the constellation $\mathcal{A}$ and finally the outer receiver decodes the symbols and recover the bit stream.

2.2 Coherent Receivers

Coherent detection was experimentally demonstrated as early as 1973 [4], but at that time complexity issues in tracking phase and polarization hindered the development of such receivers. In actuals digital coherent receivers these problems are dealt with in the electrical domain, by implementing adequate DSP algorithms, resulting in a dramatic reduction of the overall complexity [5].

Since coherent detection allows to map the entire electric field of the optical signal, within the receiver bandwidth, into the electrical domain the use of DSPs, together with fast enough electronic [6], allows also a drastic increase of the receiver sensibility.

A coherent receiver can be decomposed in three general subsystems [6] as shown in Figure (2.2):

- **Optical Front-End and ADCs**: The components that map, as linearly as possible, the optical field into discrete-time quantized signals at a particular sampling rate. Usually more than 1 sample per symbol is taken;
- **Digital Demodulator**: It corrects the distortions related to propagation impairments and non-idealities of the devices. Then it converts the digital samples into a valid sequence of symbols of the constellation at the symbol rate, i.e. exactly one sample per symbol;
- **Outer Receiver**: It Includes error correction and those functionalities which allow an optimal decoding. The output is the received bit stream.

The first two points form the *Inner Receiver* and their objective is to produce a "synchronized channel" which is as close as possible to the information theoretic communication

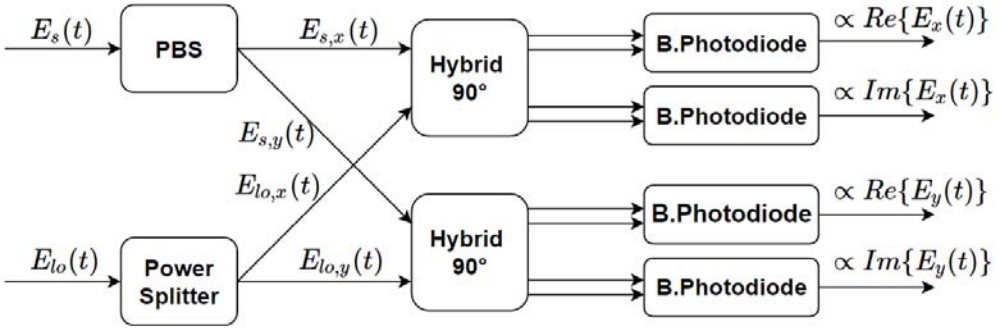

FIGURE 2.3: Schematic of a phase and polarization diverse receiver. $E_{s,x}$, $E_{s,y}$ and E_{lo} are the electric fields associated to the two polarizations x and y and to the local oscillator respectively.

channel [6]. Since the analysis of the outer receiver is beyond the scopes of this thesis, the attention will be focused on the inner one.

2.2.1 Optical Front-End

The usual phase and polarization-diverse scheme for the optical front-end of a coherent receiver is shown in Figure (2.3), where the inputs are optical signals and the outputs are electric ones. One of the most important rules that an optical Front-End must follow is that the architecture chosen to implement the receiver must have no influence on the subsequent DSPs [5]. The electric field of the input optical signal enters a polarization beam splitter which divide the x and y polarization components into two different branches. Standard SMFs used for long-distance telecommunication links are not polarization-maintaining, so the state of polarization of the optical signal is randomly changed during the propagation. The electric fields in output of the polarization beam splitter are then a combination of the original polarized transmitted components and in order to recover the original data, signal equalization is required. This topic will be discussed in a more detailed way in section 2.2.5.

Most of the commercially available coherent receivers to date implement homodyne schemes instead of heterodyne ones to recover in-phase and quadrature components because, due to the huge bandwidth associated to optical carriers, it is much easier to deal with baseband signals instead of intermediate frequency ones. To perform the recovery then a local laser source, called *Local Oscillator* (LO), is used where its frequency must be roughly equal to the one of the optical carrier [7]. It is not necessary usually to perfectly match the carrier frequency of the input optical field and of the LO, since

corrections can be applied later on in the electrical domain, by using adequate DSP algorithms.

The LO signal is splitted and combined with the x and y polarized components of the input electric field using a 90° hybrid coupler which is a 6-port, 2-input and 4-output, passive device [6, 8]. The 8 output ports of the hybrid couplers are then connected to four balanced photodiodes which manage to extract the following analog continuous-time signals:

$$\begin{pmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \end{pmatrix} = \frac{2}{5} \begin{pmatrix} Re(E_x E_{lo}^*) \\ Im(E_x E_{lo}^*) \\ Re(E_y E_{lo}^*) \\ Im(E_x E_{lo}^*) \end{pmatrix} + \frac{1}{10} \begin{pmatrix} 2|E_x|^2 + 2|E_{lo}|^2 \\ 4|E_x|^2 + |E_{lo}|^2 \\ 2|E_y|^2 + 2|E_{lo}|^2 \\ 4|E_y|^2 + |E_{lo}|^2 \end{pmatrix} \tag{2.1}$$

where E_x, E_y and E_{lo} are the complex electric fields of the input optical signal.

By using a DC block device and ensuring that the local oscillator to the signal ratio is significantly larger than the signal to noise ratio it is possible to minimize the influence of the second term and successfully extract electric signals proportional to the in-phase and quadrature components of the received optical one [5, 6].

After the optical to electrical conversion it is necessary to convert the analog signals into a set of digital ones by using *Analog to Digital Converters* (ADC). These devices can be considered as made up of two subsystems: a sampler which samples the signals in time and produce discrete-time analog output signals and a quantizer which convert these signals into a finite set of values determined by the bit resolution of the ADC. The sampling rate of the ADC is a critical parameter, but there are various possible choices including flash ADCs, flash with track and hold, and time interleaved ADCs which provide trade-offs between performances and costs or power efficiency. The most used is the time interleaved [9]. The ADC usually samples with a number of samples per symbol non integer but greater than one to exploit as much as possible the available bandwidth and preserve the signals information. In such case a proper decimation operation result necessary before equalization.

To date, the highest electrical sampling rate for a commercially-available ADC is 92 GS/s [10], however almost no systems implement such ADCs due to power consumption limitation. For a 100 Gbps (PDM-BPSK) or 400 Gbps (PDM-QPSK) channel in a WDM grid is more convenient to implement a proper number of power efficient 56

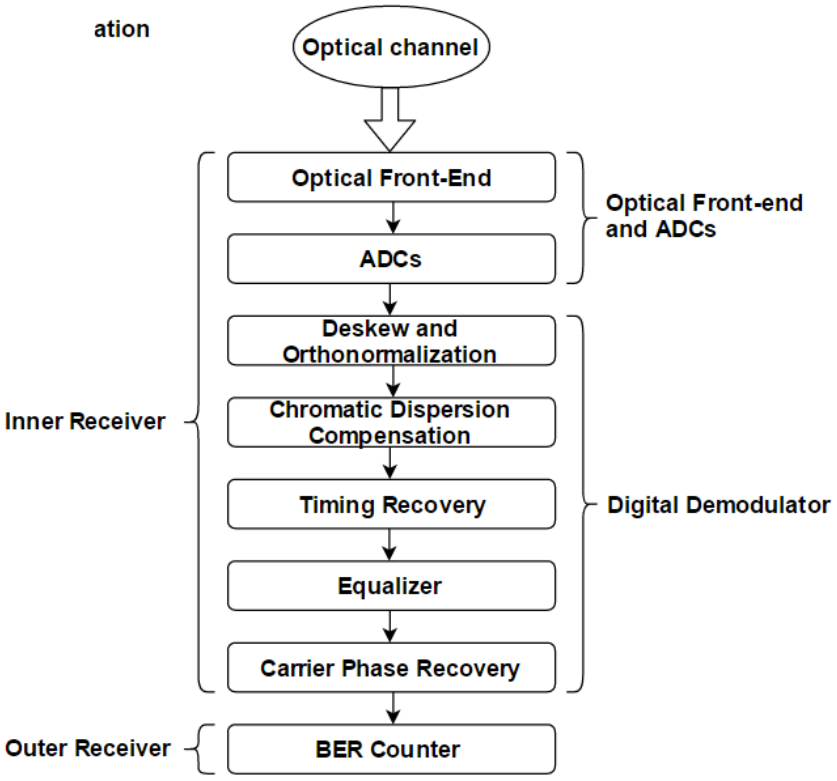

FIGURE 2.4: Complete logical scheme of a coherent receiver with all digital demodulator's modules listed

GS/s ADCs [10] and eventually increase the effective sampling rate with techniques like time interleaving or digital bandwidth interleaving. Recent optical front-ends in coherent receivers can reach sampling rates up to 160 GS/s or 240 GS/s thanks to these techniques [11].

2.2.2 Deskew and Orthonormalization

After the four ADCs the digitalized samples enter the digital demodulator subsystem which complete logical scheme can be seen in Figure (2.4).

The first module of the digital demodulator perform two different operations called deskew and orthonormalization in order to compensate for all the non-idealities introduced by the optical front-end, thus desynchronization, responsivity variations in the photodiodes and imperfections in the 90° hybrids [5].

Due to different physical branches lengths for the x and y polarization components of the optical signal, the samples streams can be disaligned of tens of samples. Deskew routines compensate for this path length mismatch, but usually a fractional delay, i.e. same position samples of the two polarization components do not start at the very same

moment due to a time difference smaller than a sampling period, still remains.

In order to correct the remaining fractional delay it is necessary to implement a retiming algorithm, but the position of this block in the logical scheme of the coherent receiver is still subject to discussions. C. Fludger et al. [5, 12] suggest to insert this block before the dispersion equalization module, while S. Savory [6] and M. Kuschnerov [7] implement it after the compensation. The main reasons for the difference in position is related to performance requirements and limitations. The simulation setup used in the second part of this thesis implements the retiming module after dispersion compensation and a reason for this choice is provided by Kuschnerov [7] which specifies that if dispersion compensation is done before of the timing recovery block, the subsequent algorithm can remains identical for compensated and uncompensated links with similar residual dispersion requirements.

After deskew it is necessary to compensate the non-idealities of the optical front-end, especially non perfect orthogonalization of the hybrid 90° couplers and the non flat frequency response of the balanced photodiodes. Some algorithms have been proposed so far, but the most implemented are the Gram-Schmidt and the Löwdin ones [6]. The Gram-Schmidt algorithm creates a set of mutually orthogonal vectors, taking the first vector as a reference against which all subsequent vectors are orthogonalized. As a result the first output vector is equal to the input one, while the others can be very different thus resulting in an increased impact of quantization noise for the displaced components. The Löwdin algorithm try instead to generate orthogonal vectors which are, in a least-mean squares sense, closest to the original ones. The complexity of this algorithm is slightly greater than the previous one, but it manage to reduces the impact of quantization noise. To date, most of the commercially-available systems implements without significant issues the Gram-Schmidt algorithm, but simulations for very high performances systems are slowly moving to other algorithms [13].

To provide an example of how the DSPs of a coherent receiver act over a signal, in each of the following section a graphical representation of the discussed module output signal will be shown. The transmitted signal in this simulation carries a PDM-16QAM modulation, as can be seen in Figure (2.5), at a carrier wavelength of 1550 nm with a symbol rate of 50 GBaud and an imposed OSNR of 40 dB. The channel simulated is 80 km long and it is followed by a single EDFA to compensate for channel loss. The output signal of the coherent front-end for one of the polarization components after desked and orthonormalization operations is shown in Figure (2.6). As can be seen the

received signal is completely different from the transmitted one and no useful data can be extracted at this stage.

2.2.3 Chromatic Dispersion Compensation

One of the key distinguishing features of a digital coherent receiver is its ability to compensate for transmission impairments, in particular *Chromatic Dispersion* (CD) and *Polarization Modes Dispersion* (PMD) [14, 15], thanks to the linear map between optical and electical domain performed by the optical front-end.

While in principle equalization can be realized in one subsystem [12, 16], it is generally beneficial to partition the problem into static and dynamic equalization [5–7, 17].

In case of single subsystem CD, PMD and *Polatization Dependant Losses* (PDL) are compensated all together and it is possible to use *Decison-Feedback Equalizers* (DFE), Viterbi equalizers (which perform better than DFEs) [16] or a system of 4 Finite Impulse Response (FIR) adaptive filters arranged in a butterfly structure [12]. In this case the

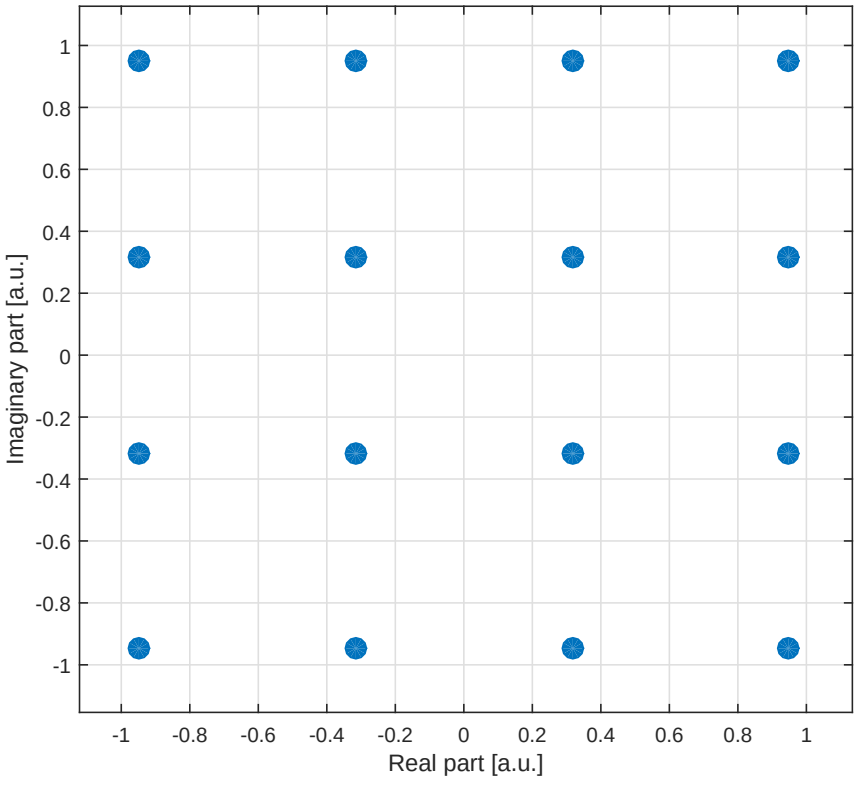

FIGURE 2.5: Graphical representation of the transmitted symbols for one polarization component.

coefficients of the filters can be updated with various techniques which will be discussed in more detail in section 2.2.4.

For diverse equalization there are two possible ways to deal with chromatic dispersion. The first one implement an adaptive FIR filter which coefficients can be estimated both using a blind estimation, for example using Godard's Constant Modulus Algorithm (CMA) [18], or with data-aided estimation by transmitting periodically a known training sequence [7]. The pros of blind estimation are that there is no overhead and that it works well with short-memory channels while the cons are that the adaptation length increases with channel memory and it is not possible to guarantee the convergence of the coefficients. With data-aided estimation a fast convergence is guaranteed but there is a significant overhead and time-varying effects may influence the lowest repetition rate of the training sequence. To date, blind estimation is preferred since the variation of the CD impulse response is not fast and convergence can be guarantee even for long filters by taking precautions [7].

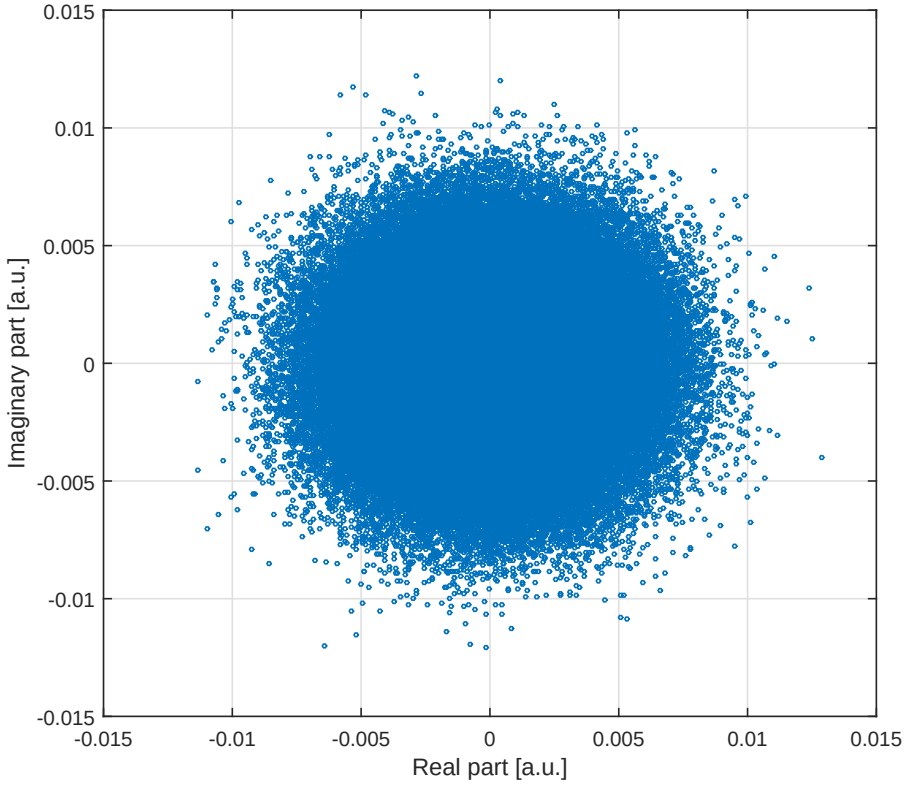

FIGURE 2.6: Graphical representation of one of the polarization components of the received signal after the optical front-end, the deskew operation and the orthonormalization. No equalization has been performed yet.

The complexity of the filter, expressed as the number of complex multiplication required to perform compensation, depends on its design, but it can be shown that frequency-domain equalizers behave better for high values of dispersion[5, 7].
To date no systems implement adaptive filters for chromatic dispersion compensation, since it is more convenient to realize static filters which length and coefficients are decided upon the theoretical CD-related impulse response of the fiber [5, 6] windowed by an adequate window (for example the Kaiser window [17]). Since the chromatic dispersion impulse response is slowly time varying, the coefficients of the static filter must be occasionally updated, but due to the slow variation of the dispersion coefficient the related complexity is negligible. With this technique in 2007, up to 100000 ps/nm of chromatic dispersion was compensated after a length of around 6400 km for a transmission at 42 Gbit/s using PM-QPSK [17] while to date, principally thanks to improved electronics and DSPs speed, an almost arbitrary amount of CD can be compensated for any modulation format [19]. The number of taps of the filter can be optimized [5] by accepting a small Q-factor penalty or by allowing an higher complexity of the algorithm, as proposed by Y. Liu et al. [19].
Another type of equalization that can be performed together with CD compensation is non-linear compensation which accounts also for non-linear impairments. The implementation of such filters however are quite challenging and are still open research fields [6].
The results of CD compensation over the received signal is shown in Figure (2.7). As can be seen the symbols are less spread, but overall the signal is still completely corrupted.

2.2.4 Timing Recovery

The main objective of the timing recovery block is to remove the intrasymbol desynchronization between the two polarization components and, eventually, to create output signals with a single sample per symbol. One of the most important problems is that in a digital coherent receiver the ADCs clock rates, defined as $1/T_{s,i},\ i = 1, .., 4$ with $T_{s,i}$ the sampling periods, are not perfectly the same and moreover they are not directly related to $1/T$ with T symbol period. It is important to highlight this problem because while it is possible to build extremely accurate clocks to match the rates, small time differences will always be present and, in the long run, they would cause dangerous errors called

cycle slips [20]. The term cycle slips is overloaded in the coherent optical communications field since it is commonly used to define both errors related to a desynchronization of the data streams and errors related to wrong carrier phase estimation as will be seen in section 2.2.6.

Both non-data-aided [21] and data-aided [22] algorithms can be employed, but often to avoid overhead the non-data-aided ones are preferred. Over the time a lot of possible algorithms were proposed, depending on the modulation and on the requirements of the system. For basic modulations (PAM, BPKS, QPSK) algorithms that try to maximize the squared modulus of the interpolated signal were proposed [6, 20, 23] while, for higher order modulations the "filter and square" algorithm could be used [12, 24]. The algorithms used presently to perform timing recovery are not changed much with respect to the older ones [25], and they all are based on the Gardner algorithm [21] or on the Godard one [26]. For high performances systems there are also slightly more precise but more complex and power consuming schemes that require an interpolation to four samples per symbol [7].

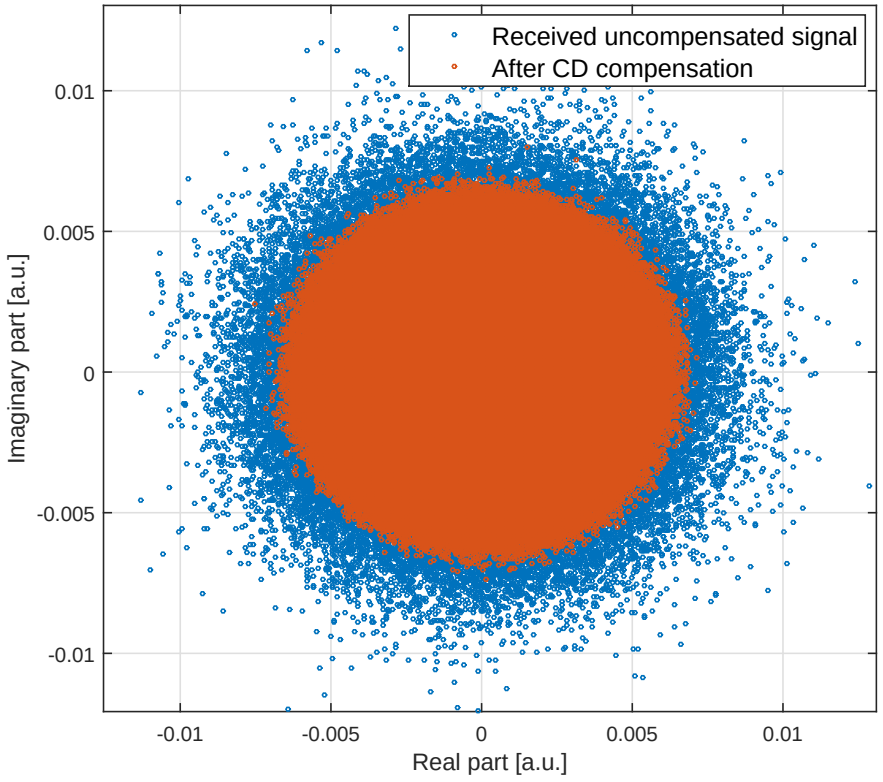

FIGURE 2.7: Graphical representation of one of the polarization components of the received signal after CD compensation.

Another aspect that is important in the design of the timing recovery block is the residual dispersion tolerance. This tolerance depends on the algorithm implemented, but usually even for robust algorithms an high residual dispersion results in worse estimates. To reduce the impact of dispersion on the timing recovery phase it is possible to apply a filter before the timing error detector [7], however it is more convenient to realize a more performing CD compensation block than implementing the filter here.

The graphical representation of the output signal after timing recovery is shown in Figure (2.8). As can be seen no major differences with the previous figures are visible, since a few more significant impairments are still corrupting the symbols. However after timing recovery the distribution of the received symbols is no more uniform and some shapes, whose geometry depends on the modulation applied, start to become visible.

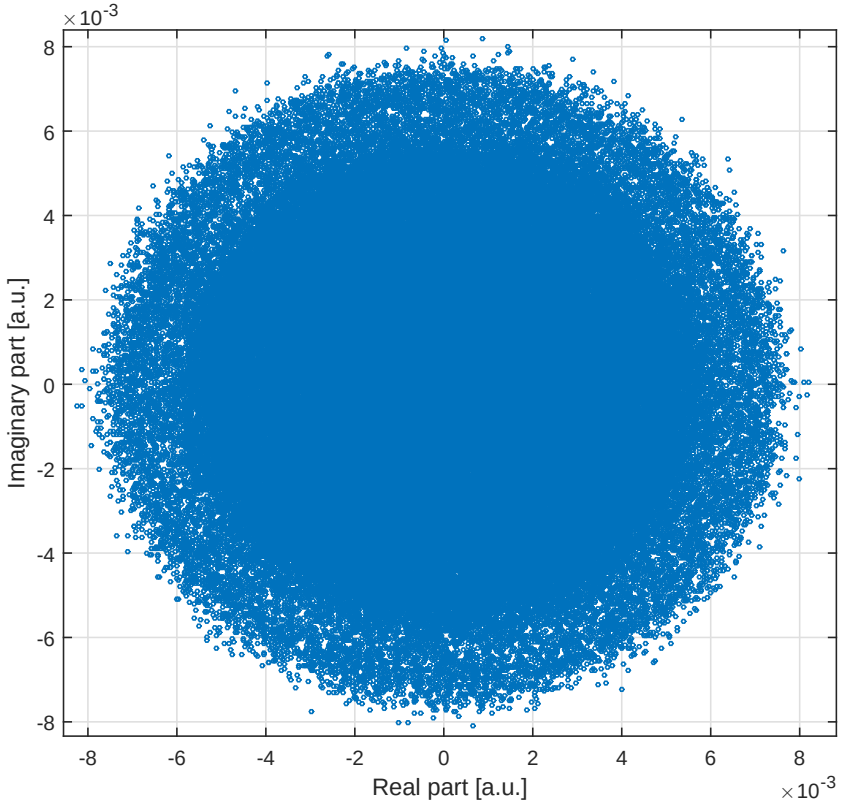

FIGURE 2.8: Graphical representation of one of the polarization components of the received signal after timing recovery.

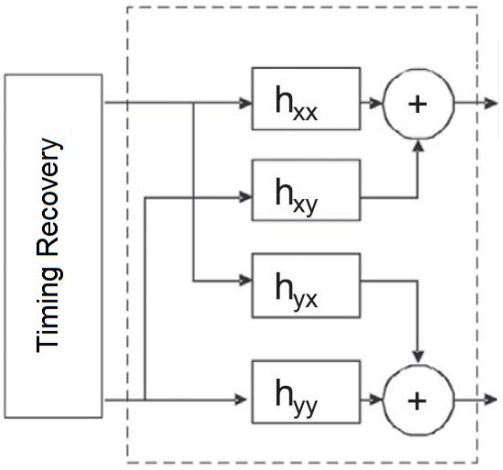

FIGURE 2.9: Butterfly structured equalizer with 4 FIR filters used to compensate polarization impairments.

2.2.5 Equalizer

Standard telecommunications fibers are not polarization maintaining, so even if the signal is transmitted over two orthogonal linear polarization states, usually defined x and y components, the received polarization states are still orthogonal, but unknown. Neglecting in first approximation non-linear effects, the optical channel can be described by a 2x2 rotation matrix called Jones matrix, which is unitary if also *Polarization Dependant Losses* (PDL) are neglected.

The main objective of the polarization impairments compensation block is to estimate the Jones matrix, inverting it and equalize the system (compensating for PDL, if considered and PMD) however, since the effects of PMD are rapidly time-varying, principally due to variations in environmental conditions (like temperature, pressure, humidity, etc.) which cause stresses over the fiber, the estimation must be done continuously with a sufficiently fast adaptive estimator.

Linear equalizers are not sufficient to properly solve these impairments, so structure derived from wireless MIMO receivers were researched and, to date, butterfly structured equalizers composed of 4 different FIR filters, Figure (2.9) are used.

The length of the filters influence the tolerance of the equalizer with respect to the impairments that it has to correct [12], but it has consequences in the adaptation speed: filters with too many coefficients cannot update at an high enough speed and this degrades the performances of the equalizer [5].

The algorithms that can be used to update the coefficients of the filters depends on the

modulation used. For constant modulus modulations, like m-PSK, CMA derived from Godard's work[18] can be used [5, 6, 17], while for higher order modulation *Radiant Directed Equalizers* (RDE) are more adequate [6, 27]. In both cases however there are some considerations that must be done before implementing the algorithm, which regard the initial tap weights and the convergence conditions. To assure convergence a training sequence can be used, but as usual this results in an undesired overhead so, often, blind optimization is performed by using a CMA algorithm to initialize in a proper way the tap weights and successively a more precise and fast DD-LMS that would have issues in converging if not well initialized [6]. Another way to guarantee a good convergence is to perform a proper number of iterations over the same block of data. In this case, at the beginning of each iteration except for the first, the coefficients must be initialized by assigning them the value they had at the end of the previous iteration.
Given that the equalizer is unconstrained with respect to its outputs, it is possible for it to converge on the same output, corresponding to the Jones matrix becoming singular. This can be avoided by monitoring the estimated Jones matrix determinant such that if it begins to approach 0 then the equalizer is reinizialized with different tap weights [5]. Another important aspect is the power consumption of the equalizer: it can be shown that up to 50% of the power required by the equalizer can be saved with appropriate implementations of the algorithm and just a slight penalty in the performance. This approach however put a significant constraint over the residual CD of the input signal [28]. The graphical representation of the output signal after polarization effects equalization is shown in Figure (2.10) where it can be finally seen that the symbols are laying on the circles characteristic of QAM modulations. Now all the noise induced by polarization related impairments is gone, and the only issue still to be compensated is phase noise.

2.2.6 Carrier Phase Recovery

The last operation that, to date, is usually performed in a commercial available coherent receiver is the so called Carrier Phase Recovery. The impairments that this block tries to compensate are related to the residual frequency offset between the transmitter laser and the LO, and the phase noise due to a non-null linewidth of the lasers. In the context of DSPs techniques for digital receivers the two impairments estimations are often separate. Compensating the frequency offset before moving to the phase noise improves the performances of the system since it reduces the overall amount of phase that the phase

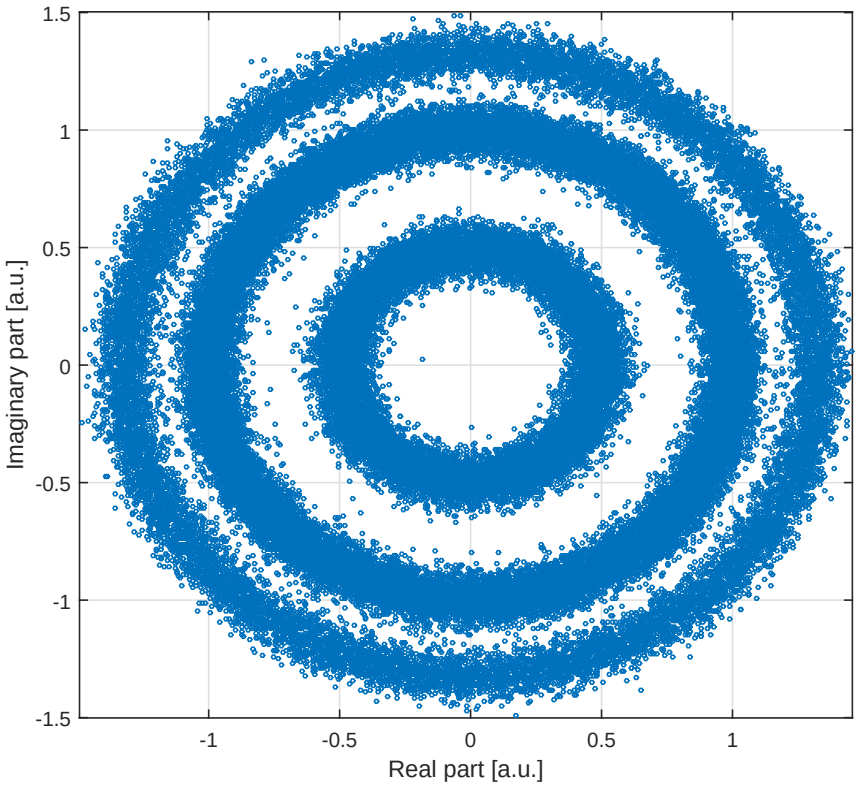

FIGURE 2.10: Graphical representation of one of the polarization components of the received signal after equalization.

recovery block has to track. Moreover many phase recovery algorithms are unbiased only for zero frequency offset.

As said in section 2.2.1, the output frequency of the local oscillator should be equal to the one of the optical carrier, but it is not uncommon to have residual frequency offset due to non-idealities and different environmental conditions between transmitter and receiver lasers. The result of the frequency offset is a progressive liner rotation of the phase of the received symbols that will results in cycle slips if not compensated. After the compensation of the linear frequency offset, the remaining phase-noise is due principally to the non null linewidth of the transmitter and local lasers. A non-null linewidth results in a random time-varying phase at receiver side that must be estimated and removed from the received streams of data to keep the symbols aligned with the constellation.

Historically many solutions to estimate carrier phase were implemented. In the beginning of coherent detection *Phase Locked Loops* (PLL) [15, 29] were applied, which are independent on the modulation format but are not very tolerant to delays. Successively it was demonstrated that slightly more complex feedforward and feedback techniques

could be implemented with better performances and tolerances [30–35]. Some of the algorithms proposed are based on the Viterbi and Viterbi estimators [36] and utilize the knowledge of the constellation to perform N-th power operation on the received data. The result of this non linear operation is a modulation-free stream of symbols that allows the computation of the time-varying phase noise and the consequent compensation. These algorithms can be improved by implementing a weighting function which depends on the ratio of the AWGN to the laser phase noise [32], but even if their performances are quite good it is hard to efficiently implement these algorithms from an hardware point of view. Viceversa, a different approach consists in implementing the so called "barycenter algorithm" which is a particularly hardware-efficient phase estimator [37], but with worse performances with respect to the previous solutions. When more advanced modulation formats are used the requirements on the laser linewidth becomes increasingly stringent, however advanced but mor complex decision-directed PPLs allow large linewidths ($\sim$1 MHz) to be tracked with a sufficient precision [27].

To date, many blind algorithms are still used, like the Viterbi and Viterbi and the advanced DD-PLL, but new approaches are being investigated like a modified version of the CMA algorithm [21, 38] which provides both low complexity and high performances. Also DD-LMS algorithms are implemented sometimes, but due to their low reliability when they start in blind mode of operation it is necessary to implement also an initial training mode that is detrimental to the performances of the receiver [38]. Finally, recently also hardware-efficient blind digital feedforward carrier recovery algorithm [39] are being implemented, since they provide precise and linewidth tolerant estimations even for higer order modulations like 16, 64 or 256-QAM.

If the estimation of the carrier phase is not good enough the residual phase noise may cause a cycle slip. If a cycle slip occurs at receiver side the consequences can be catastrophic since, if no precautions are taken, all the following symbols will be decoded wrongly. There are two possible ways to deal with this problem. The first is to avoid cycle slips by reducing as much as possible the laser linewidth to a value behind the tolerance of the algorithm. To date the tolerance of the algorithms implemented are in the order of MHz, and the linewidths of state-of-the-art lasers are around few hundred of kHz [6, 32]. The second possibility is to use differential encoding to reduce to one the number of symbols decoded wrongly due to a cycle slip.

The graphical representation of the output signal after carrier phase recovery is shown in Figure (2.11) where it can be seen the output constellation. The signal is still slightly

noisy, but the information associated to the signal, usually protected with FEC algorithm and codes, can be recovered perfectly without any major issue.

2.2.7 Other impairments: EEPN and Nonlinearities

To date coherent receivers can compensate almost arbitrary amounts of impairments like CD, PMD and phase noise, but they are still far to be perfect. In particular commonly used high speed DSPs schemes and DWDM are experiencing different impairments that were not significant problems up to few years ago, like *Equalization Enanched Phase Noise* (EEPN) and non-linear phenomena. Both these phenomena are beyond the scope of this thesis, so they will just be introduced, without entering into details

Recently it was observed that the received symbols in uncompensated coherent optical links, even after DSPs equalization, remained influenced by unexpected noise, that was defined Equalization Enhanced Phase Noise [40]. Various studies, for different modulations on receiver carrier recovery and adaptive equalizers, have been performed to

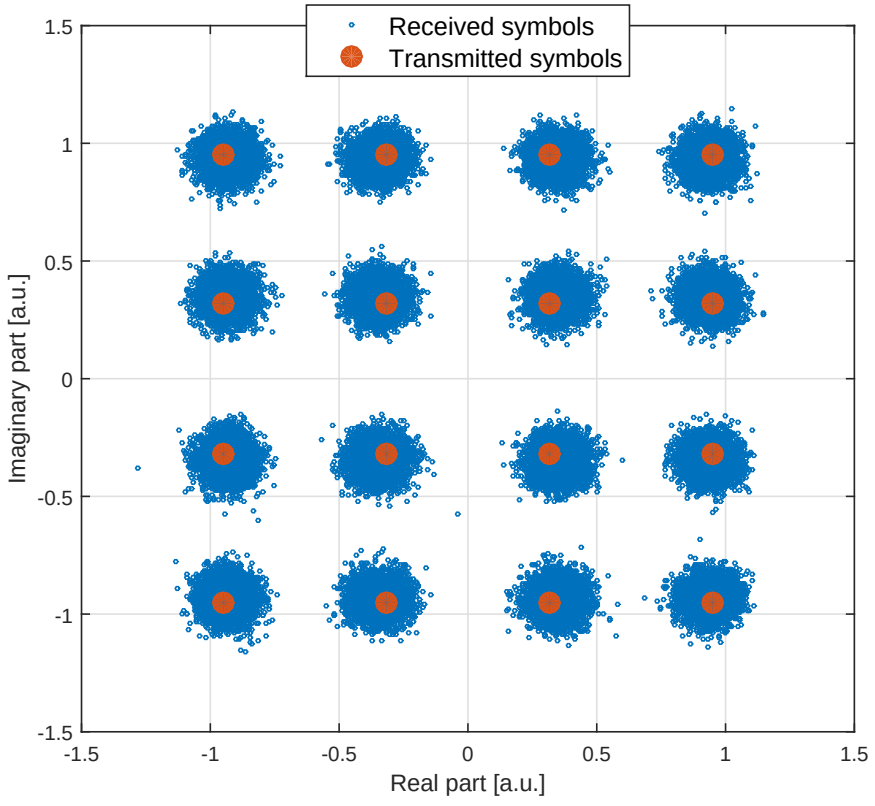

FIGURE 2.11: Graphical representation of one of the polarization components of the received signal after carrier phase recovery.

evaluate the effects of such impairment and the results shown that the phenomenon is not present in the case of channel with ideal dispersion compensation in the optical domain. Some possible schemes have been proposed so far to attenuate such effect [41], but they are still under intense research.

Optical fibers are non linear channels so, when the power of the input signal is too high, the power of the output one is no more linear with the input. In particular Four Wave Mixing, Self Phase Modulation, Raman Scattering and Brillouin Scattering are the most common phenomena and, except for some particular applications, they are undesired during propagation. The practical consequence of nonlinearities over an optical transmission is a limit to the maximum achievable spectral efficiency [42] which will result in a very dangerous capacity crunch in the near future if no countermeasures are taken. To solve this issue *Spacial Division Multiplexing* (SDM) [43] and transmission based on the results of the *Nonlinear Fourier Transform* (NFT) [44] are very active research fields. SDM tries to increase the capacity of the channel by realizing fibers with more cores or with a single core large enough to allow the propagation of more than 1 mode (the so called *Few Mode Fibers* (FMF)). The transmissions based on NFT instead try to assign information to solitons in order to easily mitigate non-linear impairments on receiver side. As said before, both field are undergoing intense research and to date it is possible to find various papers and books addressing them.

Chapter 3

Equalizer

3.1 Theoretic background

Before starting the analysis of the optimization process on the Equalizer script, it is important to understand the physics behind the impairments that the module has to correct and how this knowledge can be applied. In the next paragraphs the theory behind polarization related phenomena will be briefly, but properly, discussed and the theoretical approach to the problem, proposed by P. J. Winzer et Al. [27], will be analyzed.

3.1.1 Theory of polarization-related phenomena

Ideal single-mode optical fibers are geometrically symmetric media, with perfectly round core, no external forces acting on them and experiencing the same environmental conditions, i.e. temperature, humidity, pressure, etc., along all the length. Under these hypotheses, the two orthogonally polarized modes composing the fundamental $LP_{0,1}$ mode propagate in the same electromagnetic environment with the same frequency-dependent propagation constant.

Unfortunately none of these hypotheses holds in reality and standard SMFs are characterized by an elliptic core with position-dependent random oriented axes (usually the axes' directions of two consecutive sections of a fiber can be considered uncorrelated after some hundreds of meters) and with internal and external forces acting on the core. The principal consequence of such non-idealities is that the two orthogonally polarized

modes see different electromagnetic structures with different propagation constants causing coupling and distortions. This phenomenon is called *Polarization Modes Dispersion* (PMD) and is one of the most researched and famous impairments of optical fibers. The most problematic issues related to PMD are that it is random and rapidly time varying: any event or phenomenon that effect the electromagnetic structure of the fiber, thus for example any localized change in environmental conditions, change the propagation constants seen by the two polarization components.
Another impairment related to polarization appears when optical systems with a large variety of non ideal optoelectronic devices, like the usual undersea optical links, are considered. These non-idealities result generally in a lot of different issues, but in particular if the behaviour of the devices is polarization-dependent the orthogonally polarized components of the signal can experience *Polarization Dependent Losses* (PDLs) [45]. It is not possible to consider separately PMD and PDL since, in first analysis, the PMD distribution is altered by the PDL [46–48].
The combination of PMD and PDL can be described mathematically with the so called Jones matrix which must be estimated and inverted at receiver side to equalize the transmitted signal. Since the matrix is time varying, an adaptive method must be implemented as will be seen in the next paragraph.

3.1.2 Proposed algorithm

The algorithm proposed by P. J. Winzer et Al. [27] is blind, in order to avoid inefficient overhead, and based on the lattice filter whose structure is visible in Figure (2.9). The filters implemented are FIR with transfer functions $H_{xx}(f)$, $H_{yx}(f)$, $H_{yx}(f)$ and $H_{yy}(f)$ and a number of taps equal to N. The taps can be fractionally spaced, which mean that the filters can deal with symbols represented by more than 1 sample. The proposed implementation can be split into two consecutive steps: the first regarding the preconvergence scheme, realized with a *Constant Modulus Algorithm* (CMA), and a second which includes a more accurate training of the coefficients and a more precise equalization thanks to a *Multi Modulus Algorithm* (MMA).
When a blind algorithm is implemented the first thing that must be taken into consideration is how to properly initialize the filter coefficients because naive setups can lead to worse performances. In the case described in the paper this issue is dealt with by using a CMA scheme to properly train the coefficients and thanks to this choice it is possible

to use the most easy setup for the initial filter coefficients which is:

$$\mathbf{H} = \begin{cases} \mathbf{I}, & \text{for } n = 0,\ n \in (-(N-1)/2, (N-1)/2)\,; \\ \mathbf{0}, & \text{otherwise;} \end{cases} \tag{3.1}$$

where

$$\mathbf{H} = \begin{pmatrix} H_{xx} & H_{xy} \\ H_{yx} & H_{yy} \end{pmatrix} \tag{3.2}$$

is the matrix of the transfer functions of the FIR filters which are vectors of N elements, N is supposed odd without any loss of generality, $\mathbf{I}$ is the 2x2 identity matrix and $\mathbf{0}$ is the 2x2 null matrix. Obviously if prior knowledge allows to setup the initial coefficients to more appropriate values it is convenient to do it.

The CMA is a well-proven blind filter adaptation algorithm that is commonly used because it is simple, robust and works independent of carrier frequency and phase which are still not available at this stage. It works by minimizing the time-averaged error:

$$\langle \varepsilon_{CMA,pol} \rangle = \langle R^2 - |s_{i,pol}|^2 \rangle \tag{3.3}$$

where $pol \in (x, y)$, R is the radius of a circle in the complex plane, s_i are the equalized symbols at time index i, one per polarization, defined as:

$$s_i = \mathbf{H}_i \cdot [\ x_i \quad y_i\ ]^T \tag{3.4}$$

and $x_i,\ y_i$ are N elements long vectors representing the corrupted input samples at time instant i.

The radius R is a constant value for the CMA algorithm and this works optimally for modulations like PSKs which have the symbols spread in the complex plane over a single ring. For QAM constellations instead, which are generally composed of multiple rings, it is not possible to reduce to zero the error in Equation (3.3) but its minimization allows to compact the output symbols and yelds sufficient preconergence of the filters coefficients to apply a subsequent less robust, but more performing MMA.

The minimization of the error in Equation (3.3) is done by updating the coefficients

according to the following rules derived from a gradient analysis:

$$\begin{aligned} h^k_{pol,x} &\rightarrow h^k + \mu\varepsilon_{CMA,pol}x^*_{i-k}s_{i,pol} \\ h^k_{pol,y} &\rightarrow h^k + \mu\varepsilon_{CMA,pol}y^*_{i-k}s_{i,pol} \end{aligned} \tag{3.5}$$

where $h^k_{pol,x}$ and $h^k_{pol,y}$ stand for either h^k_{xx}, h^k_{xy}, h^k_{yx} or h^k_{yy} and denote the kth filter tap of anyone of the four FIR filters, $pol \in (x, y)$, and μ is a convergence parameter.
If the modulation of the signal requires to switch to the phase-independent MMA after CMA preconvergence the constant R must be substituted with a vector of radii $\mathbf{R} = \{R_0, \dots R_K\}$ with a number of elements equal to the number of rings of the modulation. The error equation becomes then:

$$\langle\varepsilon_{MMA,pol}\rangle = \langle R^2_{k,i} - |s_{i,pol}|^2\rangle \tag{3.6}$$

where $R_{k,i}$ represent the selected ring ($k \in (0, \dots, K)$) for the ith time instant. How to select the most convenient ring for a particular symbol is an issue that will be discussed in the practical implementation section, since there are various possible way to do it, and it is of no interest from a theoretic point of view. Except for substituting Equation (3.3) with (3.6) in the update rule (3.5), the algorithm remains the same.

3.2 Practical implementation

All the algorithms that will be considered from now on are part of the Robochameleon project, which is a coding framework and component library for simulation and experimental analysis of optical communication systems. The framework was designed to facilitate sharing code between researchers by articulating some standard methods and syntax for signal representation and function calls. Robochameleon is a project started and administered by the *Danmarks Tekniske Universitet* (DTU) but it is open to contributions from other groups, including the one in Acreo Swedish ICT AB [49].
Given the theoretic analysis realized in the previous section, the equalizer module must:

- Perform equalization;
- Compute errors;
- Update coefficients.

Moreover, to make the code more robust and adaptable in agreement with the guidelines of Robochameleon, some useful generalizations can be implemented, like:

- Matrix implementation of the convergence parameter;
- Arbitrary initialization of the FIR filters coefficients.

3.2.1 Original code

To properly describe the implementation of the core section of the module it is convenient to split the code in separate fragments, starting from the generalizations implemented to make the code more robust. In this way it is possible to introduce in the right order all the elements required to understand the code.
The first lines that will be analyzed are the ones related to the convergence coefficient:

```
% Read the input parameter mu and creates a well-formatted 2x2 matrix based
% on in.
mu = input_param.mu;
if isscalar(mu)
    mu = repmat(mu,[2 2]);
elseif isvector(mu) && numel(mu)==2
    mu = [mu(1) mu(2);...
        mu(2) mu(1)];
elseif ismatrix(mu) && size(mu)== 1*[2 2]
else
    error('Incorrect input_param.mu.');
end
```

As can be seen the convergence coefficient is passed to the module thanks to the field *mu* inside of the structure *input_param*. This structure contains a number of fields equal to the number of input parameters required to run the code and they can be scalars, vectors, matrices or strings. To avoid bugs or unexpected behaviors it is important to always verify the correctness of the data and to eventually report an error. It is moreover important to follow the Robochameleon guidelines and properly comment the code to realize a well-formatted documentation, since this will allow a possible user to know in advance how to setup correctly the module. In the fragments of the code published in this chapter almost all comments are removed, because they will be redundant with this

text and they will reduce the readability of the code.

The valid inputs for the *mu* parameter are:

- **Scalar:** In this case the resulting 2x2 matrix will be realized by setting each element of the matrix to the input value;
- **2x1 matrix:** In this case the two coefficients in input will represent respectively the convergence parameters of the main diagonal and of the anti diagonal of the resulting 2x2 matrix.
- **2x2 matrix:** In this case the resulting 2x2 matrix is equal to the input matrix. As can be seen in the code no modifications are applied to *mu* in this case, which keeps the original values.

If the input is not valid an error is printed and the run of the module is aborted.

```
% Read the input parameter H_init and create a four vectors structure which
% contains the initial coefficients of the filter based on it.
H.xx = zeros(input_param.taps,N+1);
H.xy = zeros(input_param.taps,N+1);
H.yx = zeros(input_param.taps,N+1);
H.yy = zeros(input_param.taps,N+1);
H_init = zeros(2,2,input_param.taps);
sz_H = size(input_param.H_init);
if isfloat(input_param.H_init) && isfinite(input_param.H_init)
    if isequal(sz_H,[1 1])
        H_init(:,:,ceil((input_param.taps+1)/2)) = diag([input_param.H_init
      input_param.H_init]);
    elseif isequal(sz_H,[2 1]) || isequal(sz_H,[1 2])
        H_init(:,:,ceil((input_param.taps+1)/2)) = diag(input_param.H_init);
    elseif isequal(sz_H,[2 2])
        H_init(:,:,ceil((input_param.taps+1)/2)) = input_param.H_init;
    elseif isequal(sz_H,[2 2 input_param.taps])
        H_init = input_param.H_init;
    else
        error('H_init size is incorrect');
    end
else
    error('H_init can only be numeric matrix');
end
H.xx(:,1) = squeeze(H_init(1,1,:));
H.xy(:,1) = squeeze(H_init(1,2,:));
H.yx(:,1) = squeeze(H_init(2,1,:));
H.yy(:,1) = squeeze(H_init(2,2,:));
```

As can be seen the parameters that are required this time are the number of coefficients of the FIR filters *taps*, the number of symbols to equalize N and the matrix containing the initial values of the FIR filters coefficients *H_init*. Based on the number of symbols to equalize and on the number of taps per filter it is possible to define a structure with 4 fields, one for each filter, which will contains the progressively updated filters coefficients. After memory allocation it is necessary to read the input parameter *H_init* and properly initialize the filters coefficients. The input parameter must be a finite value numerical variable of the types listed below, otherwise an error will be reported:

- **Scalar:** In this case all coefficients are initialized to 0 except for h_{xx}^0 and h_{yy}^0 which are set to the same input value.
- **2x1 Matrix:** Also in this case all coefficients are initialized to 0 except for h_{xx}^0 and h_{yy}^0 which are respectively set to the values of the 2x1 input matrix.
- **2x2 Matrix:** In this case all coefficients are initialized to 0 except for h_{xx}^0, h_{xy}^0, h_{yx}^0 and h_{yy}^0 which are respectively set to the values of the 2x2 input matrix.
- **2x2xtaps Matrix:** All coefficients are set to the corresponding values of the 2x2xtaps input matrix.

As for the convergence coefficient, if the size of the matrix does not fit the previous cases then an error is reported and the run is aborted.
At the end of the code it is possible to see how the initialized coefficients are inserted into the first block of the filter coefficients structure that will be used from now on to perform equalization and errors computation. The *squeeze* function is necessary to remove the extra dimensions of the matrix *H_init*.
The following fragment of code contains all the instructions of the main cycle, where the fundamental operations of the equalizer module are performed and also some features that were not discussed in the original paper have been included:

```
% The main section of the algorithm implements equalization, errors
% computation and update of the coefficients. It inclused also a cycle to
% perform multiple iterations over the same data and checks to avoid bad
% convergence of the coefficients. The code also realize the
% preconvergence of the coefficients with a CMA before switching to a MMA.
```

```
for iter=1:input_param.iter
    if iter>1
        H.xx(:,1) = H.xx(:,end);
        H.yx(:,1) = H.yx(:,end);
        if input_param.h_ortho||any(strcmpi(input_param.h_ortho,{'none','conv'}))
            H.xy(:,1) = H.xy(:,end);
            H.yy(:,1) = H.yy(:,end);
        elseif ~input_param.h_ortho||any(strcmpi(input_param.h_ortho,{'iter','iter+conv'}))
            H.xy(:,1) = -conj(H.yx(:,end));
            H.yy(:,1) =  conj(H.xx(:,end));
        else
            error('Bad eq h_ortho type');
        end
    end

    [Ex,~] = buffer(Ein.x,input_param.taps,input_param.taps-input_param.Nss,'nodelay');
    [Ey,~] = buffer(Ein.y,input_param.taps,input_param.taps-input_param.Nss,'nodelay');

    for n=1:N
        progress(n,N);
        Ex_est(n) = sum(H.xx(:,n).*Ex(:,n) + H.xy(:,n).*Ey(:,n));
        Ey_est(n) = sum(H.yx(:,n).*Ex(:,n) + H.yy(:,n).*Ey(:,n));
        if it>1 || n>input_param.conv
            A = abs(Ex_est(n));
            [~,i] = min(abs(R-A),[],1);
            err_x(n) = R(i)^2 - A^2;
            A = abs(Ey_est(n));
            [~,i] = min(abs(R-A),[],1);
            err_y(n) = R(i)^2 - A^2;
        else
            err_x(n) = 1 - abs(Ex_est(n))^2;
            err_y(n) = 1 - abs(Ey_est(n))^2;
        end
        H.xx(:,n+1) = H.xx(:,n)+mu(1)*err_x(n)*Ex_est(n)*conj(Ex(:,n));
        H.yx(:,n+1) = H.yx(:,n)+mu(2)*err_y(n)*Ey_est(n)*conj(Ex(:,n));
        if n==input_param.conv && any(strcmpi(input_param.h_ortho,{'conv' 'iter+conv'}))
            H.xy(:,n+1) = -conj(H.yx(:,n+1));
            H.yy(:,n+1) =  conj(H.xx(:,n+1));
        else
            H.xy(:,n+1) = H.xy(:,n)+mu(3)*err_x(n)*Ex_est(n)*conj(Ey(:,n));
            H.yy(:,n+1) = H.yy(:,n)+mu(4)*err_y(n)*Ey_est(n)*conj(Ey(:,n));
        end
    end
end
```

```
if any(isnan(Ex_est)) || any(isnan(Ey_est))
    error('Equalizer FIR filter taps did not converge.')
end
```

The first thing that must be considered is the presence of some new input parameters, in particular *iter* which must be a finite integer scalar value and represents the number of iteration the algorithm must do, *h_ortho* which must be a string and it is used to specify if forced orthogonalization of the filters coefficients is required or not, *Nss* which must be a finite integer scalar value representing the number of samples per symbol after timing recovery and *conv* which must be a finite integer scalar value representing the number of symbols that must be used for coefficients preconvergence. Moreover there is a new structure in this fragment of code called *Ein* which represent the input signal (after Deskew, Orthonormalization, Chromatic Dispersion compensation and timing recovery) with all its characteristics.

The algorithm implemented starts with a cycle that allows to iterate the code for a number of times specified by the input parameter *iter*. This is the first feature implemented in the code that was not originally proposed in the reference paper, but it allows for an increase of the performance of the module at the cost of an increased time consumption. At the beginning of each new iteration, first excluded, the coefficients in the last position of the structure H are copied in the first position as to reinitialize the filters. These values are not perfectly correct for the first symbols, since the state of polarization change in time, but they are significantly closer to the ideal ones, so CMA preconvergence is no longer necessary. This results in a decent equalization also for the first symbols, eventually reducing the overall number of errors. In the code proposed is it possible to see that the reinitialization of the filters coefficients includes also a check for forced coefficient orthogonalization.

The length of the vector of input samples required to equalize a symbol is determined by *taps*, the number of coefficients of the filters. Since each symbol is sampled with *Nss* samples the vector of input values for consecutive symbols must progressively move on by skipping the first *Nss* samples. The computation of the correct vector of input values, in the fragment of code shown, is performed thanks to the function *buffer* which returns a matrix with a number of columns equal to *taps* and a number of rows equal to the number of symbols to equalize. Thanks to this matrix it is then enough to select one by one all the columns and apply the equalization algorithm to each of them.

After all these operations it is finally possible to begin the real equalization: each iteration of the cycle equalize one symbol, compute the corresponding error and update the filters coefficients. Since the system must deal with PDM signals all the operations must be performed both for the x and y components, starting from the operations in line 28-29 which correspond to Equation (3.4). The following lines compute the errors both for CMA and MMA cases (Equations (3.3) and (3.6)). In the first case the radius of the constellation is set to 1 by default, since the signal was normalized before, while in the second case it is necessary to select the most appropriate radius in the vector $\mathbf{R}$. There are various alternatives to perform such selection and the one implemented search for the radius closest to the modulus of the symbol. This decision rule is extremely simple and works properly only if the equalization is good enough to allow a proper distinction between different rings. In the code under exam this is guaranteed by the CMA preconvergence. Finally the coefficients are updated following Equation (3.5) and eventually orthogonalized at the end of the CMA preconvergence to ensure a correct behaviour of the equalizer.

3.3 Optimization and results

Based on the work of Robert Sedgewick [50], it is possible to define software optimization as the process of modifying a software system to make some aspects of it work more efficiently or use fewer resources. In most of the cases the parameter used to compare efficiencies is the number of accesses to the memory, also because there is a direct correlation between this value and the run time of the module, as well as the global amount of memory used.
Reducing the definition above to the optimization of MATLAB scripts, there are usually two ways to perform it:

- Memory access optimization;
- MEX optimization.

The first point focus on proper definition and handling of variables and vectorization, which are means to reduce the number of accesses to the memory by exploiting the working principles of MATLAB. The second point instead differs slightly from the usual

optimization since it focuses on the capability of MATLAB to handle precompiled routines as if they are built-in functions. These routines usually do not decrease the number of accesses to the memory, nor the amount of memory used, but due to pre-compilation they reduce significantly the run time.

While a proper description of both these ways of optimization can be found in the appendix A, the physical changes made to script will be analyzed in details in the following two paragraphs, considering as parameter to optimize the run time of the single module. It is important to specify that the absolute value of the run time carries very little information, since it depends strongly on the version of MATLAB used, on the hardware of the computer and a lot of other local factors. A parameter that can then be used to express the improvements in the efficiency of the code is the *Optimization Factor* OF defined as

$$OF = \frac{T_{orig}}{T_{opt}} - 1 \tag{3.7}$$

where T_{opt} is the run time after optimization, and T_{orig} is the original run time of the routine under exam which can be obtained by running the *Profiler* module provided by MATLAB or with a proper use of the functions *tic* and *toc*.

The analysis of the OF parameter provides the following information:

- If $-1 < OF < 0$: The run time of the module after optimization is greater than the original one. It is convenient to keep the original code;
- $OF = 0$: There are no differences between the two scripts;
- $OF > 0$: The optimized code works better than the original code and the larger the OF, the better the optimization. If the OF is constant with respect to the variable used to compute the performances than the two scripts share the same limiting behaviour (maybe with different coefficients which are however not explicitly visible in the big O notation), while if the OF increases than also the limiting behaviour of the script has changed.

3.3.1 Memory access optimization

Differently from before, where it was necessary to split the code in order to introduce properly the various parameters and explain the single features of the module, to understand better how memory access optimization was performed it is more convenient now to keep the listed whole:

```
% The following code inludes all the modifications done to perform memory
% access optimization

param_numberOfTaps=obj.NumberOfTaps;

param_Ex = buffer(Ein.x,param_numberOfTaps,param_numberOfTaps-obj.Nss,'nodelay');
param_Ey = buffer(Ein.y,param_numberOfTaps,param_numberOfTaps-obj.Nss,'nodelay');

if isscalar(obj.mu)
    param_mu = repmat(obj.mu,[2 2]);
elseif isvector(obj.mu) && numel(obj.mu)==2
    param_mu = [obj.mu(1) obj.mu(2);...
       obj.mu(2) obj.mu(1)];
elseif ismatrix(obj.mu) && size(obj.mu)==1*[2 2]
    param_mu=obj.mu;
end

sz_H = size(obj.H_init);
param_H_init = zeros(2,2,param_numberOfTaps);
if isfloat(obj.H_init) && isfinite(obj.H_init)
    if isequal(sz_H,[1 1])
        param_H_init(:,:,ceil((param_numberOfTaps+1)/2)) = diag([obj.H_init obj.
      H_init]);
    elseif isequal(sz_H,[2 1]) || isequal(sz_H,[1 2])
        param_H_init(:,:,ceil((param_numberOfTaps+1)/2)) = diag(obj.H_init);
    elseif isequal(sz_H,[2 2])
        param_H_init(:,:,ceil((param_numberOfTaps+1)/2)) = obj.H_init;
    elseif isequal(sz_H,[2 2 Ntaps])
        param_H_init = obj.H_init;
    end
end

N = floor((Ein.L-param_numberOfTaps)/obj.Nss+1);
Hxx(:,1) = squeeze(param_H_init(1,1,:));
Hyx(:,1) = squeeze(param_H_init(1,2,:));
Hxy(:,1) = squeeze(param_H_init(2,1,:));
Hyy(:,1) = squeeze(param_H_init(2,2,:));
cExx=param_mu(1)*conj(param_Ex);
```

```
cEyx=param_mu(2)*conj(param_Ex);
cExy=param_mu(3)*conj(param_Ey);
cEyy=param_mu(4)*conj(param_Ey);

%% CMA / MMA
for it=1:Niter
    robolog(['Iteration ' num2str(it)], 'NFO')
    if it>1
        Hxx(:,1) = Hxx(:,end);
        Hyx(:,1) = Hyx(:,end);
        Hxy(:,1) = Hxy(:,end);
        Hyy(:,1) = Hyy(:,end);
    end

    for n=1:N
        Ex_est(n) = sum(Hxx(:,n).*param_Ex(:,n) + Hxy(:,n).*param_Ey(:,n));
        Ey_est(n) = sum(Hyx(:,n).*param_Ex(:,n) + Hyy(:,n).*param_Ey(:,n));

        if ~MMA_FLAG && n>Nconv_length, MMA_FLAG=true; end
        if MMA_FLAG
            A = abs(Ex_est(n));
            [~,i] = min(abs(param_R-A),[],1);
            err_x(n) = R2(i) - A^2;
            A = abs(Ey_est(n));
            [~,i] = min(abs(param_R-A),[],1);
            err_y(n) = R2(i) - A^2;
        else
            err_x(n) = 1 - abs(Ex_est(n))^2;
            err_y(n) = 1 - abs(Ey_est(n))^2;
        end

        Hxx(:,n+1) = Hxx(:,n)+err_x(n)*Ex_est(n)*cExx(:,n);
        Hyx(:,n+1) = Hyx(:,n)+err_y(n)*Ey_est(n)*cEyx(:,n);

        if n==Nconv_length && any(strcmpi(obj.h_ortho,{'conv' 'iter+conv'}))
            Hxy(:,n+1) = -conj(Hyx(:,n+1));
            Hyy(:,n+1) =  conj(Hxx(:,n+1));
        else
            Hxy(:,n+1) = Hxy(:,n)+err_x(n)*Ex_est(n)*cExy(:,n);
            Hyy(:,n+1) = Hyy(:,n)+err_y(n)*Ey_est(n)*cEyy(:,n);
        end
    end
end
if any(isnan(Ex_est)) || any(isnan(Ey_est))
    error('Equalizer FIR filter taps did not converge.')
end
```

The first thing that can be seen is that almost all the variables' name have changed, following the principle of the Robochameleon project regarding the readability of the code. Now almost all the parameters required for equalization are local variables which name start with the prefix *param_*. They are all defined as column vectors, as suggested in Appendix A, and they are faster to access than the fields of a structure allowing for a small, but not negligible, time saving. Another change is related to the structure containing the input parameters: differently from before the object constructor now assigns the input parameters to the corresponding properties of the object and this allows both an increased robustness of the code and the possibility of setting up default values for some of the parameters. Thanks to this change it was possible to remove the check for the correctness of the input data from the initialization of *param_mu* and *param_H_init* and even if this resulted in no major change for the run time the readability of the code is improved. The last thing about variables that is important to highlight is that some of the parameters are not defined as local variables but accessed directly as object properties because they are not called many times and it would take longer to actually create a new variable and initialize it.

Moving from variables to code optimization, it is possible to see that the buffering operation was moved outside the iteration cycle, since the input data are the same for each iteration and it is enough to compute them once for all. According to the MATLAB profiler, moreover, this function was one of the most time consuming, after equalization and coefficients update, so this change resulted in a significant amount of time saved. Outside of the cycles also four new variables were defined, called *cExx*, *cEyx*, *cExy* and *cEyy*, which accounts for the multiplication of the convergence parameter times the input data. Thanks to the buffer operation both parameters are independent of iterations and symbols position, so it is possible to compute also this multiplication once for all, thus saving time.

Inside the first cycle, accounting for different iterations, the reinitialization of the coefficient was simplified by removing the checks for forced orthogonalization. The main reason behind forced coefficients orthogonalization is to avoid the coefficients of the filters to converge on the same polarization component during the initial stages of the algorithm. After forcing this change in the coefficients this problem is solved, but the equalization errors for the symbols equalized right after usually become very high. It is pointless then to force coefficients orthogonalization more than once or in the advanced stages of the algorithm, since it is enough to do it right after CMA preconvergence. In

such a way the following cycles and iterations can smooth the increased errors and guarantee a very accurate equalization for all symbols. Inside the second cycle, accounting for symbols equalization, only the coefficients update instructions were changed, implementing the pre-computed variables *cExx*, *cEyx*, *cExy* and *cEyy*. With this last change the memory access optimization part was concluded and it was possible to move on by performing parameter optimization.
The parameters that can be optimized are:

- the number of coefficients in the filters;
- the number of symbols used for CMA preconvergence;
- the value of the convergence parameter μ;
- the number of iterations.

It is not possible to find general optimal values for these parameters, since they depend on the characteristics of the input data streams which themselves depend on the behaviour of the channel, on the residual tolerances of the previous modules and so on, but it is convenient to make some trial simulations to find appropriate values and save time. Hypothetically it should also be possible to introduce some controls and functions in order to automatically estimate and setup the best values for the parameters above, but the increased run time of the module do not justify the slight performances improvement.

3.3.2 MEX Optimization

As discussed in Appendix A, a possible way to optimize a MATLAB script is the use of MEX files. The language choose in this thesis to realize these files is C++ since it is very powerful and fast. Once again it is convenient to keep the listed whole in order to better understand how it works.

```
#include <mex.h>


int indexofSmallestElement(double* array, int size){
    int index = 0;
    for(int i = 1; i < size; i++)    {
```

```
            if(array[i] < array[index])
                index = i;
        }
        return index;
    }

    void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

        const int *dim_Ex,*dim_Ey, *dim_mu, *dim_H;
        double *Ex_re,*Ex_im,*Ey_re,*Ey_im;
        double *ris_Ex_re,*ris_Ex_im,*ris_Ey_re,*ris_Ey_im,*err_Ex,*err_Ey;
        double *mu;
        double *H_xx_re,*H_xx_im,*H_yx_re,*H_yx_im,*H_xy_re,*H_xy_im,*H_yy_re,*
          H_yy_im;
        double *H_xx_re_init,*H_xx_im_init,*H_yx_re_init,*H_yx_im_init,*H_xy_re_init
          ,*H_xy_im_init,*H_yy_re_init,*H_yy_im_init;
        double *cExx_re,*cExx_im, *cEyx_re, *cEyx_im, *cExy_re,*cExy_im, *cEyy_re, *
          cEyy_im;
        double *R;
        double *diff;
        int h_ortho_read;
        int Niter;
        int MMA_FLAG,CMA_conv;
        int ad_mu_FLAG;
        int row,col,radii,h_ortho_length;
        double A;
        int i_min;

        dim_Ex = mxGetDimensions(prhs[0]);
        row = mxGetN(prhs[0]);
        col = mxGetM(prhs[0]);
        dim_Ey = mxGetDimensions(prhs[1]);
        dim_mu = mxGetDimensions(prhs[2]);
        dim_H  = mxGetDimensions(prhs[3]);
        radii = mxGetM(prhs[9]);
        h_ortho_length = mxGetN(prhs[11]);

        if ((dim_Ex[1] != dim_Ey[1]) || (dim_Ex[0] != dim_Ey[0])) mexErrMsgTxt("The
          length of the arrays are different");
        if (dim_mu[1] != 2 && dim_mu[0] != 2)mexErrMsgTxt("Mu must be a 2x2 matrix");

        plhs[0] = mxCreateDoubleMatrix(row, 1, mxCOMPLEX);
        plhs[1] = mxCreateDoubleMatrix(row, 1, mxCOMPLEX);
        plhs[2] = mxCreateDoubleMatrix(row+1, 1, mxREAL);
        plhs[3] = mxCreateDoubleMatrix(row+1, 1, mxREAL);

        Ex_re = (double*) mxGetData(prhs[0]);
        Ey_re = (double*) mxGetData(prhs[1]);
```

```
    Ex_im = (double*) mxGetImagData(prhs[0]);
    Ey_im = (double*) mxGetImagData(prhs[1]);

    mu = (double*) mxGetData(prhs[2]);

    H_xx_re_init = (double*) mxGetData(prhs[3]);
    H_xx_im_init = (double*) mxGetImagData(prhs[3]);
    H_yx_re_init = (double*) mxGetData(prhs[4]);
    H_yx_im_init = (double*) mxGetImagData(prhs[4]);
    H_xy_re_init = (double*) mxGetData(prhs[5]);
    H_xy_im_init = (double*) mxGetImagData(prhs[5]);
    H_yy_re_init = (double*) mxGetData(prhs[6]);
    H_yy_im_init = (double*) mxGetImagData(prhs[6]);

    Niter = (int) mxGetScalar(prhs[7]);

    MMA_FLAG = 0;
    CMA_conv = (int) mxGetScalar(prhs[8]);

    R = (double*) mxGetData(prhs[9]);

    ad_mu_FLAG = (int) mxGetScalar(prhs[10]);

    h_ortho_read = (int) mxGetScalar(prhs[11]);

    ris_Ex_re= mxGetPr(plhs[0]);
    ris_Ey_re= mxGetPr(plhs[1]);
    ris_Ex_im= mxGetPi(plhs[0]);
    ris_Ey_im= mxGetPi(plhs[1]);
    cExx_re=(double*) mxMalloc(row*col*sizeof(double));
    cEyx_re=(double*) mxMalloc(row*col*sizeof(double));
    cExy_re=(double*) mxMalloc(row*col*sizeof(double));
    cEyy_re=(double*) mxMalloc(row*col*sizeof(double));
    cExx_im=(double*) mxMalloc(row*col*sizeof(double));
    cEyx_im=(double*) mxMalloc(row*col*sizeof(double));
    cExy_im=(double*) mxMalloc(row*col*sizeof(double));
    cEyy_im=(double*) mxMalloc(row*col*sizeof(double));
    err_Ex= mxGetPr(plhs[2]);
    err_Ey= mxGetPr(plhs[3]);
    H_xx_re = (double*) mxMalloc((row+1)*col*sizeof(double));
    H_xx_im = (double*) mxMalloc((row+1)*col*sizeof(double));
    H_yx_re = (double*) mxMalloc((row+1)*col*sizeof(double));
    H_yx_im = (double*) mxMalloc((row+1)*col*sizeof(double));
    H_xy_re = (double*) mxMalloc((row+1)*col*sizeof(double));
    H_xy_im = (double*) mxMalloc((row+1)*col*sizeof(double));
    H_yy_re = (double*) mxMalloc((row+1)*col*sizeof(double));
    H_yy_im = (double*) mxMalloc((row+1)*col*sizeof(double));
    diff = (double*) mxMalloc(radii*sizeof(double));
```

```
    for (int j=0;j<col*row;j++){
        cExx_re[j]=mu[0]*Ex_re[j];
        cExx_im[j]=mu[0]*(-Ex_im[j]);
        cEyx_re[j]=mu[2]*Ex_re[j];
        cEyx_im[j]=mu[2]*(-Ex_im[j]);
        cExy_re[j]=mu[1]*Ey_re[j];
        cExy_im[j]=mu[1]*(-Ey_im[j]);
        cEyy_re[j]=mu[3]*Ey_re[j];
        cEyy_im[j]=mu[3]*(-Ey_im[j]);
    }

   for (int j=0;j<col;j++){
       H_xx_re[j]=H_xx_re_init[j];
       H_yx_re[j]=H_yx_re_init[j];
       H_xy_re[j]=H_xy_re_init[j];
       H_yy_re[j]=H_yy_re_init[j];
       H_xx_im[j]=H_xx_im_init[j];
       H_yx_im[j]=H_yx_im_init[j];
       H_xy_im[j]=H_xy_im_init[j];
       H_yy_im[j]=H_yy_im_init[j];
    }

    for (int it=0; it<Niter; it++){
        for (int j=0;j<row;j++){
            ris_Ex_re[j]=0;
            ris_Ex_im[j]=0;
            ris_Ey_re[j]=0;
            ris_Ey_im[j]=0;
        }
        if (it>0){
            for (int j=0;j<col; j++){
                H_xx_re[j]=H_xx_re[(row)*col+j];
                H_xx_im[j]=H_xx_im[(row)*col+j];
                H_yx_re[j]=H_yx_re[(row)*col+j];
                H_yx_im[j]=H_yx_im[(row)*col+j];
                H_xy_re[j]=H_xy_re[(row)*col+j];
                H_xy_im[j]=H_xy_im[(row)*col+j];
                H_yy_re[j]=H_yy_re[(row)*col+j];
                H_yy_im[j]=H_yy_im[(row)*col+j];
            }
        }

        for (int n_sym=0; n_sym<row; n_sym++){
            for (int taps=0; taps<col; taps++){
                ris_Ex_re[n_sym]+=(H_xx_re[n_sym*col+taps]*Ex_re[n_sym*col+taps]-
   H_xx_im[n_sym*col+taps]*Ex_im[n_sym*col+taps])+(H_xy_re[n_sym*col+taps]*
   Ey_re[n_sym*col+taps]-H_xy_im[n_sym*col+taps]*Ey_im[n_sym*col+taps]);
```

```
145                 ris_Ey_re[n_sym]+=(H_yx_re[n_sym*col+taps]*Ex_re[n_sym*col+taps]-
        H_yx_im[n_sym*col+taps]*Ex_im[n_sym*col+taps])+(H_yy_re[n_sym*col+taps]*
        Ey_re[n_sym*col+taps]-H_yy_im[n_sym*col+taps]*Ey_im[n_sym*col+taps]);
                    ris_Ex_im[n_sym]+=(H_xx_im[n_sym*col+taps]*Ex_re[n_sym*col+taps]+
        H_xx_re[n_sym*col+taps]*Ex_im[n_sym*col+taps])+(H_xy_im[n_sym*col+taps]*
        Ey_re[n_sym*col+taps]+H_xy_re[n_sym*col+taps]*Ey_im[n_sym*col+taps]);
                    ris_Ey_im[n_sym]+=(H_yx_im[n_sym*col+taps]*Ex_re[n_sym*col+taps]+
        H_yx_re[n_sym*col+taps]*Ex_im[n_sym*col+taps])+(H_yy_im[n_sym*col+taps]*
        Ey_re[n_sym*col+taps]+H_yy_re[n_sym*col+taps]*Ey_im[n_sym*col+taps]);
                }
                if (it==0 && n_sym==CMA_conv){MMA_FLAG=1;}
150             if (MMA_FLAG){
                    A=pow(pow(ris_Ex_re[n_sym],2)+pow(ris_Ex_im[n_sym],2),0.5);
                    for (int j=0;j<radii;j++){
                        diff[j]=abs(R[j]-A);
                    }
155                 i_min=indexofSmallestElement(diff, radii);
                    err_Ex[n_sym]=pow(R[i_min],2)-pow(A,2);

                    A=pow(pow(ris_Ey_re[n_sym],2)+pow(ris_Ey_im[n_sym],2),0.5);
                    for (int j=0;j<radii;j++){
160                     diff[j]=abs(R[j]-A);
                    }
                    i_min=indexofSmallestElement(diff, radii);
                    err_Ey[n_sym]=pow(R[i_min],2)-pow(A,2);
                }
165             else {
                    A=pow(ris_Ex_re[n_sym],2)+pow(ris_Ex_im[n_sym],2);
                    err_Ex[n_sym]=1-A;
                    A=pow(ris_Ey_re[n_sym],2)+pow(ris_Ey_im[n_sym],2);
                    err_Ey[n_sym]=1-A;
170             }

                if (it==1 && n_sym==CMA_conv && h_ortho_read){
                    for (int taps=0; taps<col; taps++){
                        H_xx_re[(n_sym+1)*col+taps]=H_xx_re[n_sym*col+taps];
175                     H_xx_im[(n_sym+1)*col+taps]=H_xx_im[n_sym*col+taps];
                        H_yx_re[(n_sym+1)*col+taps]=H_yx_re[n_sym*col+taps];
                        H_yx_im[(n_sym+1)*col+taps]=H_yx_im[n_sym*col+taps];
                        H_xy_re[(n_sym+1)*col+taps]=H_yx_re[n_sym*col+taps];
                        H_xy_im[(n_sym+1)*col+taps]=-H_yx_im[n_sym*col+taps];
180                     H_yy_re[(n_sym+1)*col+taps]=H_xx_re[n_sym*col+taps];
                        H_yy_im[(n_sym+1)*col+taps]=-H_xx_im[n_sym*col+taps];
                    }
                    mexPrintf("ORTHOGONALIZATION APPLIED\n");
                }
185             else{
                    for (int taps=0; taps<col; taps++){
```

```
                    H_xx_re[(n_sym+1)*col+taps]=H_xx_re[n_sym*col+taps]+err_Ex[
        n_sym]*(ris_Ex_re[n_sym]*cExx_re[n_sym*col+taps]-ris_Ex_im[n_sym]*cExx_im[
        n_sym*col+taps]);
                    H_xx_im[(n_sym+1)*col+taps]=H_xx_im[n_sym*col+taps]+err_Ex[
        n_sym]*(ris_Ex_im[n_sym]*cExx_re[n_sym*col+taps]+ris_Ex_re[n_sym]*cExx_im[
        n_sym*col+taps]);
                    H_yx_re[(n_sym+1)*col+taps]=H_yx_re[n_sym*col+taps]+err_Ey[
        n_sym]*(ris_Ey_re[n_sym]*cEyx_re[n_sym*col+taps]-ris_Ey_im[n_sym]*cEyx_im[
        n_sym*col+taps]);
                    H_yx_im[(n_sym+1)*col+taps]=H_yx_im[n_sym*col+taps]+err_Ey[
        n_sym]*(ris_Ey_im[n_sym]*cEyx_re[n_sym*col+taps]+ris_Ey_re[n_sym]*cEyx_im[
        n_sym*col+taps]);
                    H_xy_re[(n_sym+1)*col+taps]=H_xy_re[n_sym*col+taps]+err_Ex[
        n_sym]*(ris_Ex_re[n_sym]*cExy_re[n_sym*col+taps]-ris_Ex_im[n_sym]*cExy_im[
        n_sym*col+taps]);
                    H_xy_im[(n_sym+1)*col+taps]=H_xy_im[n_sym*col+taps]+err_Ex[
        n_sym]*(ris_Ex_im[n_sym]*cExy_re[n_sym*col+taps]+ris_Ex_re[n_sym]*cExy_im[
        n_sym*col+taps]);
                    H_yy_re[(n_sym+1)*col+taps]=H_yy_re[n_sym*col+taps]+err_Ey[
        n_sym]*(ris_Ey_re[n_sym]*cEyy_re[n_sym*col+taps]-ris_Ey_im[n_sym]*cEyy_im[
        n_sym*col+taps]);
                    H_yy_im[(n_sym+1)*col+taps]=H_yy_im[n_sym*col+taps]+err_Ey[
        n_sym]*(ris_Ey_im[n_sym]*cEyy_re[n_sym*col+taps]+ris_Ey_re[n_sym]*cEyy_im[
        n_sym*col+taps]);
                }
            }
        }
    }
    mxSetData(plhs[0],ris_Ex_re);
    mxSetImagData(plhs[0],ris_Ex_im);
    mxSetData(plhs[1],ris_Ey_re);
    mxSetImagData(plhs[1],ris_Ey_im);
    mxSetM(plhs[0],row);
    mxSetN(plhs[0],1);
    mxSetM(plhs[1],row);
    mxSetN(plhs[1],1);
    mxSetData(plhs[2],err_Ex);
    mxSetData(plhs[3],err_Ey);
    mxSetM(plhs[2],row+1);
    mxSetM(plhs[3],row+1);
    mxSetN(plhs[2],1);
    mxSetN(plhs[3],1);
    mexPrintf("Outputs and errors assigned - Equalization complete\n");
    return;
}
```

To avoid pointless wastes of memory, only the fundamental library *mex.h* was included, partially limiting the capability of the code to deal with strings and complex mathematics. Both these problems however are dealt with by implementing manually the complex multiplication and exponentiation operations and by checking the string values in MATLAB before calling the function.

As can be seen, the two cycles previously realized in MATLAB are here written in C++ without any semantic change. The only main difference is that here real and imaginary part of input symbols, output symbols ad filters coefficients are separated into different vectors of double.

To deal with MEX files properly a significant number of instructions before and after the real routine are necessary, principally to allocate the memory required for the correct execution of the script and to make the output results available to MATLAB. It is also important to pay attention to memory leaks and segmentation faults which can create a lot of problems within complete simulations or practical experiments. A complete description of both these problems can be found in Appendix A.

The following few lines show the main cycle of the equalizer after MEX optimization:

```
%Main cycle of the equalizer after the MEX optimization

if strcmp(obj.h_ortho,'iter')
    obj.h_ortho=1;
else
    obj.h_ortho=0;
end

[Exout, Eyout, err_x, err_y]=EqualizerMEX(param_Ex,param_Ey,param_mu,squeeze(
      param_H_init(1,1,:)),squeeze(param_H_init(2,1,:)),squeeze(param_H_init
      (1,2,:)),squeeze(param_H_init(2,2,:)),obj.iter,obj.conv_length,param_R,obj.
      adaptive_mu,obj.h_ortho);
```

As can be seen there are no more cycles and everything is performed by the MEX function. To avoid the use of strings the parameter *h_ortho* has been converted in a boolean flag passed to the script as a normal input parameter.

3.3.3 Overall results

The results of optimization, computed with the aid of the MATLAB profiler, can be found in Figure (3.1). As can be seen the behaviour of the module in all three cases is linear with respect to the signal length, showing that the asymptotic behaviour of the script has not changed with optimization. This can also be seen by computing the Optimization Factor with the formula reported in Equation (3.7) for different signal lengths, obtaining the results reported in Table (3.1).

It is possible to see that except for some small variations between the OFs the values

Signal length [bit]	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Memory Access Optimization	0.54	0.40	0.41	0.43	0.43	0.43
MEX script	14.56	14.98	13.11	16.90	14.07	11.87

TABLE 3.1: Optimization Factors for the equalizer module

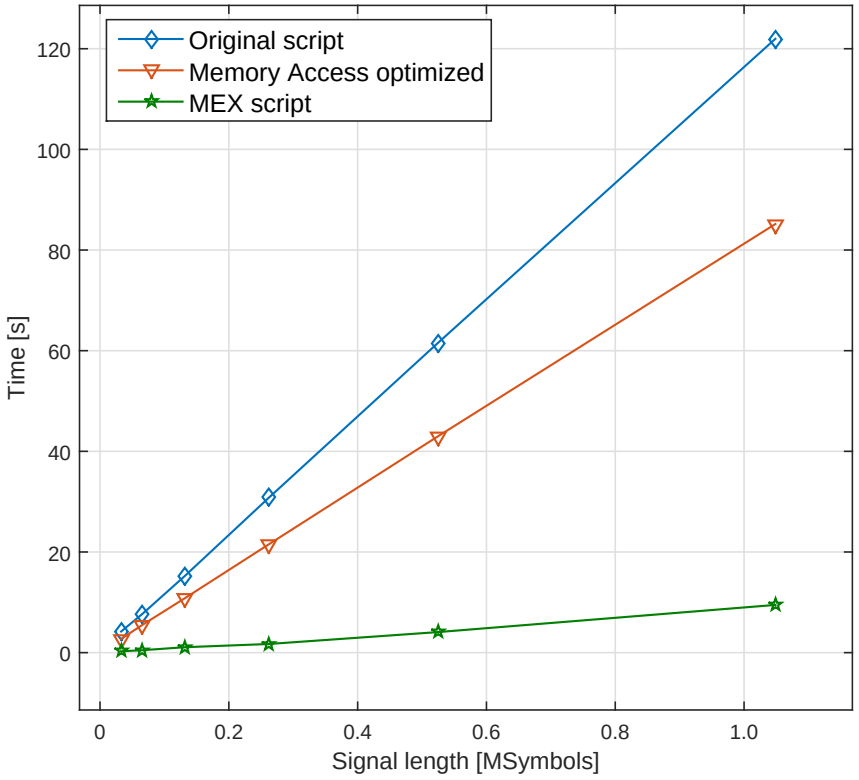

FIGURE 3.1: Time comparison between original script, memory access optimized script and MEX realized script.

Original Code	Memory Access Optimization	MEX script
0	0.44	14.25

TABLE 3.2: Average optimization factors for the equalizer module

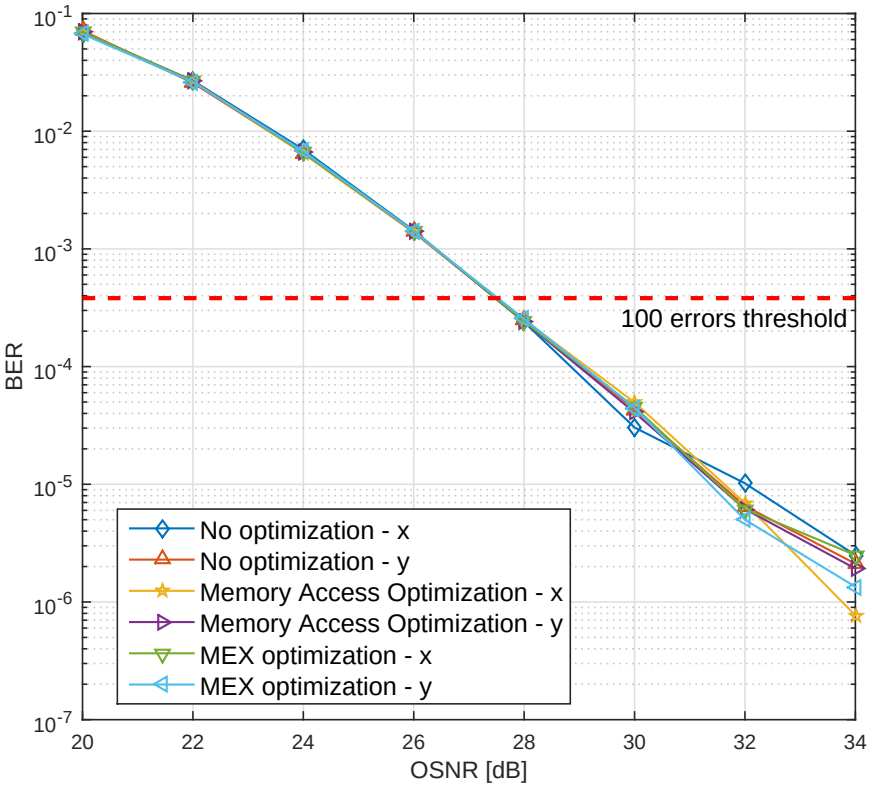

FIGURE 3.2: Performances of the equalizer module after various step of optimization.

remains almost the same for all lengths. The OFs for the MEX script has a larger variance with respect to the memory access optimization case principally due to enhancing effect that Equation (3.7) has on the small random variations in the time required to call the function and build the running environment. The most important thing to remark, however, is that on average the memory access optimization has a marginal effect on the run time, as can be seen in the values reported in Table (3.2), while the passage to MEX scripts allows for a significant improvement.

The most important thing to verify after optimization is if the performances of the module have changed with respect to the original script. This can be checked by running a proper number of repeated simulations to obtain some statistically meaningful data. In the case proposed a signal with length of 2^{18} bit is modulated at a baud rate of

50 GSymbol/s with a 16 QAM scheme, transmitted over a dispersive 80 km-long channel, followed by an EDFA with noise figure of 5 dB and resampled, on receiver side, with 2 sample/symbol. The channel is considered linear, with a first order chromatic dispersion coefficient equal to 17 ps/(ns km) and a PMD coefficient equal to 0.08 ps/$\sqrt{km}$. The simulations are done by varying the OSNR of the transmitted signal in the range from 20 to 34 dB with a step of 2 dB. For each value of OSNR the simulations are repeated 5 times and the averaged results are shown in Figure (3.2). As can be seen the performances of the equalizer have not changed with optimization and the curves are almost completely overlapped. Due to the finite length of the data and to the relatively small number of repetitions of the simulation, for high values of OSNR it is fairly hard to get errors in the transmission. The statistical description of the errors for these values of OSNR is then not accurate and this results in an increased variance. Due to all these reasons a threshold line is added in correspondence to the BER measured after 100 errors, to graphically show the data interval that can be considered meaningful.

Chapter 4

Carrier Phase Recovery

4.1 Theoretic background

Before analyzing the algorithms implemented and the optimizations performed on the *Carrier Phase Recovery* (CPR) module, it is convenient to highlight and understand the impairments that must be corrected as well as their causes. As introduced in Section 2.2.6 the CPR module has to compensate for two principal phenomena, frequency offset and phase noise, which will be theoretically analyzed in the following sections. Successively the work of T. Pfau et al. [39] on the developing of an hardware efficient scheme for CPR will be analyzed and finally the process of optimization of the algorithm implemented will be shown.

4.1.1 Frequency offset

To understand the causes and the effects of frequency offset it is necessary to introduce more in details the working principle of a phase diverse intradyne receiver [2]. To simplify the explanation it is convenient initially to consider a configuration without phase diversity, where there is no polarization demultiplexing and the states of polarization of the received signal and of the local oscillator are aligned thanks to a polarization controller. The graphical scheme is shown in Figure (4.1).
It is possible to describe the complex electric field of the transmitted optical signal as:

$$E_s(t) = A_s(t)e^{j\omega_s t} \tag{4.1}$$

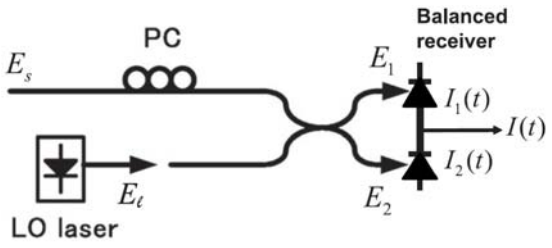

FIGURE 4.1: Configuration of a simplified coherent receiver.

where $A_s(t)$ is the complex amplitude and ω_s is the transmitting laser angular frequency. Similarly the complex electric filed of the continuous wave LO can be written as:

$$E_l(t) = A_l e^{j\omega_l t} \tag{4.2}$$

where A_l is the constant complex amplitude and ω_l is the LO angular frequency.
The powers of the two waves can be defined as:

$$\begin{aligned} P_s(t) &= k\left|A_s(t)\right|^2/2, \\ P_l &= k\left|A_l\right|^2/2. \end{aligned} \tag{4.3}$$

where the constant $k = \frac{S_eff}{\zeta}$ represent the ratio between the effective beam area and the impedance of the free space.
The two waves are then coupled and detected with a balanced photodiode that allows to suppress the DC component and maximize the beat between the signal and the LO. The key is to use a coupler which split the powers 50/50 on the two outputs while introducing a 180° phase shift to either the signal or the LO electric field. The output fields from the coupler can be written as:

$$\begin{aligned} E_1(t) &= \frac{1}{\sqrt{2}}\left(E_s(t) + E_l\right) \\ E_2(t) &= \frac{1}{\sqrt{2}}\left(E_s(t) - E_l\right) \end{aligned} \tag{4.4}$$

which generate the following photocurrents on the photodiodes realizing the balanced photodiode:

$$\begin{aligned} I_1(t) &= \frac{R}{2}\left[P_s(t) + P_l + 2\sqrt{P_s(t)P_l}\cos\left\{\omega_{IF}t + \theta_s(t) + \theta_l(t)\right\}\right] \\ I_2(t) &= \frac{R}{2}\left[P_s(t) + P_l - 2\sqrt{P_s(t)P_l}\cos\left\{\omega_{IF}t + \theta_s(t) + \theta_l(t)\right\}\right] \end{aligned} \tag{4.5}$$

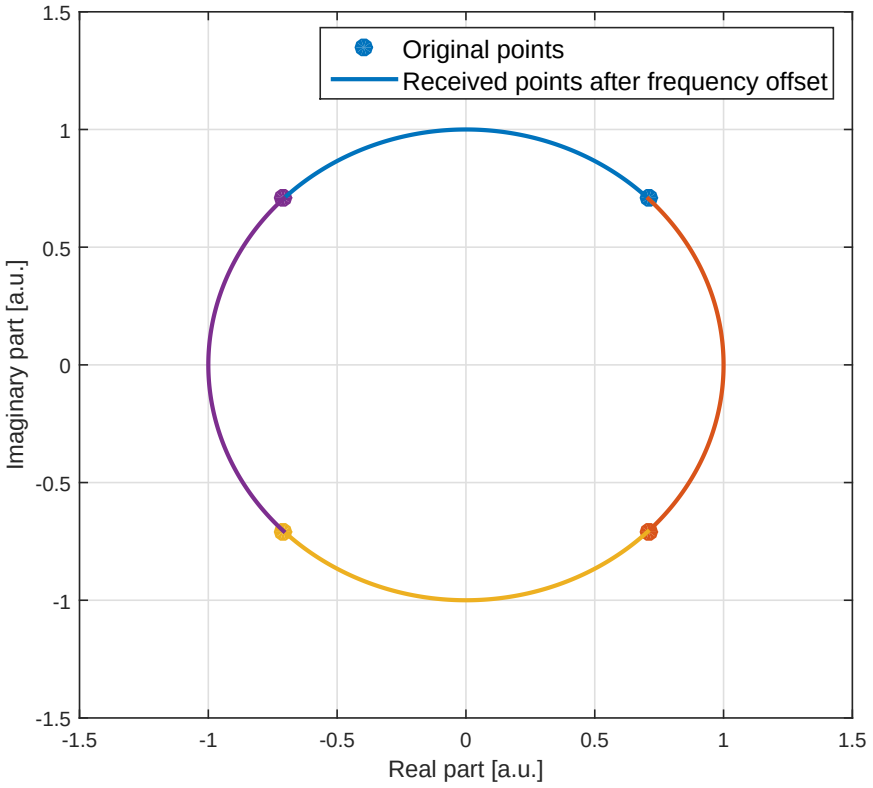

FIGURE 4.2: Effects of linear frequency offset on the received signal

with $\omega_{IF} = |\omega_s - \omega_l|$ defined as residual intermediate frequency, $\theta_s(t)$ and $\theta_l(t)$ time varying phases respectively of the transmitted signal and LO and R responsivity of the photodiodes.

The output of the balanced photodiode is given by:

$$I(t) = I_1(t) - I_2(t) = 2R\sqrt{P_s(t)P_l} \cdot \cos\{\omega_{IF}t + \theta_{sig}(t) - \theta_l(t)\} \tag{4.6}$$

where P_l is a constant and the phases includes only the random time-varying phase noises which will be discusses in the following sections.

It is important to specify that the photocurrent given by Equation (4.6) is proportional to $\sqrt{P_l}$ so, as the LO power increases a gain for the photocurrent can be obtained. The signal level can then be enhanced to overwhelm the thermal noise of the receiver obtaining an SNR, at receiver side, approaching the quantum-noise limit.

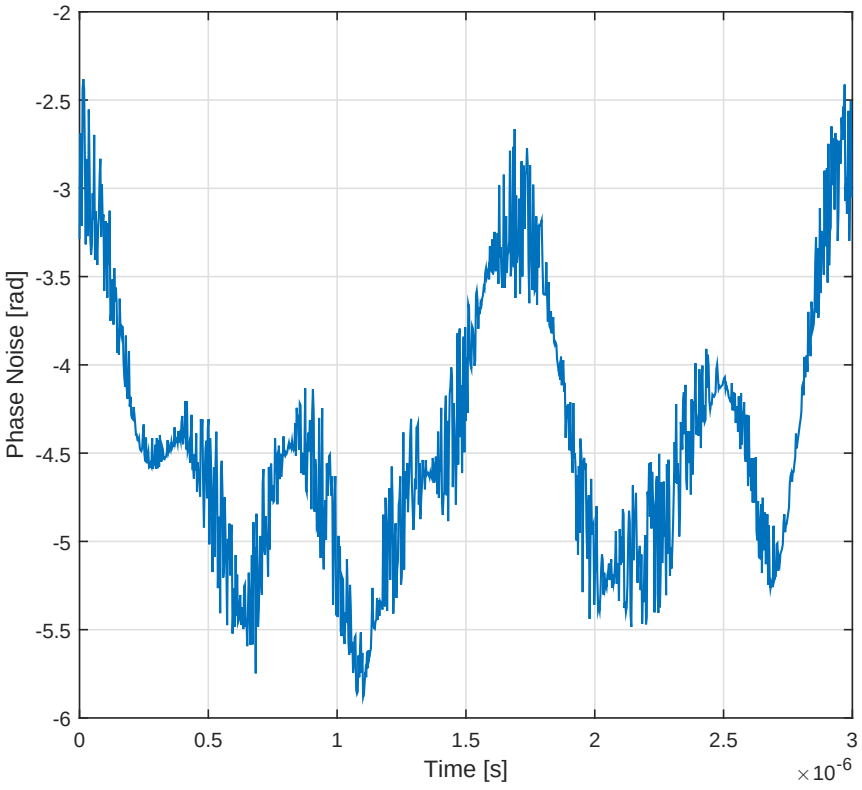

FIGURE 4.3: Example of the phase noise for a DFB laser with linewidth of approximatively 1 MHz

Considering now also the phase diversity avoided before, relations similar to Equation (4.6) can be found for both in-phase and quadrature components:

$$\begin{aligned} I_I(t) &= R\sqrt{P_s(t)P_l} \cdot \cos\{\omega_{IF}t + \theta_{sig}(t) - \theta_l(t)\} \\ I_Q(t) &= R\sqrt{P_s(t)P_l} \cdot \sin\{\omega_{IF}t + \theta_{sig}(t) - \theta_l(t)\} \end{aligned} \tag{4.7}$$

Homodyne receivers are such that $\omega_{IF} = 0$, which means that the transmitter and the LO lasers are both centered at the same frequency. Unfortunately this is not possible in reality, principally due to unavoidable non-idealities or environmental conditions, so ω_{IF} usually results relatively small (in the MHz band) but different from 0 and the receiver scheme takes the name of Intradyne.

The results of the residual ω_{IF} can be seen in Figure (4.2) where, at successive time instants, the received symbols move linearly along the unitary circle.

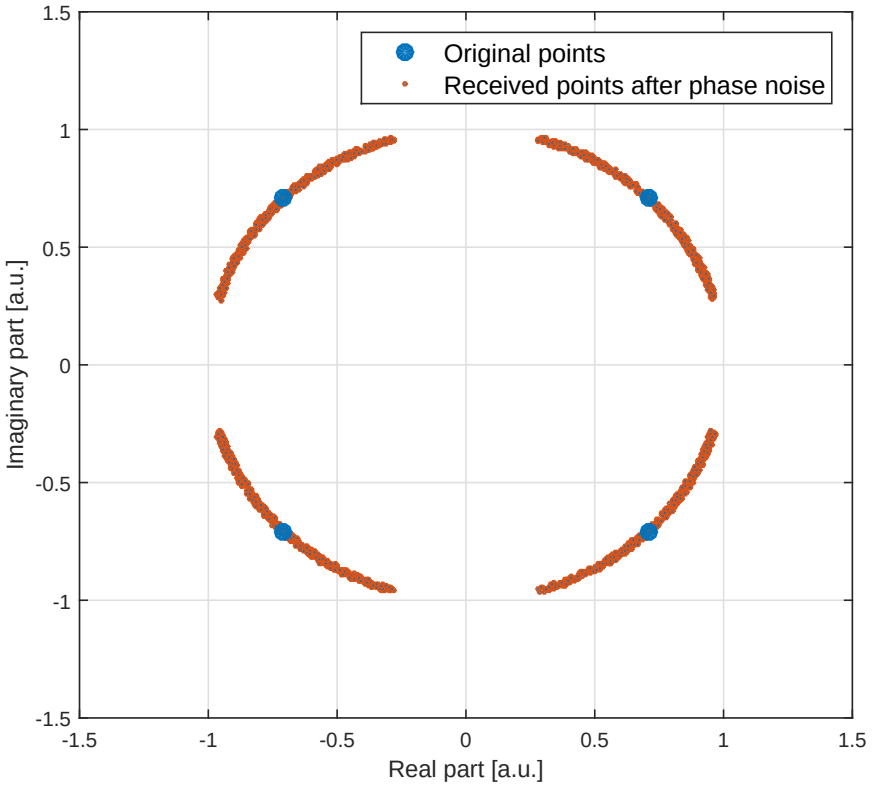

FIGURE 4.4: Effects of phase noise on the received signal

4.1.2 Phase noise

In this section only single-frequency lasers, where essentially all power is in a single resonator mode, will be considered since these are the ones used to date in high speed coherent optical systems.

A single-frequency laser will not exhibit a perfect sinusoidal oscillation of the electric field at its outputs due to fluctuations of the power and variations of the optical phase [51, 52]. The variations of the optical phase can be quantified with a very complicate phase noise *Power Spectral Density* (PSD) having units of [rad^2/Hz].

The fundamental origin of phase noise is quantum noise, in particular due to spontaneous emission of the gain medium into the resonator modes, but also optical losses can contribute to it. In addition, there can be influences due to vibrations of the cavity mirrors or to temperature fluctuations and, in many cases, there is also a coupling of intensity noise to phase noise due to the nonlinearities of the active medium. All these things cause a finite value of the linewidth of the laser, which is defined as the width of the main peak in the power spectral density of the optical field.

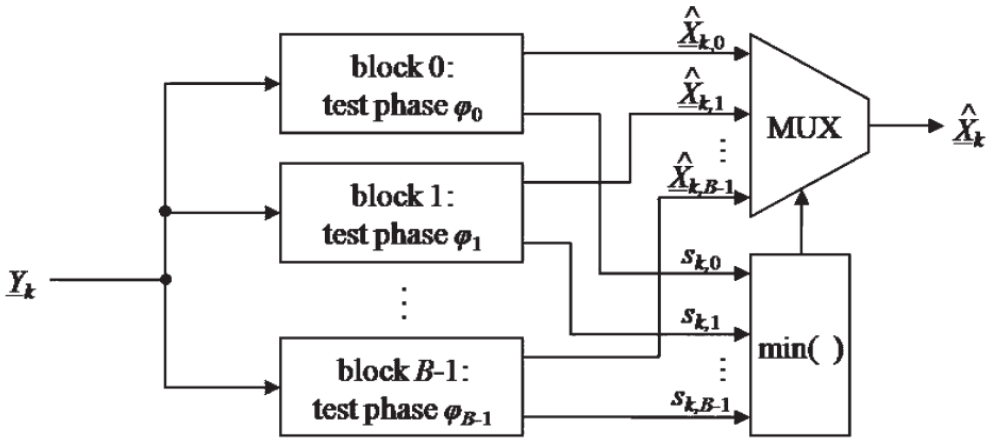

FIGURE 4.5: Feedforward carrier recovery using B test phase values φ_b

An example of the overall behaviour of the phase noise for a common *Distributed Feedback* (DFB) laser is shown in Figure (4.3) where it is possible to see a strong, low frequency, noise component which dominates the behaviour, overlapped with a small but rapidly changing one. Essentially the linewidth is determined by low-frequency noise which is the main component that must be eliminated. The effect of the phase noise on the received symbols can be seen in Figure (4.4) where also a little component of intensity noise is visible.

4.1.3 Proposed algorithm

The algorithm that is going to be presented here is based on the work of Timo Pfau et al. [39]. It presents a digital feedforward recovery algorithm for arbitrary M-QAM constellations in an intradyne coherent optical receiver. The principal advantages of the algorithm proposed are that it does not contain any feedback loop, which makes it highly tolerant against laser phase noise, and that it is possible to realize it in an hardware efficient way.

The block diagram of the proposed carrier recovery module is shown in Figure (4.5) and a more detailed structure of the test phase block is shown in Figure (4.6). In order to properly describe how the algorithm works, perfect timing recovery and equalization are assumed.

To recover the carrier phase in a pure feedforward approach the received signal in output

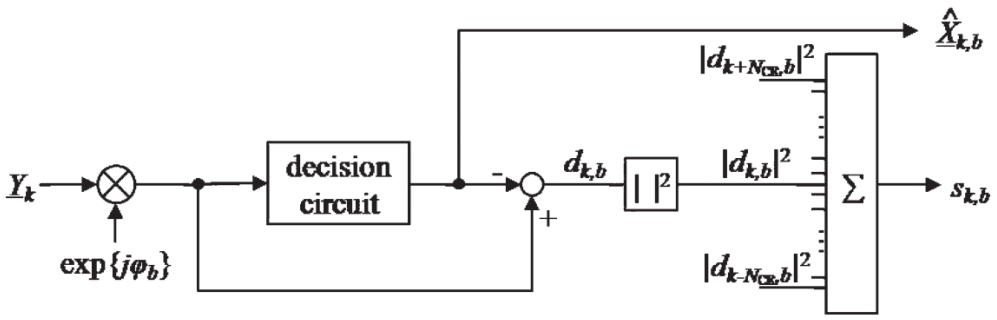

FIGURE 4.6: Insight of the test phase block for the feedforward carrier recovery diagram

from the equalizer Y_k is rotated by B test carrier phase angles φ_b with:

$$\varphi_b = \frac{b}{B} \cdot \frac{\pi}{2}, \qquad b \in \{0, 1, \ldots, B-1\}\,. \tag{4.8}$$

All rotated symbols are fed into a decision circuit and the squared distance:

$$|d_{k,b}|^2 = \left|Y_k e^{j\varphi_b} - \left\lfloor Y_k e^{j\varphi_b} \right\rfloor_D\right|^2 \tag{4.9}$$

to the closest constellation point $\left\lfloor Y_k e^{j\varphi_b} \right\rfloor_D = \hat{X}_{k,b}$ is calculated in the complex plane. In order to remove noise distortions, the distances of $2N+1$ consecutive test symbols rotated by the same carrier phase angle ϕ_B are summed up:

$$s_{k,b} = \sum_{n=-N}^{N} |d_{k-n,b}|^2\,. \tag{4.10}$$

The optimum value of the filter half width N depends on the laser linewidth times symbol rate product. Heuristically it is possible to show that $N = 6, \ldots, 10$ are all fairly good choices. After filtering, the optimum phase angles is determined by searching the minimum sum of distance values.

The proposed feedforward carrier recovery algorithm is optimized for square QAM constellations, which are the more common, but it can be generalized for arbitrary ones: if the constellation diagram is rotationally symmetric by the angle γ then φ_b must be selected as:

$$\varphi_b = \frac{b}{B} \cdot \gamma, \qquad b \in \{0, 1, \ldots, B-1\}\,. \tag{4.11}$$

For constellations without rotational symmetry $\gamma = 2\pi$ must be used.

If polarization division multiplexing is applied, there are two possibilities to implement the carrier recovery: one is to use two separate carrier recoveries for each polarization,

while the second is to have a common carrier recovery for both polarizations. To realize the second option Equation (4.10) must be modified as follows:

$$s_{k,b} = \sum_{p=1}^{2} \sum_{n=-N}^{N} \left| d_{p,k-n,b} \right|^2 . \tag{4.12}$$

where p is the index of the polarization component. Because twice as much data are available to determine the carrier phase angle, N can be halved, increasing the phase noise tolerance roughly by a factor of 2.

As can be deduced from what stated so far, no hypotheses or requirements have been made regarding the modules preceding the CPR, so the proposed algorithm is compatible with any kind of equalizer, CDC module, and timing recovery algorithm. In the last sections of the paper published by Pfau is it possible to find a complete description about how to hardware efficiently implement this algorithm, but since this is out of the scope of the thesis it will not be reported here.

4.2 Practical implementation

Considering the theoretical analysis done in the previous section, the CPR module must

- Rotate the symbol and identify the closest constellation point;
- Compute the squared distance;
- Filter $2N + 1$ symbols to remove noise distortions;
- Determine the phase angle by searching for the minimum output of the filter.

It is important to highlight that in the CPR algorithm proposed the linear frequency offset was not discussed. This is because the algorithm does not discriminate between the linear and the randomly time-varying phase noise and try to equalize them as a whole. It is definitely convenient however to remove the linear offset beforehand if possible, because this will reduce the total amount of phase the algorithm has to track, increasing the overall performances and robustness.

4.2.1 Original code

As in the previous chapter, to introduce the original code it is convenient to split it into smaller sections and analyze separately the operations implemented. Once again, the comments are removed from the listed instructions to improve the readability of the code.

```
% Read the input parameters to initialize the variables.
constellation = input_param.constellation;
R = uniquetol(abs(constellation), 1e-8);
B = input_param.test_angles;
b = (0:B-1)';
Gamma = pi/2;
phi_b = (b/(B-1))*Gamma-pi/4;
N = input_param.N;
```

The first section refers to the initialization of the variables described in the theoretical analysis. The fundamental ones, i.e. the constellation pattern, B and N are read from the structure *input_param* and from them the different test angles φ_b are computed according to Equation (4.8). In this implementation an angle of $\pi/4$ is subtracted to the angles to center them around 0.

The parameters B and N must be finite integer values, while *constellation* is a vector of finite complex numbers with dimensions 1 by M or M by 1. Since in the original code no controls on the input data were performed, if wrong variable types are used the simulation will crash without providing useful information about the error. This results in an increased difficulty in backtracking the issue and it is against the programming paradigms of Robochameleon.

The following section deals with linear frequency offset estimation and removal.

```
% Perform frequency offset estimation and compensation

f = (0:size(Ein.x,1)-1)'/Fb;
[Ex_fr,freq_off] = CPR.FOC(Ein.x,R,Fb);
Ey_fr = Ein.y.*exp(-2i*pi*f*freq_off);

...

function [Eout,est_f_off]= FOC(Ein,R,Fb)

    C13 = zeros(N,1);
```

```
    Ph_rot = exp(1i*pi/4);
    Ph_rec = exp(-1i*pi/4);
    Ein_rot = Ein*Ph_rot;
    N = size(Ein_rot,1);
    T = 1/Fb;
    f = (0:N-1)'/Fb;
    for n=1:N
        A = abs(Ein_rot(n));
        [~,i] = min(abs(R-A),[],1);
        if i==1 || i==3
            C13(n) =  Ein_rot(n);
        end
    end
    % Select non-zero indices
    ind1 = find(C13);
    % Class 1 and 3 symbols
    Sym_13 = C13(ind1);
    % 4-fold and phase tracking
    Theta_est = angle((Sym_13./abs(Sym_13)).^4);
    Theta_est = unwrap(Theta_est)/4;
    % frequency offset estimation and recovery compensation
    Theta_est_shift = circshift(Theta_est,1);
    ind_shift = circshift(ind1,1);
    delt_Theta = Theta_est(2:end)-Theta_est_shift(2:end);
    delt_ind = ind1(2:end)-ind_shift(2:end);
    Average_delta_Theta = mean(delt_Theta./delt_ind);
    est_f_off = Average_delta_Theta/(2*pi*T);
    print_str = ' o   Estimated frequency offset is %0.0f MHz\n';
    fprintf(print_str,est_f_off/1e6);
    Eout = Ein_rot.*exp(-2i*pi*f*est_f_off)*Ph_rec;
end
```

As for the equalizer also for the Carrier Phase Recovery module the structure *Ein* contains the input signals of both polarizations and all the corresponding characteristics. The frequency estimation function was developed to deal with QPSK modulations (4-QAM) and exploit the 4-fold ambiguity of the QAM modulations. Unfortunately, however, this method does not work with higher order modulations and it was basically useless since, to date, the lowest order modulation under research is the 16-QAM. Due to this no further analysis of this fragment of code will be performed.

The following fragment of code regards the main cycle of the CPR module, including all the principal operations introduced at the beginning of Section 4.2.

```
% Main cycle of the Carrier Phase Recovery Module. It estimates the phase
% noise and remove it.

% Test phase multiplication
Ex_fr = Ex_fr.';
Ey_fr = Ey_fr.';
X = exp(1i*phi_b)*Ex_fr;
Y = exp(1i*phi_b)*Ey_fr;

% preliminary decision
dx = zeros(B,length(X));
dy = zeros(B,length(Y));
for k = 1:length(X)
    [dx(:,k),~] = min(bsxfun(@minus,constellation,X(:,k).'));
    [dy(:,k),~] = min(bsxfun(@minus,constellation,Y(:,k).'));
end
dx = (abs(dx)).^2;
dy = (abs(dy)).^2;
% Averaging filtering and select correct phase
switch input_param.type
    case 'joint'
        dx = dx';
        dy = dy';
        dxy = reshape([dx(:) dy(:)]', 2*size(dx,1),[])';
        s = zeros(B,length(dx)-2*N);
        for k = 2*N+1:2:length(dxy)-2*N
            s(:,round(k/2)-N) = sum(dxy(:,k-2*N:k+2*N+1),2);
        end
        [~,indb]= min(s);
    case 'separate'
        sx = zeros(B,length(dx)-2*N-1);
        sy = zeros(B,length(dy)-2*N-1);
        for k = N+1:1:length(dx)-N-1
            sx(:,k-N) = sum(dx(:,k-N:k+N+1),2);
            sy(:,k-N) = sum(dy(:,k-N:k+N+1),2);
        end
        [~,indbx]= min(sx);
        [~,indby]= min(sy);
end
% Unwrap and phase recovery
switch input_param.type
    case 'joint'
        Theta_est = -unwrap(phi_b(indb)*4)/4;
        pn_plot(Theta_est);
        Ex_fr = Ex_fr(:);
        Ey_fr = Ey_fr(:);
        Ex_fr = Ex_fr(N+1:end-N);
```

```
            Ey_fr = Ey_fr(N+1:end-N);
            Ex_cpr = Ex_fr.*exp(-1i*Theta_est);
            Ey_cpr = Ey_fr.*exp(-1i*Theta_est);
        case 'separate'
            Theta_est_x = -unwrap(phi_b(indbx)*4)/4;
            Theta_est_y = -unwrap(phi_b(indby)*4)/4;
            Ex_fr = Ex_fr(:);
            Ey_fr = Ey_fr(:);
            Ex_fr = Ex_fr(N+1:end-N-1);
            Ey_fr = Ey_fr(N+1:end-N-1);
            Ex_cpr = Ex_fr.*exp(-1i*Theta_est_x);
            Ey_cpr = Ey_fr.*exp(-1i*Theta_est_y);
end
```

The first few lines account for the multiplication of the incoming symbols for the different test angles. Since the operation is between 1 by B matrices (the results of the exponential operations with the vector *phi_b* as input) time N by 1 matrices (the variables *Ex_fr* and *Ey_fr*), with B number of test angles and N number of symbols received, the outputs X and Y are N by B matrices.

The following lines show a cycle which compute the minimum distances between the symbols rotated by the different test angles and the points of the constellations. The function *bsxfun* apply the *@minus* operation (which is the normal subtraction) between the elements of the two matrices in input: *constellation* and X or Y. The reason because *bsxfun* is used is that the two matrices do not have the same dimensions nor the same number of elements and it will be extremely inefficient, from a run-time point of view, to implement such operation with cycles. This function instead resize properly one of the input matrix (by making copies of it) to adapt the dimensions and perform the required action with efficient matrix operations. With the *min* functions only the minimum distances for each test angle are stored in the vectors *dx* and *dy* which are squared, as described in the theoretical algorithm, right after the end of the cycle. This cycle is extremely time consuming, even if each iteration is performed with only efficient matrix operations.

After the computation of the squared distances for each test angle and symbol, the noise distortion filtering operation is performed in different ways depending on the information available. If the field *input_param.type* is set to *joint* the code filters together both polarization components, otherwise if it is set to *separate* the components are filtered separately. A value of this field different from the two proposed will result in an error,

once again uncontrolled. Except for the different way of initializing the required vectors, for each test angle $2N + 1$ consecutive distances stored in the vectors dx and dy are summed and only the index of the test phase which result in the minimum sum is stored inside *indbx* and *indby*. The output of this section of the code is then a N by 1 vector containing the indexes of the test angle best estimating the phase noise.
To perform carrier phase recovery, *indbx* and *indby* are used to recreate the phase behaviour in time. To smooth variation of phase greater than $\pm 2\pi$, the function *unwrap* is used. Finally the phase noise is removed by multiplying the received noisy symbols times the opposite of the estimated phase.

4.3 Optimization and results

Also for this module it is possible to perform Memory Access optimization and MEX optimization to improve the performances. In the following paragraphs these processes will be analyzed and the results, including the OFs (Equation (3.7)), will be shown.

4.3.1 Memory access optimization

The author of the original code of this module had already performed Memory Access optimization so, only a small improvement could be done to reduce the memory consumption.

```
%Read the input parameters to initialize the variables.
constellation = input_param.constellation;
R = uniquetol(abs(constellation), 1e-8);
B = input_param.test_angles;
b = (0:B-1)';
Gamma = pi/2;
phi_b = (b/(B-1))*Gamma-pi/4;
N = input_param.N;
rotated_constellation=constellation*(exp(-1i*phi_b).');

...

for k = 1:length(Ex_fr)
    [dx(:,k),~] = min(bsxfun(@minus,rotated_constellation,Ex_fr(k)));
    [dy(:,k),~] = min(bsxfun(@minus,rotaded_constellation,Ey_fr(k)));
end
```

In the original code the received symbols were rotated and then compared with the constellation to identify the closest points. The matrix resulting from the rotation of the symbols was N by B, with N number of symbols received and B number of test angles. Since N can become really large during a simulation the memory consumed was usually extremely significant. To optimize the usage of memory it is then possible to exploit the features of the *bsxfun* function: since it is able to compute the euclidean distance between symbols and constellation points both if the symbols or the constellation is rotated, it is convenient to rotate the latter since the resulting matrix will be M by B, with M number of points in the constellation, which is always much smaller than a N by B one. This operation unfortunately does not reduce the run time of the simulation, since the number of distances that the script has to compute remains the same, but it is still an improvement from an optimization point of view.
Another thing that can be done to reduce the run time of the module is to minimize the number of test angles B. This reduction however cannot be done indiscriminately, since the performances of the module must be maintained.

4.3.2 MEX Optimization

The following code represent the manually written C++ script which compute the euclidean distances between the points and the rotated constellation, identifies the minimum one and operate the noise distortion filtering operation.

```
#include <mex.h>

double minElement(double* array, int size)
{
    double ris=array[0];
    for(int i = 1; i < size; i++)
    {
        if(array[i] < ris)
            ris = array[i];
    }

    return ris;
}
int indexofSmallestElement(double* array, int size)
{
    int index = 0;
```

```
    for(int i = 1; i < size; i++)
    {
        if(array[i] < array[index])
            index = i;
    }

    return index;
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

    //Variables definitions
    const double M_PI=atan((float)1.0)*4;
    double *Ex_re,*Ex_im,*Ey_re,*Ey_im;
    double *Const_re,*Const_im,*cConst_re,*cConst_im;
    double *ris_Ex,*ris_Ey;
    int N,N_test_angles,M,N_avg,joint_FLAG;
    double *x_dist,*y_dist,*x_mins,*y_mins;
    double *phi_b;
    double gamma;
    double *reshaped,*sx,*sy,*tempx,*tempy,*indbx,*indby;

    //Variables initialization
    Ex_re = (double*) mxGetData(prhs[0]);
    Ey_re = (double*) mxGetData(prhs[1]);
    Ex_im = (double*) mxGetImagData(prhs[0]);
    Ey_im = (double*) mxGetImagData(prhs[1]);

    N=mxGetM(prhs[0]);

    Const_re = (double*) mxGetData(prhs[2]);
    Const_im = (double*) mxGetImagData(prhs[2]);
    M = (int) mxGetM(prhs[2]);
    N_test_angles = (int) mxGetScalar(prhs[3]);
    gamma = (double) mxGetScalar(prhs[4]);
    N_avg = (int) mxGetScalar(prhs[5]);
    joint_FLAG = (int) mxGetScalar(prhs[6]);

    //Memory allocation
    plhs[0] = mxCreateDoubleMatrix(N_test_angles, N, mxREAL);
    plhs[1] = mxCreateDoubleMatrix(N_test_angles, N, mxREAL);
    plhs[2] = mxCreateDoubleMatrix(1, (N-2*N_avg)-1, mxREAL);
    plhs[3] = mxCreateDoubleMatrix(1, (N-2*N_avg)-1, mxREAL);

    ris_Ex=mxGetPr(plhs[0]);
    ris_Ey=mxGetPr(plhs[1]);
```

```
    x_dist = (double*) mxMalloc(M*sizeof(double));
    y_dist = (double*) mxMalloc(M*sizeof(double));
    x_mins = (double*) mxMalloc(N_test_angles*sizeof(double));
    y_mins = (double*) mxMalloc(N_test_angles*sizeof(double));
    cConst_re = (double*) mxMalloc(N_test_angles*M*sizeof(double));
    cConst_im = (double*) mxMalloc(N_test_angles*M*sizeof(double));
    phi_b = (double*) mxMalloc(N_test_angles*sizeof(double));
    reshaped = (double*) mxMalloc(2*N*N_test_angles*sizeof(double));
    sx = (double*) mxMalloc(N_test_angles*(N-2*N_avg-1)*sizeof(double));
    tempx = (double*) mxMalloc(N_test_angles*sizeof(double));
    sy = (double*) mxMalloc(N_test_angles*(N-2*N_avg-1)*sizeof(double));
    tempy = (double*) mxMalloc(N_test_angles*sizeof(double));
    indbx = mxGetPr(plhs[2]);
    indby = mxGetPr(plhs[3]);

    //computing test angles
    for (int j=0; j<N_test_angles;j++){
        phi_b[j] = -M_PI/4 + (gamma)*j/(N_test_angles-1);
    }

    //computing useful matrix
    for (int j=0;j<N_test_angles;j++){
        for (int i=0; i<M;i++){
            cConst_re[M*j+i]=Const_re[i]*cos(-phi_b[j])-Const_im[i]*sin(-phi_b[j
  ]);
            cConst_im[M*j+i]=Const_re[i]*sin(-phi_b[j])+Const_im[i]*cos(-phi_b[j
  ]);
        }
    }

    //initializing matrix
    for (int j=0; j<N-2*N_avg; j++) {
        indbx[j]=0;
        indby[j]=0;
    }

    //computing distances
    for (int sym=0; sym<N; sym++){
        for (int t_ang=0;t_ang<N_test_angles;t_ang++){
            for (int j=0;j<M;j++){
                x_dist[j]=pow(Ex_re[sym]-cConst_re[M*t_ang+j],2)+pow(Ex_im[sym]-
  cConst_im[M*t_ang+j],2);
                y_dist[j]=pow(Ey_re[sym]-cConst_re[M*t_ang+j],2)+pow(Ey_im[sym]-
  cConst_im[M*t_ang+j],2);
            }
            ris_Ex[sym*N_test_angles+t_ang]=minElement(x_dist,M);
            ris_Ey[sym*N_test_angles+t_ang]=minElement(y_dist,M);
        }
```

```
    }

    if (joint_FLAG) {
        for (int sym=0; sym<2*N; sym+=2){
            for (int j=0;j<N_test_angles;j++){
                reshaped[sym*N_test_angles+j]=ris_Ex[(sym/2)*N_test_angles+j];
                reshaped[(sym+1)*N_test_angles+j]=ris_Ey[(sym/2)*N_test_angles+j
  ];
            }
        }
        for (int sym=2*N_avg; sym<(2*N-2*N_avg); sym+=2){
            for (int i=0;i<4*N_avg+2;i++){
                for (int j=0;j<N_test_angles;j++){
                    sx[(sym/2-N_avg)*N_test_angles+j]+=reshaped[(sym-2*N_avg+i)*
  N_test_angles+j];
                }
            }
            for (int j=0;j<N_test_angles;j++){
                tempx[j]=sx[(sym/2-N_avg)*N_test_angles+j];
            }
            indbx[(sym/2-N_avg)]=(double) indexofSmallestElement(tempx,
  N_test_angles)+1;
        }
    }
    else {
        for (int sym=N_avg; sym<(N-N_avg)-1; sym++){
            for (int i=0;i<2*N_avg+2;i++){
                for (int j=0;j<N_test_angles;j++){
                    sx[(sym-N_avg)*N_test_angles+j]+=ris_Ex[(sym-N_avg+i)*
  N_test_angles+j];
                    sy[(sym-N_avg)*N_test_angles+j]+=ris_Ey[(sym-N_avg+i)*
  N_test_angles+j];
                }
            }
            for (int j=0;j<N_test_angles;j++){
                tempx[j]=sx[(sym-N_avg)*N_test_angles+j];
                tempy[j]=sy[(sym-N_avg)*N_test_angles+j];
            }
            indbx[(sym-N_avg)]=(double) indexofSmallestElement(tempx,
  N_test_angles)+1;
            indby[(sym-N_avg)]=(double) indexofSmallestElement(tempy,
  N_test_angles)+1;
        }
    }

    //Setting the outputs
    mxSetData(plhs[0],ris_Ex);
    mxSetData(plhs[1],ris_Ey);
```

```
    mxSetData(plhs[2],indbx);
    mxSetData(plhs[3],indby);
    mxSetM(plhs[0],N_test_angles);
    mxSetN(plhs[0],N);
    mxSetM(plhs[1],N_test_angles);
    mxSetN(plhs[1],N);
    mxSetM(plhs[2],1);
    mxSetN(plhs[2],(N-2*N_avg)-1);
    mxSetM(plhs[3],1);
    mxSetN(plhs[3],(N-2*N_avg)-1);
}
```

Like for the Equalizer module only the *mex.h* library was included, to improve time performances. The libraries for complex operation could have been included to simplify the code, but since it is possible to implement the algorithms without them, efficiency was preferred. To deal with the two possible types of noise distortion filtering, instead of dealing with string operations which would require the inclusion of string libraries, a conversion from string to boolean was performed in MATLAB and the flag was then passed as input parameter to the MEX function.

Since a lot of variables were required to implement the code without the use of external libraries, a lot of instructions to allocate memory and initialize variables were required. The following lines shows the call of the MEX function inside the MATLAB script.

```
% Main cycle of the Carrier Phase Recovery Module after MEX optimization
if strcmpi(input_param.type,'joint')
    type_FLAG=1;
else
    type_FLAG=0;
end

[dx,dy,indbx,indby]=BPS_MEX(Ex_fr,Ey_fr,Constellation,B,Gamma,N,type_FLAG);

if type_FLAG
    Theta_est = -unwrap(phi_b(indbx)*4)/4;
    Ex_fr = Ex_fr(N+1:end-N-1);
    Ey_fr = Ey_fr(N+1:end-N-1);
    Ex_cpr = Ex_fr.*exp(-1i*Theta_est);
    Ey_cpr = Ey_fr.*exp(-1i*Theta_est);
else
    Theta_est_x = -unwrap(phi_b(indbx)*4)/4;
    Theta_est_y = -unwrap(phi_b(indby)*4)/4;
    Ex_fr = Ex_fr(:);
    Ey_fr = Ey_fr(:);
```

```
        Ex_fr = Ex_fr(N+1:end-N-1);
        Ey_fr = Ey_fr(N+1:end-N-1);
        Ex_cpr = Ex_fr.*exp(-1i*Theta_est_x);
        Ey_cpr = Ey_fr.*exp(-1i*Theta_est_y);
25  end
```

As can be seen the initial instructions convert the field *type* inside *input_param* into a boolean variable called *type_FLAG* which is then passed in input at the MEX function. The outputs of the MEX function are then used to estimate the phase noise which is then unwrapped and removed from the received symbols. The unwrapping operation is performed outside the MEX script because there are no inefficient cycles and it does not influence significantly the run time of the module. Moreover manual implementation of the unwrap function in the C++ script resulted less efficient than the version offered by MATLAB.

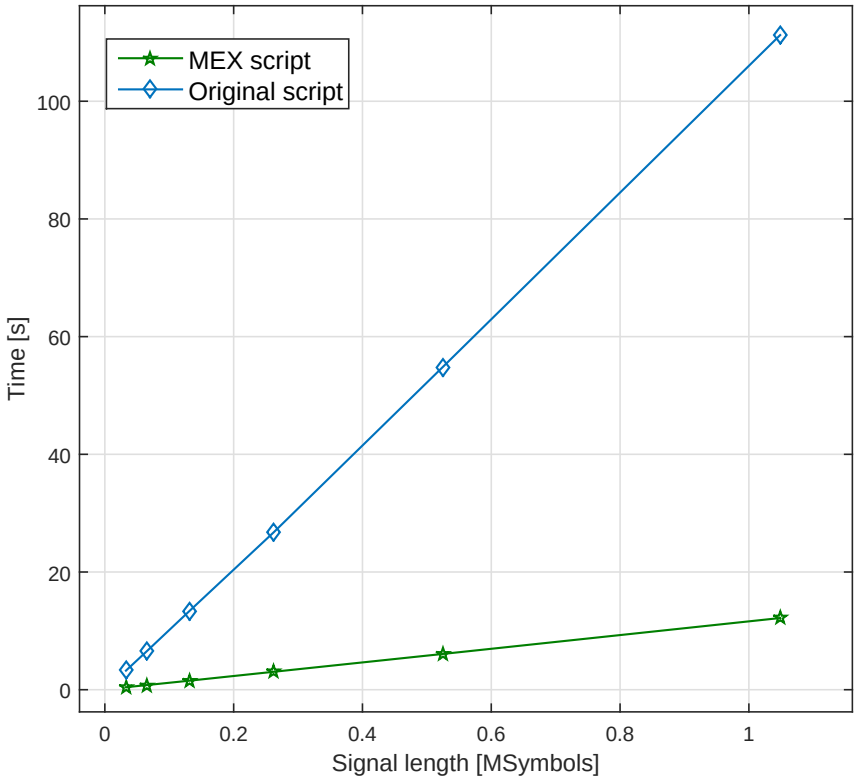

FIGURE 4.7: Time comparison between original script and MEX realized script.

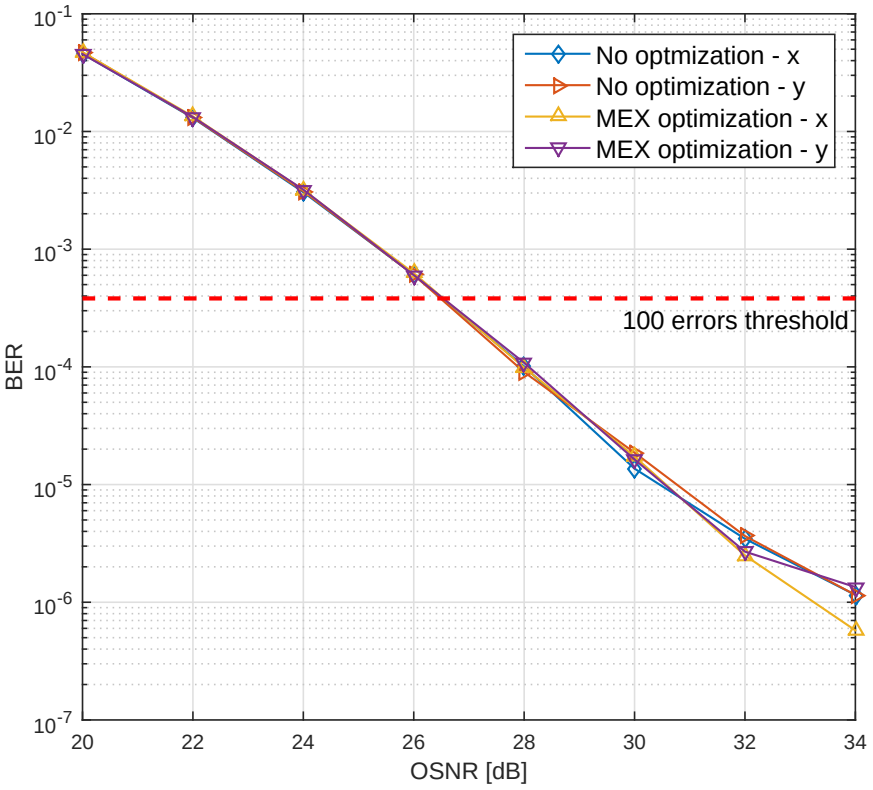

FIGURE 4.8: Performances of the CPR module before and after MEX optimization.

Signal length [bit]	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
MEX script	7.58	7.66	7.72	7.78	7.95	8.08

TABLE 4.1: Optimization Factors for the Carrier Phase Recovery module

4.3.3 Overall results

The results of optimization, computed with the aid of the MATLAB profiler, can be found in Figure (4.7). As can be seen this time memory access optimization is not included since there were no improvements in the run time.

It is graphically visible that both the original script and the MEX optimized one share the same limiting behaviour, with a run time growing linearly with respect to the signal length. This conclusion is confirmed by looking at the Optimization Factors computed in Table (4.1). Even if not as significant as for the equalizer, as can be seen comparing Table (3.2) with Table (4.2), MEX optimization for the Carrier Phase Recovery module has managed to improve significantly the time performances. The reason of this reduced OF is that most of the operations required to implement the algorithm were performed

Original Code	MEX script
0	7.84

TABLE 4.2: Average optimization factors for the Carrier Phase Recovery module

using matrix operations, which are extremely efficient in MATLAB from a runtime point of view. However, since they were inside a cycle, the MEX script still provide large improvements.

After optimization, as for the equalizer module, it is important to verify if the performances of the module have changed with respect to the original script. To do so the same setup used for the equalizer module is recalled and the results are shown in Figure (4.8). As can be seen after MEX optimization the performances remain are not worsened, since the BER curves are overlapped.

Chapter 5

Conclusions

5.1 Overview

In the first chapters of this thesis the structure and general behaviour of optical coherent receivers were analyzed. The schemes used to implement each single block of the digital demodulator were discussed, with a particular focus in latest proposed and researched algorithms.

Successively, in Appendix A, MATLAB optimization was introduced and analyzed in details, showing the significant improvements it can provide to scripts. In particular both MATLAB code optimization and optimization with MEX scripts were introduced and examples of their potentialities were shown.

The optimization of the scripts implementing the modules of an optical coherent receiver was discussed in the Chapters 3 and 4. Chapter 3 introduced the Equalizer module, the theory of the polarization related impairments it has to correct and the original algorithm implemented to deal with them. Successively it described the optimization process and discussed the obtained results. Chapter 4 introduced the Carrier Phase Recovery module, the theory behind phase noise sources and the original algorithm implemented to attenuate their effects. Successively it described the optimization process and discussed the optained results.

In both this chapters the optimization process performed allowed for a significant reduction of the run time of the modules while maintaining the same performances. Quasi real time analysis of a coherent transmission system is possible thanks to these improvements in the time efficiency of the algorithms.

5.2 Final analysis of the results

Before optimization it was not possible to run experiments in real time with coherent optical transmissions implementing 16-QAM or higher order modulations, which are the ones undergoing strong research to date. The data were transmitted, stored right after detection and digitalization and analyzed successively offline. Of course there are significant consequent drawbacks in this setup since, for example, to see the results of a simple change in a transmission parameter it is necessary to reinitialize the system, retransmit all the data and reanalyze all the received symbols.
The run time of a complete simulation was burdened in particular by the equalizer and the CPR modules. Together they accounted for around 1/3rd of the whole simulation time so, considering that a complete setup includes tens of modules, it is possible to see why they were the simulation bottleneck.
Thanks to optimization, in particular thanks to the realization of MEX scripts, the run time of the equalizer module was reduced of around 14 times while the run time of the CPR one was reduced of around 7 times. The direct consequence of such improvements is that the overall run time for a simulation is no longer burdened by those two modules, allowing for quasi real time analysis.
The improvement in the efficiency of the codes implemented in the two modules of the coherent receiver allowed not only a reduction of the run time for simulations inside the framework of Robochameleon, but also advantages in real experiments. Moreover, also transmission systems different from optical ones can benefit from these improvements, like for example wireless channels.

5.3 Future works

Numerous improvements can be realized to further increase the efficiency of the code and reduce the run time of the DSPs. A possible way is to convert in MEX scripts all the cycles of the modules realizing a coherent transmission system in Robochameleon, while another can be to further optimize the MEX script already realized, reducing the amount of memory allocated and the overall number of accesses to the memory.
Another thing that must be considered is that optimization will be required continuously in the future since new and more efficient ideas and models to correct the transmission

impairments are being developed. Significant efforts are put in this research field and, for example, during the course of this project new possible ways to implement both the equalizer and the CPR module were proposed by the Ph.D. students in the Acreo's Optical Transmissions group.

5.3.1 Equalizer and Timing Recovery

The timing recovery module try to remove the intrasymbol desynchronization between the two polarization components and create output signals with a single sample per symbol. The algorithms implemented to date in coherent optical systems are based on the Gardener or the Godard one, which are extremely efficient for BPSK modulations, but requires the reception of very long sequence of symbols to work properly for higher order modulations. However the effects of a bad timing recovery are directly visible in the errors computed by the equalizer module.
A newly proposed algorithms try to perform timing recovery together with equalization using a feedback structure: a sufficiently short sequence of received symbols is used to perform timing recovery. They are then passed to the lattice filter used to perform equalization and the corresponding errors are computed. These errors are used to update the equalizer filters coefficients and are then passed to the timing recovery module which use them to improve the estimation.
The preliminary results of such a scheme show an extreme increase in the performances, but also an higher run time. Thanks to the optimization of the equalizer module performed in this thesis the time efficiency can be improved but a MEX script, including the whole feedback scheme, can probably perform even better.

5.3.2 Carrier Phase Recovery

Also for the CPR module various improvements are being tested, in particular to increase the phase noise tolerance. The first change proposed consists in the run of two consecutive blind phase search algorithms, with the first used to estimate and remove the offset frequency ω_{IF} and the second used to estimate and remove the randomly time-varying phase noise. It is evident that the performances of this scheme are higher than the ones of the algorithm proposed in the Chapter 4, but the run time is doubled. A different way to implement the CPR module then can use once again a feedback

scheme where the results of the phase noise estimation are also used to estimate and remove the offset frequency. This last algorithm however is still under research but it seems very promising.

Appendix A

Code Optimization in MATLAB

MATLAB® is a high-level language for scientific and engineering computation developed by *MathWorks*®[1] [53]. It is a matrix-based language with built-in graphics and a vast library of pre-built toolboxes which allow an easy implementation and setup of algorithms and simulations. To date, according to the *MathWorks* website, millions of engineers and scientists worldwide use MATLAB to analyze and design the systems and products transforming our world. MATLAB is in automobile active safety systems, interplanetary spacecraft, health monitoring devices, smart power grids, and LTE cellular networks. It is used for machine learning, signal processing, image processing, computer vision, communications, computational finance, control design, robotics, and much more.

By its very nature, MATLAB is an interpreted language. The main advantage for the users is that an interpreted language makes scripts implementation and debug easier, since it allows for step-by-step analyses of the codes, but it also reduces the performances from a run time point of view, since interpreted languages are slower than compiled ones like C or C++. If this drawback becomes critical, due to very heavy and time consuming simulations or high-speed oriented scripts, it is necessary to invest some time performing code optimization, removing all possible sources of time waste from the code.

In the following sections code optimization will be thoroughly discussed, starting from MATLAB code optimization and then moving to MEX files.

[1] From now on the ® symbols will be omitted to improve readability

A.1 MATLAB scripts optimization

When MATLAB scripts have to deal with large amount of data or they must simulate high-speed routines, one of the most critical parameter to evaluate the performances of the codes is the run time. MATLAB is an interpreted language, which mean that during execution each line of a script is read, translated into a set of machine language instructions and executed singularly, without any control on what has happened before or what will happen after. Differently, compiled languages like C or C++ before the execution of the script require a tool called *compiler* to translate and optimize the whole code into an efficient machine language set of routines and instructions. Only then they run the program. In the great majority of cases the time performances of a compiled code are much greater then those of interpreted language.

To improve the efficiency and the performances of the scripts, MATLAB implements, and eventually allows to define as will be discussed in the MEX files section, interpreted instructions linked to pre-compiled and extremely optimized routines. In such a way, even if the whole script will be inevitably slower than its completely compiled version, the run times difference will not be so great. The problem that remains to be solved is how to realize MATLAB scripts which time-performances remains limited by only the intrinsic characteristics of the language and not by improper algorithm implementations. There are two possible ways to verify the results of optimization or just to check the time required for running a particular instruction or function. The first uses a very useful tool provided by MATLAB called *Profiler* which is executed automatically by running a script with the *Run and Time* command. The profiler output is a window with information about the elapsed time of all the functions used, the number of times they were called, etc. Unfortunately the profiler cannot be used for continuous simulations (for example a script that continuously receives data and elaborates them), so in those particular cases it can be useful to exploit the functions *tic* and *toc* which respectively start and stop a virtual stopwatch based on the internal timer of the computer. By inserting *tic* before the lines that must be executed, is it possible to use *toc* right after them to obtain the value of the elapsed time. Since all the *toc* functions refer to the instant when the *tic* one was called, it is also possible to use more consecutive *toc* without any loss of information.

MATLAB code optimization can be split in two principal procedures: Memory access

optimization and vectorization. Both these procedures will be analyzed in the following paragraphs.

A.1.1 Memory access optimization

Memory access optimization in MATLAB consist of three different points:

- Memory allocation;
- Column vectors;
- In-place operations.

The MATLAB language does not require the user to declare the types and sizes of variables before using them. It is thus possible to increase the size of an array merely by

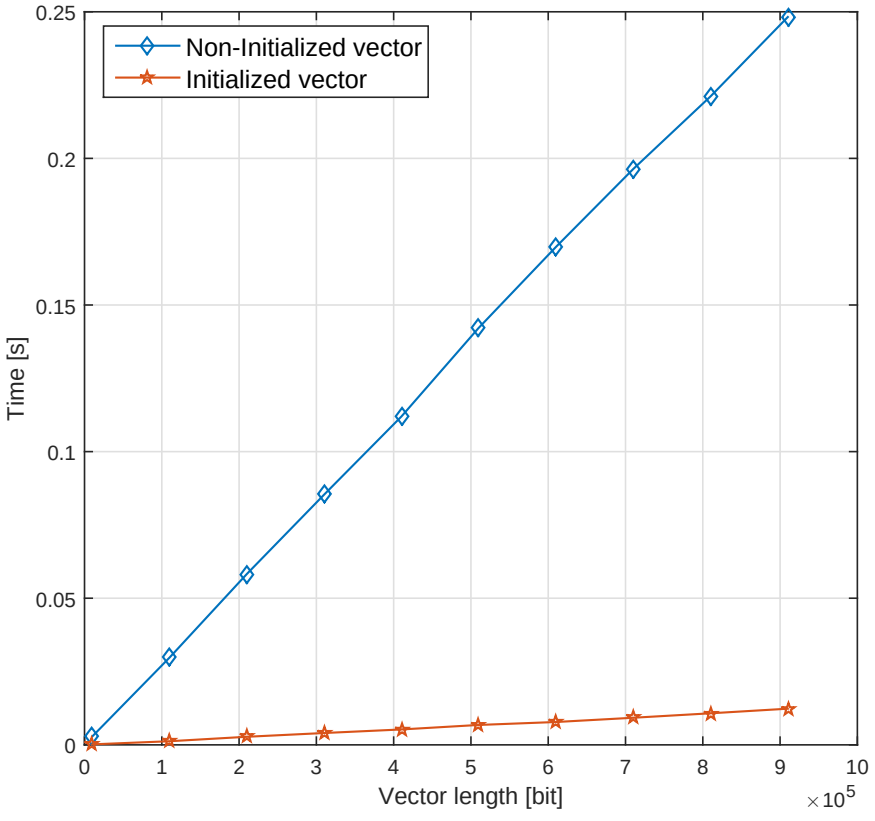

FIGURE A.1: Time required to write into a preallocated vector and a non preallocated one.

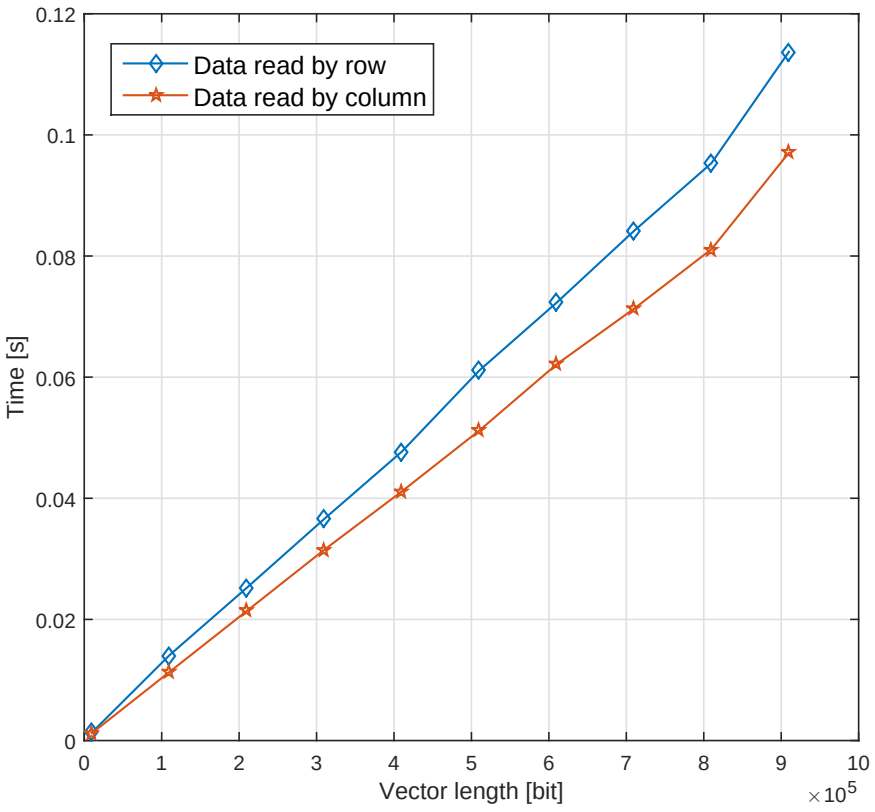

FIGURE A.2: Time required to write into a row vector and a column one.

writing into a position with index larger than the actual size. This approach is convenient for quick prototyping of code, but each time it is done, MATLAB must allocate memory for a new larger array and then copy the existing data into it. Scripts that repeat this procedure in a loop are slow and inefficient. A graphical example of the effects of memory allocation on the run time is shown in Figure (A.1). Some cautions must be taken when analyzing run times figures: the absolute values of the run times carry very little information, since they depend on the hardware of the computer used, on the instantaneous load of the CPU and on several other local factors, nevertheless it is meaningful to observe the changes in the behaviour of different curves or, in the case under exam, the difference between the slopes.

Modern CPUs use a fast cache to reduce the average time taken to access main memory. Scripts achieves maximum cache efficiency when they traverse monotonically increasing memory locations and, since MATLAB stores matrix columns in monotonically increasing memory locations, processing data column-wise results in maximum cache efficiency. The effect of accessing column vector instead of row ones is shown in Figure (A.2). As

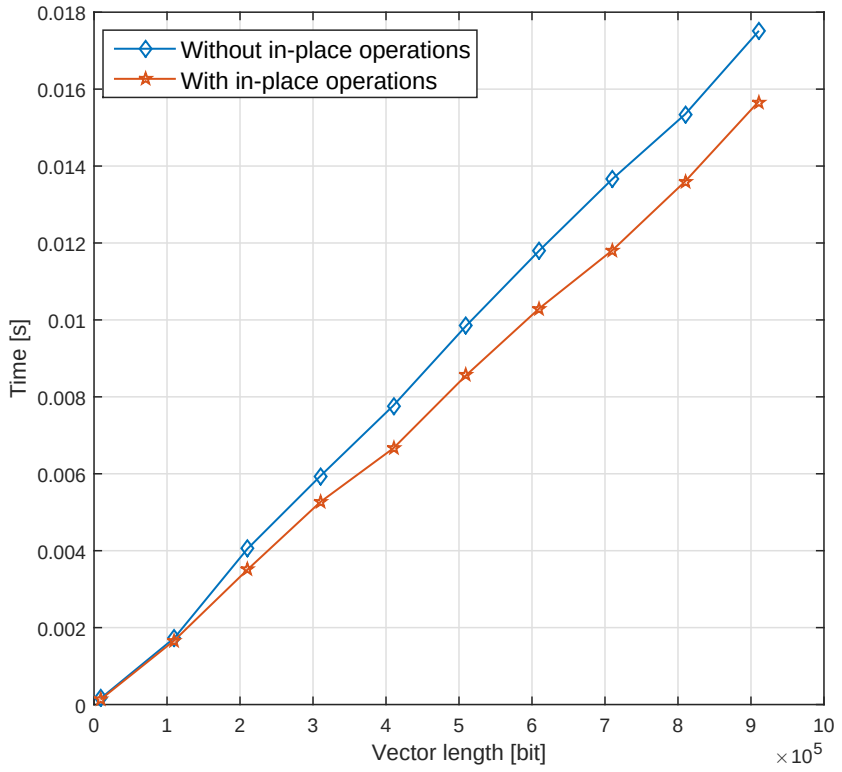

FIGURE A.3: Time required to run a script with and without the use of in-place operations.

can be seen the time optimization here is not significant as in memory allocation optimization, but nevertheless the improvement is not negligible.

In MATLAB it is easy to inadvertently make copies of variables or data. For a large data set this operation uses a lot of memory, and the allocation and copying of memory can itself be time-consuming so, to avoid creating new variables that are modified versions of the existing ones, is it possible to use in-place operations where the input variables can be used as the container for the output data. This capability is available with element-wise operators (such as .*, +), some MATLAB functions (such as *sin* and *sqrt*), and handmade M-functions. To create a function M-file that can be called in-place, the output argument must match the size and the type of one of the input arguments. The ability to call functions in-place is available only when the function itself is called from a function M-file and not from a script. The results of in-place operations against definition of new variables is shown in Figure (A.3). Once again this kind of optimization allows to decrease the run time only of a small fraction, but considering also the amount of memory saved, it is still convenient to perform it.

A.1.2 Vectorization

According to the MathWorks documentation [54], since MATLAB is optimized for operations involving matrices and vectors it is possible to define *vectorization* as the process of revising loop-based, scalar-oriented code to use MATLAB matrix and vector operations. Vectorizing the code is worthwhile for several reasons:

- **Appearance**: Vectorized mathematical code appears more like the mathematical expressions found in textbooks, making the code easier to understand.
- **Less Error Prone**: Without loops, vectorized code is often shorter. Fewer lines of code mean fewer opportunities to introduce programming errors.
- **Performance**: Vectorized code often runs much faster than the corresponding code containing loops.

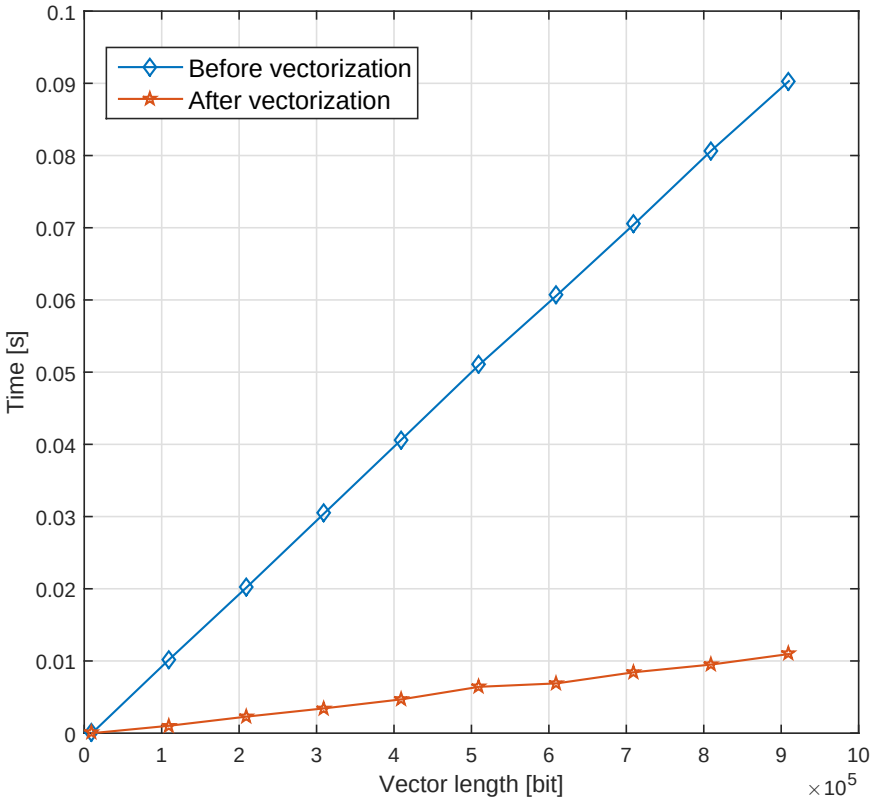

FIGURE A.4: Time required to compute a simple mathematical operation before and after vectorization.

There are several functions in MATLAB designed to improve the performances of the code by vectorization, the complete list is available on the online documentation [54], so the first thing to do while optimizing a script should be to verify if some of them can be used. To provide an example of the effects of this kind of optimization the result of a script which compute the sine function of a variable length vector with a cycle or, after vectorization, with a single matrix operation are shown in Figure (A.4). As can be seen, even in this extremely trivial situation the efficiency of the code is significantly increased, thus saving a lot of time.
Unfortunately not all cycles can be vectorized, like cycles which require the outputs of an earlier iteration to compute the current ones, so other ways must be explored to save time and increase the efficiency.

A.2 MEX files

As said before, MATLAB allows to define interpreted instructions linked to precompiled scripts realized in other languages, like C, C++ or Fortran, to allow optimization also for code fragments where it is not possible to avoid cycles. From now on the thesis will focus on scripts realized in C++, since it is extremely powerful, well documented and well-known, which make the scripts easier to modify, eventually, at a later time. Both C and Fortran are very powerful and old languages, but even if in the MATLAB documentations it is possible to find all the related instructions, they will not be considered here.
There are two possible ways to create MEX files. The first is to use the *C++ coder* application, already built-in in MATLAB, which converts semi-automatically MATLAB M-functions in MEX scripts, otherwise it is possible to write manually the C++ script and then compile it into MEX code by using the command *mex scriptName.cpp* .
The compiled output file depends on the architecture of the CPU of the computer and on the compiler used. Usually scripts compiled on 32-bit CPU or 64-bit CPU are not mutually compatibles, so it can be useful to generate two version of the same script to guarantee a proper behaviour in any situation. The effects of using different compilers to compile the same script are usually visible only from a performance point of view, with more complex (and usually licensed and expensive) compilers performing better than standard ones.

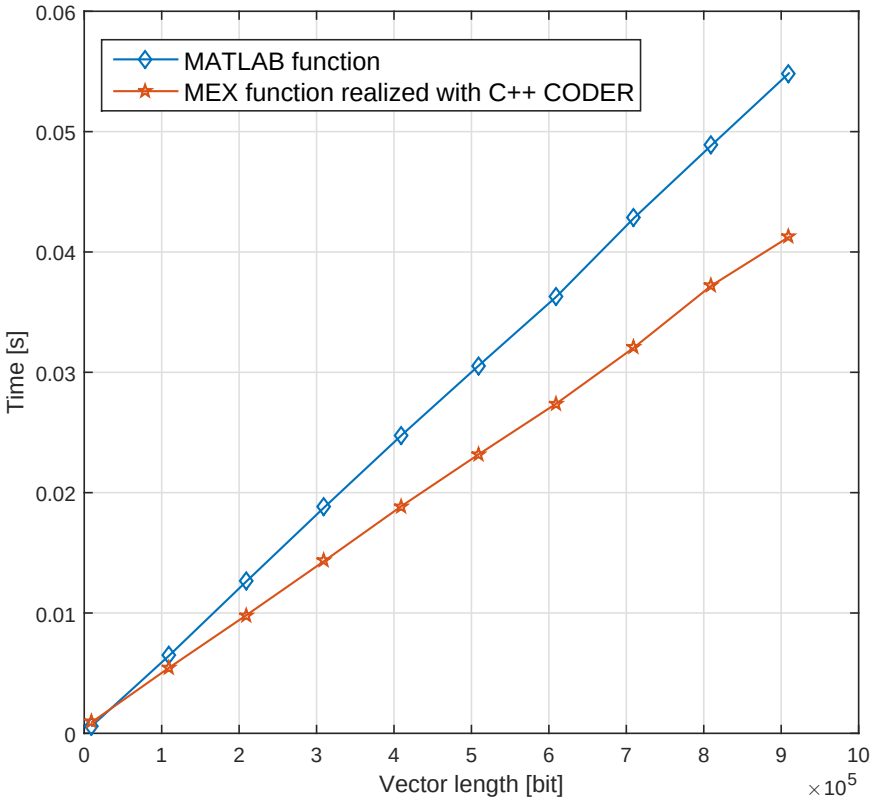

FIGURE A.5: Time required to compute a simple mathematical operation with a MATLAB function or with a MEX function realized with the C++ coder.

A.2.1 MATLAB Coder

The MATLAB coder is a built-in application that allows to create MEX files starting from MATLAB M-functions [55]. It helps in optimizing the run time of the scripts, but can also be used to realize standalone executables to prototype algorithms.

MATLAB Coder implements features which help in preparing the MATLAB algorithm for code generation by analyzing the MATLAB code and proposing data type and sizes for your inputs. It can be possible to ensure that the algorithm is ready for code generation by generating a MEX function that wraps the compiled code for execution back within MATLAB. MATLAB Coder produces a report that identifies any errors required to be fix so it is possible to iterate between fixing errors and regenerating a MEX function until everything end correctly. Unfortunately the proposed data type and sizes for the variables are not always correct or available, so in those cases it is necessary to manually setup them.

To deal with any possible type of variables of any size, the code generated by the compiler

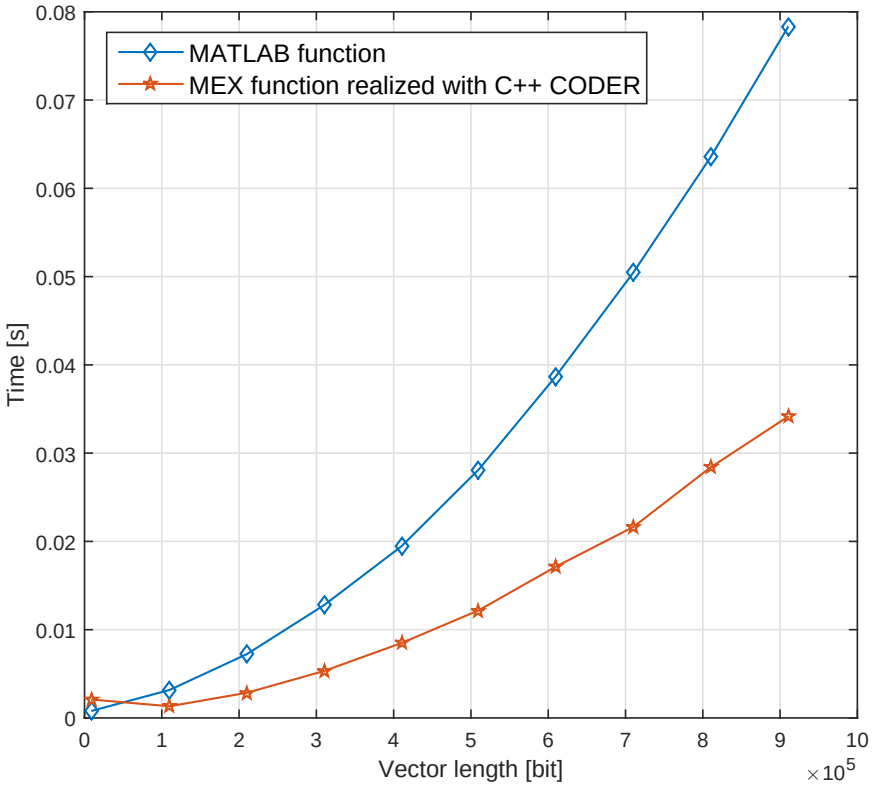

FIGURE A.6: Time required to compute a complex operation with a MATLAB function or with a MEX function realized with the C++ coder.

includes a huge number of functions and controls. This usually is very inefficient and reduce significantly the improvements provided by compiling the code. Moreover not all functions and tools available while programming with the MATLAB language have a precompiled counterpart in C++ and this usually results very annoying, since it requires to modify the MATLAB code or to implement manually all the required operations reducing once again the performances and the efficiency of the script.

A particularly powerful feature of the MATLAB coder is the Embedded Coder which allows to further optimize code efficiency and customize the generated code. Embedded Coder generates code for supported embedded processors, on-target rapid prototyping boards, and microprocessors used in mass production. It extends MATLAB Coder by providing configuration options and advanced optimizations for fine-grained control of the generated code's functions, files, and data. Embedded Coder improves code efficiency and facilitates integration with legacy code, data types, and calibration parameters used in production.

An example of the results that is possible to obtain with the MATLAB coder is shown

in Figure (A.5). The operation implemented in this example is a sinusoidal function using the value obtained in the previous iteration to compute the actual one. Since there is no real computational complexity in such an example the improvement is not so significant, but still it provides a good insight of the potentiality and on the defects of the coder: to compute with a single general MEX script the run times for a variable input vector length it was necessary to define manually, in the setup of the coder, the input vector as unbounded. Since there is no direct translation for unbounded vectors in C++, the MEX script must include functions to deal with it and this results in a drop of efficiency. Another example is shown in Figure(A.6) where this time a more complex operation was performed, with quadratic asymptotic behaviour and more accesses to memory. This time the increase in efficiency is more significant and graphically explain why it is convenient to use MEX scripts to optimize code.

A.2.2 Handmade MEX files

MEX files realized with the C++ coder provides an easy way to increase the time performances of MATLAB scripts in case of non vectorizable cycles. The coder output files, however, are not properly optimized themselves, due to inefficient but necessary checks and functions used to guarantee the required generality of the scripts. A possible solution to remove these inefficiencies is to write manually the C++ scripts.
According to the MathWorks documentation [56] to write a C/C++ script compilable into a MEX file it is required to have:

- A compiler supported by MATLAB;
- The C/C++ Matrix Library API and the C MEX Library API functions;
- The MEX build script.

To realize the C++ script it is necessary to use a gateway routine, acting as an entry point for MATLAB, which must be called *mexFunction*. The function must be of the type *void* and his complete signature is:

void mexFunction(*int* nlhs, *mxArray* *plhs[ ], *int* nrhs, *const mxArray* *prhs[ ]) {

...

}.

It is possible to notice the presence of 4 input parameters, 2 pointers to arrays and two integer parameters representing the number of elements of each pointer. The pointer **plhs[ ]* represent the array of output arguments, while **prhs[ ]* represent the array of input arguments. The names of the parameters can not be changed, otherwise the compiler will return an error.
The arrays used to access input and output parameters are of the type *mxArray* which is the fundamental type underlying MATLAB data. To perform operations like reading or writing into such vectors it is required to use the functions provided by the Matrix library and the MEX library.

A.2.2.1 Data reading

Before reading or writing into the input or output vectors it is formally suggested to verify the number of elements in input or output from the function and their type in order, eventually, to return a proper error message. While these checks may be useful during debug or in the initial stages of implementation, in an operative context focused on optimization they may become useless: it is very common to realize ad hoc scripts which execute an extremely specific operation with well know and unchanging types of input and output data.
To actually read data from the input vector **prhs[ ]* different functions must be used depending on the type of the input data. For accessing a scalar input in position i *mxGetScalar(prhs[i])* can be used, while for matrices *mxGetPr(prhs[i])* and *mxGetPi(prhs[i])* return respectively pointers to real and imaginary parts of the input in position i. To retrieve information about the size of the input it is possible to exploit the function *mxGetDimensions(prhs[i])* which return an array of dimensions, or the functions *mxGetN(prhs[i])* and *mxGetM(prhs[i])* to obtain the scalars representing the number of rows and cols. Since the return type of these functions is *size_t* a cast is eventually necessary to convert them to integer precision.

A.2.2.2 Memory allocation and data writing

It is not possible to directly insert a variable inside the output vector **plhs[ ]* because MEX scripts require the allocation of memory beforehand to avoid segmentation faults.

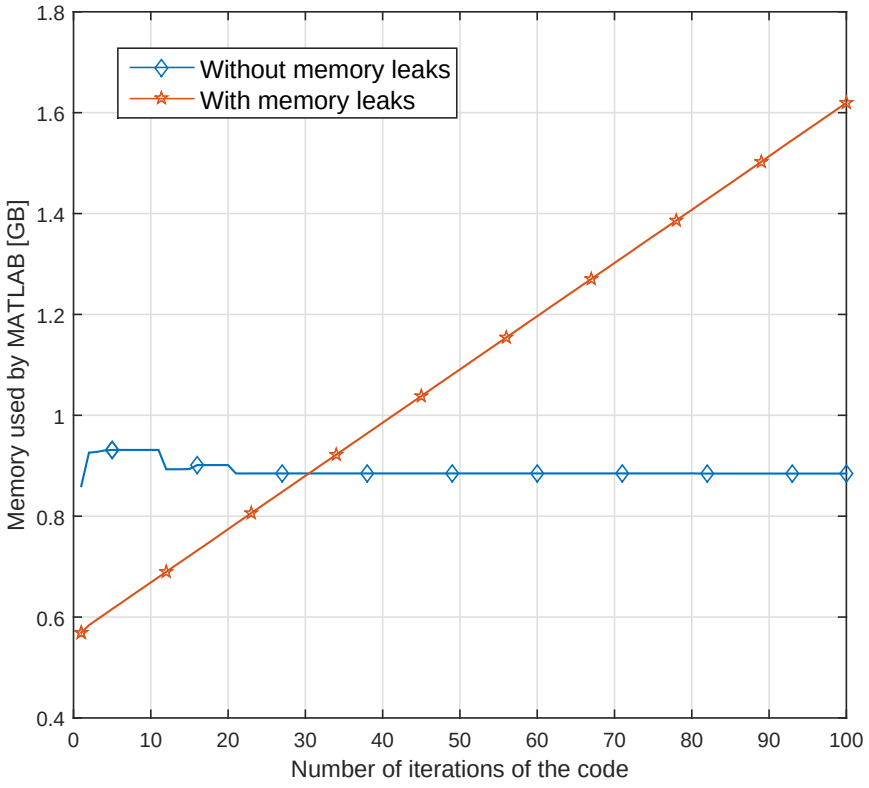

FIGURE A.7: Memory used by MATLAB for scripts with and without memory leaks

The proper procedure to link output variables is to first create a memory slot of appropriate dimension in the position i of the output vector with the instruction *mxCreateDoubleMatrix(nRows, nCols, mxType)* where *nRows* and *nCols* represent the dimension of the output slot that must be allocated and *mxType* specify the type of data (*mxREAL* or *mxCOMPLEX*) that must be stored inside. Once the memory has been allocated, it is possible to obtain pointers to the corresponding real and eventually imaginary part with the instructions *out_Re=mxGetPr(plhs[i])* and *out_Im=mxGetPi(plhs[i])*. Any kind of modifications can then be applied to the contents of the two pointers and finally it is possible to use the instructions *mxSetPr(plhs[i],out_Re)* and *mxSetPi(plhs[i],out_Im)* to make the data available to MATLAB at the end of the MEX function's run.

A.2.2.3 Principal issues

While writing C++ scripts compilable into MEX files it is possible to incur in principally two hard to troubleshoot issues:

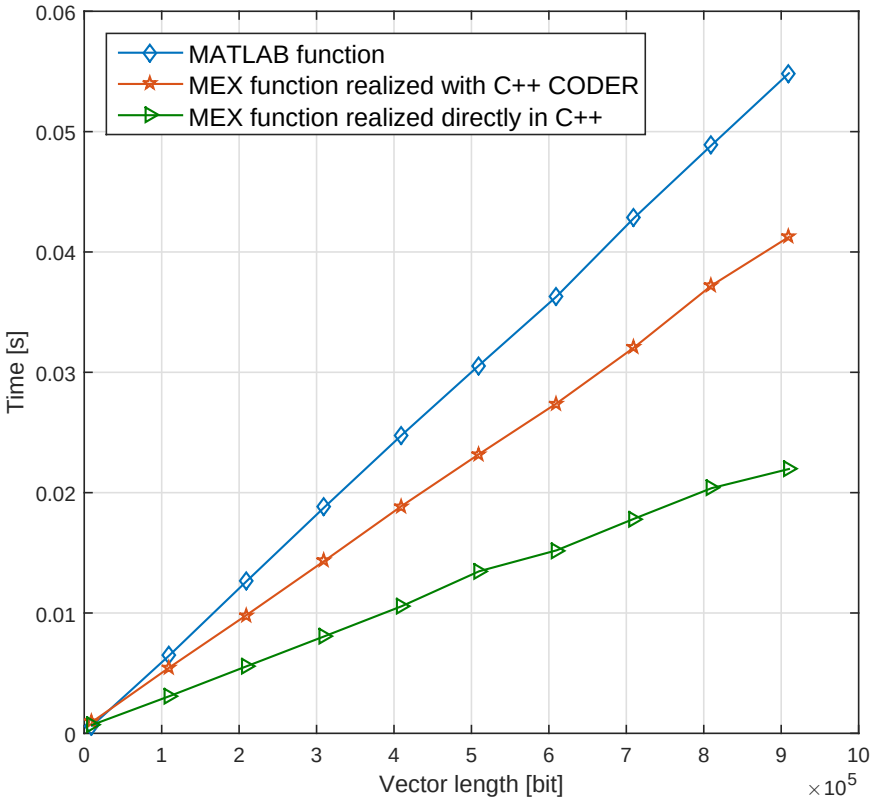

FIGURE A.8: Time required to compute a simple mathematical operation with a MATLAB function, with a MEX function realized with the C++ coder and a MEX function realized from an handmade C++ script.

- Segmentation violations;
- Memory Leaks.

Segmentation violation is an annoying issue that happens when the MEX file attempt to access protected, read-only, or unallocated memory. These types of programming errors can be difficult to track down and solve since segmentation violations do not always occur at the same point as the logical errors that cause them. If a program writes data to an unintended section of memory, an error might not occur until the program reads and interprets the corrupted data. Therefore, a segmentation violation can occur after the MEX file finishes executing.

When a MEX file returns control to MATLAB it returns the results of its computations in the output vector **plhs[ ]*. MATLAB automatically destroys any *mxArray* created by the MEX file that is not in the output vector and moreover it frees any memory that was allocated in the MEX file using the *mxCalloc*, *mxMalloc*, or *mxRealloc* functions. In

general, MathWorks recommends to realize MEX-file functions that destroy their own temporary arrays and free their own dynamically allocated memory [57] since it is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism.

It is possible to define memory leaks as the situations where some memory is allocated by a MEX script and never freed afterwards. Consecutive iterations of such script result in a continuous increase of the memory used by MATLAB, as can be seen in Figure (A.7), and while it is easy to identify the presence of memory leaks, to track down the causes may prove difficult.

A.2.2.4 Optimization results

To graphically understand why it is useful to write manually C++ script compilable into MEX file instead of using the C++ coder, it is possible to realize manually the

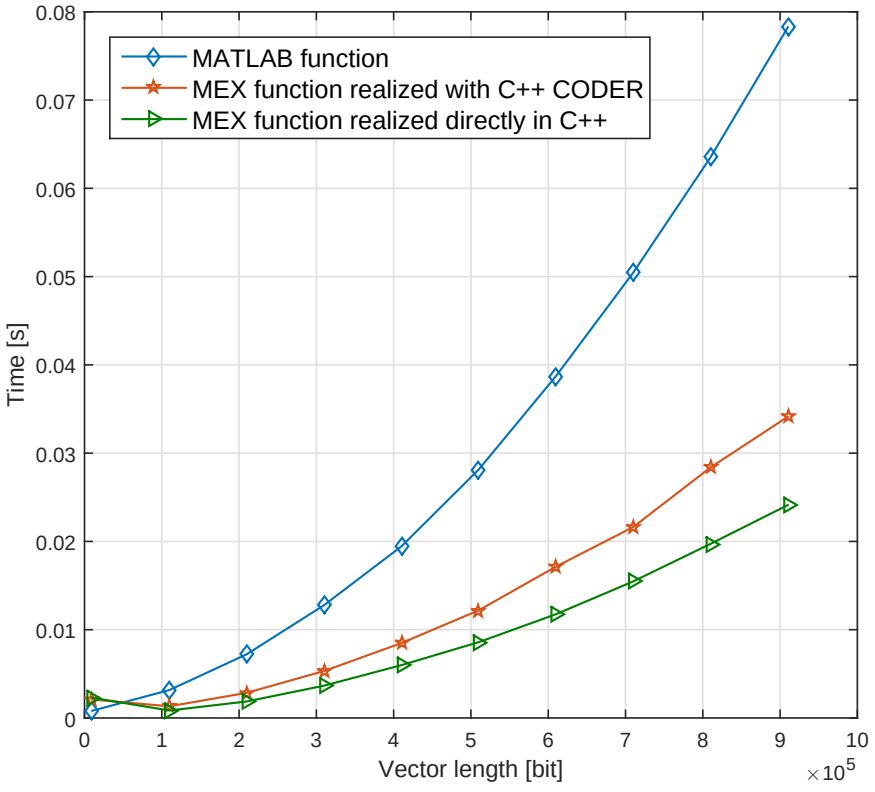

FIGURE A.9: Time required to compute a complex operation with a MATLAB function, a MEX function realized with the C++ coder and a MEX function realized from an handmade C++ script.

C++ scripts for the same operations performed in section A.2.1 and to compare the resulting run times. The Figures (A.8) and (A.9) reports the same data as Figures (A.5) and (A.6) including the run time of the handmade MEX files. As can be seen in both cases there are significant improvements, especially for the simpler operation case, Figure (A.8), where the absence of useless checks and functions allows a strong increase in efficiency. For the more complex operation, Figure (A.9, the improvement in efficiency is more marginal since the run times for the MEX routines is dominated by the computational complexity of the function which is the same for both cases.

Bibliography

[1] Cisco 2015. The zettabyte era: trends and analysis. 2015. URL http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html.

[2] Kazuro Kikuchi. Fundamentals of coherent optical fiber communications. 2016.

[3] Erik Agrell, Magnus Karlsson, AR Chraplyvy, David J Richardson, Peter M Krummrich, Peter Winzer, Kim Roberts, Johannes Karl Fischer, Seb J Savory, Benjamin J Eggleton, and others. Roadmap of optical communications. *Journal of Optics*, 18(6):063002, 2016.

[4] T. Okoshi and K. Kikuchi. *Coherent Optical Fiber Communications*. Advances in Opto-Electronics. Springer Netherlands, 1988. URL https://books.google.se/books?id=NlfljDqiFWcC.

[5] Seb J. Savory. Digital filters for coherent optical receivers. *Optic Express*, 16(2): 804–817, 2008.

[6] Seb J. Savory. Digital coherent optical receivers: Algorithms and subsystems. *IEEE Journal of selected topics in quantum electronics*, 16(5):1164–1179, 2010.

[7] Maxim Kuschnerov; Fabian N. Hauske; Kittipong Piyawanno; Bernhard Spinnler; Mohammad S. Alfiad; Antonio Napoli; Berthold Lankl. Dsp for coherent single-carrier receivers. *Journal of Lightwave Technology*, 27(16):3614–3622, 2009.

[8] Matthias Seimetz and Carl-Michael Weinert. Options, feasibility and availability of 2x4 90° hybrids for coherent optical systems. *Journal of Lightwave Technology*, 24 (3):1317–1322, 2006.

[9] David G. Nairn. Time-interleaved analog to digital converters. *IEEE 2008 Custom Integrated Circuits Conference (CICC)*, pages 289–296, 2008.

[10] Fujitsu. Datasheets, 2015. URL `http://www.fujitsu.com/cn/en/products/devices/semiconductor/fsp/asic/doc/`.

[11] G Raybon, B Guan, A Adamiecki, PJ Winzer, N Fontaine, S Chen, PJ Pupalaikis, R Delbue, K Doshi, B Bhat, et al. 160-gbaud coherent receiver based on 100-ghz bandwidth, 240-gs/s analog-to-digital conversion. In *Optical Fiber Communications Conference and Exhibition (OFC), 2015*, pages 1–3. IEEE, 2015.

[12] Chris R.S. Fludger; Thomas Duthel; Dirk Van den Borne; Christoph Schulien; Ernst-Dieter Schmidt; Torsten Wuth; Jonas Geyer; Erik De Man; Giok-Djan Khoe; Huug de Waardt. Coherent equalization and polmux-rz-dqpsk for robust 100-ge transmission. *Journal of Lightwave Technology*, 26(1):64–72, 2008.

[13] Jianjun Yu and Junwen Zhang. Recent progress on high-speed optical transmission. *Digital Communications and Networks*, 2016. URL `http://www.sciencedirect.com/science/article/pii/S2352864816300116`.

[14] Diego E Crivelli, HS Carter, and Mario R Hueda. Adaptive digital equalization in the presence of chromatic dispersion, pmd, and phase noise in coherent fiber optic systems. In *Global Telecommunications Conference, 2004. GLOBECOM'04. IEEE*, volume 4, pages 2545–2551. IEEE, 2004.

[15] Michael G Taylor. Coherent detection method using dsp for demodulation of signal and subsequent equalization of propagation impairments. *Photonics Technology Letters, IEEE*, 16(2):674–676, 2004.

[16] Wei Chen, Fred Buchali, Xingwen Yi, William Shieh, Jamie S Evans, and Rodney S Tucker. Chromatic dispersion and pmd mitigation at 10 gb/s using viterbi equalization for dpsk and dqpsk modulation formats. *Optics express*, 15(9):5271–5276, 2007.

[17] Seb J Savory, Giancarlo Gavioli, Robert I Killey, and Polina Bayvel. Electronic compensation of chromatic dispersion using a digital coherent receiver. *Optics Express*, 15(5):2120–2126, 2007.

[18] Dominique N Godard. Self-recovering equalization and carrier tracking in two-dimensional data communication systems. *Communications, IEEE Transactions on*, 28(11):1867–1875, 1980.

[19] Yumin Liu, Yuhong Zhang, Yunfeng Peng, and Zhizhong Zhang. Equalization of chromatic dispersion using wiener filter for coherent optical receivers. *IEEE Photonics Technology Letters*, 28(10):1092–1095, 2016.

[20] Heinrich Meyr, Marc Moeneclaey, and Stefan A. Fechtel. *Receiver Structure for PAM Signals*, pages 225–270. John Wiley & Sons, Inc., 2001. URL `http://dx.doi.org/10.1002/0471200573.ch4`.

[21] Floyd M Gardner. A bpsk/qpsk timing-error detector for sampled receivers. *IEEE Transactions on communications*, 34:423–429, 1986.

[22] Kurt H Mueller and Markus Müller. Timing recovery in digital synchronous data receivers. *Communications, IEEE Transactions on*, 24(5):516–531, 1976.

[23] Sun Hyok Chang, Hwan Seok Chung, and Kwangjoon Kim. Digital non-data-aided symbol synchronization in optical coherent intradyne reception. *Optics express*, 16 (19):15097–15103, 2008.

[24] Jan Bergmans. *Digital baseband transmission and recording*. Springer Science & Business Media, 2013.

[25] Jianjun Yu and Junwen Zhang. Recent progress on high-speed optical transmission. *Digital Communications and Networks*, 2016.

[26] Dominique N Godard. Passband timing recovery in an all-digital modem receiver. *Communications, IEEE Transactions on*, 26(5):517–523, 1978.

[27] PJ Winzer, AH Gnauck, CR Doerr, M Magarini, and LL Buhl. Spectrally efficient long-haul optical networking using 112-gb/s polarization-multiplexed 16-qam. *Journal of lightwave technology*, 28(4):547–556, 2010.

[28] Christoffer Fougstedt, Pontus Johannisson, Lars Svensson, and Per Larsson-Edefors. Dynamic equalizer power dissipation optimization. In *Optical Fiber Communication Conference*, pages W4A–2. Optical Society of America, 2016.

[29] William C Lindsey and Chak Ming Chie. A survey of digital phase-locked loops. *Proceedings of the IEEE*, 69(4):410–431, 1981.

[30] Ezra Ip, Alan Pak Tao Lau, Daniel JF Barros, and Joseph M Kahn. Coherent detection in optical fiber systems. *Optics express*, 16(2):753–791, 2008.

[31] Ezra M Ip and Joseph M Kahn. Fiber impairment compensation using coherent detection and digital signal processing. *Lightwave Technology, Journal of*, 28(4): 502–519, 2010.

[32] Michael G Taylor. Phase estimation methods for optical coherent detection using digital signal processing. *Journal of Lightwave Technology*, 27(7):901–914, 2009.

[33] Dany-Sebastien Ly-Gagnon, Satoshi Tsukamoto, Kazuhiro Katoh, and Kazuro Kikuchi. Coherent detection of optical quadrature phase-shift keying signals with carrier phase estimation. *Journal of Lightwave Technology*, 24(1):12, 2006.

[34] Evgeny Vanin and Gunnar Jacobsen. Analytical estimation of laser phase noise induced ber floor in coherent receiver with digital signal processing. *Optics express*, 18(5):4246–4259, 2010.

[35] Yojiro Mori, Chao Zhang, Koji Igarashi, Kazuhiro Katoh, and Kazuro Kikuchi. Unrepeated 200-km transmission of 40-gbit/s 16-qam signals using digital coherent receiver. *Optics Express*, 17(3):1435–1441, 2009.

[36] Andrew Viterbi. Nonlinear estimation of psk-modulated carrier phase with application to burst digital transmission. *Information Theory, IEEE Transactions on*, 29(4):543–551, 1983.

[37] Sebastian Hoffmann, Ralf Peveling, Timo Pfau, Olaf Adamczyk, Ralf Eickhoff, and Reinhold Noé. Multiplier-free real-time phase tracking for coherent qpsk receivers. *IEEE Photonics Technology Letters*, 21(1):137, 2009.

[38] Md Saifuddin Faruk. Blind equalization and carrier-phase recovery based on modified constant-modulus algorithm in pdm-qpsk coherent optical receivers. *Optical and Quantum Electronics*, 48(1):1–9, 2016.

[39] Timo Pfau, Sebastian Hoffmann, and Reinhold Noé. Hardware-efficient coherent digital receiver concept with feedforward carrier recovery for m-qam constellations. *Journal of Lightwave Technology*, 27(8):989–999, 2009.

[40] William Shieh and Keang-Po Ho. Equalization-enhanced phase noise for coherent-detection systems using electronic digital signal processing. *Optics Express*, 16(20): 15718–15727, 2008.

[41] Aditya Kakkar, Jaime Rodrigo Navarro, Richard Schatz, Xiaodan Pang, Oskars Ozolins, Hadrien Louchet, Gunnar Jacobsen, and Sergei Popov. Equalization enhanced phase noise in coherent optical systems with digital pre-and post-processing. In *Photonics*, volume 3, page 12. Multidisciplinary Digital Publishing Institute, 2016.

[42] René-Jean Essiambre, Gerard J. Foschini, Gerhard Kramer, and Peter J. Winzer. Capacity limits of information transport in fiber-optic networks. *Phys. Rev. Lett.*, 101:163901, Oct 2008.

[43] DJ Richardson, JM Fini, and LE Nelson. Space-division multiplexing in optical fibres. *Nature Photonics*, 7(5):354–362, 2013.

[44] Akira Hasegawa and Yuji Kodama. Signal transmission by optical solitons in monomode fiber. *Proceedings of the IEEE*, 69(9):1145–1150, 1981.

[45] Claudio Vinegoni, Magnus Karlsson, Mats Petersson, and Henrik Sunnerud. The statistics of polarization-dependent loss in a recirculating loop. *Lightwave Technology, Journal of*, 22(4):968–976, 2004.

[46] Nicolas Gisin, Mark Wegmuller, A Bessa dos Santos, and JP von des Weid. Measurements of enhanced ber fluctuations due to combined pmd and pdl effects in optical systems. In *Proc. Symp. Optical Fiber Measurements' 00*, pages 105–108, 2001.

[47] Andrea Galtarossa and Luca Palmieri. The exact statistics of polarization-dependent loss in fiber-optic links. *IEEE Photonics Technology Letters*, 15(1):57–59, 2003.

[48] Antonio Mecozzi and Mark Shtaif. The statistics of polarization-dependent loss in optical communication systems. *IEEE Photonics Technology Letters*, 14(3):313–315, 2002.

[49] DTU: Danmarks Tekniske Universitet. Robochameleon. URL `http://dtu-dsp.github.io/Robochameleon/`.

[50] Robert Sedgewick. *Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984. ISBN 0-201-06672-6.

[51] Paschotta Rüdiger. Noise in laser technology – part 1: Intensity and phase noise. *Optik and Photonik*, 48(2), 2009.

[52] RP-Photonics. Phase noise. URL `https://www.rp-photonics.com/phase_noise.html`.

[53] MathWorks. Matlab, . URL `http://se.mathworks.com/products/matlab/`.

[54] MathWorks. Matlab: Vectorization, . URL `http://se.mathworks.com/videos/optimizing-and-accelerating-your-matlab-code-107711.html`.

[55] MathWorks. Matlab: Coder, . URL `http://se.mathworks.com/products/matlab-coder/`.

[56] MathWorks. Matlab: C++ source scripts for mex files, . URL `http://se.mathworks.com/help/matlab/write-cc-mex-files.html`.

[57] MathWorks. Matlab: Memory management issues, . URL `http://se.mathworks.com/help/matlab/matlab_external/memory-management-issues.html`.