

OPTICAL WIRELESS CHANNEL SIMULATION

Research work

conducted by

NEVENA DJAJA

Institute for Broadband Communication
Graz University of Technology



Supervisor: Ao. Univ.-Prof. Dr. Erich Leitgeb
Advisor: Ao. Univ.-Prof. Dr. John Barry

Atlanta, October 2012

Abstract

Investigations into the possible applications of Radio Frequency (RF) modulation and coding techniques on Optical Wireless (OW) channels are important since they can enhance link performance and signal transmission quality. This work performs the study of such applications using the simulation framework developed as an essential part of it. The framework is built using C++ in an object-oriented manner, designed for extensibility, high-performance, and platform-independence. In addition to the often used MATLAB, it provides an efficient tool for researchers. Results of the simulations using the framework are in accordance with those reported in prior research in the field of optical wireless. These results are presented in different formats, textual and graphical, which makes them easy to compare and study. The simulation framework consists of modules that implement different functionalities of the communication channel, such as pseudo-random number generators, checksum calculators, modulators/demodulators and coding schemes. Modulation schemes that are studied using this simulator are optical On-Off Keying (OOK), Pulse-position Modulation (PPM), Binary Phase Shift Keying (BPSK), Quadrature Phase Shift Keying (QPSK), M-ary Pulse Amplitude Modulation (M-PAM), M-ary Quadrature Amplitude Modulation (M-QAM) and they can be combined with channel coding techniques such as Bose-Chaudhuri-Hocquenghem (BCH) codes, Reed-Solomon (RS) and Low Density Parity Check (LDPC) codes.

Kurzfassung

Forschungen im Bereich der Optischen Freiraumkommunikation die sich auf die mögliche Anwendung der Modulations- und Kodierungsverfahren stammend aus der Rundfunktechnik beziehen, erweisen sich als besonders wichtig um die Leistung und Qualität der Signalübertragung am Kanal zu verbessern. Diese Arbeit untersucht solche Anwendungen mit Hilfe eines Simulations-Frameworks, das innerhalb der Arbeit aufgebaut wurde. Dieses Framework wurde unter Verwendung der Programmiersprache C++ entwickelt, die eine objektorientierte Programmiersprache ist und ein erweiterbares, hochleistungsfähige Design sowie Plattformunabhängigkeit anbietet. Neben der meist verwendeten Programmiersprache zur Simulationszwecke Matlab, C++ bietet ein effizientes Tool für die Forscher. Die Ergebnisse der Simulationen mit dem entwickelten Framework stimmen mit den bisherigen Untersuchungen in dem Bereich von Optical Wireless überein. Diese sind in verschiedenen Formaten dargestellt, in Form von Textdateien und auch graphisch in Form von Plots, daher sind die Ergebnisse deutlicher und einfacher zu vergleichen und studieren. Das Simulations-Framework besteht aus verschiedenen Bausteinen, die hier Modulen genannt werden, und die verschiedene Funktionalitäten des Kommunikationskanals implementieren, wie zum Beispiel Pseudo-Zufallsgeneratoren, CRC Prüfsumme Rechner, Modulatoren/Demodulatoren und Kodierungsverfahren. Modulationsverfahren, die unter der Verwendung dieses Simulators untersucht wurden sind On-Off Keying (OOK), Pulse-position Modulation (PPM), Binary Phase Shift Keying (BPSK), Quadrature Phase Shift Keying (QPSK), M-ary Pulse Amplitude Modulation (M-PAM), M-ary Quadrature Amplitude Modulation (M-QAM) und diese können mit den folgenden Kanalkodierungsverfahren kombiniert werden: Bose-Chaudhuri-Hocquenghem (BCH) codes, Reed-Solomon (RS) and Low Density Parity Check (LDPC) codes.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

date

(signature)

Contents

1	Introduction	1
1.1	Scope of work	2
1.2	Project Requirements	4
2	Channel Model, Modulation and Coding	6
2.1	Channel Model	7
2.2	Modulation techniques	8
2.2.1	On-Off Keying (OOK)	8
2.2.2	Pulse-Position Modulation (PPM)	10
2.2.3	M-ary Phase Shift Keying (M-PSK)	10
2.2.4	M-ary Pulse Amplitude Modulation (M-PAM)	13
2.2.5	Multi-Level Quadrature Amplitude Modulation (M-QAM)	14
2.2.6	Comparison of modulation techniques	14
2.3	Channel coding	15
2.3.1	Bose, Chaudhuri and Hocquenghem (BCH) codes	16
2.3.2	Reed-Solomon (RS) block codes	17
2.3.3	Low-Density Parity-Check (LDPC)	18
3	Software Architecture	19
3.1	Framework	22
3.1.1	Simulation Setup	22
3.1.2	Error verification and exceptions handling	24
3.1.3	Simulation flow	25
3.1.4	Output format	25
3.1.5	Visualization in Matlab	27
3.2	Configuration	27
3.3	Modules	29
4	Description of Simulation Modules	31
4.1	Main modules and link interface class	31
4.1.1	CFramework	31
4.1.2	CLinkModules	33
4.1.3	CSimulationInterface	34
4.2	Other modules	35
4.2.1	ParseXML	35
4.2.2	Random Generators	36
4.2.3	CRC checksum	39
4.2.4	Modulators	41
4.2.5	Demodulators and Detection	46

5	Simulation results and discussion	47
5.1	Modulation schemes uncoded	47
5.1.1	Comparison between RF and optical	47
5.2	Modulation schemes coded	49
6	Conclusion and future work	50

List of Figures

2.1	Optical channel block diagram	6
2.2	Random OOK signal [9]	9
2.3	Random PPM signal [9]	10
2.4	BPSK constellation diagram	11
2.5	QPSK constellation diagram	12
2.6	Random PAM signal [9]	13
2.7	Optical power gain over OOK vs bandwidth efficiency [13]	15
3.1	Basic Software Architecture	20
3.2	CreateModules class definition	23
3.3	Console output	26
3.4	BER vs SNR plot	27
3.5	XML file with simulation parameters	28
5.1	BER of OOK vs M-PAM modulation	48
5.2	BER of OOK vs M-PAM modulation	48
5.3	BER of multilevel PPM modulation	49

1 Introduction

Optical wireless communications represent one of the most promising approaches for addressing the increasing demand in the global broadband access market. Current popular broadband access technologies include cable and radio-frequency links. Cable offers reliable weather-independent links, although requires point-to-point infrastructure (cable from the company to house) which results in high "last-mile" costs. Radio frequency (RF) links on the other hand can be mildly affected by weather, and they do not require point-to-point infrastructure. The issue with current RF technology is the already high-demand in RF spectrum and license requirements that are highly priced.

Commonly referred to as optical wireless (OW), Free Space Optics (FSO) offers low start-up and operational costs, high security, rapid deployment, high fibre-like bandwidths, and doesn't require licenses. The current drawback of these FSO links is that they can be severely affected by weather conditions such as fog, clouds and atmospheric turbulence [13]. To improve the performance of optical wireless links further, we investigate different channel modeling techniques. More specifically, this work looks at performance that can be achieved by deploying various modulation techniques together with channel coding techniques on the transmission channel. These modulation techniques, aim to correction of possible errors that can occur during the transmission.

A significant amount of theoretical research and testing of modulations with hardware implementation has already been done at the Institute of Broadband Communications at the TU Graz. Techniques for signal modulations such as the OOK (On-Off Keying) modulation and the PPM (Pulse Position Modulation) have been thoroughly investigated and publish in different optical journals within the FSO links research, such as [12],[20],[18],[25],[19],[24] and much more. However, limited work has been done regarding the possible application of RF modulation schemes on optical wireless links and no simulator has been developed in C++ that can test these schemes.

In this work, we investigate signal processing techniques which are predominantly used in RF communications and their possible deployment to FSO links, i.e. optical wireless channels. More specifically, we look at techniques such as PSK (Phase Shift Keying) - BPSK (Binary PSK), QPSK (Quadrature PSK), and M-PSK (M-ary PSK), followed by QAM (Quadrature Amplitude Modulation) and PAM (Pulse Amplitude Modulation).

In order to perform the investigation, a number of concrete tasks are undertaken:

- An optical wireless simulator is built. We describe the simulator in detail in further chapters, but the basic components include a simulator controller, coder, modulator, link noise generator, demodulator, and decoder. Building all the end-to-end components allows a more realistic and accurate experiments.
- Signal processing modulation techniques such as OOK, PPM, BSK, QPSK, M-QAM and M-PAM are implemented and their performance compared.
- Signal processing Forward Error Correction (FEC) techniques typically used in RF communication are described and implemented. Examples of FEC techniques are: BCH (Bose - Chaudhuri - Hocquenghem) code, RS (Reed-Solomon) code and LDPC (Low Density Parity Check) code.
- Noise generators are created to simulate link noise that may occur due to weather conditions or atmospheric irregularities.
- A number of modulation and coding techniques are combined and tested to investigate whether they are suitable for use with optical wireless.

Although typical signal processing simulations are done in Matlab, and in some cases using the C programming language, the software used for the simulator built is C++. Using C++ provides a number of advantages that are described in the following chapters. To the best of knowledge, the simulator built within this work is one of the first full optical wireless link simulators implemented in the C++ programming language.

Optical wireless channels present many unique constraints not present in RF communications. These are discussed in detail in further chapters. Therefore, typical signal processing techniques from RF communications cannot be applied directly to the optical channel; they need to be adapted to the optical channel requirements and used only in that form. Also only certain combinations of modulation and coding techniques make the channel more reliable and resilient against channel faults. Within this project the efficient solutions will be suggested and evaluated. These can further be implemented and tested with corresponding hardware design.

1.1 Scope of work

The main incentive for this project is the investigation of using different modulation and coding techniques, predominantly used in RF communications, in optical wireless communications, and also to build a software simulator that will be easy to use and extend for further research and investigation. The primary evaluation criteria is the channel error performance under different combinations of modulation and coding techniques. In addition, we discuss the advantages and disadvantages of the usage of those techniques in various scenarios.

The purpose of the investigation is contribute to the design of optical wireless links efficiently. By building and providing an extensible framework in C++ which can be extended, new modulators/demodulators or encoders/decoders schemes can be proposed and tested in software rapidly. Additionally, based on the result of the evaluation of some solutions, hardware implementations of the simulator could be proposed; however we do not include this in the scope of this project. If the necessary components are available, the proposed solution will then be developed and tested, in cooperation with other colleagues at the research institution at TU Graz.

The optical channel simulation is built in a modular manner which enables the implementation of channel blocks or modules as separate entities. More specifically, it consist of separate, mostly independent modules developed as a part of the communication channel, that are connected over the common interface, whereby the output vector of the each module becomes the input vector of the next, adjacent module in the order as they are listed. Due to this design, it is also easily expandable and results in additional modules and/or functionalities being easy to add.

The result of the simulation represents BER (Bit Error Rate) values for the given SNR (Signal To Ratio) values together with the Block error rates on the channel. This method of evaluating results is chosen, as it is a standard method of evaluating error patterns and link performance of optical wireless channel in order to research different signal processing methods for a better signal reception. They are presented in the form of textual files and console output, and also plotted using Matlab.

C++ is the programming language used due to its modular, object-oriented approach that enables channel components to be easily implemented. It can also be compiled to machine code, making the simulator fast and avoiding the overhead of an interpreter (such as Matlab) or a virtual machine. If using the programming style where certain language constructs can be avoided, the source code can be completely independent of the platform on which it is compiled. This programming style was pursued and tested on different platforms (Windows, Linux, Mac OS). Given that it can be compiled to machine code, distribution as binaries specific to individual platforms is also possible.

It is also easy to add an additional module to the program, or to modify the existing one, since the modules are independent, encapsulated with common interface that links them together. Also, C++ with its various choice on pointers, templates and dynamic memory allocation offers the possibility to develop generic and dynamic modules.

This work is organized as follows:

Chapter 1 - *Introduction*, includes the scope of this work and a short introduction to the optical wireless and free space technologies. It gives an overview over the different channel techniques used in the simulation. The requirements on the software development are also given in this chapter.

Chapter 2 - *Channel Model, Modulation and Coding*, here the channel model is described and its block diagram is provided, as typically used for such simulations. Also the channel model for optical wireless is presented briefly. All modulation schemes and channel coding techniques that are being investigated are described within this chapter.

Chapter 3 - *Software Architecture*, explains how the communication channel simulation is implemented in C++; it gives the description of the framework setup, configuration and simulation flow, together with the short description of each module. In addition, it discusses the ways the output format is given and presents the configuration file with parameters.

Chapter 4 - *Description of Simulation Modules*, presents in detail the different modules or building blocks of the simulator. We start with the framework description followed by a detailed description of the simulation components: the simulation interface, the linking modules, random generator module, modulators/demodulators, encoders/decoders and channel types.

Chapter 5 - *Simulation results and discussion*, the results of the simulation after the execution of the main routine (C++) are presented, in the form of the console output, formatted files, and plots using Matlab GUI. Also a discussion of advantages and disadvantages of various results is given and commentary on the presented results is provided, as appropriate.

Chapter 6 - *Conclusion and future work*, presents the conclusion of this research work and proposals for the future research work in this field.

1.2 Project Requirements

The following requirements were setup as part of this work:

- Design and develop a general simulation framework that implements the simulation set-up and imports configuration parameters dynamically, at the run-time.
- Build an XML file that contains all simulation parameters and import them in the simulation configuration file.
- Implement a parser that extracts simulation parameters from the XML file in the way that if there were any changes, only the XML file and the simulation configuration file need to be changed
- Implement the interface between the modules, such that buffers for input and output values are dynamically allocated. Additional modules should be easily supported.
- Each module for the each block in the block-diagram presentation of the simulation channel should be realized as one class in C++.

-
- Platform independent source code with possibility to compile it and run it on different platforms (Windows, Mac, Linux) without changing the actual code.
 - Output in the form of the CSV (Comma Separated Value) file containing all the necessary results and the console output. Afterwards, the results are plotted using the Matlab plots.

2 Channel Model, Modulation and Coding

Communication channels are typically presented in a simplified manner using a block diagram. Each block is a module or building block, which in most cases corresponds to a mathematical model. For an optical communication channel, such a block diagram is presented in figure 2.1. It consists of a transmission segment, transmission channel and a reception segment. This is a simplified presentation of the channel, as it can generally contain more or fewer blocks, depending on the signal processing methods deployed in a specific case.

The transmission part of the channel consists of a random generator that generates statistically independent numbers, followed by a channel encoder for the Forward Error Correction (FEC) algorithms. A modulator then modulates the bits onto the carrier signal, to which electrical to optical converter is connected. Hence, until electro-optical conversion module, all transmission modules are equivalent or similar to those in a RF channel.

The actual transmitter of the optical channel is a light source, usually a Light Emitting Diode (LED) or a Laser Diode (LD). It sends out optical signals through the channel that will be received at the reception segment. The channel itself introduces certain impairments and can be modeled as an Additive White Gaussian Noise (AWGN) channel (as typical in communication channel simulations), or as a Fading channel. Both of these will be described in the following sections.

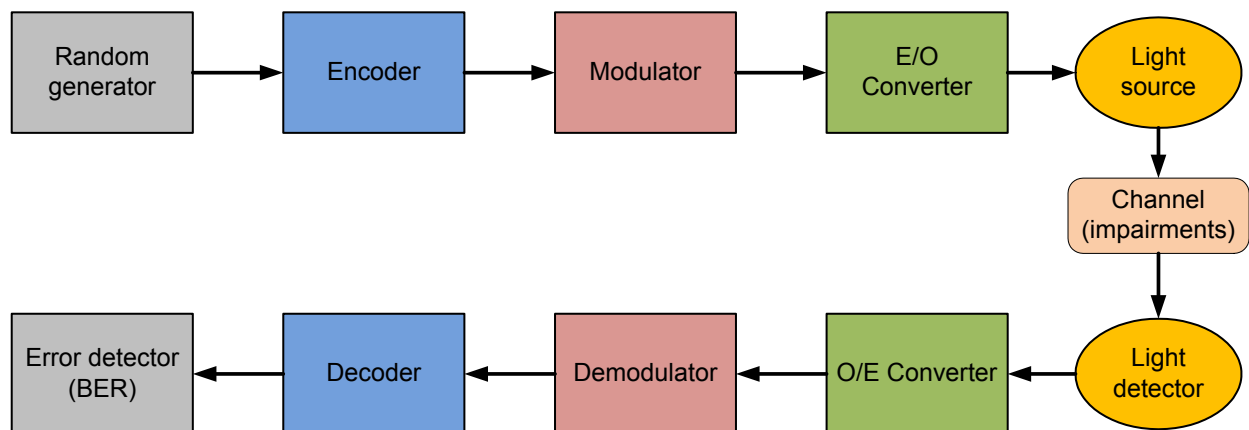


Figure 2.1: Optical channel block diagram

The reception segment contains modules in the opposite order from the transmission segment. This begins with the light detector, which in most cases is a Photo-diode (PD), followed by the optical to electrical converter - together, they constitute the reception part of the optical channel. Next in the sequence are a channel demodulator for demodulation of modulated signals and decoder. In case an error correction method is used on the channel, an error detector module and bit error rate calculation module are present. Therefore in a simple presentation of an optical wireless channel the signal is electrically processed, converted to light, transmitted through the wireless optical channel, received by a light detector, and finally converted back to an electrical signal for further processing.

Within this project, both channel coding pair and modulation pair from the block diagram 2.1 are included in the simulation, whereby the electric to optical (E/O) converter is implemented as a part of the modulator module and optical to electric (O/E) converter as a part of the demodulator module. The error detector block takes the bit sequence coming out of the decoder and compares it to the transmitted bit sequence, which is the output of the random generator block in the transmission segment. It then calculates the Bit Error Rate, which is number of different bits over number of bits in the block. It also calculates the block error rate, by dividing the BER with the number of blocks.

Different BERs are simulated for different channel characteristics and simulation settings; for example, these can be only modulation without the coding block, or coding with some simple form of modulation.

2.1 Channel Model

When modeling the optical channel, there are certain conditions and constraints different from Radio Frequency (RF) communication channels that need to be considered [13],[41],[23],[9]. The information in an optical channel is transmitted at the optical intensity $I(t)$ of the transmitted signal, which is defined as the optical power emitted per solid angle, in units Watts per steradian [W/sr] (for more details, see [13]). This differs from a typical RF channel, where the information is contained in a signal's amplitude, frequency or phase. Since optical power can never take negative values, the signal containing the information transmitted through the optical channel has to remain non-negative at all times.

$$I(t) \geq 0 \tag{2.1}$$

The other important constraint rises from concerns about the eye and skin safety [41],[13]. This has been addressed in numerous guidelines and standards by international regulatory bodies, such

as the International Electrotechnical Commission (IEC) in standard IEC60825-1 [7] and the American National Standards Institute (ANSI) in their publications [14]. The limitation on the optical signal power that can be emitted from the optical source is mathematically expressed as:

$$\lim_{x \rightarrow +\infty} \frac{1}{2T} \int_{-T}^T I(t) dt \leq P \quad (2.2)$$

This means that the average amplitude over the signal period, which is equal to the normalized average optical power, is limited to some fixed value P according to the above mentioned standards. These two constraints are considered in the whole simulation.

2.2 Modulation techniques

Modulation is the process of mapping one information signal onto another signal, called the *carrier signal*, according to a given set of rules [16]. Its purpose is to convey the information signal over a wireless medium and in order to do so, it has to be modulated and shifted to some higher frequencies suitable for wireless transmission. The most suitable set of modulations for the optical channel transmission, keeping in mind the above mentioned constraints, are digital modulation techniques [13],[41].

In a digital modulation scheme, an information signal is a sequence of bits originating from a digital information stream. The modulation scheme modifies a carrier signal based on a set of modifications called the modulation alphabet. The modulation alphabet is a set of bits or groups of bits, called symbols, that represent the changes of the carrier signal used to carry the information signal. In a typical RF system there are different signal values that can be modified/modulated, such as amplitude, frequency and phase of the carrier signal. In optical channels, the optical intensity is most commonly modulated and the signal is received with photo-detectors that have a surface area usually many times larger than the optical wave-length to be detected. Hence the processed signal comes as an average of thousands of received wavelengths.

For simulations executed in this project, the popular modulations used for indoor and outdoor optical wireless channels are described in the following sections.

2.2.1 On-Off Keying (OOK)

On-Off Keying (OOK) together with Pulse-Position Modulation (PPM) (discussed in the next section) are some of the most popular techniques used in optical communications. Both are often used in commercial systems due to their power efficiency. OOK is one of the simplest binary modulation schemes, with only two values for signal changes. This scheme employs two different signal intensities, each with equal duration and probability during the symbol interval. In its

simplest form, one optical pulse represents one bit of the symbol sequence and the other bit is represented by the absence of the pulse [41].

There are two line coding schemes — Return-to-zero (RZ) and Non-return-to-zero (NRZ) — which are extensively used. In the NRZ line coding scheme, the pulse duration equals the bit duration, whereas in the RZ line coding scheme, the pulse duration equals to the half of the bit duration (the other half is equal to zero). The intensity of the pulse is $2P$ where P represents an average optical power. A random OOK signal is presented in figure 2.2.

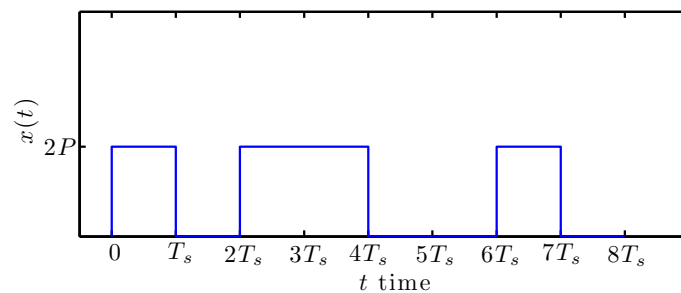


Figure 2.2: Random OOK signal [9]

In channel simulations with an Additive White Gaussian Noise (AWGN) channel (as typically done in signal processing models for communication channels), the analytical/theoretical probability of the bit error rate for OOK is given as:

$$P_{b,OOK} = Q\left(\frac{PRh}{\sqrt{R_s\sigma_n^2}}\right) \quad (2.3)$$

where the probability error function $Q(x)$ is given as:

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} \exp\left(\frac{-u^2}{2}\right) dx \quad (2.4)$$

Bit error rate (BER) is the number of erroneous bits received divided by the total number of bits sent, and it is used in that form for all further uncoded and coded modulation schemes. A general form of the equation is:

$$P_b = \lim_{N_s \rightarrow \infty} \frac{N_e}{N_s} \quad (2.5)$$

In the simulations, a finite number of bits N_s are used in a bit sequence. In case when a channel coding method is implemented in addition to the modulation, another measure of error is also used, namely the Symbol Error Rate (SER), which is calculated analogous to BER in 2.5, only using symbols instead of bits.

2.2.2 Pulse-Position Modulation (PPM)

Pulse-position modulation (PPM) is another popular technique used in optical communications. It is basically the coded version of OOK. In M-ary PPM, each symbol interval is divided into a series of M subintervals or chips. Information is transmitted by applying the positive optical intensity to one chip while the other chips remain zero. For example, in 4-PPM there are four symbols and each of them is represented by a different positive value for the chip duration in every interval.

Since all chips are non-overlapping in time, each symbol is orthogonal to all other symbols [13] and hence the symbols don't interfere.

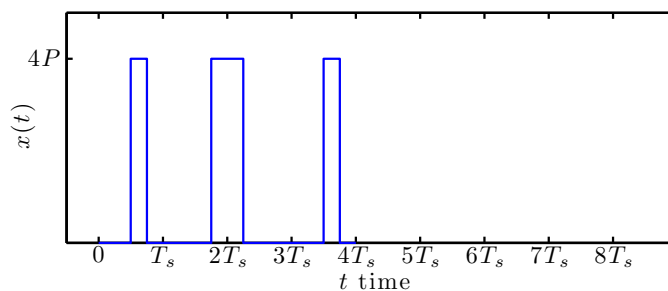


Figure 2.3: Random PPM signal [9]

The analytical probability of error, $P_{b,PPM}$ (shown in equation 2.13), where M is the number of different chips, R the bit rate and $Q(\cdot)$ the probability error function 2.4. It is an approximate theoretical value [13].

$$P_{b,PPM} = \frac{M}{2} \cdot Q\left(P\sqrt{\frac{M \log_2 M}{2R\sigma^2}}\right) \quad (2.6)$$

PPM is more bandwidth efficient modulation scheme compared to the binary scheme OOK. However, both schemes are considered to be spectrally inefficient in terms of the bandwidth [13],[41]. In terms of the optical power needed for the transmission, they are considered as power efficient schemes.

2.2.3 M-ary Phase Shift Keying (M-PSK)

The following modulations are widely used in radio communications, hence their possible usage and adaptation to optical wireless is investigated within this research work. While widely used modulations in optical wireless communications, such as OOK and PPM, belong to a group of baseband pulse modulation (since they operate in the baseband part of the spectrum), the modulations presented in this section belong to the bandpass or passband modulations. Due to constraints when working with optical signals (mentioned in equation 2.1 and 2.2) most of the conventional

RF modulation techniques can not be directly applied to optical signals [41]. The ones presented bellow, can be applied to optical signals and they are simulated.

M-PSK stands for M-ary Phase Shift Keying, where M is a number of symbols used in modulation, and in this case the number of different carrier phases. Hence it groups bits into a finite number of symbols that change the phase of the carrier signal. There are different types of PSK, depending on the number of symbols in the symbol alphabet.

These techniques are used as Subcarrier intensity Modulations (SCM) in order to better exploit the bandwidth of an analog optical signal. It means that a number of baseband signals are up-converted in frequency before modulating their characteristics, such as intensity, frequency or phase of the optical carrier. In order to satisfy the requirements for the optical communications, these signals have to be non-negative, so a certain DC-offset is added in order for them to comply with the requirements [41] This has another drawback - its consuming more optical power, hence systems with applied SCM become less power efficient. For example, in a typical AWGN channel with direct detection, BPSK and QPSK both require around 1.5dB more optical power than OOK or 2-PPM schemes.

2.2.3.1 Binary Phase Shift Keying (BPSK)

Binary Phase Shift Keying (BPSK) is the simplest PSK modulation, where the symbol alphabet consists of only two values, bit 0 and bit 1 [16]. Signal phases are separated by the angle of 180° . Each bit is mapped to one signal value or a symbol, which in the case of optical communications can only be positive values.

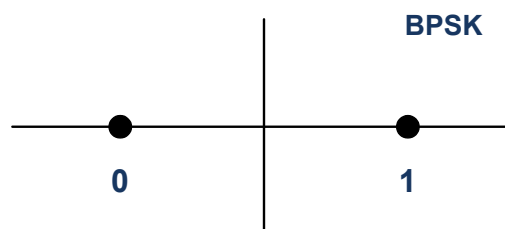


Figure 2.4: BPSK constellation diagram

BPSK values are shown on the constellation diagram in Figure 2.4. This diagram is used to represent the symbols from the modulation alphabet in a two-dimensional plane at sampling times, without taking into account the frequency of the carrier signal. It is convenient for representing digital modulations in a simple way. It can clearly be seen that BPSK has only two values that represent signal changes.

Theoretical value of the bit error probability for the BPSK modulation in digital systems is given in equation 2.8, with $erfc(\cdot)$ as the complementary error function, and E_b/N_0 signal to noise ratio (analogue to SNR) [27].

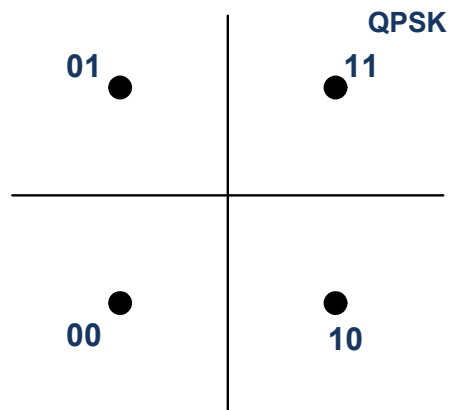


Figure 2.5: QPSK constellation diagram

$$P_{b,BPSK} = \frac{1}{2} \operatorname{erfc} \left(\sqrt{\frac{E_b}{N_0}} \right) \quad (2.7)$$

BPSK is not very spectral efficient scheme, since it doesn't utilize the available bandwidth efficiently. Its bandwidth efficiency can be improved when this scheme is applied to subcarrier modulations.

2.2.3.2 Quadrature Phase Shift Keying (QPSK)

QPSK stands for Quadrature PSK, and its alphabet consists of four symbols. Each symbol is represented with two bits and it results in the following alphabet: 00, 01, 10, 11. QPSK is less noise-prone than BPSK and its usually used to double the data rate compared with BPSK at the same bandwidth, or to keep the data rate the same at half of the bandwidth of BPSK [40]. However, in optical scenario, it acts similar to BPSK - hence it requires twice of the bandwidth than OOK and the optical power requirement is higher.

Theoretical bit error probability for QPSK modulation (from [15]), can be approximated to:

$$P_{b,QPSK} = \operatorname{erfc} \left(\sqrt{\frac{E_s}{2N_0}} \right) \quad (2.8)$$

with $\operatorname{erfc}(\cdot)$ as a complementary error function.

Figure 2.5 shows a typical constellation diagram of QPSK, with four symbol values equidistant from each other.

2.2.4 M-ary Pulse Amplitude Modulation (M-PAM)

M-ary Pulse Amplitude Modulation is modulation in which the amplitudes of pulses are varied according to the information signal that is going to be transmitted over the channel. Applied to an optical channel, it can be seen as the generalization of OOK to a set of M symbols [13]. Every PAM symbol with duration T_s is created by scaling the signal amplitude by M different factors. If the input data is uniformly distributed (signals with the same amplitude value equally distributed), the symbols are equally probable and the average power requirement, i.e. the power intensity for the eye-safety for the optical signal, is fulfilled.

PAM modulations are more bandwidth efficient than PSK modulations, however they are not used a lot in free space optical channels because of their sensitivity to channel loss and lower power efficiency [41].

2.2.4.1 4-PAM

This is a special case of PAM, where a two-bit modulator is used and it maps the signal amplitude to one of four possible levels. In figure 2.6 a random PAM signal is presented, with four different values for four different symbols that represent different pulse signals. Pulse amplitude levels can be any value as long as they meet optical channel constraints.

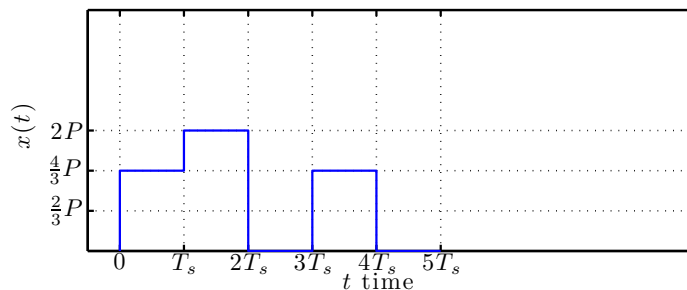


Figure 2.6: Random PAM signal [9]

Symbol error rate for 4-PAM (as in [15]) with symbols belonging to the alphabet $+1, -1, +3, -3$. Assuming equal probability of all the symbols and using the Gaussian probability distribution (as in [31]) results in the equation 2.9:

$$P_{s,4PAM} = \frac{3}{4} \operatorname{erfc} \left(\sqrt{\frac{E_s}{5N_0}} \right) \quad (2.9)$$

2.2.5 Multi-Level Quadrature Amplitude Modulation (M-QAM)

In Quadrature amplitude modulation two carriers are shifted in phase by 90 degrees comprising of in-phase and quadrature components. Then they are modulated using the Amplitude shift keying (ASK) digital modulation. Therefore, the resulting output is a combination of two modulation techniques, phase and amplitude modulation, PSK and ASK. PAM scheme and its implementation are basically the same for an optical channel as for the RF channel, ensuring that optical channel constraints are met.

The conventional way to adapt QAM to an optical channel is to add a DC bias to make the negative values become non-negative (so the channel requirement in 2.1 is met) and ensures that an average power stays within the certain limits (eye-safety constraint specified by channel requirement in 2.2).

This method is bandwidth efficient, but its drawbacks are that different states of modulation signal are closer together so the whole signal is more susceptible to noise. Both, PAM and QAM modulation schemes have more spectral efficiency due to transmitting more information per symbol, but they are less power efficient [13].

2.2.5.1 16-QAM

This higher-order modulation transmits four bits per symbol, and an example of its alphabet is given in [15]. Since it is a higher-order modulation, symbols are closer together. For analytical symbol error calculation, we use the following alphabet:

$$\{\pm 1 + \pm 1j, \pm 1 + \pm 3j, \pm 3 + \pm 3j, \pm 3 + \pm 1j\} \quad (2.10)$$

Assuming the Gaussian probability distribution of noise (additive white noise model) and that all symbols are equally likely, the symbol error rate can be calculated using equation 2.11:

$$P_{s,16QAM} = \frac{3}{2} \text{erfc} \left(\sqrt{\frac{E_s}{10N_0}} \right) \quad (2.11)$$

2.2.6 Comparison of modulation techniques

When comparing modulation schemes in most of the cases two factors are considered; the optical power efficiency and spectral efficiency (i.e. how effectively the available bandwidth is utilized) [13]. The optimal solution for the wireless optical channel would be to maximize both of them. Binary level modulations offer the best power efficiency while on the other hand, they have

lower bandwidth efficiency. Multilevel modulation schemes offer higher bandwidth efficiency, though they have a lower power efficiency.

In Figure 2.7 the comparison between different modulation schemes is given by taking into consideration the optical power gain over the simple OOK modulation scheme and the bandwidth efficiency of the given scheme [13]. As described later (in the discussion of results), typically phase-key shifting modulation techniques, such as BPSK and QPSK, require more optical power than OOK, about 1.5db. PAM and QAM as multilevel modulation schemes offer a better bandwidth efficiency, but they have lower power efficiency and increased sensitivity to the channel loss.

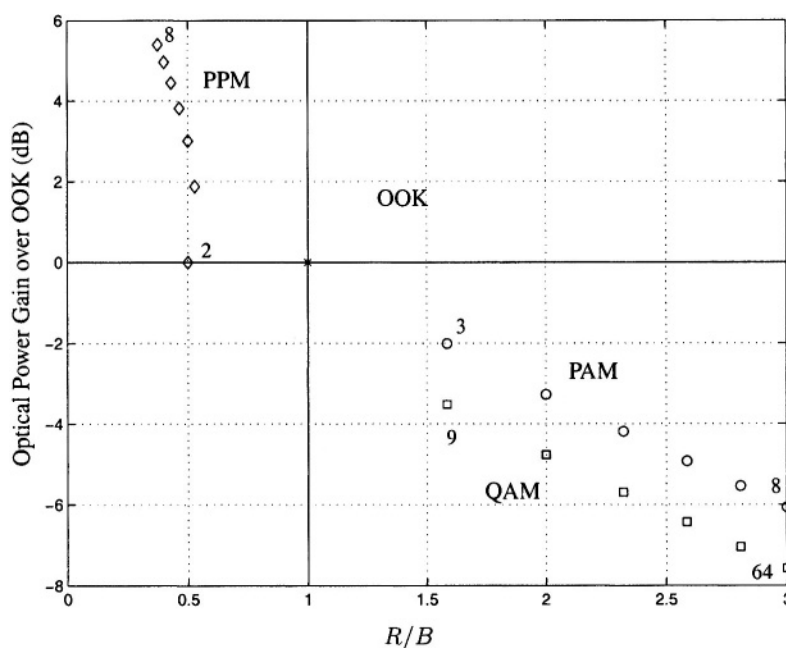


Figure 2.7: Optical power gain over OOK vs bandwidth efficiency [13]

2.3 Channel coding

Channel coding or Forward Error Correction (FEC) includes a set of methods used to improve performance of the channel by introducing redundant bits for detection and correction of possible channel errors. As more redundant bits are added, less information is sent through the channel, which leads to lower data throughput. Therefore, depending on the system requirements, there is always a trade-off between data throughput that aims at less delay on the channel and faster communication versus better error correction directed towards less errors and better channel performance [22],[21],[3],[37].

There are several types of FEC codes, depending on the classification of the same. They can be linear, using linear algebra and polynomial arithmetic, and non-linear. They can operate on chunks

of data, in which case they are called block codes, or on stream of data bits continuously inserting redundant bits for error detection and correction; in which case they are denoted as convolutional codes. Another characteristic sometime used in a description of some block codes is the cyclic nature of the code. Cyclic block code means that any cyclic shift of a codeword is also a codeword. Hence, when codewords are added together and shifted circularly, the result is still a codeword. This is a very practical characteristic.

There are various channel coding algorithms together with decoding algorithms, each of them has its own advantages and disadvantages for optical wireless communications. In this section three methods are presented and they are to be implemented in the software simulation; these are binary Bose, Chaudhuri and Hocquenghem (BCH) code, Reed-Solomon (RS) error correction code, Low Density Parity Check (LDPC) code. The use of RS together with a higher-order PPM scheme has shown an improvement in channel performance for strong attenuations [34] (due to fog or weather conditions).

2.3.1 Bose, Chaudhuri and Hocquenghem (BCH) codes

Bose, Chaudhuri and Hocquenghem (BCH) codes are powerful random error correcting codes [22],[3],[4],[37]. They belong to a group of cyclic block error correcting codes and they are able to correct multiple errors in the transmitted bit sequence. Decoding of BCH codes is possible in an easy way, using an algebraic method called syndrome decoding [4],[37],[30]. There are two versions of this code: binary – that are built upon binary fields with elements 0, 1; and non-binary – that are built upon binary fields with more than two elements. A widely used non-binary BCH code is Reed-Solomon (RC) code.

According to the definition of binary BCH codes [4],[37], for every $m \geq 3$ and $t < 2^{m-1}$ there is a binary BCH code that has the following parameters as in 2.12.

$$\begin{aligned}
 \text{Block length: } n &= 2^m - 1 \\
 \text{Parity check bits: } n - k &\leq mt \\
 \text{Minimum distance: } d &\geq 2t + 1
 \end{aligned}
 \tag{2.12}$$

Codewords belong to a finite field called Galois field that contains a finite number of elements as well as other properties that are convenient for building good codes [29]. Information bits are represented in the form of a polynomial, and the codewords are built by taking the remainder after division of that polynomial by a chosen generator polynomial. The generator polynomial is selected so it gives the code its cyclical characteristics – all codewords are multiples of the generator polynomial.

BCH codes are typically denoted as $BCH(n, k)$ with n as the total number of bits, and k as the number of information bits. For example, $BCH(31, 16)$ has codewords of 31 bits length, 15 check

bits, and it is able to correct 3 errors, which means a minimum distance between codewords is 7. Information bits get zeros appended, in the number equals to the degree of the generator polynomial (here 15). Then they are divided by the generator polynomial (*XORing* elements of both of them) and the remainder of polynomial division becomes the sequence of check bits. The codeword is built by appending that sequence to the information bits. In this way the information and check bits can be recognized in the resulting codeword; and this is called *systematic encoding*.

By dividing the codeword by the generator polynomial again, it can be checked if there were errors in transmission. If the remainder is zero, there are no errors in the received codeword; if the remainder is non-zero, the received codeword is erroneous. This remainder is called the *syndrome* and it is used in the decoding algorithm for the location of the erroneous bits and correction of the same.

Decoding is performed in a couple of steps. First the *syndrome* is computed from the received codeword and the generator polynomial. Then the error location polynomial is found from a set of equations derived from the syndrome. And lastly, the error location polynomial is used to detect the erroneous bits and correct them. Further details about the BCH decoding algorithm can be read in [30].

2.3.2 Reed-Solomon (RS) block codes

Reed-Solomon block codes are non-binary BCH codes. They are capable of correcting burst errors as well as random symbol errors. In RS codes, information bits are gathered in blocks, that have a fixed length and can further be divided into symbols, which are m -bits long [21]. Since this code uses symbols (information bits), error correction information is added in the form of parity symbols. A RS coded word consists of information symbols together with attached parity symbols.

The notation $RS(n,k)$ means n is the total number of symbols, or one codeword, and k is the number of data or information symbols being encoded. Maximum number of errors that can be corrected by the RS code is $t = \lfloor (n - k)/2 \rfloor$ within one code block [21]. If more than t errors occur, the code will not perform correctly and symbols may be decoded incorrectly, hence the original information can be lost.

RS codes are based on finite fields, also called Galois fields $GF(2^m)$. Originally, k message symbols are viewed as coefficients of a polynomial $p(x)$ over a finite field. The field has a property that any defined operation on the finite field element results again in another element of the field. In addition, it has a finite number of elements, for example the finite field of integers. Detailed explanation of RS codes can be read in [22],[21]. The error probability for the RS decoded symbol is given in equation 2.13:

$$P_{s,RS} = \frac{1}{2^m - 1} \sum_{j=t+1}^{2^m-1} j \binom{2^m - 1}{j} P_s (1 - P_s)^{2^m-1-j} \quad (2.13)$$

where P_s stands for the symbol error probability of the deployed modulation scheme. As previously stated, RS codes can correct up to t symbols, no matter how many bits get corrupted in any of these t symbols. That makes them suitable for burst error correction. However, if the number of erroneous symbols exceeds t , even if for only one bit error $t+1$, the codeword will not be correctly decoded and the RS algorithm fails. Hence using RS codes on symbols has its advantages, but also disadvantages.

RS codes have gained a lot of attention in optical wireless communications, especially in FSO links [24], [11], [17]. There are low-complexity hardware RS decoders available, that are well tested and can achieve high data rates [8], which makes them one of the most popular channel codes for optical wireless applications.

2.3.3 Low-Density Parity-Check (LDPC)

Low-density parity-check (LDPC) codes are linear error-correcting codes, which means that they can be decoded in a time that is linearly proportional to their block length. They are called *low density* due to the fact that their parity-check matrix contains only few 1's compared to the number of 0's [21],[30]. Also their performance is very close to the channel capacity of many different channels. When evaluating bit error performance on the channel, these codes exhibit a typical characteristic abrupt drop of the BER curve at a certain level of SNR (signal to noise ratio).

The codeword is built with a help of the low-density parity-check matrix $H_{(n-k) \times k}$, where n is the length of the codeword and k is the number of information bits. From this matrix, a new generator matrix is created according to the $G = [I_k|P]$. By multiplying the generator matrix with the sequences of all possible bit combinations of k bits in the length of 2^k bits, new codewords are generated. For each sequence a new codeword in the length of n bits is generated.

LDPC codes usually work with very long codewords, which makes processing them (encoding and decoding) very slow. However, with improved hardware development and decoding algorithms, processing times have reduced and it is possible to design more time efficient LDPC codes. They are adopted as a standard for Digital Video Broadcast - Satellite (DVB-S2) communications, where there can reach data rates close to 1Gbit/s [32]. Their usage in such an environment together with their aforementioned advantages makes them interesting for investigation and a candidate for usage in the optical wireless communications.

3 Software Architecture

The framework for the simulator is organized in different logical sections that are implemented in the form of modules. Each of the modules to be simulated is connected over a common simulation interface. The basic software architecture is shown in figure 3.1, in the form of a simple block diagram. It consists of three main blocks: Framework, Configuration and Modules.

The first block, namely the Framework, contains main routines for the setup of the whole simulation; it is responsible for creation of modules, also called building blocks, verification of allocated memory for the modules and the general simulation flow. Those routines are separated in again another abstract sections for the purpose of better understanding and they are referred to as Creation, Verification and Simulation section. They are executed in their respective order to reassure the correct flow of the simulation.

The second block, namely the Configuration block, is responsible for set up of configuration parameters for the simulation. It obtains those parameters from a file in XML format and they are extracted with help of a small parser function implemented as a separate module – a separate module is written for this purpose, instead of reusing existing XML libraries to avoid dependencies on libraries which may cause platform independence issues. Those parameters represent the values that each module needs for its execution, such as input and output values, and other constant parameters needed for the simulation. They are also used for dynamical creation of class objects in each module. That way, if a change of parameters is needed, only a change in the XML file is required – no additional compilation is needed (assuming the previous build included the additional modules).

In the third block, referred to as the Modules block, is where all the modules are included. Each module implements a block of functionality (in a class) and hence they are mostly independent from each other. Each module represents one of the communication channel elements from the block diagram of optical communication channel in Figure 2.1 from Chapter 2. For example, a module can be a random numbers generator, CRC checksum generator, modulator/demodulator, or encoder/decoder. They are linked together via a common Simulation Interface class, from which each of them inherits. Input and outputs are chained via a linking class that links all the modules together and each of them includes it. This will be discussed in detail in the following section, and a detailed description of all the modules is given in Chapter 4.

The implemented simulation framework, as mentioned, is organized into separate logical sections. Each section is partitioned as a folder and contains a number of C++ classes that implement

different functionalities. Classes are organized into header files with *.h* extensions and source files with *.cpp* extensions. In C++, header files are responsible for declaration of methods, data members, overloaded operators and other functionalities needed for the class to work properly. Corresponding to header files, source files are responsible for the implementation of everything what is declared in the header file. The exception to this is for template functions, that are special kind of functions that operate with generic types and therefore they require the implementation to be in the header file.

Here is a short overview of different groups in which the software is organized and the classes that are contained in those sections:

Framework Contains classes that are responsible for the creation of the simulator framework and for the setup of the entire simulation. These classes are contained within the following files: `Framework.h`, `Framework.cpp`, `LinkModules.h`, `LinkModules.cpp`, `SimulationInterface.h`

Configuration Contains classes that provide configurability and implements parsing function-

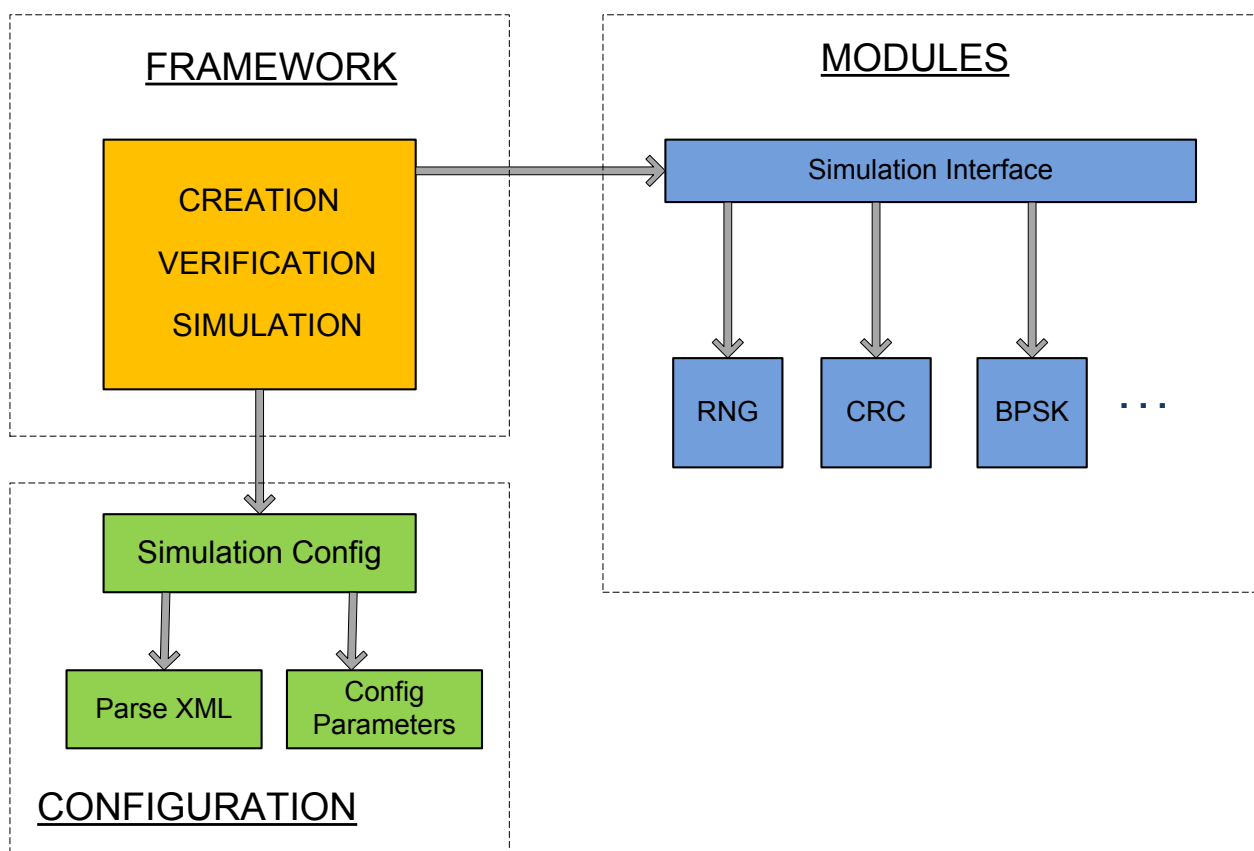


Figure 3.1: Basic Software Architecture

ality. With their help the configuration parameters from the local XML file can be extracted, converted, and made available to the Framework. The following files provide the classes for this functionality: `ParseXML.h`, `ParseXML.cpp`, `SimulationConfig.h`, `SimulationConfig.cpp`.

Channel Classes in this group implement the communication channel, i.e. the part between the reception and transmission segments of the channel. It contains the following files: `AWGN.h`, `AWGN.cpp`

CRC checksum Implements cyclic redundancy checksum computation algorithm. For that purpose it contains the following classes: `CRCgenerator.h`, `CRCgenerator.cpp`, `CRCTable.h`, `CRCTable.cpp`

RandomGenerators Group with random generators for the generation of pseudo-random bit sequences, employing either uniform or normal probability distribution. It contains the following classes: `RandomGenerator.h`, `ParametersWH32`, `NormalVecWH32.h`, `NormalVecWH32.cpp`, `BinaryVecWH32.h`, `BinaryVecWH32.cpp`

Modulators Contains the implementations of all modulators used in the simulation. The modulation can be switched between them. The classes included are the following: `BPSKmod.h`, `BPSKmod.cpp`, `OOKmod.h`, `OOKmod.cpp`, `PPMmod.h`, `PPMmod.cpp`, `QPSKmod.h`, `QPSKmod.cpp`, `PAMmod.h`, `PAMmod.cpp`, `QAMmod.h`, `QAMmod.cpp`

Demodulators Contains the implementations of all demodulators corresponding to the modulators used in the simulation. Demodulators and Modulators should be invoked in pairs for the particular simulator, namely, if one type of modulator is picked, its corresponding demodulator needs to be picked. The following classes are demodulators: `BPSKdemod.h`, `BPSKdemod.cpp`, `OOKdemod.h`, `OOKdemod.cpp`, `PPMdemod.h`, `PPMdemod.cpp`, `QPSKdemod.h`, `QPSKdemod.cpp`, `PAMdemod.h`, `PAMdemod.cpp`, `QAMdemod.h`, `QAMdemod.cpp`

Encoders Group of encoders that implement error correcting techniques on the channel. Some of them are yet to be implemented. It contains: `BCHencoder.h`, `BCHencoder.cpp`, `RSencoder.h`, `RSencoder.cpp`, `LDPCencoder.h`, `LDPCencoder.cpp`.

Decoders Group of all decoders corresponding to the channel encoders. They go in a pair with encoders and have to be invoked in the same simulation. Some of them are yet to be implemented. The following are meant to be in this group: `BCHdecoder.h`, `BCHdecoder.cpp`, `RSdecoder.h`, `RSdecoder.cpp`, `LDPCdecoder.h`, `LDPCdecoder.cpp`

The following is a description of each block contained in the actual software architecture, organization of different modules and the connection between them.

3.1 Framework

Design of a good and functional framework that satisfies requirements from Chapter 1 is one of the most important parts of the software architecture for the simulation to work properly. The first step was to choose the appropriate data structure to store the objects from different modules. It had to have constant linear access times, be easy to access and add new elements, and easy to manipulate. Also, it would be responsible for creation of modules and setup of the input and output values. Ultimately, the data structure picked was the `std::vector` container from the Standard Templates Library (STL) [1],[35].

Vector containers are implemented as a dynamic array of objects, with extended functionalities. They store data in contiguous locations and they can be accessed with iterators or using an offset to the pointer value to an element [1]. They can store elements of any type (also class objects) and they are easily expandable/resizeable. They have linear access times. From that reason this structure is chosen the core data structure for this project, among other structures such as list, dequeue, map, etc.

As in Figure 3.1 the modules used in a simulation have to be created in the order as they appear in the communication channel. They are created dynamically, each as an object of its corresponding class. Their input and output data sequences are connected via the `LinkModules` class, whereby the output data sequence of the previous module becomes the input data sequence for the next one. Therefore the modules have to use the same data type.

To keep track of the modules and the sequence in which they need to be run, the vector container is used. The elements of the vector container are objects of `SimulationInterface` class, which is an abstract class inherited by all the modules. In the framework class, the elements of the vector are populated with `SimulationInterface` objects and when iterated through them one simulation is executed with current modules. Simulation can have number of runs, usually 10.000 and more.

The framework is responsible for setting up a stable structure where different modules can be switched and easily used for different simulations. This is realized over the communication with the user, who has to choose which module he/she wants to use in the current simulation. Let it be noted that some modules come in pairs, such as modulators with demodulators, followed by encoders with decoders. A switch is implemented with a set of branch conditions using *if/else* statements that check the user input and match it with the possible choices.

3.1.1 Simulation Setup

In order to run a simulation, an object has to be created and verified for errors. This is done in the Framework class. All of the modules for the current simulation are created using the method `void CreateModules(string modulationChoice, string codingChoice)`. It takes

two arguments of *string* type; first one `modulationChoice` refers to the choice of the modulation scheme that should be used, the second one `codingChoice` refers to the choice of the coding scheme that should be used. The users are prompted for both values in the `main.cpp` source program and depending of what the input is, the corresponding method is executed. As mentioned in the previous section, for this purpose an *if/else* branch condition is implemented and if the condition is true, the user is informed about his choice over the screen output and methods corresponding to that module are executed.

The snapshot of the piece of code that implements the switch functionality is given in Figure 3.2.

```
void CFramework::CreateModules(std::string modulationChoice, std::string
codingChoice)
{
    if (modulationChoice == "BPSK") {
        std::cout << "You chose BPSK modulation" << std::endl;
        modulationBPSK();
    }
    else if (modulationChoice == "QPSK") {
        std::cout << "You chose QPSK modulation" << std::endl;
        modulationQPSK();
    }
    else if (modulationChoice == "OOK") {
        std::cout << "You chose OOK modulation" << std::endl;
        modulationOOK();
    }
    else if (modulationChoice == "PPM") {
        std::cout << "You chose PPM modulation" << std::endl;
        modulationPPM();
    }
    else if (modulationChoice == "PAM") {
        std::cout << "You chose PAM modulation" << std::endl;
        modulationPAM();
    }
    else if (modulationChoice == "QAM") {
        std::cout << "You chose QAM modulation" << std::endl;
        modulationQAM();
    }
    else {
        std::cout << "You choice is not valid, try again!" << std::endl;
        modulationBPSK();
    }
}
```

Figure 3.2: CreateModules class definition

Also depending on user's input different member functions are called and each of these function is responsible for the following:

- I Allocation of all objects used for the current simulation.

- II Setting input pointers to corresponding values (the output of the prior module is set to the input of the following module, with the first module generating the output).
- III Pushing the data into the vector container in the order they are supposed to be linked on the channel.

As a result of these steps, the vector container is populated with all objects needed for the current simulation and the pointers are setup to point to the right input and output sequences. The next step is to verify if those objects are allocated properly with enough memory for each of them and if the pairs of input/output values are set up correctly.

3.1.2 Error verification and exceptions handling

While creating the objects and allocating memory for them dynamically it is possible that there is not enough memory for allocation or some other errors, that can make the simulation unsuitable for execution. These errors need to be handled during the creation itself, within the `void CreateModules(string modulationChoice, string codingChoice)` method. This is done by wrapping the code inside the `CreateModules` function in a *try/catch* block that will deal with exceptions right at the declaration and allocation of objects.

In case that an error related to a lack of memory is detected, a user is informed over the screen message that contains type of an error and a text that there is no enough memory to continue execution. The program exits with code `exit(1)` and the simulation won't be executed. In case that any other type of error occurs that can prevent the simulation from executing, the same procedure is executed and the program execution stops. That way, the simulation will proceed only if all objects are created properly and there is enough memory to store them.

After the objects are created, they are pushed into the vector, hence it has to be asserted that they are created in the correct order and that there are no empty elements inside of the vector. For that purpose, the method `bool Verify()` is implemented within the Framework. It iterates through all the elements of the vector containing the modules and executes the method recursively every time a new iteration is performed. The results are stored in a variable that has a boolean type and can take `true` or `false` values. If the verification results in a `true` value, a message is printed to the screen and only then simulation is executed. If it results to a `false` another message is printed to the screen, and the `Simulation` method is not being executed.

The `bool Verify()` method is inherited from the abstract class `CSimulationInterface`, whereby in the abstract class it is only declared, and its implementation is provided in every class that inherits `CSimulationInterface` class, according to the rules of C++ programming language [1],[35].

All these error checks are used to detect possible errors (related to the creation of objects for different modules) are detected right at the beginning of the simulation, in a fail-fast manner.

If the creation of modules and verification completes, the simulation should deliver only useful results.

3.1.3 Simulation flow

In the case of successful error verification, method `void Simulate()` is executed and it provides the simulation using the chosen modules. This method contains two loops; the outer one that iterates over different SNR (Signal to Noise Ratio) values that are denoted as *current simulation parameter* values and the inner one that performs a number of simulation runs for the current SNR value. The maximum number of iterations is given in the XML configuration file.

After each iteration with the current simulation parameter, bits that are erroneous during the channel transmission are calculated and the counter for the bit error rate is incremented. These bits are calculated by comparing the output sequence of the first module in the communication channel (typically Random Generator) and the output of the last module (typically Demodulator or Decoder). The number of erroneous bits is divided by the block length of the input sequence, and also by the number of simulation runs to calculate the average bit error rate.

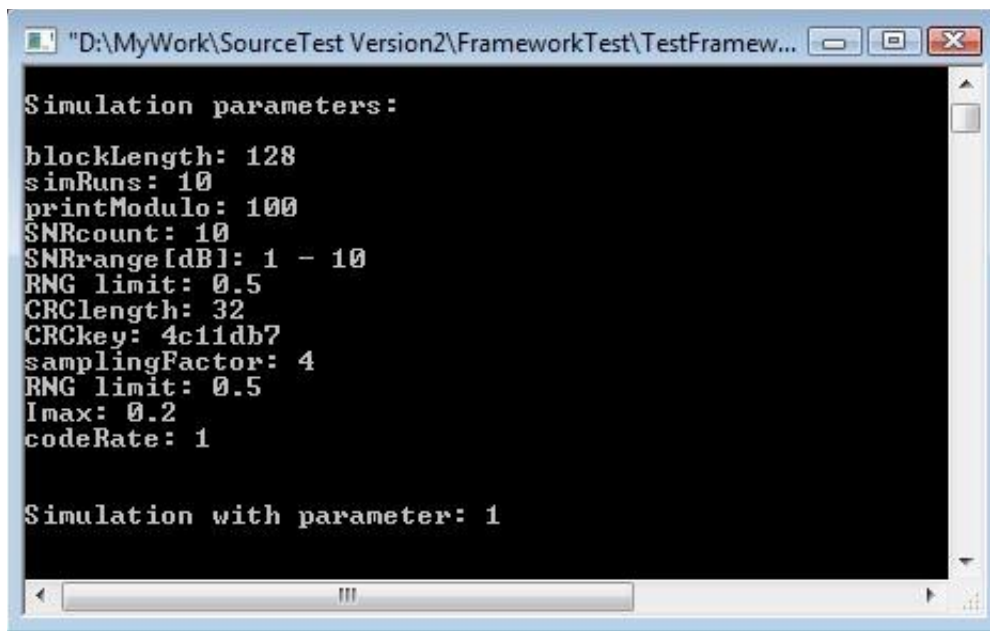
The output format of the BER (bit error rate) values is described in detail in the following section. They are written to a file in CSV format together with the corresponding SNR value for each iteration. They are also printed on the screen in the form of console output.

The `void Simulate()` method is inherited from the abstract class `CSimulationInterface` and it is implemented appropriately in all the other modules. So when it runs in the Framework class, it actually passes through all modules declared in the `CreateModules(string modulationChoice, string codingChoice)` method and executes the simulation on each of them in the order as they appear.

3.1.4 Output format

The results of the simulation are provided in different output formats. There are two possible formats provided within the software implementation; the first one is on the screen console (or command prompt) during the execution of the program, while the program is running, results are written on the screen. The second one is in form of the CSV-file (Comma Separated Values), where results are stored for later use. One snapshot of the output printed out on the console during the simulation execution is given in Figure 3.3.

First the user is prompted for input on the choice of modulation and coding method he/she wants to use. Possible combinations are given in a separate file (README). This choice is then confirmed on the next line, followed by the information about the verification status. If successful the simulation continues, whereas if unsuccessful, the simulation won't be executed.



```

Simulation parameters:
blockLength: 128
simRuns: 10
printModulo: 100
SNRcount: 10
SNRrange[dB]: 1 - 10
RNG limit: 0.5
CRClength: 32
CRCkey: 4c11db7
samplingFactor: 4
RNG limit: 0.5
Imax: 0.2
codeRate: 1

Simulation with parameter: 1

```

Figure 3.3: Console output

After the value of the error rate is calculated for every simulation parameter it is printed on the console, preceded by the value of the simulation parameter. Every 100 simulations there is information provided about the simulation progress. This is done so that the user can stay informed if the simulation takes long to run. The last line presents some text indicating the end of the simulation.

The parameters stored in the output file in the CSV-format are written in the form of four columns as in 3.1:

$$\begin{array}{r}
 7, 0.0007775, 0.000772674815, 2 \\
 8, 0.0002018, 0.000190907774, 1 \\
 9, 3.26e - 05, 3.36272284e - 05, 1 \\
 10, 3.6e - 06, 3.87210822e - 06, 0
 \end{array} \tag{3.1}$$

The first column lists the values of the current simulation parameter SNR for each iteration; its typically 10-20 values. The second column represents the Bit Error Rate (BER) values computed for all simulation runs for the corresponding SNR value. Third column represents the BER values from the theoretical/analytical computation. From the BER rate another error rate can be calculated, namely the Frame Error Rate (FER) if the chunks of data are grouped into frames, then the BER is divided by the frame length. Its values are in the forth column of Figure 3.1 which is an example of a part of CSV file for SNR values equal to 10, 11, 12, 13.

Hence, not only are the simulation results calculated and exported to CSV files, but also the theoretical values of BER (sometimes called SER - Symbol Error Rate, since in most of the cases we are dealing with symbol alphabet) are computed and written into files, in the third column in 3.1. Hence their values can easily be compared and used for graphical presentation if needed.

3.1.5 Visualization in Matlab

Results of the simulations are also presented graphically in the form of Matlab plots, where they can be viewed better and also conclusions about their behavior can be made. For this purpose, a small Matlab script is created that reads the CSV file using the method `csvread(filename)` and plots the BER from values the simulation and the theoretical BER vs the SNR values. An example of such a plot is given in Figure 3.4.

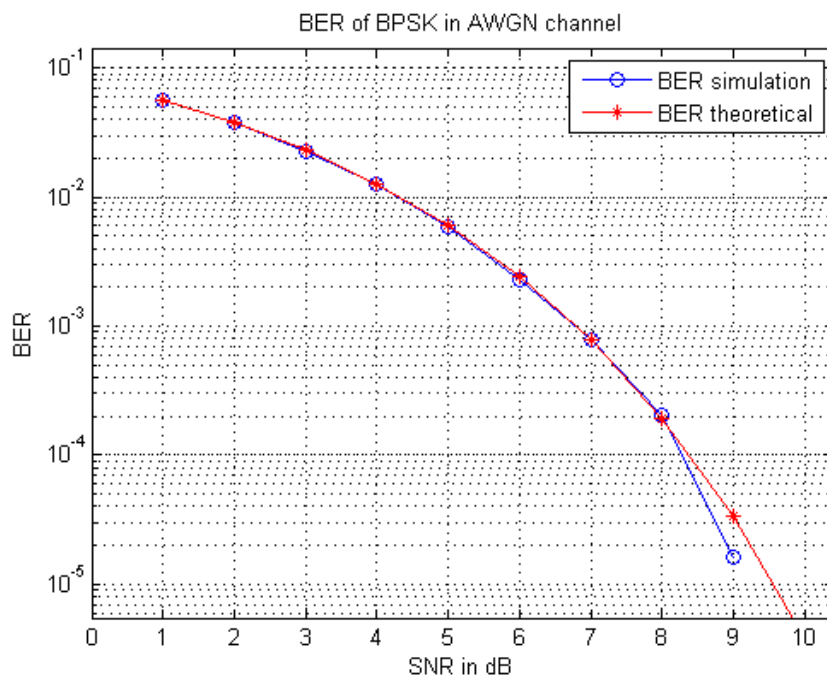


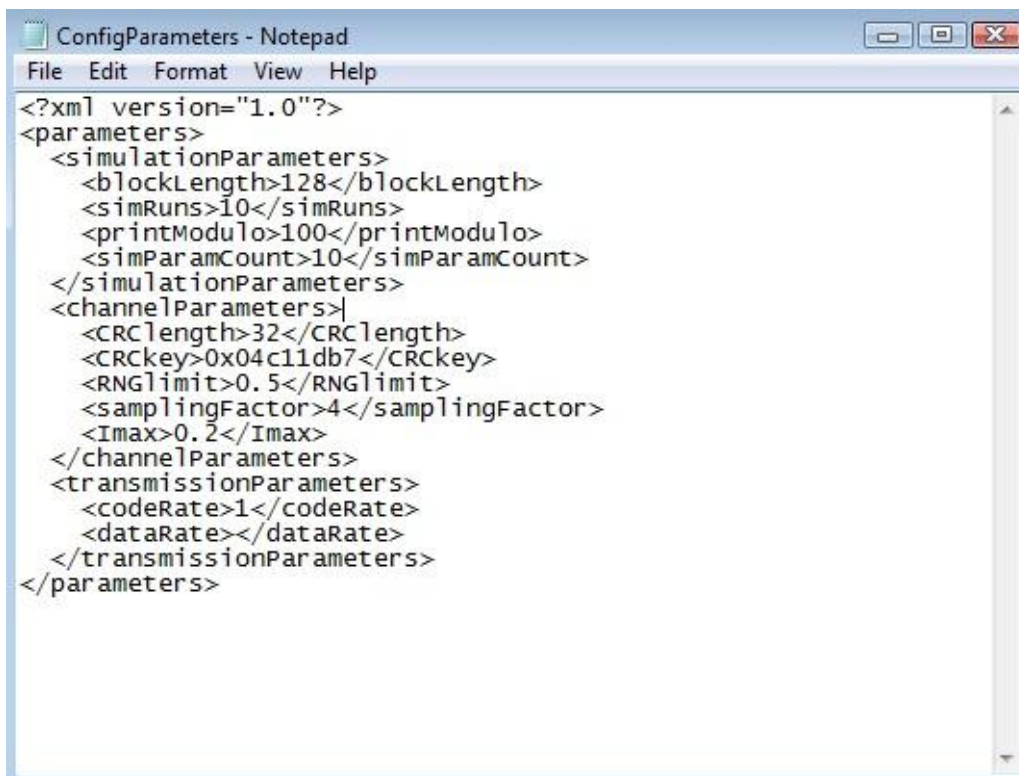
Figure 3.4: BER vs SNR plot

3.2 Configuration

Data for the configuration is given in the XML file `ConfigParameters.xml`. It is imported in the Framework using the file `SimulationConfig.cpp` that declares all configuration parameters and for that it invokes methods from the class `ParseXML`. These methods extract the data from the configuration file in form of strings and convert them to relevant data types. If any

parameter value has changed, it needs to be changed only in the XML file and will be automatically reassigned in the `SimulationConfig.cpp` during the run time, which means it doesn't need to be compiled again, which saves time.

In case that another parameter is added to the configuration file though, it also has to be declared in `main.cpp`, and passed to the `Framework` class, which will use it to setup the modules when appropriate. Also SNR values are defined here in the form of an array. A short example of the XML file is given in Figure 3.5.



```
<?xml version="1.0"?>
<parameters>
  <simulationParameters>
    <blockLength>128</blockLength>
    <simRuns>10</simRuns>
    <printModulo>100</printModulo>
    <simParamCount>10</simParamCount>
  </simulationParameters>
  <channelParameters>
    <CRCLength>32</CRCLength>
    <CRCKey>0x04c11db7</CRCKey>
    <RNGLimit>0.5</RNGLimit>
    <samplingFactor>4</samplingFactor>
    <Imax>0.2</Imax>
  </channelParameters>
  <transmissionParameters>
    <codeRate>1</codeRate>
    <dataRate></dataRate>
  </transmissionParameters>
</parameters>
```

Figure 3.5: XML file with simulation parameters

In the following, the configuration parameters for the current simulation setup are given and its function is explained shortly.

- **const char *XMLfilePath**
Path for the XML configuration file
- **const unsigned long blockLength**
Length of a block of data i.e. input bit sequence
- **const unsigned long simRuns**
Number of simulation runs per simulation parameter
- **const unsigned long printModulo**
Number of iterations after a status message is printed out

- **const int simParamCount**
Number of simulation parameters (SNR)
- **const unsigned long CRClength**
Length of the CRC checksum key
- **const unsigned long CRCkey**
CRC checksum key value
- **const int samplingFactor**
Sampling factor for modulation
- **const unsigned long sampleRate**
Sample rate for modulation
- **const float codeRate**
Code rate of the channel coding scheme
- **const float RNGLimit**
Limit value for Random generator
- **float SimParam[]**
Array containing current simulation parameter (SNR) values

These configuration parameters and much more can be used in the simulation depending on which modulation technique is employed and whether a coding technique is used on the channel or not. Adding of new parameters or modifying the existing ones is made easy with this configuration setup.

3.3 Modules

According to the block diagram from Figure 3.1 Modules can be seen as a separate section connected to the Framework over the Configuration block.

Every block in the transmission channel (from the block diagram in Figure 2.1 in Chapter 2) is implemented as a separate class, also referred to as a module. On an abstract level, a module can represent some functionality, or a routine that executes certain methods and/or algorithms. In the simulation program, modules are developed as either basic building blocks, inherited classes or preprocessor macros, and they are used in different parts of the program.

Basic building blocks are considered the classes that can be either base classes, possibly abstract base class, derived classes or stand-alone/regular classes that are neither inherited nor derived classes in C++. For example, `CSimulationInterface` is a basic abstract class and does not contain any implementation, and it is inherited by most of the modules. On the other hand, classes such as `CRandomGenerator`, `CBPSKmod`, `CAWGN` and other elements of the communication

channel are all derived classes that inherit the abstract base class. And the third group, classes such as `ParseXML` or `CLinkModules` are neither inherited nor derived, and they can be included within other classes implementations. In addition, `CLinkModules` is a template class, whose template parameter value can be set in any other module including it.

Different kind of modules, in an abstract way, are preprocessor macros, declared between `#define` and `#endif` statements, so called guards. As an example, within the Random Generator module a step calculation for the generator is implemented as a macro.

Description of modules and the methods within them is given in detail in the following chapter.

4 Description of Simulation Modules

In this chapter a detailed description of the modules is given as well as the description of all methods that make them functional entities. As mentioned in the previous chapter, modules are implemented as classes in C++, which can be either base classes or derived classes that inherit the base class. Beside that, also other classes with no inheritance can be used.

Modules which need to be simulated all inherit from a common interface class and accessed through it. A new module can be added independently when it is implemented as a new class, and it can inherit one of the existing classes or it can be created as a macro module also within another module. Modules which don't inherit from the common interface class are accessed through the Framework and usually provide a specific function to the Framework (e.g., `SimulationConfig` and `ParserXML`). More details are provided below.

4.1 Main modules and link interface class

The first block in figure 3.1 from chapter 3 represents the *Framework* and how it is connected to the *Configuration* block via the `SimulationConfig` module. The framework utilizes the configuration block to retrieve configuration parameters from an XML file on disk. This is described in more detail below.

4.1.1 CFramework

This class contains methods to set up the framework and prepare for simulation. This includes creating the necessary simulation modules – elements of the communication channel; linking their inputs and outputs; and finally verifying that the simulation is ready to run. Also within `CFramework` the simulation method is executed that iterates through all current modules in the communication channel. Its object is allocated in the `main.cpp` source file at the beginning of the simulation. In order to execute the simulation with all other modules, `CFramework` declares a vector container and populates it with elements, which are objects of other classes. The vector takes elements of type `SimulationInterface` and its declared as following:

```
std::vector<CSimulationInterface*> m_ModuleList
```

Core methods implemented within this class are:

```
CreateModules(string modulationChoice, string codingChoice)
Verify()
Simulate()
```

Beside that, all configuration parameters are declared in this module and their values from the configuration file assigned correspondingly. Also input and output pointers are defined as following:

```
CLinkModules<bool>* m_InputData
CLinkModules<bool>* m_OutputData
```

which means they are instances of the `CLinkModules` template class and have boolean type (zeros and ones).

Methods

```
CFramework()
```

- Constructor of the class, responsible for assigning the data members. It also creates new input and output object of the interface class `CLinkModules`, by allocating their memory dynamically with the `new` operator. Input and output sequence have the length of `blockLength` from the configuration file.

```
~CFramework()
```

- Destructor of the class, it deletes all pointers to allocated memory for vector objects and assigns them to `NULL`. It also deletes the allocated input and output data.

```
void CreateModules(string modulationChoice, string codingChoice)
```

- Prompts user for input for the `modulationChoice` and `codingChoice` parameters, then uses them in *if/else* statement to decide which method should be executed. This method can be either different type of modulation or coding, or both of them combined together, depending on the user input. Once when the appropriate method is called, objects of each module are created dynamically, then output values of the previous module are assigned to the input values of the current one, and finally those objects are pushed into the vector container in the order they appear in the simulation. The input/output values are assigned with help of the `Get` and `Set` accessor functions from the `CLinkModules` class.

- **Return value:** `void`

```
bool Verify()
```

- Checks for errors in allocated objects in the vector container and in case they exist it returns the `false` value. It is executed after the `void CreateModules(string moduleChoice, string codingChoice)` method is called, and it iterates through the whole vector container reassuring that every element is verified for its existence. It prints

out *"Successful verification"* in case that return value is `true` or *"Verification failed"* in case it returns `false`.

- Return value: `bool`, `TRUE` or `FALSE`

`void Simulate()`

- Iterates through all the modules i.e. communication channel blocks within two loops. First one takes the current parameter value which equals to the current SNR value, `ChSimParam` and inside of the loop it executes another loop with the number of simulations as an iterator, `SimRuns`. That way, for every SNR value, number of simulations are performed, starting with the random bit sequence generation, and then applying other modules to it. After looping through all SNR values, the bit error rate (BER) is calculated, as well as the frame error rate (FER) for one block length. During the simulation, messages are printed to the screen and also the error rate values are written to the output file. This is all implemented `Simulate()` method.
- Return value: `void`

4.1.2 CLinkModules

This is a template class, it contains pointers for input and output data sequences and provides memory allocation methods for them. It also makes input and output vectors accessible to the public interface. In addition to that this class implements `CompareBits` method that computes the number of different input and output bits for the bit error calculation.

This module, `CLinkModules` is included in all other modules that belong to communication channel and its objects are allocated in every one of them. The reason for that is to keep all input and output sequences of the same type which makes linking them together much easier, as well as the adding another channel block, if needed. It also makes the whole simulator more general and applicable to various scenarios.

Its data members are pointers to input and output sequence, `T* m_pInput`, `T* m_pOutput` respectively and an integer variable for length of the sequences, `unsigned long m_length`.

Methods

`CLinkModules(unsigned long length)`

- Constructor of the class, responsible for assigning the data members to input arguments and allocating memory dynamically for the input and output sequences `T *m_pInput` and `T *m_pOutput` in the corresponding length.

`~CLinkModules()`

- Destructor of the class. It takes care of the deletion of the allocated memory, after which it assigns all pointer to NULL value.

```
SetInputPtr(T *dataInput)
```

- Access method to set the input pointer to the input data. That way it is possible to assign input values from outside of the class, which makes it publicly accessible. Since its type is templated, it can take any data type as an argument.
- Return value: void

```
T* GetOutputPtr()
```

- Access method, returns the output pointer for the output data. Similar to set access method, it allows the outside of the class to communicate with it. It can be of any data type, depending how it is called during the execution.
- Return value: pointer to the memory allocated for the output data.

```
unsigned long CompareBits(const CLinkModules<T> *inputData)
```

- This method implements the calculation of erroneous bits. It does that by comparing input bit sequence from the `inputData` array of type `CLinkModules` with output bit sequence of the same type (here, member of the class). Every time a different bit value is encountered, the counter is incremented. Hence it returns the number of different bits, which represent the erroneous bits for the bit error calculation.
- Return value: unsigned long

Beside that, class `CLinkModules` contains necessary methods for operators overloading.

4.1.3 CSimulationInterface

This is an abstract class, declared only in a header file, and it contains pure virtual methods whose implementation is then implemented in all of its derived classes. All modules from the communication channel (generators, modulators, encoders..etc) inherit this class; hence it is an interface class that connects everything together.

Methods

```
virtual bool Verify()=0
```

- Abstract classes methods are declared by appending `=0` to the end of the method and putting the word `virtual` at the beginning. Where implemented, it performs error verifications as described in the previous chapter 3.
- Return value: virtual bool

```
virtual void Simulate(float SimParam=0.0)=0
```

- The same here. Where this method is implemented, it executes the simulation and error counting for the calculation of bit error rate, as described in the previously.
- Return value: `virtual void`

4.2 Other modules

Up to now, the classes contained within the Framework module are described and their methods are given. Other modules and its classes implement the rest of the optical communication channel and represent different functionalities. They mostly inherit the `CSimulationInterface` class and they have their own virtual destructor to reassure that the destructor of the corresponding derived class is going to be executed, not one of the base class. This is important, since during the execution of the program, a lot of memory is being allocated dynamically (during the run time), hence this memory has to be released whenever its not needed anymore in the program. Otherwise, it more memory is allocated than available, the whole program will crash.

4.2.1 ParseXML

This class performs the parsing of the XML file that contains all configuration parameters, and converting them from string data types to the corresponding data types depending on the parameter value. Methods of `ParseXML` are called and executed in the source file `SimulationConfig.cpp`, which is responsible for the configuration setup. If any changes or modifications to the actual parameters are made in the configuration file `ConfigParameters.xml` they will still be parsed and used within `SimulationConfig.cpp` without any further changes. This class works only with files in XML format.

Methods

```
CParseXML(const char* filePath, std::string tagName)
```

- Constructor, takes two parameters: the path of the XML file and tag names in the file and assigns them to corresponding data members. Tag names represent parameter names.

```
std::string readXml()
```

- The function that actually extract the values between XML tags and stores them in the form of strings.
- Return value: `string`

```
T StringToNumber(const std::string &Text)
```

- Template function, it converts the string value from the parsing function to any numeric value, depending on the parameter.
- Return value: depending how it is called/executed

4.2.2 Random Generators

This module contains different pseudo-random generators, using computational algorithms for binary and normal distribution, based on 32-bit [0,1) pseudo-random generator from Wilchmann-Hill [2] and using the Box Muller method [6]. This method is implemented according to the [2] and the following equations:

$$\begin{aligned}
 ix &= 11600 \times (ix \bmod 185127) - 10379 \times (ix/185127) \\
 iy &= 47003 \times (iy \bmod 45688) - 10479 \times (iy/45688) \\
 iz &= 23000 \times (iz \bmod 93368) - 19423 \times (iz/93368) \\
 it &= 33000 \times (it \bmod 65075) - 8123 \times (it/65075)
 \end{aligned} \tag{4.1}$$

It requires four different initial values, called seed values to generate the necessary parameters. In addition to that, an offset value has to be added to each parameter if the result of the operation is negative (less than zero), according to the following:

$$\begin{aligned}
 ix_{offset} &= 2147483579 \\
 iy_{offset} &= 2147483543 \\
 iz_{offset} &= 2147483423 \\
 it_{offset} &= 2147483123
 \end{aligned} \tag{4.2}$$

Hence in case that $ix < 0, iy < 0, iz < 0, it < 0$ the values for ix, iy, iz, it are given in the equation 4.3. This algorithm for calculation of values is implemented within a macro function `CALC_STEP(ix, iy, iz, it)` inside of the `ParametersWH32.h` file. This function is then invoked in each of the Random Generators modules (binary and normal distribution) within the `Simulate(float SimParam)` method.

$$\begin{aligned}
 ix &= ix + ix_{offset} \\
 iy &= iy + iy_{offset} \\
 iz &= iz + iz_{offset} \\
 it &= it + it_{offset}
 \end{aligned}
 \tag{4.3}$$

The offset values, and all the ix, iy, iz, it parameters together with other necessary values the random number generation are given in `ParametersWH32.h`.

4.2.2.1 CRandomGenerator

This is an abstract class that only inherits `CSimulationInterface` class, and contains its methods `Verify()` and `Simulate(float SimParam)` as pure virtual methods.

4.2.2.2 CBinaryVecWH32

This class inherits the `CRandomGenerator` class and implements Wichmann-Hill 32 bits binary pseudo-random generator. It generates uniformly distributed numbers between $[0,1]$, which are then compared against some threshold value and depending on the result of comparison, binary values 0 and 1 are assigned. This threshold value can be changed in the XML configuration file.

Methods

`CBinaryVecWH32(long seed[4], unsigned long length, float limit)`

- Constructor, takes an array of four elements `seed[4]` as one of the input arguments and initializes it so some default values. It represents the seed sequence for the generator. Constructor also assigns a new `CLinkModules, bool> *m_Output` pointer object for the output bit sequence.

`~CRCgenerator()`

- Destructor of the class, destroys the allocated object of the `CLinkModules` class.

`CLinkModules<bool>* GetOutputPtr()`

- Get access method. It is necessary so the output bit sequence can be accessed and used in the following module as an input sequence.
- Return value: boolean pointer to the buffer of type `CLinkModules`

Other methods, such as `Verify()` and `Simulate(float SimParam)` are methods inherited from the `SimulationInterface` class and as mentioned in previous chapter 3, they provide error checking functionality and ensure the correct simulation flow. In the `Simulate(float SimParam)` method the macro function `CALC_STEP(ix, iy, iz, it)` is executed with the relevant parameters and from that values the output values of the random generator are computed depending on the type of the probability distribution, i.e. if its uniform or normal distribution. In case of the binary distribution, computation is executed as in equation 4.5.

$$n = (ix/2147483579.0 + iy/2147483543.0 + iz/2147483423.0 + it/2147483123.0) \leq limit \quad (4.4)$$

If n is evaluated to `TRUE`, the value of 1 is set in the output sequence, if its evaluated to `FALSE`, the value 0 is set. That way the output sequence is generated in the pseudo-random computational way, and this sequence is used in the simulation i.e. CRC checksum generation, then modulation and coding methods from other modules are applied to it.

4.2.2.3 CNormalVecWH32

In this module Wichmann-Hill 32 bits pseudo-random generator is implemented, that generates pseudo-random numbers normally distributed between [0,1) [2]. Hereby the Box-Mueller method is used to convert two uniformly distributed random numbers to two normally distributed random numbers. For that purpose it executes the `CALC_STEP(ix, iy, iz, it)` function twice for different seed values and uses to result to calculate pseudo-random numbers according to the equation 4.5.

$$\begin{aligned} n1 &= (ix[0]/2147483579.0 + iy[0]/2147483543.0 + iz[0]/2147483423.0 + it[0]/2147483123.0) \\ n2 &= (ix[1]/2147483579.0 + iy[1]/2147483543.0 + iz[1]/2147483423.0 + it[1]/2147483123.0) \\ out &= \mu + \sigma * \sqrt{-2 * \log(n1) * \cos(2 * n2 * \pi)} \end{aligned} \quad (4.5)$$

These results correspond to (Gaussian) normal distribution, with μ as the mean value and σ as the standard deviation. This class inherits `CRandomGenerator` class.

Methods

```
CNormalVecWH32(long seed[2][4], unsigned long length, float mu, float sigma)
```

- Constructor, takes two-dimensional array `seed[2][4]` of seed values as initial values for the generator, then length of the buffer to store data and `mu` as the mean value of Gaussian normal distribution, and `sigma` meaning the standard deviation as input parameters. It initializes those values, assigning the memory dynamically for data to be stored as the object of linking class `CLinkModules` in the length of parameter `length`.

```
~CNormalVecWH32()
```

- Destructor of the class, makes sure the buffer object of class `CLinkModules` allocated dynamically is destroyed.

```
CLinkModules<bool>* GetOutputPtr()
```

- Get access function. Its necessary so the output bit sequence can be accessed and used in the following module as an input sequence.
- Return value: boolean pointer to the buffer of type `CLinkModules`

Inherited method `Verify()` is implemented to provide error checking functionality of this module, and `Simulate(float SimParam)` has its own implementation within this class. It executes the function `CALC_STEP(ix, iy, iz, it)` and calculates parameters according to the equation 4.5.

4.2.3 CRC checksum

Cyclic redundancy checksum (CRC) calculation is implemented in the `CRCgenerator` class with the help of the `CRCtable` class. It operates with the 32 bits (4 bytes) long keys, which corresponds exactly the length of unsigned long data type of standard processors. Hence, this data type is used as the key data type. For the simulation and testing purposes the CRC-32 is used that is typically used in Ethernet protocol frame, and this key equals to `0x04c11db7`. This key can easily be replaced with some other CRC-32 key.

4.2.3.1 CRCgenerator

This class implements the algorithm for generation of the Cyclic Redundancy Checksum (CRC) [39]. It does that by using the table of pre-calculated keys implemented within `CRCtable` class for all possible 8 bits values for the CRC key ($2^8 = 256$ values). A bunch of shifted keys are XORed with the given byte [33] and saved for the latter use. This table can be used to evaluate the CRC of any message bitwise. This table is initialized once when the `CRCgenerator` object is created and it will be reinitialized only if the key changes.

Methods

`CRCgenerator(unsigned long key, unsigned long length)`

- Constructor, initializes the table with pre-computed values and creates two objects of the `CLinkModules<bool>` class, one for the input and the other for the output sequence. The length of the *key* parameter is given in bytes, hence it has to be multiplied with 8 to obtain the value in bits. The *length* parameter represents the input length for the CRC module. The output length is larger than the input for the length of the 4 bytes (32 bits) of the calculated checksum, since the checksum is appended to the input sequence. The constructor contains the method `Initialize()` that is called on `CRCtable` object and initializes the table of shifted keys.

`~CRCgenerator()`

- Destructor of the class, makes sure that both the input and the output objects are deallocated and deleted.

`void PutByte(unsigned byte)`

- This method XORs the 8 bit values obtained from the table implemented in `CRCtable` with the bytes already in the register after the top byte is shifted out of the register. It then assigns the new byte value into the register. Top byte XORed with the message bytes gives the index of the table with shifted keys.
- Return value: `void`

`void Simulate(float SimParam)`

- In this method the algorithm for generating the checksum is implemented, using the previously explained method `PutByte`. After the checksum is calculated, it is appended to the input data sequence, which makes the output of the CRC module. `x`
- Return value: `void`

4.2.3.2 CRCtable

This class implements the static table that contains all possible shifted keys combinations. Table is represented in the form of an array with length of $2^8 = 256$ elements. These elements are the result of the XOR operation of the register content and the CRC key. Register content is updated in every loop iteration and it takes the values of all possible bit combinations. these are then XORed with a bunch of shifted keys corresponding to a given byte and stored for latter.

Methods

`CRCtable(unsigned long key)`

- Constructor, assigns key value to its corresponding data member.

```
void Initialize()
```

- Builds a static table of pre-calculated values, where the XOR operation is applied to given bytes in the register and shifted keys, they are XORed together to create the table. If top bit in the register equals to one, the current value of the register is XORed with the key value; if not, its shifted one spot to the left until the loop reaches the value one for the top bit.
- Return value: void

```
unsigned long GetTableElement(unsigned index)
```

- Access a table element with the given index, returns an array element, since table is implemented as an array.
- Return value: unsigned long

4.2.4 Modulators

Modulation techniques used in the simulation are described in chapter 2 and 3. A brief overview of the methods that implement those modulation schemes is given in the following sections. Most of them are implemented in the current software version, and some of them are yet to be implemented.

4.2.4.1 OOK

OOK class implements the On-Off Keying modulation for optical channel as described in chapter 2. It inherits the `CSimulationInterface` class, for the purpose of implementing its virtual methods, `Verify()` and `Simulate(float SimParam)` and it includes `CLinkModules` class, since it allocates its objects for input and output sequence. OOK belongs to a class of baseband modulation, hence it does not require to be translated to a carrier with much higher frequency.

Methods

```
cOOKmod(unsigned long length, unsigned long bitRate, unsigned long  
sampleRate)
```

- Constructor, creates objects of `CLinkModules` class for the input and output sequences and initializes the bit rate and the sample rate to corresponding argument values.

```
~cOOKmod()
```

- Destructor of the class, deletes the object of `CLinkModules`.


```
bool Verify()
```

- Verifies the proper creation of the modules; method inherited from `SimulationInterface` class
- Return value: `bool`

```
Simulate(float SimParam=0.0)
```

- It implements the algorithm for the OOK simulation. It takes two different pulse values into consideration. Output vector of this operation is simple sequence of bits, since its a baseband modulation. It is used as an input for the AWGN module.
- Return value: `void`

4.2.4.2 PPM

PPM class implements the Pulse Position modulation for optical channel as described in chapter 2. Similar to OOK, it inherits the `CSimulationInterface` class, for the purpose of implementing its virtual methods, `Verify()` and `Simulate(float SimParam)` and it includes `CLinkModules` class, since it allocates its objects for input and output sequences.

Methods

```
cPPMmod(unsigned long length, unsigned long bitRate, unsigned long  
sampleRate)
```

- Constructor, creates objects of `CLinkModules` class for the input and output sequences and initializes the bit rate and the sample rate to corresponding argument values.

```
~cPPMmod()
```

- Destructor of the class, deletes the objects of `CLinkModules`.

```
bool Verify()
```

- Verifies the proper creation of the modules
- Return value: `bool`

```
Simulate(float SimParam=0.0)
```

- Implements the algorithm for creating the PPM signal and executes simulation with the current value of the simulation parameter (SNR).
- Return value: `void`

4.2.4.3 BPSK

BPSK modulation is usually used in its simplest form in Matlab simulations without taking into consideration the carrier frequency. It converts the input bits 0 and 1 into values -1 and 1 respectively (or some other values) and adds white noise (AWGN) on the channel and demodulates by looking at the received values and comparing them to a certain threshold value. The other way of implementing it is with carrier frequency, bit rate and the sample rate.

Methods

```
cBPSKmod(unsigned long length, unsigned long bitRate, unsigned long
carrierFactor, unsigned long sampleRate)
```

- Constructor, initializes data members and constructs an object of type `CLinkModules<bool>` for the input values and `CLinkModules<float>` object for output values. Type `float` is chosen for the output value since after multiplication of the bits with the sinusoidal carrier signal their value will change from boolean to floating point numbers.

```
bool Verify()
```

- Verifies the proper creation of the elements
- Return value: `bool`

```
Simulate(float SimParam=0.0)
```

- Implements the algorithm for creating BPSK signal, and executes simulation with the current value of the simulation parameter (SNR).
- Return value: `void`

4.2.4.4 16-PAM

In 16-PAM modulation $M = 16$ and the symbol alphabet consists of 16 symbols modulated with 4 bits. M can be replaced with any other number in the form 2^n , where $n = 2, 3, 4, 5, \dots$. For the sake of simplicity here 16 – PAM is described.

Methods

```
cOOKmod(unsigned long length)
```

- Constructor, initializes data members and constructs an object of type `CLinkModules<bool>` for the input values and `CLinkModules<foat>` object for output values.

```
Verify()
```

- Verifies the proper creation of the elements

- Return value: `bool`

`Simulate(float SimParam=0.0)`

- Implements the algorithm for creating the 16-PAM signal and executes simulation with the current value of the simulation parameter (SNR).
- Return value: `void`

4.2.4.5 4-QAM

4-Quadrature Amplitude modulation is similar to QPSK and 4 symbols are used to transmit the information.

Methods

`cOOKmod(unsigned long length)`

- Constructor initializes data members and constructs an object of type `CLinkModules<bool>` for the input values and `CLinkModules<float>` object for output values.

`Verify()`

- Verifies the proper creation of the elements
- Return value: `bool`

`Simulate(float SimParam=0.0)`

- Implements the algorithm for creating the PPM signal and executes simulation with the current value of the simulation parameter (SNR).
- Return value: `void`

4.2.4.6 AWGN

In a typical wireless communication system, the channel between the transmitter and the receiver section is simulated with the Additive Gaussian White Noise (AWGN) channel model [31]. This well known mathematical model is suitable for the simulation of the optical wireless channel and according to it the main source of the noise in the transmission channel is the noise produced in the receiver, which is caused by thermal movements of the electrons in the receiver equipment. This model This effect is perfectly described using the Gaussian probability distribution for the amplitude values that represent a linear addition of wideband or white noise with the constant spectral density [38].

In the simple case of BPSK modulation, variance of the additive white noise is calculated according to the equation 4.6.

$$\sigma = \sqrt{\frac{1}{2 \cdot 10^{\frac{E_s/N_0}{10}}}} \quad (4.6)$$

where $\frac{E_s}{N_0}$ represents Energy per Symbol to Noise Power Spectral Density ratio, defined as [22]:

$$\frac{E_s}{N_0} = \frac{E_b}{N_0} + 10 \log CR [dB] \quad (4.7)$$

with $\frac{E_b}{N_0}$ as the Energy per Bit to Noise Power Spectral density ratio, and CR is the Code Rate of the channel coding method used in the simulation.

Methods

`C_AWGN(unsigned long frameLength, float codeRate)`

- Constructor, allocates `CLinkModules` objects for input and output data and `CNormalVecWH32` object instance for the noise generation, both in the length of the parameter `frameLength`. It also sets up the value for σ (standard deviation) that will be used for the calculation of gaussian white noise. `s`

`virtual ~C_AWGN()`

- Destructor, destroys all `CLinkModules` and `CNormalVecWH32` objects.

`virtual bool Verify()`

- Inherited method from the `SimulationInterface` abstract base class
- **Return value:** `bool`

`virtual void Simulate(float SimParam=0.0)`

- Inherited method from the `SimulationInterface` abstract base class. It implements the additive white noise according to equations 4.6,4.7 using the current value of the Signal to Noise ratio. It adds the value of the noise to the input data sequence and assigns it to output data sequence.
- **Return value:** `void`

`CLinkModules<float>* GetOutputPtr()`

- access get function, returns a pointer of type `CLinkModules` with elements of type `float`
- **Return value:** `CLinkModules<float>*`

`void SetInputPtr(CLinkModules<float>* dataInput)`

- access set function, sets the value of the `CLinkModules` pointer for the input sequence

- Return value: void

4.2.5 Demodulators and Detection

This module is in the reception segment of the communication channel 2.1 can only be applied together with the Modulators module in the transmission segment. Depending on which modulation is applied on the channel, and included in `CreateModules`, the same kind of demodulation has to be included and executed.

5 Simulation results and discussion

The application of different signal processing techniques on optical wireless channels offer different channel performance and error rates. Bit Error Rate (BER) simulations are usually executed with varying combinations of modulations and coding techniques in order to investigate the best signal performance. In this chapter some of these results are presented and compared. Each of the combinations has its own advantages and disadvantages; it all comes up to the trade-off between different modulation and coding schemes on the channel. Matlab is used to create plots for results from the simulation framework due to it being the de facto tool for doing so and the visual ease by which the results can be consumed in this form.

First the results of simulations employing only modulation techniques are presented and compared, followed by the results of simulations employing channel coding schemes together with modulation schemes. A brief discussion is provided as to why some schemes are preferred in practical use over others.

5.1 Modulation schemes uncoded

The most commonly used modulations techniques are the binary modulation schemes that consist of only two different intensity levels to represent the binary information. These schemes have become very popular (e.g., OOK and PPM). On the other hand, multilevel schemes that utilize more than two levels of a certain signal parameter can achieve higher data rates than binary schemes, but they are not widely used in FSO due to their bad performance at fading events. In the simulations performed with the simulator it is assumed that the modulator has a fixed bandwidth in GHz , which means that it can produce rectangular pulses of a fix duration.

5.1.1 Comparison between RF and optical

Multilevel modulation schemes, such as M-PAM are known to improve spectral efficiency, especially with increasing M by utilizing the bandwidth more efficiently (than for example simple OOK modulation). On the other hand they require higher signal to noise ratio at the receiver, which means an increased optical power.

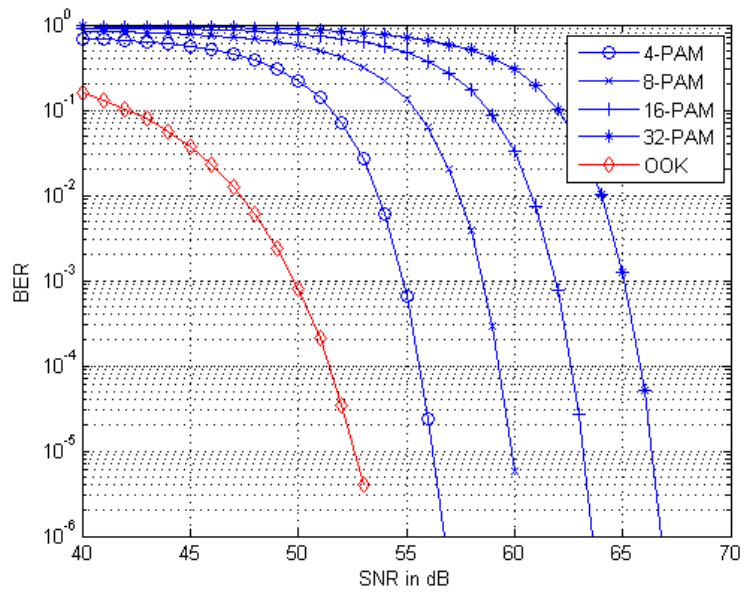


Figure 5.1: BER of OOK vs M-PAM modulation

In Figure 5.1, simple OOK is compared with PAM for $M = 2, 4, 8, 16$. It can clearly be seen that in order to achieve the same bit error rate employing the higher order PAM, more optical power, approximately 1.8 dBs, is needed at the receiver.

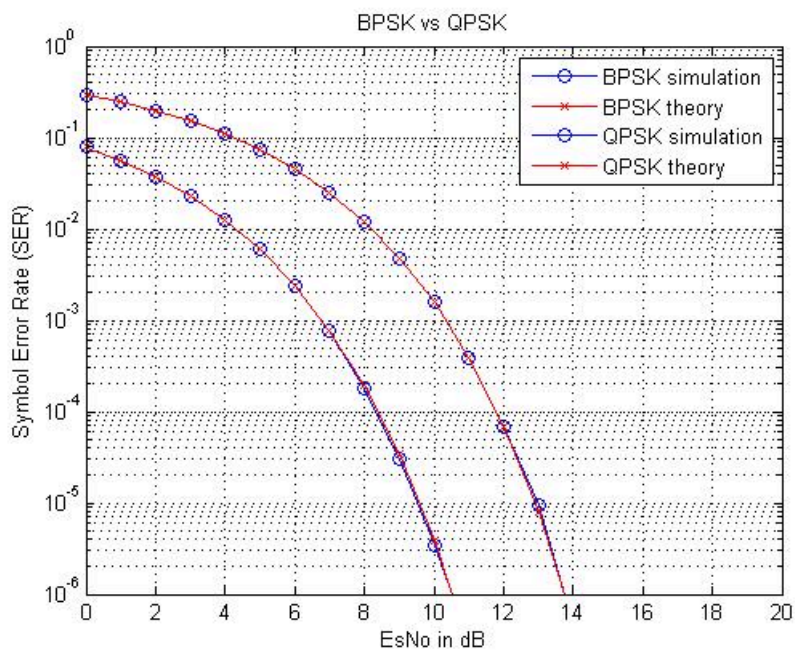


Figure 5.2: BER of OOK vs M-PAM modulation

Phase shift keying modulations, such as BPSK and QPSK in figure 5.2 require a receiver with higher sensitivity. Due to their BER curve more signal power is needed at the receiver, than when using optical intensity modulations (OOK, PPM). They both require approximately $2dB$ more optical power.

PPM and PAM modulation are compared using the same symbol rate and bandwidth, which makes the comparison more accurate. They are multilevel modulation techniques, which simply means that more bits are encoded into a symbols, and bandwidth can be utilized more efficiently. From figure 5.3 it can be seen that the SNR level necessary for the error-free reception reduces when the signal constellation size M increases.

5.2 Modulation schemes coded

For the additional channel performance gain by employing error correcting methods, popular codes such as BCH, RS and LDPC can be used. They are well described in chapter 2 and their implementations can be found in previous work and Matlab simulations. Basic declarations for these coding schemes have been provided and can be implemented easily.

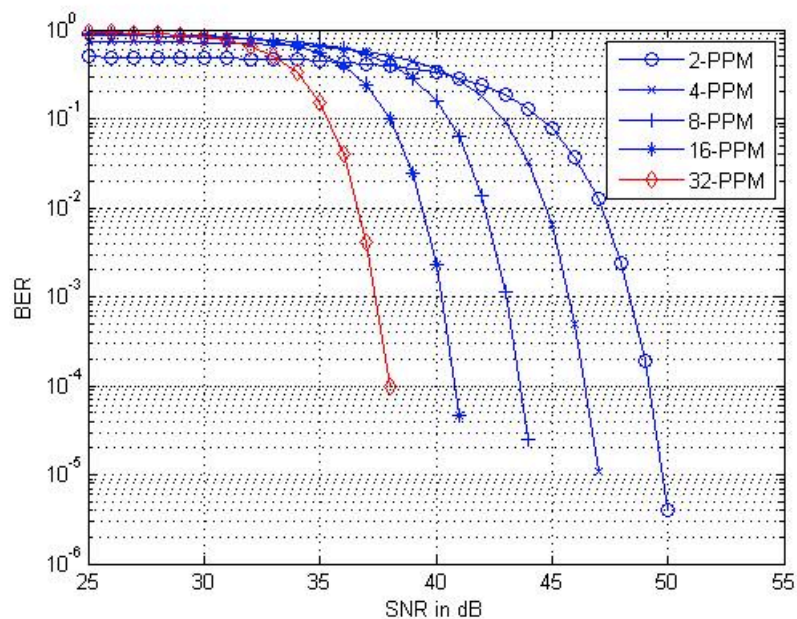


Figure 5.3: BER of multilevel PPM modulation

6 Conclusion and future work

Optical wireless (OW) communications is a growing popular field, hence for outdoors and indoors applications a lot of research work is being done. This simulator has practical application for testing of channel schemes that are being investigated. OW links are mostly attractive because of their high data rate, unlicensed bandwidths, and low operational costs, however they cannot replace RF links completely, especially since they can be severely affected by inclement weather. Therefore OW links are best when used together in combination with RF links in some kind of a hybrid structure [5]. In this case, each system can undertake a role that best corresponds to their capabilities, and they will both operate in conjunction with each other. Hybrid systems are currently under a lot of research and this kind of a simulator where RF schemes are applied to OW field aims to support that research.

The simulator framework created in this work utilizes advanced concepts of the C++ programming language, such as object-oriented modular approach for the simulation of communication channel. These channels are built as a composition of a sequence of building blocks that can be connected to each other more or less independently. Creating the framework in this manner (as a composition of building blocks), was one of the main challenges of this work, but allows for easy development and inclusion of new modules/blocks. As a result, this framework can be a useful tool for deploying different signal processing techniques on the channel, and for examination and analysis of the results. Having been built in an extensible, cross-platform (platform-independent), and low-memory manner, the framework can run on a variety of hardware – this included writing a XML parsing module so that additional platform specific libraries or install instruction aren't needed. Building this in C++ has the additional benefit, that code compiles to machine code and has a low execution time.

Beside building a framework that can support all the different independent functionalities, the challenge was to link them together correctly. Correctly in this case means that it not only works as intended, but also has no memory leaks and other possible code problems (e.g., comparison of *int* and *long*). This was sometimes hard to accomplish, due to the encapsulation feature and method of sequencing blocks to create a channel. To aid in doing so, `valgrind` [?] and `cppcheck` [?] were used in addition to compiler options “-Wall” (warnings all) and “-Wextra” (warnings extra). This results in code that has no language errors and has zero memory leaks. Having no memory leaks was important, especially since simulations are typically executed on large amounts of data (due to long input bit sequences), and improperly managed memory would likely lead to out of

memory errors. The whole simulator was tested for the bit sequence lengths $10e7$ and higher, and it concluded that it performs correctly even for the higher data sizes.

In addition to the aforementioned, the requirement for the software to run on all operating systems was also challenging, due to different kind of compilers and libraries on different operating systems. For that purpose operating system specific code is avoided and no libraries are used. More specifically, instead of using a library for XML parsing, a module to do this was created from scratch.

Channel simulations in signal processing are usually executed in Matlab and there are various libraries and communication tools available for signal processing of communication channels. In Matlab, the code itself is easier to write (especially dealing with bit sequences in form of vectors or matrices) and signal processing methods are readily available. On the other hand, simulations in Matlab are both memory and processor intensive, typically taking an order of magnitude or two longer to run. Within this work, only the plotting features of Matlab are utilized, for visualizing the results.

Further work on the simulator can be done by adding the modules that implement different channel coding techniques together with modulation techniques, and also applying the newly developed subcarrier modulation and multiplexing techniques. Also in the Framework, the technique for switching modules (i.e. taking decisions on which modules are going to be used in current simulation) can be enhanced and adapted for different usages. An additional channel model such as fading channel can also be implemented in addition to the existing AWGN channel model used.

As already mentioned, future work with the simulator could involve deeper investigation into the hybrid RF and OW schemes, specifically determining and implementing a transition point when one scheme should be used over the other.

Bibliography

- [1] C++ reference manual. <http://www.cplusplus.com/reference/>.
- [2] I.D. Hill B.A. Wichmann. Generating good pseudo-random numbers. *Computational Statistics & Data Analysis*, 51(3), December 2006.
- [3] Elwyn R. Berlekamp. *Algebraic Coding Theory, Revised 1984 edition*. Aegean Park Pr, 1984.
- [4] Richard E. Blahut. *Algebraic Codes for Data Transmission*. Cambridge University Press, 2003.
- [5] D.K. Borah, A.C. Boucouvalas, C.C. Davis, S. Hranilovic, and K. Yiannopoulos. A review of communication-oriented optical wireless systems. *EURASIP Journal on Wireless Communications and Networking*, 2012(1):91, 2012.
- [6] G.E.P. Box and M.E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.
- [7] International Electrotechnical Commission. Safety of laser products (iec60825-1). 2007. <http://webstore.iec.ch/webstore/webstore.nsf/ArtNum/037864>.
- [8] C.S.Choi and H.Lee. High throughput four-parallel rs decoder architecture for 60ghz mmwave wpan systems. *Consumer Electronics (ICCE), 2010 Digest of Technical Papers International Conference*, January 2010.
- [9] Martin Czaputa. Throughput maximizing modulation and coding techniques for free-space optical links. Master's thesis, Graz University of Technology, December 2010.
- [10] W.O. Popoola S. Rajbhandari M. Amiri F .Ghassemlooy and S. Hashemi. A synopsis of modulation techniques for wireless infrared communications. 2007. ICTON MW007.
- [11] Patrice Causse Fang Xu, Ali Khalighi and Salah Bourenane. Channel coding and time-diversity for optical wireless links. *Optic Express*, 17, 2009.
- [12] M. Gebhart, E. Leitgeb, S.S. Muhammad, B. Flecker, C. Chlestil, M. Al Naboulsi, F. de Fornel, and H. Sizun. Measurement of light attenuation in dense fog conditions for fso applications. In *Optics & Photonics 2005*, pages 58910K–58910K. International Society for Optics and Photonics, 2005.

- [13] Steve Hranilovic. *Wireless optical communication systems*. Springer Science, 2005.
- [14] American National Standards Institute. Standards: The foundation of a successful laser safety program. <http://www.lia.org/publications/ansi>.
- [15] David G. Messerschmit John R. Barry, Edward A. Lee. *Digital Communication*. Springer, third edition, 2004.
- [16] C. Langton. Intuitive guide to principles of communications. 2005.
- [17] L.C. Le Bidan Raphaël, J. Christophe, A. Patrick, and P. Ramesh. Reed-solomon turbo product codes for optical communications: From code optimization to decoder design. *EURASIP Journal on Wireless Communications and Networking*, 2008.
- [18] E. Leitgeb, M. Gebhart, U. Birnbacher, S. Sheikh Muhammad, and C. Chlestil. Applications of free space optics for broadband access. *Optical Networks and Technologies*, pages 579–586, 2005.
- [19] E. Leitgeb, S.S. Muhammad, C. Chlestil, M. Gebhart, and U. Birnbacher. Reliability of fso links in next generation optical networks. In *Transparent Optical Networks, 2005, Proceedings of 2005 7th International Conference*, volume 1, pages 394–401. IEEE, 2005.
- [20] E. Leitgeb, S.S. Muhammad, B. Flecker, C. Chlestil, M. Gebhart, and T. Javornik. The influence of dense fog on optical wireless systems, analysed by measurements in graz for improving the link-reliability. In *Transparent Optical Networks, 2006 International Conference on*, volume 3, pages 154–159. IEEE, 2006.
- [21] S. Lin and JR. D. J.Costello. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 2004.
- [22] T. K. Moon. *Error Correction Coding: mathematical methods and algorithms*. Wiley, 2005.
- [23] Sajid SHEIKH MUHAMMAD. *Investigations in Modulation and Coding for Terrestrial Free Space Optical Links*. PhD thesis, Graz University of Technology, March 2007.
- [24] S.S. Muhammad, T. Javornik, I. Jelovčan, Z. Ghassemlooy, and E. Leitgeb. Comparison of hard-decision and soft-decision channel coded m-ary ppm performance over free space optical links. *Transactions on Emerging Telecommunications Technologies*, 20(8):746–757, 2009.
- [25] S.S. Muhammad, E. Leitgeb, and O. Koudelka. Multilevel modulation and channel codes for terrestrial fso links. In *Wireless Communication Systems, 2005. 2nd International Symposium on*, pages 795–799. IEEE, 2005.
- [26] Damon Danieli Paul M. Embree. *C++ Algorithms for Digital Signal Processing*. Prentice Hall, Inc., 1999.
- [27] J. G. Proakis. *Digital Communications*. McGraw-Hill, 2000.

- [28] Ziran Sun Roberto Ramirez-Iniguez, Sevia M. Idrus. *Optical Wireless Communications: IR for Wireless Connectivity*. Taylor & Francis Group, LLC, 2008.
- [29] Harald Niederreiter Rudolf Lidl. *Introduction to Finite Fields and their Applications*. Cambridge University Press, 1994.
- [30] Daniel J. Costello Shu Lin. *Error Control Coding*. Prentice Hall, 2004.
- [31] B. Sklar. *Digital Communications: Fundamentals and Applications*. Prentice Hall, 2001.
- [32] C.S.Park S.M.Kim and S.Y.Hwang. A novel partially parallel architecture for high-throughput ldpc decoder for dvb-s2. *Consumer Electronics, IEEE Transactions*, 66:820–825, May 2010.
- [33] Reliable Software. Optimized crc calculation. 2001. [Online].
- [34] E. Leitgeb S.S. Muhammad, B. Flecker and M. Gebhart. Characterization of fog attenuation in terrestrial free space optical links. *Optical Engineering*, 46(6), 2007.
- [35] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, 1997.
- [36] W. Tranter, K. Shanmugan, T. Rappaport, and K. Kosbar. *Principles of communication systems simulation with wireless applications*. Prentice Hall Press, 2003.
- [37] Stephen B. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice-Hall, 1994.
- [38] Wikipedia. Additive white gaussian noise, 2012. http://en.wikipedia.org/wiki/Additive_white_Gaussian_noise.
- [39] Wikipedia. Cyclic redundancy check. 2012. http://en.wikipedia.org/wiki/Cyclic_redundancy_check.
- [40] Wikipedia. Phase shift keying. 2012. http://en.wikipedia.org/wiki/Phase-shift_keying.
- [41] Sujan Rajbhandari Zabih Ghassemlooy, Wasiu Popoola. *Optical wireless Communications: System and Channel Modelling with MATLAB*. CRC press, August 2012.
- [42] X. Zhu and J.M. Kahn. Free-space optical communication through atmospheric turbulence channels. *Communications, IEEE Transactions on*, 50(8):1293–1300, 2002.