CONVOLUTIONAL CODED GENERALIZED DIRECT SEQUENCE SPREAD SPECTRUM

MADAN VENN

Bachelor of Electrical Engineering
Jawaharlal Nehru Technological University, India
May, 2004

Submitted in partial fulfillment of requirements for the degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

At the

CLEVELAND STATE UNIVERSITY

May, 2008

This thesis has been approved

For the Department of **ELECTRICAL AND COMPUTER ENGINEERING**

And the College of Graduate Studies by

_____

Dr. Fuqin Xiong, Thesis Committee Chairperson

_____

Department/Date

_____

Dr. Murad Hizlan, Thesis Advisor

_____

Department/Date

_____

Dr. Ana Stankovic, Thesis Committee Member

_____

Department/Date

_____

Dr. Chansu Yu, Thesis Committee Member

_____

Department/Date

ACKNOWLEDGEMENTS

Many thanks go to my advisor, Dr. Murad Hizlan, for giving me an opportunity to work with him, and for his helpful guidance and technical advice, without which much of this work would not have been possible. I would also like to thank all the professors with whom I have taken classes during my masters program.

I am grateful to the department of Electrical and Computer Engineering and Professor Eugenio Villaseca for providing financial support during my masters program.

I am thankful to my family and friends for standing by me while I went crazy and stayed up nights at the end trying to finish this on time. I also thank my lab mates and colleagues, especially Vijay Nomula and Indrasena Parayatham for their support.

CONVOLUTIONAL CODED GENERALIZED DIRECT SEQUENCE SPREAD SPECTRUM

MADAN VENN

ABSTRACT

In this thesis we investigate the worst-case performance of coded ordinary and coded generalized direct sequence spread spectrum (DSSS) systems in a communication channel corrupted by an unknown and arbitrary interfering signal of bounded power. We consider convolutional codes with Viterbi decoding in order to compare the performance of coded ordinary and coded generalized DSSS systems. For the generalized DSSS system, we use a pulse stream of +1,-1 and 0 as the spreading sequence, which is different from ordinary DSSS system which uses the typical sequence with pulse values of +1 and -1.

A C program for performing Monte-Carlo simulations is written in order to evaluate and compare the performance of coded ordinary and coded generalized DSSS systems. Plots of the worst-case error probability versus signal-to-interference ratio are presented for different code rates and constraint lengths of the convolutional code. Simulation results of the worst-case performance of ordinary and generalized DSSS show that generalized DSSS consistently performs appreciably better than ordinary DSSS. Simulation is performed for various code rates, various constraint lengths of the convolutional code and various lengths of the convolutional interleaver. Over all these

simulations, it is observed that the difference between ordinary and generalized DSSS gets more pronounced as the channel gets worse.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER I

INTRODUCTION

1.1 Background

It all started when Guglielmo Marconi invented wireless telegraph. From then on wireless communications has gone through lots of inventions. Particularly during the past twenty years, the mobile radio communications industry has grown by orders of magnitude, fueled by digital and RF circuit fabrication improvements, new large-scale circuit integration, and other miniaturization technologies which make portable radio equipment smaller, cheaper, and more reliable. Digital switching techniques have enabled the large scale deployment of affordable, easy-to-use radio communication networks. The innovations will continue at an even greater pace in the coming years.

In our daily life we come across a wide array of communication devices, the most common being the cellular phone, GPS, radio, television and wireless internet. Although

there is rapid growth in wired communications, the biggest challenges lie in developing wireless systems. Research is being done to improve the robustness of the channel and provide error free transmission in a wireless communication system.

With the inventions in wireless personal communications field over the last several years, the method of communication known as spread spectrum has gained a great deal of importance. Spread spectrum involves the spreading of the desired signal over a bandwidth much larger than the minimum bandwidth necessary to send the information signal. It was originally developed by the military as a method of communication that is less sensitive to intentional interference or jamming by third parties, but has become very popular in the realm of personal communications recently. Spread spectrum methods can be combined with multiple access methods to create code division multiple access (CDMA) systems for multi-user communications with very good interference suppression. Two very common types of spread spectrum schemes that are in use today are direct sequence spread spectrum (DSSS) and frequency hopping spread spectrum (FHSS). Usually FHSS devices use less power and are cheaper, but DSSS systems have better performance and are more reliable. In this thesis we will also consider a newer, more robust class of proposed spread spectrum systems called generalized spread spectrum. Detailed description of a spread spectrum communication system is presented in Chapter 2 of this thesis.

Channel coding is used to reduce the errors caused during transmission. Block codes and convolutional codes are the two widely used methods for channel coding. Detailed description of channel coding and its applications are presented in Chapter 2 of

this thesis. The work in this thesis relates to applying convolutional codes to ordinary and generalized DSSS in order to compare their worst-case performance.

1.2 Motivation

The work in this thesis relates to the simulation of worst-case performance of ordinary and generalized direct sequence spread spectrum using convolutional codes with a Viterbi decoder in order to compare coded ordinary and coded generalized direct sequence spread spectrum.

Most often spread spectrum is used in situations where we would like to suppress some type of interference in the channel other than additive white Gaussian noise. Information regarding the nature of such interference is not available or it changes with time in a random manner, which causes the correct estimation of channel properties unrealistic. The usual approach to this particular problem is to assume a precise statistical description of the channel and evaluate the performance of communication system based on such assumptions. A much better approach when considering a robust communication system is to make no statistical assumptions about the channel and perform a worst-case analysis based on no more than an average power limit on the interference. In this thesis we follow this particular approach.

Significant amount of research has been performed over a long time in the field of direct sequence spread spectrum by applying the above interference, called the arbitrarily varying channel, by Dr. Hizlan and Dr. Hughes. In [1] they have shown that the

asymptotically optimal benchmark communication system in such situations consists of a random modulator that uniformly distributes any given message vector on the surface of an *N*-dimensional sphere, and a correlation receiver. They have also shown that spread spectrum is only a special suboptimal instance of the family of such modems. Since the asymptotically optimal system provides only a theoretical and impractical benchmark, later Hizlan [2] has proposed a practical *generalized* direct sequence spread spectrum system which improves upon *ordinary* direct sequence spread spectrum in the direction of the benchmark result. In [3], Hizlan described the performance analysis of coded ordinary DSSS in the arbitrarily varying channel. In [4] Vellala used block codes to show that coded generalized spread spectrum performed consistently better than coded ordinary direct sequence spread spectrum in the worst-case. Our aim in this thesis is to use convolutional codes in order compare the worst-case coded performance of generalized and ordinary direct sequence spread spectrum systems.

Consequently, in this thesis we consider coded ordinary and generalized direct sequence spread spectrum systems with a convolutional encoder and a Viterbi decoder. The spreading sequence used in the generalized system is a pulse stream with pulse values of +1, -1 and 0, which are different from the usual sequence with pulse values of +1 or -1 used in ordinary system.

1.3 Related Work

In [1] Hizlan and Hughes derive a random linear modem and detector that asymptotically minimize the transmitted power for a given encoder as the block length of the encoder becomes large. The optimal modem turns out to be independent of the encoder and the optimal detector is the standard correlation receiver. The asymptotically optimal modem is a random modem that distributes a codeword uniformly on the surface of an N-dimensional sphere. An upper bound to the performance of any encoder used with the optimal modem and detector is derived. It is shown that the coding gain achieved on the arbitrarily varying channel is larger than that of the comparable Gaussian channel. The results given in [1] provide a benchmark for robust communications against which a variety of spread spectrum modems and robust detectors could be compared. In [1] the authors show that DSSS, which is referred to as ordinary DSSS in this thesis, is only a special case of random modulation, and that random modulation becomes asymptotically optimal as N gets larger, minimizing the signal-to-interference ratio required to guarantee a given worst-case performance level, when the message symbol is uniformly distributed on the surface of N-dimensional sphere. This is only a theoretical benchmark of what could possibly be achieved and is difficult to implement in practice.

The communication system in [2] is inspired by [1] and it talks about a generalization of uncoded DSSS. Generalization improves upon ordinary DSSS by allowing the transmitted vector to more closely approximate a uniform distribution on the surface of an N-dimensional sphere while still being practical to implement. As detailed in section 2.6, along with the vertices for ordinary DSSS, midpoints of the edges and

faces of the cube are considered as possible transmitted vectors, and these points are projected radially onto the surface of a 3-*D* sphere. This idea, when extended to *N*-dimensional signal space is called generalized DSSS. Though uniform distribution on the surface of the sphere may not necessarily be optimal for finite *N*, and is impractical to implement, it does provide a benchmark against which worst-case performance of generalized DSSS can be compared. Bounds to the worst-case error probability of this generalized DSSS system are obtained and they show an improvement in worst-case performance over ordinary DSSS.

In [3], Hizlan described the performance analysis of coded ordinary DSSS in the arbitrarily varying channel. He derived a simple upper bound to the worst-case error probability incurred by the communication system including a binary block code, pseudorandom interleaving and a correlation receiver, operating on a channel corrupted by thermal noise and by an unknown interfering signal of bounded power. He also found that the derived upper bound for this channel is exponentially tight as the block length of the code became large. In comparing the performance of coded ordinary DSSS with coded optimal random modem and detector, Hizlan found that for low-rate codes, there was a significant performance difference between ordinary DSSS and the optimal system, while the difference subsided for high-rate codes.

In [4] Vellala described the performance of coded ordinary and coded generalized direct sequence spread spectrum systems with various cyclic, BCH and burst error correcting codes. His simulation results of the worst-case performance of ordinary and generalized DSSS for several block codes showed that generalized DSSS consistently performed better than ordinary DSSS. In [5], Ranga Kalakuntla considered further

generalization of uncoded DSSS to 5 levels. In [6], Hariharan Ramaswamy worked on theoretical properties of 3- and 5-level sequences, and considered software and hardware methods for their generation.

[1], [2], [3] and [4] talk about ordinary DSSS, generalization, performance analysis of ordinary DSSS and performance analysis of coded generalized DSSS using block codes only. The performance analysis of coded generalized DSSS using convolutional coding is not considered, so we step ahead and simulate the worst-case performance of coded generalized DSSS using convolutional codes with a Viterbi decoder for different code rates and constraint lengths, and compare the performance of coded generalized DSSS with ordinary DSSS in this thesis.

1.4 Thesis Structure

This thesis considers the worst-case performance of a coded generalized direct sequence spread spectrum system in comparison to that of a coded ordinary direct sequence spread spectrum system, both using convolutional codes with Viterbi decoding and operating in the arbitrarily varying channel. Chapter 2 contains a description of the spread spectrum communication system, both ordinary and generalized. Channel model and a measure of the worst-case system performance are described in Chapter 3. An introduction to convolutional codes and their decoding techniques are discussed in Chapter 4. The simulation of the communication system is talked about in Chapter 5.

Chapter 6 includes numerical results and observations. Chapter 7 talks about conclusions and future work. Also, simulation codes used in this thesis are found in Appendices.

CHAPTER II

SPREAD SPECTRUM COMMUNICATION SYSTEM

2.1 A Digital Communication System

Communication systems are mainly classified into analog and digital. The most important feature of a digital communication system is that it deals with a finite set of discrete messages, in contrast to an analog communication system in which the messages are continuous. In a digital communication system, the message to be transmitted, whether analog or discrete, is processed in a digital form, i.e. as a sequence of binary digits obtained after source encoding. A basic communication system consists of a transmitter, receiver and a channel through which the information is transmitted. The main objective at the receiver of the digital system is not to reproduce a waveform with precision but instead determine from a noise-perturbed signal which of the finite set of waveforms had been sent by the transmitter. The channel characteristics generally affect

the design of the basic elements of the system, a description of which is given in this section.

A digital communication system may have components such as channel coding, interleaving, modulation and spreading techniques, which will be discussed in detail in the coming sections.

2.2 Channel Coding

When information is transmitted over a channel in the presence of noise, errors will occur. The task of channel coding is to represent the source information in a manner that minimizes the error probability in decoding. Channel coding refers to the class of signal transformations designed to improve communications performance by enabling the transmitted signals to better combat the effects of various channel impairments, such as noise, interference, and fading as described in [7]. The main purpose of channel coding is to reduce the probability of bit error at the cost of expanding the bandwidth. In a coded digital system, each information sequence is first passed to a channel encoder which introduces some carefully designed structure to a data word in order to protect it from transmission errors. This process is also termed as forward error correction, which improves the capacity of a channel by adding some carefully designed redundant information to the data being transmitted through the channel.

Convolutional coding and block coding are the two major forms of channel coding as described in [8]. We choose convolutional codes in this thesis. A block code

is described by two integers, n and k, and a generator matrix or polynomial. The integer k is the number of data bits that form an input to a block encoder. The integer n is the total number of bits in the associated codeword out of the encoder. A characteristic of linear block codes is that each codeword n-tuple is uniquely determined by the input message k-tuple. The ratio k/n is called the rate of the code and gives a measure of the added redundancy. A convolutional code is characterized by three integers, n, k, and K, where the ratio k/n has the same code rate significance as that for block codes. However n does not define a block or codeword length as in the case of block codes. The integer K is termed as constraint length and it represents the number of k-tuple stages in the encoding shift register. An important feature of convolutional codes is that the encoder has memory, i.e. the n-tuple emitted by the convolutional encoding procedure is not only a function of an input k-tuple but is also a function of the previous K-1 input k-tuples. Convolutional codes operate on serial data, one or a few bits at a time whereas block codes operate on relatively large (typically, up to a couple of hundred bytes) message blocks. There are a variety of useful convolutional codes, and a variety of algorithms for decoding the received coded information sequences to recover the original data. In practice, n and k are small integers and K is varied usually between three and eight to control the redundancy. A detailed description of convolutional codes is presented in chapter four of this thesis. In this thesis we consider convolutional codes with different values of n and K while keeping k a constant equal to one.

2.3 Interleaving

A memoryless channel is characterized with random errors but a channel with memory such as fading and multi-path exhibits mutually dependent signal transmission impairments. Also, some channels suffer from switching noise and other burst noise. All of these time-correlated impairments result in statistical dependence among successive symbol transmissions. Hence, the disturbances tend to cause errors that occur in bursts, instead of isolated events. Most block or convolutional codes are designed to combat random independent errors. By applying these codes to channels with memory causes degradation in error performance. A technique which requires knowledge of the duration of the channel memory and not the exact channel statistical characterization is the use of time diversity or interleaving. Interleaving the coded message before transmission and deinterleaving after reception causes bursts of channel errors to be spread out in time and thus to be handled by the decoder as if they were random errors. So in many applications data is interleaved just before transmission. Most error control codes work much better when error in the received sequence is spread far apart.

There are a number of interleavers to choose from for the system described in this thesis. We use convolutional and pseudorandom interleavers in this thesis and compare their performance by keeping other parameters constant. In order to reduce the complexity of using a deinterlever at the receiver end, we used interleaving over the interference for the purpose of system simulation, i.e. interference is interleaved before adding it to the channel symbols.

2.3.1 Convolutional Interleavers

A convolutional interleaver has memory and its operation depends not only on current symbols but also on previous symbols. In a convolutional interleaver the code symbols are sequentially shifted into the bank of *N* registers. The first send the data directly through, after that each successive register provides *J* symbols more storage than the preceding one did. The data is sequentially entered into each bank, one per symbol. The data is read out in the same manner using a commutator switch. The deinterleaver performs the inverse operation, therefore the input and output commutators for both interleaving and deinterleving must be synchronized. The symbol depth of the interleaver is, of course, chosen to match the symbol length of the convolutional encoder.

The performance of a convolutional interleaver is very similar to that of a block interleaver. It is more complicated than a simple row vs. column block interleaver. The most important advantage of this structure over block interleavers is a reduction by two in the memory and end-to-end throughput delay.

2.3.2 Pseudorandom Interleavers

The pseudorandom interleaver uses a fixed random permutation and maps the input sequence according to the permutation order. They are generated by using a random number generator to produce permutations map of integers from 1 to *N*. To create the pseudorandom interleaver map, generate *n* random numbers and rearrange them in ascending order or descending order. Therefore every permutation involving a

block size of N is achieved. Pseudorandom interleaving is a random mapping between input and output positions, generated by means of a pseudorandom number generator.

2.3.3 Block Interleavers

The block interleaver is the most commonly used interleaver in a communication system. It writes in column wise from top to bottom and left to right and reads out row wise from left to right and top to bottom. A block interleaver basically accepts the coded symbols in blocks from the encoder and rearranges them without repeating or omitting any of the symbols in the block. The number of symbols in each block is fixed for a given interleaver. Block interleavers tend to give poor performance because they do not break apart certain input sequences which result in low weight code words.

2.4 Modulation

Modulation is the process by which symbols are transformed into waveforms that are compatible with the characteristics of the channel. It is the process of varying a periodic waveform in order to use that signal to convey a message. Normally a high-frequency sinusoid waveform is used as carrier signal. The three key parameters of a sine wave are its amplitude, its phase and its frequency, all of which can be modified in accordance with a low frequency information signal to obtain the modulated signal.

The frequency of the carrier signal is usually much greater than the highest frequency of the input message signal. According to Nyquist sampling theorem the simulation sampling rate F_s must be greater than two times the sum of the carrier frequency and the highest frequency of the modulated signal in order to recover the message correctly. There are two different modulation techniques available: one is baseband and the other is bandpass. In this thesis we use baseband modulation for the purpose of simulation, also known as the low pass equivalent method, since it requires less computation.

A device that performs modulation is known as a modulator and a device that performs the inverse operation of modulation is known as a demodulator. Analog and digital modulation facilitate frequency division multiplexing (FDM), where several low pass information signals are transferred simultaneously over the same shared physical medium, using separate band pass channels. Modulation can also be used to minimize the effects of interference. A class of such modulation schemes, known as spread-spectrum modulation, requires a system bandwidth much larger than the information bandwidth for interference rejection, and is studied in detail in the further sections of this thesis.

2.4.1 Analog Modulation

The aim of analog modulation is to transfer an analog low pass signal, for example an audio signal or TV signal, over an analog band pass channel, for example a limited radio frequency band or a cable TV network channel.

2.4.2 Digital Modulation

The aim of digital modulation is to transfer a digital bit stream over an analog band pass channel, for example a public switched telephone network or a limited radio frequency band.

In this thesis we assume a linear modulation scheme such as phase shift keying (PSK) or quadriphase shift keying (QPSK).

2.5 Direct Sequence Spread Spectrum System

2.5.1 Spread-Spectrum Communication Systems

Spread spectrum communications is one of the widely used data communication schemes nowadays. These techniques are used for a variety of reasons, including the establishment of secure communications, increasing resistance to natural interference and jamming, and to prevent detection. It has many features that make it suitable for secure communications, multiple access scenarios, and many other properties that are desirable in a modern communication system.

Spread Spectrum is a method of transmission in which the signal occupies a bandwidth in excess of the minimum necessary to send the information. It employs direct sequence, frequency hopping or a hybrid of these, which can be used for multiple

access and/or multiple functions. This technique decreases the potential interference to other receivers while achieving privacy. Spread spectrum generally makes use of a sequential noise-like signal structure to spread the normally narrowband information signal over a relatively wide band of frequencies. The receiver correlates the received signals to retrieve the original information signal. The band spread is accomplished by means of a code which is independent of the data and synchronized reception with the code at the receiver is used for de-spreading.

In spread spectrum the signal that has a limited defined bandwidth is spread to occupy a higher bandwidth, with its power spread over a wide range, by multiplying that signal with a higher frequency sequence. The spreading will significantly reduce the possibility of corrupting the data, intentionally or unintentionally. This is one of the main features of spread spectrum, the interference suppression capability. When the spread signal is interfered by additive white Gaussian noise (AWGN), we will not notice any significant improvement if we choose spread spectrum. But, when an intentional noise is applied, it is usually band limited to the range we are using. When we spread the signal, the intentional noise (usually termed the jammer) will make one of two choices. It will either spread its band limited power spectral density over the new bandwidth, which will reduce its effect on our signal, or stay at its original bandwidth, which will cause it to affect only a portion of our data. Such effect might be further reduced by error correction coding at the receiver end. This means that in both cases, the choice of spreading will reduce the jammer's effect significantly. While the typical interference encountered by a modern spread spectrum signal will not be arising from a jammer, the idea of a jammer has been historically used to illustrate the interference suppression capability of spread

spectrum. Some of the more interesting and desirable properties of spread spectrum can be summarized as:

- Good anti jamming performance.
- Low power spectral density.
- Interference limited operation, i.e. the whole frequency spectrum is used.
- Multi path effects are reduced considerably with spread spectrum applications.
- Random access probabilities, i.e. users can start their transmission at any time.
- Privacy due to the use of unknown random codes.
- Multiple access, i.e. more than one user can share the same bandwidth at the same time.

Spread spectrum systems are classified according to the ways that the original data is modulated by the PN code. The most commonly employed spread spectrum techniques are the following:

Direct Sequence Spread Spectrum (DSSS): In DSSS, the baseband signal is multiplied by a pseudorandom code or pseudonoise (PN) signal, which has a higher bit rate than the original signal. This will spread the spectrum of the baseband signal. In next section, DSSS technique is described in detail.

Frequency Hopping Spread Spectrum (FHSS): Frequency-hopping spread spectrum (FHSS) is a method of transmitting radio signals by rapidly switching a carrier among many frequency channels, using a pseudorandom sequence known to both the transmitter and the receiver. This will result in modulating different portions of the data

signal with different carrier frequencies. This technique makes the data signal hop from one frequency to another over a wide range and this hopping rate is a function of the information rate of the signal. The specific order in which frequencies are occupied is a function of a code sequence. The transmitted spectrum of a frequency hopping is different from that of the direct sequence system.

Hybrid System (DS/FFH): This is a combination of both the direct sequence and frequency hopping techniques. Here, one data bit is divided over frequency hop channels i.e. carrier frequencies. In each frequency hop channel one complete PN code is multiplied with the data signal.

In this thesis, the emphasis is going to be on the DSSS System. A detailed description of DSSS system is given in next section.

2.5.2 Direct Sequence Spread Spectrum Digital Communication Systems

Direct sequence spread spectrum is one of the most widely used spread spectrum techniques. The basic elements of DSSS digital communication system are illustrated in Figure 2.1. We observe that in addition to the basic elements of a conventional digital communication system, a spread spectrum system includes two identical pseudorandom sequence generators, one interfacing with the modulator and the other with the demodulator. As with all spread spectrum schemes, DSSS uses a unique code to spread the baseband signal, allowing it to have all the advantages of spread spectrum techniques. A random or pseudonoise signal is used to spread the baseband signal, causing fast phase transitions in the carrier frequency that contains data. The basic method for

accomplishing spreading is shown in Figure 2.2. The spreading sequence is a pulse stream with pulse values of +1, -1. After spreading the base-band signal, the resulting spread signal is then modulated and transmitted through the specified medium. Binary phase shift keying (BPSK) is a widely used digital modulation scheme for spread spectrum systems and we use the same in this thesis.

When the modulated data is received at the demodulator port, the signal is de-modulated using a BPSK demodulator that has a synchronized carrier frequency with the transmitter one. The spread signal will be at the output of the demodulator. This is then multiplied with the locally generated PN sequence. If the locally generated PN sequence is correlated with the one that was used in transmitter, the signal is de-spread, yielding the original signal. The spectrum spreading is illustrated in Figure 2.3, which shows the convolution of two spectra, the narrow spectrum corresponding to the message signal and the wide spectrum corresponding to the signal from the PN generator.

Spreading factor of the spread spectrum is an important parameter which defines the overall gain of the system. It is also termed as processing gain, which is defined by:

$$G_p = \frac{BW_t}{BW_i}$$

is the ratio of the transmission bandwidth BW_t and the information bandwidth BW_i. It helps in determining the number of users that can be allowed in a multiple access system, the amount of multi-path effect reduction and the difficulty to jam or detect signals. For spread spectrum systems, it is always better to choose a high processing gain. But this comes as a trade off with system complexity.

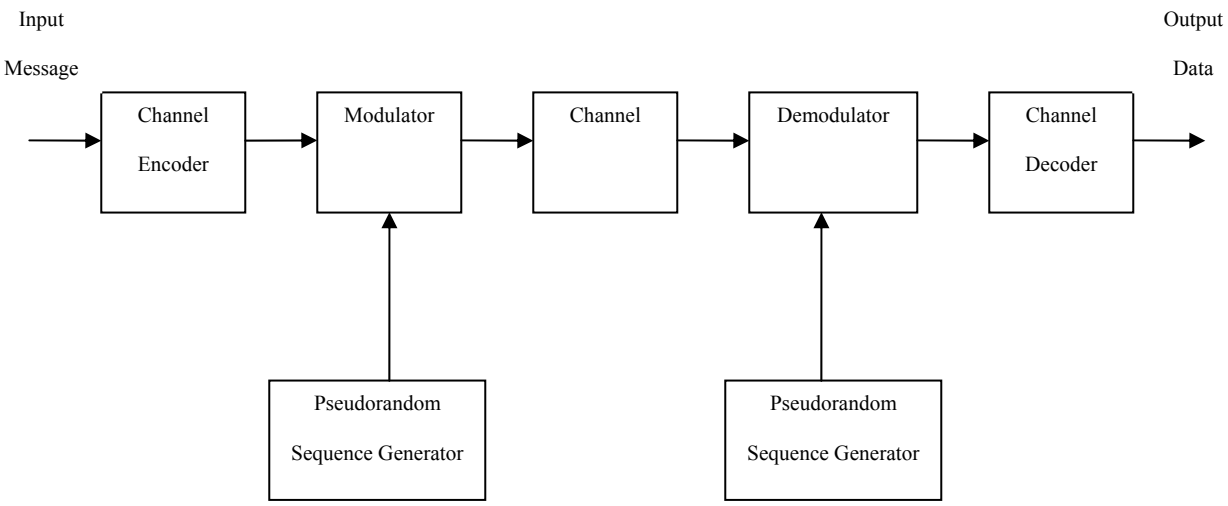

Figure 2.1: Model of direct sequence spread spectrum digital communication system

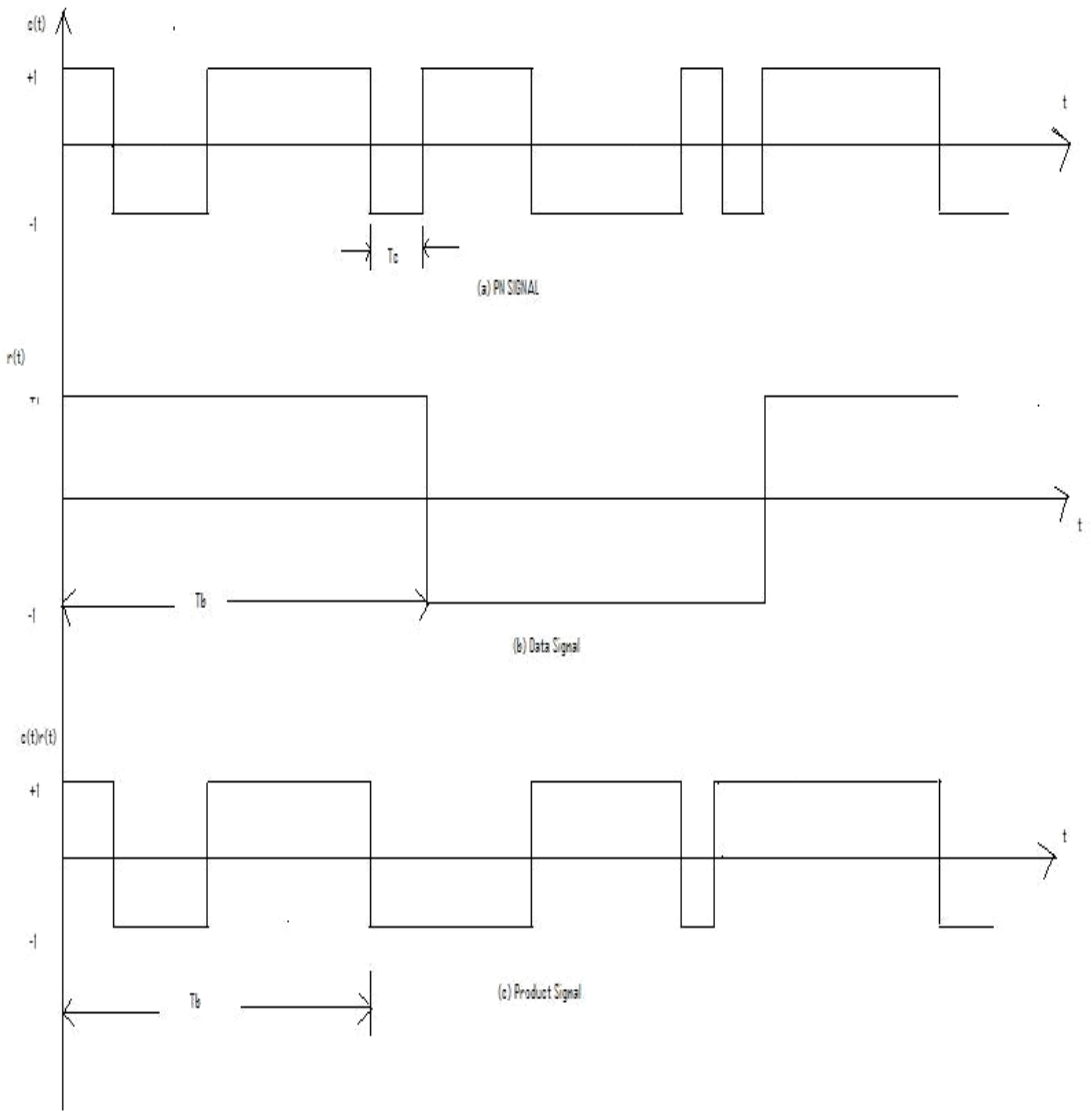

Figure 2.2: Spreading code with pulse values +1,-1

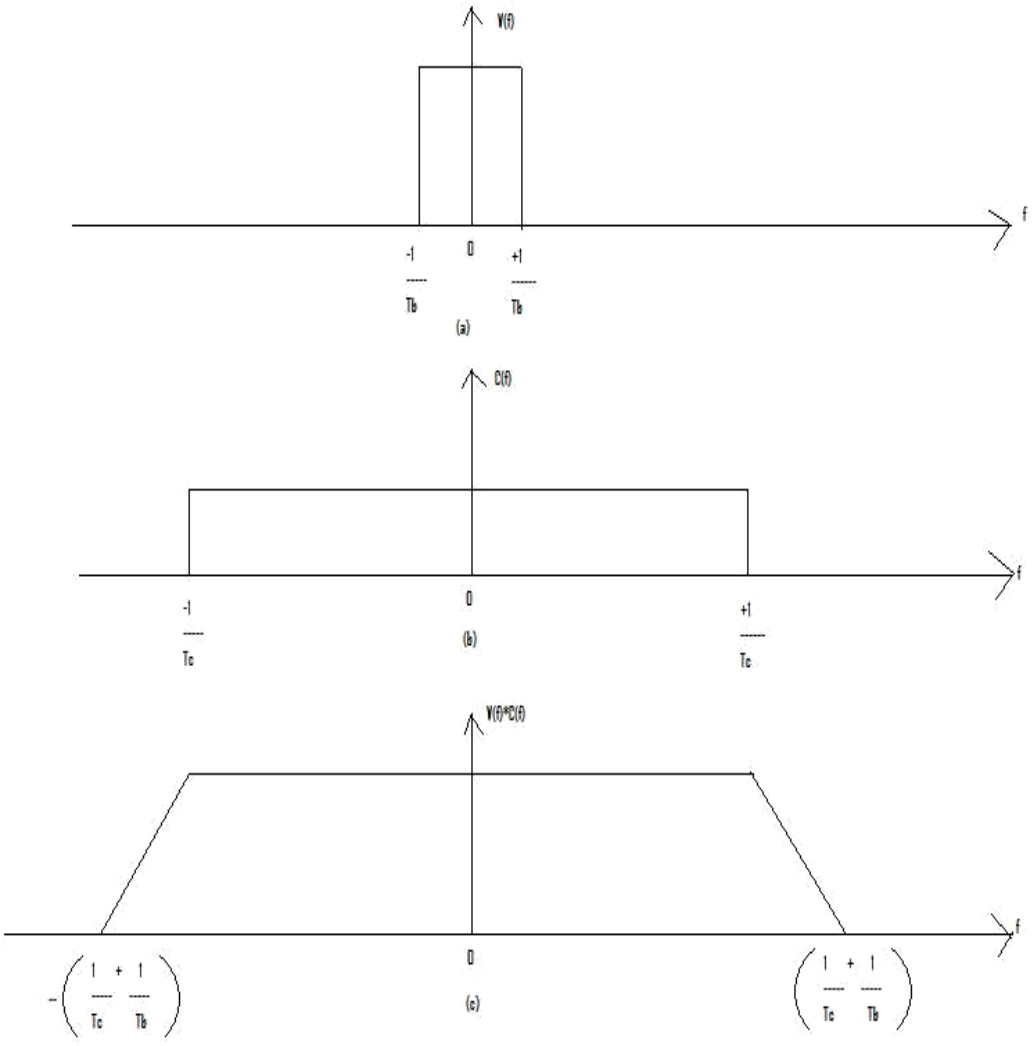

Figure 2.3: The spectrum spreading

A DSSS digital communication system can be classified into four major parts, which are: pseudo noise sequence generator, spreading and modulation (transmitter), demodulation and de-spreading (receiver), PN synchronization. Each part of the DSSS communication system is described in detail as follows.

Pseudo Noise Sequence Generator:

Pseudo Noise (PN) signals play a key role in DSSS systems, as they are the ones responsible for the spreading and de-spreading of the baseband signal. These signals are generated in a deterministic way but appear to be random or noise-like. PN sequences are considered to have noise like properties for an outsider, but they are known to the two devices using them. They are considered pseudo random because the sequences are actually deterministic and are known to both the transmitter and the receiver.

There are three basic properties that can be applied to a periodic binary sequence as a test of the appearance of randomness. They are balance property, run property and correlation property. One of the well known and easy to generate PN sequences are the maximum length sequences (MLS). MLS satisfy all three PN properties. An MLS is generated by the use of shift registers and some logic circuitry in its feedback path. A feedback shift register is said to be linear if its feedback logic circuit consists entirely of modulo-2 adders (XOR gates).

DSSS Transmitter:

In DSSS the baseband waveform is multiplied by the PN sequence. The PN is produced using a PN generator. This generator consists of a shift register, and a logic circuit that determines the PN signal. After spreading, the signal is modulated and

transmitted. The most widely used modulation scheme is binary phase shift keying (BPSK).

In BPSK a transition from a one state to a zero state (or the other way around) will cause a 180 degree phase shift in the carrier signal. A BPSK modulator consists of a multiplier circuit that directly multiplies the incoming signal with the carrier frequency generated by the local oscillator. Other transmitter schemes also exist. Some of them use the PN spreading after the baseband signal is modulated using BPSK. This will spread the passband signal. In the receiver, the de-spreading takes part before the signal gets demodulated. Based on the system architecture, one might decide which scheme to use.

DSSS Receiver:

In the demodulator section, we simply reverse the process. We demodulate the BPSK signal first, pass it through a low pass filter, and then de-spread the filtered signal, to obtain the original message. The receiver carrier frequency should be synchronized with the transmitter one for data detection.

As for the PN sequence in the receiver, it should be an exact replica of the one used in the transmitter, with no delays, otherwise it might cause severe errors in the incoming message. Usually a delay locked loop is used to overcome this issue, and lock the timing of the transmitted PN sequence with the one locally generated. Once the incoming PN code is correlated with the locally generated one, we can de-spread the signal.

After the signal gets multiplied with the PN sequence, the signal de-spreads, and we obtain the original bit signal that was transmitted. The signal is then applied to a

decision device that will take care of the signal shaping, and leveling. The original data signal is then obtained. In the presence of noise, extra circuitry is needed to compensate the signal degradation that affects the transmitted signal.

PN Synchronization:

In a spread spectrum system, the generated PN code at the receiver end must be aligned to the received PN sequence, otherwise, the PN code misalignment will result in ineffective de-spreading of the signal. Synchronization is usually accomplished first by an acquisition of the initial PN code alignment and then followed by a tracking process to eliminate a possible new phase shift introduced to the received signal during the signal reception process. Without synchronization, the spread spectrum will appear as noise and ineffective de-spreading will be achieved at the receiver end. Therefore, synchronization of the PN code is crucial for data reception.

Interference is added to the spread spectrum signal during transmission through the channel. The characteristics of the interference depend to a large extent on its origin. Usually the interference is categorized as being either broadband or narrowband relative to the bandwidth of the information bearing signal, and either continuous in time or pulsed in time. In this thesis we don't apply any specific constraints or statistical meanings to the interference except a fundamental power limitation on the interfering signal.

2.6 Generalized Direct Sequence Spread Spectrum

It is shown in [2] that ordinary DSSS can be improved by allowing the transmitted vector to more closely approximate a uniform distribution on the surface of an *N*-dimensional sphere. In [1] it is shown that ordinary DSSS is only a special case of random modulation, and that random modulation becomes asymptotically optimum (as *N* gets large), minimizing the signal-to-interference ratio required to guarantee a given worst-case performance level, when the message symbol is uniformly distributed on the surface of an *N*-dimensional sphere.

Figure 2.4 shows all the possible transmitted vectors for ordinary DSSS as black dots when modeled as a 3-dimensional signal. When ordinary DSSS is modeled in the N–dimensional signal space, all the message symbols are randomly distributed on the vertices of an N-dimensional cube space using a random chipping sequence of {-1, +1}, where N is the number of chips per symbol. Now in generalizing ordinary DSSS, along with the vertices of ordinary DSSS the midpoints of the edges and faces of the 3-D cube are considered cube as possible transmitted same energy vectors and these points are projected radially onto the surface of a 3-D sphere as shown in Figures 2.5 and 2.6.

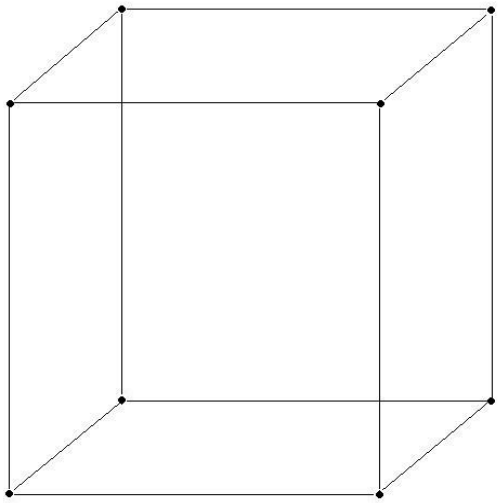

Figure 2.4: Transmission vector distribution for DSSS

The resulting transmitted vectors for $N = 3$ can be expressed as $\frac{1}{\sqrt{3a}}(\pm a, \pm a \pm a)$ for the vertices, $\frac{1}{\sqrt{2a}}(\pm a, \pm a, 0)$, $\frac{1}{\sqrt{2a}}(\pm a, 0, \pm a)$ and $\frac{1}{\sqrt{2a}}(0, \pm a, \pm a)$ for the edges, and $\frac{1}{a}(\pm a, 0, 0)$, $\frac{1}{a}(0, \pm a, 0)$ and $\frac{1}{a}(0, 0, \pm a)$ for the faces, for a total of 26 possible unit-energy vectors.

Generalized DSSS can be viewed as the use of a novel chipping sequence where a 0 is allowed in the chip sequence in addition to {-1, +1}. This method has a transmitted vector distribution which more closely approximates a uniform distribution on the surface of an *N*-dimensional sphere than ordinary DSSS, with a slight increase in system complexity. As shown in [2] when extended to the *N*-dimensional signal space, this

would produce 3^N-1 possible unit energy vectors of all hamming weights 1 through N for a generalized DSSS.

$$\sqrt{\frac{1}{\sum_{j=1}^{N} a_j^2}}\{-a,0,+a\}^N$$

The spreading sequence for a generalized DSSS is a pulse stream with pulse values of +1, -1 or 0, a deviation from the usual sequence of +1 or -1. The spreading technique for generalized DSSS is shown in Figure 2.7. The baseband message signal is a rectangular pulse of duration T_b. This signal gets multiplied by the PN sequence generator, which has the PN code sequence of -1, +1, 0. Therefore the bandwidth of the message signal is spread into the wider bandwidth occupied by PN generator signal.

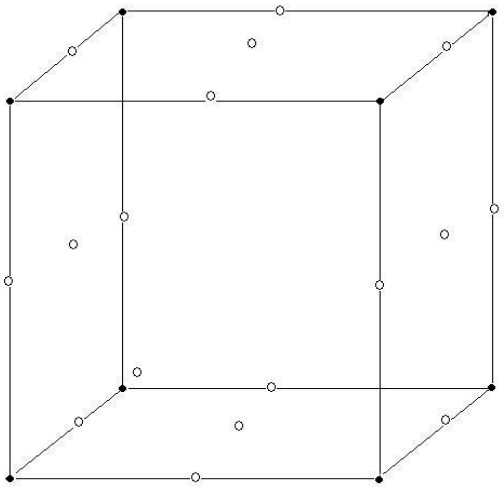

Figure 2.5: Transmission vector generalization for DSSS

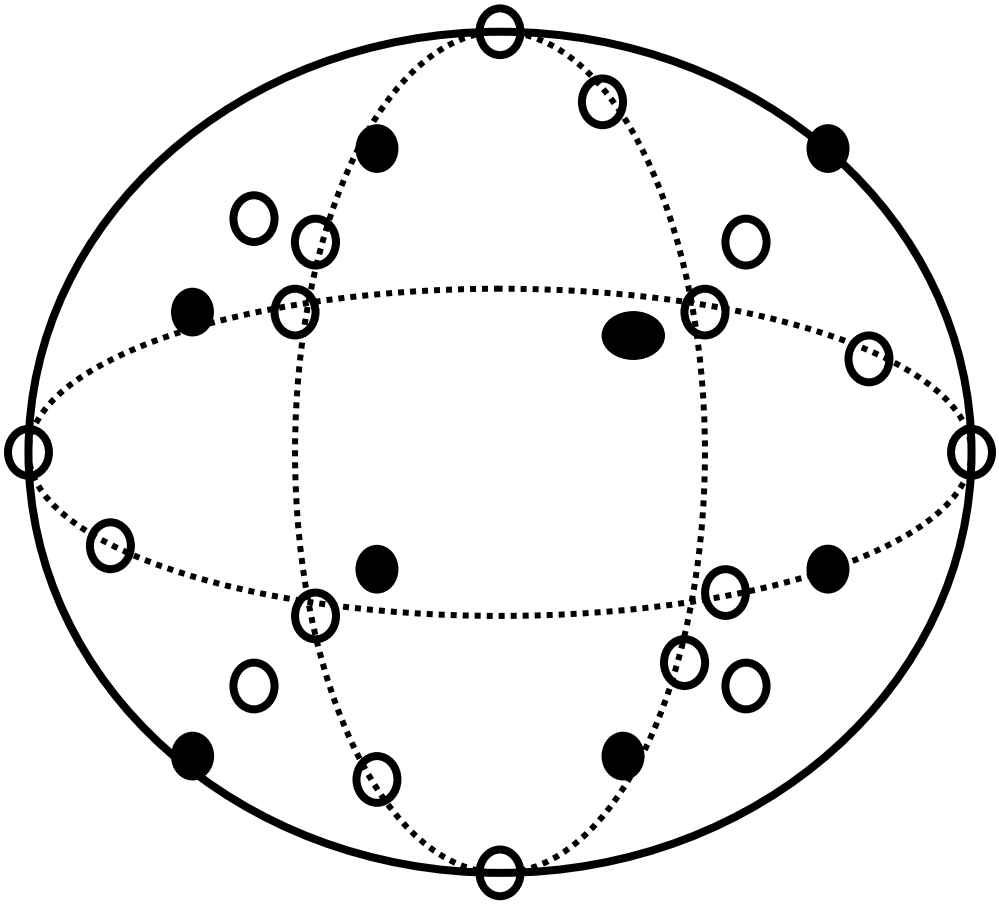

Figure 2.6: Transmission vector distribution on sphere for DSSS

2.7 Channel Assumptions

For most communication systems, importance is given to channels whose accurate statistical models are known. However, in many practical communication situations the communicator does not have access to a complete statistical description of the interfering signals in the channel. The channel statistics may change with time in an unknown and arbitrary way, making to it impractical to predict the channel properties, such as hostile jamming or multiple access interference from other non-cooperative transmitters.

For the analyses of robust communication systems, highly optimistic models of the interfering signal are commonly used while channel modeling. In case of anti-jam applications, several models for the interfering signal have been proposed. Among these, pulse jamming, where the jammer transmits at full power for a fraction of the time and keeps silent for the remainder, was considered to be one of the worst forms of interference. Broadband and partial-band noise jamming have also been considered. Multi-tone and repeat-back jamming has also been investigated in many contexts. Continuous wave jamming and blades system have been employed in different communication systems. Gaussian approximation has also been employed as a model for a large number of transmitters. For multiple access applications, performance analysis is usually based on exact error probability using fixed signature sequences for the transmitters.

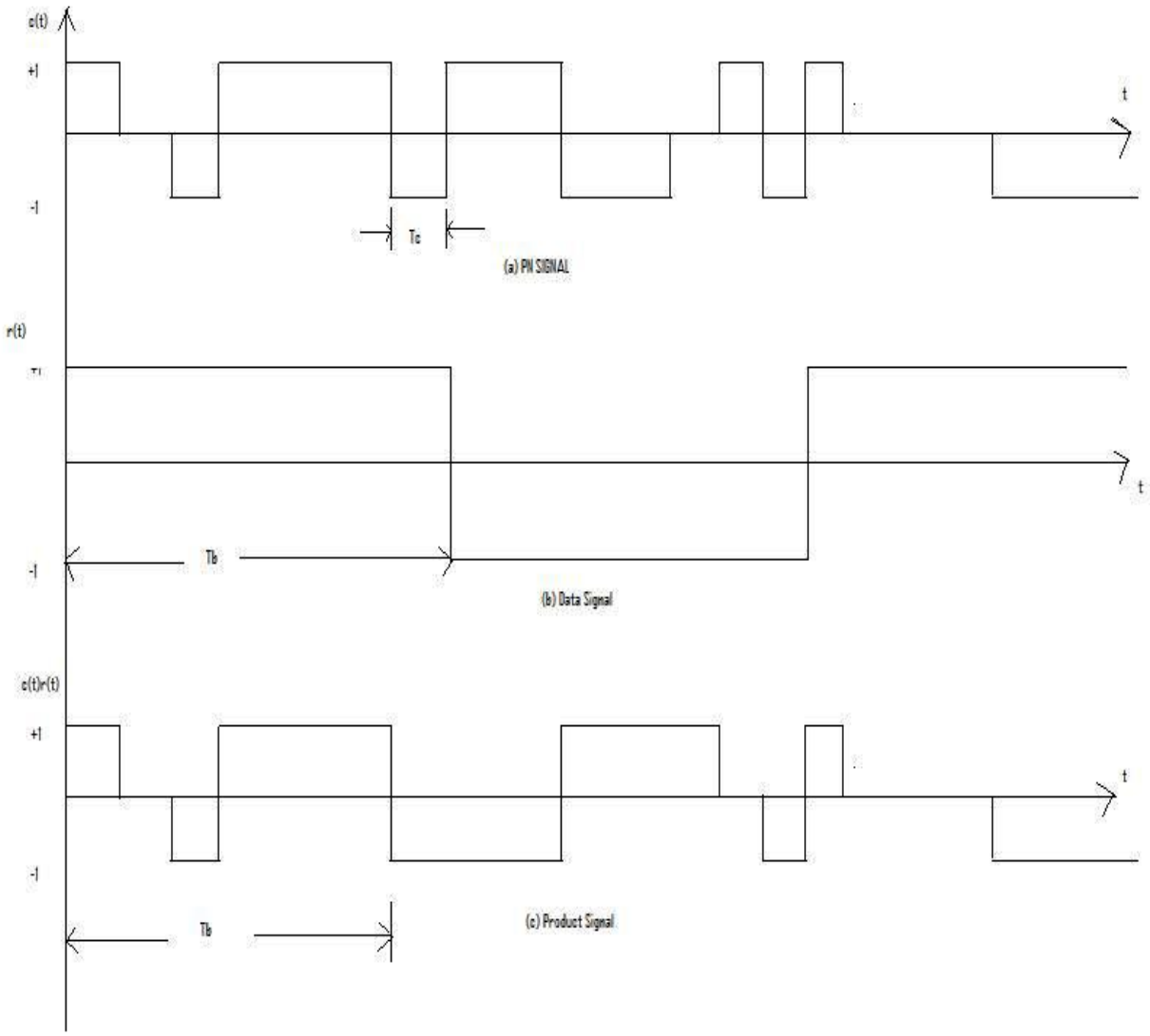

Figure 2.7: Spreading sequence with pulse values -1, 0, +1

Any communication system which assumes a fixed statistical description for interfering signal suffers from a weakness that it is being highly optimistic. Since a jammer has the same design options as a transmitter the most damaging signal that the jammer can produce may not necessarily be from these simple models. Hence the worst-case performance analysis of a communication system should consider all the possible models for the interference. The definition of a robust communication requires a worst-case analysis over all possible interference situations in order to guarantee a minimum level of reliability in information transfer.

In modeling the channel, as mentioned by Hizlan in [3], we choose to be on the side of excessive pessimism thus consider a channel model in which nothing is known about the interference except that it is bounded in power. Furthermore, it is also independent of the transmitted signal and thermal noise. Therefore, our measure of reliability is the worst-case error probability over all such unknown signals.

Figure 2.8 describes the basic channel model considered in this thesis. Looking into the model, basically an integer message $m \in \{1,...,M\}$ is sent over a waveform channel in a time period of duration T seconds. The transmitted signal $x(t)$ is corrupted by two independent interference signals resulting in a received signal which is given by:

$$Y(t) \cong x(t) + W(t) + S(t), \quad 0 \le t \le \mathrm{T} \tag{2.1}$$

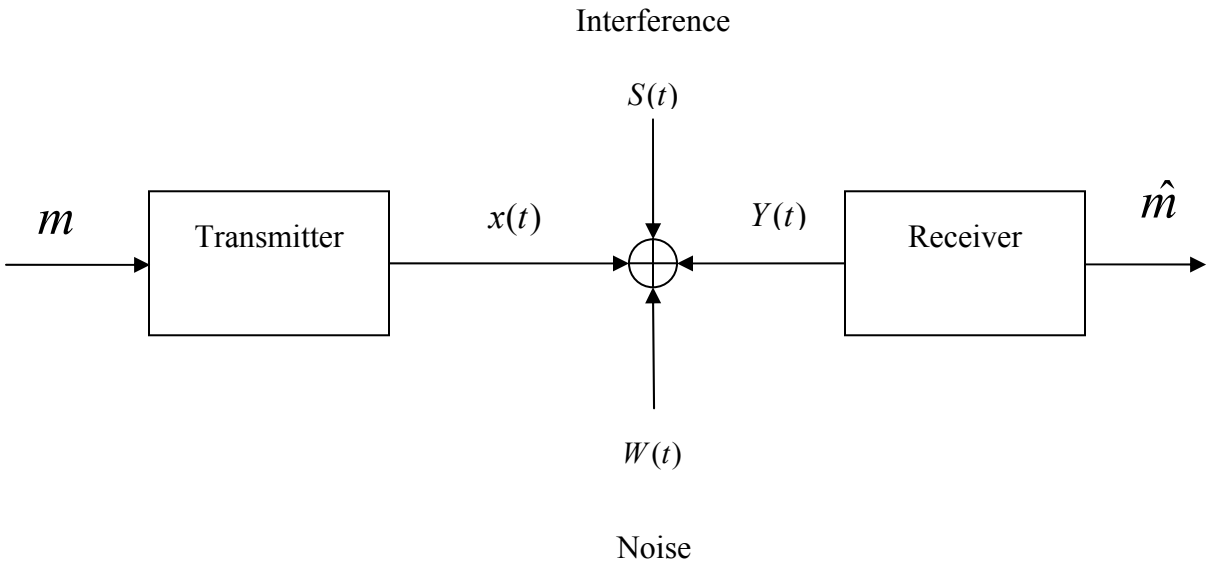

Figure 2.8: Channel Model

Making use of correlation the receiver guesses $\hat{m}$ from $Y(t)$. The signal $W(t)$ represents white Gaussian noise process with one-side power spectral density N_0 *W/Hz*. The signal $S(t)$ represents different sources of interference with partially unknown statistics, such as jammers, non-cooperative transmitters, etc. Here, no such restrictions are imposed on $S(t)$ except that it is independent of m and $W(t)$, and also its time-averaged power does not exceed P_J :

$$\frac{1}{T}\int_0^T S^2(t)dt \le P_J \tag{2.2}$$

The important feature of this channel is that the interfering signal can change with time in an arbitrary way, subject only to the fundamental limitation of bounded power. Therefore $S(t)$ can have arbitrary, time-varying, non-Gaussian statistics and it may also possess memory.

CHAPTER III

CHANNEL MODEL

3.1 Communication System Model

Spread spectrum techniques and especially direct sequence modulation have long been employed as a means of achieving good communication when the statistical description of the channel interference is at least partially unknown. In this chapter we consider a communication system composed of a convolutional encoder, convolutional/pseudorandom interleaving, direct sequence modulation and Viterbi decoder, operating on a channel corrupted by thermal noise and by an unknown interfering signal of bound power. The channel model used by Hizlan and Hughes in [1] considers a much broader class of interference signals than has previously been considered. Our aim is to simulate the worst-case error probability of this coded generalized DSSS communication system and compare results with ordinary DSSS for different code rates and constraint lengths of convolutional codes.

3.1.1 Waveform Model

The communication system block diagram is shown in Figure 3.1. The data generator gives out $5 \times K$ bits, where K is the constraint length of the convolutional encoder. We have chosen the message length as $5 \times K$ since it is the decoding depth of the Viterbi decoder used in this thesis as described in chapter 4. The convolutional encoder coverts these bits into $L = \frac{5 \times K}{r}$ encoded symbols by adding some redundancy for error checking at the receiver, where r is the code rate of convolutional encoder. The transmitter generates L coded symbols every T seconds. The encoded symbols are then transmitted by DS modulation with N pseudo-noise chips per code symbol.

The N pseudonoise chips are generated randomly using a PN generator and the sequence generated is from {-1, 0, +1} for generalized DSSS and from {-1, +1} for ordinary DSSS. These randomly generated chips are then multiplied with a normalization factor (equal to 1 for ordinary DSSS) in order to account for the energy lost due to the "0" chip in the generalized sequence. Assume that a given transmitted message of length L is called message m. The basic channel model is illustrated in Figure 3.1 and a detailed description is given in section 2.5. Here we replace $x(t)$ by $x^m(t)$ to show the dependence of the transmitted signal on the message m. During transmission, $x^m(t)$ is corrupted by two independent, additive noise processes so that

$$Y(t) \cong x^m(t) + W(t) + S(t), \quad 0 \le t < T$$

is received. Here $W(t)$ is a white Gaussian noise process with one-sided power spectral density N_0 W/Hz and $S(t)$ is an arbitrary signal independent of m and $W(t)$.

The codeword associated with message *m* are $x^m = \left(x_0^m, ..., x_{L-1}^m\right)$ for convolutional coded generalized direct-sequence spread spectrum. The symbols are convolutional or pseudorandom interleaved to form an interleaved code waveform

$$Z^m(t) \cong \sum_{l=0}^{L-1} x_l^m u(t - J_l T_S), \quad 0 \le t < T \tag{3.1}$$

where $u(t) = 1$ in the interval $[0, T_S)$ and vanishes outside, and $T_S = T/L$ is the symbol duration. In equation (3.1) the index sequence $\{J_0, ..., J_{L-1}\}$ represents interleaving of the symbols, where $\{J_0, ..., J_{L-1}\} = \{C_0, ..., C_{L-1}\}$ for convolutional interleaving and $\{J_0, ..., J_{L-1}\} = \{P_0, ..., P_{L-1}\}$ for pseudorandom interleaving.

The interleaved code word is binary phase-shift (BPSK) modulated and DS spread by the PN sequence.

$$X^m(t) \cong c\sqrt{2E/T}\cos(wt)C(t)Z^m(t), \quad 0 \le t < T \tag{3.2}$$

where *E* is the energy per code word at the receiver, $T_C = T_S/N$ is the chip duration, *w* is the carrier frequency with $w > 2\pi T_c^{-1}$, and $C(t)$ is the spreading waveform

$$C(t) \cong \sum_{i=0}^{NL-1} A_i v(t - iT_C), \quad 0 \le t < T.$$

Here, $v(t)$ is a low-pass chip waveform that satisfies $\int_0^{T_C} v^2(t)dt = T_C$ and vanishes outside the interval $[0, T_C)$.

The pseudo-noise sequence $\{A_i\}$ is modeled as an independent identically distributed sequence of random variables which satisfy $Pr\{A_i = +1\} = Pr\{A_i = 0\} = Pr\{A_i =$

-1} = 1/3 for generalized DSSS and $Pr\{A_i = +1\} = Pr\{A_i = -1\} = 1/2$ for ordinary DSSS, and they are independent of $\{J_0, \ldots, J_{L-1}\}$. Therefore, the energy normalization constant

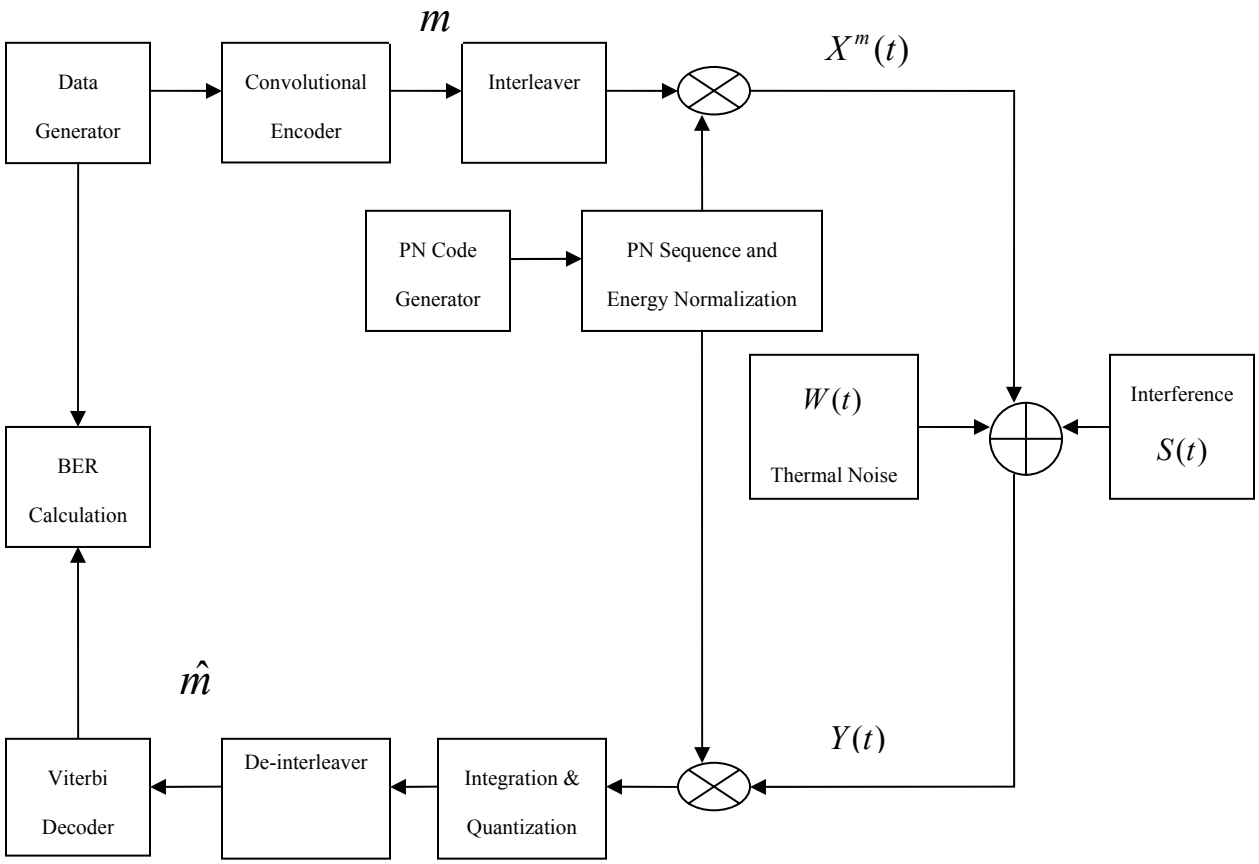

Figure 3.1: System Model

c takes the form as shown in [2, equation 2] with $c = \sqrt{\frac{NL}{u}}$, where u represents the total number of non-zero chips per L encoded symbols in the PN sequence.

The deinterleaver at the receiver end can be represented as

$$Z^{\hat{m}}(t) \cong \sum_{l=0}^{L-1} x_l^m u(t - B_l T_s), \quad 0 \le t < T \tag{3.3}$$

where $u(t) = 1$ in the interval $[0, T_S)$ and vanishes outside, and $T_S = T/L$ is the symbol duration. In equation (3.3) the index sequence $\{B_0, ..., B_{L-1}\}$ represents deinterleaving of the symbols, where $\{B_0, ..., B_{L-1}\} = \{D_0, ..., D_{L-1}\}$ for convolutional deinterleaving and $\{B_0, ..., B_{L-1}\} = \{Q_0, ..., Q_{L-1}\}$ for pseudorandom de-interleaving.

During channel simulation in order to avoid the overhead of interleaving and deinterleaving, we directly interleave the interference in the channel while adding it to the signal as described in Figure 5.1 of Chapter 5. In such a situation the deinterleaved signal at the receiver end can be represented as

$$\hat{Y}(t) \cong x^m(t) + W(t) + S(t), \quad 0 \le t < T \tag{3.4}$$

Where $W(t)$ is a zero-mean white Gaussian noise process with one-sided power spectral density $N_0 W / Hz,$, and $S(t)$ is an interleaved unknown and arbitrary interfering signal. The signal $S(t)$ represents interference from sources with unknown statistics, such as multiple-access interference, jamming and impulsive noise. In this thesis, we consider a communication situation in which nothing is known about $S(t)$ except that it is independent of $\{A_i\}$, $W(t)$, $\{J_0, ..., J_{L-1}\}$, $\{B_0, ..., B_{L-1}\}$ and that its energy is

constrained. Therefore $S(t)$ may be random or deterministic, narrow-band or wide-band, stationary or time-varying, Gaussian or non-Gaussian.

In order to bound the error probability of a receiver for (3.4), we must place some constraint on the interference energy $\int_0^T S^2(t)dt$. Here the interference energy is strictly bounded, i.e.

$$\int_0^T S^2(t)dt \le TP_I . \tag{3.5}$$

At the receiver end, the same normalized PN code sequence used at the transmitter end is multiplied with the received noisy chips before any hard decision is performed over the received signal. After the multiplication the chips are converted to symbols and then fed to decision box. In our case the decision box will compare the input signal to zero threshold and output +1 if it is greater than zero and -1 if it is less than zero. Once the symbols are out of the decision box, they are fed to a Viterbi decoder to get back the original message bits.

At the receiver end first we integrate all the received bits described as

$$X^{\hat{m}}(t) \cong \int_0^{T_s} \hat{Y}(t)C(t)\cos(wt), \quad 0 \le t < T_s \tag{3.6}$$

and later fed to decision box to perform hard quantization, whose function is described as

$x^{\hat{m}}(t) = 1$ if $X^{\hat{m}}(t) < 0$

$x^{\hat{m}}(t) = 0$ if $X^{\hat{m}}(t) \ge 0$.

3.1.2 Worst-Case Probability of Error

In this section we investigate the worst-case performance of the system described above, following the development in [3]. As discussed in [3], we convert the waveform channel described above into an equivalent vector channel representation which is easier for simulation.

Given $S = s$, the conditional probability of error of the receiver as given in [3, equation 8] is

$$\varepsilon\left(s, v^2\right) \cong \frac{1}{NL} \sum_{m=1}^{NL} \Pr\{\hat{m} \neq m / S = s\} \tag{3.7}$$

where $v^2 \cong 2E/NLN_0$ is the chip signal-to-noise power ratio.

We calculate error probability when nothing is known about S except a constraint on the energy of $S(t)$. Note that the energy constraint (3.5) implies that $\boldsymbol{S}$ [3, equation 9] satisfies

$$P(S) \cong \frac{1}{NL} \sum_{i=0}^{NL-1} S_i^2 \leq E/NL\sigma^2 \tag{3.8}$$

for the vector channel representation. The interference energy over L symbols is therefore limited to a maximum of

$$\sum_{i=0}^{NL-1} S_i^2 = \frac{NLE}{NL\sigma^2}$$

$$= \frac{NLE}{\left(\frac{E_b}{N_i}\right)} \tag{3.9}$$

with signal-to-interference ratio as

$$\frac{E_b}{N_i} \cong \frac{E}{P_i T} . \frac{W}{R} = NL\sigma^2$$

where $W = 1/T_C$ is the system bandwidth, $R = 1/T$ is the data rate and $\sigma^2 = E/T\,P_i$ is the signal-to-interference power ratio.

For the purpose of simulating the worst-case performance of the system, we assume a canonical distribution for the interference and consider *D* chips out of the NL chips to be affected by interference at maximum power, where $1 \le D \le NL$. Now the interference energy vector over *D* chips is defined as

$$S_j = \sqrt{\frac{NLE}{D\left(\frac{E_b}{N_i}\right)}}, j = 0,1,....D-1$$

$$S_j = 0, j = D-1,...NL-1$$

We obtain the worst-case error probability by maximizing the simulated error probability over *D* as it varies from 1 to NL chips, thus maximizing over the interference.

CHAPTER IV

INTRODUCTION TO CONVOLUTIONAL CODES

The history of error-correction coding began in 1948 with the publication of a landmark paper from Claude Shannon "A mathematical theory of communication" [9]. Since Shannon's work, much effort has been devoted to the implementation for controlling errors in noisy environment. In this chapter we speak about the codes that are used in this thesis.

Convolutional and block codes are the most widely used codes today. In this thesis we choose convolutional codes with Viterbi decoding as error control codes.

4.1 Convolutional Codes

Convolutional codes are usually described using two parameters: the code rate and the constraint length. The ratio of k/n is called the code rate (r) where n denotes the number of channel symbols output by the convolutional encoder and k denotes the

number of input bits fed to the convolutional encoder in a given encoder cycle. The code rate of the encoder is a measure of the efficiency of the code. Usually *k* and *n* parameters range from 1 to 8 and the code rate from 1/8 to 7/8.

The constraint length parameter, *K*, denotes the "length" of the convolutional encoder, i.e. how many *k*-bit stages are available to feed the combinatorial logic that produces the output symbols. The quantity *K* is defined by

$$K = k(M + 1)$$

Closely related to *K* is the parameter *M*, which indicates how many encoder cycles an input bit is retained and used for encoding after it first appears at the input to the convolutional encoder. The *M* parameter can be thought of as the memory length of the encoder.

4.1.1 Structure of the convolutional code

A binary convolutional code is generated by passing the information sequence to be transmitted through a linear finite-state shift register. The convolutional code structure is easy to draw from its parameters. First draw the *M* boxes to represent the *M* memory registers. Then draw *n* modulo-2 adders to represent the *n* output bits. Now connect the memory registers to the adders using the generator polynomial. For a (*rate k/n, K*) convolutional code, the shift register consists of *K*-1 stages and *n* linear modulo-2 function generators. The input data is shifted into and along the shift registers a single bit at a time producing a *n*-tuple output for each shift. To illustrate the working of a convolutional encoder, consider the (*rate 1/2, K = 3*) convolutional encoder shown in

Figure 4.1 where the output $U = [U_{i1}, U_{i2}]$ at instant i is obtained from the message bit x_i at instant i and the previous $(K - 1)$ (which equals 2 in our example) message bits x_{i-1} and x_{i-2}.

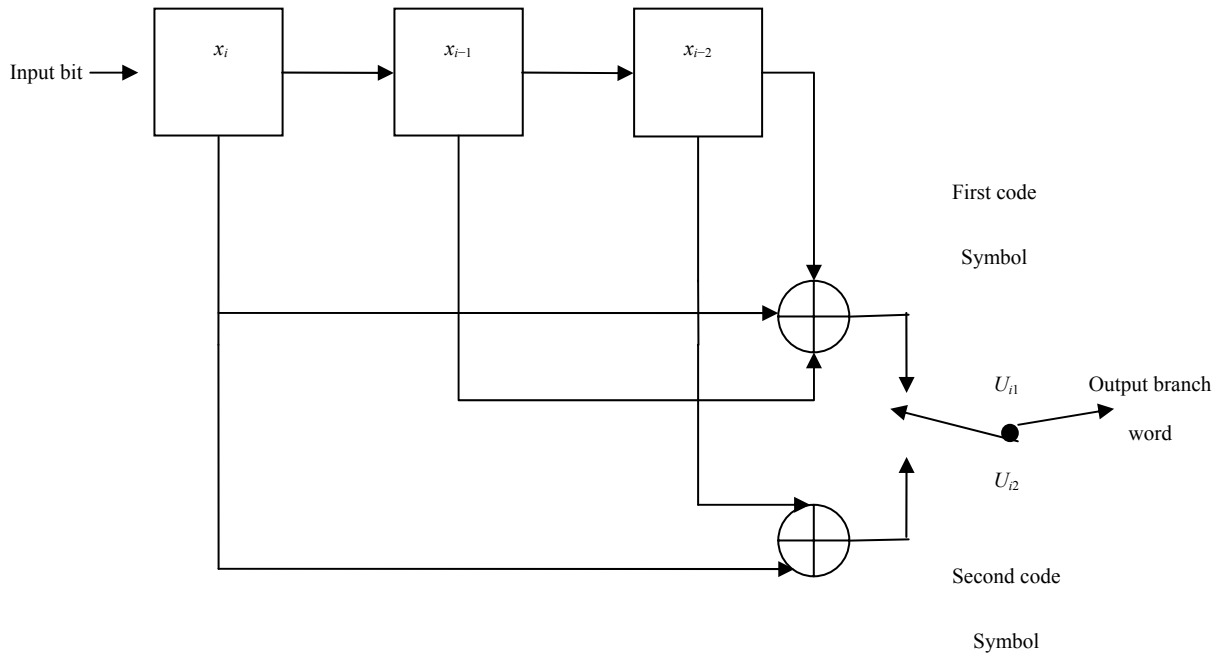

Figure 4.1: Convolutional Encoder (rate 1/2, $K = 3$)

The operator $\oplus$ is the modulo-2 adder operator. Initially, the shift registers are assumed to be in the all zero state. Suppose the first input bit is a 1. At the next clock cycle, the initial content of the registers is moved towards the right by one bit and the message bit occupies the leftmost register. The registers are therefore set to states 1, 0, and 0, i.e., $x_i = 1$, $x_{i-1} = 0$, and $x_{i-2} = 0$. The value of the two output bits U_{ij}, for $i = 1$ and $1 \leq j \leq 2$, are given by

$$U_{i2} = x_i \oplus x_{i-2}$$

$$U_{i1} = x_i \oplus x_{i-2} \oplus x_{i-2}$$

Suppose now that the second bit is a 1. The registers are set to 1, 1, and 0 at the next clock cycle and the codeword U_1 is 11. By following this procedure, the codeword's U_i can be generated for the remaining message bits. The selection of which bits are to be added to produce the output bit is called the generator polynomial (g) for that output bit. The polynomials give the code its unique error protection quality. The generator polynomials for the above encoder are g_1 = [111] and g_2 = [101]. One code can have completely different properties from another one depending on the polynomials chosen.

4.1.2 States of Convolutional Code

Convolutional encoders will exist in different states at different times. Some complex encoders have long constraint lengths and simple ones have short in deciding the number of states they can be in. The (*rate 1/2, K = 3*) code in Figure 4.1 has a constraint length of 3. The number of combinations of bits in the memory are called the states of the code and are defined by

Number of states = 2^{K-1}

where K = the constraint length of the code.

Let us examine the states of the code (*rate 1/2, K = 3*) shown above. This code outputs 2 bits for every one input bit. It is a rate 1/2 code. Its constraint length is 3 and

memory length is 2. The total number of states is equal to 4. The four states of this (*rate 1/2, K = 3*) code are: 00, 01, 10, 11.

4.2 Convolutional Codes with Higher Inputs

We can also create codes where k is more than one such as the (*rate 2/3, K = 8*) code. This code takes in 2 bits and outputs 3 bits. The number of memory registers is 4. The constraint length is 5 x 2 = 10. The code has 16 states.

The procedure for drawing the structure of a (*rate k/n, K*) code where *k* is greater than 1 is as follows. First draw *k* sets of *M* boxes. Then draw *n* adders. Now connect *n* adders to the memory registers using the coefficients of the n_{th} (*kM*) degree polynomial. What you will get is a structure like the one in Figure 4.2 for code (*rate 2/3, K = 8*), It has 3 memory registers, 2 input bits and 3 output bits.

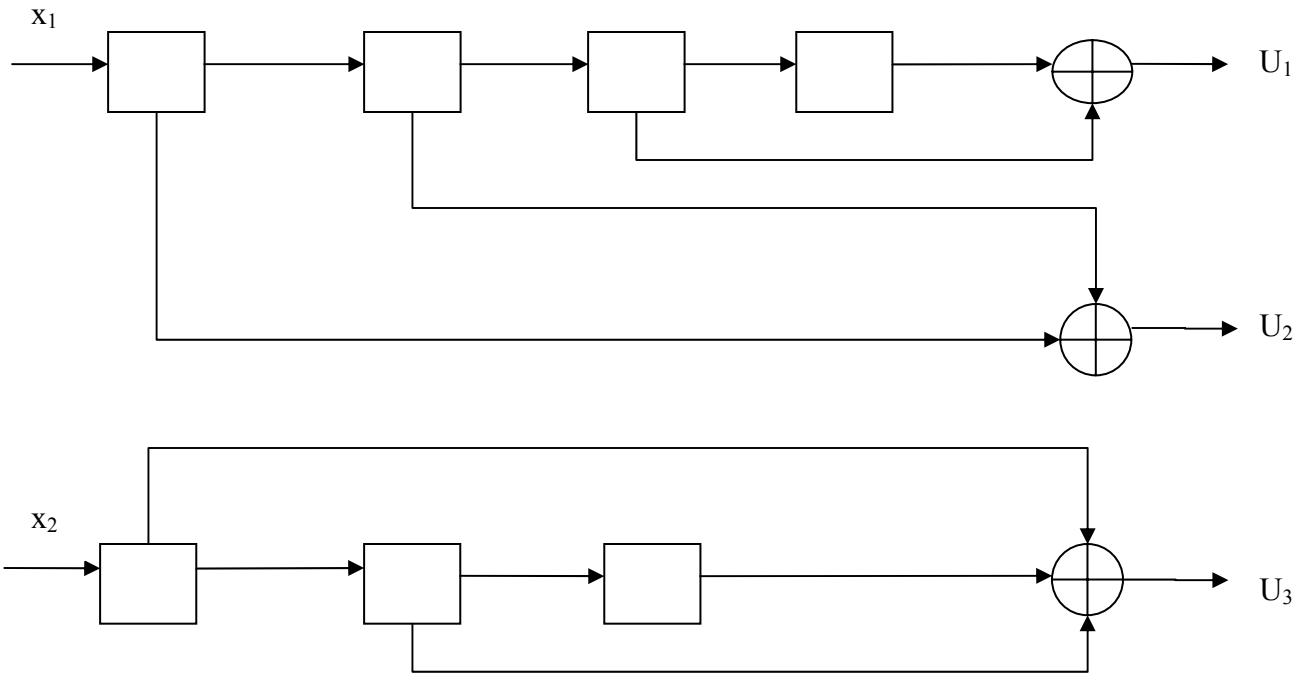

Figure 4.2: A (rate 2/3, $K = 8$) convolutional code.

4.3 Systematic vs. Non-systematic Convolutional Code

A systematic convolutional code is one in which the input *k*-tuple appears as part of the output branch word *n*-tuple associated with that *k*-tuple. In a systematic convolutional code the output bits contain an easily recognizable sequence of the input bits. The systematic version of the above (*rate 1/2, K = 3*) code of Figure 4.1 is shown in Figure 4.3, It has the same number of memory registers, and one input bit and two output bits. The output bits consist of the original two bits and a third parity bit. Looking at the code we see that of the two output bits, one is exactly the same as the one input bit. The second

bit is similar to a parity bit produced from a combination of three bits using a single polynomial.

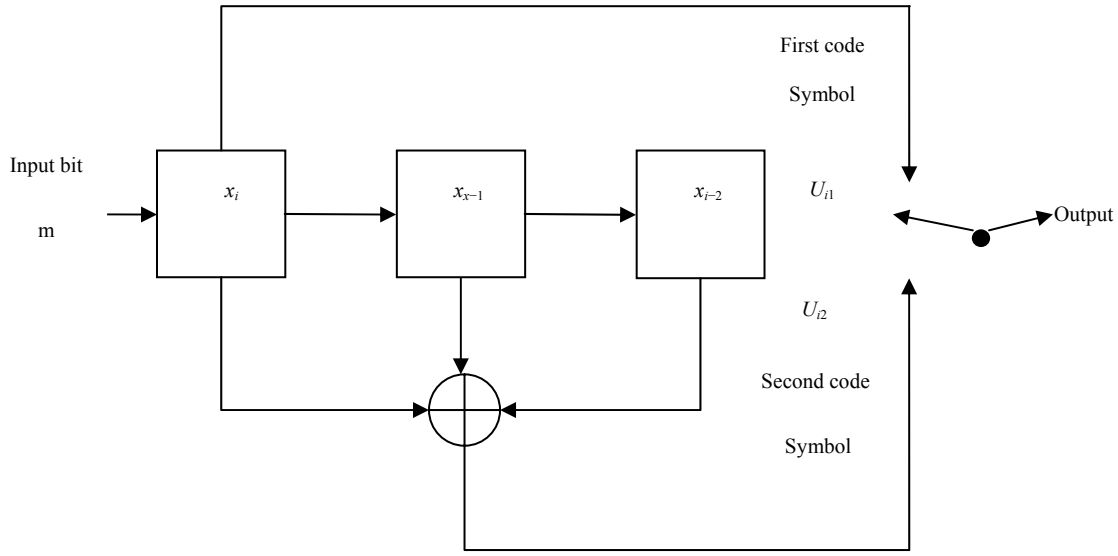

Figure 4.3: The systematic version of the (rate 1/2, $K = 3$) convolutional code.

Systematic codes are often preferred over the non-systematic codes because they allow quick look. Another important property of systematic codes is that they are non catastrophic, which means that errors can not propagate catastrophically. All these properties make them very desirable. But transforming a non-systematic convolutional code into a systematic code reduces the maximum possible free distance for a given constraint length and rate. However the error protection properties of systematic codes are the same as those non-systematic codes. In this thesis, we considered non-systematic codes.

4.4 Encoder Design

The hardware of the encoder is much simpler than that of the decoder because the encoder does no math. The encoder for convolutional code uses a table look up to do the encoding. The look up table consists of four items:

- Input bit.
- The State of the encoder, which is one of the 4 possible states for the example (*rate 1/2, K = 3*) code.
- The output bits. For the code (*rate 1/2, K = 3*), since two bits are output, the choices are 00,01,10,11.
- The output state which will be the input state for the next bit.

For the code (*rate 1/2, K = 3*) the look-up table is shown below in table I.

TABLE I: LOOK-UP TABLE FOR THE ENCODER OF CODE (*RATE 1/2, K = 3*)

I_1	S_1	S_2	O_1	O_2	S_1	S_2
1	0	0	0	0	0	0
1	0	0	1	1	1	0
0	0	0	1	1	0	0
1	0	0	0	0	1	0
0	0	1	1	0	0	0
0	0	1	0	1	1	0

This look up table uniquely describes the code (*rate 1/2, K = 3*). It is different for each code depending on the parameters and the polynomials used.

Flushing Bits

Now if we are only going to send the data bits given above, in order for the last bit to affect three ($K = 3$) pairs of output symbols, we need to output two more pairs of symbols. This is accomplished in our example encoder by clocking the convolutional encoder two (M) more times, while holding the input at zero. This is called flushing the encoder, and results in two more pairs of output symbols. If we don't perform the flushing operation, the last M bits of the message have less error-correction capability than the first through (M-1) bits had. This is a pretty important thing to remember if you're going to use this Forward Error Correction technique in a burst-mode environment. So there should be a step of clearing the shift register at the beginning of each burst. The encoder must start in a known state and end in a known state for the decoder to be able to reconstruct the input data sequence properly.

4.5 Encoder Representation

The encoder can be represented in several different but equivalent ways. They are

- Generator Representation
- State Diagram Representation
- Tree Diagram Representation
- Trellis Diagram Representation

4.5.1 Generator Representation

Generator representation shows the hardware connection of the shift register taps to the modulo-2 adders. A generator vector represents the position of the taps for an output. A "1" represents a connection and a "0" represents no connection. For example, the two generator vectors for the encoder in Figure 1 are g_1 = [111] and g_2 = [101] where the subscripts 1 and 2 denote the corresponding output terminals.

4.5.2 State Diagram Representation

The state of a rate 1/n convolutional encoder is defined as the contents of the rightmost K – 1 stages. In order to determine the next output it is sufficient to know the

present state and the next input. The state diagram shows the state information of a convolutional encoder. The state information of a convolutional encoder is stored in the shift registers. In the state diagram, the state information of the encoder is shown in the rectangular boxes. Each new input information bit causes a transition from one state to another. The path information between the states, denoted as *x/U*, represents input information bit *x* and output encoded bits *U*. It is customary to begin convolutional encoding from the all zero state. The state diagram for the convolutional encoder of Figure 4.1 is shown is Figure 4.4.

For example, the input information sequence x= {1101} (begin from the all zero state) leads to the state transition sequence S = {10, 11, 01, 10} and produces the output encoded sequence U = {11, 01, 01, 00}.

4.5.3 Tree Diagram Representation

Although the state diagram completely describes the encoder, it is not easy to track encoder transitions as a function of time since the diagram cannot represent time history. The tree diagram representation shows all possible information and encoded sequences for the convolutional encoder. It is somewhat better than a state diagram but still not the preferred approach for representing convolutional codes.

Here instead of jumping from one state to another, we go down branches of the tree depending on whether a 1 or 0 is received. Figure 4.5 shows the tree diagram for the encoder in Figure 4.1 for four input bit intervals.

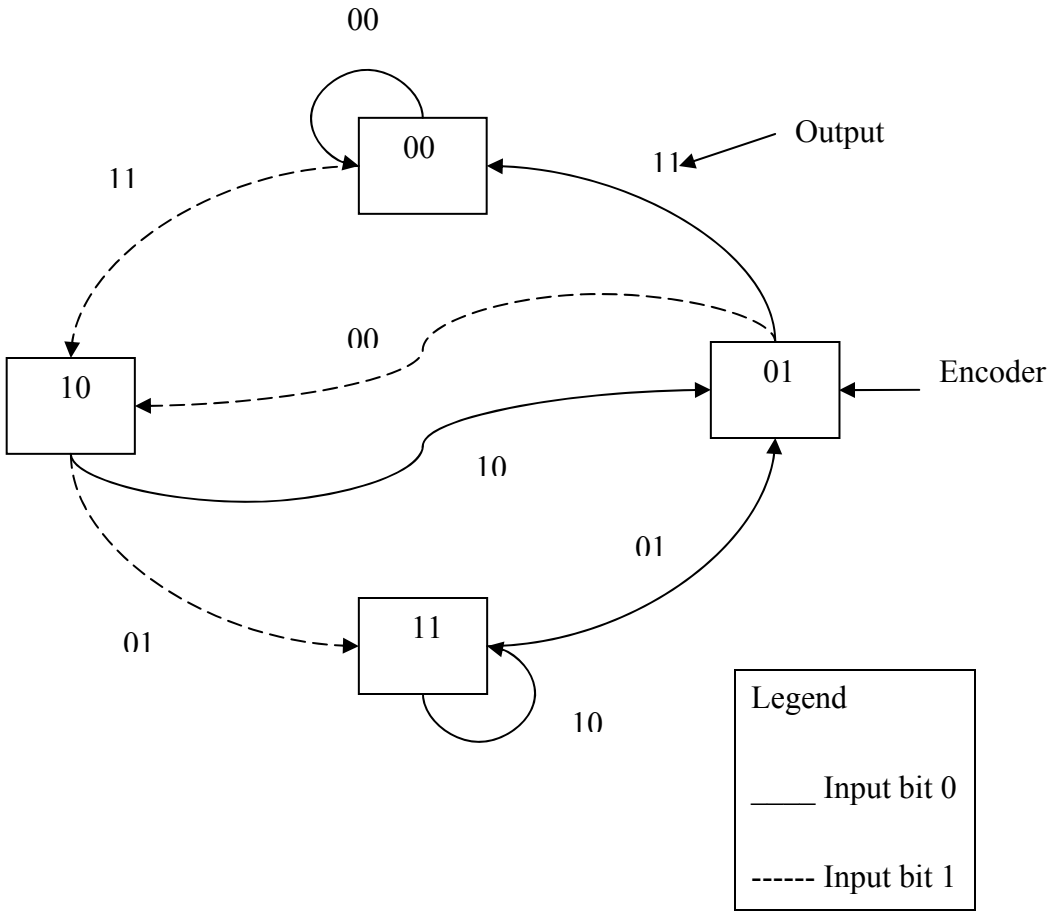

Figure 4.4: Encoder state diagram (rate 1/2, $K = 3$).

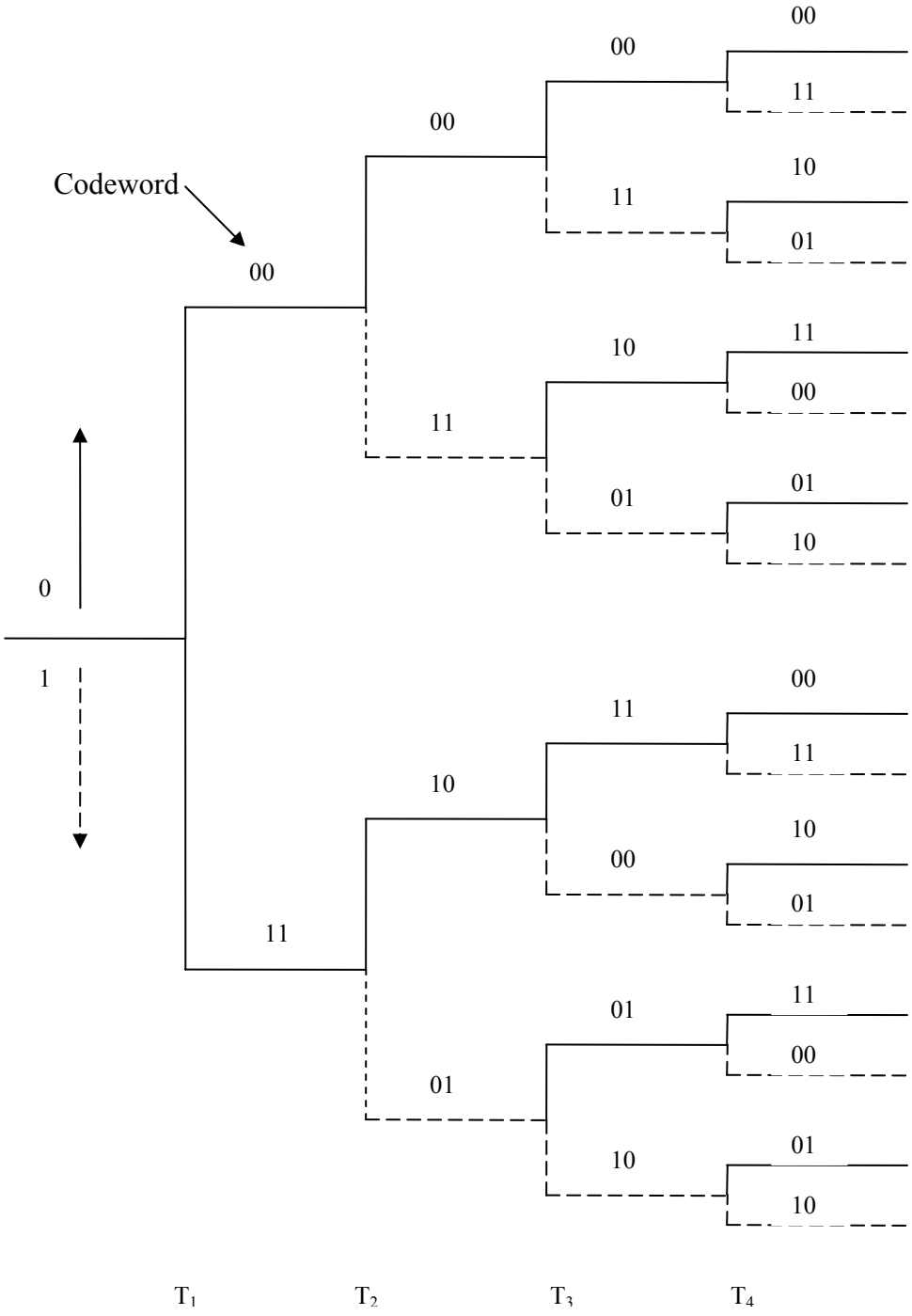

Figure 4.5: Tree representation of encoder (rate 1/2, $K = 3$)

In the tree diagram, a solid line represents input information bit 0 and a dashed line represents input information bit 1. The corresponding output encoded bits are shown on the branches of the tree. An input information sequence defines a specific path through the tree diagram from left to right. For example, the input information sequence x= {1101} produces the output encoded sequence U = {11, 01, 01, 00}. Each input information bit corresponds to branching either upward (for input information bit 0) or downward (for input information bit 1) at a tree node.

4.5.4 Trellis Diagram Representation

Trellis diagrams are generally preferred over both the tree and the state diagrams because they represent linear time sequencing of events. The x-axis is represented by discrete time and all possible states are shown on the y-axis. We move horizontally through the trellis with the passage of time. Each transition means new bits have arrived.

The trellis diagram is drawn by lining up all the possible states ($2K$-1) in the vertical axis. Then we connect each state to the next state by the allowable code words for that state. There are only two choices possible at each state. These are determined by the arrival of either a 0 or a 1 bit. The lines show the input bit and the output bits are shown on top of the line. The trellis diagram is unique to each code, same as both the state and tree diagrams are. We can draw the trellis for as many periods as we want. Each period repeats the possible transitions.

We always begin at state 00. Starting from here, the trellis expands and in K bits becomes fully populated such that all transitions are possible. The transitions then repeat

from this point on. This means that any two nodes having the same state label, at the same time T_i, can be merged since all the succeeding paths will be indistinguishable. The trellis diagram provides a more manageable encoder description than the tree diagram as it exploits the repetitive structure of the encoder. For this particular reason trellis representation is used while decoding a particular sequence. The trellis diagram for the convolutional encoder of Figure 4.1 is shown is Figure 4.6.

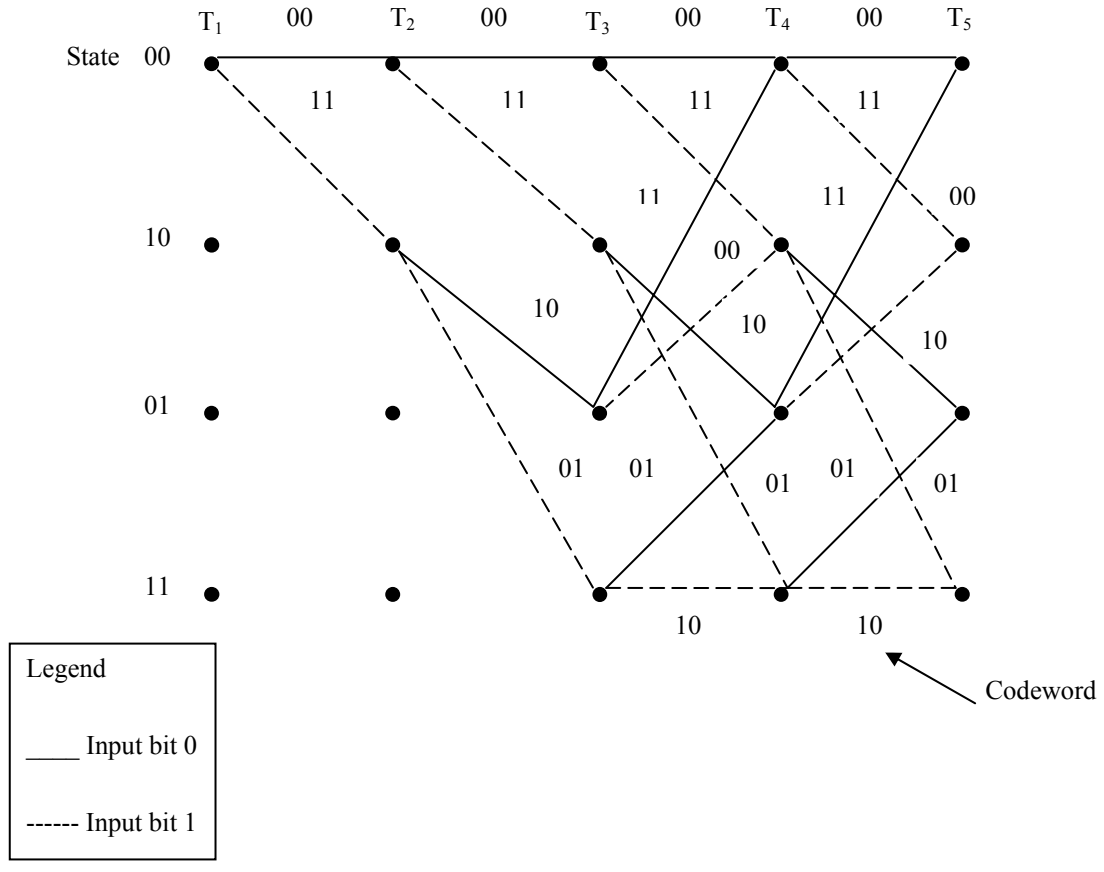

Figure 4.6: Encoder trellis diagram (rate 1/2, $K = 3$).

Encoding using the trellis diagram

In drawing trellis diagram, we use the same convention that we used with the state diagram that a solid line represents the output generated by an input bit, 0, and a dashed line represents the output generated by an input bit, 1. We can only start at point 1. At each unit of time the trellis requires 2^{K-1} nodes to represent 2^{K-1} possible encoder states. The trellis in our example assumes a fixed periodic structure after trellis depth 3 is reached. The path taken by the bits of the example sequence (1101) is shown by the lines shown is Figure 4.7. The corresponding output branch words are shown as labels on the trellis branches. We see that the trellis diagram gives exactly the same output sequence as the other three methods, which are graphical, state and the tree diagram. Though all of these diagrams look similar, we should recognize that they are unique to each code.

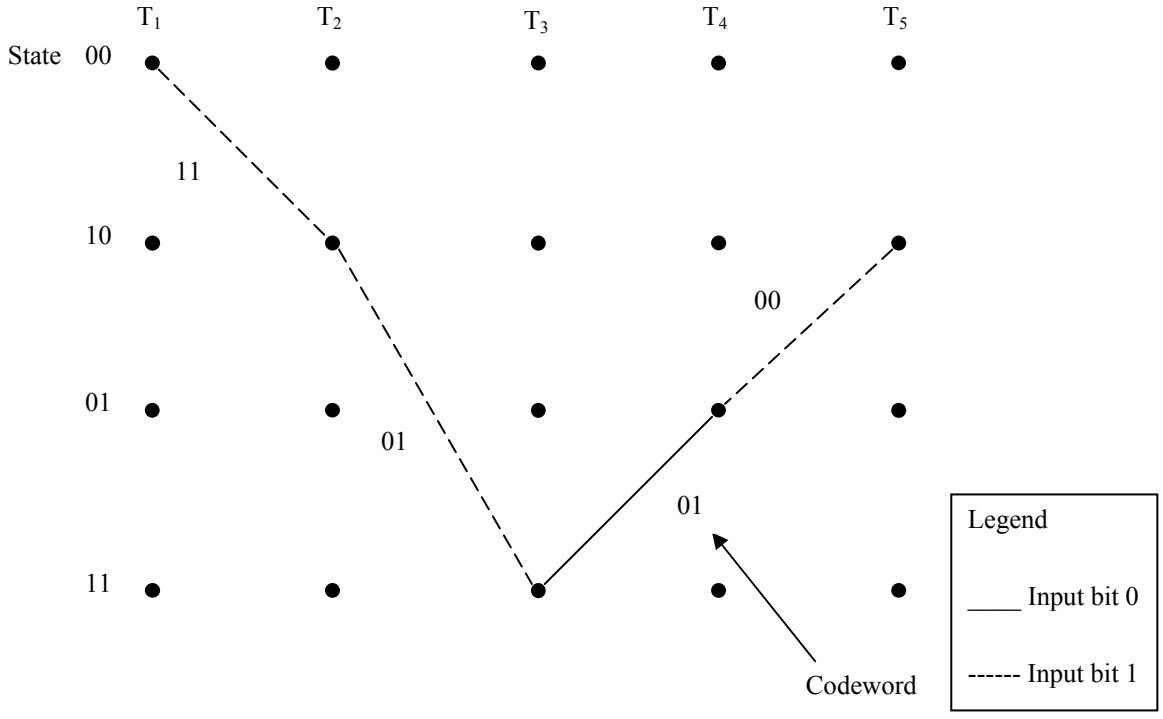

Figure 4.7: Trellis Diagram, Input sequence (1101), Output sequence (11, 01, 01, 00).

4.6 Decoding

There are several different approaches to decoding of convolutional codes. These are grouped in two basic categories.

a) Sequential Decoding --- Fano algorithm.

b) Maximum likelyhood decoding --- Viterbi decoding.

These methods represent two different approaches to the same basic idea behind decoding. In this thesis, we use Viterbi decoding to decode the convolutional encoded sequence.

Basic Decoding Principle

Let's assume that four bits were sent via a rate *1/2* code and we receive eight bits. Now the received eight bits may or may not have errors. However we know from the encoding process that all these bits map uniquely. So a 4-bit sequence will have a unique 8-bit output. But due to errors, we can receive any and all possible combinations of the eight bits.

The permutation of four input bits results in sixteen possible input sequences. Each of these has a unique mapping to an eight bit output sequence by the code. These form the set of permissible sequences and the decoder's task is to determine which one was sent.

Let's say we received 11010101. In order to decode the received sequence we can use one of the following two methods,

- We can compare this received sequence to all permissible sequences and pick the one with the smallest Hamming distance (or bit disagreement).
- We can do a correlation and pick the sequences with the best correlation.

The first method is basically what is behind hard decision decoding and the second is the soft-decision decoding. The bit disagreements show that we still get an ambiguous answer and do not know what was sent. As the number of bits increase, the number of calculations required to do decoding in this brute force manner increases such that it is no longer practical to do decoding this way. We need to find a more efficient method that does not examine all options and has a way of resolving ambiguity such as here where we have two possible answers.

If a message of length q bits is received, then the number of total possible codewords is 2^q. The basic idea behind decoding is to decode the received sequence without checking each and everyone of these 2^q codewords.

4.6.1 Sequential Decoding

Sequential decoding was one of the first methods proposed for decoding a convolutionally encoded bit stream. Sequential decoding has the advantage that it can perform very well with long constraint length convolutional codes, but it has a variable decoding time.

In sequential decoding you are dealing with just one path at a time. You may give up that path at any time and turn back to follow another path but the important thing is

that only one path is followed at any one time. This method also allows both forward and backward movement through the trellis. The decoder keeps track of its decisions each time it makes an ambiguous decision. But if the tally increases faster than some threshold value, decoder gives up that path and retraces the path back to the last fork where the tally was below the threshold. Since Viterbi decoding is used in this thesis, we will not discuss sequential decoding in depth.

4.6.2 The Viterbi Convolutional Decoding Algorithm

Viterbi decoding as described in [10] by Andrew J Viterbi is the best known implementation of maximum likelihood decoding. It reduces the computational load by taking advantage of the special structure in the encoder trellis. In this thesis we have chosen $5 \times K$ as the decoding depth of the Viterbi decoder as described in [7], and also research has shown that a decoding depth of $5 \times K$ is sufficient for Viterbi decoding with the type of codes used during simulation. Any deeper traceback increases decoding delay and decoder memory requirements, while not significantly improving the performance of the decoder. Viterbi decoding has the advantage that it has a fixed decoding time. It is well suited to hardware decoder implementation. The advantage of Viterbi decoding over other decoding is that the complexity of a Viterbi decoder is not a function of the number of symbols in the codeword sequence.

First the Viterbi decoder examines an entire received sequence of a given length. The decoder computes a metric for each path and makes a decision based on this metric. The Viterbi algorithm removes from consideration those trellis paths that could not

possibly be candidates for the maximum likelihood choice. All paths are followed until two paths converge on one node. Then the path with the higher metric is kept and the one with lower metric is discarded, this path is called the surviving path. This selection of surviving paths is performed for all the states. Following this pattern, the decoder advances deeper into the trellis, making decisions by eliminating the least likely paths.

For a q-bit sequence, the total number of possible received sequences are 2^q, however, of these only 2^K are valid. The Viterbi algorithm applies the maximum-likelihood principles to limit the comparison to 2 to the power of K surviving paths instead of checking all paths. The early rejection of the unlikely paths reduces the decoding complexity.

The most common metric used is the Hamming distance metric. This is just the degree of similarity between the received codeword and the allowable codeword. All these metrics are added together so that the path with the smallest total metric is the correct sequence. Table II describes the hamming metric for a (rate 1/2, $K = 3$) code.

TABLE II: EACH BRANCH HAS A HAMMING METRIC DEPENDING ON WHAT WAS RECEIVED AND THE VALID CODEWORDS AT THAT STATE

Received Bits	**Valid Codeword 1**	**Valid Codeword 2**	**Hamming Metric 1**	**Hamming Metric 2**
00	00	11	0	2
10	00	11	1	1
01	10	01	2	0

4.6.3 An Example of Viterbi Convolutional Decoding

In this example we consider Hamming distance for finding the path metric. The encoder in this example is shown in Figure 4.1 and the encoder trellis diagram is shown in Figure 4.6. A similar trellis can be used to represent the decoder, which is shown in Figure 4.8. The principle idea behind decoding procedure can be best understood by comparing the Figure 4.6 encoder trellis with the Figure 4.8 decoder trellis. In the decoder trellis each branch at T_i is labeled with the hamming distance between the received code symbols and the corresponding branch codeword from the encoder trellis. The example in Figure 4.8 shows a message sequence, x, the corresponding codeword sequence, U, and a noise corrupted received sequence R. The branch words seen on the encoder trellis branches are for the encoder in Figure 4.1. These branch words are known to the encoder and the decoder prior to transmission. As the code symbols are received, each branch of the decoded trellis is labeled with a metric of similarity (Hamming distance) between the received code symbols and each of the branch words at that time interval. To label the decoder branches at time T_1 with the appropriate hamming distance metric, we look at the encoder trellis in Figure 4.6. Here we see that a state 00 to 00 transition yields an output branch word of 00 and we received 11, therefore on the decoder trellis for state 00 to 00 transitions we get the branch metric 2. Similarly for 00 to 01 transition, we get the branch metric as 0 and continuing this approach we can fill the decoder trellis with the corresponding branch metrics. The decoding algorithm will use these Hamming distance metrics in order to find the minimum distance path through the trellis.

The principle behind Viterbi decoding is that of two paths merging to a single state in the trellis, one of them can always be eliminated by deciding the optimum path among them. The cumulative Hamming path metric of a given path at time T_i is defined as the sum of the branch Hamming distance metrics along the path up to time T_i.

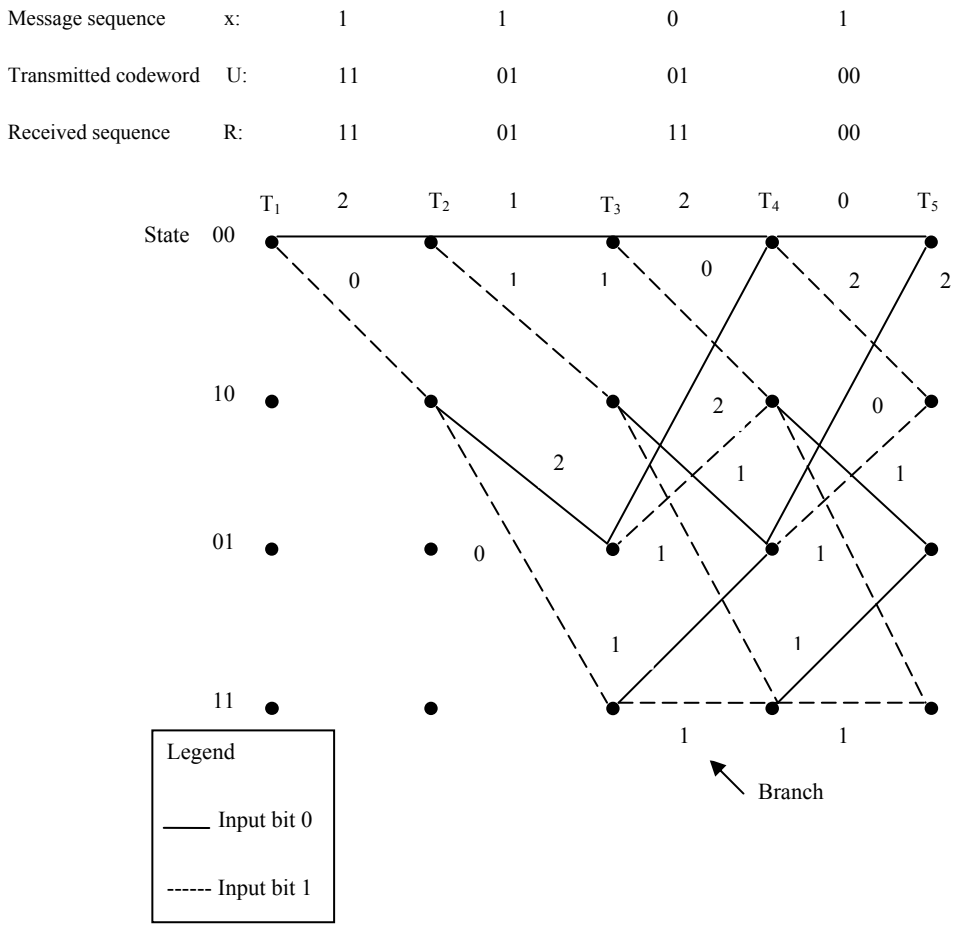

Figure 4.8: Decoder trellis diagram (rate 1/2, K = 3).

When two paths try to merge at the same node then the one with smallest path metric is chosen as the optimum path. At any time T_i there are 2^{K-1} states in the trellis and each state can be entered by means of two paths. Viterbi decoding consists of computing the metrics for the two paths entering each state and eliminating one of them. The decoder

does this computation for each of the 2^{K-1} nodes at time T_i. Then the decoder moves to time T_{i+1} and repeats the same process. With Viterbi decoder the first bit is not decoded until the path metric computation has proceeded to a much greater depth into the trellis. This leads to a decoding delay which can be as much as five times the constraint length in bits for a particular decoder.

To illustrate the algorithm, let us decode a received sequence 11 10 11 00 using the Viterbi algorithm.

a) At time T_1, we have received bit 11. The decoder always starts at state 00. From this point it has two paths available, either 00 or 10. The decoder computes the branch metric for both of these and will continue simultaneously along both of these branches in contrast to the sequential decoding where a choice is made at every decision point. The branch metric for state 00 to 00 is 2 and for state 00 to 10 is 0 as shown in Figure 4.9a.

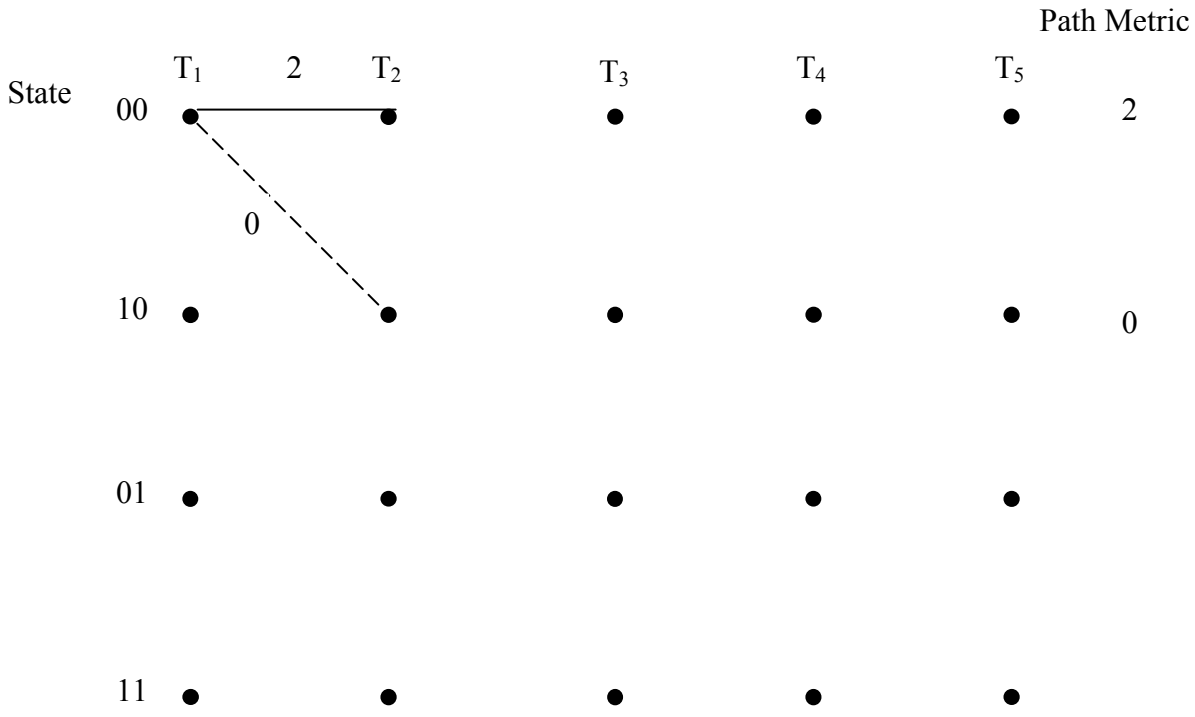

Figure 4.9a: Survivors at T_1

b) At time T_2 as shown in Figure 4.9b, the decoder fans out from these two possible states to four states. The branch metrics for these branches are computed by looking at the agreement with the codeword and the incoming bits which are 10. The new path metric is shown on the right of the trellis.

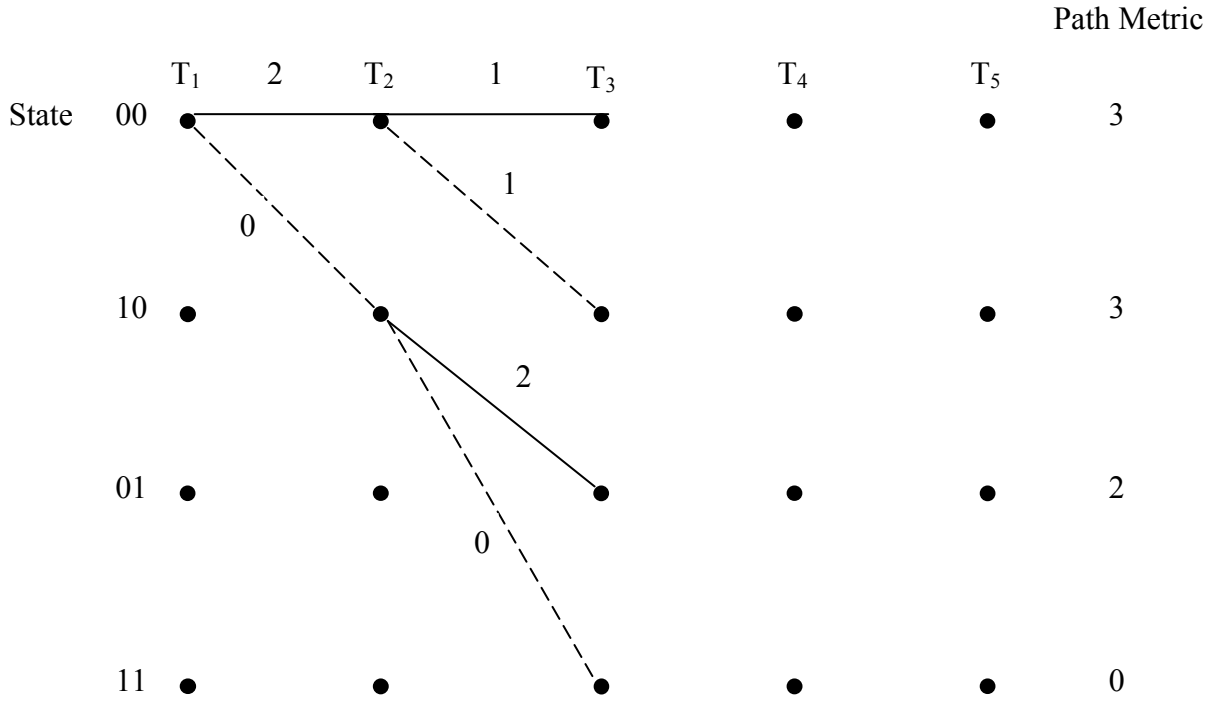

Figure 4.9b: Survivors at T_2

c) At time T_3 as shown in Figure 4.9c, there are again two branches diverging from each state. As a result there are two paths entering each state at T_4. The path metrics are calculated for bits 01 and added to pervious metrics from T_2.

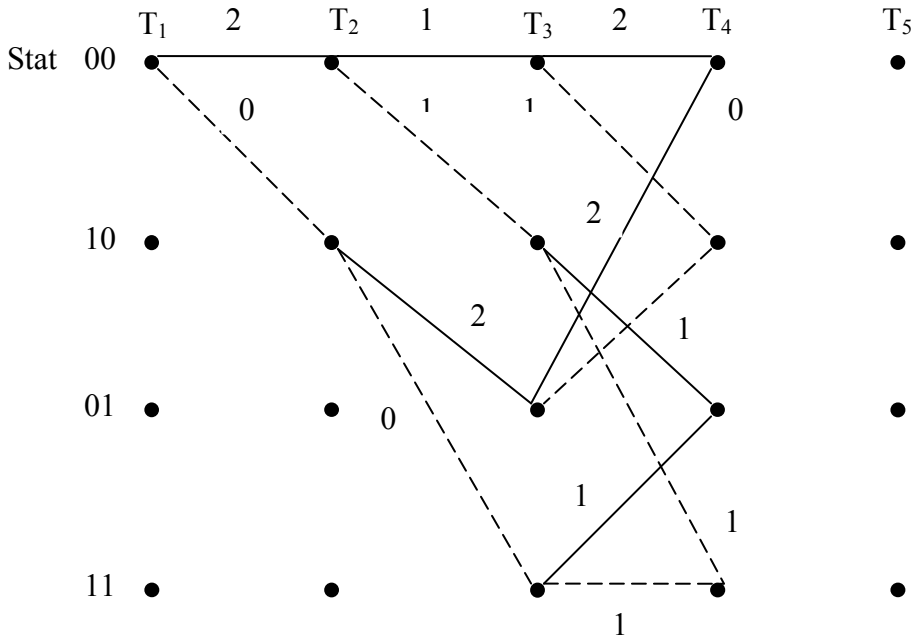

Figure 4.9c: Survivors at T_3

As noted previously, one path entering each state can be eliminated by choosing the one having the larger cumulative path metric. If the metrics of the two entering paths are equal then one path is chosen for elimination by arbitrary rule. The surviving paths at this stage are shown in Figure 4.9d. At this point in the decoding process there is only a single surviving path between T_1 and T_2. Therefore, the decoder can now decide that the state transition which occurred between T_1 and T_2 was 00 to 10. Since this transition is produced by input bit 1, the decoder gives out 1 as the first decoded bit.

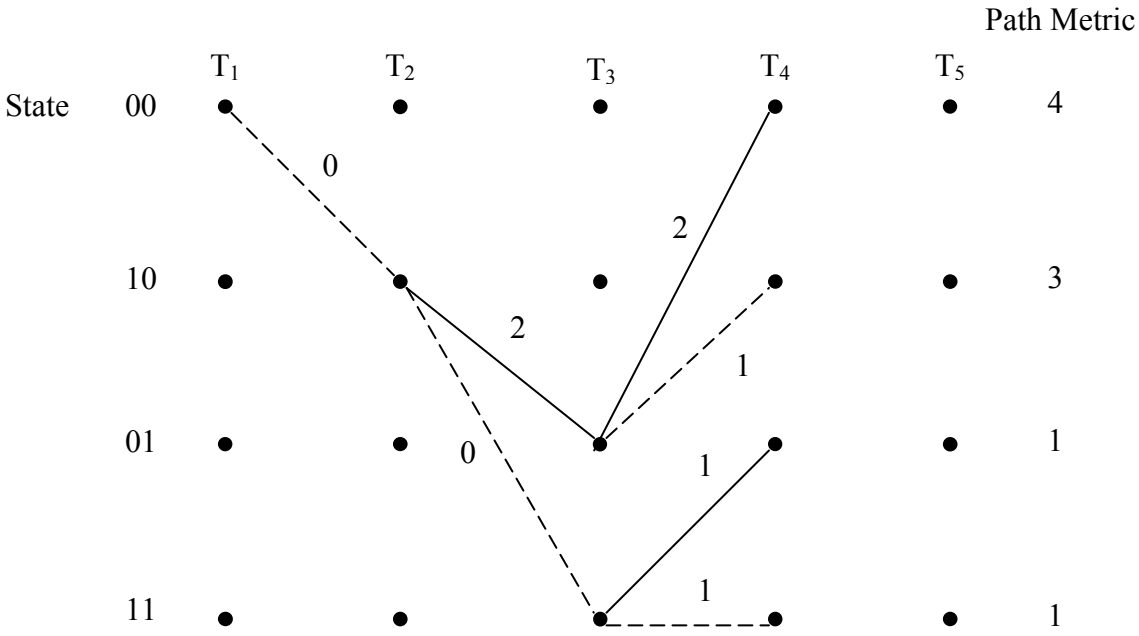

Figure 4.9d: Path Deciding at T_3

d) At time T_4 as shown in Figure 4.9e, the received bits are 00. Again the metrics are computed for all paths. We discard all larger metrics but keep both if they are equal. Figure 4.9e shows the survivors at time T_4.

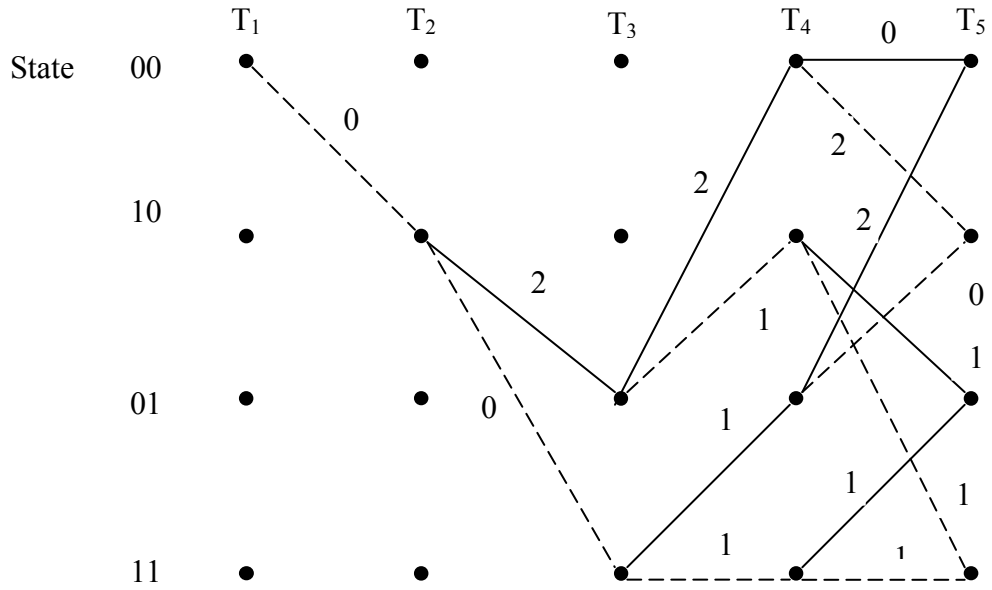

Figure 4.9e: Survivors at T_4.

The 4-step trellis is complete. We now look at the path with the lowest metric. We have a winner and it is the one which has a path metric 1 as shown in Figure 4.9f. The path traced by states 10, 11, 01, 10 and corresponding to bits 1101 is the decoded sequence. The decoded path is indicated by the dark line in Figure 4.9f.

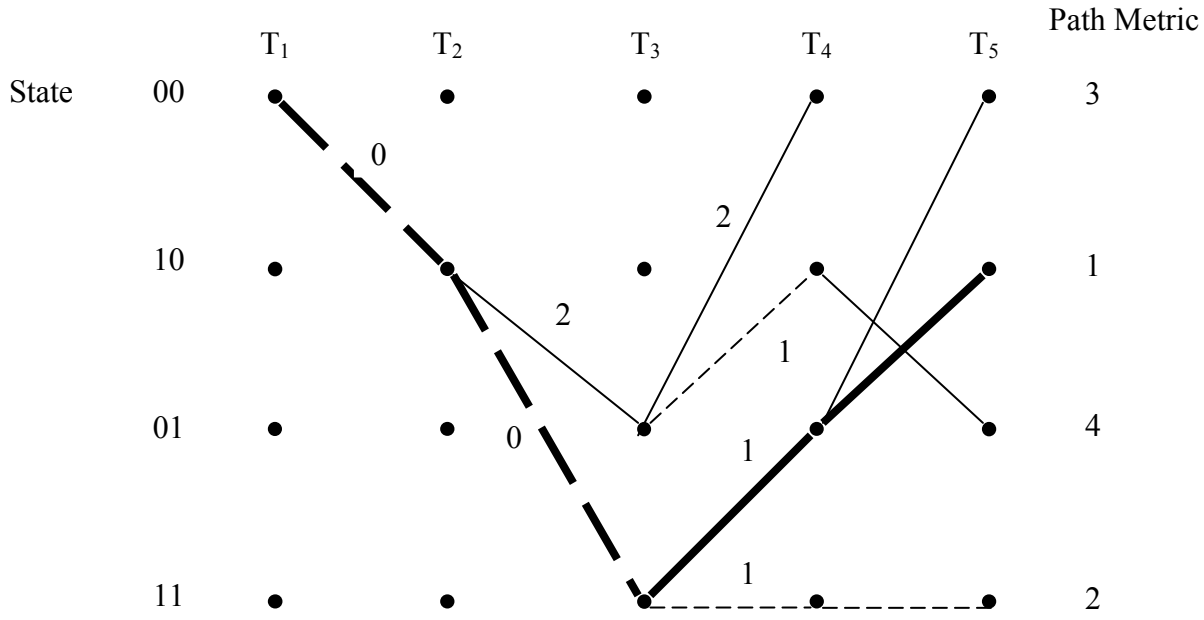

Figure 4.9f: Final Path at T_4.

4.7 Hard and Soft-Quantization

An ideal Viterbi decoder would work with infinite precision, or at least with floating-point numbers. In practical systems, we quantize the received channel symbols with one or a few bits of precision in order to reduce the complexity of the Viterbi decoder. If the received channel symbols are quantized to one-bit precision ($< 0V = 1, \geq 0V = 0$), the result is called hard-decision data. If the received channel symbols are quantized with more than one bit of precision, the result is called soft-decision data. A Viterbi decoder with soft decision data inputs quantized to three or four bits of precision

can perform about 2 dB better than one working with hard-decision inputs. The usual quantization precision is three bits. More bits provide little additional improvement.

In this thesis, hard decision decoding is used since our aim is only to compare the performance of ordinary and generalized DSSS.

CHAPTER V

SIMULATION OF COMMUNICATION SYSTEM MODEL

Simulation of a convolutional coded DSSS communication system involves a significant amount of computation and it is therefore very time consuming. This chapter describes the constraints involved in simulation and also talks about Monte Carlo simulation, data generator, convolutional encoder, convolutional interleaver, PN generator and Viterbi decoder.

C platform was used to simulate the communication system and the data collected was plotted for analysis. The graphs shown in Chapter 6 are the graphs obtained from simulation results for different constraint lengths and code rates.

5.1 Monte-Carlo Simulation for BER Measurement

For any communication system bit error rate (BER) gives a good measure of its performance. There are two methods to measure BER. First method is called error

counting, which counts a pre-determined number of symbols and trials, then makes a note of the total number of errors incurred. The second is called symbol counting, which counts a pre-determined number of errors, and then makes a note of the total number of symbols required to produce these errors. In this thesis we use error counting for all the simulations.

The signal and noise sources used in the communication system are random in nature and therefore the results obtained in terms of BER are also random. Usually the errors on the measurement of BER arise because it is impossible to have infinite number of trials to make the estimate. When the probability of bit error is calculated by taking the ratio of the number of errors to a given number of trials, then the measurement could be in error because of the fact that underlying probability of error is small compared to the infinite number of trials.

In order to obtain the desired "true" result we have to perform "infinite number" of simulation runs, which is impossible to achieve in reality. Instead, the reliability of the measurement could be determined, i.e. one can be confident in a quantifiable way that the resulting BER is a good representation of the "true" result. To know more about the theory involved with the reliability of BER measurements refer to [11]. In order to avoid confusion it is always better to express BER in terms of a degree of confidence. Thus the results from a Monte Carlo simulation should always be quoted in the form of confidence intervals with an associated probability. For example: "there is a 96% probability that the actual BER is between 0.003 and 0.0035", where 96% is known as the confidence level, and the difference between 0.003 and 0.0035 is the confidence interval. The confidence interval decreases as the number of trials increases by keeping the confidence level

constant, whereas the confidence level increases with the increase in number of trials by keeping the confidence interval constant. A simple formula is used to determine the confidence limits for a given confidence level. Let N be the number of trials and n be the number of errors. Let p be the estimated bit error rate. Then, from [11]

$$P[x_+ \le p \le x_-] = 1 - \beta \tag{5.1}$$

where

$$x_\pm = \frac{n}{N}\left[1 + \frac{d_\beta{}^2}{2n}\left(1 \pm \sqrt{\frac{4n}{d_\beta{}^2} + 1}\right)\right] \tag{5.2}$$

and d_β is given by

$$\frac{1}{\sqrt{2\pi}} \int_{-d_\beta}^{d_\beta} e^{-t^2/2} dt = 1 - \beta \tag{5.3}$$

Now for a confidence level of Y%, we can set

$$\beta = 1 - \frac{Y}{100} \tag{5.4}$$

Using the above given formula the necessary length of the simulation runs can be determined. We consider $\beta = 0.05$ resulting in $d_\beta = 1.96$ in this thesis. However for many practical measurements, "the rule of thumb" that counting 10 errors gives a BER within a factor of 2 with 95% confidence is used. In order to obtain a smaller spread in BER we need to count more errors. We consider 10000 trials for all the simulations presented in this thesis.

5.2 Generating the Message Data

Generating the data to be transmitted through the channel can be accomplished quite simply by using a random number generator. C language provides rand() function which produces a uniform distribution of numbers on the interval from 0 to a maximum value. Here, we use rand() function to generate random numbers as described in [12]. Using this function, we can say that any value less than half of the maximum value is a zero; any value greater than or equal to half of the maximum value is a one.

5.3 Simulation of Convolutional Encoder

The encoder is simulated by first filling the two tables. They are the next state and output symbol for the convolutional encoder. A detailed description of convolutional encoder functionality is described in Chapter 4. The table III given below is often called a state transition table. We will refer to it as the *next state* table.

TABLE III: NEXT STATE TABLE

	Next State, if	
Current State	*Input = 0:*	*Input = 1:*
00	00	10
01	00	10
10	01	11
11	01	11

Now let us look at table IV given below that lists the channel output symbols, given the current state and the input data, we will refer to as the *output* table:

TABLE IV: OUTPUT TABLE

	Output Symbols, if	
Current State	*Input = 0:*	*Input = 1:*
00	00	11
01	11	00
10	10	01
11	01	10

You should now see that with these two tables, you can completely describe the behavior of the example (rate 1/2, $K = 3$) convolutional encoder. Note that both of these tables have $2^{(K - 1)}$ rows and 2^k columns, where K is the constraint length and k is the number of bits input to the encoder for each cycle. These two tables will come in handy when we start discussing the Viterbi decoder algorithm.

In a convolutional encoder, generator polynomial represents the shift register connections to the modulo-two adders. They are usually denoted as $g_i = [101]$ where the subscript *i=1, 2, 3, etc.* denote the corresponding output terminals. A "1" represents a connection and a "0" represents no connection. Choice of generator polynomials does influence the performance of convolutional codes. We considered the generator polynomials listed in [5]. Table V describes all the generator polynomials used in this thesis for different codes.

TABLE V: GENERATOR POLYNOMIALS OF VARIOUS CODES

Code Rate (r)	**Constraint Length (*K*)**	g_1	g_2	g_3	g_4	g_5	g_6	g_7
1/2	3	111	101	-	-	-	-	-
1/2	5	11111	10101	-	-	-	-	-
1/2	7	1111111	1101101	-	-	-	-	-
1/3	3	111	101	110	-	-	-	-
1/5	3	111	101	110	011	010	-	-
1/7	3	111	101	110	011	010	111	101

Mapping the one/zero output of the convolutional encoder onto an antipodal baseband signaling scheme is simply a matter of translating zeroes to +1s and ones to -1s. This can be accomplished by performing the operation $y = 1 - 2x$ on each convolutional encoder output symbol.

5.4 Pseudonoise Sequence and Energy Normalization

The spreading sequence used in the simulation of coded generalized DSSS is different from the ordinary sequence. The sequence used is {–1, 0, +1} instead of {–1, +1} and it is generated using a random number generator with some arbitrary seed. The random number generators used for this purpose in this thesis are ran0( ) and ran2( ) described in [12]. They produce uniform deviates which are random numbers which lie within a specific range from 0 to 100, each number with equal probability of occurring.

Now the generated number is divided by 3 and if the remainder is equal to 0 we consider 1 to be generated, if the remainder is 1 we consider 0 to be generated and if the remainder is 2 then we consider -1 to be generated. For each symbol we calculate total number of -1, 0 and +1 chips generated.

In order to account for the energy lost due to the extra 0-valued chip in the generalized sequence we need to normalize the PN generated chips before they are transmitted across the noisy channel. The normalization factor will consolidate the energy lost due to the "0" chip in the {1, 0, +1} sequence. At the receiver end the same normalized PN sequence is multiplied with the received noisy chips before hard decision is performed. The normalization factor is given by $c = \sqrt{\frac{NL}{u}}$, where u represents the total number of non-zero chips per L encoded symbols in the PN sequence.

5.5 Simulation of Channel

We consider a channel model in which nothing is known about the interference except that it is bounded in power. Furthermore, it is also independent of the transmitted signal and thermal noise. In order to avoid the overhead of interleaving and deinterleaving, we directly interleave the interference in the channel while adding it to the signal which is described in Figure 5.1. Our measure of reliability is the worst-case error probability over all such unknown signals. In order to simulate the worst-case performance, first we distribute the interference over D chips out of total NL chips using

convolutional or pseudorandom interleaving and later we maximize the simulated BER for any given D over all values of D to get the worst-case performance. We use chip-level interleaving in this thesis as it performs better than symbol-level interleaving as described in [3].

Adding noise to the transmitted channel symbols produced by the convolutional encoder involves generating Gaussian random numbers, scaling the numbers according to the desired energy per chip to noise density ratio and adding the scaled Gaussian random numbers to the channel symbol values. However, in this thesis we did not add Gaussian noise to better emphasize the performance due to the presence of arbitrary interference.

5.6 Integration and Hard Quantization

At the receiver end, the same normalized PN code sequence used at the transmitter end is multiplied with the received noisy chips before any hard decision is performed over the received signal. A waveform channel would require an integrator for the correlation operation whereas the vector representation, which we are using for simulation purposes, requires only vector correlations i.e. chip by chip multiplication. After the multiplication the chips are converted to symbols and then fed to decision box. In our case the decision box will compare the input signal to zero threshold and output +1 if it is greater than zero and –1 if it is less than zero. Once the symbols are out of the decision box, they are fed to Viterbi decoder to get back the original message bits.

5.7 Simulation of Viterbi Decoder

A detailed description of the Viterbi decoder functionality is given in Chapter 4. Here we talk about various steps involved in simulating it in software. In this thesis we have chosen $5 \times K$ as the decoding depth of the Viterbi decoder as described in [7], and also research has shown that a decoding depth of $5 \times K$ is sufficient for Viterbi decoding with the type of codes used in this thesis. Any deeper traceback increases decoding delay and decoder memory requirements, while not significantly improving the performance of the decoder. To implement a Viterbi decoder in software, the first step is to build some data structures around which the decoder algorithm will be implemented. These data structures are best implemented as arrays. The primary six arrays that we need for the Viterbi decoder are as follows:

- A copy of the convolutional encoder next state table, the state transition table of the encoder. The dimensions of this table (rows × columns) are $2^{(K-1)} \times 2^k$. This array needs to be initialized before starting the decoding process.
- A copy of the convolutional encoder output table. The dimensions of this table are $2^{(K-1)} \times 2^k$. This array needs to be initialized before starting the decoding process.
- An array or table showing for each convolutional encoder current state and next state, what input value (0 or 1) would produce the next state, given the current state. This array is called the input table. Its dimensions are

$2^{(K-1)} \times 2^{(K-1)}$. This array needs to be initialized before starting the decoding process.

- An array to store state predecessor history for each encoder state for up to $K \times 5 + 1$ received channel symbol pairs. We'll call this table the state history table. The dimensions of this array are $2^{(K-1)} \times (K \times 5 + 1)$. This array does not need to be initialized before starting the decoding process.
- An array to store the accumulated error metrics for each state computed using the add-compare-select operation. This array will be called the accumulated error metric array. The dimensions of this array are $2^{(K-1)} \times 2$. This array does not need to be initialized before starting the decoding process.
- An array to store a list of states determined during trace back. It is called the state sequence array. The dimensions of this array are $K \times 5 + 1$. This array does not need to be initialized before starting the decoding process.

5.8 Simulation of Coded Generalized DSSS Communication System

In order to avoid the overhead of interleaving and de-interleaving process as described in Figure 3.1, we directly interleave the interference in the channel while adding it to the signal which is described in Figure 5.1. The steps involved in simulating a spread spectrum communication system using convolutional coding and Viterbi decoding are as follows:

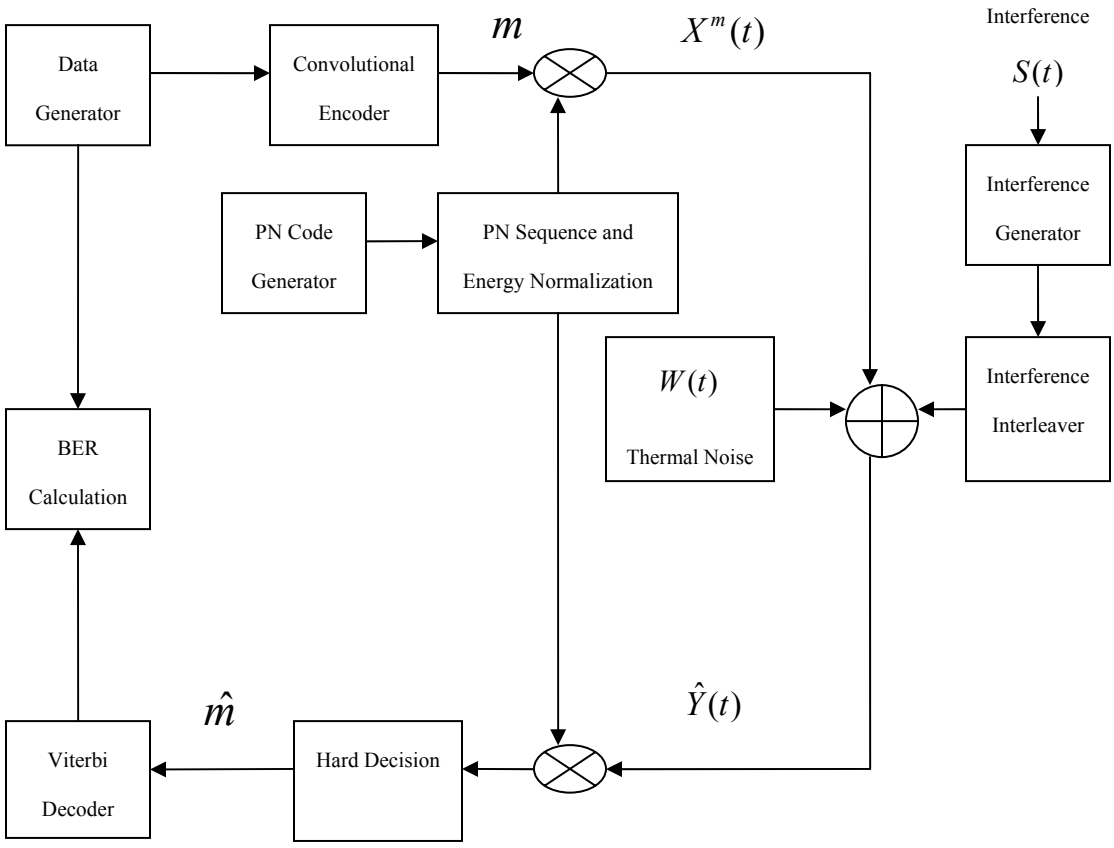

Figure 5.1: Simulated System Model

- Generate the data to be transmitted through the channel. In our case it is always in blocks of 5 × (constraint length) i.e. 5 × K binary data bits.
- Convolutionally encode the data and map the one/zero channel symbols onto an antipodal baseband signal, producing transmitted channel symbols i.e. $L = \frac{5 \times K}{r}$ symbols where r is the code rate of encoder.
- Multiply the baseband signals with normalized (for generalized DSSS) pseudonoise code sequence of length N. Here the symbols get converted into chips.
- Now distribute the interference over D chips out of total $N \times L$ chips using convolutional or pseudorandom interleaving, later the simulation result maximizes over all the values of D.
- Multiply the received noisy chips with the same generalized pseudonoise code sequence used at the transmitter and later perform 1-bit quantization of the received channel symbols which is termed as hard-decision; here chips are converted back to symbols.
- Perform Viterbi decoding on the quantized received channel symbols which results in binary data bits.
- Compare the decoded data bits to the transmitted data bits and count the number of errors in order to calculate the BER of the communication system for the given value of D.

- Maximize calculated BER over all values of D to find the worst-case BER.

In this thesis we accurately model the effects of interference even by bypassing the steps of modulating the channel symbols onto a transmitted carrier, and then demodulating the received carrier to recover the channel symbols. We choose this method because it avoids complexity and at the same time it will not affect the performance of the system.

CHAPTER VI

NUMERICAL RESULTS

In this thesis we consider convolutional codes with different constraint lengths and code rates to compare the worst-case error performance of coded ordinary and coded generalized DSSS schemes. Constraint lengths considered are K = 3, 5 and 7 and code rates considered are r = 1/2, 1/3, 1/5 and 1/7. Results are obtained by keeping one variable constant while varying the other. Conclusions are drawn from the observations.

In this chapter, plots showing "generalized" correspond to the coded generalized DSSS communication system and plots showing "ordinary" correspond to the ordinary DSSS. The E_b/N_i used in the plots is defined as follows:

$$E_b = \frac{5 \times K \times N}{r}$$

K = Constraint length

$5 \times K$ = Decoding depth

N = Chip length

r = Code rate of convolutional encoder

where E_b is defined signal energy. N_i is the interference considered for a bit set of $\frac{5 \times K \times N}{r}$ as described in section 3.1.2 of Chapter 3.

First we compare codes with the same constraint length and see how they perform as code rate varies. We compare the performance of various codes and compare the difference between the curves for ordinary DSSS and generalized DSSS for each code, keeping constraint length constant. This is followed by comparison of codes with the same code rate and different constraint lengths of the convolutional code. We compare the performance of different codes and compare the difference between the curves for ordinary DSSS and generalized DSSS for each code, keeping code rate constant. Then we compare codes with same code rate and same constraint length with varying lengths of the pseudorandom chip sequence (N). We compare the performance of various codes and compare the difference between the curves for ordinary DSSS and generalized DSSS. Then we compare codes with same code rate and same constraint length by varying the interleaver between convolutional and pseudorandom. We compare the performance of various codes and compare the difference between the curves for ordinary DSSS and generalized DSSS. Then we compare codes with same code rate and same constraint length with varying the decoding depth of the Viterbi decoder. We compare the performance of various codes and compare the difference between the curves for ordinary DSSS and generalized DSSS. Finally we compare codes with same code rate and same constraint length by varying the number of rows of convolutional interleaver.

6.1 Convolutional Codes with Same Constraint Length

In this section we compare the worst-case performance of convolutional codes with the same constraint length and see how they perform as code rate varies. All the codes in this section are used with a chip sequence length of $N = 10$, decoding depth of $5K$ and convolutional interleaving with the number of rows as 5.

The codes shown in Figures 6.1, 6.2, 6.3 and 6.4 have the same constraint length of $K = 3$. They have code rates of $r = 1/2, 1/3, 1/5$ and $1/7$ respectively. Comparing the performance at 10^{-3} level, we get Table VI.

TABLE VI: VARYING CODE RATE (R)

Code	Code Rate (r)	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.1	1/2	-0.5	0.5	1
Figure 6.2	1/3	-2	-0.6	1.4
Figure 6.3	1/5	-2.5	-1.0	1.5
Figure 6.4	1/7	-4	-2.4	1.6

From the results in this section it is observed that with constant constraint length of the convolutional code as code rate decreases performance improves, at the same time the difference between two systems increases. It is expected that decreasing code length (increasing redundancy) would result in better performance. It is interesting to also see that the difference between ordinary and generalized DSSS increases with decreasing code length, as stipulated by Hizlan [3].

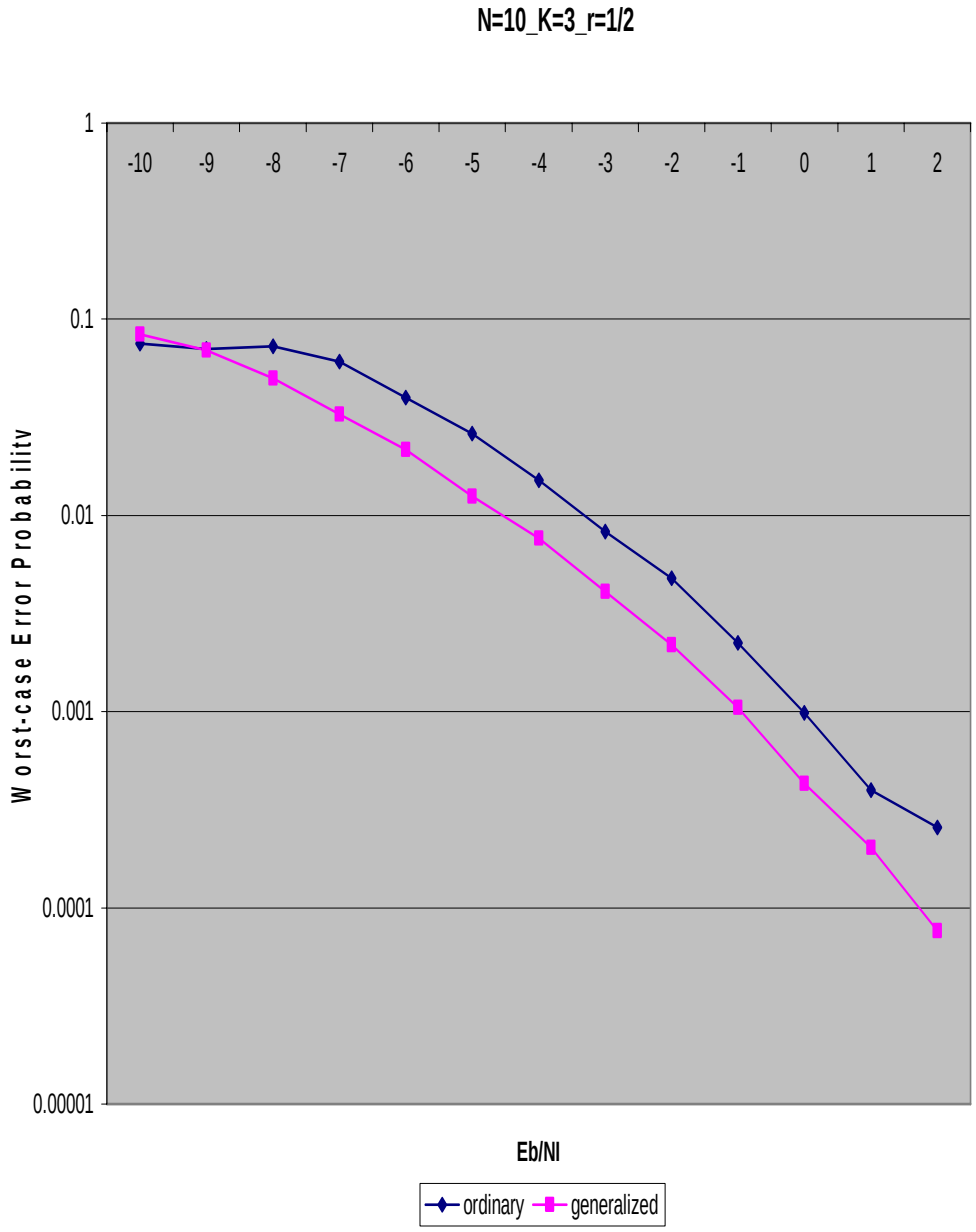

Figure 6.1: Code Rate, $r = 1/2$

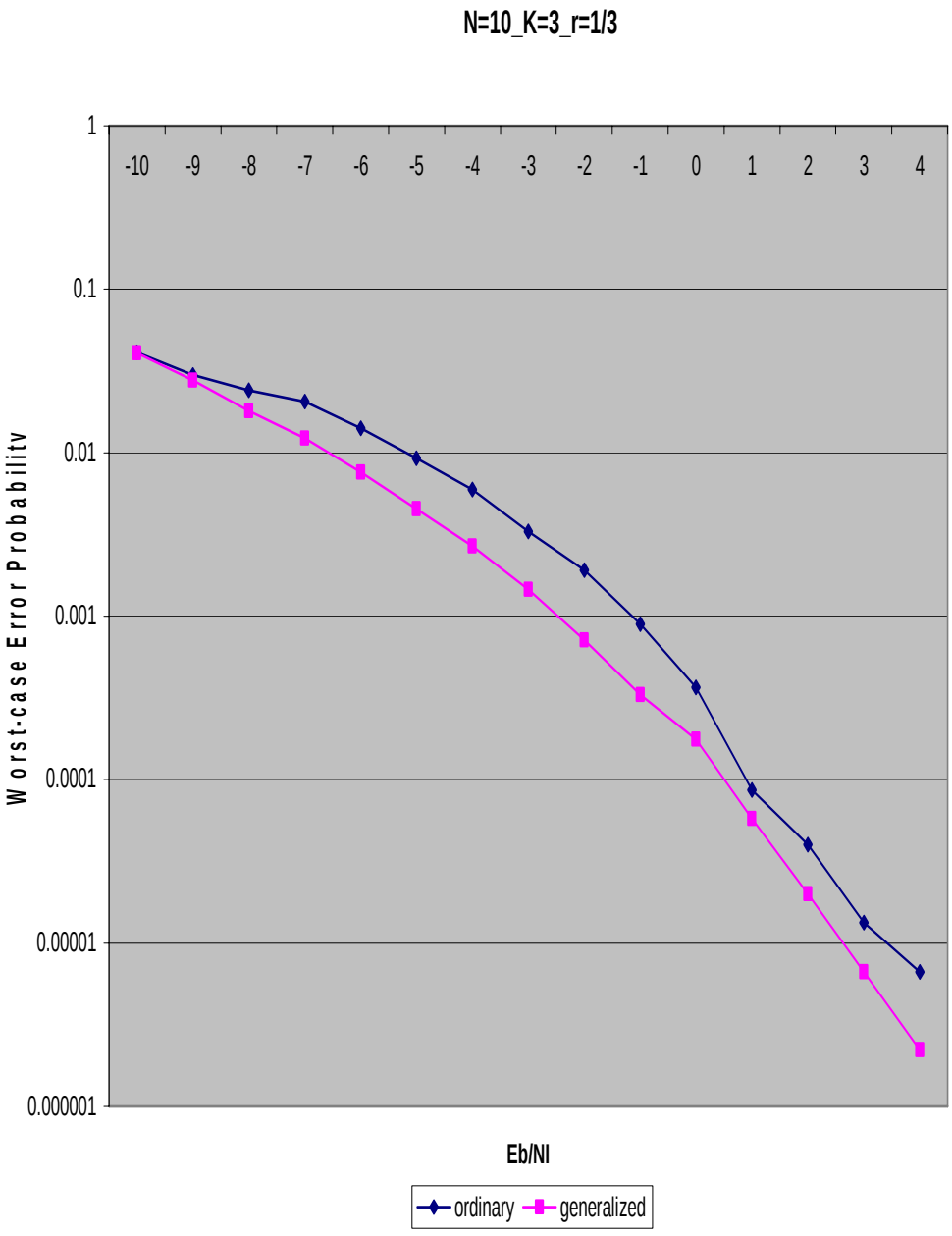

Figure 6.2: Code Rate, $r = 1/3$

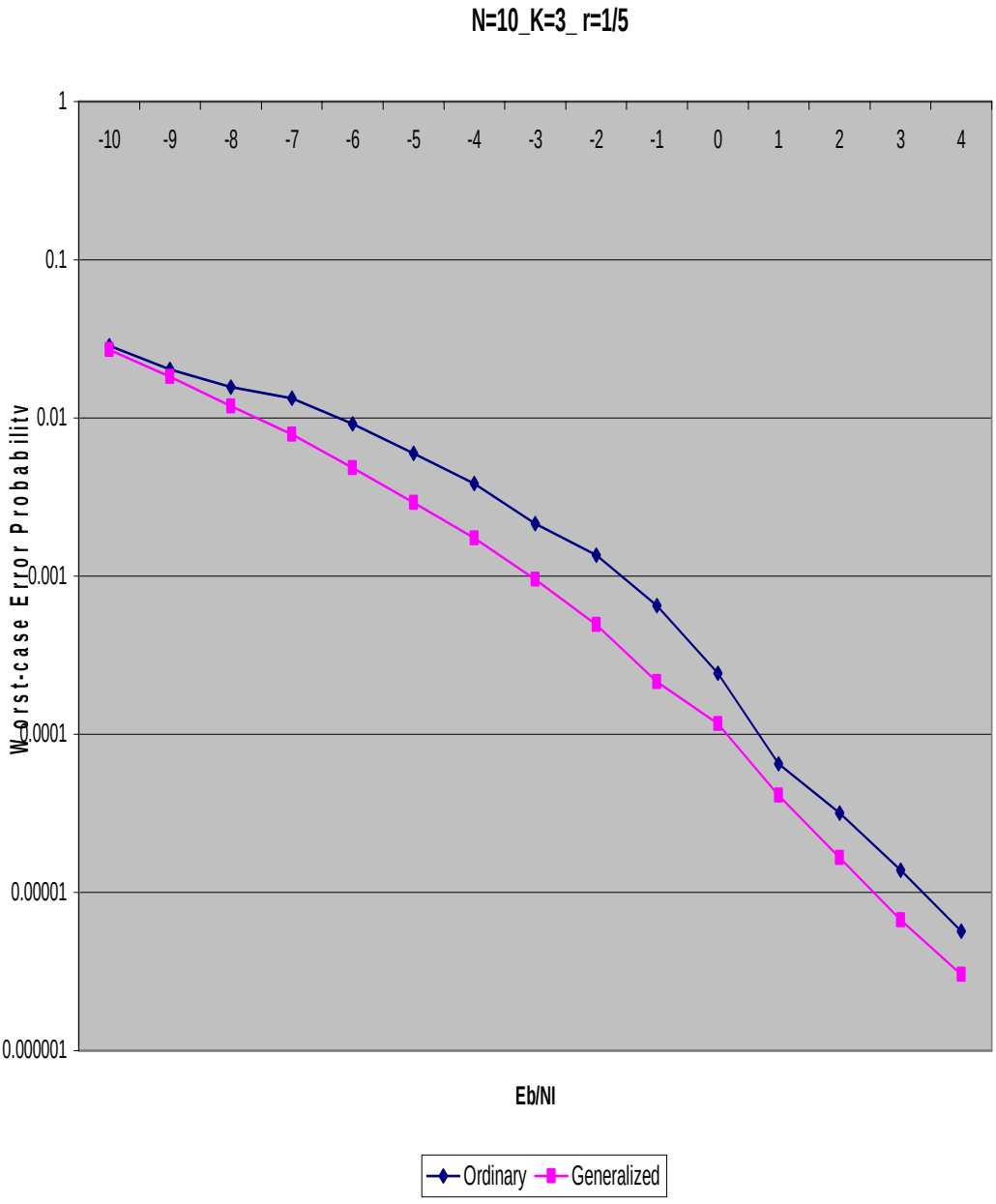

Figure 6.3: Code Rate, r = 1/5

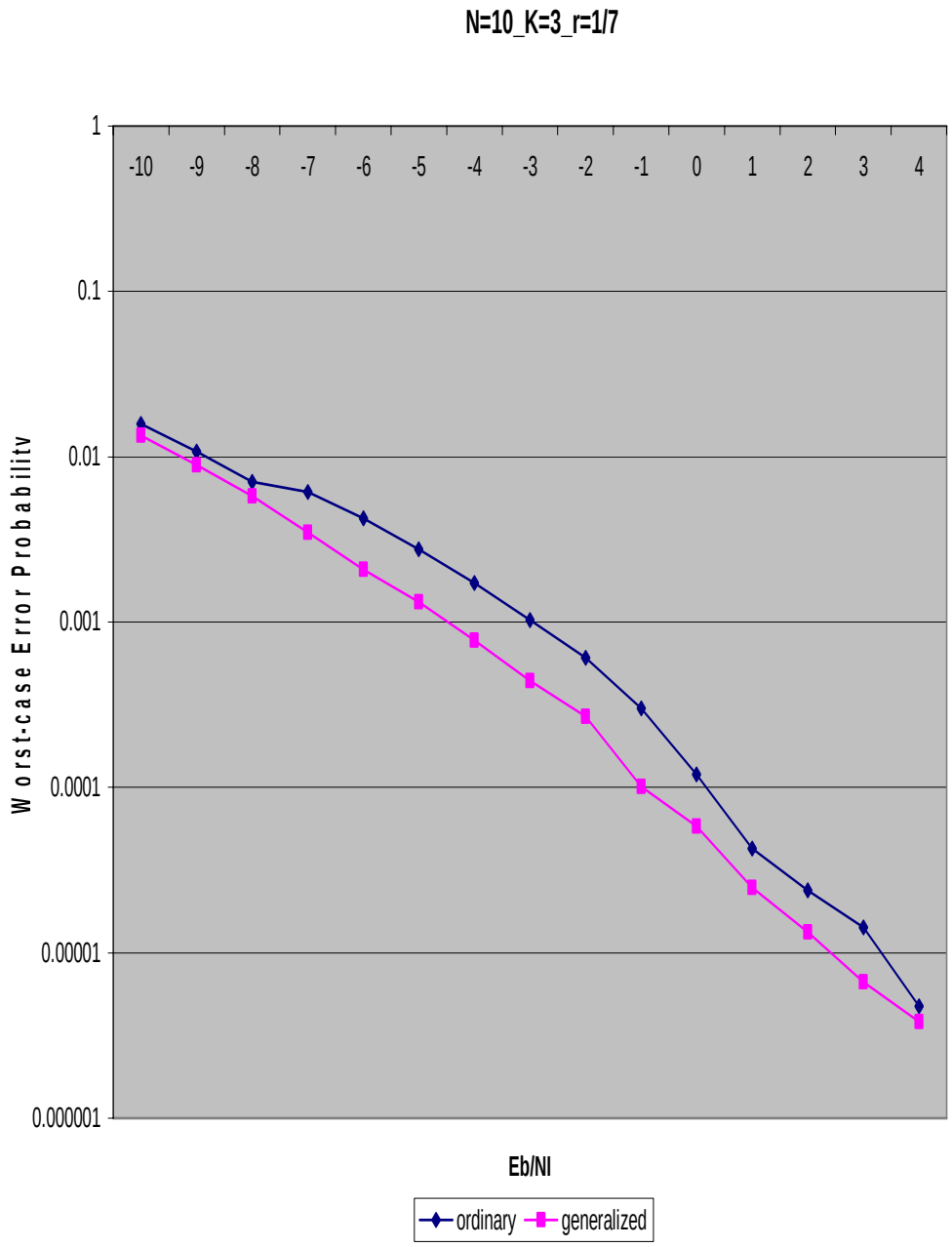

Figure 6.4: Code Rate, $r = 1/7$

6.2 Convolutional Codes with Same Code Rate

In this section we compare performance of convolutional codes with the same code rate and see how they perform as constraint length varies. All the codes used in this section have a chip length of $N = 10$, decoding depth of $5K$ and use convolutional interleaving with number of rows as 5.

The codes shown in Figure 6.5, 6.6 and 6.7 have the same code rate of $r = 1/2$. They have constraint lengths of $K = 3$, 5 and 7 respectively. Comparing the performance at 10^{-3} level, we get Table VII.

TABLE VII: VARYING CONSTRAINT LENGTH (K)

Code	Constraint Length (K)	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.5	3	-0.5	0.5	1
Figure 6.6	5	-0.6	0.3	0.9
Figure 6.7	7	-3.9	-3.1	0.8

From the results in this section it is observed that with constant code rate of the convolutional code as constraint length increases performance improves, at the same time the difference between two systems decreases. Again, it is expected that larger constraint lengths produce better results. It is interesting to see that the difference between the two systems gets smaller with increasing constraint length. This would suggest that generalized DSSS becomes increasingly more beneficial as coding memory is decreased.

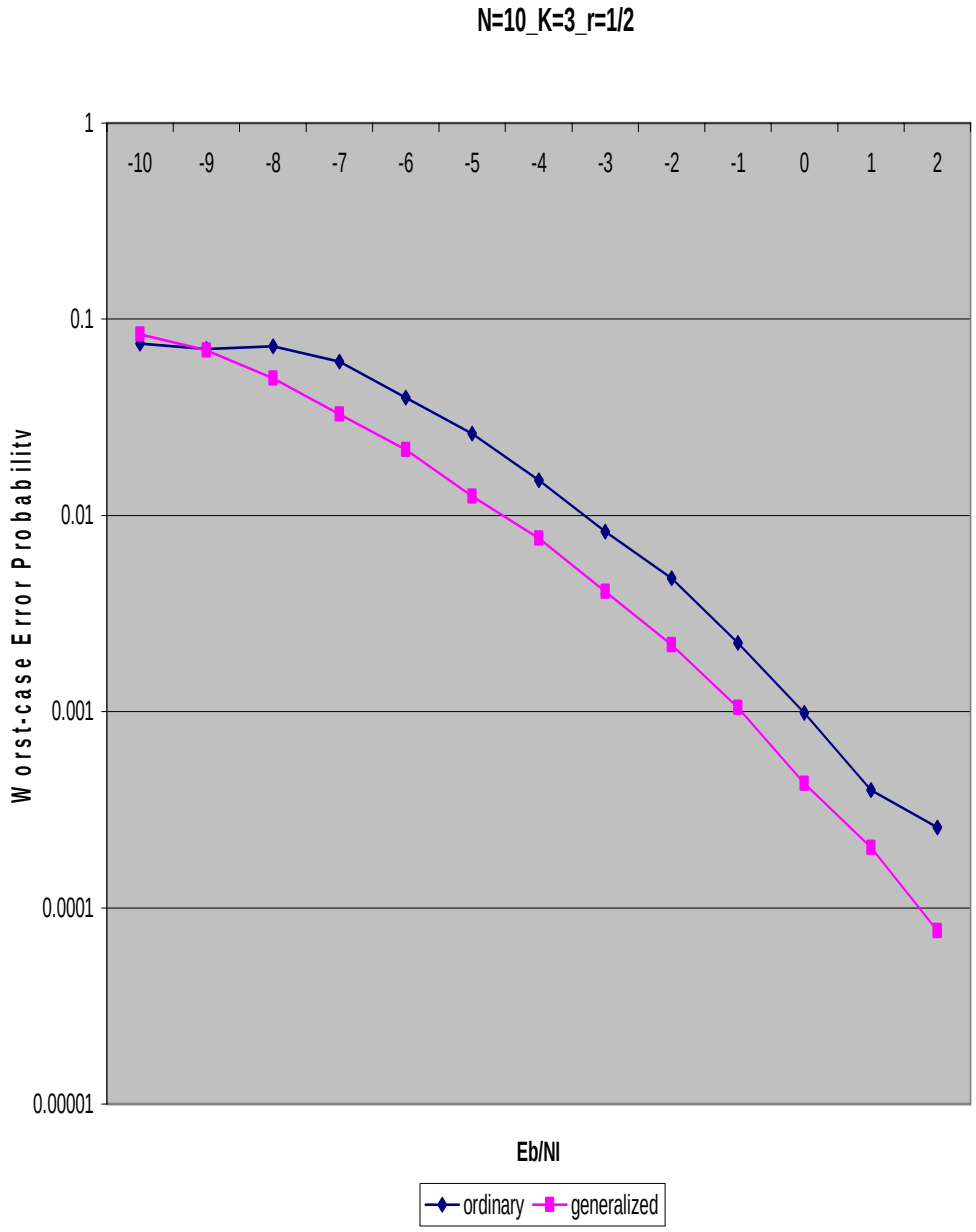

Figure 6.5: Constraint length, $K = 3$

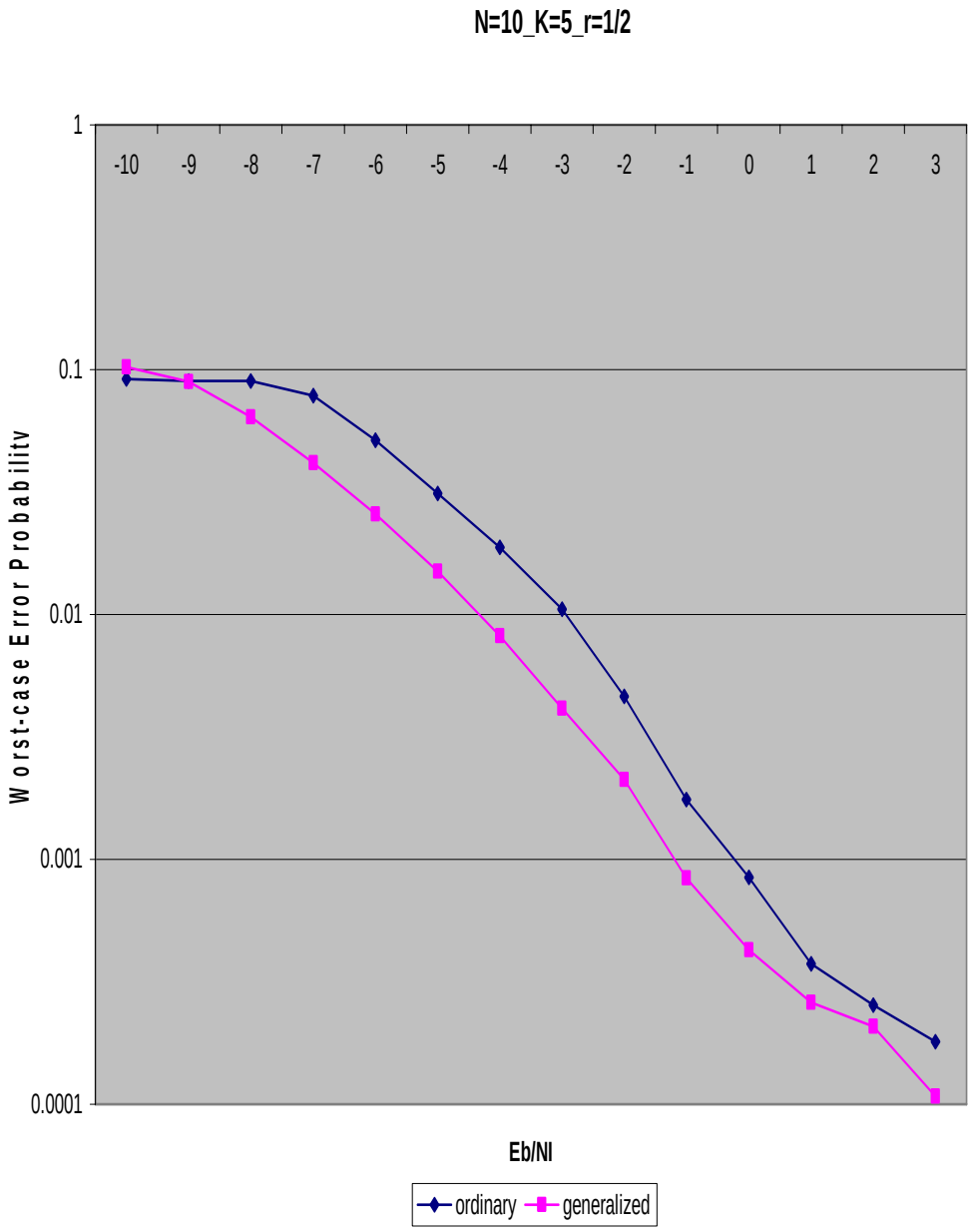

Figure 6.6: Constraint length, $K = 5$

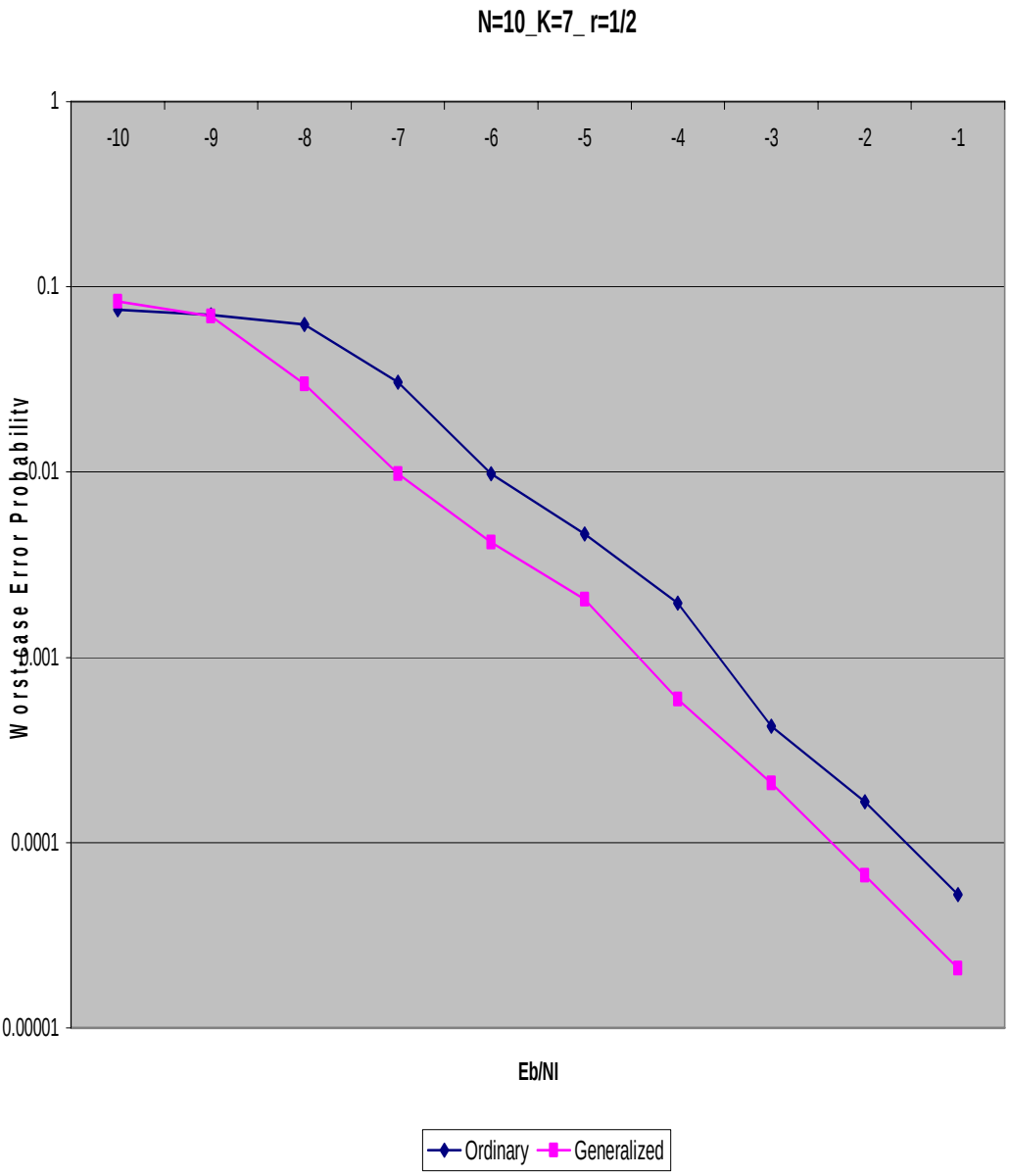

Figure 6.7: Constraint length, $K = 7$

6.3 Same Code Rate and Constraint Length with Varying Chip Length

In this section we compare codes with same code rate and same constraint length with varying length of pseudo-random chip sequence (N). All the codes used in this section have decoding depth of $5K$ and use convolutional Interleaving with number of rows as 5.

The codes shown in Figure 6.8 and 6.9 have constraint length of $K = 3$ and code rate of $r = 1/2$. They have chip length (N) of 10 and 20 respectively. Comparing the performance at 10^{-3} level, we get Table VIII.

TABLE VIII: VARYING CHIP LENGTH FOR $K = 3$ AND $R = 1/2$

Code	Chip Length (N)	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.8	10	-0.5	0.5	1
Figure 6.9	20	-3.6	-2.8	0.8

The codes shown in Figure 6.10 and 6.11 have the same constraint length of $K = 3$ and same code rate of $r = 1/3$ respectively. Comparing the performance at 10^{-3} level, we get Table IX.

TABLE IX: VARYING CHIP LENGTH FOR $K = 3$ AND $R = 1/3$

Code	Chip Length (N)	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.10	10	-2.0	-0.6	1.4
Figure 6.11	20	-5.8	-4.7	0.9

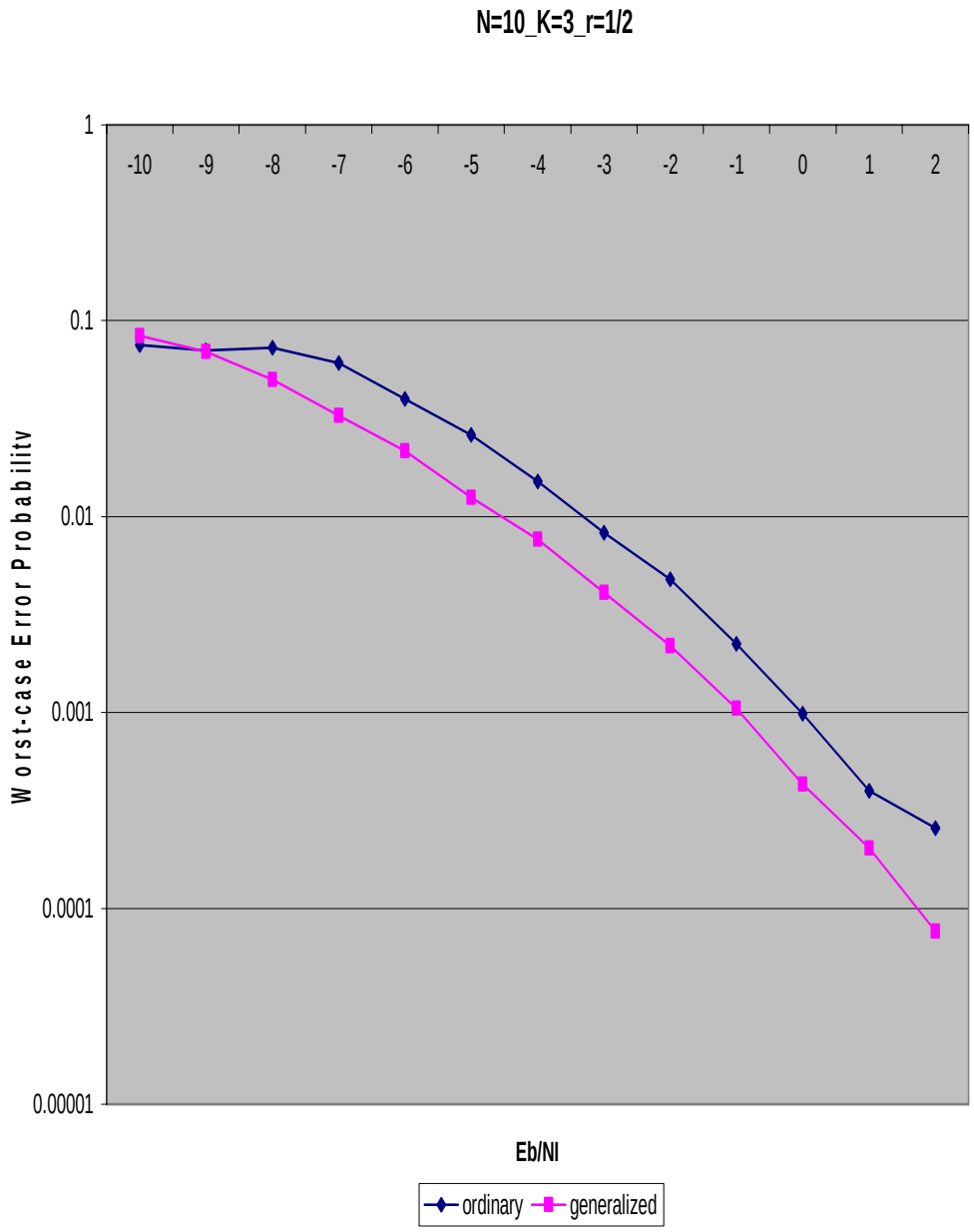

Figure 6.8: Chip Length, $N = 10$

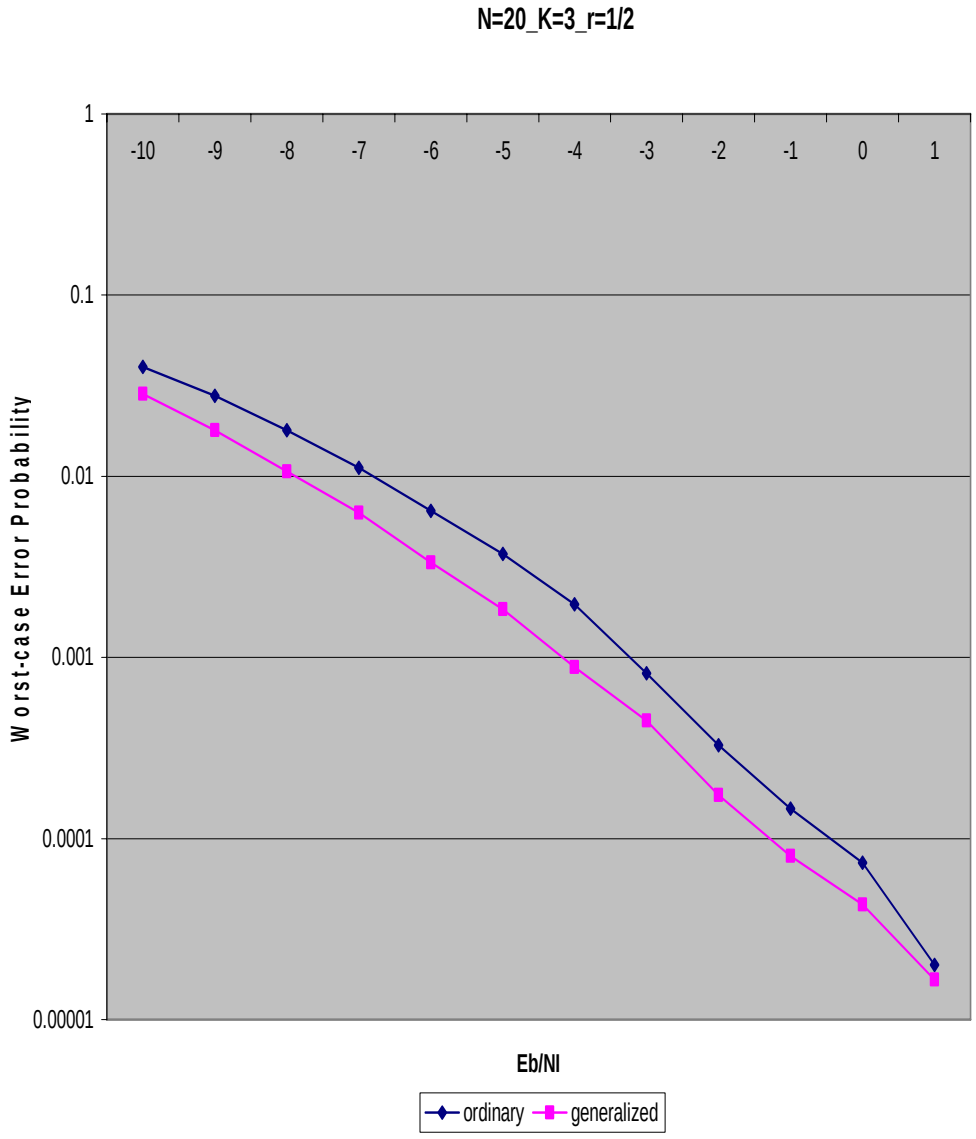

Figure 6.9: Chip Length, $N = 20$

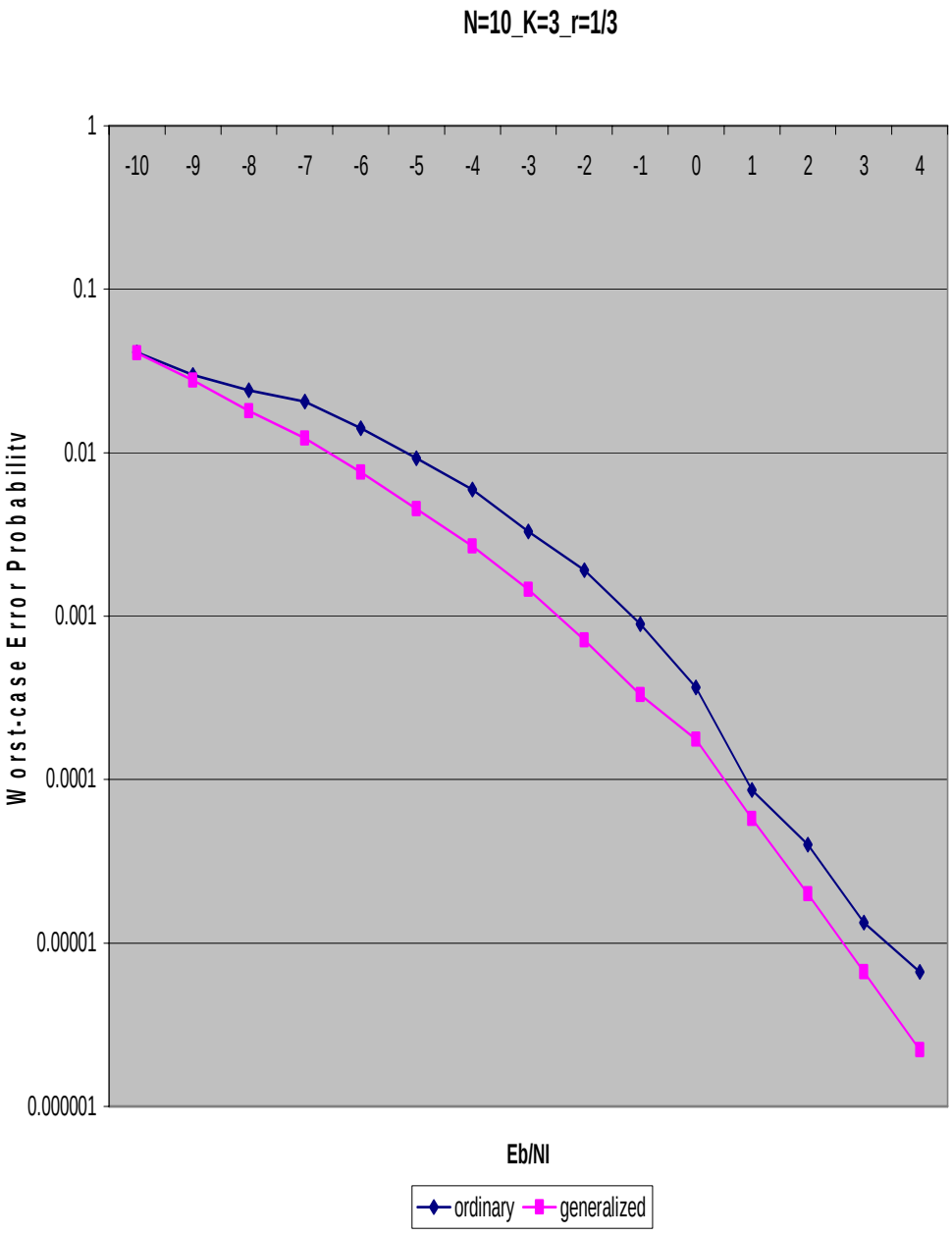

Figure 6.10: Chip Length, $N = 10$

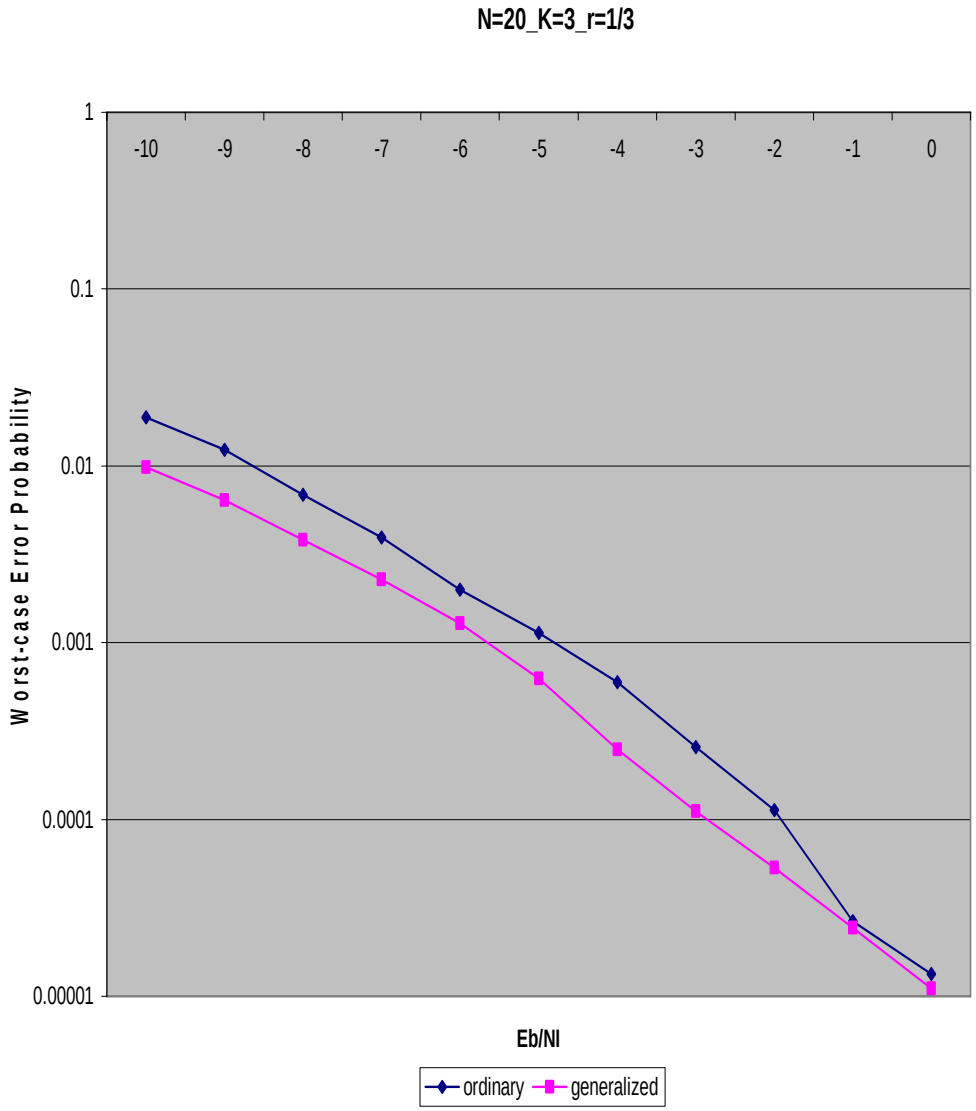

Figure 6.11: Chip Length, $N = 20$

The codes shown in Figure 6.12 and 6.13 have the same constraint length of K = 3, same code rate of r = 1/5 and used convolutional Interleaving. Comparing the performance at 10^{-3} level, we get Table X.

TABLE X: VARYING CHIP LENGTH FOR K = 3 AND R = 1/5

Code	Chip Length (N)	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.12	10	-2.5	-1.0	1.5
Figure 6.13	20	-5.1	-4.0	1.1

The codes shown in Figure 6.14 and 6.15 have the same constraint length of K = 3 and same code rate of r = 1/7 respectively. Comparing the performance at 10^{-3} level, we get Table XI.

TABLE XI: VARYING CHIP LENGTH FOR K = 3 AND R = 1/7

Code	Chip Length (N)	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.14	10	-4.0	-2.4	1.6
Figure 6.15	20	-4.3	-3.0	1.3

The codes shown in Figure 6.16 and 6.17 have constraint length of K = 3 and code rate of r = 1/2. Comparing the performance at 10^{-3} level, we get Table XII.

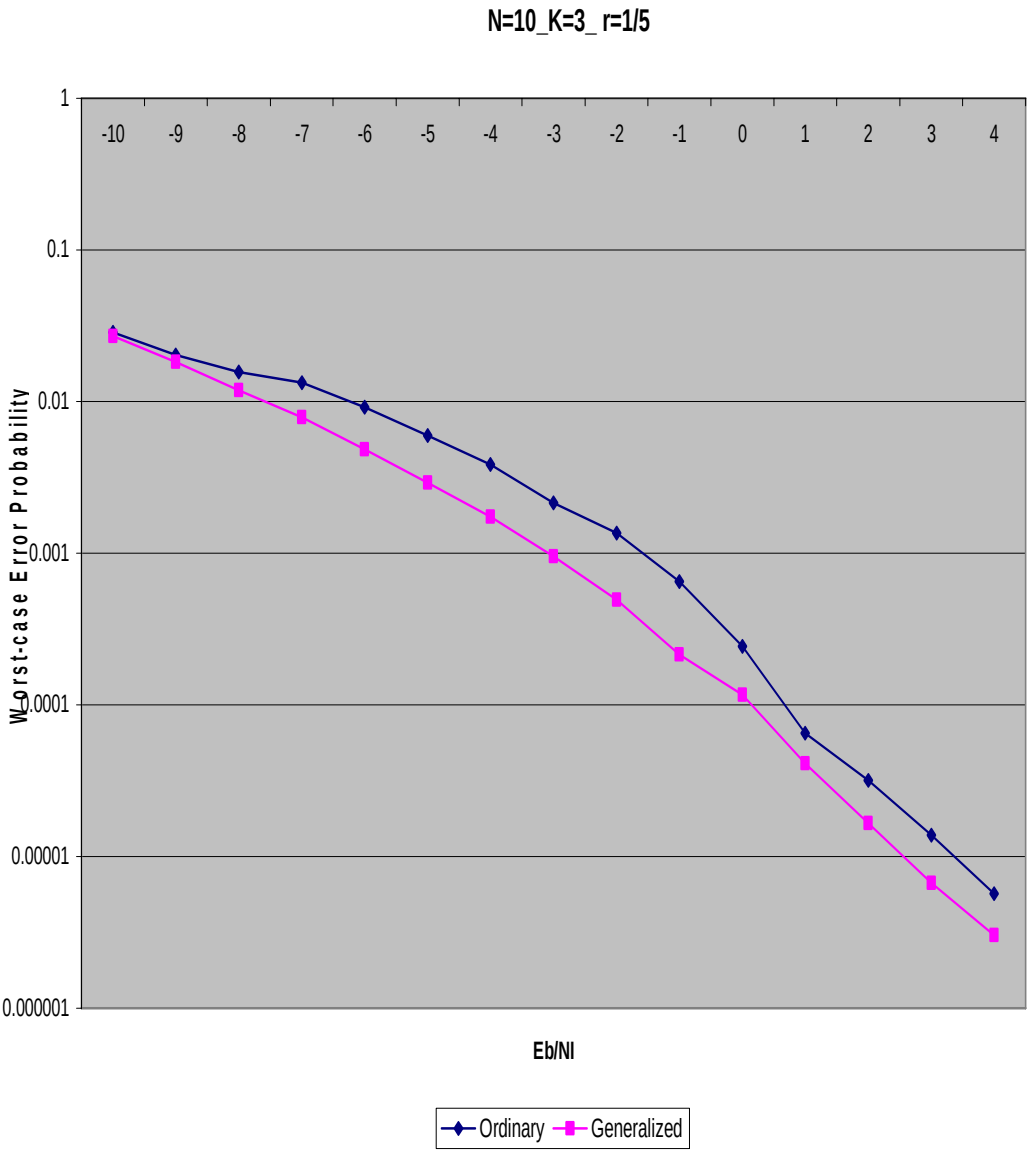

Figure 6.12: Chip Length, $N = 10$

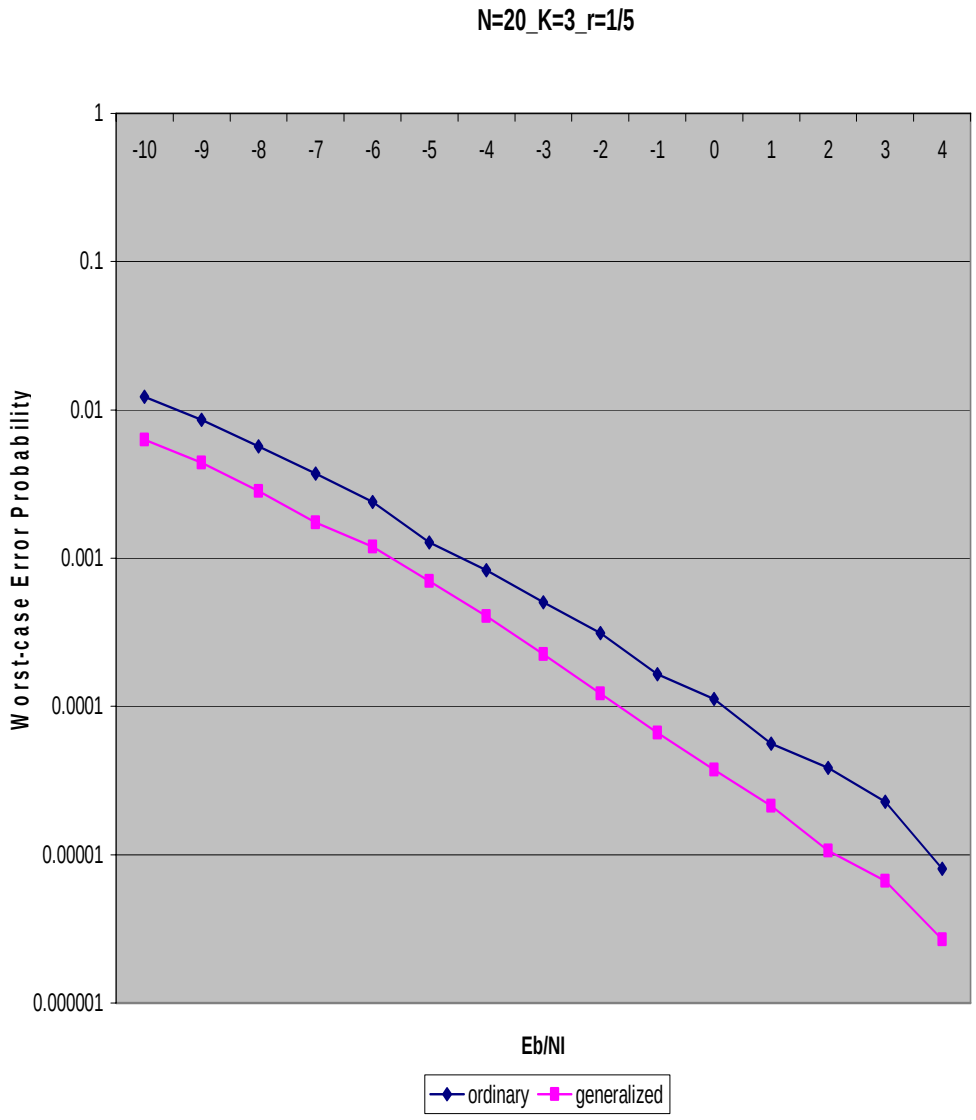

Figure 6.13: Chip Length, $N = 20$

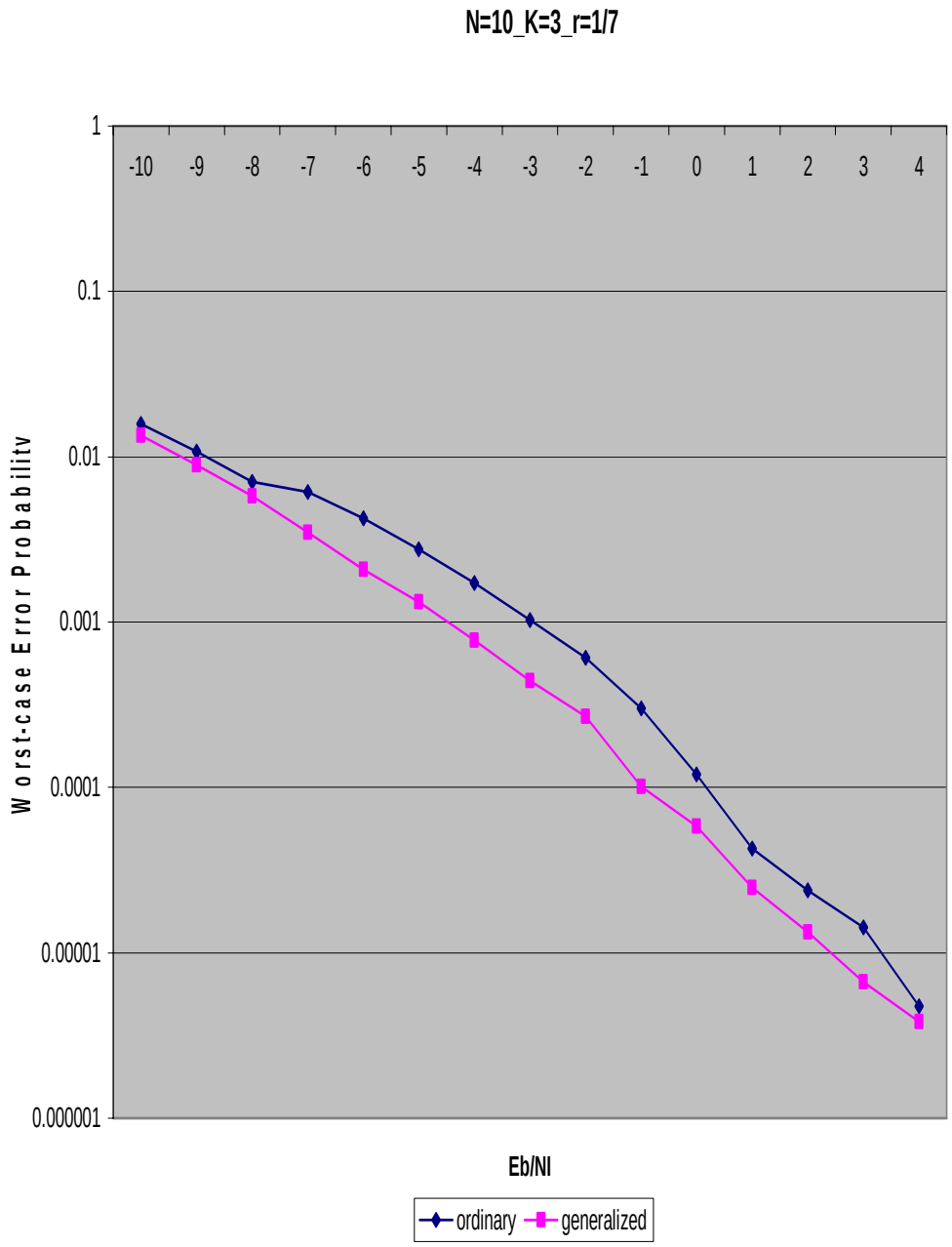

Figure 6.14: Chip Length, $N = 10$

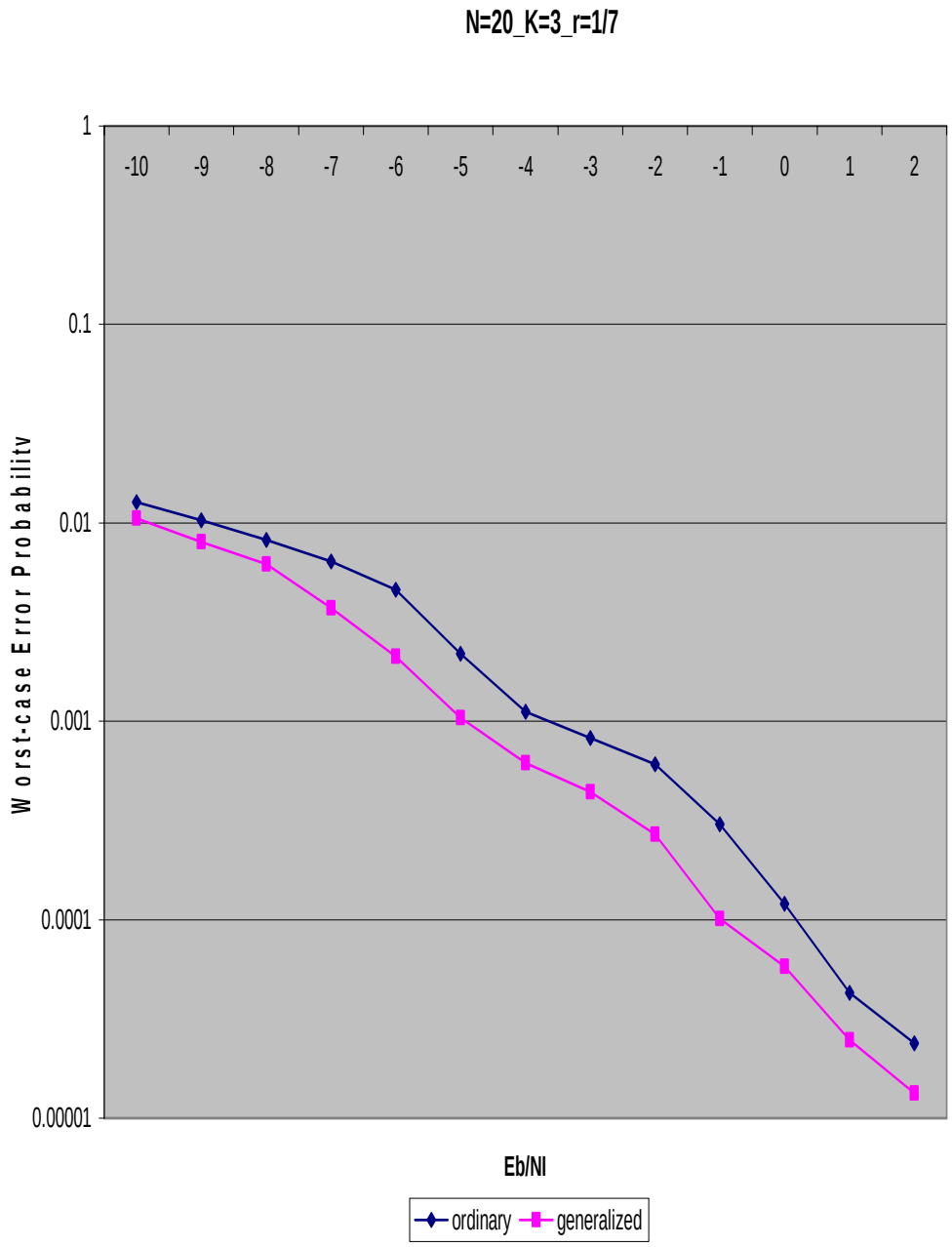

Figure 6.15: Chip Length, $N = 20$

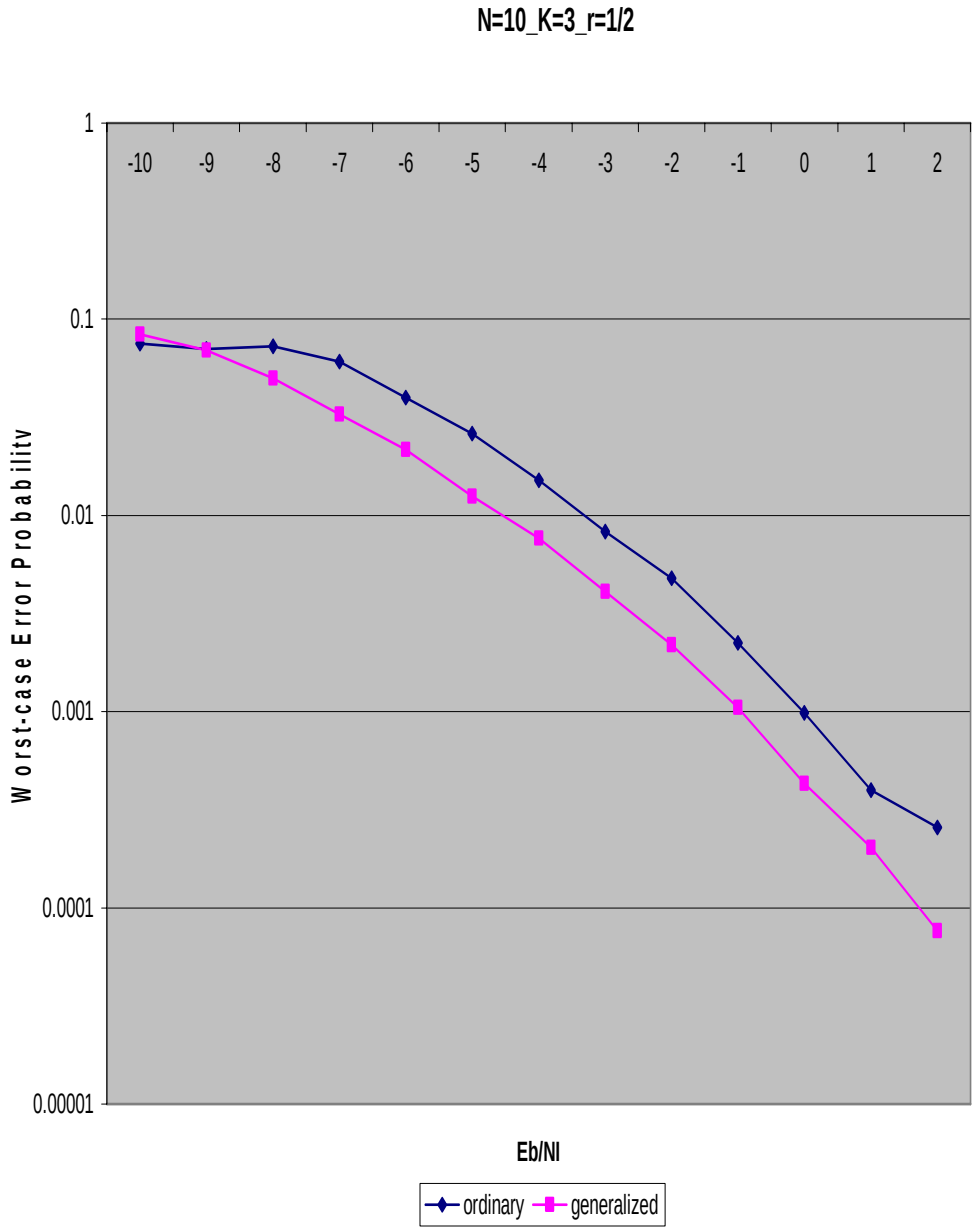

Figure 6.16: Chip Length, $N = 10$

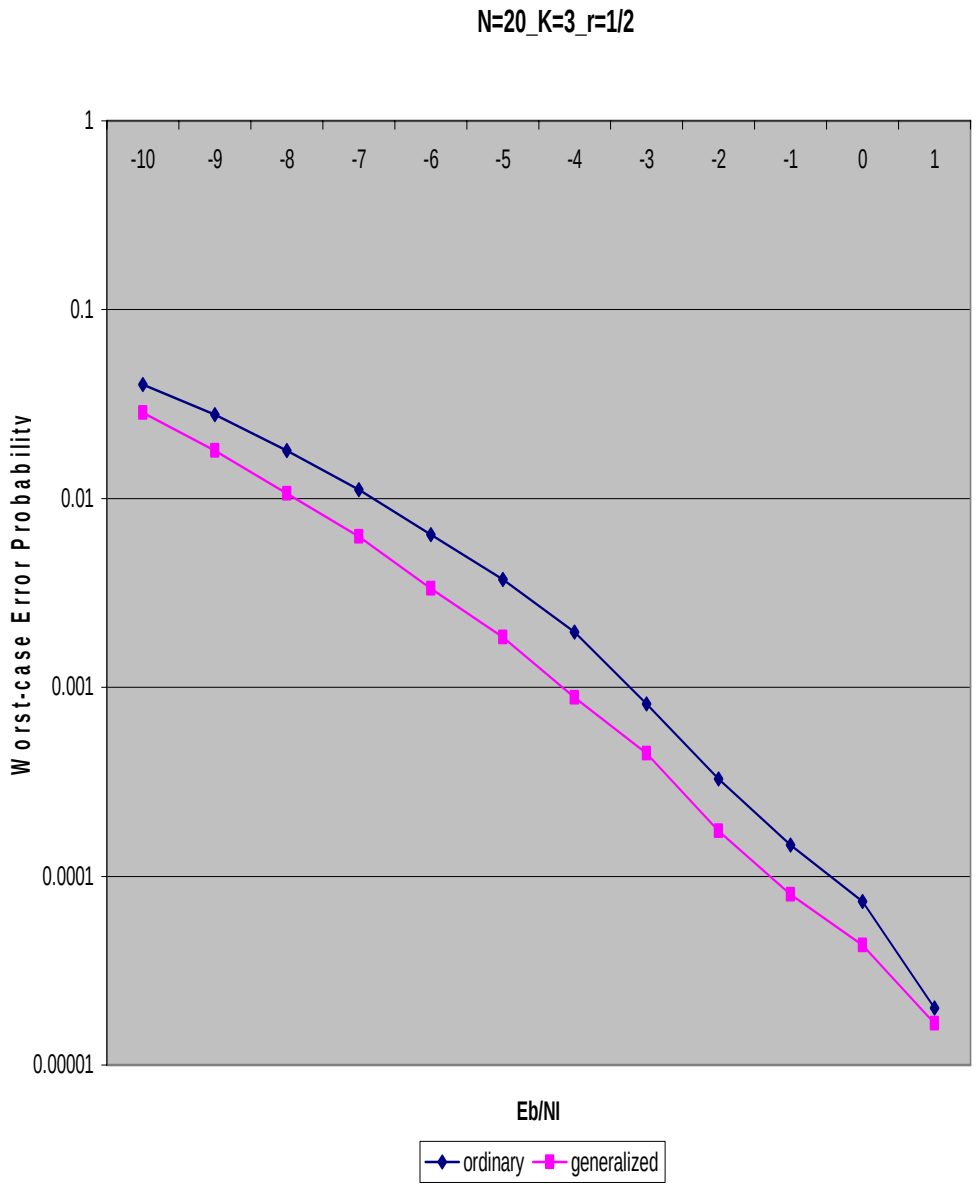

Figure 6.17: Chip length, $N = 20$

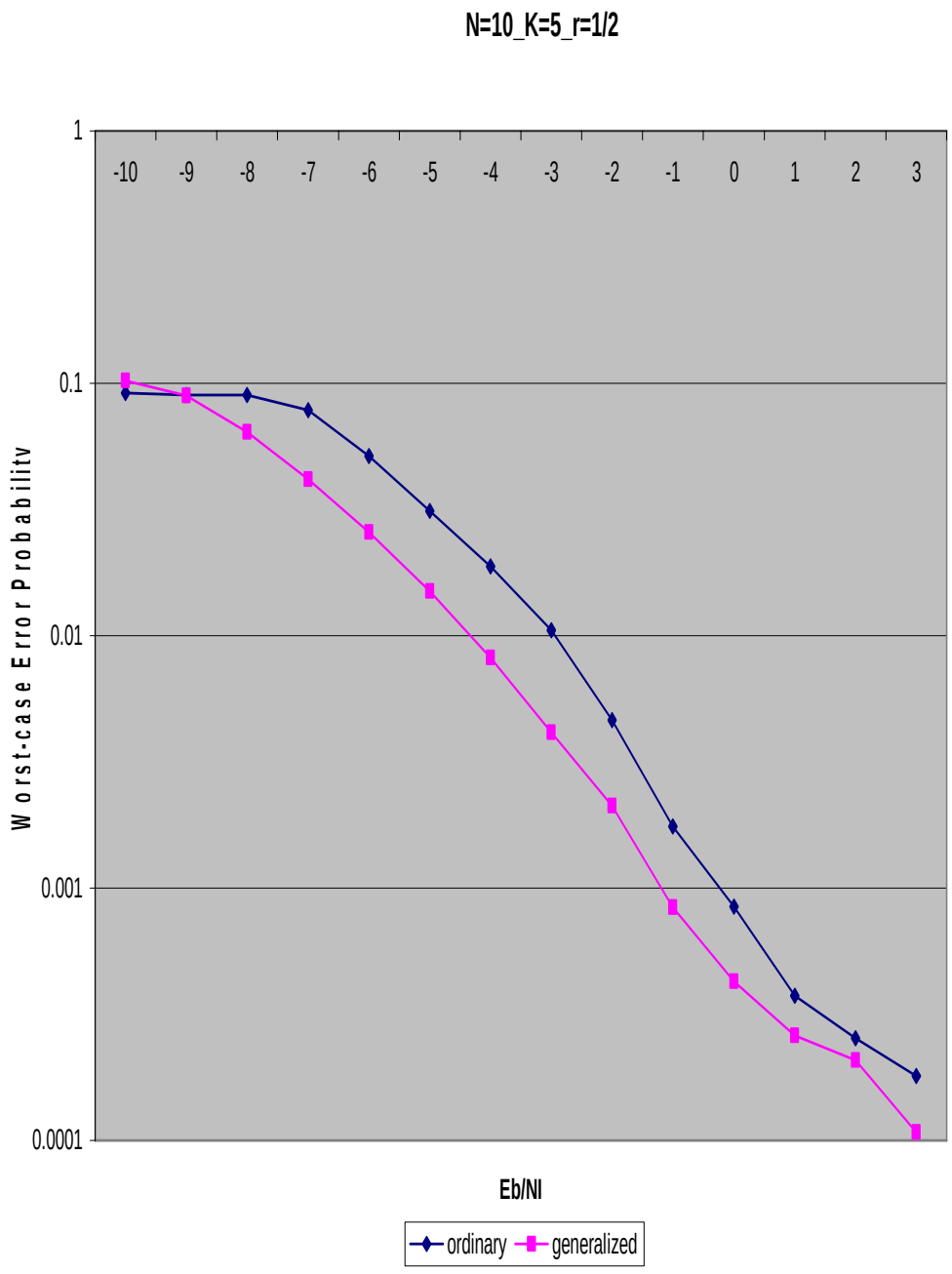

Figure 6.18: Chip length, $N = 10$

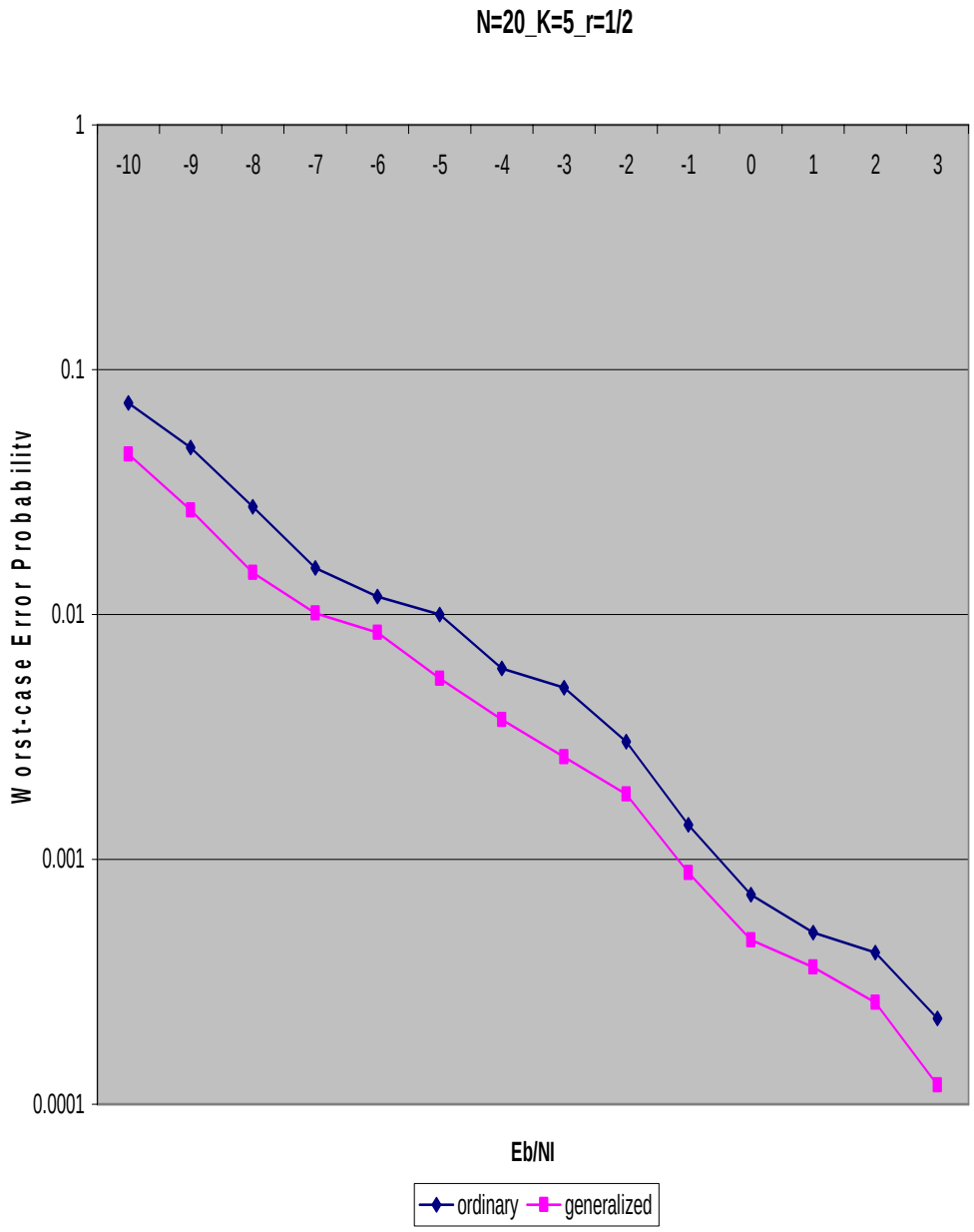

Figure 6.19: Chip Length, $N = 20$

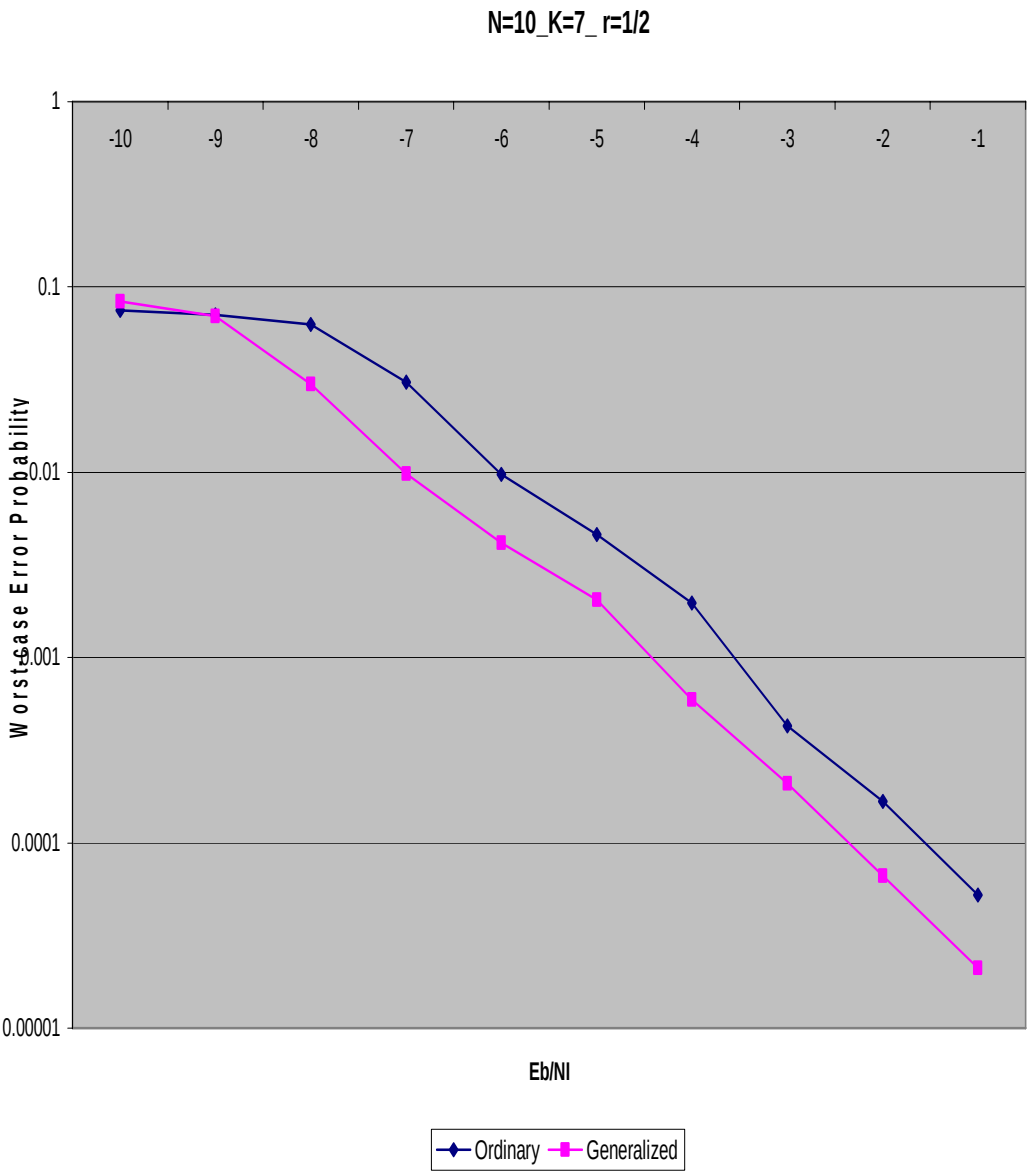

Figure 6.20: Chip Length, $N = 10$

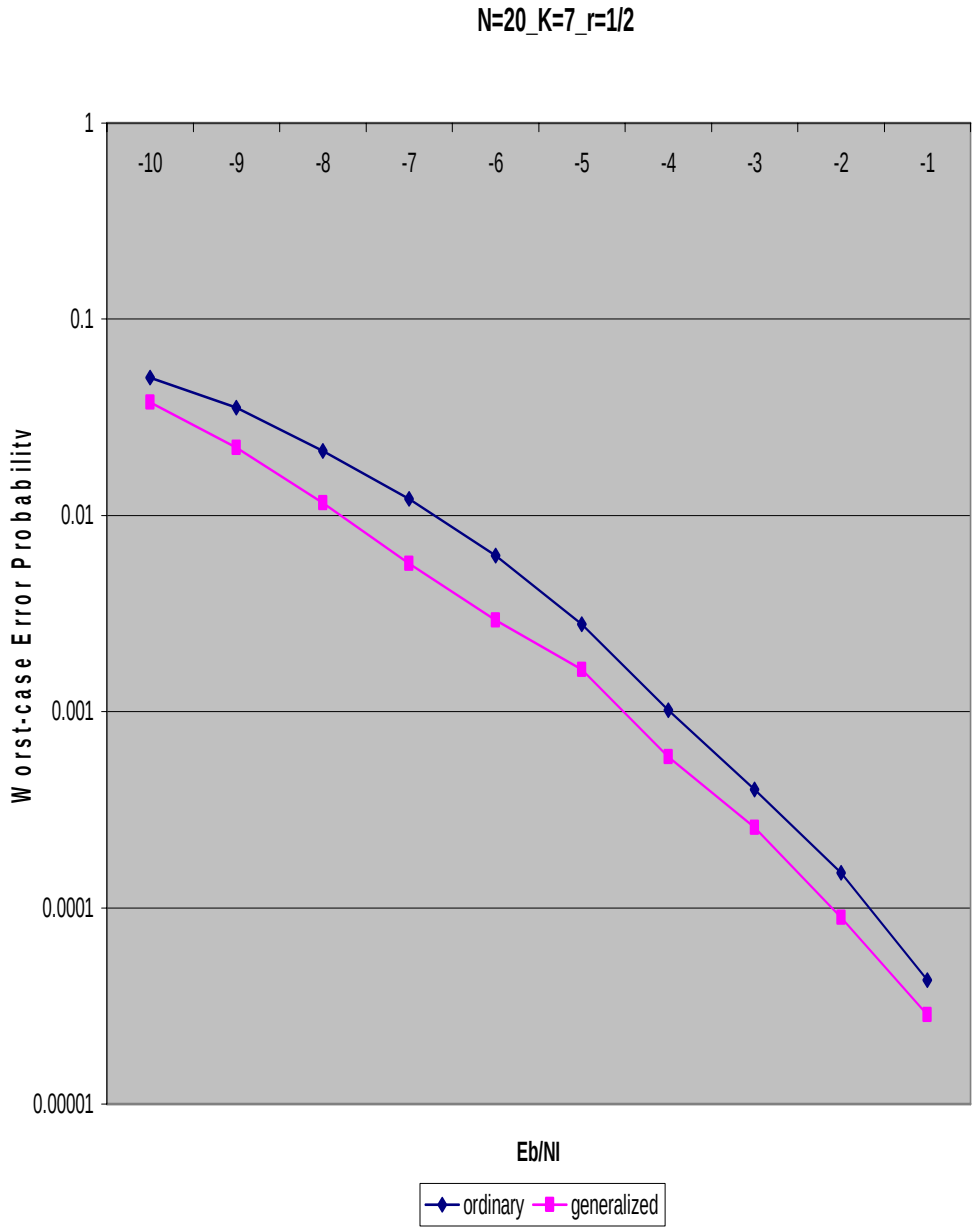

Figure 6.21: Chip Length, $N = 20$

TABLE XII: VARYING CHIP LENGTH FOR $K = 3$ AND $R = 1/2$

Code	Chip Length (N)	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.16	10	-0.5	0.5	1
Figure 6.17	20	-3.6	-2.8	0.8

The codes shown in Figure 6.18 and 6.19 have constraint length of $K = 5$ and code rate of $r = 1/2$. Comparing the performance at 10^{-3} level, we get Table XIII.

TABLE XIII: VARYING CHIP LENGTH FOR $K = 5$ AND $R = 1/2$

Code	Chip Length (N)	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.18	10	-0.6	0.3	0.9
Figure 6.19	20	-0.7	0	0.7

The codes shown in Figure 6.20 and 6.21 have constraint length of $K = 7$ and code rate of $r = 1/2$. Comparing the performance at 10^{-3} level, we get Table XIV.

TABLE XIV: VARYING CHIP LENGTH FOR $K = 7$ AND $R = 1/2$

Code	Chip Length (N)	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.20	10	-3.9	-3.1	0.8
Figure 6.21	20	-4.0	-3.5	0.5

From the results in this section it is observed that with same constraint length and same code rate as chip length (N) increases performance improves, at the same time the

difference between two systems decreases. Again, the first observation here is obvious. The second observation suggests that generalized DSSS becomes increasingly more beneficial compared to ordinary DSSS as other parameters of the communication system (in this case N) are varied to make it less robust.

6.4 Same Code Rate and Constraint Length with Varying Interleaver

In this section we compare codes with same code rate and same constraint length by varying Interleaver between convolutional and random. All the codes used in this section have chip length of $N = 10$, decoding depth of $5K$ and number of rows of convolutional Interleaver as 5.

The codes shown in Figure 6.22 and 6.23 have the same constraint length of $K = 3$, same code rate of $r = 1/2$ and they use convolutional, random Interleaver respectively. Comparing the performance at 10^{-3} level, we get Table XV.

TABLE XV: VARYING INTERLEAVER FOR $K = 3$ AND $R = 1/2$

Code	Interleaver	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.22	Convolutional	-0.5	0.5	1
Figure 6.23	Random	-1.5	-1.0	0.5

The codes shown in Figure 6.24 and 6.25 have constraint length of $K = 3$ and code rate of $r = 1/3$. Comparing the performance at 10^{-3} level, we get Table XVI.

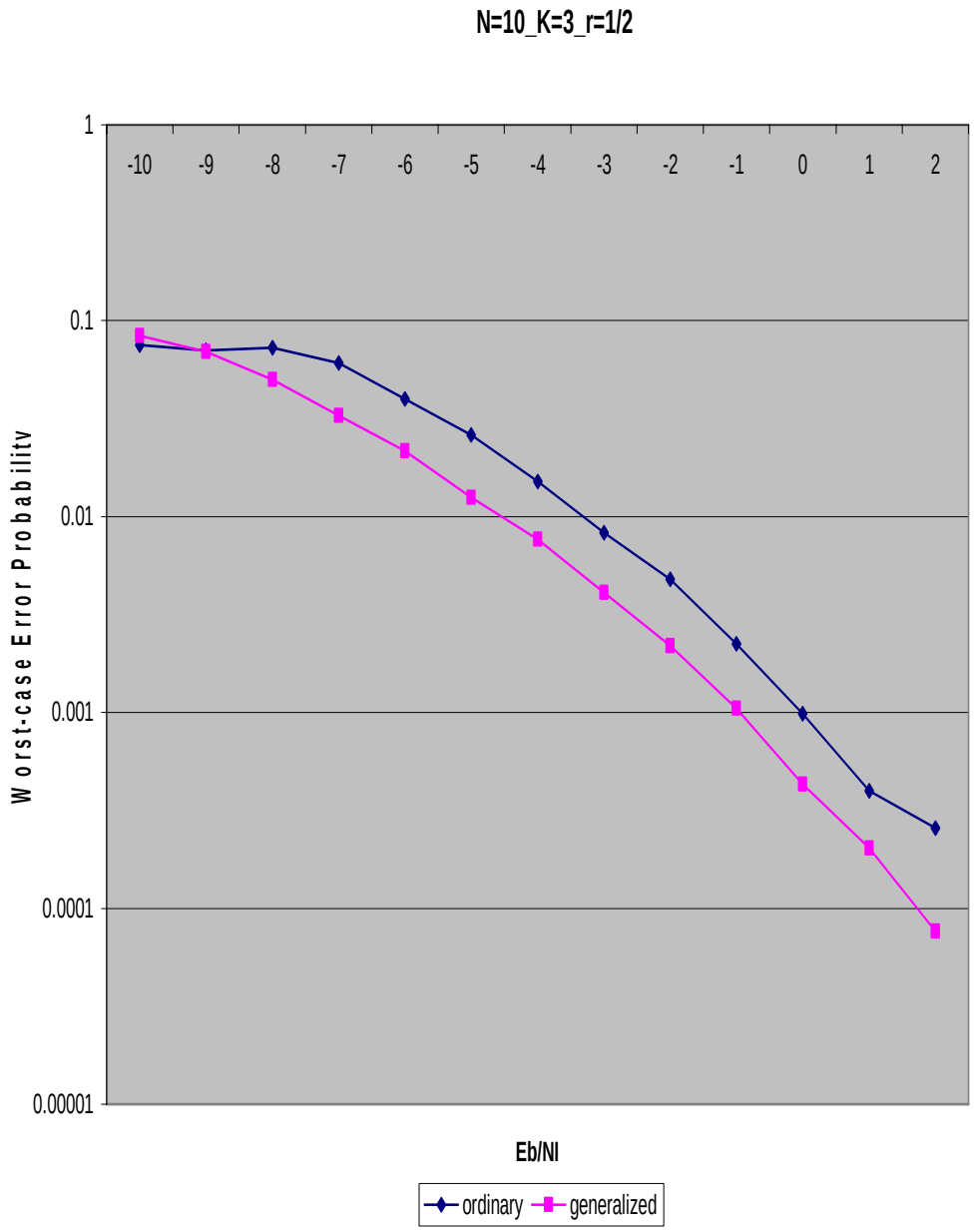

Figure 6.22: Convolutional Interleaver

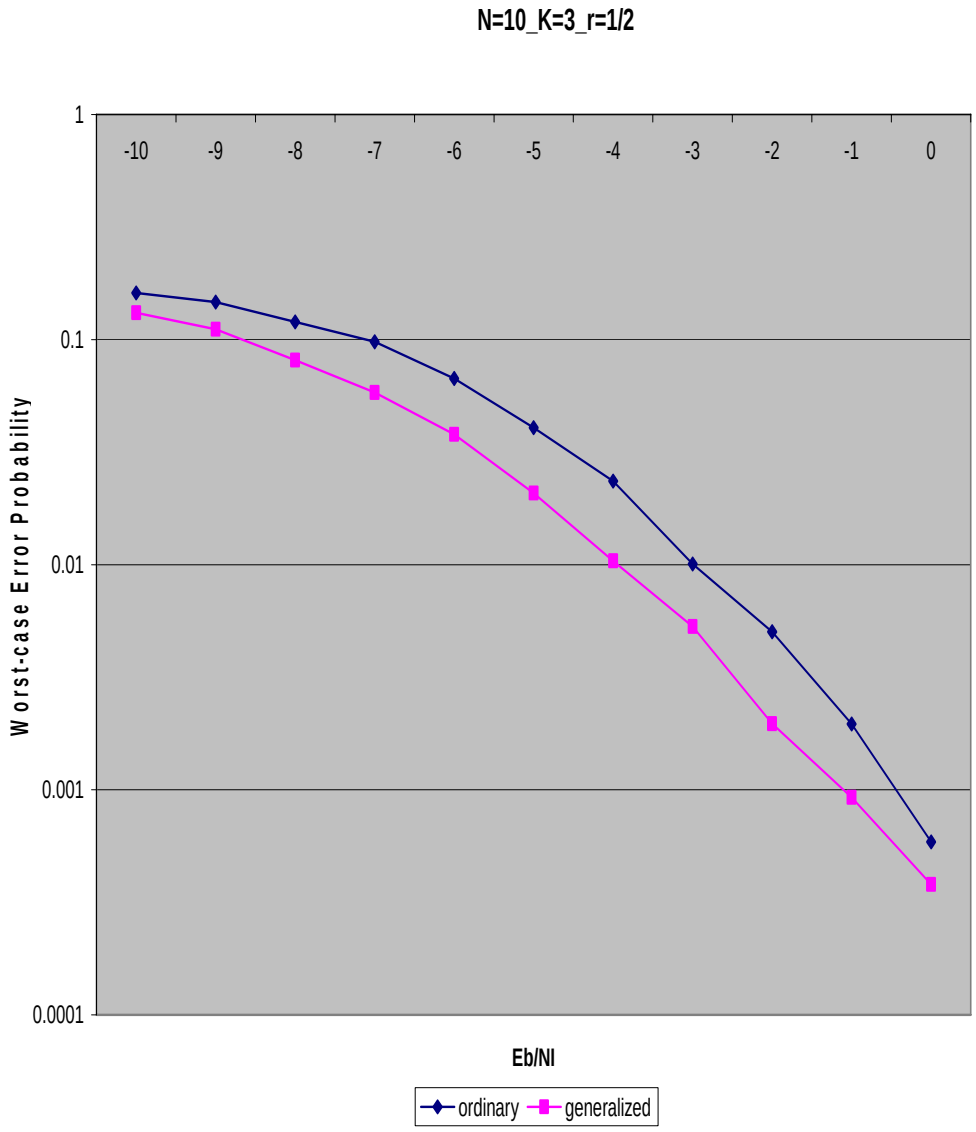

Figure 6.23: Random Interleaver

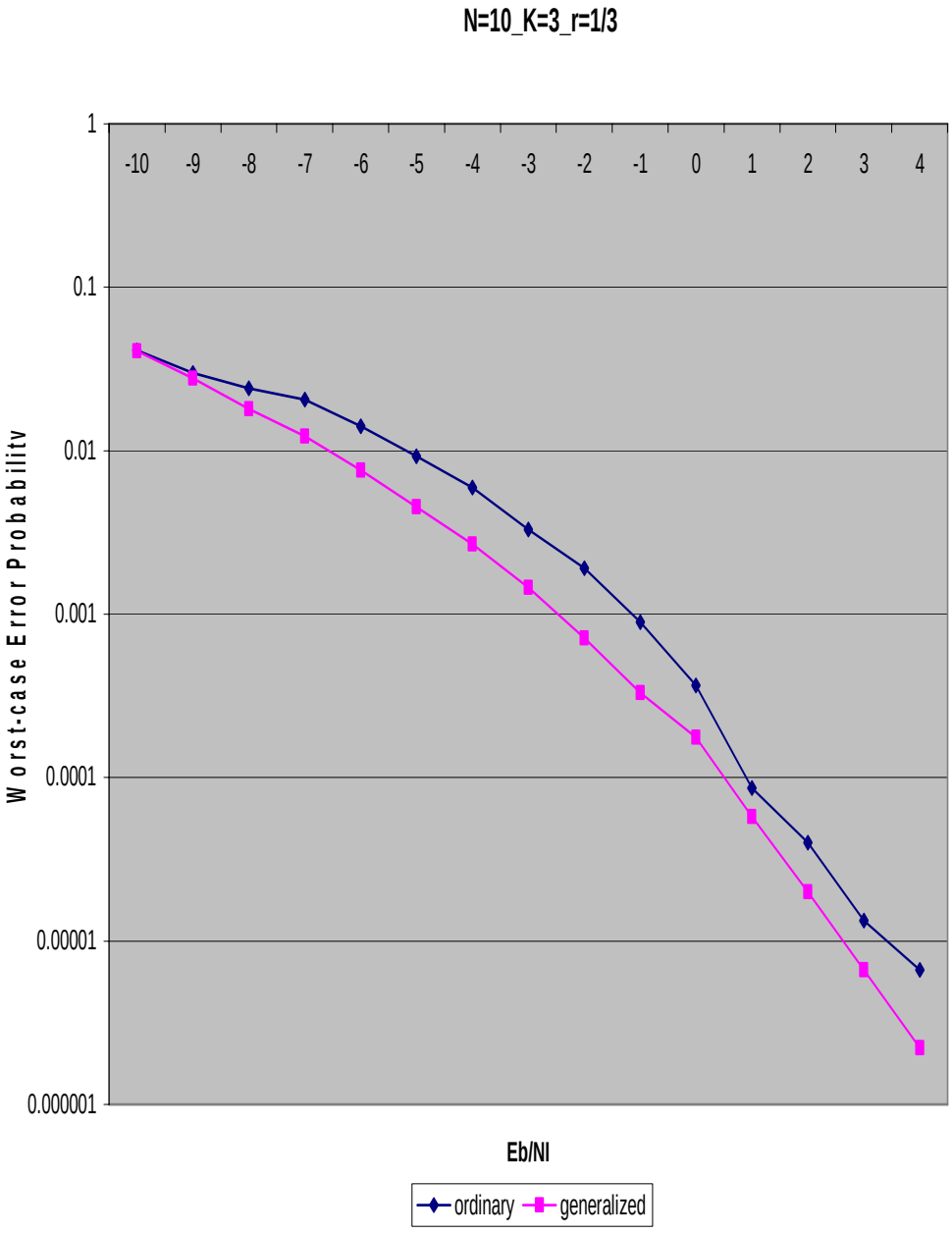

Figure 6.24: Convolutional Interleaver

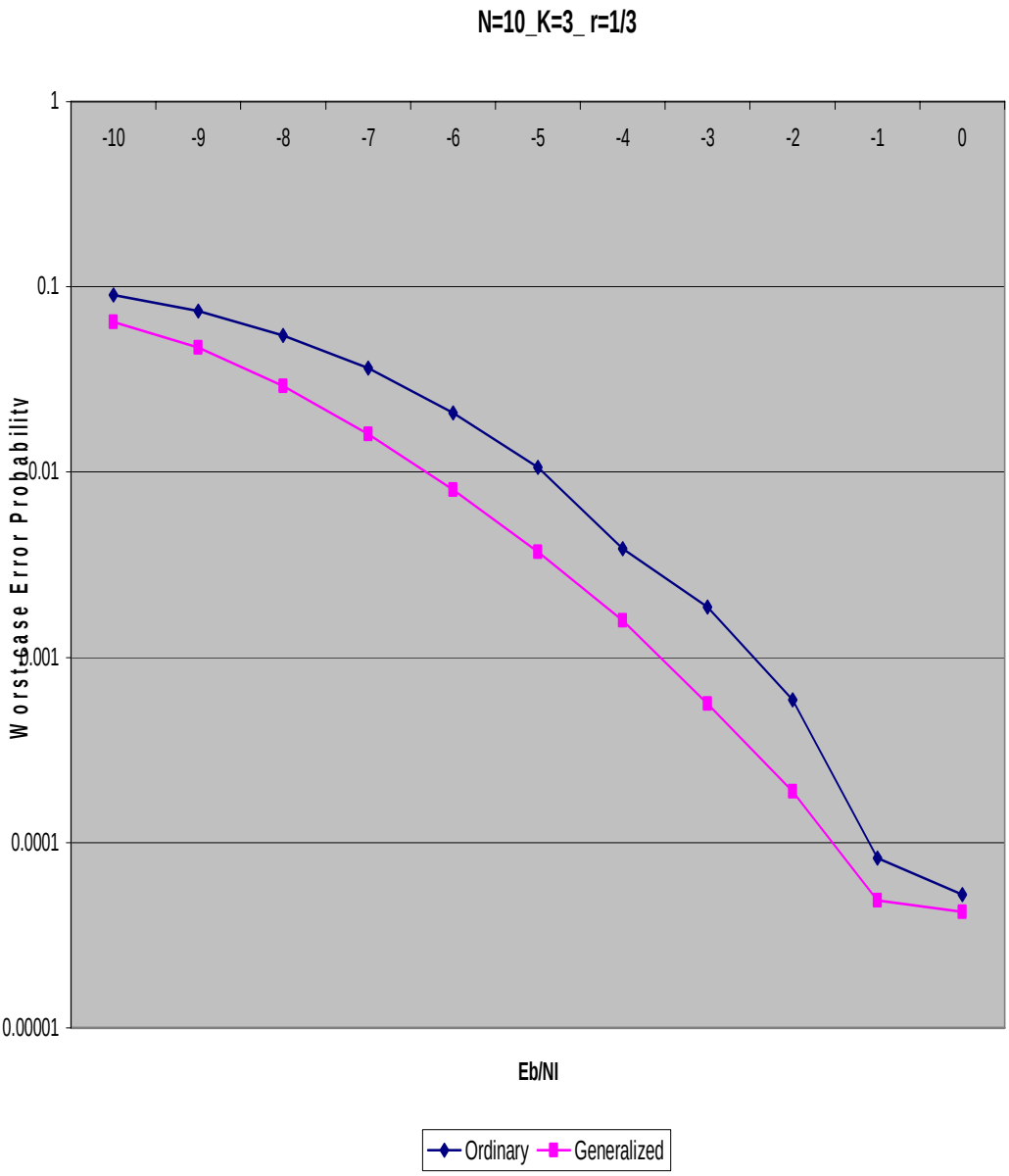

Figure 6.25: Random Interleaver

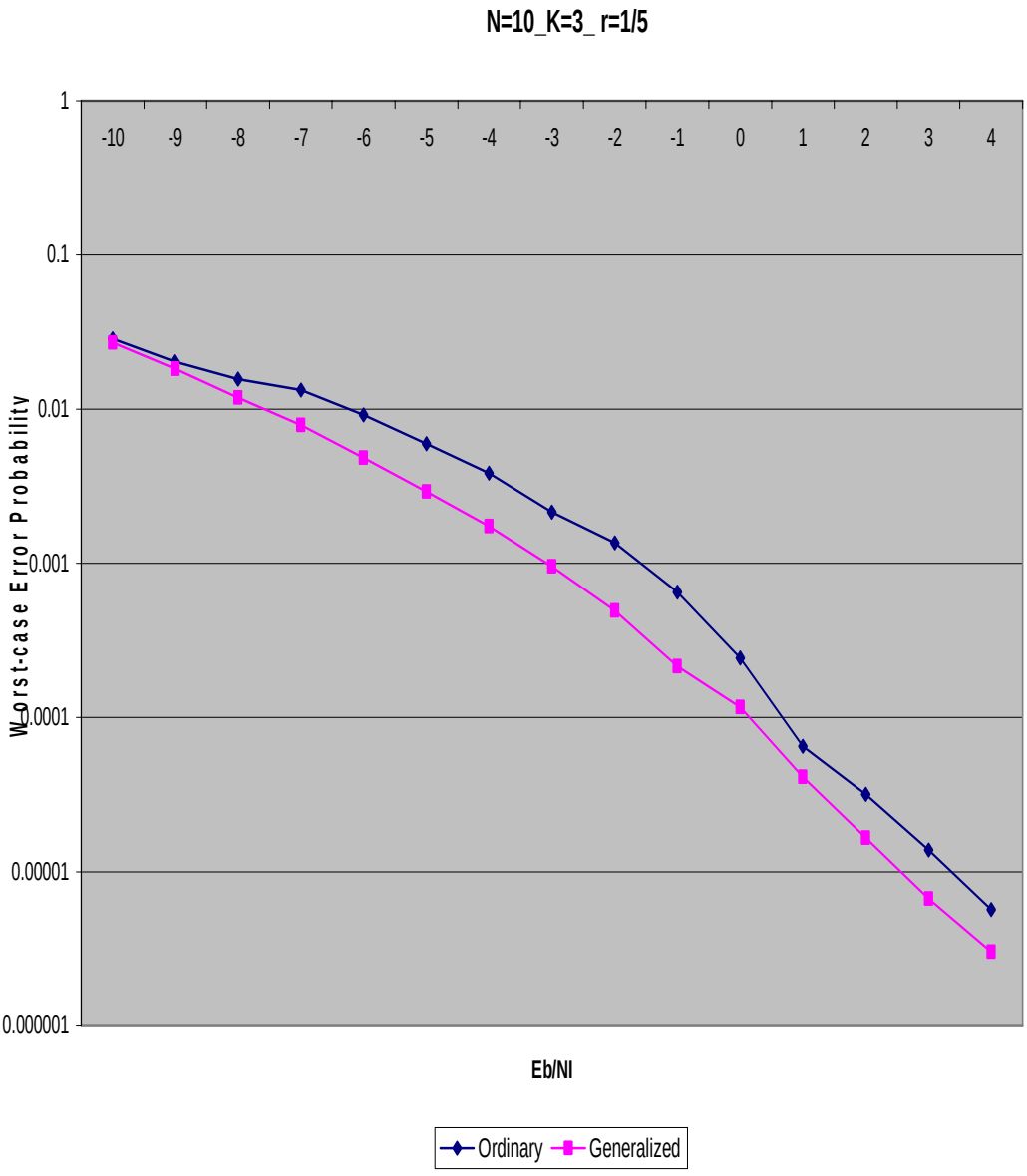

Figure 6.26: Convolutional Interleaver

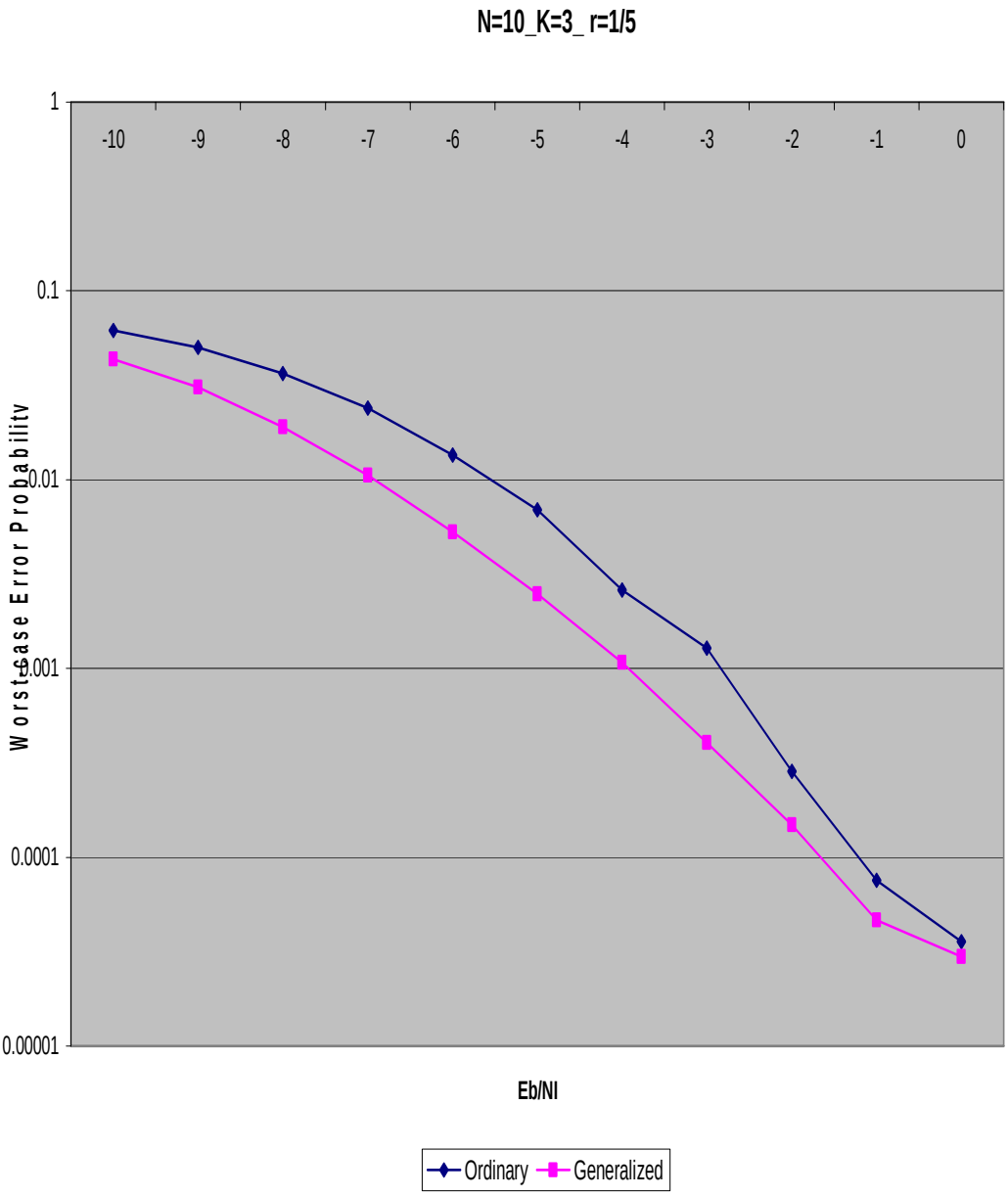

Figure 6.27: Random Interleaver

TABLE XVI: VARYING INTERLEAVER FOR $K = 3$ AND $R = 1/3$

Code	Interleaver	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.24	Convolutional	-2.0	-0.6	1.4
Figure 6.25	Random	-3.0	-2.0	1

The codes shown in Figure 6.26 and 6.27 have constraint length of $K = 3$ and code rate of $r = 1/5$. Comparing the performance at 10^{-3} level, we get Table XVII.

TABLE XVII: VARYING INTERLEAVER FOR $K = 3$ AND $R = 1/5$

Code	Interleaver	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.26	Convolutional	-2.5	-1.0	1.5
Figure 6.27	Random	-3.5	-2.3	1.2

The codes shown in Figure 6.28 and 6.29 have constraint length of $K = 3$ and code rate of $r = 1/7$. Comparing the performance at 10^{-3} level, we get Table XVIII.

TABLE XVIII: VARYING INTERLEAVER FOR $K = 3$ AND $R = 1/7$

Code	Interleaver	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.28	Convolutional	-4.0	-2.4	1.6
Figure 6.29	Random	-4.1	-3.0	1.1

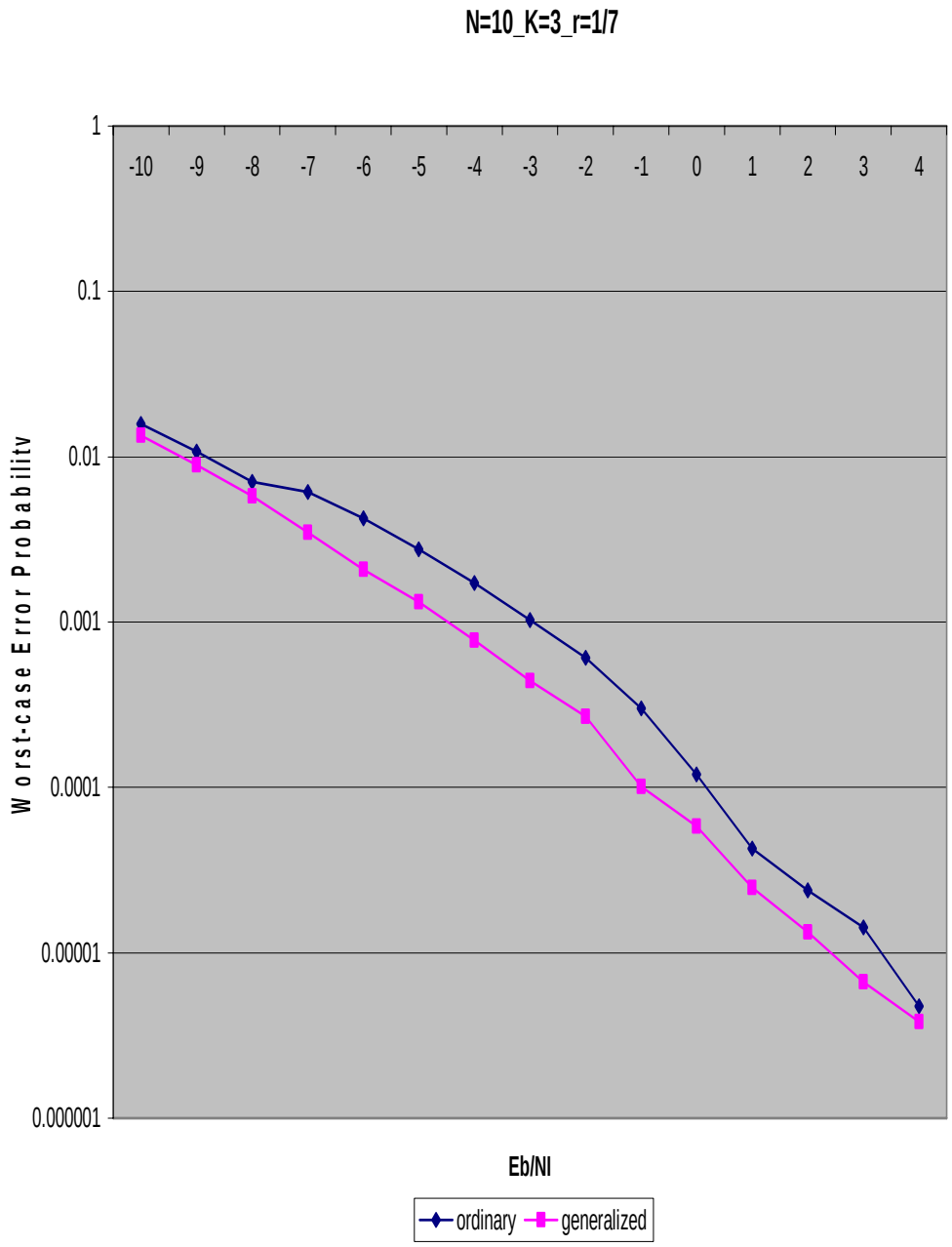

Figure 6.28: Convolutional Interleaver

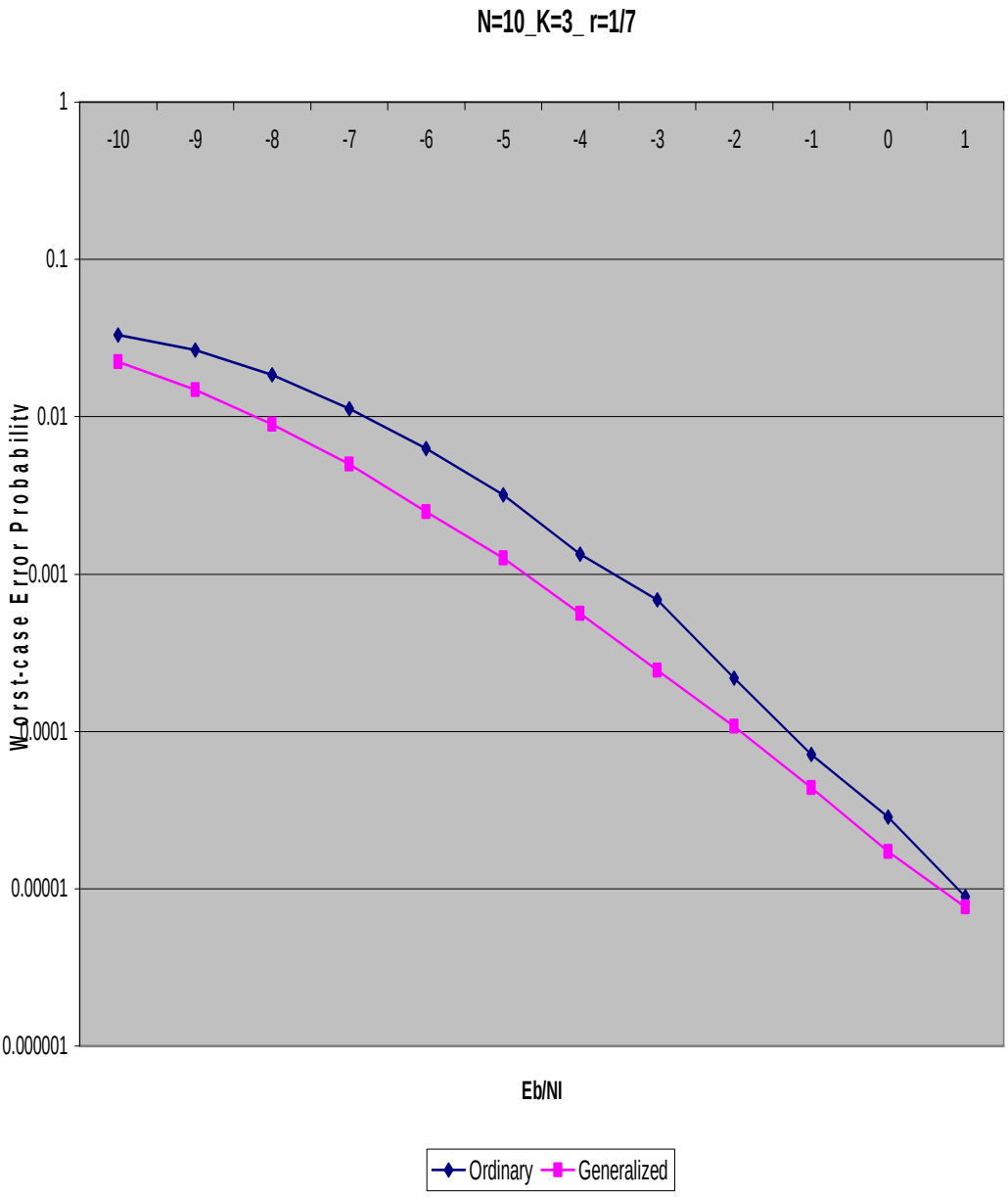

Figure 6.29: Random Interleaver

The codes shown in Figure 6.30 and 6.31 have constraint length of K = 3 and code rate of r = 1/2 respectively. Comparing the performance at 10^{-3} level, we get Table XIX.

TABLE XIX: VARYING INTERLEAVER FOR K = 3 AND R = 1/2

Code	Interleaver	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.30	Convolutional	-0.5	0.5	1
Figure 6.31	Random	-1.5	-1.0	0.5

The codes shown in Figure 6.32 and 6.33 have constraint length of K = 5 and code rate of r = 1/2 respectively. Comparing the performance at 10^{-3} level, we get Table XX.

TABLE XX: VARYING INTERLEAVER FOR K = 5 AND R = 1/2

Code	Interleaver	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.32	Convolutional	-0.6	0.3	0.9
Figure 6.33	Random	-0.8	-0.1	0.7

The codes shown in Figure 6.34 and 6.35 have constraint length of K = 7 and code rate of r = 1/2 respectively. Comparing the performance at 10^{-3} level, we get Table XXI.

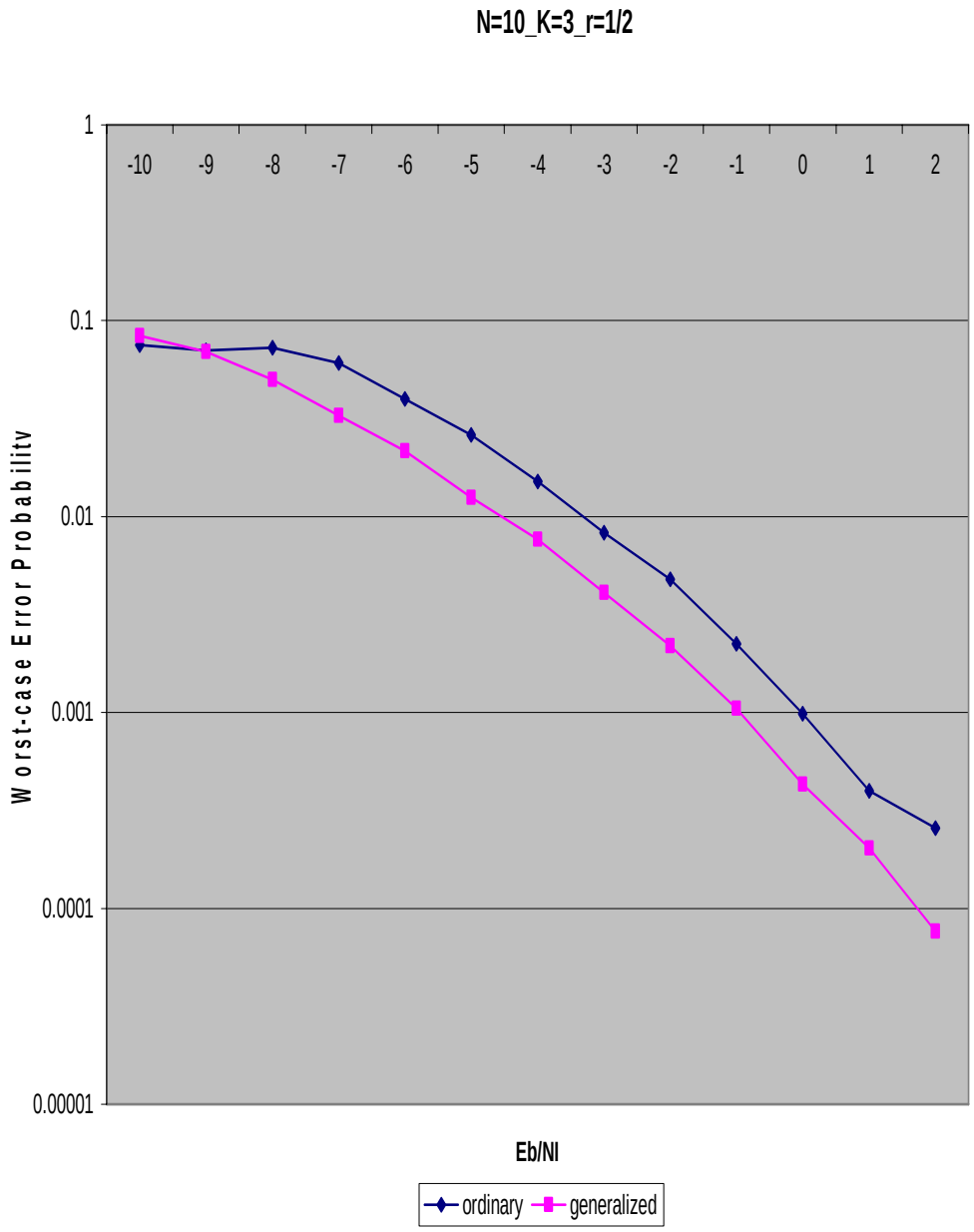

Figure 6.30: Convolutional Interleaver

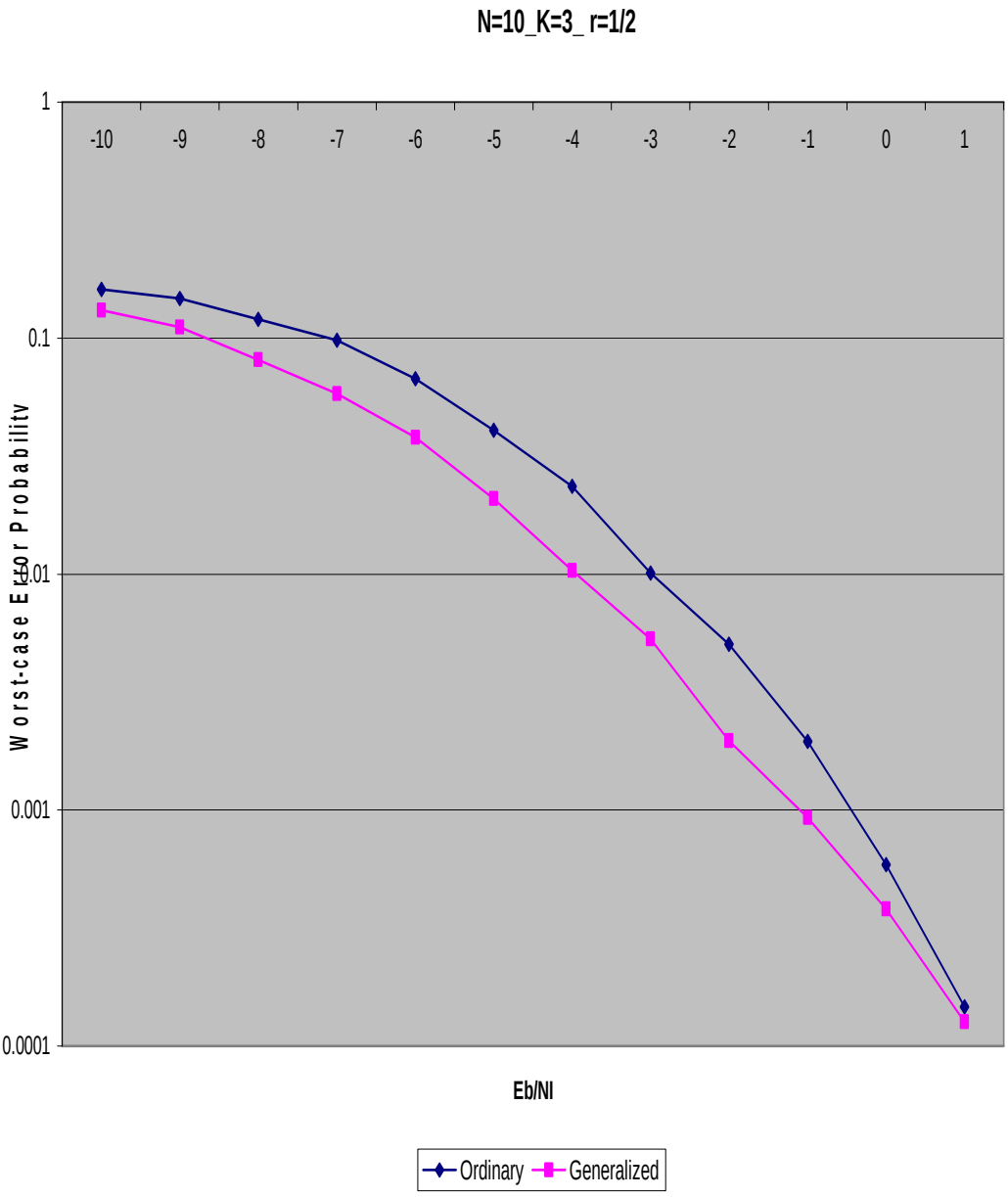

Figure 6.31: Random Interleaver

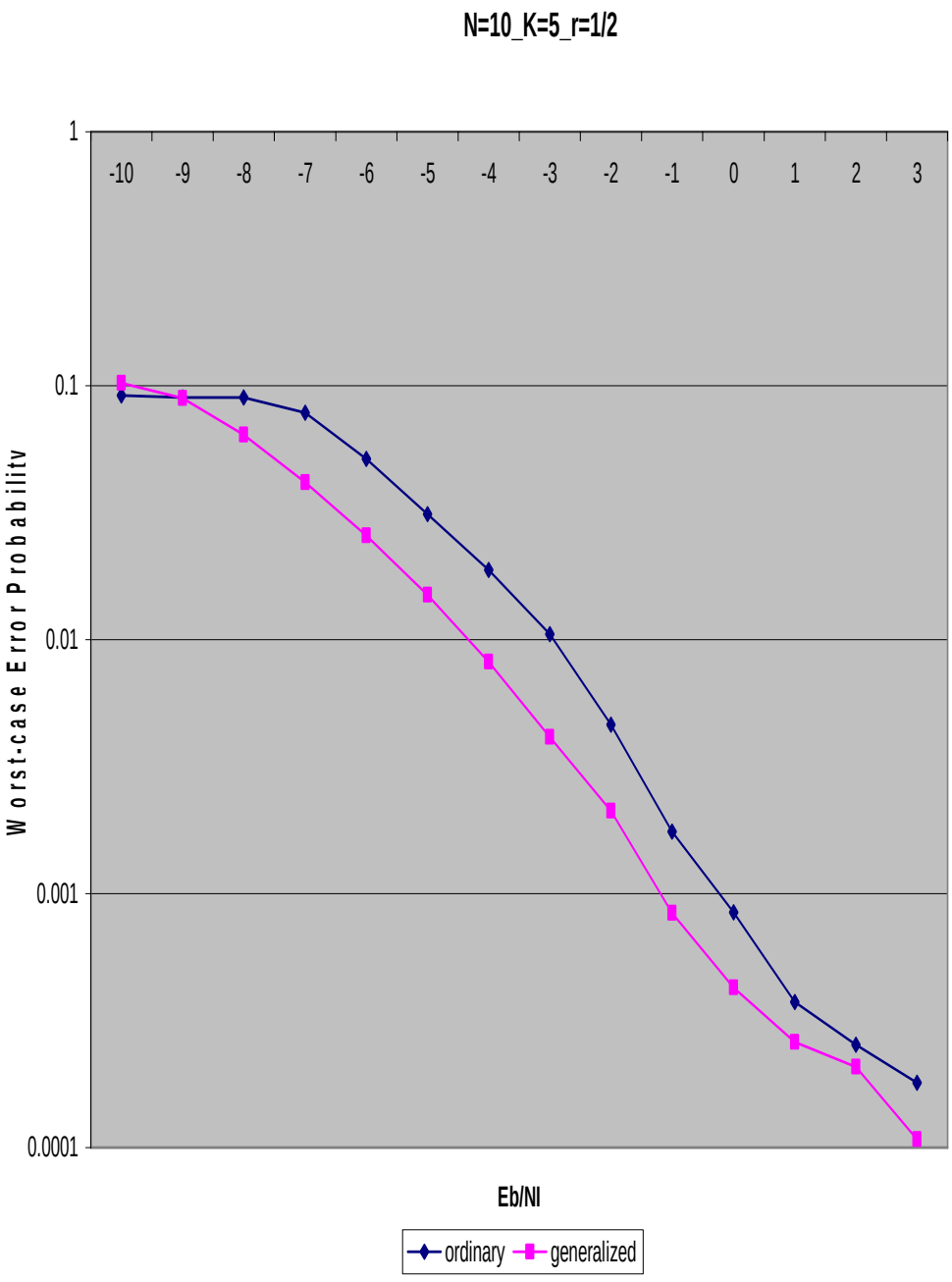

Figure 6.32: Convolutional Interleaver

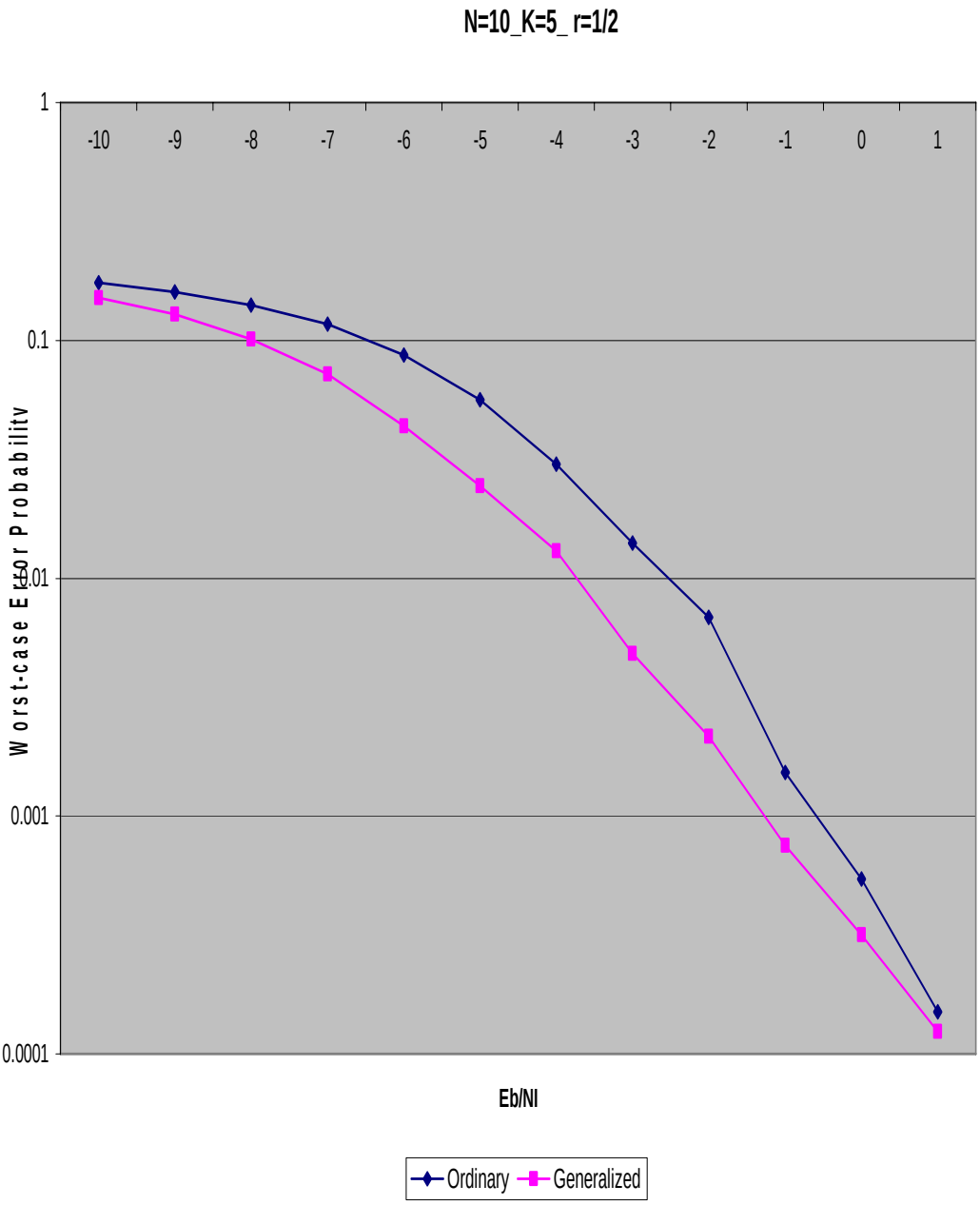

Figure 6.33: Random Interleaving

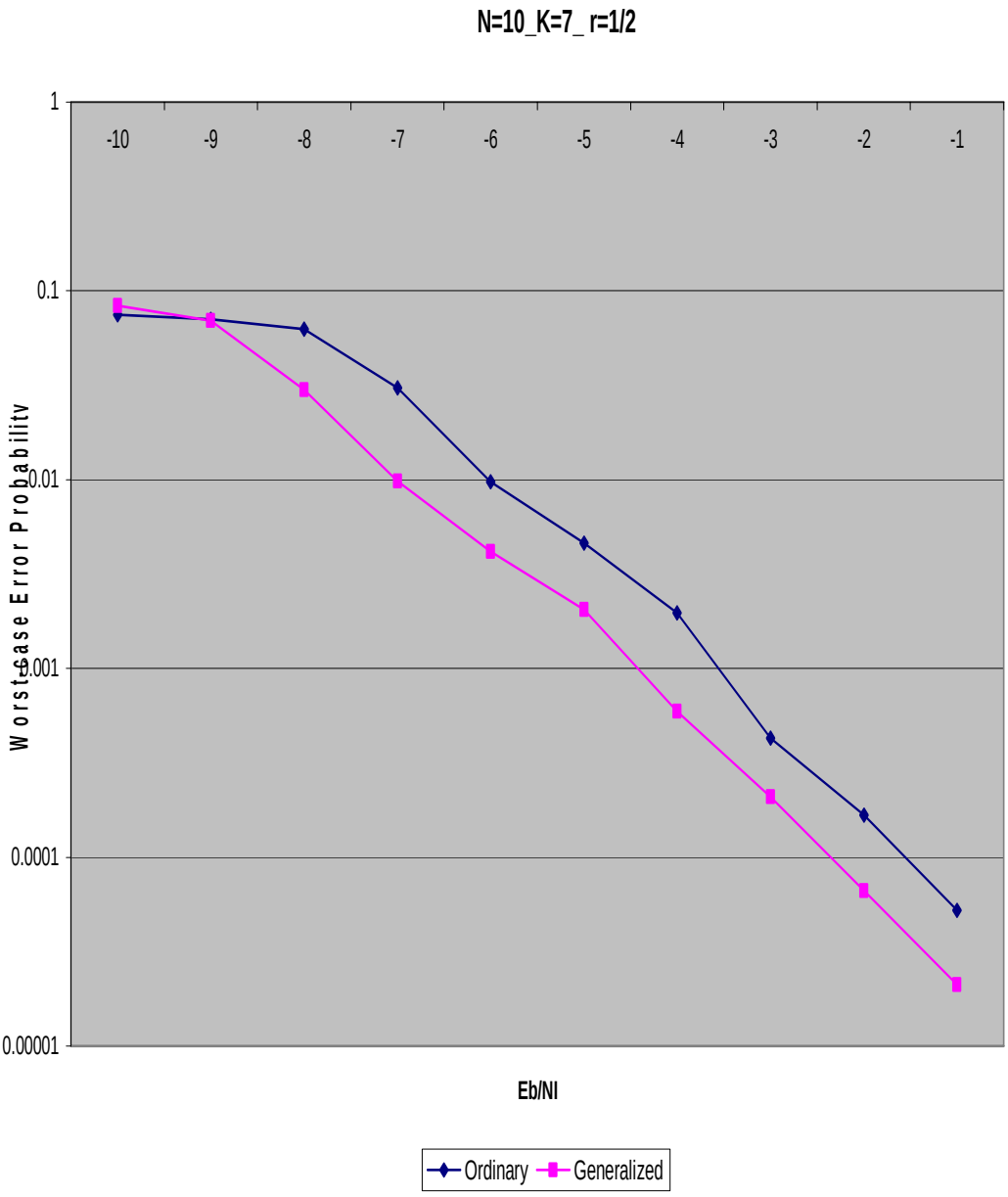

Figure 6.34: Convolutional Interleaver

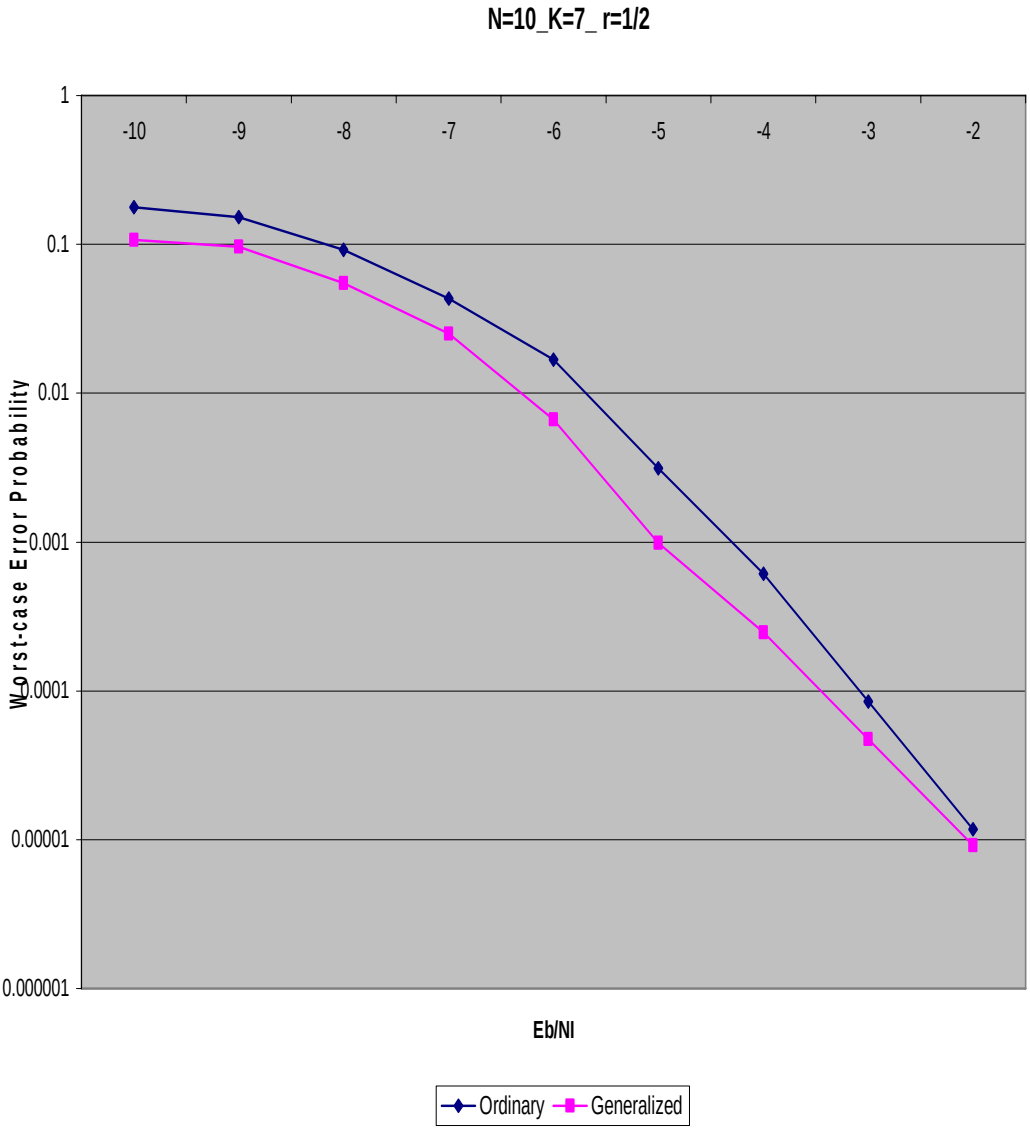

Figure 6.35: Random Interleaving

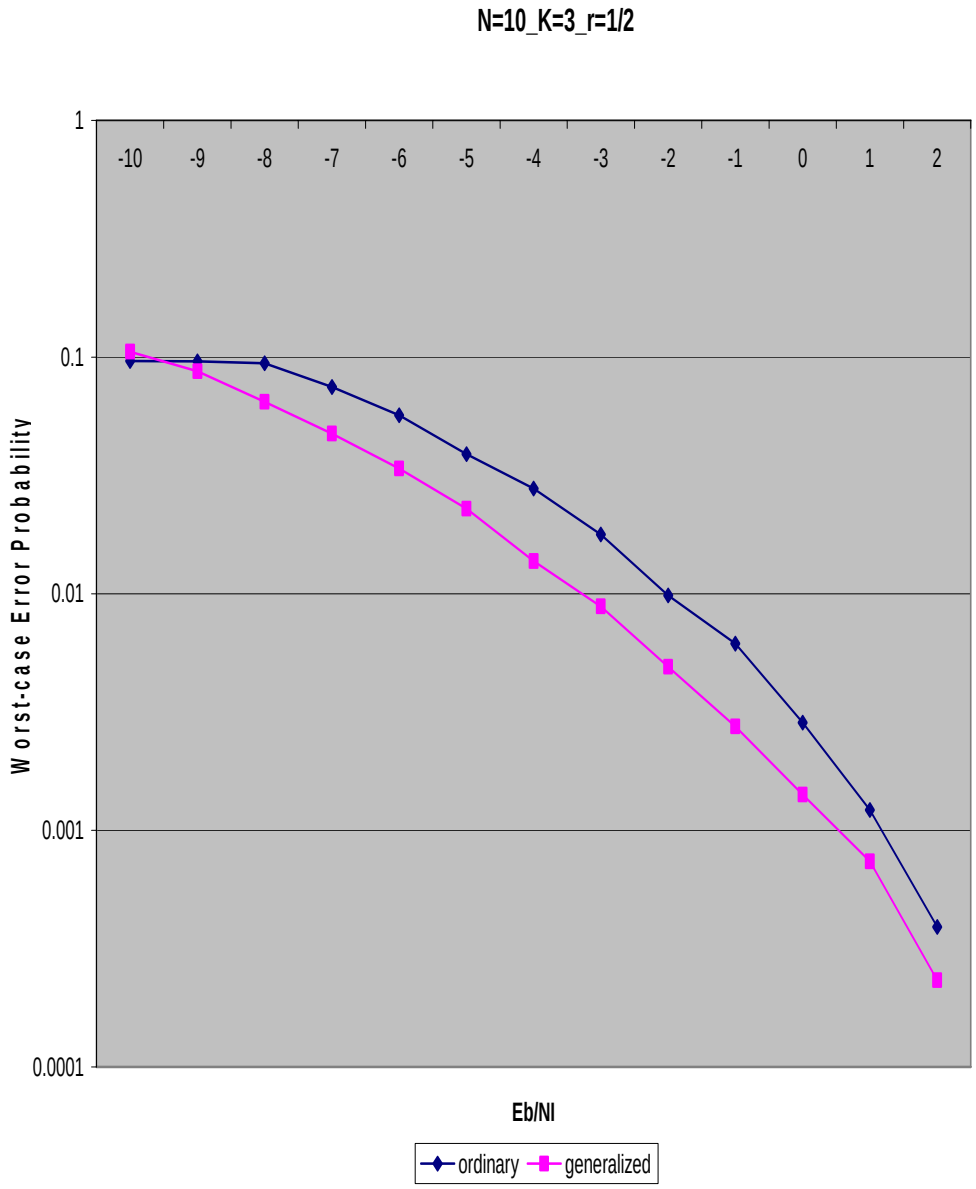

Figure 6.36: Decoding depth = 3K

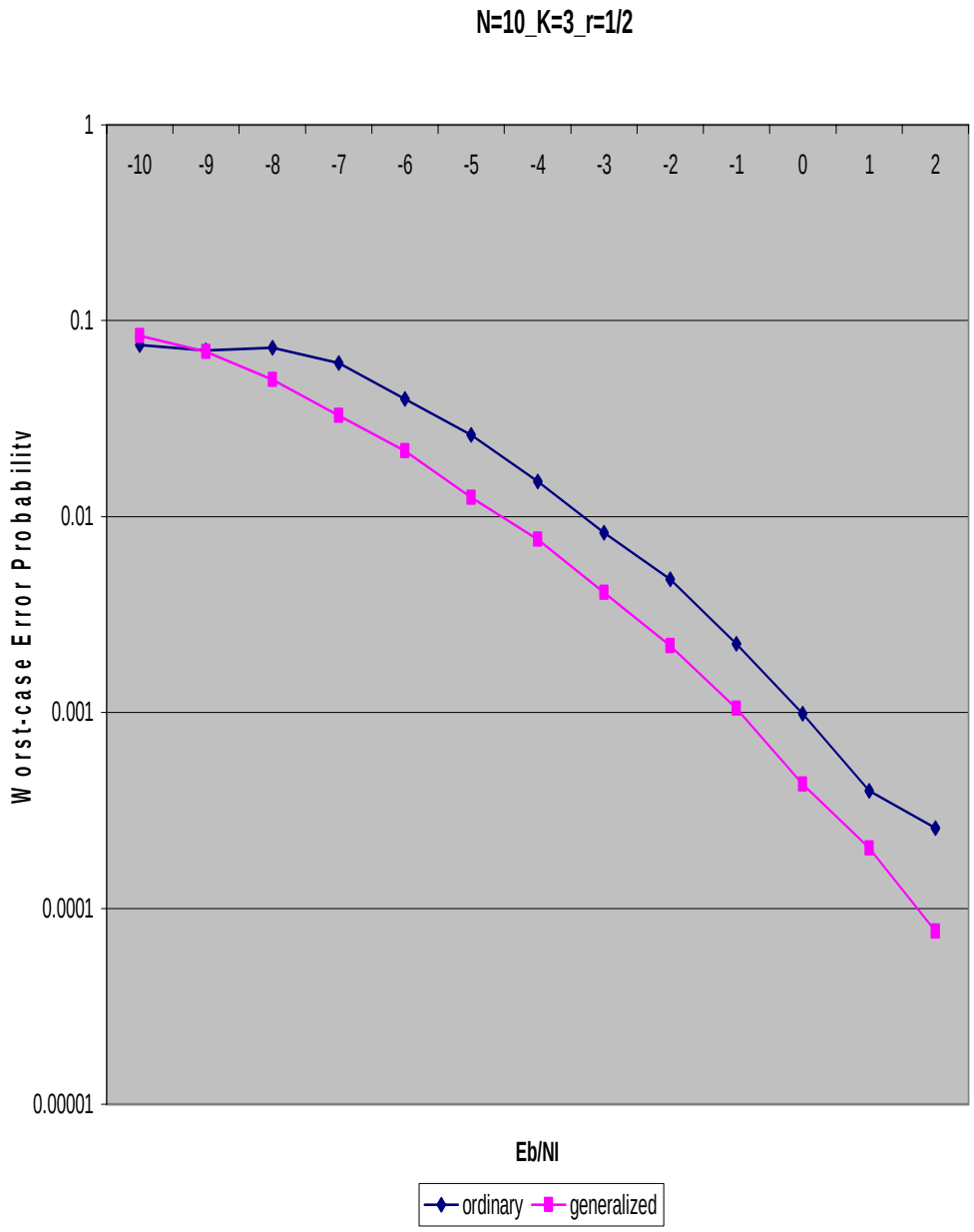

Figure 6.37: Decoding Depth = 5*K*

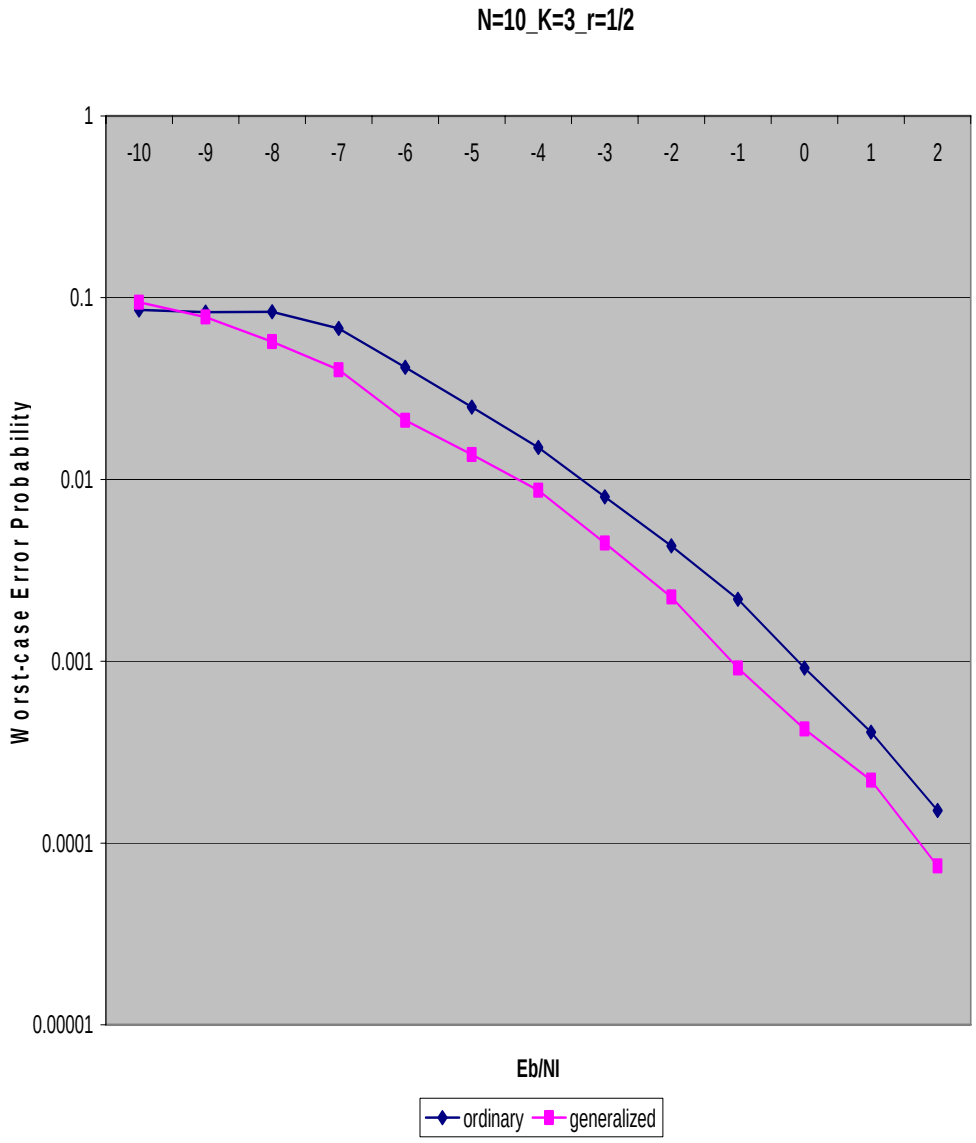

Figure 6.38: Decoding Depth = 7*K*

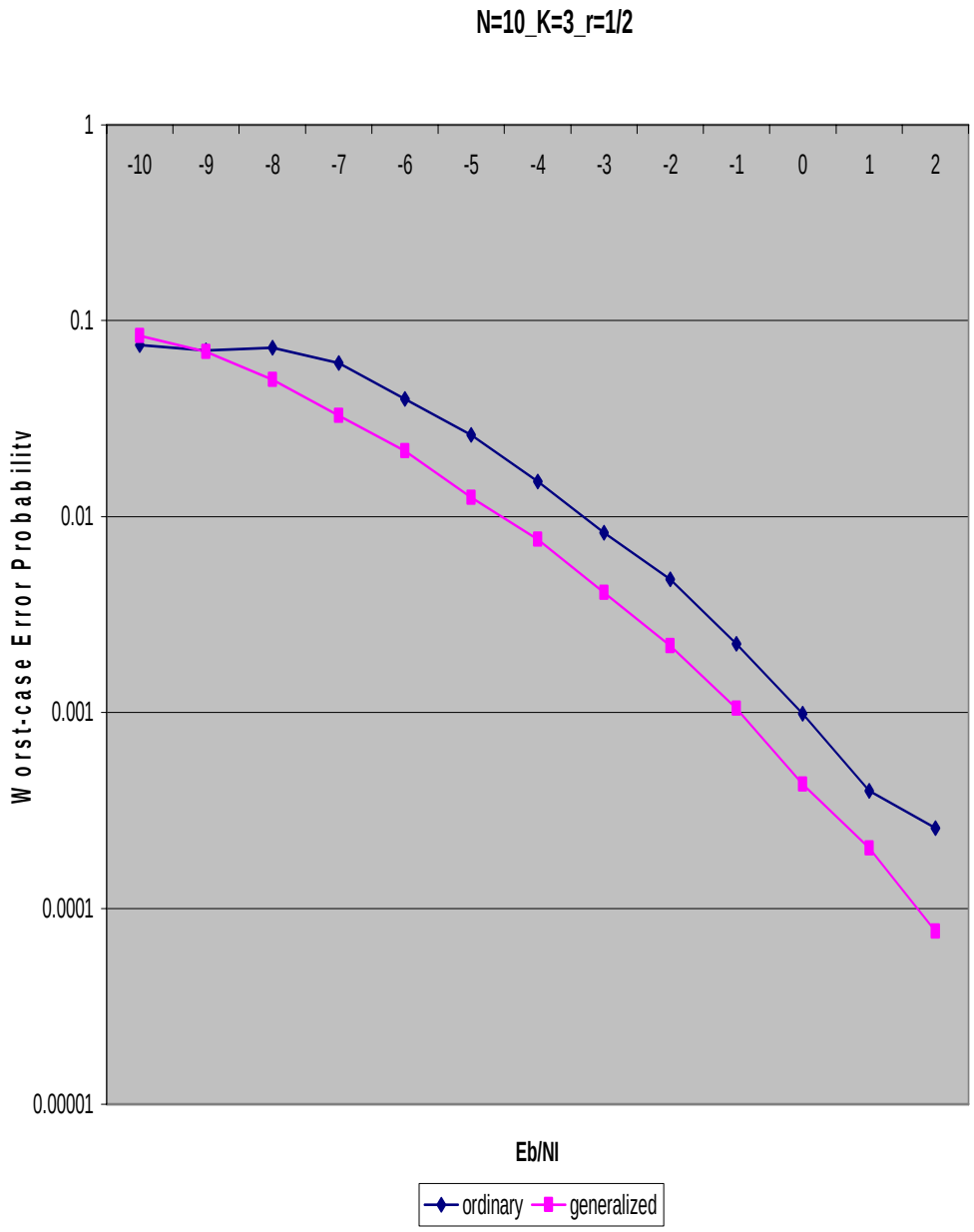

Figure 6.39: Convolutional Interleaver rows = 5

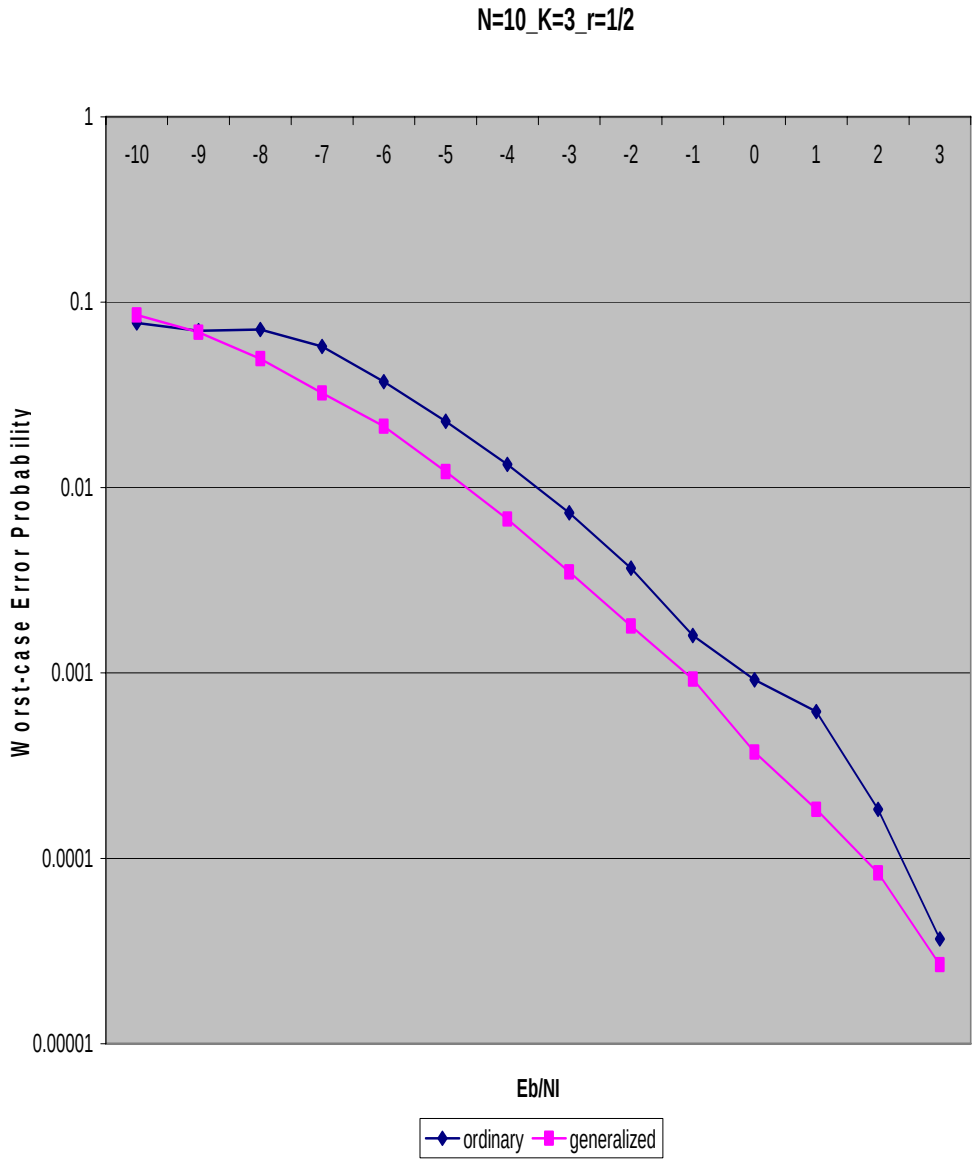

Figure 6.40: Convolutional Interleaver Rows = 8

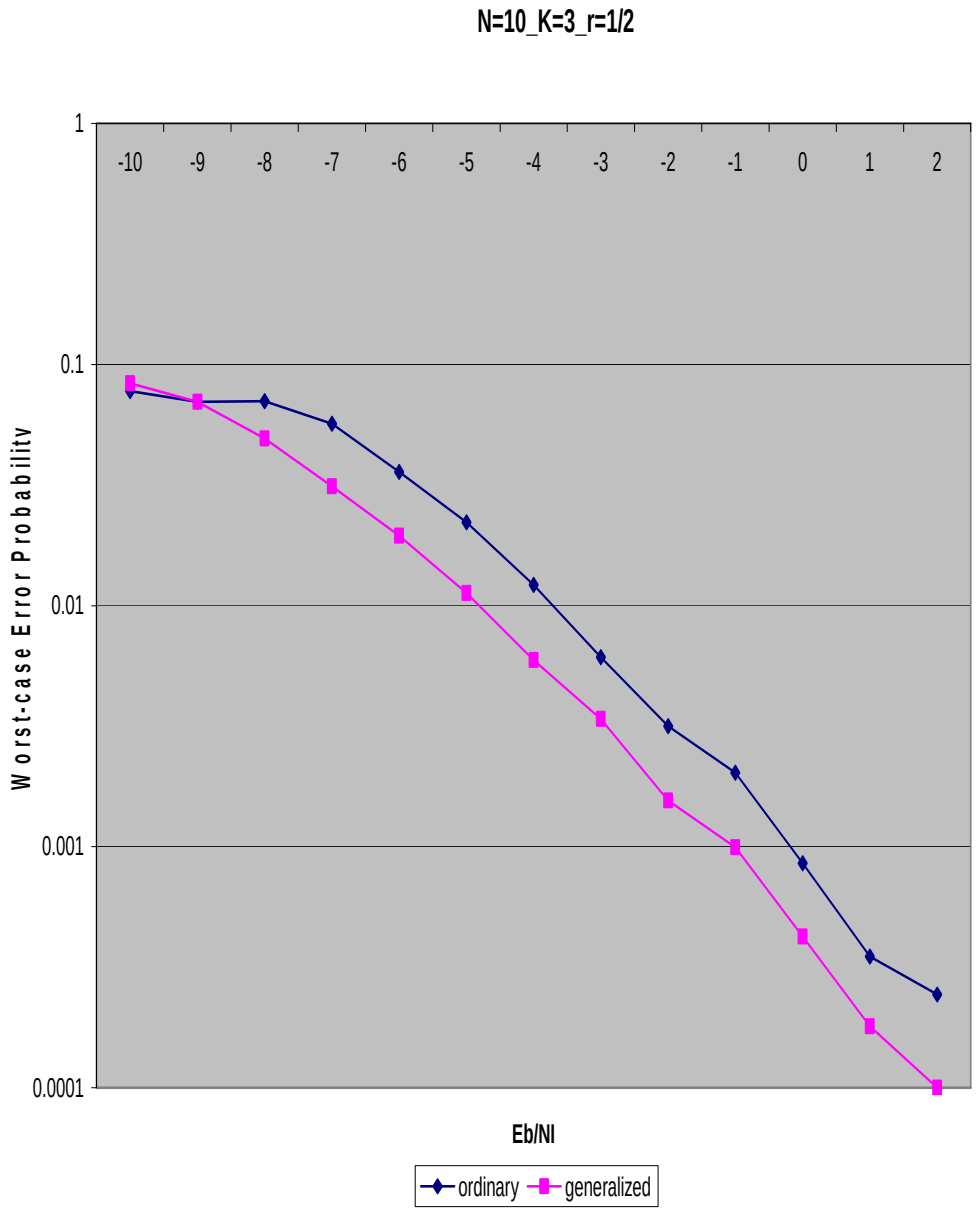

Figure 6.41: Convolutional Interleaver Rows = 10

TABLE XXI: VARYING INTERLEAVER FOR $K = 7$ AND $R = 1/2$

Code	Interleaver	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.34	Convolutional	-3.9	-3.1	0.8
Figure 6.35	Random	-4.4	-3.8	0.6

From the results in this section it is observed that for constant constraint length and code rate random interleaver performs better than convolutional interleaver, at the same time the difference between two systems decreases from convolutional to random. The first observation suggests that randomization helps with robustness. This is well inline with the arguments presented for the optimal random modem for robust communication [1].

6.5 Same Code Rate and Constraint Length with Varying Decoder Depth

In this section we compare codes with same code rate and same constraint length with varying decoding depth of Viterbi decoder. All the codes used in this section have a chip length of $N = 10$ and use convolutional Interleaving with number of rows as 5.

The codes shown in Figure 6.36, 6.37 and 6.38 have constraint length of $K = 3$ and code rate of $r = 1/2$. They have decoding depth of $3K$, $5K$ and $7K$ respectively. Comparing the performance at 10^{-3} level, we get table XXII.

TABLE XXII: VARYING DECODING DEPTH

Code	Decoding Depth	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.36	3*K*	1	1.7	0.7
Figure 6.37	5*K*	-0.5	0.5	1
Figure 6.38	7*K*	-0.7	0.4	1.1

From the results in this section it is observed that with same constraint length and same code rate, as decoding depth increases performance improves, at the same time the difference between two systems also increases. The first observation is expected since a Viterbi decoder performs well with an increase in its decoding depth. It is also interesting to see that generalized spread spectrum becomes increasingly more beneficial as interleaving depth is increased.

6.6 Same Code Rate, Constraint Length with Varying Interleaver Rows

In this section we compare codes with same code rate and same constraint length with varying rows of convolutional Interleaver. All the codes used in this section have a chip length of $N = 10$ and decoding depth of $5K$.

The codes shown in Figure 6.39, 6.40 and 6.41 have constraint length of $K = 3$, code rate of $r = 1/2$ and use convolutional Interleaving with number of rows as 5, 8 and 10 respectively. Comparing the performance at 10^{-3} level, we get Table XXIII.

TABLE XXIII: VARYING CONVOLUTIONAL INTERLEAVER ROWS

Code	Interleaver Rows	Generalized (dB)	Ordinary (dB)	Gain
Figure 6.39	5	-0.5	0.5	1
Figure 6.40	8	-0.5	0.4	0.9
Figure 6.41	10	-0.5	0.3	0.8

From the results in this section it is observed that with same constraint length and same code rate as number of rows of convolutional interleaver increases performance improves, at the same time the difference between two systems decreases. The first observation is expected since with increasing number of rows the interleaver works better, behaving more like a random interleaver. The second observation also makes sense since generalized DSSS becomes increasingly more beneficial as other parameters of the system (in this case, the number of rows) are varied to make the system less robust.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Simulation results of the worst-case performance of ordinary and generalized DSSS show that generalized DSSS performs consistently better than ordinary DSSS. This observation has been well known and studied. Other observations are:

1. Performance of codes with same constraint length improves as code rate decreases, at the same time the difference between two systems increases. It is expected that decreasing code length (increasing redundancy) would result in better performance. It is interesting to also see that the difference between ordinary and generalized DSSS increases with decreasing code length, as stipulated by Hizlan [3].
2. Performance of codes with same code rate improves as constraint length increases, at the same time the difference between two systems decreases. Again, it is expected that larger constraint lengths produce better results. It is interesting to see that the difference between the two systems gets smaller with increasing

constraint length. This would suggest that generalized DSSS becomes increasingly more beneficial as coding memory is decreased.

3. Performance of codes with same constraint length and same code rate improves as chip length increases, at the same time the difference between two systems decreases. Again, the first observation here is obvious. The second observation suggests that generalized DSSS becomes increasingly more beneficial compared to ordinary DSSS as other parameters of the communication system (in this case *N*) are varied to make it less robust.

4. Performance of codes with same constraint length and same code rate is better with random interleaver than convolutional interleaver, at the same time the difference between two systems decreases from convolutional to random interleaving. The first observation suggests that randomization helps with robustness. This is well inline with the arguments presented for the optimal random modem for robust communication [1].

5. Performance of codes with same constraint length and same code rate improves as the decoding depth of Viterbi decoder increases, at the same time the difference between two systems increases. The first observation is expected since a Viterbi decoder performs well with an increase in its decoding depth. It is also interesting to see that generalized spread spectrum becomes increasingly more beneficial as interleaving depth is increased.

6. Performance of codes with same constraint length and same code rate improves as number of rows of convolutional Interleaver increases, at the same time the difference between two systems decreases. The first observation is expected since

with increasing number of rows the interleaver works better, behaving more like a random interleaver. The second observation also makes sense since generalized DSSS becomes increasingly more beneficial as other parameters of the system (in this case, the number of rows) are varied to make the system less robust.

Looking at these results as a whole, we can say that generalized DSSS increasingly outperforms ordinary DSSS as code rate is decreased. This result confirms the conjecture in [3]. We also see that generalized DSSS increasingly outperforms ordinary DSSS as other parameters of the spread spectrum system are varied to make it less robust. This suggests that generalized DSSS is increasingly more beneficial as the conditions worsen. Furthermore, we see that random interleaving is better than convolutional interleaving for robust communications.

As future work, it is suggested that the worst-case performance of further generalized (five-level) DSSS with convolutional codes can be evaluated as an extension.

REFERENCES

1. B. L. Hughes and M. Hizlan, "An Asymptotically Optimal Random Modem and Detector for Robust Communication," IEEE Trans. Inform. Theory, vol 36, pp. 810-821, July 1990.

2. Dr. Murad Hizlan, *Cleveland State University*, "A Generalization of Direct-Sequence Spread-Spectrum". Submitted for publication.

3. M. Hizlan and B. L. Hughes, "Worst-Case Error Probability of a Spread Spectrum System in Energy Limited Interference," IEEE Trans. Comm.., vol 39, pp. 1193-1196, August 1991.

4. Manohar Vellala, *Masters Thesis from Cleveland State University 2004*, "Coded Generalized Direct-Sequence Spread-Spectrum with Specific Codes".

5. Ranga Kalakuntla, *Masters Thesis from Cleveland State University 2004*, "Further Generalized Direct-Sequence Spread-Spectrum".

6. Hariharan Ramaswamy, *Masters Thesis from Cleveland State University 2004*, "Generation of Generalized Signature Sequences".

7. Bernard Skalar, Digital communications: Fundamentals and Applications. Pearson Education 2002.

8. Shu Lin and Daniel J. Costello, Error Control Coding: Fundamentals and Applications, Prentice-Hall 1983.

9. Claude Shannon, A Mathematical Theory of Communication, Bell System Tech. J. 27 (1948) 379–423, 623–656.

10. A. J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," IEEE Transactions on Information Theory , vol. IT-13, April, 1967, pp. 260-269.

11. Michel C. Jruchim, Pilip Balaban, and K. Sam Shanmugan, Simulation of Communication Systems, Applications of Communication Theory. Plenum press, 1992.

12. William H Press, Brian P Flannery, Saul A Teukolsky, Numerical Recipes in C, Cambridge University Press. 1998.

13. H.L. Van Trees, Detection Estimation and Modulation Theory. Wiley, 1968.

14. J. M Wozencraft and I.M Jacobs, Principles of Communication Engineering. Wiley 1965.

15. Marvin K. Simon, Jim k. Omura, Robert A. Schotz and Barry K. Levitt, Spread Spectrum Communications Hand Book. McGraw-Hill Inc 1994.

16. R. Ziemer, R. Peterson and D. Borth, Introduction to Spread Spectrum Communications, Prentice Hall, 1995.

APPENDICES

APPENDIX A

/* This program computes the Monte Carlo simulation of generalized DSSS based on a canonical distribution of arbitrary interference using convolutional encoder and Viterbi decoder */

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>
#include <limits.h>
#define MAX_SIZE 50
#define TWOTOTHEM 4
#define MAXMETRIC 128
#define MAXINT 16384

void generateSequence(int dataLength,int *dataSequence);
void multiplyAndIntegrate(int dataLength,int codeLength,int
**codeSequence,double **signalPlusNoise,double
*decodedSequence);
void multiply(int dataLength,int codeLength,int
*dataSequence,double **codeSequence,double
**multipliedSequence);
void addSignalAndNoise(int dataLength, int codeLength,
double **multipliedSequence, double *noiseArray, double
**signalPlusNoise);
void generateCode(int dataLength,int codeLength,int
**codeArray);
void generateNoise(int snr,int length,int
*interleaveArray,int D,double *noiseArray);
void convertToSignal(int *dataSequenceAfterCoding,int
dataLength, int *dataSequenceInSignal);
void acceptEquation(int **equationSet,int k, int rate);
int nextNumber();
void compareAndDecide(int dataLength,double
*decodeSequence,int *sequenceAfterDeciding);
int getErrors(int dataLength,int *originalSequence,int
*receiveSequence);
double calculateBER(long totalNoOfBits,long
totalNoOfErrors);
void vd(int g[2][K],float es_ovr_no,float es_ovr_ni,long
channel_length,float *channel_output_vector,int
*decoder_output_matrix);
```

```
void conv_encoder(int g[2][K], long data_len, int
*in_array, int *out_array);
int quant(float channel_symbol);
int soft_metric(int data, int guess);
void increasePower(int **inputSequence,double
**outputSequence);
long int idum;

int main()
{

  int **equationSet;
  int dataSequenceArray[MAX_SIZE];
  int dataSequenceAfterCodingArray[MAX_SIZE];
  int dataSequenceInSignalArray[MAX_SIZE];
  double decodedSequenceArray[MAX_SIZE];
  int sequenceAfterDecisionArray[MAX_SIZE];
  int originalDecodedDataArray[MAX_SIZE];
  int interleaveArray[MAX_SIZE];
  int codedDataSequence[MAX_SIZE];
  int codeSequenceArray[MAX_SIZE][MAX_SIZE];
  double multipliedSequenceArray[MAX_SIZE][MAX_SIZE];
  double generalizedCodeSequenceArray[MAX_SIZE][MAX_SIZE];
  double noiseSequenceArray[MAX_SIZE];
  double signalPlusNoiseArray[MAX_SIZE][MAX_SIZE];
  int sequenceAfterDeConvolution[MAX_SIZE];
  long noOfErrors,noOfBits;
  int codeLength,rate =0, dataLength=0, rows,snr,D,loops;
  dataLength = 5K;
  codeLength = N;
  rate = r;
  double worstBER,ber;
    #if K == 3        /* polynomials for K = 3 */
    int g[2][K] = {{1, 1, 1},
                   {1, 0, 1}};
    #endif

    #if K == 5        /* polynomials for K = 5 */
    int g[2][K] = {{1, 1,  1, 0, 1},
                   {1, 0,  0, 1, 1}};
    #endif

    #if K == 7        /* polynomials for K = 7 */
    int g[2][K] = {{1,  1, 1, 1,  0, 0, 1},
                   {1,  0, 1, 1,  0, 1, 1}};
    #endif
```

```
    #if K == 9        /* polynomials for K = 9 */
    int g[2][K] = {{1, 1, 1,  1, 0, 1,  0, 1, 1},
                   {1, 0, 1,  1, 1, 0,  0, 0, 1}};
    #endif

/* Input parameters */

     printf("Enter constraint length K [int]:   ");
     scanf("%d", &K);
     getchar();*/
     printf("Enter number of PN generator chips per bit N
[int]:   ");
     scanf("%d", &N);
     getchar();
        printf("Enter code rate [int]:   ");
     scanf("%d", &r);
     getchar();
     printf("Enter interleaver seed for interleaving
[negative long int]:   ");
     scanf("%ld", &idum);
     getchar();

     for (snr = 0; snr < 30;  snr++)  {

          worstBER=0;
          for( D = 0; D<5K*rate*N;D++ ) {

               for( loops =0; loops <10000; loops++)   {

generateSequence(dataLength, dataSequenceArray);

conv_encoder(g,dataLength,dataSequenceArray,codedDataSequen
ce);

convertToSignal(codedDataSequence, dataLength,
dataSequenceInSignalArray);

generateCode(dataLength*rate, codeLength, (int
**)codeSequenceArray);

increasePower((int **)codeSequenceArray,(double
**)generalizedCodeSequenceArray);

multiply(dataLength*rate, codeLength,
dataSequenceInSignalArray, (double
**)generalizedCodeSequenceArray, (double
**)multipliedSequenceArray);
```

```
generateNoise(snr, dataLength*rate*codeLength,
interleaveArray, D, noiseSequenceArray);


addSignalAndNoise(dataLength*rate, codeLength, (double
**)multipliedSequenceArray, noiseSequenceArray, (double
**)signalPlusNoiseArray);

multiplyAndIntegrate(dataLength*rate, codeLength, (int
**)codeSequenceArray ,(double **)signalPlusNoiseArray,
decodedSequenceArray);

compareAndDecide(dataLength*rate,decodedSequenceArray,seque
nceAfterDecisionArray);

vd(g,snr,snr,dataLength*rate,(float
*)sequenceAfterDecisionArray,(int
*)sequenceAfterDeConvolution);

                    noOfErrors = noOfErrors +
getErrors(dataLength, sequenceAfterDeConvolution,
dataSequenceArray);

                    noOfBits = noOfBits + dataLength*rate;

               }

               ber = calculateBER(noOfBits, noOfErrors);

               if(worstBER < ber)  {
                    worstBER=ber;
               }

               noOfBits=0;
               noOfErrors=0;
          }

          printf("the worst BER for %d snr is %f ",snr,
worstBER);
     }
  return 0;
}

/* random data generation for message */
void generateSequence(int dataLength,int *dataSequence){
     int i;
```

```
     for( i=0; i< dataLength; i++) {
          dataSequence[i]= (int)ran2()%2;
     }
}

/* at the receiver end multiply and integrate the chips
received from noisy channel */
void multiplyAndIntegrate( int dataLength, int codeLength,
                                   int **codeSequence,
double **signalPlusNoise,
                                   double *decodedSequence
)
{
     int i=0;

     for(i=0; i < dataLength; i++ )     {
          double temp=0;
          int j=0;

          for(j=0; j < codeLength; j++)      {
               temp = temp + (codeSequence[i][j] *
signalPlusNoise[i][j]);
          }

          decodedSequence[i] = temp;
     }

}

/* calculate number of zeroes in the generalized sequence*/
void increasePower(int **inputSequence,double
**outputSequence){

        int noOfZeros=0,noOfOnes=0;
        double value=0,addedValue=0;
        int i,j;
        for(i=0;i<DATA_LENGTH;i++){

            noOfZeros=0;

            for(j=0;j<CODE_LENGTH;j++){

                if(inputSequence[i][j]==0) noOfZeros++;

            }
```

```
            noOfOnes=CODE_LENGTH-noOfZeros;
            value=sqrt((double)CODE_LENGTH/noOfOnes);
            addedValue=value-1;
            for(j=0;j<CODE_LENGTH;j++){

                if(inputSequence[i][j]==1)
outputSequence[i][j]=inputSequence[i][j]+addedValue;

                else if(inputSequence[i][j]==-1)
outputSequence[i][j]=inputSequence[i][j]-addedValue;

            }
        }
    }

/* multiply symbols and chips at the transmitter end */
void multiply(int dataLength, int codeLength, int
*dataSequence,
                 double **codeSequence, double
**multipliedSequence)    {
    int i,j;

    for(i = 0; i < dataLength; i++)    {
         for(j = 0; j < codeLength; j++)    {

    multipliedSequence[i][j]=codeSequence[i][j]*dataSequen
ce[i];
         }
    }
}

/* add noise to signal */
void addSignalAndNoise(int dataLength, int codeLength,
double **multipliedSequence,
             double *noiseArray, double **signalPlusNoise)
    {
    int n=0,i=0,j;

    for(i = 0; i < dataLength; i++)    {
         for(j = 0; j < codeLength; j++)    {

               signalPlusNoise[i][j] = noiseArray[n] +
multipliedSequence[i][j];
               n++;
         }
    }
```

```
}

/* generate code symbols */
void generateCode(int dataLength, int codeLength, int
**codeArray)   {
     int i = 0,j,temp=0;

     for(i = 0;i < dataLength; i++)     {
          for(j=0;j<codeLength;j++)     {

               temp=(int)ran2%3;

                if(temp==0) codeArray[i][j]=0;

                else if(temp==1) codeArray[i][j]=-1;

                else codeArray[i][j]=1;
          }
     }
}

/* generate noise */
void generateNoise(int snr, int length, int
*interleaveArray, int D,
                              double *noiseArray)   {
     int i = 0;
     for( i = 0; i < D; i++)  {

          noiseArray[interleaveArray[i]] =
sqrt(length/(pow(10,(double)(snr-10)/10)*D));

     }
}

/* convert to baseband signals */
void convertToSignal(int *dataSequenceAfterCoding, int
dataLength, int *dataSequenceInSignal){
     int i = 0;

     for(i = 0; i < dataLength;i++)     {

          if(dataSequenceAfterCoding[i] == 0) {
               dataSequenceInSignal[i] = -1;
          }
          else {
               dataSequenceInSignal[i] = 1;
          }
```

```
      }

}

int nextNumber(){

    //return random number
    return 0;
}

/* generator polynomials of encoder */
void acceptEquation (int **equationSet, int k, int rate) {
     int i,j;
     for(i = 0; i < rate; i++)     {

          for(j = 0; j < k; j++)   {

          printf("Enter the value of row %d and column %d
",i,j);
          scanf("%d",&equationSet[i][j]);
          }
     }
}

/* hard decision *
void compareAndDecide(int dataLength, double
*decodeSequence, int *sequenceAfterDeciding)  {
     int i;

     for( i = 0;  i < dataLength; i++) {

          if(decodeSequence[i] < 0)     {
          sequenceAfterDeciding[i] = 0;
          }
          else  {
          sequenceAfterDeciding[i] = 1;
          }
     }
}

/* calculate errors */
int getErrors(int dataLength,int *originalSequence,int
*receiveSequence){
     long result = 0;
     int i;

     for(i = 0; i < dataLength; i++)    {
```

```
          if(originalSequence[i] != receiveSequence[i] )
     {
               result++;
          }
     }

     return result;

}

/* final ber calculation*/
double calculateBER(long totalNoOfBits,long
totalNoOfErrors){

        double result=0;

        result = totalNoOfErrors/totalNoOfBits;

        return result;

    }

/* convolutional encoder*/
void conv_encoder(int g[2][K], long data_len, int
*in_array, int *out_array)
{
    int m= K-1;                /* K - 1 */
    long t,S;                  /* bit time, symbol time */
    int j, k;                  /* loop variables */
    int *unencoded_data;       /* this is the pointer to
data array */
    int shift_reg[K];          /* the encoder shift
register */
    int sr_head;               /* index to the first entry
in the sr */
    int a,b;            /* the upper and lower xor gate
outputs */

   long channel_length = ( data_len + m ) * 2;

    /* allocate space for the zero-padded input data array
*/
   unencoded_data = (int
*)malloc((data_len+m)*sizeof(int));
    if (unencoded_data == NULL) {
        printf("\n conv_encoder.c: Can't allocate enough
memory for unencoded data!  Aborting...");
```

```
        exit(1);
    }

    //unencoded_data = in_array ;
    /* read the input data and store it in the array */
   for (t = 0; t < data_len; t++)
        *(unencoded_data + t) = *(in_array + t);

    /* now zero-pad the end of the data */
    for (t = 0; t < m; t++) {
        *(unencoded_data + data_len + t) = 0;
    }

    /* Initializing the shift register */
    for (j = 0; j < K; j++)
 {
        shift_reg[j] = 0;
    }

   /* In order to speed things up a little, the shift
register will be operated
      as a circular buffer, so it needs a head
pointer.we'll just be overwriting the oldest entry with the
new data. */

    sr_head = 0;

    /* initializing the channel symbol output index */
    S = 0;

    /* Here the encoding process begins */
    /* now compute the upper and lower modulo-two adder
outputs, one bit at a time */
    for (t = 0; t < data_len + m; t++)
 {
    shift_reg[sr_head] = *( unencoded_data + t );
     a = 0;
     b = 0;
        for (j = 0; j < K; j++)
     {
         k = (j + sr_head) % K;
         a ^= shift_reg[k] & g[0][j];
         b ^= shift_reg[k] & g[1][j];
        }

      /* write the upper and lower xor gate outputs as
channel symbols */
```

```
     *(out_array + S) = a;
        S = S + 1;
       // printf(" %d\n",a);

     *(out_array + S) = b;
        S = S + 1;
       // printf("%d\n",b);

        sr_head -= 1;    /* This is equivalent to shifting
everything right one place */
        if (sr_head < 0) /* we need to make sure that the
pointer modulo K is adjusted */
            sr_head = m;
    }
/* now transform the data from 0/1 to +1/-1 */
    for (t = 0; t < channel_length; t++)
     {

        /*if the binary data value is 1, the channel symbol
is -1; if the
        binary data value is 0, the channel symbol is +1.
*/
        *(out_array+t) = 1 - 2 *  *( out_array + t );

      // printf("%d\n",*( out_array + t ));

     }
/*now the dynamically allocated array is made free */
      free(unencoded_data);

}

/* viterbi decoder */
void vd(int g[2][K],float es_ovr_no,float es_ovr_ni,long
channel_length,float *channel_output_vector,int
*decoder_output_matrix)
{
     int i, j, l, ll;                          /* loop
variables */
    long t;                                   /* time */
     int memory_contents[K];                   /* input +
conv. encoder sr */
     int input[TWOTOTHEM][TWOTOTHEM];          /* maps
current/nxt sts to input */
     int output[TWOTOTHEM][2];                 /* gives
conv. encoder output */
```

```
     int nextstate[TWOTOTHEM][2]; /* for current st, gives
nxt given input */

     int accum_err_metric[TWOTOTHEM][2]; /* accumulated
error metrics */

     int state_history[TWOTOTHEM][K * 5 + 1];  /* state
history table */
     int state_sequence[K * 5 + 1]; /* state sequence list
*/
     int *channel_output_matrix;     /* ptr to input matrix
*/

     int binary_output[2]; /* vector to store binary enc
output */

        int branch_output[2]; /* vector to store trial enc
output */

     int m, n, number_of_states, depth_of_trellis, step,
branch_metric,sh_ptr, sh_col, x, xx, h, hh, next_state,
last_stop; /* misc variables */
     /* n is 2^1 = 2 for rate 1/2 */
     n = 2;

     /* m (memory length) = K - 1 */
     m = K - 1;

     /* number of states = 2^(K - 1) = 2^m for k = 1 */
     number_of_states = (int) pow((double)2, m);

     depth_of_trellis = K * 5;

void deci2bin(int d, int size, int *b);
int bin2deci(int *b, int size);
int nxt_stat(int current_state, int input, int
*memory_contents);
void init_adaptive_quant(float es_ovr_no,float es_ovr_ni);
int quant(float channel_symbol);
int soft_metric(int data, int guess);

     /* initialize data structures */
     for (i = 0; i < number_of_states; i++)
     {
          for (j = 0; j < number_of_states; j++)
               input[i][j] = 0;
          for (j = 0; j < n; j++)
```

```
          {
               nextstate[i][j] = 0;
               output[i][j] = 0;
          }
          for (j = 0; j <= depth_of_trellis; j++)
          {
               state_history[i][j] = 0;
          }
          /* initial accum_error_metric[x][0] = zero */
          accum_err_metric[i][0] = 0;
          /* by setting accum_error_metric[x][1] to MAXINT,
we don't need a flag */
          /* MAXINT is simply the largest possible integer,
defined in values.h */
          accum_err_metric[i][1] = MAXINT;
     }

     /* generate the state transition matrix, output
matrix, and input matrix - input matrix shows how FEC
encoder bits lead to next state */
     for (j = 0; j < number_of_states; j++)
     {
          for (l = 0; l < n; l++)
          {
               next_state = nxt_stat(j, l,
memory_contents);
               input[j][next_state] = l;

          /* now compute the convolutional encoder output
given the current state number and the input value */
               branch_output[0] = 0;
               branch_output[1] = 0;

               for (i = 0; i < K; i++)
               {
                    branch_output[0] ^= memory_contents[i]
& g[0][i];
                    branch_output[1] ^= memory_contents[i]
& g[1][i];
               }

     /* next state, given current state and input */
               nextstate[j][l] = next_state;

               /* output in decimal, given current state
and input */
               output[j][l] = bin2deci(branch_output, 2);
```

```
          } /* end of l for loop */

     } /* end of j for loop */

#ifdef DEBUG
     printf("\nInput:");

     for (j = 0; j < number_of_states; j++)
     {
          printf("\n");

          for (l = 0; l < number_of_states; l++)
               printf("%2d ", input[j][l]);

     } /* end j for-loop */

     printf("\nOutput:");

     for (j = 0; j < number_of_states; j++)
     {
          printf("\n");
          for (l = 0; l < n; l++)
               printf("%2d ", output[j][l]);

     } /* end j for-loop */

     printf("\nNext State:");

     for (j = 0; j < number_of_states; j++)
     {
          printf("\n");

          for (l = 0; l < n; l++)
               printf("%2d ", nextstate[j][l]);

     } /* end j for-loop */

#endif

     channel_output_matrix = (int*) malloc(channel_length *
sizeof(int));

     if (channel_output_matrix == NULL)
     {
          printf(        "\nsdvd.c: Can't allocate memory
for channel_output_matrix! Aborting...");
```

```
          exit(1);
     }

     /* now we're going to rearrange the channel output so
it has n rows, and n/2 columns where each row corresponds
to a channel symbol for a given bit and each column
corresponds to an encoded bit */
     channel_length = channel_length / n;

  /*quantization for specified es_ovr_no*/
   init_adaptive_quant(es_ovr_no,es_ovr_ni);

     /* quantize the channel output--convert float to short
integer */
     /* channel_output_matrix = reshape(channel_output, n,
channel_length) */
     for (t = 0; t < (channel_length * n); t += n)
     {
          for (i = 0; i < n; i++)
               *(channel_output_matrix + (t / n) + (i *
channel_length) ) = quant( *(channel_output_vector + (t +
i) ) );
     } /* end t for-loop */

     /* End of setup. Start decoding of channel outputs
with forward traversal of trellis!
     Stop just before encoder-flushing bits.  */
    for (t = 0; t < channel_length - m; t++)
     {
          if (t <= m)
               /* assume starting with zeroes, so just
compute paths from all-zeroes state */
               step = (int)pow((double)2, m - t * 1);
          else
               step = 1;

          /* set up the state history array pointer for
this time t */
          sh_ptr = (int) ( ( t + 1 ) % (depth_of_trellis +
1) );

          /* repeat for each possible state */
          for (j = 0; j < number_of_states; j+= step)
          {
               /* repeat for each possible convolutional
encoder output n-tuple */
               for (l = 0; l < n; l++)
```

```
               {
                    branch_metric = 0;

/* compute branch metric per channel symbol, and sum for
all channel symbols in the convolutional encoder output n-
tuple */

/* convert the decimal representation of the encoder output
to binary */
                    binary_output[0] = ( output[j][l] &
0x00000002 ) >> 1;
                    binary_output[1] = output[j][l] &
0x00000001;

                    /* compute branch metric per channel
symbol, and sum for all channel symbols in the
convolutional encoder output n-tuple */
                    branch_metric = branch_metric + abs( *(
channel_output_matrix +( 0 * channel_length + t ) ) - 7 *
binary_output[0] ) +
                         abs( *( channel_output_matrix +( 1
* channel_length + t ) ) - 7 * binary_output[1] );

                    /* now choose the surviving path--the
one with the smaller accumlated error metric... */
                    if ( accum_err_metric[ nextstate[j][l]
] [1] > accum_err_metric[j][0] +branch_metric )
                    {
                         /* save an accumulated metric
value for the survivor state */
                         accum_err_metric[ nextstate[j][l]
] [1] = accum_err_metric[j][0] +branch_metric;

                         /* update the state_history array
with the state number of the survivor */
                         state_history[ nextstate[j][l] ]
[sh_ptr] = j;

                    } /* end of if-statement */

               } /* end of 'l' for-loop */

          } /* end of 'j' for-loop -- we have now updated
the trellis */

          /* for all rows of accum_err_metric, move col 2
to col 1 and flag col 2 */
```

```
          for (j = 0; j < number_of_states; j++)
          {
               accum_err_metric[j][0] =
accum_err_metric[j][1];
               accum_err_metric[j][1] = MAXINT;

          } /* end of 'j' for-loop */

          /* now start the traceback, if we've filled the
trellis */
          if (t >= depth_of_trellis - 1)
          {

               /* initialize the state_sequence vector--
probably unnecessary */
               for (j = 0; j <= depth_of_trellis; j++)
                    state_sequence[j] = 0;

               /* find the element of state_history with
the min. accum. error metric */
               /* since the outer states are reached by
relatively-improbable runs of zeroes or ones,
               search from the top and bottom of the
trellis in */
               x = MAXINT;
               for (j = 0; j < ( number_of_states / 2 );
j++)

               {
                    if ( accum_err_metric[j][0] <
accum_err_metric[number_of_states - 1 - j][0] )

                    {
                         xx = accum_err_metric[j][0];

                         hh = j;

                    }
                    else
                    {
                         xx =
accum_err_metric[number_of_states - 1 - j][0];
                         hh = number_of_states - 1 - j;
                    }
                    if ( xx < x)
                    {
                         x = xx;
```

```
                    h = hh;
               }
          } /* end 'j' for-loop */

          /* now pick the starting point for traceback
*/
          state_sequence[depth_of_trellis] = h;

          /* now work backwards from the end of the
trellis to the oldest state in the trellis to determine the
optimal path. The purpose of this is to determine the most
likely state sequence at the encoder based on what channel
symbols we received. */

          for (j = depth_of_trellis; j > 0; j--)
          {
               sh_col = j + ( sh_ptr -
depth_of_trellis );

               if (sh_col < 0)

                    sh_col = sh_col + depth_of_trellis
+ 1;
               state_sequence[j - 1] = state_history[
state_sequence[j] ] [sh_col];

          } /* end of j for-loop */

          /* now figure out what input sequence
corresponds to the state sequence in the optimal path */
          *(decoder_output_matrix + t -
depth_of_trellis + 1) = input[ state_sequence[0] ] [
state_sequence[1] ];
     } /* end of if-statement */
} /* end of 't' for-loop */

/* now decode the encoder flushing channel-output bits
*/
for (t = channel_length - m; t < channel_length; t++)

{
     /* set up the state history array pointer for
this time t */
     sh_ptr = (int) ( ( t + 1 ) % (depth_of_trellis +
1) );
```

```
          /* don't need to consider states where input was
a 1, so determine what is the highest possible
          state number where input was 0 */
          last_stop =  number_of_states /(int)
pow((double)2, t - channel_length + m);

          /* repeat for each possible state */
          for (j = 0; j < last_stop; j++)
          {
               branch_metric = 0;
               deci2bin(output[j][0], n, binary_output);

               /* compute metric per channel bit, and sum
for all channel bits in the convolutional encoder output n-
tuple */
               for (ll = 0; ll < n; ll++)
               {
                    branch_metric = branch_metric +
soft_metric( *(channel_output_matrix +(ll * channel_length
+ t)), binary_output[ll] );
               } /* end of 'll' for loop */

               /* now choose the surviving path--the one
with the smaller total metric... */
               if ( (accum_err_metric[ nextstate[j][0] ][1]
> accum_err_metric[j][0] +branch_metric) /*|| flag[
nextstate[j][0] ] == 0*/)
               {

/* save a state metric value for the survivor state */
                    accum_err_metric[ nextstate[j][0] ][1]
= accum_err_metric[j][0] + branch_metric;

                    /* update the state_history array with
the state number of the survivor */
                    state_history[ nextstate[j][0]
][sh_ptr] = j;
               } /* end of if-statement */

          } /* end of 'j' for-loop */

          /* for all rows of accum_err_metric, swap columns
1 and 2 */
          for (j = 0; j < number_of_states; j++)
          {
               accum_err_metric[j][0] =
accum_err_metric[j][1];
```

```
                accum_err_metric[j][1] = MAXINT;

          } /* end of 'j' for-loop */

          /* now start the traceback, if i >=
depth_of_trellis - 1*/
          if (t >= depth_of_trellis - 1)
          {
                /* initialize the state_sequence vector */
                for (j = 0; j <= depth_of_trellis; j++)
state_sequence[j] = 0;

                /* find the state_history element with the
minimum accum. error metric */
                x = accum_err_metric[0][0];
                h = 0;
                for (j = 1; j < last_stop; j++)
                {
                     if (accum_err_metric[j][0] < x)
                     {
                          x = accum_err_metric[j][0];
                          h = j;
                     } /* end if */

                } /* end 'j' for-loop */

                state_sequence[depth_of_trellis] = h;
/* now work backwards from the end of the trellis to the
oldest state in the trellis to determine the optimal
path.The purpose of this is to determine the most likely
state sequence at the encoder based on what channel symbols
we received. */
                for (j = depth_of_trellis; j > 0; j--)
                {
                     sh_col = j + ( sh_ptr -
depth_of_trellis );
                     if (sh_col < 0)
                          sh_col = sh_col + depth_of_trellis
+ 1;
                     state_sequence[j - 1] = state_history[
state_sequence[j] ][sh_col];
                } /* end of j for-loop */
                /* now figure out what input sequence
corresponds to the optimal path */
                *(decoder_output_matrix + t -
depth_of_trellis + 1) = input[ state_sequence[0] ][
state_sequence[1] ];
```

```
          } /* end of if-statement */

     } /* end of 't' for-loop */

     for (i = 1; i < depth_of_trellis - m; i++)
     {
          *(decoder_output_matrix + channel_length -
depth_of_trellis + i) =  input[ state_sequence[i] ] [
state_sequence[i + 1] ];
     }
     for(i=0;i<channel_length-m;i++)
     {
       printf("\n decoder output matrix is
%d",*(decoder_output_matrix + i));
        getchar();
     }
     /* free the dynamically allocated array storage area
*/
     free(channel_output_matrix);
     return;
} /* end of function vd */

int nxt_stat(int current_state, int input, int
*memory_contents)
     {
          int binary_state[K - 1];             /* binary
value of current state */
          int next_state_binary[K - 1];        /* binary
value of next state */
          int next_state;                      /* decimal
value of next state */
          int i;                               /* loop
variable */
        void deci2bin(int d, int size, int *b);
        int bin2deci(int *b, int size);
/* convert the decimal value of the current state number to
binary */
          deci2bin(current_state, K - 1, binary_state);
/* given the input and current state number, compute the
next state number */
          next_state_binary[0] = input;

          for (i = 1; i < K - 1; i++)
               next_state_binary[i] = binary_state[i -
1];/* convert the binary value of the next state number to
decimal */
```

```
          next_state = bin2deci(next_state_binary, K -
1);/* memory_contents are the inputs to the modulo-two
adders in the encoder */
          memory_contents[0] = input;
          for (i = 1; i < K; i++)
              memory_contents[i] = binary_state[i - 1];
          return(next_state);
     }

     /* this function converts a decimal number to a binary
number, stored   as a vector MSB first,having a specified
number of bits with leading   zeroes as necessary */
     void deci2bin(int d, int size, int *b)
     {
          int i;
          for(i = 0; i < size; i++)
               b[i] = 0;
          b[size - 1] = d & 0x01;

          for (i = size - 2; i >= 0; i--)
          {
               d = d >> 1;
               b[i] = d & 0x01;
          }
     }

     /* this function converts a binary number having a
specified number of bits to the corresponding decimal
number ith improvement contributed by Bryan Ewbank
2001.11.28 */

     int bin2deci(int *b, int size)
     {
          int i, d;
          d = 0;
          for (i = 0; i < size; i++)
               d += b[i] << (size - i - 1);
          return(d);
     }

/* Function to generate uniform deviates using idum*/

#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
```

```
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

float ran2(void)
{
     int j;
     long k;
     static long idum2=123456789;
     static long iy=0;
     static long iv[NTAB];
     float temp;

     if (idum <= 0) {
          if (-(idum) < 1) idum = 1;
          else idum = -(idum);
          idum2 = (idum);
          for (j=NTAB+7; j>=0; j--) {
               k = (idum)/IQ1;
               idum = IA1*(idum - k*IQ1) - k*IR1;
               if (idum < 0) idum += IM1;
               if (j < NTAB) iv[j] = idum;
          }
          iy = iv[0];
     }
     k = (idum)/IQ1;
     idum = IA1*(idum - k*IQ1) - k*IR1;
     if (idum < 0) idum += IM1;
     k = idum2/IQ2;
     idum2 = IA2*(idum2 - k*IQ2) - k*IR2;
     if (idum2 < 0) idum2 += IM2;
     j = iy/NDIV;
     iy = iv[j] - idum2;
     iv[j] = idum;
     if (iy < 1) iy += IMM1;
     if ((temp=AM*iy) > RNMX) return RNMX;
     else return temp;
}
```

APPENDIX B

/* This program computes the Monte Carlo simulation of ordinary DSSS based on a canonical distribution of arbitrary interference using convolutional encoder and Viterbi decoder */

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>
#include <limits.h>
#define MAX_SIZE 50
#define TWOTOTHEM 4
#define MAXMETRIC 128
#define MAXINT 16384

void generateSequence(int dataLength,int *dataSequence);
void multiplyAndIntegrate(int dataLength,int codeLength,int
**codeSequence,double **signalPlusNoise,double
*decodedSequence);
void multiply(int dataLength,int codeLength,int
*dataSequence,double **codeSequence,double
**multipliedSequence);
void addSignalAndNoise(int dataLength, int codeLength,
double **multipliedSequence, double *noiseArray, double
**signalPlusNoise);
void generateCode(int dataLength,int codeLength,int
**codeArray);
void generateNoise(int snr,int length,int
*interleaveArray,int D,double *noiseArray);
void convertToSignal(int *dataSequenceAfterCoding,int
dataLength, int *dataSequenceInSignal);
void acceptEquation(int **equationSet,int k, int rate);
void compareAndDecide(int dataLength,double
*decodeSequence,int *sequenceAfterDeciding);
int getErrors(int dataLength,int *originalSequence,int
*receiveSequence);
double calculateBER(long totalNoOfBits,long
totalNoOfErrors);
void vd(int g[2][K],float es_ovr_no,float es_ovr_ni,long
channel_length,float *channel_output_vector,int
*decoder_output_matrix);
```

```
void conv_encoder(int g[2][K], long data_len, int
*in_array, int *out_array);
int quant(float channel_symbol);
int soft_metric(int data, int guess);
int nextNumber();
long int idum;

int main()
{

  int **equationSet;
  int dataSequenceArray[MAX_SIZE];
  int dataSequenceAfterCodingArray[MAX_SIZE];
  int dataSequenceInSignalArray[MAX_SIZE];
  double decodedSequenceArray[MAX_SIZE];
  int sequenceAfterDecisionArray[MAX_SIZE];
  int originalDecodedDataArray[MAX_SIZE];
  int interleaveArray[MAX_SIZE];
  int codedDataSequence[MAX_SIZE];
  int codeSequenceArray[MAX_SIZE][MAX_SIZE];
  double multipliedSequenceArray[MAX_SIZE][MAX_SIZE];
  double noiseSequenceArray[MAX_SIZE];
  double signalPlusNoiseArray[MAX_SIZE][MAX_SIZE];
  int sequenceAfterDeConvolution[MAX_SIZE];
  long noOfErrors,noOfBits;
  int codeLength,rate =0, dataLength=0, rows,snr,D,loops;
  dataLength = 5K;
  codeLength = N;
  rate = r;
  double worstBER,ber;
    #if K == 3        /* polynomials for K = 3 */
    int g[2][K] = {{1, 1, 1},
                   {1, 0, 1}};
    #endif

    #if K == 5        /* polynomials for K = 5 */
    int g[2][K] = {{1, 1,  1, 0, 1},
                   {1, 0,  0, 1, 1}};
    #endif

    #if K == 7        /* polynomials for K = 7 */
    int g[2][K] = {{1,  1, 1, 1,  0, 0, 1},
                   {1,  0, 1, 1,  0, 1, 1}};
    #endif

    #if K == 9        /* polynomials for K = 9 */
    int g[2][K] = {{1, 1, 1,  1, 0, 1,  0, 1, 1},
```

```
                    {1, 0, 1,  1, 1, 0,  0, 0, 1}};
    #endif

/* Input parameters */

     printf("Enter constraint length K [int]:   ");
     scanf("%d", &K);
     getchar();*/
     printf("Enter number of PN generator chips per bit N
[int]:   ");
     scanf("%d", &N);
     getchar();
        printf("Enter code rate [int]:   ");
     scanf("%d", &r);
     getchar();
     printf("Enter interleaver seed for interleaving
[negative long int]:   ");
     scanf("%ld", &idum);
     getchar();

     for (snr = 0; snr < 30;  snr++)  {

          worstBER=0;
          for( D = 0; D<5K*rate*N;D++ ) {

               for( loops =0; loops <10000; loops++)
                         {

generateSequence(dataLength, dataSequenceArray);

conv_encoder(g,dataLength,dataSequenceArray,codedDataSequen
ce);

convertToSignal(codedDataSequence, dataLength,
dataSequenceInSignalArray);

generateCode(dataLength*rate, codeLength, (int
**)codeSequenceArray);

multiply(dataLength*rate, codeLength,
dataSequenceInSignalArray, (int **)codeSequenceArray,
(double **)multipliedSequenceArray);

generateNoise(snr, dataLength*rate*codeLength,
interleaveArray, D, noiseSequenceArray);
```

```
addSignalAndNoise(dataLength*rate, codeLength, (double
**)multipliedSequenceArray, noiseSequenceArray, (double
**)signalPlusNoiseArray);

multiplyAndIntegrate(dataLength*rate, codeLength, (int
**)codeSequenceArray ,(double **)signalPlusNoiseArray,
decodedSequenceArray);

compareAndDecide(dataLength*rate,decodedSequenceArray,seque
nceAfterDecisionArray);

vd(g,snr,snr,dataLength*rate,(float
*)sequenceAfterDecisionArray,(int
*)sequenceAfterDeConvolution);

                    noOfErrors = noOfErrors +
getErrors(dataLength, sequenceAfterDeConvolution,
dataSequenceArray);

                    noOfBits = noOfBits + dataLength*rate;

               }

               ber = calculateBER(noOfBits, noOfErrors);

               if(worstBER < ber)  {
                    worstBER=ber;
               }

               noOfBits=0;
               noOfErrors=0;
          }

printf("the worst BER for %d snr is %f ",snr, worstBER);
     }
  return 0;
}

/* random data generation for message */
void generateSequence(int dataLength,int *dataSequence){
     int i;

     for( i=0; i< dataLength; i++) {
          dataSequence[i]= (int)ran2()%2;
     }
}
```

```
/* at the receiver end multiply and integrate the chips
received from noisy channel */
void multiplyAndIntegrate( int dataLength, int
codeLength,int **codeSequence, double
**signalPlusNoise,double *decodedSequence )
{
     int i=0;

     for(i=0; i < dataLength; i++ )     {
          double temp=0;
          int j=0;

          for(j=0; j < codeLength; j++)      {
               temp = temp + (codeSequence[i][j] *
signalPlusNoise[i][j]);
          }

          decodedSequence[i] = temp;
     }

}

/* multiply symbols and chips at the transmitter end */
void multiply(int dataLength, int codeLength, int
*dataSequence,
                 double **codeSequence, double
**multipliedSequence)    {
     int i,j;

     for(i = 0; i < dataLength; i++)    {
          for(j = 0; j < codeLength; j++)    {


     multipliedSequence[i][j]=codeSequence[i][j]*dataSequen
ce[i];
          }
     }
}

/* add noise to signal */
void addSignalAndNoise(int dataLength, int codeLength,
double **multipliedSequence,
                                             double
*noiseArray, double **signalPlusNoise)  {
     int n=0,i=0,j;

     for(i = 0; i < dataLength; i++)    {
```

```
          for(j = 0; j < codeLength; j++)    {

               signalPlusNoise[i][j] = noiseArray[n] +
multipliedSequence[i][j];
               n++;
          }
     }

}

/* generate code symbols */
void generateCode(int dataLength, int codeLength, int
**codeArray)   {
     int i = 0,j,temp=0;

     for(i = 0;i < dataLength; i++)     {
          for(j=0;j<codeLength;j++)     {

               temp=(int)ran2%3;

                if(temp==0) codeArray[i][j]=0;

                else if(temp==1) codeArray[i][j]=-1;

                else codeArray[i][j]=1;
          }
     }
}

/* generate noise */
void generateNoise(int snr, int length, int
*interleaveArray, int D,
                              double *noiseArray)   {
     int i = 0;
     for( i = 0; i < D; i++)  {

          noiseArray[interleaveArray[i]] =
sqrt(length/(pow(10,(double)(snr-10)/10)*D));

     }
}

/* convert to baseband signals */
void convertToSignal(int *dataSequenceAfterCoding, int
dataLength, int *dataSequenceInSignal){
     int i = 0;
```

```
     for(i = 0; i < dataLength;i++)     {

          if(dataSequenceAfterCoding[i] == 0) {
               dataSequenceInSignal[i] = -1;
          }
          else {
               dataSequenceInSignal[i] = 1;
          }
      }
}

int nextNumber(){

    //return random number
    return 0;
}

/* generator polynomials of encoder */
void acceptEquation (int **equationSet, int k, int rate) {
     int i,j;
     for(i = 0; i < rate; i++)     {

          for(j = 0; j < k; j++)   {

          printf("Enter the value of row %d and column %d
",i,j);
          scanf("%d",&equationSet[i][j]);
          }
     }
}

/* hard decision */
void compareAndDecide(int dataLength, double
*decodeSequence, int *sequenceAfterDeciding)  {
     int i;

     for( i = 0;  i < dataLength; i++) {

          if(decodeSequence[i] < 0)     {
          sequenceAfterDeciding[i] = 0;
          }
          else  {
          sequenceAfterDeciding[i] = 1;
          }
     }
}
```

```
/* calculate errors */
int getErrors(int dataLength,int *originalSequence,int
*receiveSequence){
     long result = 0;
     int i;

     for(i = 0; i < dataLength; i++)    {
          if(originalSequence[i] != receiveSequence[i] )
     {
               result++;
          }
     }

     return result;

}

/* final ber calculation*/
double calculateBER(long totalNoOfBits,long
totalNoOfErrors){
        double result=0;
        result = totalNoOfErrors/totalNoOfBits;
        return results;
    }

/* convolutional encoder*/

void conv_encoder(int g[2][K], long data_len, int
*in_array, int *out_array)
{
    int m= K-1;                /* K - 1 */
    long t,S;                  /* bit time, symbol time */
    int j, k;                  /* loop variables */
    int *unencoded_data;       /* this is the pointer to
data array */
    int shift_reg[K];          /* the encoder shift
register */
    int sr_head;               /* index to the first entry
in the sr */
    int a,b;            /* the upper and lower xor gate
outputs */

   long channel_length = ( data_len + m ) * 2;

    /* allocate space for the zero-padded input data array
*/
```

```
   unencoded_data = (int
*)malloc((data_len+m)*sizeof(int));
    if (unencoded_data == NULL) {
        printf("\n conv_encoder.c: Can't allocate enough
memory for unencoded data!  Aborting...");
        exit(1);
    }

    //unencoded_data = in_array ;
    /* read the input data and store it in the array */
   for (t = 0; t < data_len; t++)
        *(unencoded_data + t) = *(in_array + t);

    /* now zero-pad the end of the data */
    for (t = 0; t < m; t++) {
        *(unencoded_data + data_len + t) = 0;
    }

    /* Initializing the shift register */
    for (j = 0; j < K; j++)
 {
        shift_reg[j] = 0;
    }

   /* In order to speed things up a little, the shift
register will be operated
      as a circular buffer, so it needs a head
pointer.we'll just be overwriting the oldest entry with the
new data. */

    sr_head = 0;

    /* initializing the channel symbol output index */
    S = 0;

    /* Here the encoding process begins */
    /* now compute the upper and lower modulo-two adder
outputs, one bit at a time */
    for (t = 0; t < data_len + m; t++)
 {
    shift_reg[sr_head] = *( unencoded_data + t );
     a = 0;
     b = 0;
        for (j = 0; j < K; j++)
     {
         k = (j + sr_head) % K;
         a ^= shift_reg[k] & g[0][j];
```

```
        b ^= shift_reg[k] & g[1][j];
       }

      /* write the upper and lower xor gate outputs as
channel symbols */
     *(out_array + S) = a;
       S = S + 1;
      // printf(" %d\n",a);

     *(out_array + S) = b;
       S = S + 1;
      // printf("%d\n",b);

       sr_head -= 1;    /* This is equivalent to shifting
everything right one place */
       if (sr_head < 0) /* we need to make sure that the
pointer modulo K is adjusted */
           sr_head = m;

    }
/* now transform the data from 0/1 to +1/-1 */
    for (t = 0; t < channel_length; t++)
     {

       /*if the binary data value is 1, the channel symbol
is -1; if the
       binary data value is 0, the channel symbol is +1.
*/
       *(out_array+t) = 1 - 2 *  *( out_array + t );

      // printf("%d\n",*( out_array + t ));

     }
/*now the dynamically allocated array is made free */
      free(unencoded_data);

}

/* viterbi decoder */
void vd(int g[2][K],float es_ovr_no,float es_ovr_ni,long
channel_length,float *channel_output_vector,int
*decoder_output_matrix)
{
     int i, j, l, ll;               /* loop variables */
    long t;                         /* time */
     int memory_contents[K]; /* input + conv. encoder sr */
```

```
     int input[TWOTOTHEM][TWOTOTHEM]; /* maps current/nxt
sts to input */
     int output[TWOTOTHEM][2]; /* gives conv. encoder
output */
     int nextstate[TWOTOTHEM][2];/* for current st, gives
nxt given input */

     int accum_err_metric[TWOTOTHEM][2];/* accumulated
error metrics */

     int state_history[TWOTOTHEM][K * 5 + 1];  /* state
history table */
     int state_sequence[K * 5 + 1];/* state sequence list
*/
     int *channel_output_matrix; /* ptr to input matrix */

     int binary_output[2];   /* vector to store binary enc
output */


        int branch_output[2]; /* vector to store trial enc
output */

     int m, n, number_of_states, depth_of_trellis, step,
branch_metric,sh_ptr, sh_col, x, xx, h, hh, next_state,
last_stop; /* misc variables */
     /* n is 2^1 = 2 for rate 1/2 */
     n = 2;

     /* m (memory length) = K - 1 */
     m = K - 1;

     /* number of states = 2^(K - 1) = 2^m for k = 1 */
     number_of_states = (int) pow((double)2, m);
     depth_of_trellis = K * 5;
void deci2bin(int d, int size, int *b);
int bin2deci(int *b, int size);
int nxt_stat(int current_state, int input, int
*memory_contents);
void init_adaptive_quant(float es_ovr_no,float es_ovr_ni);
int quant(float channel_symbol);
int soft_metric(int data, int guess);

     /* initialize data structures */
     for (i = 0; i < number_of_states; i++)
     {
          for (j = 0; j < number_of_states; j++)
```

```
                input[i][j] = 0;
          for (j = 0; j < n; j++)
          {
                nextstate[i][j] = 0;
                output[i][j] = 0;
          }
          for (j = 0; j <= depth_of_trellis; j++)
          {
                state_history[i][j] = 0;
          }
          /* initial accum_error_metric[x][0] = zero */
          accum_err_metric[i][0] = 0;
          /* by setting accum_error_metric[x][1] to MAXINT,
we don't need a flag */
          /* MAXINT is simply the largest possible integer,
defined in values.h */
          accum_err_metric[i][1] = MAXINT;
     }

     /* generate the state transition matrix, output
matrix, and input matrix - input matrix shows how FEC
encoder bits lead to next state */
     for (j = 0; j < number_of_states; j++)
     {
          for (l = 0; l < n; l++)
          {
                next_state = nxt_stat(j, l,
memory_contents);
                input[j][next_state] = l;

                /* now compute the convolutional encoder
output given the current state number and the input value
*/
                branch_output[0] = 0;
                branch_output[1] = 0;

                for (i = 0; i < K; i++)
                {
                     branch_output[0] ^= memory_contents[i]
& g[0][i];
                     branch_output[1] ^= memory_contents[i]
& g[1][i];
                }

     /* next state, given current state and input */
                nextstate[j][l] = next_state;
```

```
/* output in decimal, given current state and input */
               output[j][l] = bin2deci(branch_output, 2);

          } /* end of l for loop */

     } /* end of j for loop */

#ifdef DEBUG
     printf("\nInput:");

     for (j = 0; j < number_of_states; j++)
     {
          printf("\n");

          for (l = 0; l < number_of_states; l++)
               printf("%2d ", input[j][l]);

     } /* end j for-loop */

     printf("\nOutput:");

     for (j = 0; j < number_of_states; j++)
     {
          printf("\n");
          for (l = 0; l < n; l++)
               printf("%2d ", output[j][l]);

     } /* end j for-loop */

     printf("\nNext State:");

     for (j = 0; j < number_of_states; j++)
     {
          printf("\n");

          for (l = 0; l < n; l++)
               printf("%2d ", nextstate[j][l]);

     } /* end j for-loop */

#endif

     channel_output_matrix = (int*) malloc(channel_length *
sizeof(int));

     if (channel_output_matrix == NULL)
     {
```

```
printf(        "\nsdvd.c: Can't allocate memory for
channel_output_matrix! Aborting...");
          exit(1);
     }
/* now we're going to rearrange the channel output so it
has n rows, and n/2 columns where each row corresponds
to a channel symbol for a given bit and each column
corresponds to an encoded bit */
     channel_length = channel_length / n;

  /* adaptive quantization for specified es_ovr_no)*/
   init_adaptive_quant(es_ovr_no,es_ovr_ni);
     /* quantize the channel output--convert float to short
integer */
     /* channel_output_matrix = reshape(channel_output, n,
channel_length) */
     for (t = 0; t < (channel_length * n); t += n)
     {
          for (i = 0; i < n; i++)
               *(channel_output_matrix + (t / n) + (i *
channel_length) ) = quant( *(channel_output_vector + (t +
i) ) );
     } /* end t for-loop */
     /* End of setup. Start decoding of channel outputs
with forward traversal of trellis!
     Stop just before encoder-flushing bits.  */
    for (t = 0; t < channel_length - m; t++)
     {
          if (t <= m)
               /* assume starting with zeroes, so just
compute paths from all-zeroes state */
               step = (int)pow((double)2, m - t * 1);
          else
               step = 1;

          /* set up the state history array pointer for
this time t */
          sh_ptr = (int) ( ( t + 1 ) % (depth_of_trellis +
1) );

          /* repeat for each possible state */
          for (j = 0; j < number_of_states; j+= step)
          {
               /* repeat for each possible convolutional
encoder output n-tuple */
               for (l = 0; l < n; l++)
               {
```

```
                    branch_metric = 0;

/* compute branch metric per channel symbol, and sum for
all channel symbols in the convolutional encoder output n-
tuple */
/* convert the decimal representation of the encoder output
to binary */
                    binary_output[0] = ( output[j][l] &
0x00000002 ) >> 1;
                    binary_output[1] = output[j][l] &
0x00000001;

                    /* compute branch metric per channel
symbol, and sum for all channel symbols in the
convolutional encoder output n-tuple */
                    branch_metric = branch_metric + abs( *(
channel_output_matrix +( 0 * channel_length + t ) ) - 7 *
binary_output[0] ) +
                         abs( *( channel_output_matrix +( 1
* channel_length + t ) ) - 7 * binary_output[1] );

                    /* now choose the surviving path--the
one with the smaller accumlated error metric... */
                    if ( accum_err_metric[ nextstate[j][l]
] [1] > accum_err_metric[j][0] +branch_metric )
                    {
                         /* save an accumulated metric
value for the survivor state */
                         accum_err_metric[ nextstate[j][l]
] [1] = accum_err_metric[j][0] +branch_metric;

                         /* update the state_history array
with the state number of the survivor */
                         state_history[ nextstate[j][l] ]
[sh_ptr] = j;

                    } /* end of if-statement */

               } /* end of 'l' for-loop */

          } /* end of 'j' for-loop -- we have now updated
the trellis */

          /* for all rows of accum_err_metric, move col 2
to col 1 and flag col 2 */
          for (j = 0; j < number_of_states; j++)
          {
```

```
               accum_err_metric[j][0] =
accum_err_metric[j][1];
               accum_err_metric[j][1] = MAXINT;

          } /* end of 'j' for-loop */

          /* now start the traceback, if we've filled the
trellis */
          if (t >= depth_of_trellis - 1)
          {

               /* initialize the state_sequence vector--
probably unnecessary */
               for (j = 0; j <= depth_of_trellis; j++)
                    state_sequence[j] = 0;

               /* find the element of state_history with
the min. accum. error metric */
               /* since the outer states are reached by
relatively-improbable runs of zeroes or ones,
               search from the top and bottom of the
trellis in */
               x = MAXINT;
               for (j = 0; j < ( number_of_states / 2 );
j++)
               {
                    if ( accum_err_metric[j][0] <
accum_err_metric[number_of_states - 1 - j][0] )
                    {
                         xx = accum_err_metric[j][0];

                         hh = j;
                    }
                    else
                    {
                         xx =
accum_err_metric[number_of_states - 1 - j][0];
                         hh = number_of_states - 1 - j;
                    }
                    if ( xx < x)
                    {
                         x = xx;
                         h = hh;
                    }
               } /* end 'j' for-loop */
```

```
               /* now pick the starting point for traceback
*/
               state_sequence[depth_of_trellis] = h;

               /* now work backwards from the end of the
trellis to the oldest state in the trellis to determine the
               optimal path. The purpose of this is to
determine the most likely state sequence at the encoder
               based on what channel symbols we received.
*/
               for (j = depth_of_trellis; j > 0; j--)
               {
                    sh_col = j + ( sh_ptr -
depth_of_trellis );
                    if (sh_col < 0)
                         sh_col = sh_col + depth_of_trellis
+ 1;
                    state_sequence[j - 1] = state_history[
state_sequence[j] ] [sh_col];
               } /* end of j for-loop */
               /* now figure out what input sequence
corresponds to the state sequence in the optimal path */
               *(decoder_output_matrix + t -
depth_of_trellis + 1) = input[ state_sequence[0] ] [
state_sequence[1] ];
          } /* end of if-statement */
     } /* end of 't' for-loop */
     /* now decode the encoder flushing channel-output bits
*/
     for (t = channel_length - m; t < channel_length; t++)
     {
          /* set up the state history array pointer for
this time t */
          sh_ptr = (int) ( ( t + 1 ) % (depth_of_trellis +
1) );
          /* don't need to consider states where input was
a 1, so determine what is the highest possible
          state number where input was 0 */
          last_stop =  number_of_states /(int)
pow((double)2, t - channel_length + m);

          /* repeat for each possible state */
          for (j = 0; j < last_stop; j++)
          {
               branch_metric = 0;
               deci2bin(output[j][0], n, binary_output);
```

```
               /* compute metric per channel bit, and sum
for all channel bits in the convolutional encoder output n-
tuple */
               for (ll = 0; ll < n; ll++)
               {
                    branch_metric = branch_metric +
soft_metric( *(channel_output_matrix +(ll * channel_length
+ t)), binary_output[ll] );
               } /* end of 'll' for loop */

               /* now choose the surviving path--the one
with the smaller total metric... */
               if ( (accum_err_metric[ nextstate[j][0] ][1]
> accum_err_metric[j][0] +branch_metric) /*|| flag[
nextstate[j][0] ] == 0*/)
               {
                    /* save a state metric value for the
survivor state */
                    accum_err_metric[ nextstate[j][0] ][1]
= accum_err_metric[j][0] + branch_metric;

                    /* update the state_history array with
the state number of the survivor */
                    state_history[ nextstate[j][0]
][sh_ptr] = j;

               } /* end of if-statement */

          } /* end of 'j' for-loop */

          /* for all rows of accum_err_metric, swap columns
1 and 2 */
          for (j = 0; j < number_of_states; j++)
          {
               accum_err_metric[j][0] =
accum_err_metric[j][1];
               accum_err_metric[j][1] = MAXINT;

          } /* end of 'j' for-loop */

          /* now start the traceback, if i >=
depth_of_trellis - 1*/
          if (t >= depth_of_trellis - 1)
          {
               /* initialize the state_sequence vector */
               for (j = 0; j <= depth_of_trellis; j++)
state_sequence[j] = 0;
```

```
/* find the state_history element with the minimum accum.
error metric */
               x = accum_err_metric[0][0];
               h = 0;
               for (j = 1; j < last_stop; j++)
               {
                    if (accum_err_metric[j][0] < x)
                    {
                         x = accum_err_metric[j][0];
                         h = j;
                    } /* end if */

               } /* end 'j' for-loop */


               state_sequence[depth_of_trellis] = h;

               /* now work backwards from the end of the
trellis to the oldest state in the trellis to determine the
optimal path. The purpose of this is to determine the most
likely state sequence at the encoder based on what channel
symbols we received. */
               for (j = depth_of_trellis; j > 0; j--)
               {
                    sh_col = j + ( sh_ptr -
depth_of_trellis );

                    if (sh_col < 0)
                         sh_col = sh_col + depth_of_trellis
+ 1;
                    state_sequence[j - 1] = state_history[
state_sequence[j] ][sh_col];
               } /* end of j for-loop */

               /* now figure out what input sequence
corresponds to the optimal path */
               *(decoder_output_matrix + t -
depth_of_trellis + 1) = input[ state_sequence[0] ][
state_sequence[1] ];

          } /* end of if-statement */

     } /* end of 't' for-loop */

     for (i = 1; i < depth_of_trellis - m; i++)
     {
```

```
          *(decoder_output_matrix + channel_length -
depth_of_trellis + i) =  input[ state_sequence[i] ] [
state_sequence[i + 1] ];

     }
     for(i=0;i<channel_length-m;i++)
     {
       printf("\n decoder output matrix is
%d",*(decoder_output_matrix + i));
        getchar();
     }
     /* free the dynamically allocated array storage area
*/
     free(channel_output_matrix);

     return;

} /* end of function vd */

int nxt_stat(int current_state, int input, int
*memory_contents)
     {
          int binary_state[K - 1];             /* binary
value of current state */
          int next_state_binary[K - 1];        /* binary
value of next state */
          int next_state;                      /* decimal
value of next state */
          int i;                               /* loop
variable */
        void deci2bin(int d, int size, int *b);
        int bin2deci(int *b, int size);
          /* convert the decimal value of the current state
number to binary */
          deci2bin(current_state, K - 1, binary_state);

          /* given the input and current state number,
compute the next state number */
          next_state_binary[0] = input;

          for (i = 1; i < K - 1; i++)
               next_state_binary[i] = binary_state[i -
1];/* convert the binary value of the next state number to
decimal */
          next_state = bin2deci(next_state_binary, K -
1);/* memory_contents are the inputs to the modulo-two
adders in the encoder */
```

```
          memory_contents[0] = input;

          for (i = 1; i < K; i++)
              memory_contents[i] = binary_state[i - 1];
          return(next_state);
     }

     /* this function converts a decimal number to a binary
number, stored   as a vector MSB first,having a specified
number of bits with leading   zeroes as necessary */
     void deci2bin(int d, int size, int *b)
     {
          int i;
          for(i = 0; i < size; i++)
               b[i] = 0;
          b[size - 1] = d & 0x01;

          for (i = size - 2; i >= 0; i--)
          {
               d = d >> 1;
               b[i] = d & 0x01;
          }
     }

     /* this function converts a binary number having a
specified number of bits to the corresponding decimal
number ith improvement */

     int bin2deci(int *b, int size)
     {
          int i, d;
          d = 0;
          for (i = 0; i < size; i++)
               d += b[i] << (size - i - 1);
          return(d);
     }

/* Function to generate uniform deviates using idum*/

#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
```

```
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
float ran2(void)
{
     int j;
     long k;
     static long idum2=123456789;
     static long iy=0;
     static long iv[NTAB];
     float temp;
     if (idum <= 0) {
          if (-(idum) < 1) idum = 1;
          else idum = -(idum);
          idum2 = (idum);
          for (j=NTAB+7; j>=0; j--) {
               k = (idum)/IQ1;
               idum = IA1*(idum - k*IQ1) - k*IR1;
               if (idum < 0) idum += IM1;
               if (j < NTAB) iv[j] = idum;
          }
          iy = iv[0];
     }
     k = (idum)/IQ1;
     idum = IA1*(idum - k*IQ1) - k*IR1;
     if (idum < 0) idum += IM1;
     k = idum2/IQ2;
     idum2 = IA2*(idum2 - k*IQ2) - k*IR2;
     if (idum2 < 0) idum2 += IM2;
     j = iy/NDIV;
     iy = iv[j] - idum2;
     iv[j] = idum;
     if (iy < 1) iy += IMM1;
     if ((temp=AM*iy) > RNMX) return RNMX;
     else return temp;
}
```