2008

Physical layer secrecy channel coding

Bandali K. Akkawi

Louisiana State University and Agricultural and Mechanical College, bakkawi@hotmail.com

PHYSICAL LAYER SECRECY CHANNEL CODING

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical & Computer Engineering

by
Bandali K. Akkawi
B.Sc., Princess Sumaya University for Technology, Amman-Jordan, 2005
August 2008

Acknowledgments

First of all, I would like to express my deepest gratitudes to my supervisor Dr. Shuangqing Wei for his consistent guidance and great support throughout the course of this thesis, without which this work could not be done. I appreciate his great patience and kindness, his timely ideas and suggestions. I have learned a lot from him in the past two years and adored his commitment and love for his work. I also would like to thank Dr. Morteza Naraghi-Pour and Dr. Xue-Bin Liang for being members of my thesis defense committee.

I also want to thank my friend George Amariucai for his helping hand, suggestions and ideas throughout the time we spent working together.

And last but not least, I would like to express my greatest gratitudes and thanks to my parents and brothers who always stand by me, support me, and be there for me. They never fail to encourage and give me enough confidence to continue my journey . . .

Table of Contents

Abstract

Wireless communications is expanding and becoming an indispensable part of our daily life. However, due to its channel open nature, it is more vulnerable to attacks, such as eavesdropping and jamming which jeopardize the confidentiality of wireless data, compared to its counter-part, wireline communications. Security in wireless communication is thus a very important factor that should be perfected to accommodate the rapid growth of wireless communication today.

Motivated by information theoretic secrecy definitions, we adopt a simple way to define the secrecy of a system by looking at its Bit-Error-Rate (BER) curves, the correlation of error vectors and Log Likelihood Ratios (LLRs) of the decoded information bits. The information bit errors and LLRs of a physical layer secure system should be uncorrelated and the BER curve should have an acceptable sharp transition from high to low BERs at prescribed signal to noise ratio (SNR) thresholds.

We study catastrophic codes and Serial Concatenated Convolutional Codes (SCCC) as two candidates. For the former, we provide both detailed analytical and simulation results, to demonstrate how we can change the encoding parameters to make the resulting BER curves have the intended properties. For SCCC, we study two options. One is having a catastrophic code as an inner code. The other is to use regular SCCC. Several approaches are proposed to change the shape of the resulting BER curves.

In addition, the correlation present in their information bit errors and LLRs are investigated to see how it can be used to detect or even correct errors. We find that regular SCCC codes have strong correlation in their error vectors which is captured by the associated LLRs. In low SNR regions, eavesdropper can easily make reliable decisions on which packets to drop based on LLRs, which thus undermines the security of the main channel data. On the other hand, by selecting proper outer codes, SCCC with catastrophic encoder does not have such a weakness.

We conclude that Catastrophic convolutional codes, as well as serial concatenated catastrophic codes have desired properties. Therefore, they can be considered promising approaches to achieving practical secrecy in wireless systems.

Chapter 1

Introduction

1.1 Wireless Communication and Security

The time we live in is very much described as the communication era, where modern technology has enabled us to communicate with each other literally with a single click. Internet has grown very rapidly that it has made modern society depend so heavily on its functions and abilities. Recently, wireless applications start being ubiquitous, replacing all wired technology, and making us connected whenever and wherever we might be [1]. With wireless communication and public networks, more and more concern is focused on the security features of modern technology [2]. Every human being wants to make sure that whatever he is doing is kept secret and that no one should be able to gain any private information from what is being transmitted through the complex worldwide networks.

Unfortunately, wireless communications are found to be relatively easier to attack than conventional wired networks [3]. The more popular wireless communications becomes, the easier to find a way to crack into it, and maybe even use it to crack into the wired networks as well. Attacking methods have become much more sophisticated and innovative with wireless [4]. Cracking has also become much easier and more accessible with easy-to-use Windows-based and Linux-based tools being made available on the web at no charge [2].

Finding appropriate methods to secure wireless networks is one of the major topics in research today. One of the most widely used security mechanisms is encryption. Encryption is a mathematical process through which confidentiality can be ensured [5]. It is based on mathematical functions (encryption/decryption algorithms) to encrypt plain-text messages into cipher-texts. For this reason, it can also be called computational security. In most cases, two related functions (*keys*) are employed, one for encryption and the other for decryption [6]. Cryptography is manifested in methods such as symmetric and public-key encryptions. With symmetric-key encryption, the encryption key can be calculated from the decryption key and vice versa. With most symmetric algorithms, the same key is used for both encryption and decryption. In this scenario, the key must be totally kept secret between the two communicating parties. If any outsider were able to get the key, then the confidentiality of the communications is failed, since it can decrypt and understand all the messages transmitted. On the other hand, in public

key encryption, one public key is used to encrypt a message and another private key to decrypt it. The public key can be widely distributed so that anyone can encrypt a possible message, while the private key is kept secret by the receiver which uses it to decrypt the message. Although the two keys are mathematically related, however, it is very practically difficult to derive the private key from the public one [7].

A well-known example of public-key cryptography is the RSA method, which generates the keys as follows [8]:

- Choose two distinct large prime numbers p and q.

- Compute $n = pc$ (the modulus for both the public and private keys).

- Compute the tent $\varphi(n) = (p-1)(q-1)$.

- Choose an integer e such that $1 < e < \varphi(n)$, and e and $\varphi(n)$ are coprime.

- Compute d to satisfy the congruence relation $de \equiv 1 (\mathrm{mod}\ \varphi(n))$.

- e is released as the public key exponent, and d is kept secret as the private key exponent.

- Encryption of a message m is done by producing the cypher $c = m^e \bmod n$.

- Decryption of c is similarly done by $m = c^d \bmod n$.

The RSA as can be seen is a completely mathematical system. Its security is a function of some mathematical problems such as factoring large prime numbers [8] (i.e. p and q from n). For this reason, the best way to keep the system secure is to choose n as large as possible (typically, they are 1024-2048 bits long). Some experts believe that 1024-bit keys may become breakable in the near term (though this is disputed); few see any way that 4096-bit keys could be broken in the foreseeable future [9]. Therefore, it is generally presumed that RSA is secure if n is sufficiently large.

As seen, computational security always assumes that the communicating parties are more knowledgeable than the attacker (e.g. knowledge of keys), and is deemed secure whenever there are no publicly known efficient attacks that crack into the system. Because of the computational nature of these schemes, they rely on the assumption that the attacker does not have enough computational power to crack the system [10]. Thus, if an attacker could arrange enough computational power, breaking the system will then be a possibility.

In addition to computational security schemes, another form of security is actively being researched today, which is information theoretic security [11]. Unlike computational security methods that rely on practical mathematical difficulties in decryption to imply security, information theoretic approach tend to go further and puts perfect secrecy as the ultimate goal, where the attacker will not be able to extract anything from its received message, not because the message (plain-text) is very powerfully encrypted, but simply because the received message is too noisy to understand. It is also notable

that, unlike computational security, information theoretic secrecy tries to avoid giving an advantage of prior knowledge to the communicating parties, and thus making them and the attacker equally knowledgeable. However, information theoretic security does rely on the differences of channel qualities between the communicants and the attacker to provide security, as will be shown in the next section.

1.2 Information Theoretic Secrecy

Today, most security systems are implemented through the use of cryptographic protocols working above the already established physical layer (such as RSA, AES etc). These protocols exploit the almost error-free physical channel and take it simply as a perfect transportation medium between the communicants. When using such protocols, the physical message will be perfectly clear to anyone that can intercept it, its only difficulty would be breaking the cypher being used. These methods provide a reliable way to achieve security, but nevertheless they are neither perfectly secret nor secure since the whole physical message can be easily caught by any interceptor.

Perfect secrecy, however, is presented and introduced in information theoretic concepts, in which the physical layer itself ought to be designed in such a way that even if an interceptor received the message, it would not be able to decode or infer any information from it. Lately, this problem started to get more attention due to an urgent need to re-examine the security issues in communications.

Perfect secrecy was first introduced by Shannon [12], where he introduced the one-time pad concept, proving that a plain-text message M can be sent in perfect secrecy by transmitting a cipher-text $C = M \otimes K$ produced by adding the original message M to a random key or pad K, where $\otimes$ is the mod 2 addition. The transmitted C gives absolutely no additional information about the original message M, thus, the *a priori* probability of a plain-text message M is the same as the *a posteriori* probability of a plain-text message M given the corresponding cipher-text C. Mathematically, this is expressed as $H(M) = H(M|C)$ or $I(M;C) = 0$, where $H(M)$ is the entropy of the plain-text and $H(M|C)$ is the conditional entropy of the plain-text given the cipher-text C and $I(M;C)$ is the mutual information between the plain-text and cypher-text. In order to keep the message secure, the pad must be at least as long as the message (i.e.$H(K) \geq H(M)$) and must be used only once (hence the name one-time pad).

Later Wyner built further on Shannon's notion of perfect secrecy and introduced the famous "*Wire-Tap Channel*" [13] where he investigated the secrecy of messages transmitted over a channel (from Alice to Bob) with a wire-tapper (Eve) listening through a degraded channel hoping to get any possible information. He proved that perfect secret communications between Alice and Bob is possible whenever Eve's channel is degraded. The basic model for this communication system can be seen in Figure 1.1.

The system can be described as follows. Alice, being the source, produces the information sequence as identically independent distributed random variables $A_k, k = 1, 2, \ldots$
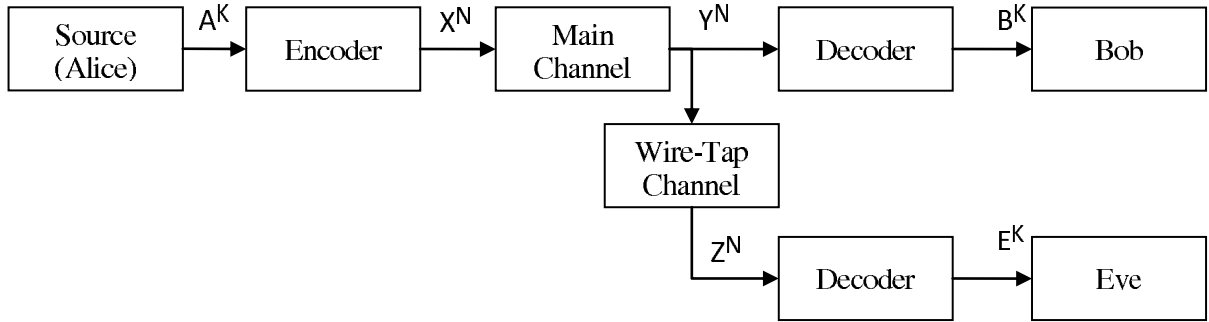
Figure 1.1: Wire-tap channel model

that take values from the finite set $\mathbb{A}$ with cardinality $|\mathbb{A}|$. The probability law defining $\{A_k\}$ is known, and the entropy of the source is

$$H(A_k) \;=\; H_A \;=\; -\sum_{x \in \mathbb{A}} P_x(x) \log_2 P_x(x), \quad \{P_x(x) : x \in \mathbb{A}\}. \tag{1.1}$$

The encoder is a mapping rule $(f_E : \mathbb{A}^K \to \mathbb{X}^N)$, where the K source variables $A^K = (A_1, \ldots, A_K)$ are the encoder input and the N outputs $X^N = (X_1, \ldots, X_N)$ are the encoder output. Both the main channel and the wire-tap channel are regarded as discrete memoryless channels with transition probabilities $Q_M(y|x)$ and $Q_W(z|y), x \in \mathbb{X}, y \in \mathbb{Y}, z \in \mathbb{Z}$, where $\mathbb{Y}, \mathbb{Z}$ are finite alphabet sets at the output of the main and wiretap channels respectively. A cascaded channel can be constructed from Alice to Eve to have a transition probability of

$$Q_{MW}(z|x) \;=\; \sum_{y \in \mathbb{Y}} Q_W(z|y) Q_M(y|x) \tag{1.2}$$

The decoder is a reverse mapping $(f_D : \mathbb{Y}^N \to \mathbb{B}^K$ or $f_D : \mathbb{Z}^N \to \mathbb{E}^K)$ that should output in ideal conditions the same input sequence A^K.

An important measure of secrecy in Wyner's paper [13] is the equivocation rate (Δ) which is defined as

$$\Delta \;\triangleq\; \frac{1}{K} H(A^K | Z^N) \tag{1.3}$$

This equivocation rate is taken as the criterion of wire-tapper's confusion. The larger Δ is, the more confused the wire-tapper is about the original message A^K. Wyner goes on to define a region $\mathbb{R}$ within the rate-equivocation space that includes all the achievable (R,d) pairs such that

$$\begin{array}{rcl} \frac{H_A K}{N} & \geq & R - \epsilon \\ \Delta & \geq & d - \epsilon \\ P_e & \leq & \epsilon \end{array} \tag{1.4}$$

where P_e is Bob's average error rate

$$P_e \;=\; \frac{1}{K} \sum_{k=1}^{K} Pr\{A_k \neq B_k\} \tag{1.5}$$

4

In defining the region, it can be seen that $(R, d) \in \mathbb{R}$ satisfies

$$
\begin{array}{ccccc}
0 & \leq & R & \leq & C_M \\
0 & \leq & d & \leq & H_A
\end{array}
\qquad (1.6)
$$

where $C_M = \sup_{p(x)} I(X;Y)$ is the main channel capacity. Another variable $\Gamma(R)$ is also used to define $\mathbb{R}$, $\Gamma(R)$ measures the maximum information that can be shared between Alice and Bob without the knowledge of Eve (i.e. secret information) at any given rate R.

$$
\Gamma(R) \triangleq \sup_{p(x)} I(X;Y|Z) = \sup_{p(x)} [I(X;Y) - I(X;Z)]
\qquad (1.7)
$$

which equals to $\sup_{p(x)} I(X;Y) - \sup_{p(x)} I(X;Z) = C_{AB} - C_{AE}$ whenever both the main channel (Alice to Bob) and the cascade of the main and wiretap channel (Alice to Eve) are symmetric [14]. Under these definitions, Wyner proved that the achievable region must also be contained in the region where

$$
Rd \leq H_A \Gamma(R)
\qquad (1.8)
$$

and thus $\mathbb{R}$ is given in Figure 1.2
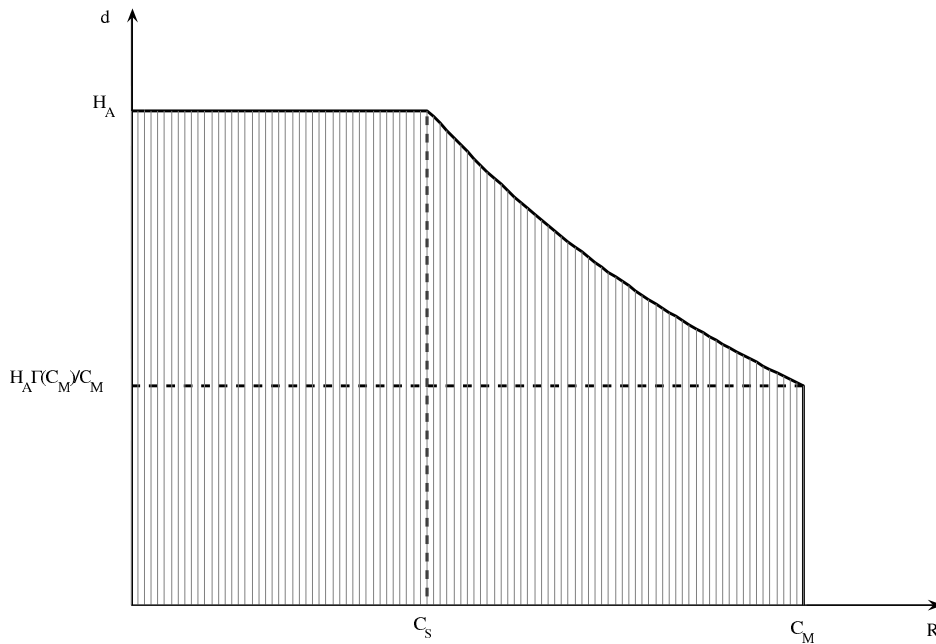
Figure 1.2: Region $\mathbb{R}$

In his proof, Wyner described an encoding scheme in which the encoder in Figure 1.1 is divided into a source and channel encoders as shown in Figure 1.3. The source encoder maps the information sequence A^K into one of M symbols or words W ($F_E : A^K \rightarrow \{1, 2, \ldots, M\}$). The channel encoder consists of M subcodes $C_1, C_2, \ldots, C_M$, each with M_2 points, totaling $M_1 = M \times M_2$ points $\mathbf{x}_m \subseteq \mathbb{X}^N, m = \{1, 2, \ldots, M_1\}$. Each subcode C_i consists of

$$
C_i = \{\mathbf{x}_{(i-1)M_2+1}, \ldots, \mathbf{x}_{iM_2}\}, \quad 1 \leq i \leq M
\qquad (1.9)
$$

5
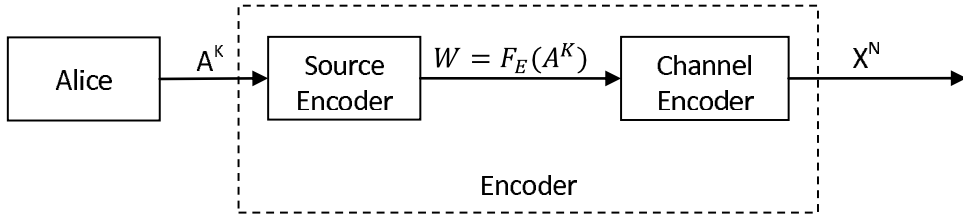
Figure 1.3: Wyner Encoder

When Alice produces a random variable A^K, the source encoder maps it to some $W = i \in \{1, 2, \ldots, M\}$. The channel encoder then randomly chooses one point from the subcode C_i, and thus producing

$$X^N = \mathbf{x}_{(i-1)M_2+j}, \quad 1 \le j \le M_2 \tag{1.10}$$

The decoder role is to map back the received signal (Y^N for Bob, or Z^N for Eve) to one of the channel code words $\mathbf{x}_m, m \in \{1, 2, \ldots, M_1\}$. From which the subcode C_i is easily known and hence the original source coded $W = i$ is found. In order for the transmission to have above-zero secrecy at an above-zero rate, M_1 and M_2 should be selected such that $M_1 \le 2^{N \times I(X;Y)}$ and $M_2 \le 2^{N \times I(X;Z)}$, where $I(X;Y)$ and $I(X;Z)$ are the mutual information between X, Y and X, Z respectively.

As an example, consider an error-free main channel and a BSC wiretap channel, with crossover probability p_0, as shown in Figure 1.4, and let the encoder rate to be $1/N$. Let
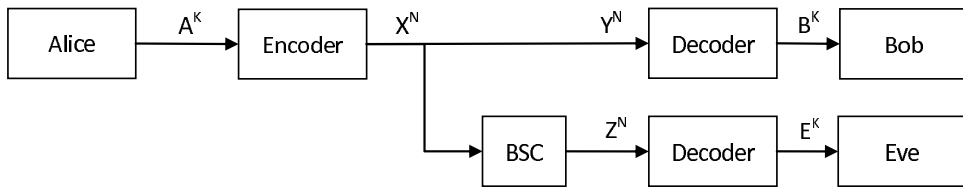
Figure 1.4: Error-free main channel and a BSC wiretap channel

C_0 be the subset (subcode) of binary N space, $\{0, 1\}^N$, consisting of those N vectors with even parity (i.e. an even number of ones). Let $C_1 \subseteq \{0, 1\}^N$ be the subset of vectors with odd parity. The encoder works as follows. When $A^1 = i, (i = 0, 1)$, the encoder output X^N, a randomly chosen vector in C_i. Thus the encoder has the following transition probability

$$Pr\{X^N = \mathbf{x} | A^1 = i\} = \begin{cases} 2^{-(N-1)}, & \mathbf{x} \in C_i \\ 0, & \mathbf{x} \notin C_i \end{cases} \tag{1.11}$$

for $i = 0, 1$. Clearly, Bob's decoder can recover A^1 from X^N perfectly with $P_e = 0$. The wiretapper, Eve, on the other hand observes Z^N, the output of the BSC corresponding

to the input X^N. Let $\mathbf{z} \in \{0, 1\}^N$ be a vector, say, of even parity. Then

$$
\begin{aligned}
Pr\{A^1 = 0 | Z^N = \mathbf{z}\} &= Pr\left\{ \begin{array}{c} \text{The BSC makes an} \\ \text{even number of errors} \end{array} \right\} \\
&= \sum_{\substack{j=0 \\ j \text{ even}}}^{N} \binom{N}{j} p_0^j (1 - p_0)^{N-j} = \frac{1}{2} + \frac{1}{2}(1 - 2p_0)^N.
\end{aligned}
\tag{1.12}
$$

The last equality can be verified by applying the binomial formula to

$$
[(1 - p_0) \pm x p_0]^N = \sum_{j=0}^{N} \binom{N}{j} p_0^j (1 - p_0)^{N-j} (\pm x)^j.
\tag{1.13}
$$

Then,

$$
2 \sum_{j \text{ even}} \binom{N}{j} p_0^j (1 - p_0)^{N-j} = (1 - p_0 + 1.p_0)^N + (1 - p_0 - 1.p_0)^N
\tag{1.14}
$$

$$
= 1 + (1 - 2p_0)^N.
\tag{1.15}
$$

Similarly, for $\mathbf{z} \in \{0, 1\}^N$ of odd parity,

$$
\begin{aligned}
Pr\{A^1 = 0 | Z^N = \mathbf{z}\} &= Pr\left\{ \begin{array}{c} \text{The BSC makes an} \\ \text{odd number of errors} \end{array} \right\} \\
&= \tfrac{1}{2} - \tfrac{1}{2}(1 - 2p_0)^N.
\end{aligned}
\tag{1.16}
$$

Therefore, for all $\mathbf{z} \in \{0, 1\}^N$,

$$
H(A^1 | Z^N = \mathbf{z}) = h[\frac{1}{2} - \frac{1}{2}(1 - 2p_0)^N],
\tag{1.17}
$$

where,

$$
h[\lambda] = -\lambda \log_2 \lambda - (1 - \lambda) \log_2 (1 - \lambda), \ 0 \leq \lambda \leq 1
\tag{1.18}
$$

so that,

$$
\begin{aligned}
\Delta &= H(A^1 | Z^N) = h[\tfrac{1}{2} - \tfrac{1}{2}(1 - 2p_0)^N] \\
&\rightarrow 1 = H(A^1), \quad \text{as} \quad N \rightarrow \infty
\end{aligned}
\tag{1.19}
$$

Thus, as $N \rightarrow \infty$, the equivocation at the wire-tap approaches the unconditional source entropy, so that communication is accomplished in perfect secrecy. Unfortunately, as $N \rightarrow \infty$, the transmission rate $K/N \rightarrow 0$.

Although the transmission rate in the example above went to zero in order to achieve perfect secret communications between Alice and Bob, Wyner proved that above-zero rates can be obtained while maintaining perfect secrecy as shown in Figure 1.2. But at the same time, near channel capacity transmission can never achieve complete secrecy, and a reduced transmission rate (R) must be used to increase the equivocation (Δ), which is bound by its upper limit (H_A). Therefore the highest rate that achieves this

$\Delta = H_A$ becomes the secrecy capacity rate ($C_S = \Gamma(C_S)$) at which Alice and Bob can communicate in perfect secrecy leaving Eve with nothing valuable.

The existence of C_S is guaranteed whenever the main channel capacity (C_M) is larger than the cascaded channel capacity (C_{MW}) and satisfies

$$0 \;\; < \;\; C_M - C_{MW} \;\; \leq \;\; C_S \;\; \leq \;\; C_M. \tag{1.20}$$

In [15], the wiretap model was extended to Gaussian channels, modeling the main and wire-tap channels as Additive Wight Gaussian Noise (AWGN) channels such that

$$Y \;\; = \;\; X + N_M \tag{1.21}$$
$$Z \;\; = \;\; Y + N_W \tag{1.22}$$

where N_M, N_W are have normal distributions $\mathcal{N}(0, \sigma_M^2)$ and $\mathcal{N}(0, \sigma_W^2)$ respectively. Limiting the channel power to P, we get

$$
\begin{aligned}
C_M &= \tfrac{1}{2} \log \left(1 + \tfrac{P}{\sigma_M^2} \right) \\
C_{MW} &= \tfrac{1}{2} \log \left(1 + \tfrac{P}{\sigma_M^2 + \sigma_W^2} \right) \\
C_S &= C_M - C_{MW}
\end{aligned}
\tag{1.23}
$$

and the region $\mathbb{R}$ becomes defined by

$$
\begin{aligned}
R &\leq C_M \\
d &\leq 1 \\
Rd &\leq C_S
\end{aligned}
\tag{1.24}
$$

which consequently leads to the conclusion that confidential communication in perfect secrecy is always possible in degraded channels where the signal-to-noise ratio (SNR) of the main channel exceeds that of the cascaded channels.

Csiszár and Körner in [16] then studied broadcast channels with confidential messages, as a generalization of the wire-tap channel. Where the sender (Alice) also wishes to transmit common information to both the legitimate receiver (Bob) and the wire-tapper (Eve) in addition to the private (confidential) information to the legitimate receiver. Moreover, they generalized Wyner's model by proving that Alice-to-Eve channel need not be a degraded version of the main channel, but it is sufficient to be more noisy than the main channel for a perfect secret communication to be possible.

These works, as can be seen, concentrate on information theoretic bounds, not giving much insight on how to construct practical systems, and thus these problems were not given much attention until Maurer [17] used a similar concept to computational security, and proved that Alice and Bob can generate and use a secret key even when they have a worse channel than Eve's. This is proved through using an extra insecure yet authenticated public channel. In [17],[18] and [19], the secret key distribution problem in wiretap channels was studied extensively, with the objective to let Alice and Bob share a secret common k-bit key about which Eve's conditional entropy is maximized. In this context of key distribution, the k bits can be unknown to Alice before transmission, which contrasts

the secure message communication where Alice has a k-bit message and wants to transmit it to Bob as in Wyner. Nonetheless, powerful ideas such as common randomness, advantage distillation and privacy amplification were developed and deployed in this context [19][20], and several key distribution protocols were developed and studied, where most require some level of interactive communication between Alice and Bob to arrive at a common and secret key. Information exchange at this implementation point is based on the use of parallel, error-free public channel available to both legitimate communicants.

Coding schemes problems using only the forward channel with no parallel channel did not receive much attention. Some examples of coding schemes were provided in [13] and [17], and a condition for constructing such codes for a modified wiretap channel as introduced in [21] was studied in [22]. However, code construction methods and their connection to security have not been explored much. Recently, low density parity check (LDPC) codes got the attention in this area, especially after proving that coding schemes for various generalized wiretap channel scenarios do exist based particularly on the use of LDPC codes [23]. In [24], for example, a secret key agreement protocol over the Gaussian wiretap channel was investigated. The protocol is based on efficient information reconciliation method based on LDPC codes, which allows two parties having access to correlated continuous random variables to agree on a common bit string. In other results, the use of a public error-free channel was avoided as seen in [25] and [26], where Thangaraj et al presented some secrecy-achieving codes for a Binary Erasure Channel (BEC) wiretap channel and an error-free main channel. They also provided some code constructions and conditions to achieve perfect secrecy when the two channels are BEC, as well as some coding solutions using good error detecting codes when the wiretap channel is a BSC and the main channel is error-free achieving perfect secrecy asymptotically and leading the rate to zero. In other works, Bloch et-al [27][28] developed another protocol for secret key agreement for Gaussian channels, that performs close to secrecy capacity limits determined in [27], over wide range of channel values without the need for a noiseless, authenticated public channel. The presented protocol makes use of opportunistic transmission, message reconciliation and establishes the secret key through privacy amplification.

The wire-tap problem was also present in Multiple Access Channels (MAC), where communications consist of more than one user trying simultaneously to send or receive data with a shared destination. Each user may send two types of data: shared common information and some confidential messages intended only to the legitimate recipient. The problem arises when the users also receive each others' messages though some channel (because of the broadcast nature of wireless communications). This means that each user may extract the others' messages, including what supposed to be confidential data. In this scenario, a possible approach is to make each user treat the others as possible wire-tappers wishing to keep them as ignorant as possible. This approach, for example, is studied for two users in [29], in which different cases are considered, including counterparts for [13] and [16] having only one user sending confidential messages. Another case is studied is when both users have some confidential data to be sent to the destination. For the latter case, an inner bound on the capacity-equivocation region is obtained. A similar problem is also studied in [30], where communication is done over a Gaussian broadcast channel. In this paper, however, one common transmitter (multi-antenna) wants to communicate

with different users (again two users) sending independent confidential information to each one of them. Each user would like to obtain its own confidential message in a reliable and safe manner making sure that the other is ignorant of it. Secrecy-rate regions and bounds are developed based on Gaussian codebooks achieving this goal.

1.3 Our Objective

Motivated by Wyner's original wiretap channel [13] and its equivalent Gaussian model presented in [15], our work is concentrated on investigating some practical schemes that will exploit the extra noise found at Eve, to render its information practically useless, while making sure that Bob receives and decodes the message correctly.

The model we used is presented in Figure 1.5. As it can be seen, the model is slightly
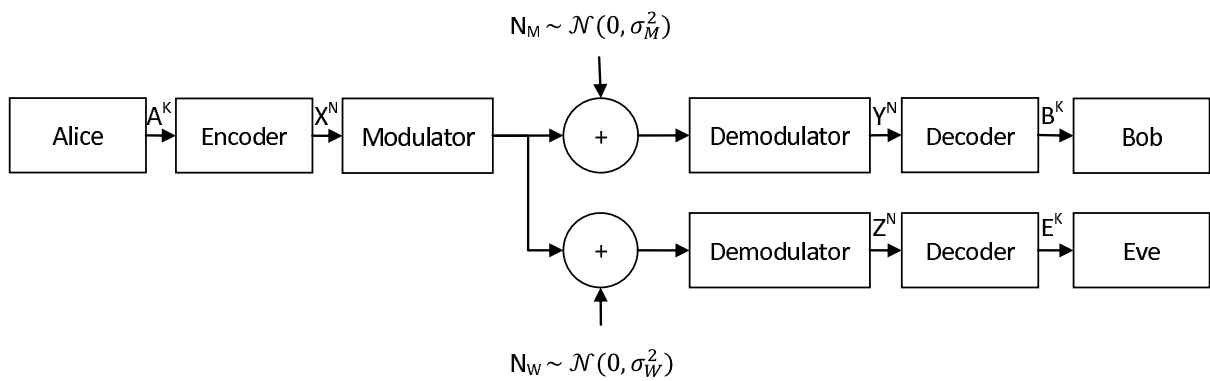
$$N_M \sim \mathcal{N}(0, \sigma_M^2)$$

| Alice | A^K | Encoder | X^N | Modulator | | + | | Demodulator | Y^N | Decoder | B^K | Bob |

| + | | Demodulator | Z^N | Decoder | E^K | Eve |

$$N_W \sim \mathcal{N}(0, \sigma_W^2)$$

Figure 1.5: Wire-tap Gaussian channel model

different than that in Figure 1.1 but similar to the model of Csiszár and Körner in [16]; Eve's channel is no longer two concatenated channels but rather one Gaussian channel that is by assumption worse than the main (Alice to Bob) channel (i.e. $\sigma_W^2 > \sigma_M^2$). In other words, whenever Alice sends a message, the SNR of the message received at Eve is lower than that at Bob.

As a deviation from the definition of secrecy presented in [13], we adopt another way to define the secrecy of a system. In most communications systems, the bit error rate (BER) of any system is used to show its reliability regarding reception and decoding. For example, if a system's BER is 10^{-5}, it can be said that it is a reliable system (of course depending on the application being used), while on the other hand, if it has a BER of 0.5, then the system is practically useless, since each bit decoded has a 50% probability that it might be wrong. This obvious measure of a communication system reliability can lead to a new definition of security that we have used in our investigations. A wiretap system modeled in Figure 1.5 can be said to be practically secure if Bob received Alice's message with a BER below a certain threshold (again, depending on the application used), while Eve received the message with a BER higher than 0.1, for example, and have uncorrelated errors at Eve's SNR. From here, we can say that our main objective of this study is to investigate the construction of a code that behaves similar to the BER curve shown in
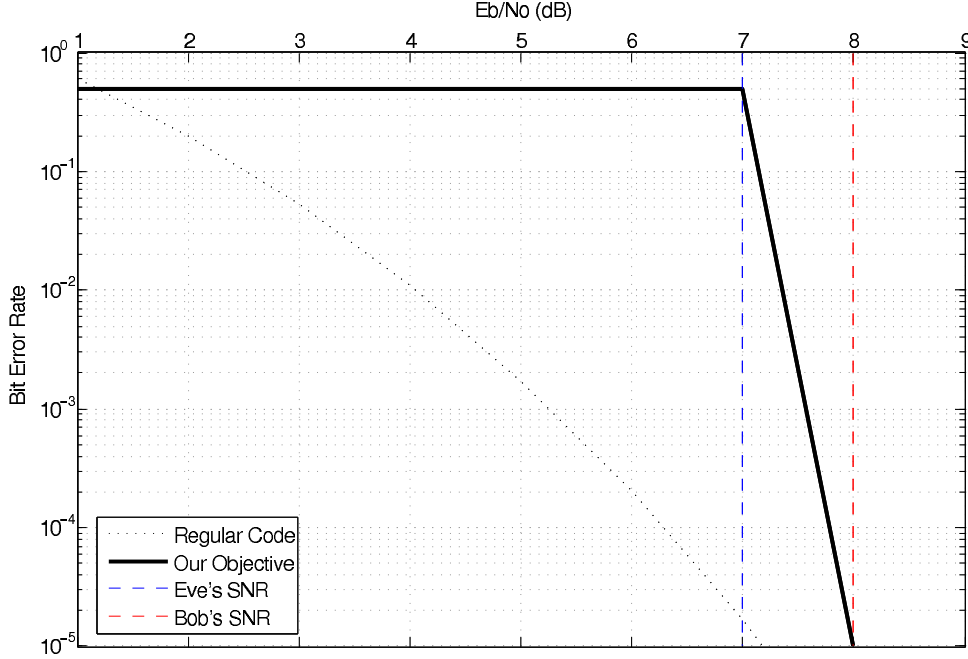
Figure 1.6: The desired code should have a BER curve that has a sharp transition at an SNR higher than Eve's but lower than Bob's

Figure 1.6, in which there is a sharp transition at an SNR higher than Eve's while at the same time having lower BER than that of Bob. In addition, we need the errors in the code to be uncorrelated. In this way, Eve's received message is practically useless while Bob's message is received and decoded reliably. Needless to say, since Bob and Eve's SNRs are variables depending on the systems and environments they are in, the desired code need also to be flexible in a way that allows us to shift the curve to lower or higher SNRs as needed.

In Wyner's [13] approach, the SNRs at Bob and Eve are used to find the secrecy capacity C_S at which the desired code should achieve perfect secrecy asymptotically. We, on the other hand, take another approach to the problem. We employ codes of constant rates and try to see how much more secure can we make them and at what expenses. Consider a certain application. The BER above which the data is useless is $P_{e1} = 1 \times 10^{-1}$, and the BER below which everything is clear is $P_{e2} = 1 \times 10^{-4}$. Let $SNR_1 = \text{SNR}(P_{e1})$ and $SNR_2 = \text{SNR}(P_{e2})$, the region where $\text{SNR} < SNR_1$ can thus be considered useless, and the region $\text{SNR} > SNR_2$ can be considered perfectly clear, while the buffer zone region $SNR_1 < \text{SNR} < SNR_2$ is neither. We start with some code with rate R as the regular convolutional code in Figure 1.7. As seen, this code has $SNR_1 < 1\text{dB}$ and $SNR_2 = 5\text{dB}$, with all the region in between belong to the buffer zone, which is not perfectly secure. In order to secure the low SNR region, we must increase SNR_1 to some requirement that is considered above Eve's channel capability. This increase will have as its expense the increase of SNR_2 as well, to satisfy P_{e2}. Of course the ideal case is if SNR_2 could equal SNR_1 with a BER curve having a very sharp transition at that SNR. However, according to Wyner's work, any secret transmission must be done below the secrecy capacity C_S which is a function of the difference between SNR_1 and SNR_2, and
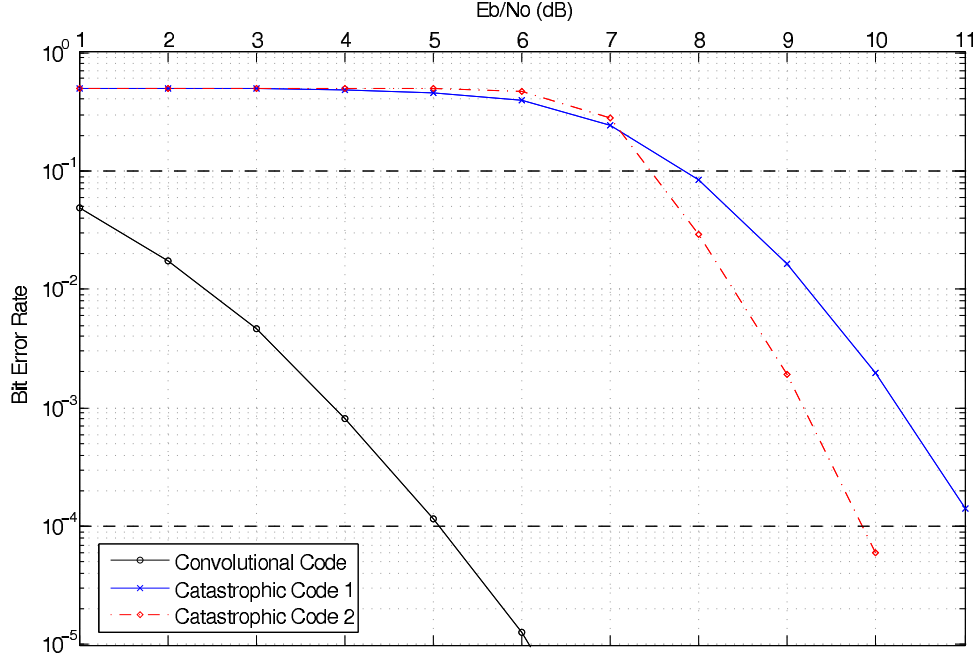
11

Figure 1.7: Regular convolutional code compared with two samples of catastrophic codes

hence that ideal case does not hold theoretically because a rate $R < C_S = 0$ must be used for transmission. Therefore, a buffer zone must always be present for secret transmission to take place. We then use other codes with the same rate R of the regular convolution code to achieve the SNR_1 requirement. Catastrophic codes, studied in Chapter 2, were shown to have a better desired curve as the two sample codes in the same Figure 1.7. Take the catastrophic code 1 for example. SNR_1 is raised to 8dB, at the expense of increasing SNR_2 to about 11.25dB, while for code 2, we are able to get $SNR_1 = 7.5$dB with a lower $SNR_2 = 9.75$dB. To compare the three codes, we introduce a metric $\rho = f(R, P_{e1}, P_{e2})$; defined as

$$\rho \;=\; \frac{SNR_2 - SNR_1}{SNR_2} \tag{1.25}$$

where $0 \leq \rho \leq 1$, with 0 being the best and 1 being the worst. Comparing the three codes will yield that the regular code's $\rho > 0.8$, while that of the catastrophic codes 1 and 2 to be $\rho = 0.289$ and $\rho = 0.231$ respectively. This clearly shows how the example of new catastrophic codes better serve for a more secure communication.

1.4 Our Contribution

In this thesis, we present a study of mainly two types of codes. The first code is almost always avoided in coding applications, catastrophic codes. Catastrophic codes are known to have a very bad BER at low to mid SNRs, which enhances at higher SNRs. The very bad BER at low SNRs is what we were first interested in, for that behavior is, in a way, our objective for Eve. We present a detailed study of catastrophic codes in Chapter 2 and their possible use as secure codes containing some new ideas, including:

- Introducing the concept of parallel catastrophic paths and parallel states, and how catastrophic codes behavior is based on them.

- Studying the catastrophic errors' events and how they affect the behavior of decoded bits throughout the transmitted sequence.

- Studying the effects of finite packet length transmission of catastrophic codes, as well as the lattice labeling of the transmitted symbols. How controlling these two parameters can change the BER curves.

- Providing a simple theoretical method to estimate the BER curve of catastrophic codes.

- Analyzing decoded bits Log-Likelihood Ratios (LLRs) to better understand the correlation between these bits and the decoding errors of the uncoded information bits.

Then in Chapter 3, we study the powerful Serially Concatenated Convolutional Codes (SCCC) [31]. SCCC codes have the desired characteristic of a very sharp transition from a very bad BER to almost error-free transmission in a span of less than 1-2 dBs. However, this transition is at a very low SNR which must be shifted to a higher SNR for our objective to be met. A study is presented including the following:

- See whether catastrophic codes can be used in SCCC to achieve our objective

- Study different methods to shift the SNR and see their effects on SNR_1 and SNR_2.

- Analyze the LLRs as done in Chapter 2 and propose a new simple ARQ method to be used in concatenated codes.

Finally, the thesis is concluded in Chapter 4 with a summary and prospecting future work.

Chapter 2

Catastrophic Codes

In this chapter, we will study the use of catastrophic codes in the context of physical layer security. Catastrophic codes have been always avoided due to their poor performance in low-mid SNRs, but their behavior in high SNRs should not be separated from the normal codes [32] since catastrophic codes perform similar if not even better than normal convolutional codes in such regions. Because of this, there is a sharper transition from high to low BER located at mid-high SNRs. This behavior is what interests us and is studied in this here.

2.1 Introduction to Convolutional Codes

2.1.1 Convolutional Codes

• **Brief Description**

A convolutional code is generated by passing an information sequence to be transmitted through a linear finite-state shift register. The shift register consists of L bits (the constraint length) that are used to generate n linear algebraic functions as shown in Figure 2.1. The information sequence is passed k bits at a time, generating n output bits for each time. The code rate is thus k/n.

The encoder is usually defined by its generating polynomial G, which has n numbers (usually in octal) representing the n outputs, each of which has L binary bits representing the register constraint length as presented in [33]. Each number in G represents one linear function generator that is composed of a modulo-2 adder that adds all the shift register bits that correspond to 1 in its binary form. Another way to represent the convolutional code as presented in [34] is using a $k \times n$ matrix $G(D)$ consisting of code generating polynomials such that

$$G(D) \;=\; \begin{bmatrix} g_{11}(D) & \dots & g_{1n}(D) \\ \vdots & & \vdots \\ g_{k1}(D) & \dots & g_{kn}(D) \end{bmatrix} \tag{2.1}$$

In this latter case, the encoder is represented in $\lceil L/k \rceil$ registers each with 1 input bit and n outputs bits. These registers are added up together to form the k/n encoder, hence

14
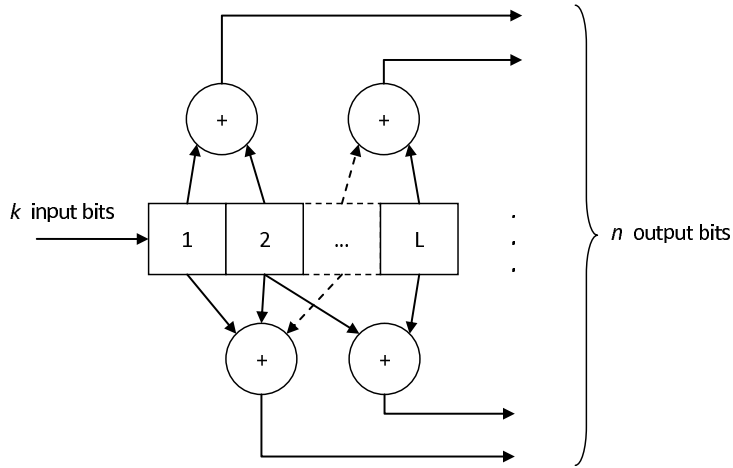
Figure 2.1: Convolutional Encoder

the $k \times n$ matrix G(D). In most of the thesis, we will present the encoder as presented in [33], unless on certain occasions where $G(D)$ is better suited for the context.

There are 2^{L-k} possible states in any convolutional code (a state is represented by the first $L-k$ bits in the register). Depending on the state of the register, each input (k-bits) will shift the register to a different state outputting the corresponding n output bits. And thus it follows that each convolutional encoder can be represented by a state diagram, where transitions between states are defined by the input as well as the corresponding output.

An example of a regular convolutional code can be seen in Figure 2.2. This is a
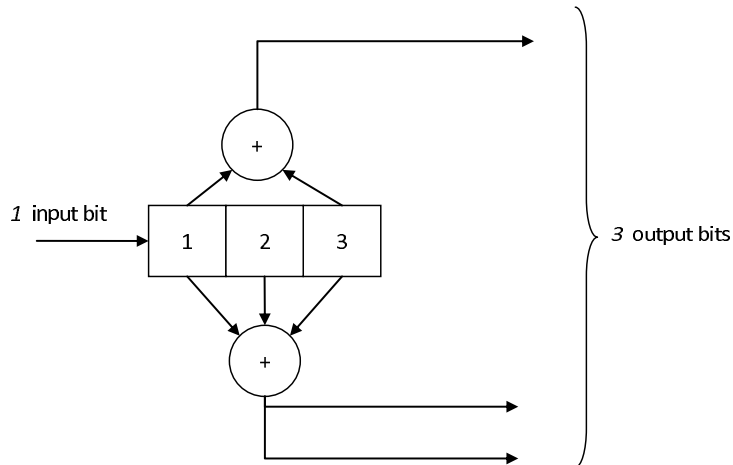
Figure 2.2: Convolutional Encoder Example with $G = [5, 7, 7]$

4-state 1/3 rate code with a generating polynomial $G = [5, 7, 7] = [101, 111, 111]$ or $G(D) = [1 + D^2, 1 + D + D^2, 1 + D + D^2]$.

15

A trellis diagram can be constructed to represent all the transitions that occur between states from start to finish such as the one shown in Figure 2.3. Usually all convolutional

Figure 2.3: Trellis Diagram for the 4 states 1/3 rate code with G=[5,7,7]

coding systems start with the all-zero state and then continue onwards. This trellis diagram is very helpful in the decoding process of convolutional codes.

• Hard and Soft Decision Decoding in the Viterbi Algorithm

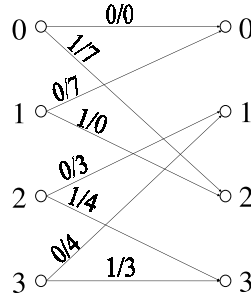It is well known that Viterbi algorithm is the optimum decoding algorithm to minimize the sequence error probability and produce the maximum likelihood (ML) sequence of inputs in a convolutional code. In many cases, the n output bits from the encoder produced at time $t : t \in \{1, 2, \ldots, T\}$ are mapped to one of M symbols (M $= 2^n$) present in an M-ary modulation scheme, where T is the decoding period. Let $\mathbb{A}$ be the set of M symbols and let $a_i \in \mathbb{A}, i \in \{1, 2, \ldots, M\}$ represent each symbol. Without loss of generality, we assume that each symbol $a_i \in \mathbb{Z}$ is a complex number $a_i = a_{i,R} + ja_{i,I}$. Once a symbol $s_t = a_i$ is transmitted through a channel, it will get distorted by noise. The received symbol (r_t) will thus be a deviation from the originally transmitted s_t. In an AWGN channel for example, $r_t = s_t + d_t$, where $d_t = d_{t,R} + jd_{t,I}$ and $d_i, d_j \sim \mathcal{N}(0, \sigma_d^2)$. The received symbols sequence $(r_t|_{t=1}^T)$ is then used for decoding.

ML decoding is done by searching for the optimum path p_{opt} among all possible paths $\mathbb{P}$ in the trellis diagram of the code. Viterbi algorithm minimizes the number of paths that need to be searched by keeping the most probable path (surviving path) to each state and eliminating all others, so that at every time instant t, there is only one surviving path for each state. Each path $p \in \mathbb{P}$ has a path metric (PM) which is calculated by adding all the branch metrics $(\mu_{p,t}|_{t=1}^T)$ of all transitions along p.

$$
\begin{aligned}
PM(p) &= \sum_{t=1}^T \mu_{p,t} \\
p_{opt} &= \arg\min_{p \in \mathbb{P}} PM(p).
\end{aligned}
\tag{2.2}
$$

There are two ways to calculate the branch metric for each transition (i.e. Hard and Soft decoding). Each trellis branch has a corresponding output associated with it (i.e the n-bits). Let these n-bits be represented as $\eta_{p,t} = \{\eta_{p,t,1}, \eta_{p,t,2}, \ldots, \eta_{p,t,n}\}$ depending on the trellis path and time, and let $\alpha_{p,t}$ be the mapping of $\eta_{p,t}$ into $\mathbb{A}$. In hard decoding,

16

the branch metric of any transition is the Hamming distance between the demodulated symbol r_t and the branch symbol $\eta_{p,t}$ and is calculated as follows:

$$\mu_{p,t} = \sum_{j=1}^{n} (\eta_{p,t,j} \oplus \rho_{t,j}) \tag{2.3}$$

where $\oplus$ is the XOR binary operator, and $\rho_t = \{\rho_{t,1}, \rho_{t,2}, \ldots, \rho_{t,n}\}$ is the nearest symbol in $\mathbb{A}$ mapped back to its original n-bits.

$$\rho_t = (i)_b : \left\{ a_i = \arg \min_{\forall a_i \in \mathbb{A}} \|r_t - a_i\|^2 \right\} \tag{2.4}$$

where $(i)_b$ is the n-bits binary representation of the ith symbol $a_i \in \mathbb{A}$.

As for soft decoding, the branch metric $\mu_{p,t}$ depends on the Euclidean distance between the received symbol r_t and the branch symbol $\alpha_{p,t}$

$$\mu_{p,t} = \|r_t - \alpha_{p,t}\|^2 \tag{2.5}$$

When p_{opt} is found, a look at the input k-bits along p_{opt} will give the optimum decoded information sequence.

2.1.2 Catastrophic Codes

Catastrophic codes are those convolutional codes that exhibit a characteristic behavior called *catastrophic error propagation*, where a small number of errors in the received sequence, transmitted through a noisy channel, can result in a large error sequence in the decoded information bits.

Catastrophic codes are easily recognized from the state diagram of the code. A code is defined as catastrophic when a non-zero state returns back to the same state through a zero Hamming distance path. For example take the code with rate $1/3$ and $G = [5, 3, 6]$, this code has a state diagram shown in Figure 2.4. The numbers on each transition represent the corresponding input/output. This means that one can loop around this
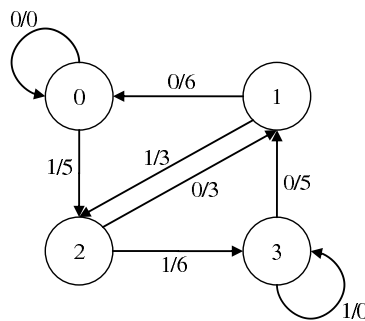

Figure 2.4: Catastrophic Code ($G = [5, 3, 6]$) State Diagram

zero-distance path an infinite number of times without increasing the distance relative to the all-zero path and thus result in an infinite number of errors. A trellis diagram of
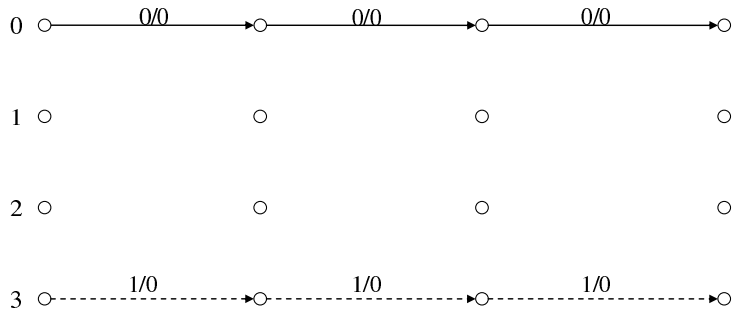
Figure 2.5: Trellis Diagram of a Catastrophic code with looping around a wrong state

such a case is shown in Figure 2.5, where the zero loop is mistaken for the loop around state 3.

A necessary and sufficient condition for catastrophic convolutional encoders was obtained in [35]; Let $\Delta_i(D)$ be the determinant of the ith $k \times k$ minor matrix of $G(D)$, then $G(D)$ represents a non-catastrophic code if and only if

$$\text{GCD} \, \{\Delta_1(D), \ldots, \Delta_{\binom{n}{k}}(D)\} \;\; = \;\; D^d \tag{2.6}$$

for some $d \geq 0$, where GCD is the Greatest Common Divisor of the polynomials. These codes are non-catastrophic because they have an inverse function and hence a unique decoded sequence, while on the other hand, catastrophic codes do not have this unique inverse and hence more than one possible decoded sequence. For this reason, catastrophic codes are avoided in real systems designs.

2.2   Study of Catastrophic Codes

In this section, we will present a study of catastrophic codes to better understand their behavior and analyze their BER curves. In this study, will introduce several new ideas related specifically to catastrophic codes, which will enables us to look at what actually happens in catastrophic codes, how to understand it and what can we do to control it.

2.2.1   Parallel Paths and Parallel States

The first notion we will introduce is the parallel paths and parallel states, which is very important when discussing catastrophic codes and their behavior. In any normal non-catastrophic convolutional code, the most common decoder used is the soft-decision Viterbi decoder, which minimizes the sequence errors as it finds the most likely transmitted sequences of messages yielding the shortest Euclidean distance from the received signal. In fact, this decoder works great because of one simple but yet very important reason. In all convolutional codes, there is only one unique *"Correct Path"* which is the actual trellis path taken by the encoder before transmission. Non-catastrophic codes does not have any *Parallel Paths* to this unique *"Correct Path"* and hence, it is the only possible path to follow while decoding. A *Parallel Path* can be defined as a path that

can be mistaken which, given the same sequence of received symbols, yields a different uncoded message than the correct path. A more detailed definition with some examples will follow afterwords in the case of catastrophic codes. As for non-catastrophic codes, it is known that errors happen all the time in any transmission, and may cause a slight deviation in the decoded trellis path, but eventually all the deviated wrong paths will merge again with the One and Only *"Correct Path"*. A simple visual analogy would clear the concept; Suppose there is a long iron rod with a small magnetic ball attached to it, the rod represents the correct path of the trellis and the ball represents the decoded sequence. Noise can be represented by shaking the rod a little. Hence, if no noise is present, the small magnetic ball will move along the iron rod with no problems; while if some noise is present, the ball will jump up and down leaving the correct path (rod) a little but eventually end up coming back to the rod again.

Now let us look at an example of a normal 4-states, rate 1/3 convolutional code using generating polynomials $G = [4, 5, 1]$ having a trellis as shown in Fig 2.6. The numbers on each branch represent the corresponding input/output. As it can be seen, there is
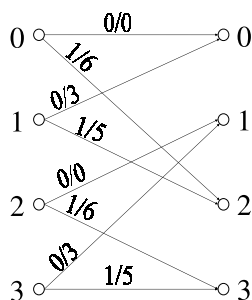
Figure 2.6: Trellis Diagram for the 4 states 1/3 rate code with G=[4,5,1]

no all-zeros loop from any non-zero state to itself, hence this code is not catastrophic. Suppose that an error-free signal of 3 symbols is received consisting of 3 $n-$bit zeros (i.e 000 000 000), the decoded sequence will be 0 0 0. This decoded path is the same *"Correct Path"* that was used to encode this message and hence, no confusion is present in any non-catastrophic code.

The case in catastrophic codes is different, because there is at least one *Parallel Trellis Path* to the actual *"Correct Path"* where the decoded trellis sequence might get stuck. Let us look at the definition of what a *"Parallel Path"* is. Take a look first at parallel lines, they are two or more lines that does not share the same starting point and continue endlessly without meeting. The same can be said about parallel trellis paths. Two paths are said to be parallel if, when decoding the same sequence of received symbols, they start from different code states and continue without intersecting to the end yielding the same path metrics, and hence, both paths are equally likely. At each time instant (i.e. at each trellis section) along the whole trellis, the state of the first path and the state of the second parallel path are together called *"Parallel Trellis States"*.

By definition, catastrophic codes are convolutional codes that contain a non-zero state which can come back to itself yielding an encoded sequence of all-zeros. The transitions

involved going from that state and returning back can be described as being parallel to the all-zero states transitions. Thus it can be easily put in the notion of *"Parallel Paths and States"* by taking the path taken from the non-zero state (which would be a *"Parallel State"* to the zero state) to itself as a *"Parallel Path"* to the all-zero states path.

As another example, consider a catastrophic 4-states 1/3 convolutional code with generating polynomials $G = [5, 3, 6]$ having its trellis shown in Fig 2.7. In this trellis, aside
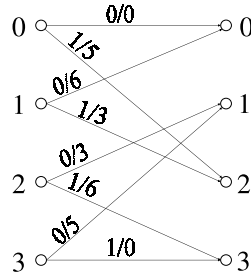

Figure 2.7: Trellis Diagram for the 4 states 1/3 code with $G = [5, 3, 6]$

from the normal loop in the zero state, there is one more loop in state 3, where it returns to itself with an output of zero and non-zero inputs. This indicates the presence of a parallel path that may trick the decoder and suck it inside an endless sequence of wrong paths. Suppose as before, that an error-free signal of 3 symbols is received consisting of 3 $n-$bit zeros. Looking at the trellis, it can be seen that there are 2 possible paths to be taken; The all-zero path resulting in a decoded bits sequence of 0 0 0, or the all state 3 path resulting in decoded bits 1 1 1. And hence, it is easily seen that states 0 and 3 are parallel.

As seen in the previous example, any catastrophic code has at least two parallel paths, the *"Correct Path"* that is no longer the one and only path present as in normal convolutional codes. We could end up with at least one other parallel path which can be called the *"Wrong Path"*. When errors happen, the transitions in the decoded path no longer follow the one and only correct path, because there is another path out there which can easily be followed that will not increase the path metric. If we return to the iron rod and the small magnetic ball example, a catastrophic code can be said to have more than one iron rod (the correct rod, which the ball should stick to and the wrong rod which acts as a parallel path to the correct one). In this case, when some excess noise happens, the magnetic ball will jump from the correct rod to the wrong one and stick with the wrong rod until another powerful error happens that takes it back again to the first correct rod.

The number of parallel paths and states can be easily obtained by studying the code. The number of parallel paths corresponds to the number of states from which a received sequence of all-zeros drive back to the same starting states. These states are parallel states to each other and form one of the groups of parallel states in the given code. The other states which are not part of this group will form other groups by using the same principle. Rather than using an output sequence of all-zeros, other sequence of outputs can be used. A *"Parallel States Group"* is a group of parallel states sharing the same set of outputs permuted between all the possible outgoing branches from each parallel state.

In previous example, state 3 is considered parallel to state 0 and thus they both form a parallel states group, since they both have the same set of outputs {0,5} but for different inputs. If we take the other states (1 and 2) that are not part of the first group, it can be noticed that they too are parallel to each other, in the sense that they both return to themselves having a sequence of two 3's as output, and hence form another parallel group.

To further clarify the notations, take another example of a 4-state but 2/3 catastrophic convolutional code having generating polynomials of $G = [5, 12, 17]$ with its trellis shown below in Fig 2.8. Because of the many branches found in this diagram, the branch labels are put in order on the right side of the figure, showing the input/output of the incoming branches into the ending state. As seen, this code has 3 self-loops other than the zero
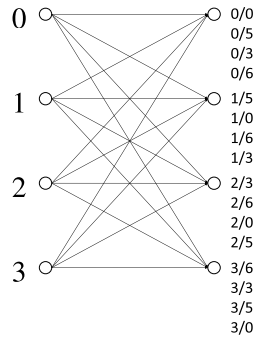
Figure 2.8: Trellis Diagram for the 4 states 2/3 code with $G = [5, 12, 17]$

loop, since all states return to themselves with a received zero making them all parallel to each other. Hence, if the same error-free signal of 3 symbols is received consisting of 3 $n-$bit zeros, it can be decoded in 4 different ways (each corresponding to a different parallel path). It can decoded by passing through the all-zero states resulting in decoded bits of 00 00 00, or the all-one states resulting in 01 01 01, the all-two state resulting in 10 10 10 or the all-three states resulting in 11 11 11. Since all states are parallel to each other, this code has one parallel states group including all states. Hence there are 4 parallel paths that a received sequence can be decoded through, only one is correct, and all other 3 being wrong.

2.2.2 Catastrophic Events

After introducing the parallel paths and parallel states, the question that need to be answered is what is a catastrophic event and how it is initialized. A catastrophic event can be defined as the sequence of errors that shift the decoding from one path to another parallel path. This shift can be identified when comparing the actual encoded correct path with the decoded path, the error event is that sequence of errors that lead the decoded path to a state parallel to the state of the correct path, when they both have started from the same state. In soft-decision Viterbi decoding, this means that the Euclidean distance between the received signal and the path leading to the parallel state is smaller than the distance between the received signal and the correctly transmitted path. Since after any catastrophic event, the path metric at the wrong parallel state is less than that of the correct state, and also the two states are parallel, then all the subsequent coming

signals will affect the two paths in the same manner and thus the path metric of both will increase in exactly the same amount keeping the wrong parallel path in the lead unless another catastrophic event takes place.

Note that there is a big difference between a catastrophic event or error and a regular error that happens in any usual transmission. The normal errors deviate the decoded path from the correct path for a small period of time, after which they will join again given that the signals are received correctly for some time after the error. While a catastrophic error shifts the correct path to a complete parallel path that they will never join again even if the received signals were perfectly error-free. Only another catastrophic error has a chance of joining them again.

Example: Look at the code in Fig 2.7 and suppose that an all-zero message is transmitted, making the correct trellis path consisting of the sequence of zero states (note that states 0 and 3 are parallel). Since noise is present in any transmission, the fist two received output symbols may be closer to the sequence of outputs 5 and 6 rather than the correct transmitted sequence of two zeros. When that happens, the path metric at state 3 will be less than that at state 0 causing a catastrophic error. See Fig 2.9. Since a catas-
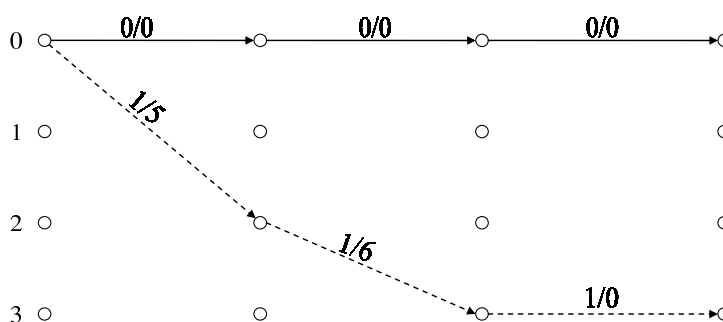
Figure 2.9: Trellis Diagram of a Catastrophic Event

trophic error already took place, even if the noise burst is ended and the next signal is received correctly as zero, the decoded path will never go back to the previous path (since they are parallel paths) and thus the decoder will be sucked in an endless loop of wrong decoding until another catastrophic event happens taking it away and putting it in another parallel path (the correct path in case of only two parallel paths as in this example).

As it can be seen, this event started at state 0 and ended at the parallel state 3, rather than the correct state 0. This catastrophic event can be said to have a length of 2 (the length is the period from the start of the error burst until ending up in a parallel state). In this code there are 16 possible catastrophic events of such length, since we have 4 states to begin and end with, and only one wrong parallel path, remembering that this code has only two parallel paths. The list of all these 16 events can be found below in Table 2.1, where 0-0-0 represents an evolution of state transitions. Notice that all wrongly received paths end with a parallel state to the actual transmitted path. The transmitted signals are presented as tx1 and tx2 for the first and second transmissions respectively and rx1 and rx2, represent the first and second received signals. As it can be seen, Table 2.1 exhausts all possible paths of length 2. Catastrophic events of length larger than 2 are

22

Table 2.1: All possible catastrophic events of length 2 in the code used in Fig 2.7.

Transmitted Path States	Received Path States	Signals [(tx1,tx2),(rx1,rx2)]
0-0-0	0-2-3	[(0,0),(5,6)]
0-0-2	0-2-1	[(0,5),(5,3)]
0-2-1	0-0-2	[(5,3),(0,5)]
0-2-3	0-0-0	[(5,6),(0,0)]
1-0-0	1-2-3	[(6,0),(3,6)]
1-0-2	1-2-1	[(5,6),(3,3)]
1-2-1	1-0-2	[(3,3),(5,6)]
1-2-3	1-0-0	[(3,6),(6,0)]
2-1-0	2-3-3	[(3,6),(6,0)]
2-1-2	2-3-1	[(3,3),(6,5)]
2-3-1	2-1-2	[(6,5),(3,3)]
2-3-3	2-1-0	[(6,0),(3,6)]
3-1-0	3-3-3	[(5,6),(0,0)]
3-1-2	3-3-1	[(5,3),(0,5)]
3-3-1	3-1-2	[(0,5),(5,3)]
3-3-3	3-1-0	[(0,0),(5,6)]

sure to be found in any code, but from our observations while working on this particular code, the errors dominating the low SNR region always happen to have a length of 2, while they disappear as SNR increases. This observation is later emphasized when we perform theoretical calculation based on this assumption, and results in theoretical curves that almost exactly match the simulations.

2.2.3 Decoded Bits Behavior After One or More Catastrophic Events

As mentioned before, a catastrophic event shifts the decoded path from one parallel path to another. Hence, between any two catastrophic events, the path in between will be one of the possible parallel paths discussed earlier. Since parallel paths are related in one way or another, a relation between decoded bits in different parallel paths is expected. Let us look at the two catastrophic codes found above in Figures 2.7 and 2.8

At any instant in a trellis, most probably the decoded state is in the same parallel group as the correct state since errors usually have a low probability of happening while catastrophic events have their effect long after the actual occurrence of the event. Take the first catastrophic code in Fig 2.7 as an example, this code has 2 parallel paths and thus 2 parallel states per parallel group. Hence, if the correct state is 0, the decoded state is either 0 or 3 depending on the current parallel path taken in decoding. Of course if the decoded state is the same as the correct state, then the parallel path taken is the correct one and thus no errors should occur. But what if the decoded state is 3, which definitely belong to the other wrong parallel path? To see the effect, we need to see what happens to the next received signals and how they are decoded. The next received

symbol should be either 0 or 5, since they represent the two branches emerging from state 0, while at the same time, they also represent the two branches leaving from state 3, but with different information bits (input). Thus, whatever the actual value of the next signal, the decoding will not sense anything wrong even if it is at state 3, and will make the decision accordingly. Decoding the symbols back into the information bits is what matters, so we want to see the difference between decoding while starting at state 0 or state 3. Supposing that state 0 is the correct current state, an error vector can be defined to be the difference in information bits between the correct path (starting at state 0) and the wrong path (starting at state 3). Table 2.2 summarizes the differences. The same thing can be said about the other parallel group which include states 1 and 2, see Table 2.3.

Table 2.2: Decoded bits in states 0 and 3 and the corresponding error vector

Received Signal	Decoded bits when starting at		Error Vector
	state 0	state 3	
0	0	1	1
5	1	0	1

Table 2.3: Decoded bits in states 1 and 2 and the corresponding error vector

Received Signal	Decoded bits when starting at		Error Vector
	state 1	state 2	
3	1	0	1
6	0	1	1

Another helpful example to show the error vectors in a better way is to see the effect of a catastrophic event of the 2/3 4-state encoder used with its trellis shown in Fig 2.8. Since all states fall into the same parallel group, any catastrophic event will lead to even more confusion than the example above. Table 2.4 summarizes the decoded bits and error vectors for this encoder. It must be noted that the error vectors are calculated assuming that the current state in the correct parallel path is 0. However, if the current state is taken as 1,2 or 3, the error vectors will be the same but permuted between the decoded paths that might be taken by the decoder.

Table 2.4: Decoded bits in all states and the corresponding error vectors for the 2/3 encoder assuming that the correct state of transmission is state 0

Received Signal	Decoded bits when starting at state				Error vector assuming decoded parallel path is at state			
	0	1	2	3	0	1	2	3
0	00	01	10	11	00	01	10	11
3	10	11	00	01	00	01	10	11
5	01	00	11	10	00	01	10	11
6	11	10	01	00	00	01	10	11

It can be seen that error vectors correspond to the different parallel paths that might be taken while decoding. Thus at the start of any transmission, the decoder will operate in the correct parallel path since no catastrophic errors are yet present, but when a catastrophic event happens, the parallel path taken will change and thus the error vector between the correct data and the decoded data will no longer be zero. This error vector will repeat for every decoded signal until another catastrophic event takes place. Thus, when a message is received having c catastrophic errors, the decoded message will consist of $c + 1$ parts each with a specific error vector that differs from the previous one. So in the case of the sample code in Fig 2.7, the decoded message parts will oscillate between completely correct and completely wrong (assuming only catastrophic errors occur), while if the code in Figure 2.8 is used, the decoded message parts will take any value from the 4 possible error vectors that differs from the previous one. Figures 2.10 and 2.11 show an example of the error vectors between the sent and received data for both codes in Figures 2.7 and 2.8.

Figure 2.10: Sample of a 50-bit transmission containing 5 catastrophic events using the code in Figure 2.7 (Assuming that only catastrophic errors exist)

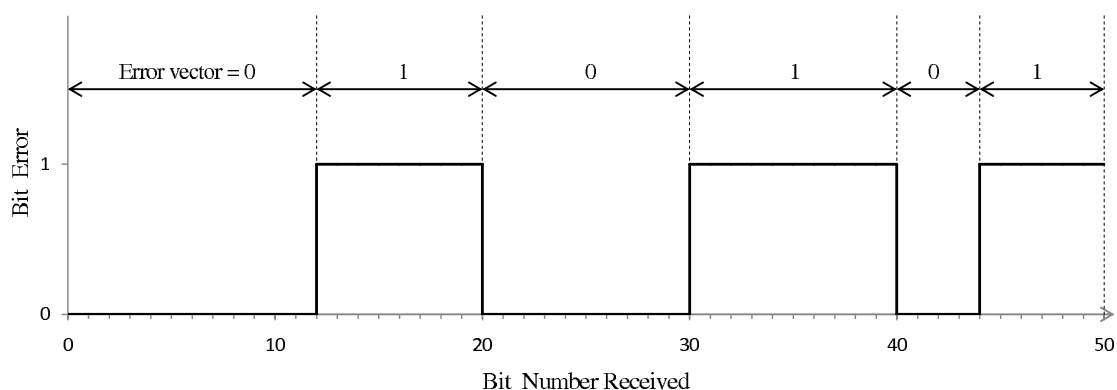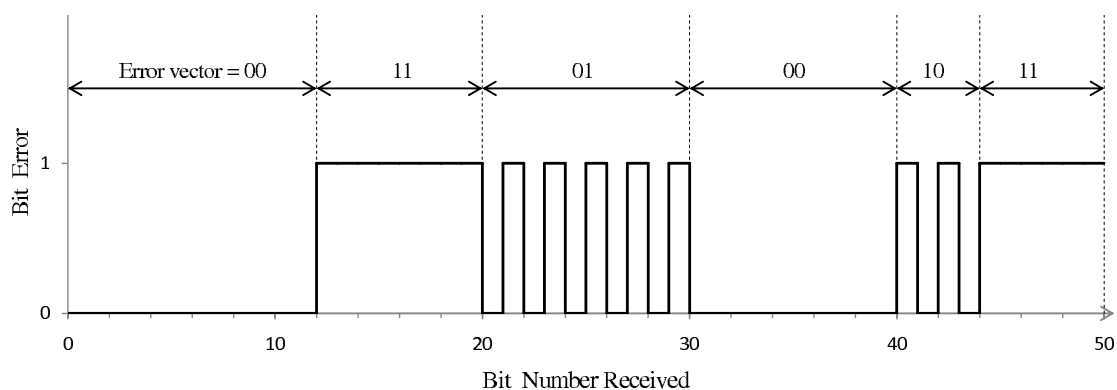Figure 2.11: Sample of a 50-bit transmission containing 5 catastrophic events using the code in Figure 2.8 (Assuming that only catastrophic errors exist)

25

2.2.4 Performance Statistics

Bit Error Rate (BER) and Packet Error Rate (PER) are used to describe the performance of communication systems. PER is mentioned because in catastrophic codes, transmitting an infinite number of bits will, with high probability, produce at least one catastrophic event. When we measure PER, a specific number of bits, which can be called a *Packet Length* (which will be discussed later), is used for transmission. Many packets are used to get an average value of PER.

- **Single Packet**

A single packet transmitted (simulated only one packet at each Signal to Noise Ratio (SNR)), will have a very interesting BER curve. For all low-medium SNR values, the BER will be very high (around 0.5), but suddenly, an SNR will come that not a single bit was in error (we are talking about packets of finite length). This means that at all lower SNRs, catastrophic errors happened skyrocketing the BER values, while at that specific SNR, no catastrophic events took place and thus no bit errors happened what so ever. Thus a very sharp transition is found. Figure 2.12 shows an example.
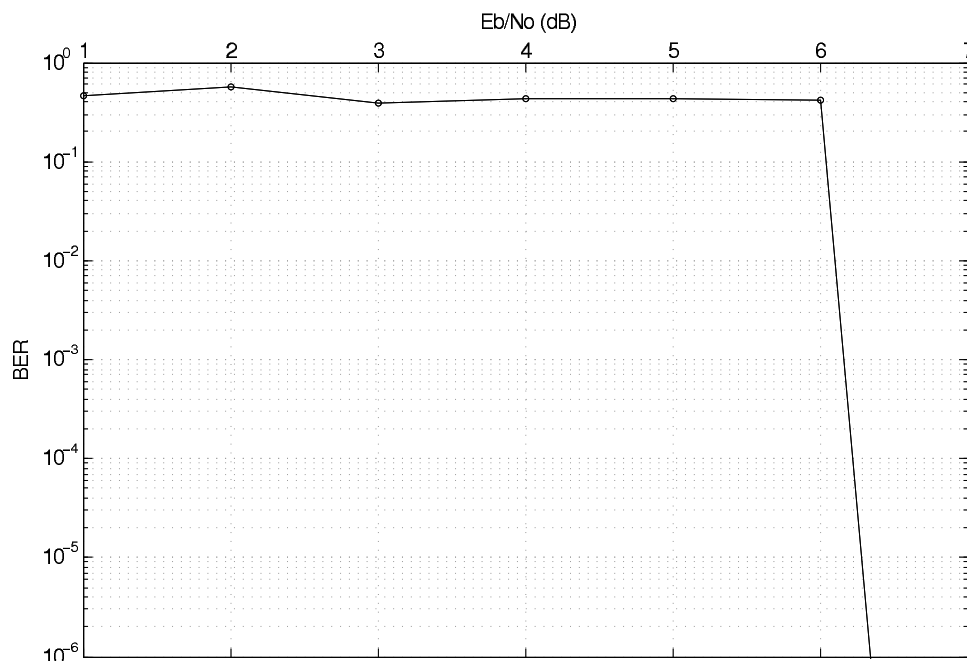
Figure 2.12: Example of a transmission of a single 1000 bit packet using the catastrophic code $G = [5, 3, 6]$

- **Large Number of Packets**

When a large number of packets is transmitted, the SNR value at the drop-off will not be the same for all packets since each transmission is unique in its own way. This urges us to take the average of all packets and put them all together in one BER curve. Another statistic that can be used here is the PER, where a packet error is defined whenever a packet is received with at least one bit error. Because of this averaging, some changes

in the BER will occur. Of course for low SNR, where all kinds of errors are high, the PER is maintained at one while BER is around 0.5, where at high SNRs, both PER and BER will become quite small. But in the middle SNR range, we no longer see that sharp transition as that when using one packet, however, it is still much steeper slope than regular convolutional codes, as seen in Figure 2.13.
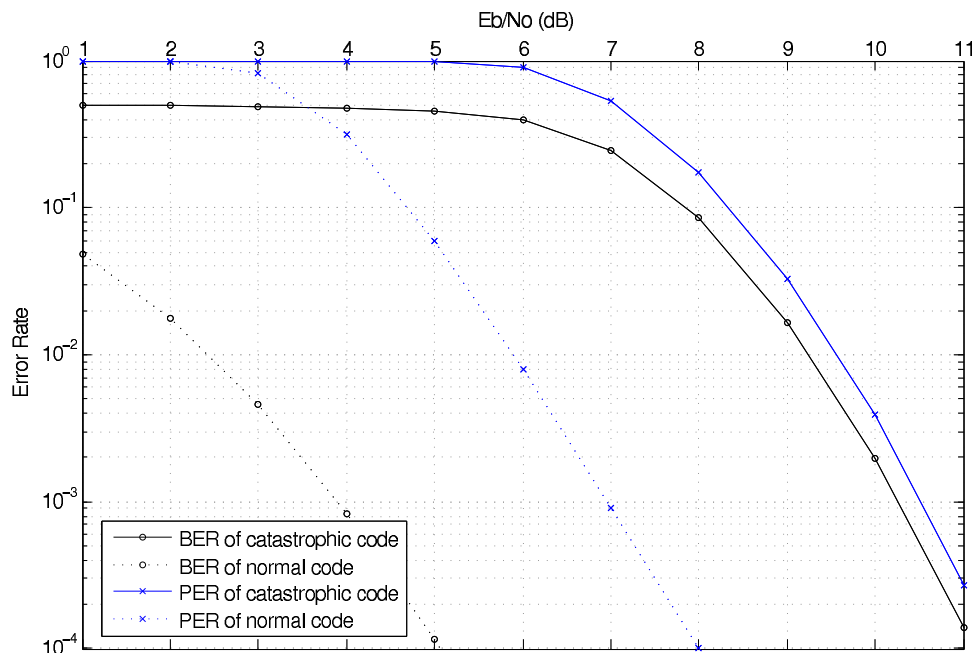
Figure 2.13: Example of a transmission of 100000 packets of 1000-bit each using the code $G = [5, 3, 6]$

At this point, it is beneficial to introduce two main types of packet received with errors.

1. Normal packet error, that contain few bits in error (similar to most coding schemes).

2. Catastrophic packet error, that contains many bit errors (around half the packet length)

In the mid-high SNR range, most of the packets will be received error free or with errors of the first type, but nevertheless, some errors of type 2 will eventually appear. Although their appearance will not affect the PER that much, but it will cause drastic consequences for BER, since, in one type 2 error, so many bits will be in error to an extent that they very likely exceed the total number of regular type 1 errors that happen in all the other packets combined, thus increasing the average BER to a value that does not precisely reflect the exact situation we have, and thus giving a false impression of the code BER performance. From here it can be seen how the sharp transition found in a single packet transmission was lost when many packets were used. Looking at the simulation results in Figure 2.13, it can be noticed that the PER and BER curves have about a 3dB difference (i.e. PER is double the BER). Take SNR = 10 dB for example, the PER $\approx 4 \times 10^{-3}$ while BER $\approx 2 \times 10^{-3}$; Knowing that the simulations included 100000 packets, then the

approximate number of error-free packets is $100000 \times (1 - 4 \times 10^{-3}) = 99600$ and only $100000 \times 4 \times 10^{-3} = 400$ packets contained errors. These 400 packets must then contain all the bit errors present. Well, there are 100000 packets of 1000 bits each, making a total of 10^8 bits and the BER $\approx 2 \times 10^{-3}$ meaning that there are about $10^8 \times 2 \times 10^{-3} = 200000$ bit errors as total. This means that the 400 error containing packets have 200000 bit errors of their total 400000 bits at a BER of 0.5. From this simple calculation it is seen that out of the 100000 packets transmitted 99600 were received error-free and 400 packets were received with a BER of about 0.5.

2.2.5   Parameters Affecting Performance and Results

• Packet Length

Catastrophic codes do not behave in the same way as all other codes. An error in any normal code will not have any big consequences, mostly, there will be some period of uncertainty after which everything will be clear again. While in a catastrophic code, a single error may lead to an infinite number of errors. Any catastrophic code will behave very much like any other code until the first catastrophic event happens, thus this first event is one of major importance. The BER will be small (comparable or even better than normal convolutional codes) until this event happens, which drags BER to very high numbers because of the large sequence of errors that follow any catastrophic event landing on a wrong parallel path (the first event will surely land on a wrong path). Hence, a very important factor is the position of the first catastrophic event in any transmission.

As in any probability problem, an infinite message length using a catastrophic code will, with high probability, have a catastrophic event and thus a high BER. This is why the length of the transmitted message becomes very important. The longer the message, the more probable a catastrophic event will occur sending the BER to astronomical highs. Thus the packet length must be considered when dealing with such codes.

To show the dependence of the catastrophic code BER curve on various packet lengths, we ran simulations using one catastrophic code (G=[5,3,6], trellis shown in Figure 2.7) while changing the packet lengths used. Take a look at the BERs and PERs of 1000, 10000 and 100000 bit length packets in Figure 2.14. It can easily be noticed that the larger the packet size, the larger the SNR required to get to the transition between high and low BER. Thus the packet length is of significant importance when talking about performance of catastrophic codes. One more observation that can be made is that the larger the packet the steeper the BER slope after the SNR threshold, theoretical calculation will confirm our observation.

• Constellation and Labeling

A second important factor is the constellation mapping. Constellation size, positioning and labeling affect all coding schemes including catastrophic codes. However, in catastrophic codes, they have a bigger effect due to the fact that catastrophic events depends highly on the Euclidean distances between the different parallel paths. Whenever the correct path and a wrong path of states are near each other, there is a high chance that
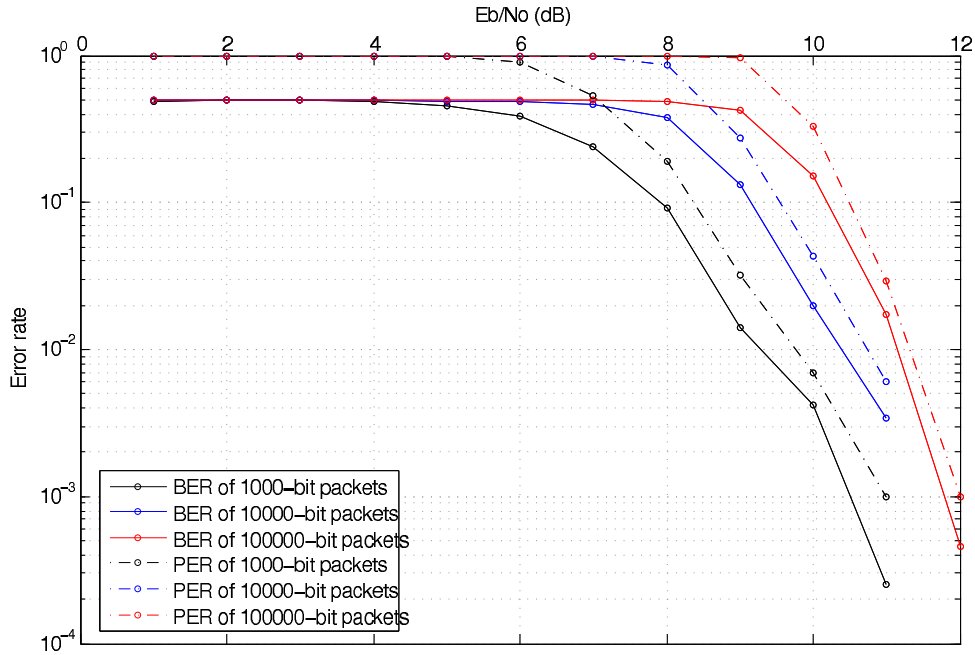
Figure 2.14: BER and PER comparison of different packet lengths

an error of catastrophic nature will occur, while on the other hand, expanding the distance as much as possible decreases the catastrophic errors probability. Thus, labeling also plays a big part in the value of the SNR where the drop of waterfall effect occurs. The larger the distance between parallel paths, the lower the SNR needed to have a transmission with no catastrophic errors.

Again, as in packet lengths, we ran simulations to show our results using the same code as above. The constellations are designed specifically for the catastrophic code being used. For this reason, we will provide a study of this particular catastrophic code, knowing that all other catastrophic codes can be studied in the same way.

Before looking at the simulation results, a deeper study of catastrophic events is needed in order to draw the constellations. From Table 2.1 we can find all the possible combinations of (rx1,tx1) and (rx2,tx2). Although there are 16 possible individual events, there are actually only 4 groups of 4 events each, where each group has the same combinations considering the Euclidean distance between the rx's and tx's. For example, the first event which has [(0,0),(5,6)] catastrophic sequence means that for a catastrophic event to happen, a 0 must be mistaken as 5, followed by another 0 mistaken as 6, but this is the same as the 4th event of [(5,6),(0,0)] only that for a catastrophic event to happen, 5 must be mistaken as 0 followed by a 6 mistaken as 0. Generally speaking, the only thing that matters is the distance between rx1 and tx1 and the distance between rx2 and tx2. Thus all the 4 possible groups are summarized in Table 2.5.

We can see that each group has a common element between (tx1,rx1) and (tx2,rx2). This element is important in the case of initializing a catastrophic event since two signals are needed to be mistaken to initiate the event, both containing this common element.

Table 2.5: All possible catastrophic groups in the code used in Fig 2.7.

(tx1,rx1)	(tx2,rx2)
(0,5)	(0,6)
(0,5)	(5,3)
(6,3)	(0,6)
(5,3)	(3,6)

Thus the squared error distance needed to start a catastrophic error is the summation of the squared distances between this common element and two others. For example, the first group in Table 2.5 represents the catastrophic events involving 0 with 5 and 0 with 6, thus the squared error distance d_{cat}^2 needed to start the catastrophic event is

$$d_{cat_1}^2 = d_{0,5}^2 + d_{0,6}^2 \qquad (2.7)$$

The same can be done to all the other groups

$$\begin{aligned} d_{cat_2}^2 &= d_{0,5}^2 + d_{5,3}^2 \\ d_{cat_3}^2 &= d_{6,3}^2 + d_{0,6}^2 \\ d_{cat_4}^2 &= d_{5,3}^2 + d_{3,6}^2 \end{aligned} \qquad (2.8)$$

An average value $d_{cat}^2 = \frac{1}{4}\sum_{i=1}^{4} d_{cat_i}^2$ can be used to describe the labeling and the BER behavior of catastrophic codes as can be seen in the following. Let us look at some examples to see the effects of constellation mapping and labeling. The code we are using has 3 output bits, which requires a constellation of size 8, however, only 4 out the 8 possible outputs are being used, specifically (0,3,5 and 6). Figure 2.15 shows 4 different constellations that are used with their specific d_{cat}^2.

Simulation results using these different constellations are shown Figures 2.16. It can be easily noticed that d_{cat}^2 is directly related to the performance of catastrophic codes. A low d_{cat}^2 means catastrophic events are easily triggered and thus BER remains high until a larger SNR, while a high d_{cat}^2 on the other hand produces a BER curve that has a lower SNR threshold. Change in constellation as it seems does not change the slope after the SNR threshold as in the case of packet length.

We can thus conclude that both packet length and constellation positioning and labeling play very important roles in the performance of catastrophic codes.

2.2.6 Theoretical Calculation

In this section, we will engage in theoretical study of the performance of catastrophic codes in terms of the analytical results on BER and PER. In order to calculate BER and PER, the probability of a catastrophic event must be calculated first. For this reason, theoretical results are also code specific, because different codes require different calculations using the same procedure. Calculating the probability of a catastrophic event involves an analytical study of all possible catastrophic events described previously in Table 2.1 (all catastrophic events are assumed to be of length 2, larger lengths were

$$d_{cat}^2 = 4 \qquad\qquad d_{cat}^2 = 6$$

$$d_{cat}^2 = 2.586 \qquad\qquad d_{cat}^2 = 7.414$$
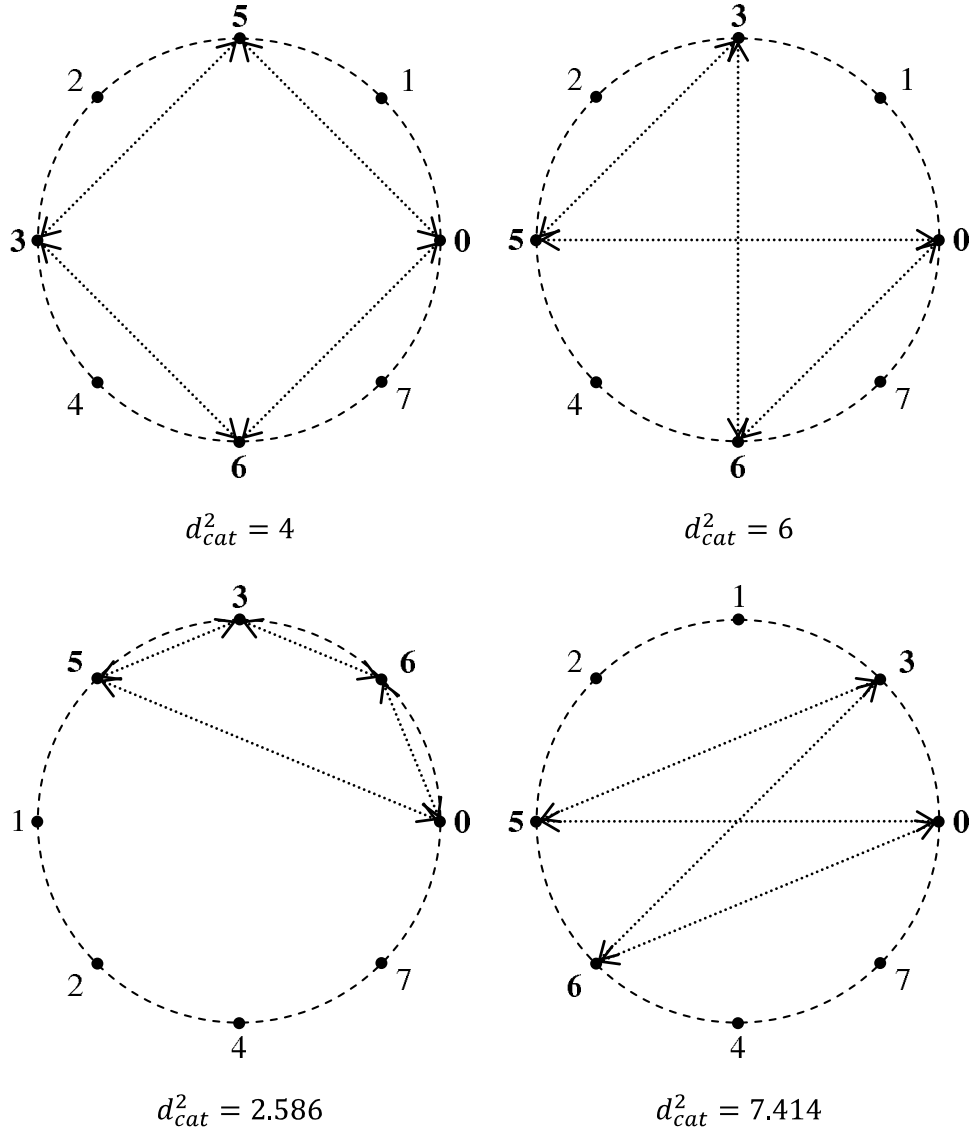
Figure 2.15: Different constellations with their corresponding d_{cat}^2

ignored because of our observation in simulations before, where only length 2 catastrophic errors were present. This assumption is verified when the calculated theoretical results match the simulations to a very high degree.). For any single event, the probability that the wrong path is selected rather than the correct path, given that the correct path was transmitted, must be calculated. We will study codes with catastrophic errors of length 2, thus let

$$
\begin{aligned}
c1 &= c_{1i} + jc_{1j} \\
c2 &= c_{2i} + jc_{2j}
\end{aligned}
\qquad (2.9)
$$

be the transmitted signals, and

$$
\begin{aligned}
r1 &= r_{1i} + jr_{1j} \\
r2 &= r_{2i} + jr_{2j}
\end{aligned}
\qquad (2.10)
$$
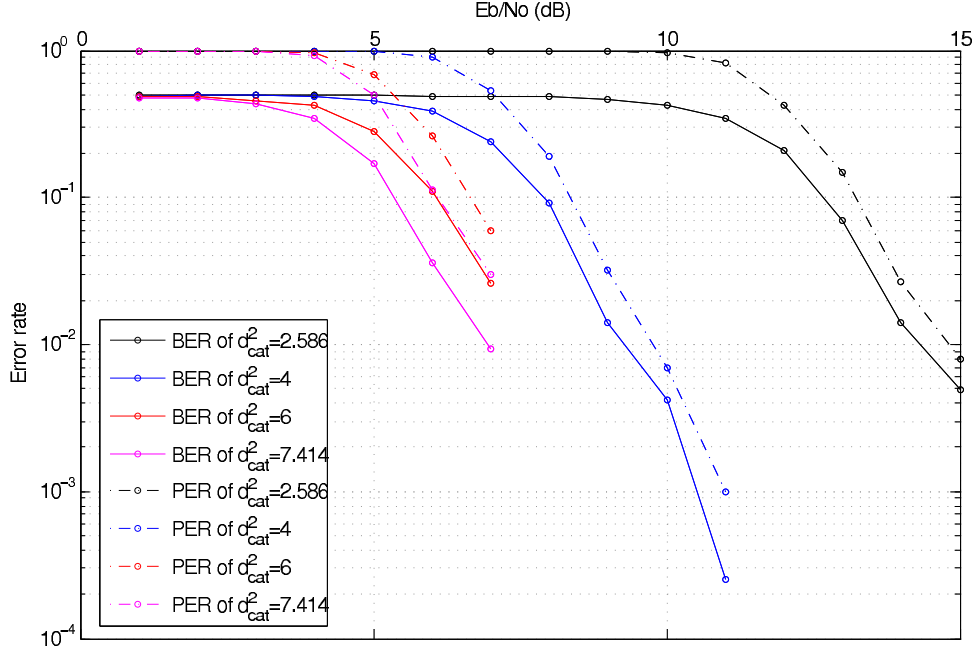
Figure 2.16: BER and PER comparison of different constellations using 1000 packets of 1000-bit each, using the same code $G = [5, 3, 6]$

be the received signals with each component

$$r_{xy} = c_{xy} + n_{xy} \tag{2.11}$$

where n_{xy} are Gaussian independent variables with a zero mean and σ^2 variance ($\mathcal{N}(0, \sigma^2)$) where $xy \in \{1i, 1j, 2i, 2j\}$ and let $w1 = w_{1i} + jw_{1j}$, $w2 = w_{2i} + jw_{2j}$ denote the wrong signals that must be received to shift the path into another parallel path. The probability of a catastrophic event given the transmitted signal $P[E|Tx]$ is thus

$$
\begin{aligned}
P[E|Tx] &= P[|r1 - w1|^2 + |r2 - w2|^2 < |r1 - c1|^2 + |r2 - c2|^2] \\
&= P[r_{1i}e_{1i} + r_{1j}e_{1j} + r_{2i}e_{2i} + r_{2j}e_{2j} < K]
\end{aligned} \tag{2.12}
$$

where

$$
\begin{aligned}
e_{xy} &= c_{xy} - w_{xy} & \forall x \in \{1, 2\}, y \in \{i, j\} \\
K &= (C - W)/2 \\
C &= c_{1i}^2 + c_{1j}^2 + c_{2i}^2 + c_{2j}^2 \\
W &= w_{1i}^2 + w_{1j}^2 + w_{2i}^2 + w_{2j}^2.
\end{aligned} \tag{2.13}
$$

Let

$$R = r_{1i}e_{1i} + r_{1j}e_{1j} + r_{2i}e_{2i} + r_{2j}e_{2j} \tag{2.14}$$

which consequently has a normal distribution of $\mathcal{N}(\mu_R, \sigma^2)$ where

$$
\begin{aligned}
\mu_R &= c_{1i}e_{1i} + c_{1j}e_{1j} + c_{2i}e_{2i} + c_{2j}e_{2j} \\
\sigma^2 &= (e_{1i}^2 + e_{1j}^2 + e_{2i}^2 + e_{2j}^2)\sigma^2
\end{aligned} \tag{2.15}
$$

since its components are all independent Gaussian random variables. It follows that,

$$P[E|Tx] \;\; = \;\; \frac{1}{2}\left(1 + erf\left(\frac{K - \mu_R}{\sigma\sqrt{2}}\right)\right) \tag{2.16}$$

where

$$erf(x) \;\; = \;\; \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt \tag{2.17}$$

Hence,

$$P[E] \;\; = \;\; \frac{1}{N}\sum_{i=1}^{N}\left[\frac{1}{2}\left(1 + erf\left(\frac{K_i - \mu_R}{\sigma\sqrt{2}}\right)\right)\right] \tag{2.18}$$

where K_i is K calculated for the ith catastrophic event and N being the total number of different catastrophic errors present in the code (16 in the case of using the code G=[5,3,6]).

$P[E]$ gives the probability of having a catastrophic event after 2 signal periods, an approximate value P_e can be used to simplify calculation as it gives the probability of a catastrophic event after only 1 period of time. This value P_e can be said to be in a case similar to a BSC channel, where there is $1 - P_e$ probability that no catastrophic errors will happen and P_e probability that a catastrophic error will happen after each symbol. Since this code cannot have a catastrophic error by the first symbol alone, the path taken will not shift to another parallel path, but rather behave as a normal error, which can be corrected if the next symbols are received correctly. Since the first symbol did not initiate the catastrophic error then it is following the BSC-like probability $1 - P_e$. The second symbol will initiate the catastrophic error and hence follow the BSC-like probability P_e. From here, it is seen that $P[E] = (1 - P_e) \times P_e \Rightarrow P_e^2 - P_e + P[E] = 0$, which makes

$$P_e \;\; = \;\; \frac{1 - \sqrt{1 - 4P[E]}}{2}. \tag{2.19}$$

A table of probability of k catastrophic errors to happen within a packet containing n transmissions can easily then be constructed using the following equation

$$P[k] \;\; = \;\; \frac{n!}{k!(n-k)!}P_e^k(1 - P_e)^{(n-k)}. \tag{2.20}$$

PER can be easily calculated by taking the catastrophic error-free transmission ($k = 0$ catastrophic errors)at all SNRs and compute PER= $1 - P[k = 0]$, where $P[k = 0]$ is the probability of having 0 catastrophic errors, and thus PER represents the probability of sending a message with n transmissions and receiving it with no catastrophic errors at all. This procedure ignores any normal errors and only focus on catastrophic ones because of the minor changes that normal errors have on PER or BER as compared to catastrophic errors are negligible.

As for BER, it can be calculated as follows:

$$BER(\sigma) \;\; = \;\; \sum_{k=1}^{n} P[k] * \frac{\lfloor (k+1)/2 \rfloor}{k+1} \tag{2.21}$$

where $\frac{\lfloor (k+1)/2 \rfloor}{k+1}$ is the average BER when k catastrophic errors are present in a code having 2 parallel paths (as the example code used in Figure 2.7). In such a code, there are $k+1$ parts between catastrophic events, which fluctuate the decoded path between the two possible error vectors (i.e. all-zeros or all-ones), $\lfloor (k+1)/2 \rfloor$ parts will correspond to the all-ones error, while the rest correspond to the all-zero error, and hence the average BER is computed on the assumption that all events are equally apart as $\frac{\lfloor (k+1)/2 \rfloor}{k+1}$. Table 2.2.6 gives the theoretical PER and BER for the example code $G = [5, 3, 6]$ using a packet length of ($n = 1000$) and $d_{cat}^2 = 4$.

Table 2.6: Theoretical PER and BER for the sample code $G = [5, 3, 6]$ using a packet length of ($n = 1000$) and $d_{cat}^2 = 4$.

	SNR per bit										
	1	2	3	4	5	6	7	8	9	10	11
PER%	100	100	100	100	99.75	90.90	53.90	17.40	3.30	0.40	$< 10^{-2}$
BER%	49.62	49.03	48.08	47.10	45.83	39.65	24.56	8.44	1.64	0.19	$< 10^{-2}$

The following Figure 2.17 shows the theoretical calculation as compared to the actual simulation results. As it can be seen from the figure, theoretical calculation is very close to simulation results.
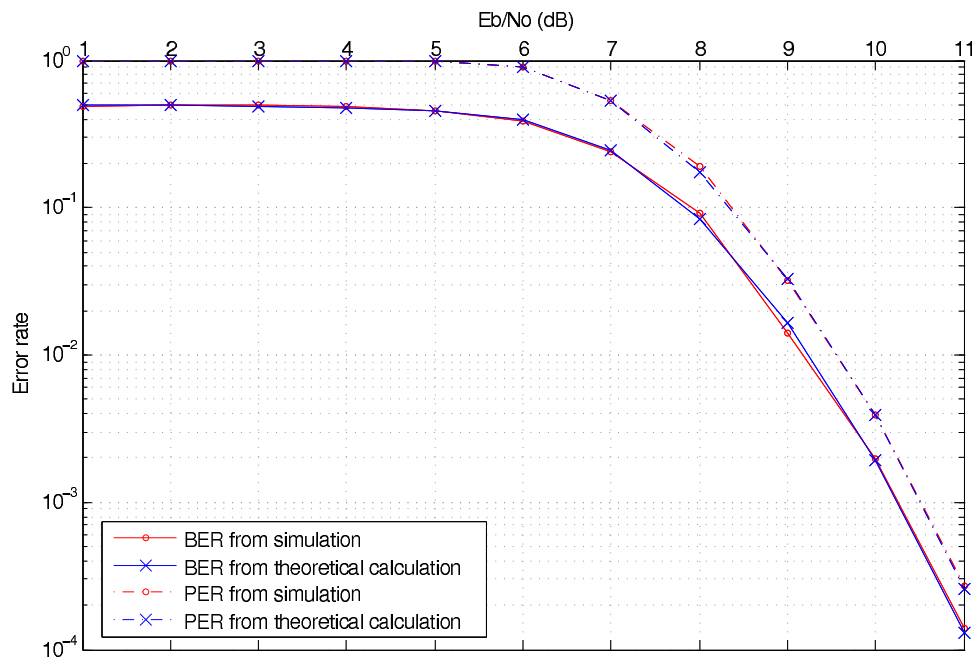
Figure 2.17: Theoretical results as compared to simulation

An interesting phenomenon is observed when calculating the theoretical BER curves for longer packet lengths (larger n). As noticed before in the simulations of different packet lengths, the slope of the BER curve after the SNR threshold increases as the packet length increases, this observation is emphasized with the theoretical results. The following Figure 2.18 shows the theoretical BER for various packet lengths using the constellation where $d_{cat}^2 = 4$. As it can be seen from the figure, the curve drop continues to shift to
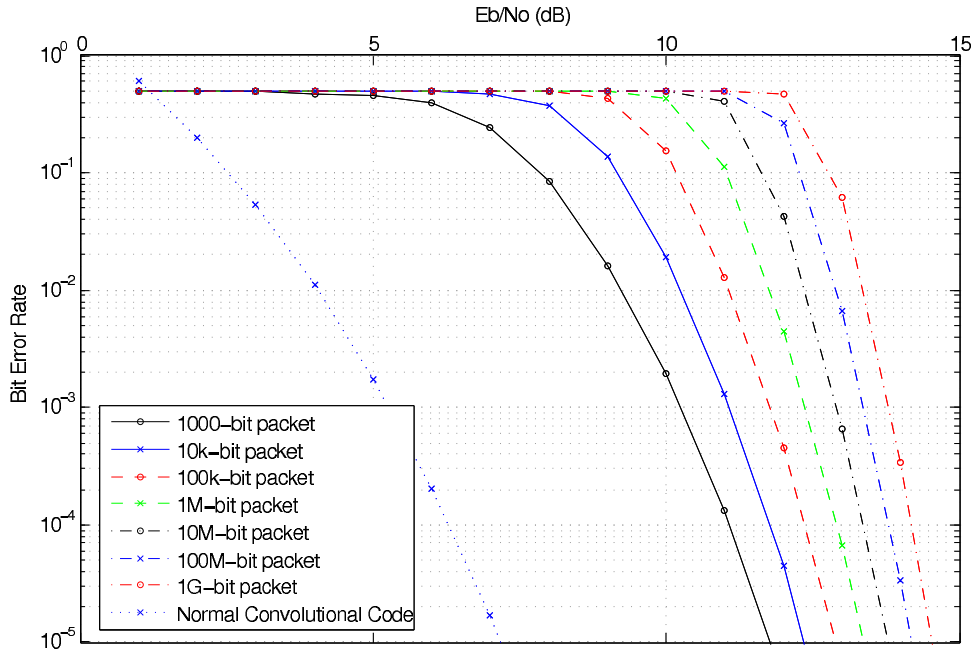
Figure 2.18: Packet length effect for the code $G = [5, 3, 6]$ with $d_{cat}^2 = 4$

larger threshold SNRs as packet length increases, while at the same time increasing the slope and nearing a waterfall phenomenon as packet length reaches infinity. Of course these results do not count the normal errors that happen in catastrophic codes, since as we said earlier, normal errors are negligible at high SNRs, but nonetheless, these normal errors are expected to determine the error floor of the waterfall drop of catastrophic codes.

In another similar BER theoretical calculation, Figure 2.19 shows the theoretical BER for various packet lengths using the constellation where $d_{cat}^2 = 6$ in Figure 2.15. In this figure, it is obvious that constellation changes did not affect the slopes of the BER after the threshold, but it did change the SNR threshold itself by shifting it about 2 dB to the left. This observation again emphasizes our previous observation made through simulations.

2.2.7 Taking Control of Catastrophic Codes BER Curves

From the previous sections, we can now conclude that the parameters of the catastrophic codes (i.e. packet length and constellation labeling) can be used to control the SNR threshold as well as the BER slope beyond the threshold. Table 2.7 summarizes the observations:

Table 2.7: Summary of catastrophic code parameters effects

Parameter	SNR Threshold	Slope after threshold
Larger Packet Length	Increase	Increase
Larger d_{cat}^2	Decrease	None
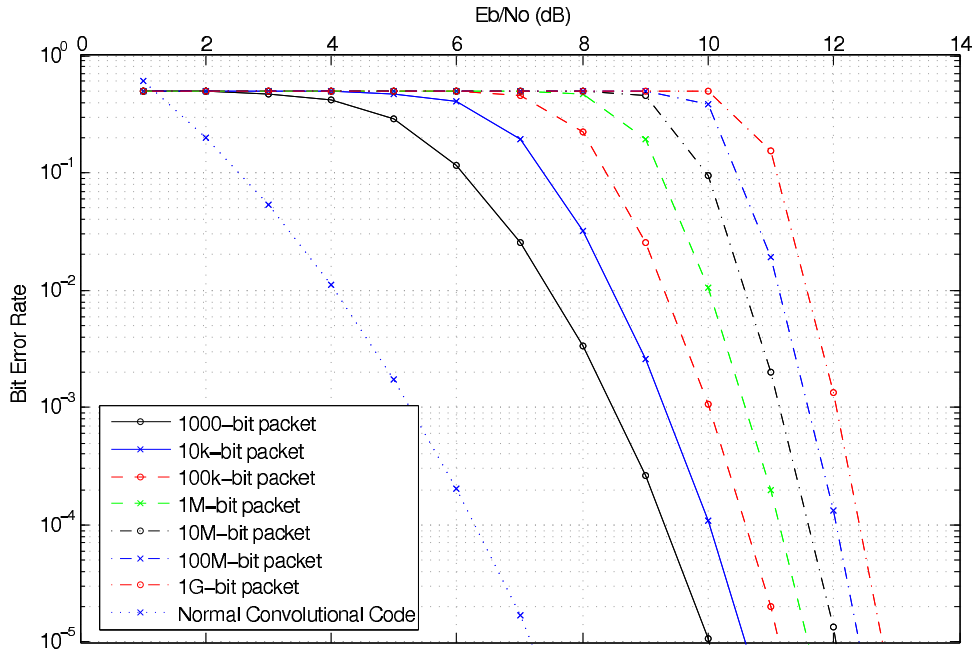
Figure 2.19: Packet length effect for the code $G = [5, 3, 6]$ with $d_{cat}^2 = 6$

As can be seen, the labeling plays the part of shifting the curves while the packet length steepens the slope and shifts the curve. A good approach is to use higher packet lengths with larger d_{cat}^2 so that to increase the slope without shifting the curve further to the right (This is done by canceling the shift that occurred when increasing the packet length, by introducing an opposite shift using the constellation and labeling). Figure 2.20 is an example done in simulation. It can be seen that we were able to gain slope without shifting the curves.

Using this approach of controlling the catastrophic codes through the packet lengths and labeling is a way of constructing the code we intended to design at first as presented in Chapter 1. Although catastrophic codes cannot produce codes below a certain SNR with a steep slope, but nonetheless, they act in a way similar to what we intended to do in the mid-high SNR region.

2.2.8 Simulation Procedure for Viterbi Decoding

In this section, we provide a flowchart illustrating how simulations are conducted. Simulation procedure is simple. At least 1000 packets of a specific number of bits (1000,10000,...,etc.) is simulated for each value of E_b/N_0. Each packet starts with all-zero state, randomly generates the data, encode it and transmitted over an AWGN channel. The data being received is decoded using the soft-Viterbi algorithm as described in section 2.1.1. The following Figure 2.21 shows a brief flowchart of the code.
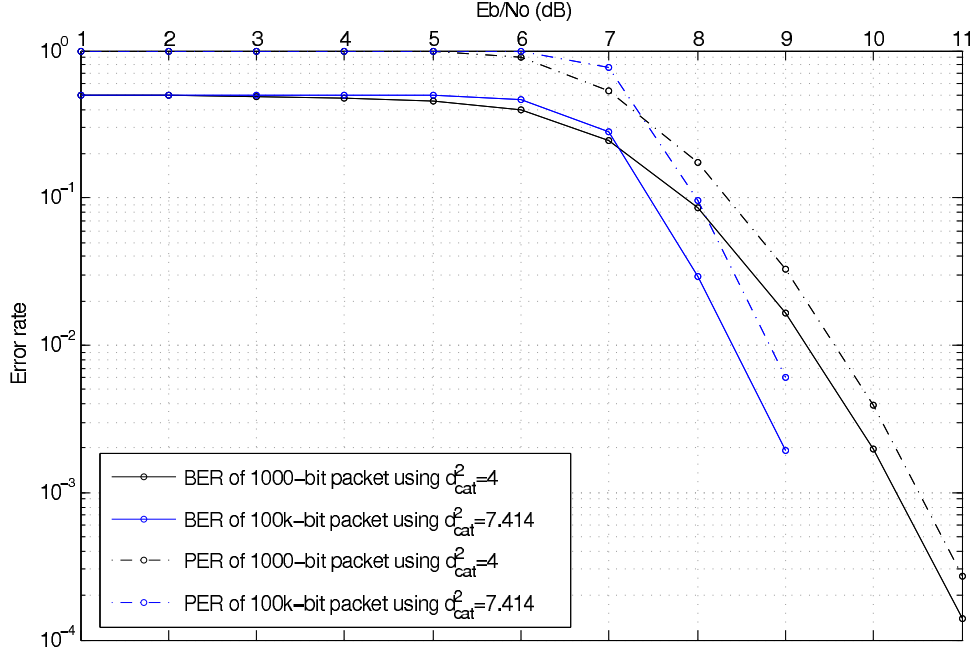
Figure 2.20: Simulation comparison between 1000 and 100000 packet lengths using $d_{cat}^2 = 4$ and $d_{cat}^2 = 7.4$ respectively

2.3 BCJR Decoding of Catastrophic Codes

2.3.1 Description of the BCJR Decoding Algorithm

Viterbi algorithm finds the most likely path that is taken in any trellis, and thus minimizes the sequence error probability (SER); i.e. it minimizes the probability $P(\hat{\mathbf{v}} \neq \mathbf{v}|\mathbf{r})$ that the decoded (ML) codeword $\hat{\mathbf{v}}$ is not equal to the transmitted codeword $\mathbf{v}$ given the received sequence $\mathbf{r}$. Although Viterbi algorithm is optimal in minimizing SER, it may not be optimal in minimizing the BER which we are actually interested in. To minimize the BER, the posteriori probability $P(\hat{u}_l = u_l|\mathbf{r})$ that an information bit u_l at is correctly decoded must be maximized. Such algorithm is called a maximum a posteriori probability (MAP) decoder.

In 1974, Bahl, Cocke, Jelinek and Raviv [36] introduced a MAP decoder, which is known as the BCJR algorithm. This algorithm calculates the a posteriori L-values

$$L(u_l) \equiv \ln\left[\frac{P(u_l = 1|\mathbf{r})}{P(u_l = 0|\mathbf{r})}\right] \tag{2.22}$$

called the APP L-values, of each information bit u_l, which are then used to decide the output which is given by

$$\hat{u}_l = \begin{cases} 1 & \text{if} \quad L(u_l) > 0 \\ 0 & \text{if} \quad L(u_l) < 0 \end{cases} \tag{2.23}$$

The decoded bits $\hat{u}_l$ are then used to compute the error sequence as $e_l = |\hat{u}_l - u_l|$.

Figure 2.21: A simplified flowchart of the code used in the simulation, using Viterbi decoding

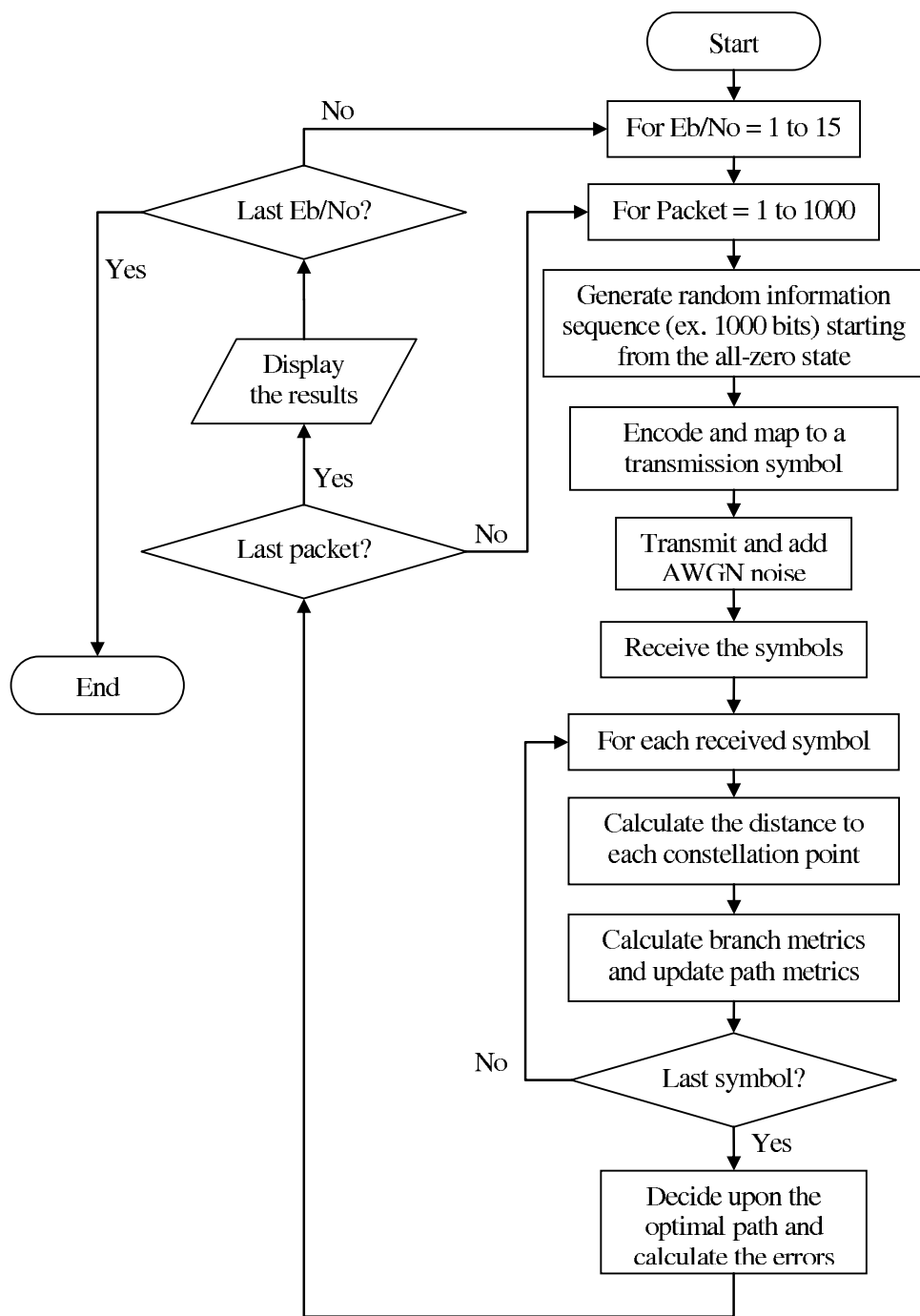The BCJR algorithm is similar to Viterbi in the sense of branch metrics and path metrics. Every received symbol will transfer the trellis from one state s' to another s, $s, s' \in \mathbb{S}$ where $\mathbb{S}$ is the total set of possible states. The branch metrics $\gamma_l(s', s)$ at symbol index l is calculated by $\frac{L_c}{2}\mathbf{r}_l.\mathbf{v}_l$ where $L_c = 4E_s/N_0$ is the channel reliability factor and E_s/N_0 is the symbol energy to noise ratio (SNR). The branch metrics are then used to calculate two other metrics, the forward $\alpha_l(s)$ metric (similar to Viterbi path metric) and the backwards $\beta_{l-1}(s')$ metric (a reversed version of the forward metric). In the Log-domain BCJR algorithm, where there are K transmitted symbols of a $1/n$ code rate, the forward metrics are calculated as follows:

$$\alpha_0(s) = \begin{cases} 0 & s = 0 \\ -\infty & s \neq 0 \end{cases} \tag{2.24}$$

where $\alpha_0(s)$ is the initial forward metric, $\alpha_0(0)$ is initialized to zero since in every packet transmission, we start from the all-zero state, while all other $\alpha_0(s)$ are set to $-\infty$. And

$$\alpha_l(s) = \ln \sum_{s' \in \sigma_{l-1}} e^{[\gamma_l(s',s) + \alpha_{l-1}(s')]} \tag{2.25}$$

for $l = \{1, 2, \ldots, K\}$, where σ_l is the set of all possible states at symbol index l. In order to calculate the backward metrics, we have to initialize the last state of the trellis. There are two ways to do that depending on the situation, if the transmission is always terminated by a sequence of zeros to drive the last state to all-zero state then

$$\beta_K(s) = \begin{cases} 0 & s = 0 \\ -\infty & s \neq 0 \end{cases} \tag{2.26}$$

only if the whole sequence of received data is available. However, if the sequence is not terminated by zeros, or the received data does not represent the whole sequence, then $\beta_K(s)$ is initialized by the value of $\alpha_K(s)$

$$\beta_K(s) = \alpha_K(s) \quad , \quad \forall s \in \sigma_K \tag{2.27}$$

After initializing $\beta_K(s)$, the rest of the backward metrics can then be calculated as follows:

$$\beta_{l-1}(s') = \ln \sum_{s' \in \sigma_l} e^{[\gamma_l(s',s) + \beta_l(s)]} \tag{2.28}$$

When all $\alpha_l(s), \beta_l(s)$ and $\gamma_l(s)$ are available, then the APP L-values $L(u_l)$ are calculated as follows:

$$L(u_l) = \ln \left\{ \sum_{(s',s) \in \sum_l^1} e^{\alpha_{l-1}(s') + \gamma_l(s',s) + \beta_l(s)} \right\} - \ln \left\{ \sum_{(s',s) \in \sum_l^0} e^{\alpha_{l-1}(s') + \gamma_l(s',s) + \beta_l(s)} \right\} \tag{2.29}$$

where $(s', s) \in \sum_l^1$ corresponds to all the state transitions $s' \to s$ at time l such that the input bit u_l is one, while $(s', s) \in \sum_l^0$ corresponds to all state transitions such that the input bit u_l is zero.

2.3.2 Simulation Results

The simulations results of convolutional codes decoded using the BCJR algorithm did not enhance the BER curves as compared to the Viterbi decoding algorithm. This is seen in Figure 2.22. This observation is shared in [37], where the authors conclude that both algorithms produce very similar results especially when the information bits {0,1} are i.i.d with {0.5,0.5} probabilities.
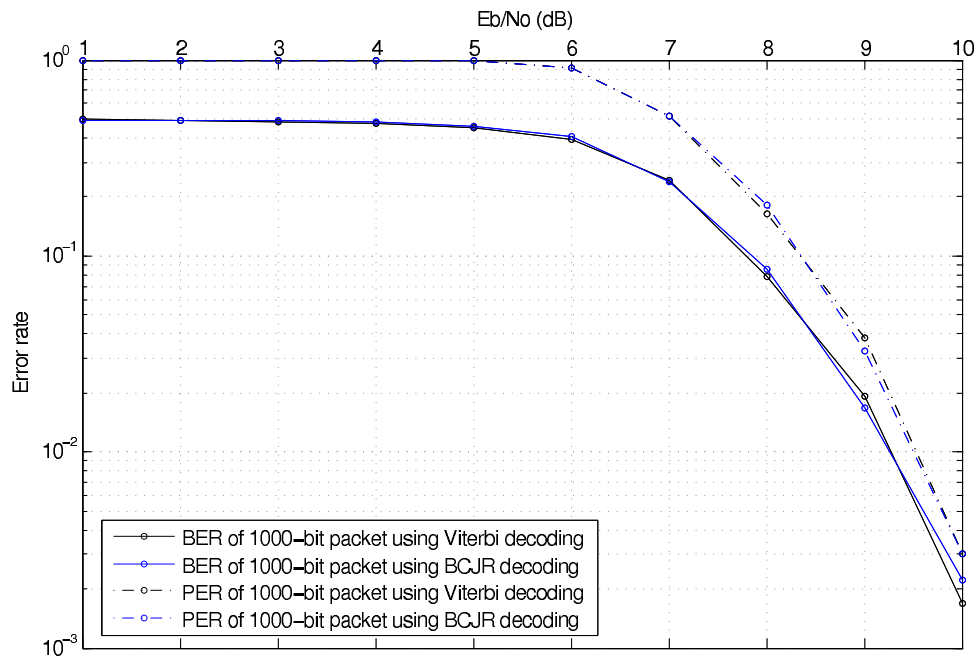
Figure 2.22: BCJR decoding produces similar results to Viterbi decoding using the 1/3 code G=[5,3,6] and $d_{cat}^2 = 4$

2.4 Terminating Simulation with All-Zero State

Until now, all discussion involved coding and simulation of a truncated code (i.e. not terminated with all-zero state). In normal codes, terminating the simulation with a sequence of zeros forcing the final state to go to zero does not affect the actual performance of the code, however, in catastrophic codes, the situation is different. As described earlier in this chapter, the decoded path of a catastrophic code fluctuates between the different parallel paths present in a given code. At the beginning of every packet transmission, the starting state is fixed (usually all-zero state) and thus, the decoded path at the section before the first catastrophic error is always the correct path as presented earlier. However, after that first catastrophic error, the decoded path will fluctuate between all the parallel

40

paths. Forcing the trellis to go to the all-zero state at the end of the packet will give a second known point in decoding which forces the decoded path to consist of the correct path not only at the beginning of the trellis but also at the end of it. This will eliminate at most one catastrophic error and thus shifts the BER curves to the left. Below, we can find some simulation results in Figure 2.23 showing the effect. It can be concluded that forcing to all-zero state is another parameter that can be used in designing the desired code which increases the slope after the threshold.
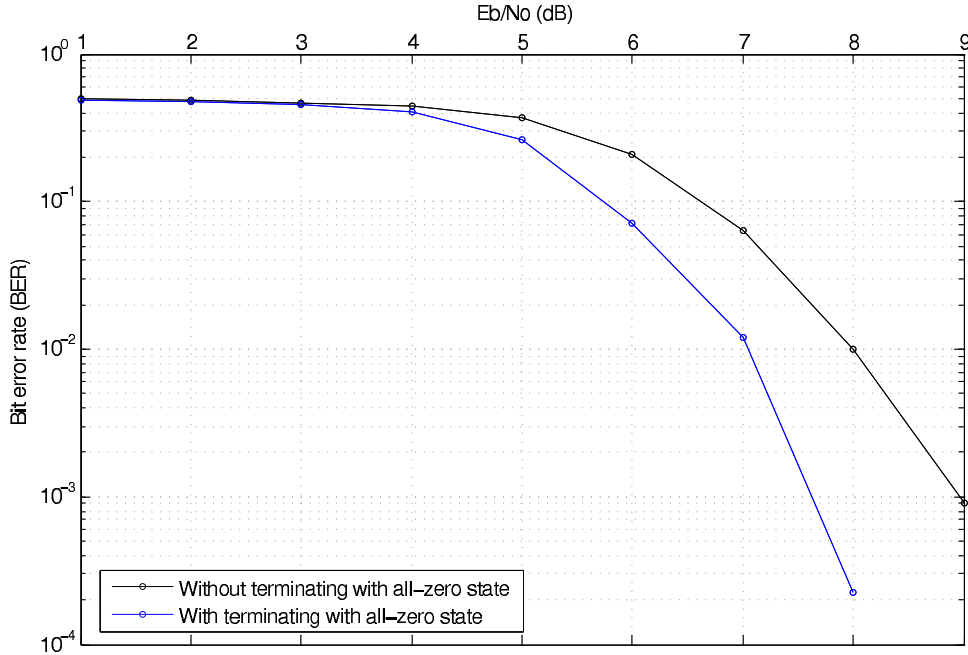
Figure 2.23: Comparing simulations with All-zero state termination. the code = $[5, 3, 6]$ with 1000-bit packet is used and $d_{cat}^2 = 6$

2.5   Second Order Statistics of Information Bit LLRs and Errors

After studying the BER curves of catastrophic codes, we wanted to see the correlation of errors and also study the information bits soft output LLRs before the actual hard decoding. This study is made hoping to infer more on catastrophic codes, especially in the low SNR region before the BER drops. As we saw previously, the catastrophic code does not produce independent errors, knowing that we have an error somewhere enables us to correct not only that error but also all of the surrounding bits. This dependence makes us wonder what Eve can infer from the received data, can she correct anything or know the positions of catastrophic errors?

To understand the results better, we simulated a normal non-catastrophic convolutional code of $1/3$ rate and having a generating matrix $G = [5, 7, 7]$. In the figures that follow, we will see 5 sub-figures in each, the 2 on the left show the soft LLR output drawn

for 3 successive 1000-bit packets in the time domain along with the error vector (one figure showing the actual value and the other the absolute value of LLRs), the error vector is such that the bits in error will have a value of 1 and those bits received correctly will have a value of 0. On the right side, there are 3 sub-figures, the top showing the empirical probability density of the soft LLRs, and the bottom two showing the auto-covariance of the LLRs' and errors.

The following Figure 2.24, Figure 2.25 and Figure 2.26 show the results for 3 different SNRs 1,3 and 6 respectively.
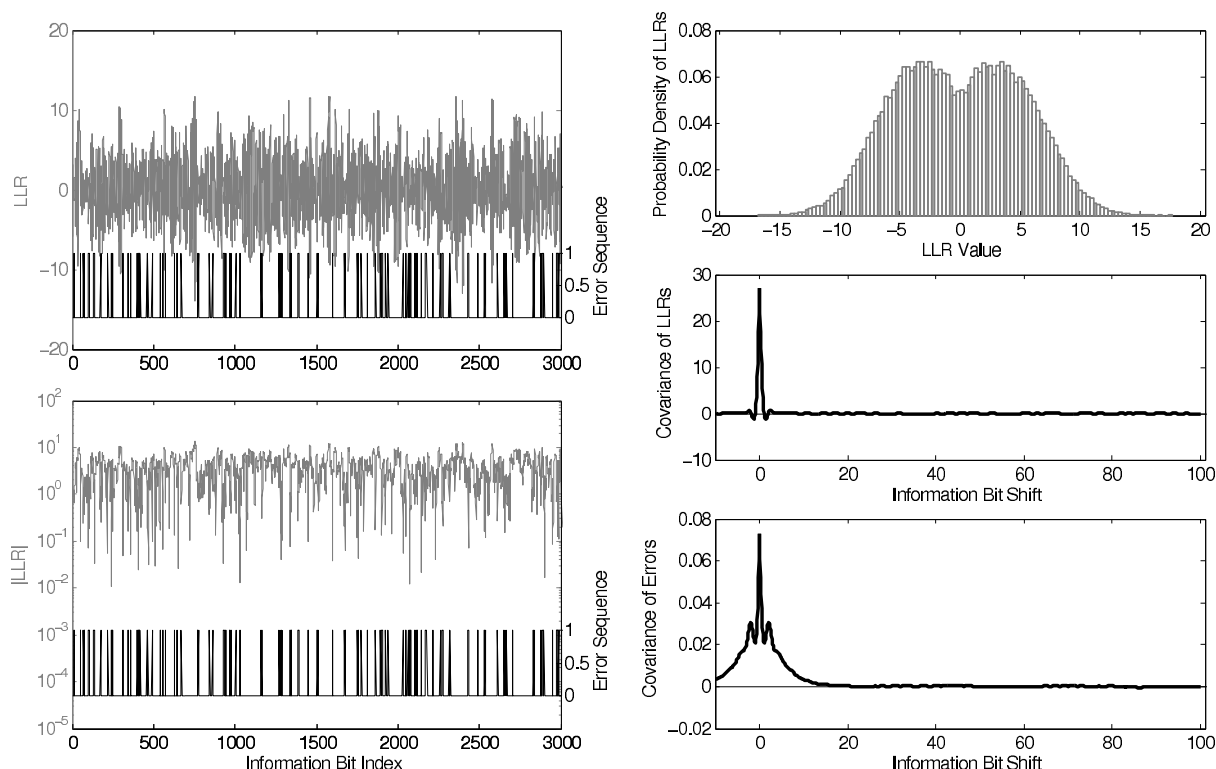
Figure 2.24: BCJR decoding of a normal convlolutional code at SNR = 1 dB (BER = $8.15{\times}10^{-2}$)

The following observations can be made from these figures regarding BCJR decoding of a normal convolutional code.

- Absolute value of LLRs $|LLR|$ increases as SNR increases.

- The LLRs probability density function is a two Gaussian-like Bell-shaped peaks representing the information bits (negative LLR for 0, and positive LLR for an information bit of 1). These two peaks move apart as SNR increases and each becomes a district Gaussian-like distribution.

- LLRs are not correlated at all.

- Error correlation is low and present only for the neighboring 10-20 bits. This correlation decreases to zero as SNR increases.

42
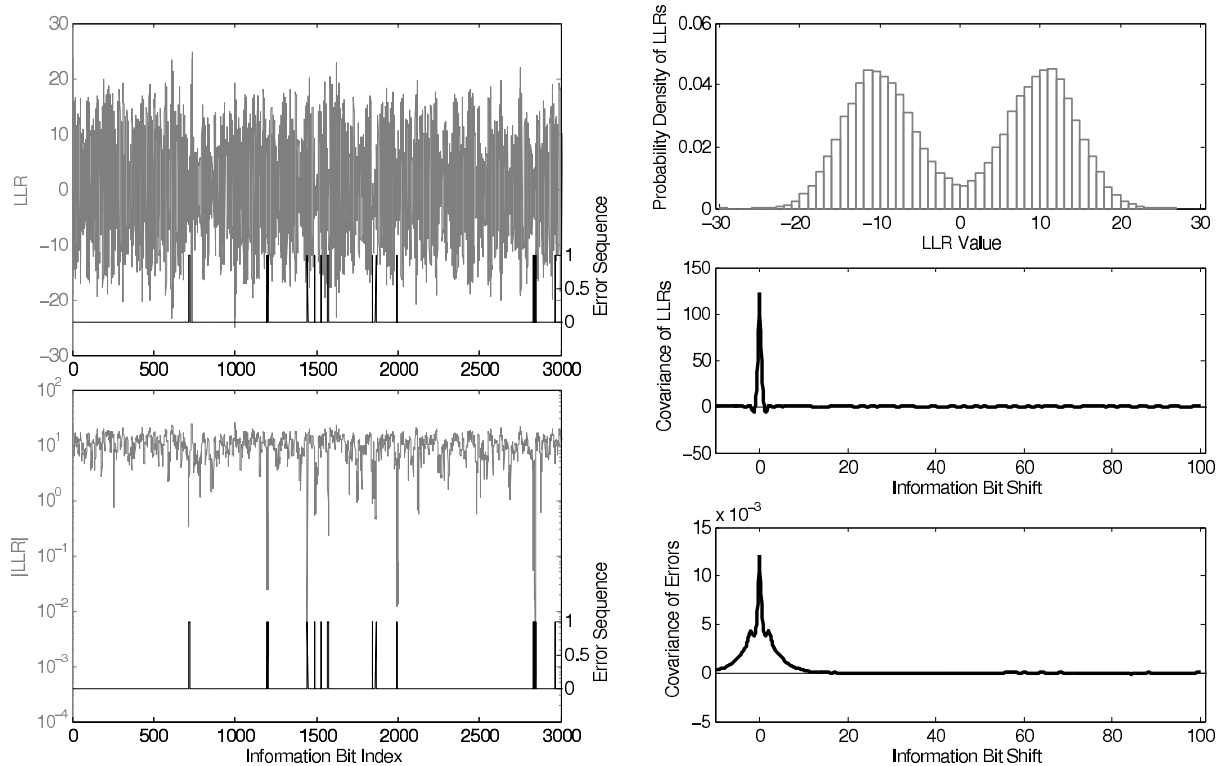
Figure 2.25: BCJR decoding of a normal convlolutional code at SNR = 3 dB (BER = 1.41×10^{-2})
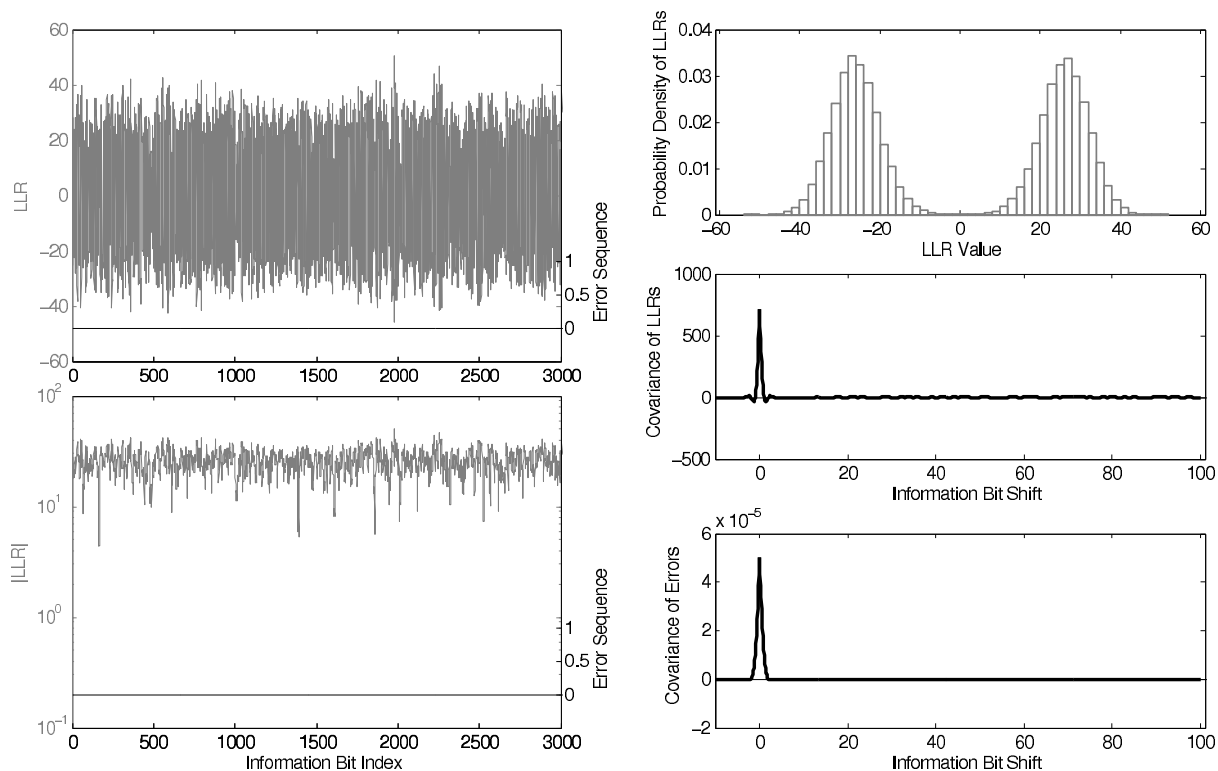
Figure 2.26: BCJR decoding of a normal convlolutional code at SNR = 6 dB (BER = 1.13×10^{-4})

In the next set of figures we will see the results when a catastrophic code is used and thus will be able to see differences from the normal non-catastrophic codes. The following Figure 2.27, Figure 2.28 and Figure 2.29 show the results of catastrophic convolutional codes at SNRs 1,3 and 6, respectively.
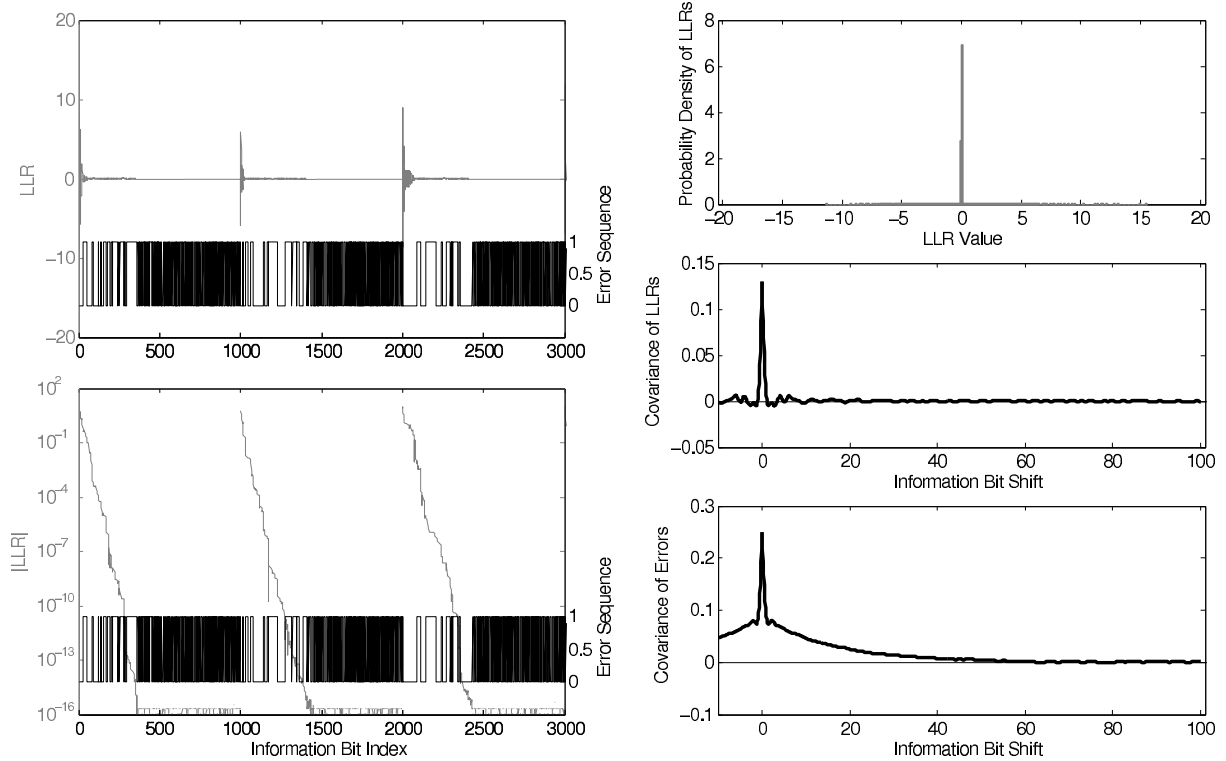
Figure 2.27: BCJR decoding of a catastrophic convlolutional code at SNR = 1 dB (BER = 4.94×10^{-1} and PER = 1)

These new figures differ greatly from the previous ones. The following observations can be made:

- Absolute value of LLRs does not behave at all like before. It starts at a point and keeps its value for a while, then decreases suddenly to another value and keeps that value for a while and so on, it keeps decreasing and actually resembles in a way a staircase that keeps going down. Of course, the beginning of a new packet resets the starting $|LLR|$ to a relatively high value.

- The time when a certain $|LLR|$ value is kept constant increases as SNR increases, which results in decreasing the number of the distinct sudden drops present in the $|LLR|$ throughout a single packet.

- Catastrophic errors are clearly seen, when a catastrophic error happens, the error vector stays at 1 for a while before another catastrophic error takes it back to 0.

- Catastrophic errors seems to happen only when there is a drop in $|LLR|$. On the other hand, a drop in $|LLR|$ does not always mean that a catastrophic error happens, it only points to the possibility that a catastrophic error may be present.

44

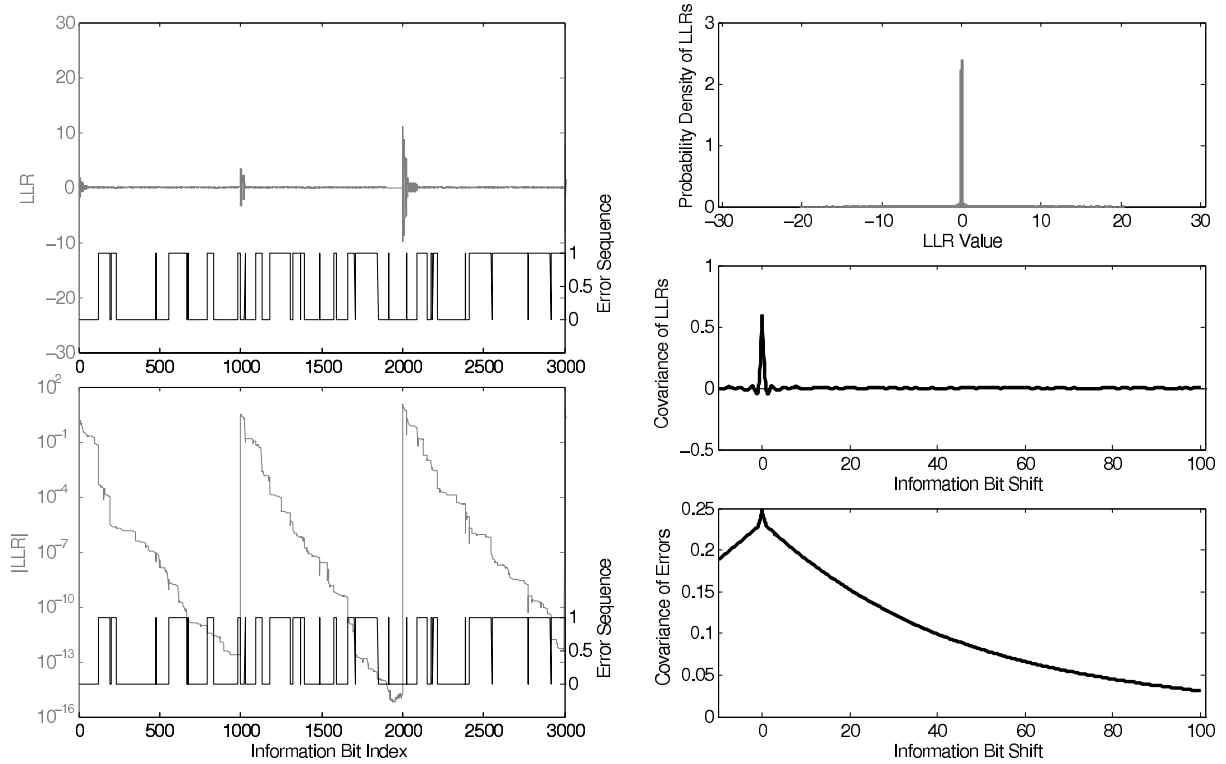Figure 2.28: BCJR decoding of a catastrophic convlolutional code at SNR = 3 dB (BER = 4.74×10^{-1} and PER = 1)
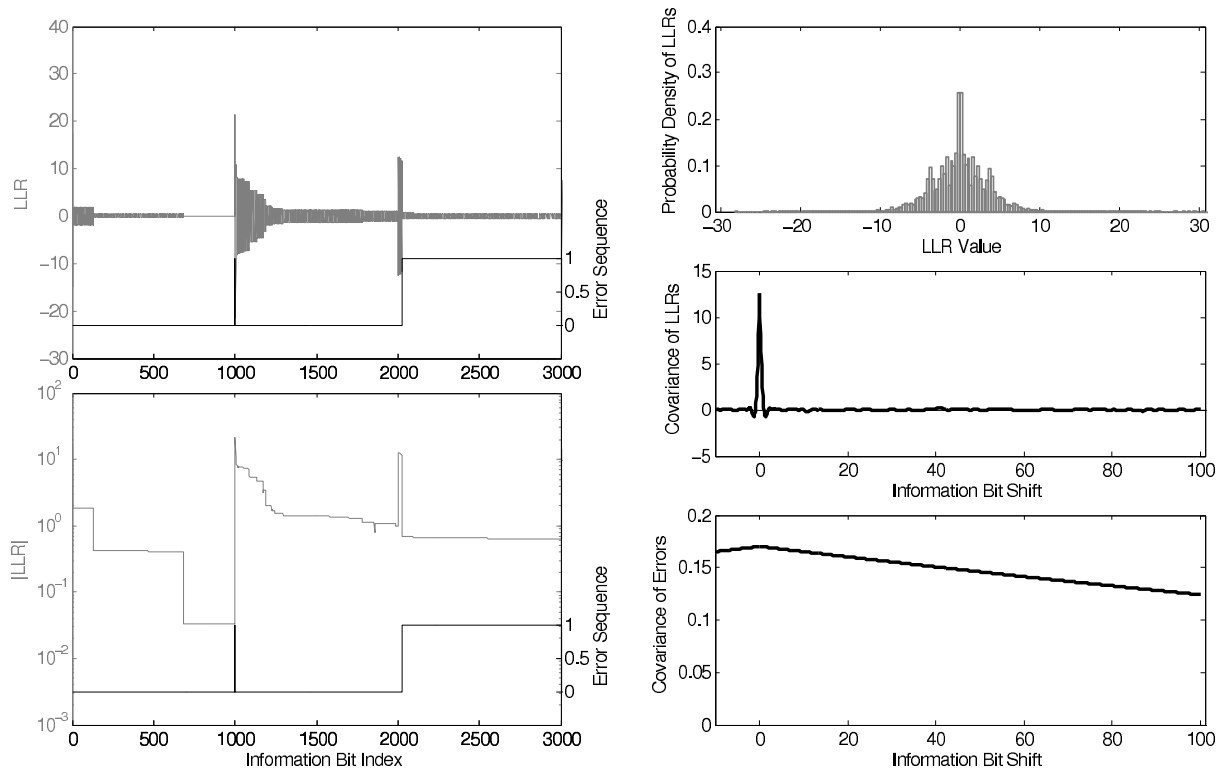

Figure 2.29: BCJR decoding of a catastrophic convlolutional code at SNR = 6 dB (BER = 2.07×10^{-1} and PER = 4.89×10^{-1})

- The probability density function of LLRs has lost its Gaussian-like property and now it is all concentrated around zero.

- Errors are highly correlated, and correlation increases with SNR. Errors are highly correlated because catastrophic codes decoding tend to have many long periods of constant errors (i.e. error vector $= 1$). As SNR increases, the number of catastrophic errors decreases. This increases the period between successive catastrophic events, and thus increases the duration of those long periods resulting in higher correlation to adjacent time periods.

From the results we got, it can be said that catastrophic codes does not produce independent errors, and thus cannot be considered perfectly secret, even when the BER is high. However, correcting catastrophic errors in the low SNR region does not seem to be a possibility even when some relation is found between the possible positions of the errors and the absolute values of $|LLR|$s. In the next section, we will present a simple method to see if any improvement in decoding catastrophic codes are possible by exploiting this relation.

2.6 Trial to Improve Catastrophic Decoding through Watching LLRs

The relationship between catastrophic errors and $|LLR|$s is interesting in the sense that we might be able to detect a possible catastrophic event while decoding and thus correct it. In this section, we propose a simple method to detect possible catastrophic events and try to correct them hoping to yield better decoding of catastrophic codes.

In order to decode catastrophic codes correctly, we should find a way to eliminate catastrophic errors. Catastrophic errors might be eliminated if the code needs at least 2 symbols to produce the error (as the code we used $G = [5, 3, 6]$). Since at least 2 symbols are needed, then 1 symbol in error will not be able to produce a catastrophic error. Our method can be summarized in the following steps:

1. Decode the whole sequence first and get the soft $|LLR|$ values of the whole packet.

2. Decide the position of the bits where $|LLR|$ drops happen. This can be done using a simple criterion. For example, a position of an $|LLR|$ drop can be decided if 2 or 3 symbols are keeping their values at $|LLR_1| \pm \delta$, where δ is some small number, and the next 2 or 3 symbols have a value of $|LLR_2| \pm \delta$, where $\frac{|LLR_2|}{|LLR_1|} < \Delta$ for some $0 < \Delta < 1$.

3. Extract the received symbols at those bit positions, and replace them with a (0,0) symbol (i.e. remove a possibly defected symbol and replace it with a received value that gives same probabilities to all possible symbols). In this case, we are hoping that this symbol represents one of the two symbols that are needed to cause a catastrophic error and that this symbol is defected. By removing this symbol and replacing it with a symbol that gives the same probability to all received symbols,

we are removing the effect of the that symbol and hoping that such a scenario will not drifting the decoder into a possible catastrophic error.

4. Decode again the new adjusted received sequence and compare the results.

Some simulations were done to test this method using different values of Δ. The results were almost identical to regular curves except for large Δ when the method produces worse BERs. Figure 2.30 shows the results. The fact that results did not get worse by
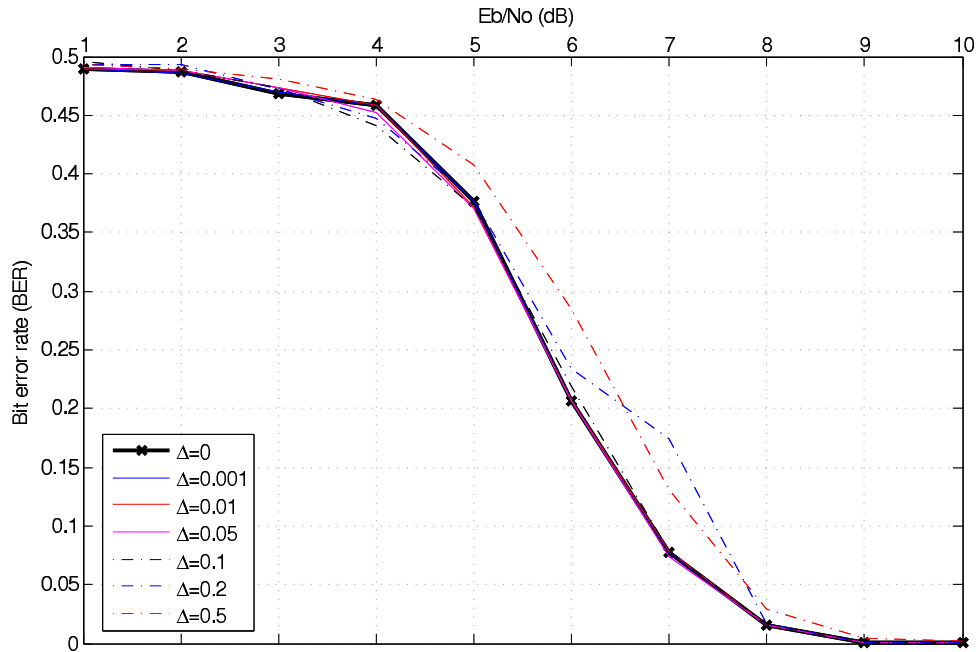

Figure 2.30: Simulations of the correcting method using different values of Δ

this method is in itself a sign that a more advanced study of catastrophic codes with $|LLR|$s may lead to a new way in decoding catastrophic codes. Notice for example the slight improvement done at SNR = 4 dB, especially for $\Delta = 0.1$.

Chapter 3

Serial Concatenated Convolutional Codes (SCCC)

Concatenated codes are known for their sharp transition in bit error probability termed as the waterfall effect [38]. This transition is very sharp that the BER drops at once from high values to very low BERs in a span of 1dB or less. This sharp transition is what we desire for our objective. The only thing that needs adjustment is the position of this drop, for it always happens at low SNRs for the rates of interest, which is not what we want for a more secure setting. In this chapter, we will study Serially Concatenated Convolutional Codes (SCCC) and see how their drop can be shifted to higher SNRs without loosing the original sharpness. Another problem we will also investigate is the use of catastrophic codes in SCCC. The questions we pose are: will catastrophic codes shift the low SNR threshold of SCCC to higher values, or will it destroy the sharpness of SCCC?

3.1 Concatenated Codes

3.1.1 Introduction

In the search for codes that approach Shannon's capacity limit, turbo codes [39] were introduced in 1993. These codes succeeded in achieving a random-like code similar to what was originally envisioned by Shannon [40]. Because of this, these codes were able to achieve exceptionally good performance, that generally for any code rate and information block length larger than 10^4, turbo codes with iterative decoding can achieve BERs as low as 10^{-5} within 1 dB from the Shannon's limit. Turbo codes are a special case of the more general concatenated codes. Concatenated codes are comprised basically of at least two simple codes (e.g. convolutional codes) arranged in concatenations (parallel or serial) along with a pseudo-random interleaver (π). A simple turbo code is a parallel concatenated code having a block diagram as shown in Figure 3.1. The information sequence enters the first encoder as is (the usual case of any encoder), while the same information sequence gets permuted and then enters the second encoder. The outputs of the two coders are combined to form the output of the turbo code. In serial concatenation [31] however, as shown in Figure 3.2, the information sequence enters the first encoder (the outer encoder), then the output of this first encoder gets permuted through the random interleaver, and then enters into the second encoder (inner encoder). The output
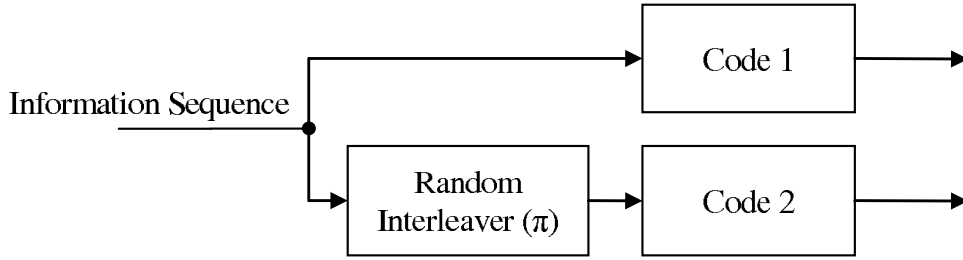
Figure 3.1: Parallel concatenated encoder block diagram

of the concatenated code thus becomes the same output of the second inner encoder. A k/n SCCC can be represented as $k/N/n$ where k is the number of information bits encoded at a time, N is the number of output bits of the outer encoder and also the input of inner encoder, and n is the number of output bits from the inner encoder.
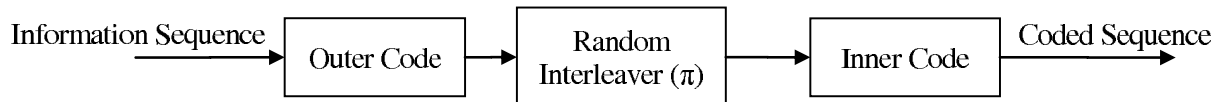
Figure 3.2: Serial concatenated encoder block diagram

3.1.2 Overview of Decoding Process

Decoding of concatenated codes involves an iterative process using soft-input-soft-output (SISO) decoders. A demodulator first calculates the soft output of each coded bit received, these values are then fed to the SISO decoders which work together to best estimate the original information sequence. A SISO decoder is a four port device as shown in Figure 3.3 that takes as input the soft information of coded bits sequence $\mathbf{P}(\mathbf{c}; I)$ and uncoded
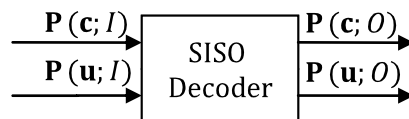
Figure 3.3: SISO Module

bits sequences $\mathbf{P}(\mathbf{u}; I)$ of a specific code. From the two sequences, it produces a better soft output estimate to both of them: $\mathbf{P}(\mathbf{c}; O)$ and $\mathbf{P}(\mathbf{u}; O)$. The soft data can be represented as the posteriori probability of each bit being zero or one. A details description of the algorithm involved will be presented afterwords.

In parallel decoding as shown in Figure 3.4, there are two SISO modules or decoders corresponding to each code 1 and 2. In each iteration, the first decoder will have two inputs the demodulator output as $\mathbf{P}_1(\mathbf{c}; I)$, and the π^{-1} of the soft uncoded output bits of decoder 2 in the previous iteration as $\mathbf{P}_1(\mathbf{u}; I)$ (In the first iteration, uniform density is used, i.e. $\mathbf{P}_1(\mathbf{u}; I_i) = 0.5 \ \forall i \in I$). The outputs of this decoder are $\mathbf{P}_1(\mathbf{c}; O)$ which is
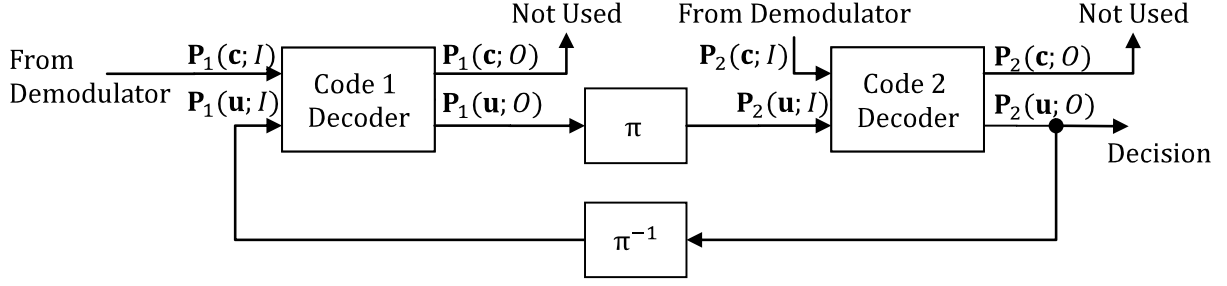
Figure 3.4: Parallel concatenated decoder block diagram

not used and $\mathbf{P}_1(\mathbf{u};O)$ which gets permuted before getting to the second decoder making $\mathbf{P}_2(\mathbf{u};I) = \pi[\mathbf{P}_1(\mathbf{u};O)]$. The second decoder also uses the demodulator output as $\mathbf{P}_2(\mathbf{c};I)$, and then produces its results as $\mathbf{P}_2(\mathbf{c};O)$ which is not used, and the final data $\mathbf{P}_2(\mathbf{u};O)$ which is either used for the final decision making or passed to $\mathbf{P}_1(\mathbf{u};I) = \pi^{-1}[\mathbf{P}_2(\mathbf{u};O)]$ for another iteration.

The serially concatenated decoding is done is a similar but different fashion. The block diagram of an SCCC decoder is shown in Figure 3.5. In SCCC, the coded demodulator
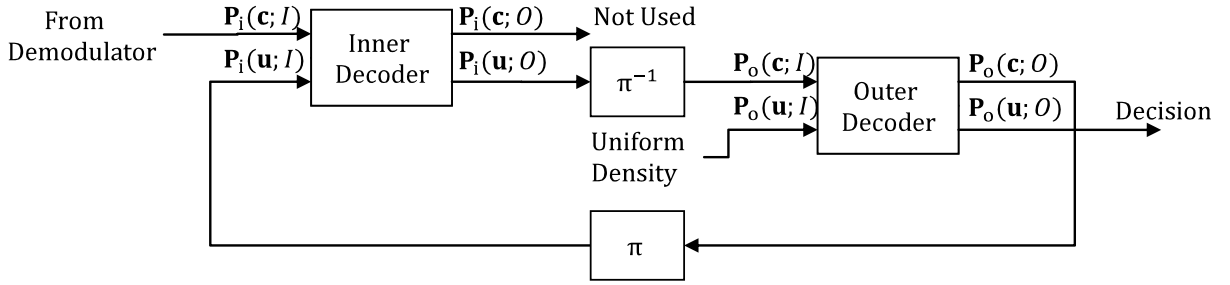
Figure 3.5: Serial concatenated decoder block diagram

data is taken only by the inner decoder (as $\mathbf{P}_i(\mathbf{c};I)$), which takes also the permuted outer decoder coded bits output $\mathbf{P}_o(\mathbf{c};O)$ of the previous iteration. The inner decoder produces two sequence of outputs $\mathbf{P}_i(\mathbf{c};O)$, which is not used, and $\mathbf{P}_i(\mathbf{u};O)$, which is inverse permuted and used as input to the outer decoder coded sequence $\mathbf{P}_o(\mathbf{c};I) = \pi^{-1}[\mathbf{P}_i(\mathbf{u};O)]$. The outer decoder takes another uniform density input which represents that the original message is uniformly distributed between ones and zeros. Unlike turbo codes (PCCC), both of the outer decoder's outputs are used. $\mathbf{P}_o(\mathbf{c};O)$ is permuted and used as the uncoded input for the inner decoder next iteration $\mathbf{P}_i(\mathbf{u};I) = \pi[\mathbf{P}_o(\mathbf{c};O)]$ if needed, and $\mathbf{P}_o(\mathbf{u};O)$ is used for the final decision making.

3.1.3 Detailed View of Different SCCC Modules

In this section, we will present a more detailed view of the several parts involved in concatenated coding. Particularly, the inteleaver used within the encoding and decoding processes, the SISO APP module, and the soft-bit demodulation required.

● **Interleaving**

The random like code of concatenated codes is much influenced by the random interleaver. Pseudo-random interleaving patterns can be generated in many ways for use in concatenated codes [34]. We used a simple method to generate the random interleaving function. The method is known by Fisher-Yates [41] or Knuth shuffle, and is summarized in the following steps:

For an interleaver with K bits:

1. In a K-bit array A, store the numbers 1 though K in order ($A_i = i, \forall i \in \{1, 2, \ldots, K\}$).

2. let $k = 1$

3. Pick a random number r between 1 and K inclusive.

4. Swap A_r with A_k

5. increment k by one

6. repeat from step 3 until $k = K$

The resulting array A is taken to be the interleaving function (π). The inverse interleaving function (π^{-1}) is easily constructed from A. Let B be a K-bit array representing π^{-1}, then $B_{A_i} = i, \forall i \in \{1, 2, \ldots, K\}$.

● **SISO APP Module**

Another very important part is the SISO decoder. The SISO decoder is based in part on the BCJR algorithm described in section 2.3.1. We changed the algorithm used from log-based BCJR algorithm described above to another, even better algorithm, described in [42]. This algorithm takes as input the probabilities of coded and uncoded bits of a specific code and outputs an enhanced *a posteriori* probabilities describing both of them. One difference between the BCJR algorithm used before and this SISO algorithm is that this one uses the normal-scale multiplication procedure rather than the log-scale summation described earlier. The use of this algorithm was beneficial in our scenario because of one numerical problem faced during simulation. In the log algorithm, the summation terms of the trellises tend to go to very high numbers reaching infinity as iterations proceed, while for the algorithm adopted, the use of the product rather than summation helps in solving this numerical simulation. A description of the algorithm follows below.

In order to describe the SISO algorithm, we need to define some quantities that will be used in the description. In a convolutional k/n code, the sequences of input symbols are represented by $\mathbf{U} = (U_l)_{l \in \mathbb{L}}$ which is defined over a time index set $\mathbb{L}$ and drawn from an alphabet set $\mathbb{U} = \{u_1, \ldots, u_{2^k}\}$. Each input symbol U_l consists of k bits $U_l^j, j = 1, 2, \ldots, k$ with realization $w^j \in \{0, 1\}$. To the sequence of input symbols, we associate the sequence of *a priori* probability distributions $\mathbf{P}(\mathbf{u}; I) = (P_l(u; I))_{l \in \mathbb{L}}$, where $P_l(u; I) = \prod_{j=1}^k P_l(w^j; I)$. The output coded sequences is similarly defined as $\mathbf{C} = (C_l)_{l \in \mathbb{L}}$ drawn from the alphabet $\mathbb{C} = \{c_1, \ldots, c_{2^n}\}$. Each output symbol C_l consists of n bits $C_l^j, j = 1, 2, \ldots, n$ with realization $c^j \in \{0, 1\}$. To the sequence of output symbols, we

associate the sequence of *a priori* probability distributions $\mathbf{P}(\mathbf{c};I) = (P_l(c;I))_{l \in \mathbb{L}}$, where $P_l(c;I) = \prod_{j=1}^{n} P_l(c^j;I)$. The assumption that *a priori* input distributions of symbols can be represented as the product of marginal distribution of bits is valid since we use a random bit-interleaver.

As in the description of BCJR algorithm before, at each time instant l, there is a trellis section consisting of transitions ("edges") between the states (from state s^S to state s^E). The set of trellis states is represented by $\mathbb{S}$ with realizations $s \in \mathbb{S}$. The transitions between these states can be represented by $e \in \mathbb{E}$, where $\mathbb{E}$ is the total number of trellis transitions present in each trellis section. Each transition (e) is associated with a starting and ending states $s^S(e)$ and $s^E(e)$, as well as an input uncoded symbol $u(e)$ and an output coded symbol $c(e)$, see Figure 3.6. Each transition e can be uniquely distinguished using
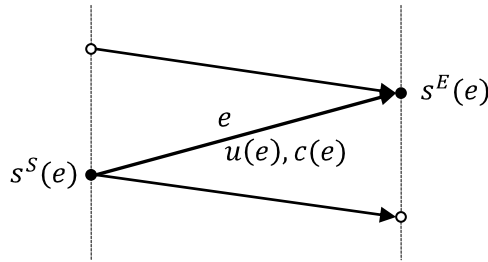
Figure 3.6: An edge of the trellis section

its starting state and uncoded input symbol: $\left(s^S(e), u(e)\right)$.

The SISO module as stated earlier is a 4 port device with input sequences of probability distributions $\mathbf{P}(\mathbf{c};I)$ and $\mathbf{P}(\mathbf{u};I)$, and outputs the sequences $\mathbf{P}(\mathbf{c};O)$ and $\mathbf{P}(\mathbf{u};O)$ based on the inputs and on its knowledge of the trellis section or code. For a finite time index set $\mathbb{L} = \{1, \ldots, L\}$, the SISO algorithm can be explained in two steps: The first step is to compute the output probability distributions $\tilde{P}_l(c^j;O)$ and $\tilde{P}_l(u^j;O)$ for the jth bit within each symbol at time l, where

$$\tilde{P}_l(c^j;O) = \tilde{H}_{c^j} \sum_{e:C_l^j(e)=c^j} A_{l-1}[s^S(e)]P_l[u(e);I]P_l[c(e);I]B_l[s^E(e)] \tag{3.1}$$

$$\tilde{P}_l(u^j;O) = \tilde{H}_{u^j} \sum_{e:U_l^j(e)=u^j} A_{l-1}[s^S(e)]P_l[u(e);I]P_l[c(e);I]B_l[s^E(e)] \tag{3.2}$$

and $A_l(.)$ for $l = 1, \ldots, L$ and $B_l(.)$ for $l = 0, \ldots, L-1$ are obtained through the forward and backward recursions as:

$$A_l(s) = \sum_{e:s^E(e)=s} A_{l-1}[s^S(e)]P_l[u(e);I]P_l[c(e);I] \tag{3.3}$$

$$B_l(s) = \sum_{e:s^S(e)=s} B_{l+1}[s^E(e)]P_{l+1}[u(e);I]P_{l+1}[c(e);I] \tag{3.4}$$

The forward recursions compute the forward path metric for each state by taking all the edges that lead to that state $\left(e : s^E(e) = s\right)$, update the previous metrics $A_{l-1}[s^S(e)]$, by

multiplying them with the *a priori* probabilities of the corresponding uncoded $P_l[u(e); I]$ and coded $P_l[c(e); I]$ symbols at time l, and summing them up all together. While the backward recursions compute the backward path metric for each state by taking all the edges that start from that state $e : s^S(e) = s$, update the metrics present at the corresponding end states $B_{l+1}[s^E(e)]$, by multiplying them with the *a priori* probabilities of the corresponding uncoded $P_{l+1}[u(e); I]$ and coded $P_{l+1}[c(e); I]$ symbols at time $l + 1$, and summing them up all together. This method is similar to the BCJR algorithm, with the exception that the uncoded symbols are also used in computing the metrics, rather than using the coded symbols alone as in BCJR.

The initial values for the recursions are $A_0(s) = 1$ if $s = S_0$ and $A_0(s) = 0$ otherwise, and $B_L(s) = 1$ if $s = s_L$ and $B_L(s) = 0$ otherwise only if s_L is known, or $B_L(s) = A_L(s)$, $\forall s$ if s_L in not known. The quantities $\tilde{H}_{c^j}$, $\tilde{H}_{u^j}$ are normalization constants such that $\sum_{c^j} \tilde{P}_l(c^j; O) = 1$ and $\sum_{u^j} \tilde{P}_l(u^j; O) = 1$ respectively.

In the second step, from equations (3.1) and (3.2), it is apparent that $P_l(c^j(e); I)$ in the first equation and $P_l(u^j(e); I)$ in the second do not depend on e by definition of the summation indexes, and thus can be extracted from the summations. Thus defining the new quantities $P_l(c^j; O) \triangleq H_{c^j} \frac{\tilde{P}_l(c^j; O)}{P_l(c^j; I)}$ and $P_l(u^j; O) \triangleq H_{u^j} \frac{\tilde{P}_l(u^j; O)}{P_l(u^j; I)}$, where H_{c^j}, H_{u^j} are normalization constants such that $\sum_{c^j} P_l(c^j; O) = 1$ and $\sum_{u^j} P_l(u^j; O) = 1$. It can be easily verified that $P_l(c^j; O)$ and $P_l(u^j; O)$ can be obtained through the expressions:

$$P_l(c^j; O) = H_{c^j} \tilde{H}_{c^j} \sum_{e:C_l^j(e)=c^j} A_{l-1}[s^S(e)]P_l[u(e); I] \left(\prod_{\substack{i=1 \\ i \neq j}}^{n} P_l[c^i(e); I] \right) B_l[s^E(e)] \quad (3.5)$$

$$P_l(u^j; O) = H_{u^j} \tilde{H}_{u^j} \sum_{e:U_l^j(e)=u^j} A_{l-1}[s^S(e)] \left(\prod_{\substack{i=1 \\ i \neq j}}^{k} P_l[u^i(e); I] \right) P_l[c(e); I]B_l[s^E(e)] \quad (3.6)$$

The probability distributions $P_l(c^j; O)$, $P_l(u^j; O)$ are computed based on the code constraints and obtained using the probability distributions of all bits of the sequence except the distributions $P_l(c^j; I)$, $P_l(u^j; I)$ of the jth bit within the lth symbol, respectively. In the literature of concatenated coding $P_l(c^j; O)$, $P_l(u^j; O)$ would be called *extrinsic bit information*. They represent the "added value" of the SISO module to the *a priori* distributions $P_l(c^j; I)$, $P_l(u^j; I)$. These $P_l(c^j; O)$, $P_l(u^j; O)$ are the quantities that are used as inputs for the other SISO modules in an iterative decoding environment.

● Soft Bit Demodulation

It is seen in Figures 3.4 and 3.5 that one important input to the SISO module ($\mathbf{P}(\mathbf{c}; I)$) is coming from the demodulator. As described above, the SISO module accepts as input the probability distributions of each coming bit, for both coded and uncoded sequences. In this situation, the received sequence of coded symbols should be demodulated in a way that produces probability distributions of each coded bit rather than relying only on the distances between the received symbols and any constellation points, as in Viterbi or BCJR described earlier. The following is a description of the demodulation algorithm used to provide the soft bit information as probability distributions of each based on [43].

Let $C \triangleq \{c_0, c_1, \ldots, c_{N-1}\}$ be the coded message to be transmitted over a channel. And let the modulation scheme to be used consists of M ($M = 2^m$) symbols (such as MQAM or MQSP). Every m bits from C can be considered as an m-dimensional vector $U|_{u_0}^{u_{N/m-1}}, u_l = \{u_{l,0}, u_{l,1}, ..., u_{l,m-1}\}$ which will be mapped to a constellation point represented by two real valued symbols (A_l, B_l) at time l. In an AWGN channel, the received signal can be written as (X_l, Y_l)

$$\begin{aligned} X_l &= A_l + I_l \\ Y_l &= B_l + J_l \end{aligned} \tag{3.7}$$

where I_l, J_l are two independent Gaussian noises with zero mean and variance σ_N^2. For each bit $u_{l,i}$, the constellation symbols are divided into two parts. Let $C_1(i)$ be the set of symbol points (X_n, Y_n) with their corresponding i^{th} bit being 1, and $C_0(i)$ the set of points with the corresponding bit being zero. Then, the bit-level probability distributions can be calculated using Bayes rule as

$$P\{u_{l,i} = 1 | X_l, Y_l\} = \frac{\tilde{P}\{X_l, Y_l | u_{l,i} = 1\}}{\tilde{P}\{X_l, Y_l | u_{l,i} = 0\} + \tilde{P}\{X_l, Y_l | u_{l,i} = 1\}} \tag{3.8}$$

$$P\{u_{l,i} = 0 | X_l, Y_l\} = \frac{\tilde{P}\{X_l, Y_l | u_{l,i} = 0\}}{\tilde{P}\{X_l, Y_l | u_{l,i} = 0\} + \tilde{P}\{X_l, Y_l | u_{l,i} = 1\}} \tag{3.9}$$

since $P\{u_{l,i} = 0\} = P\{u_{l,i} = 1\} = \frac{1}{2}$, where

$$\begin{aligned} \tilde{P}\{X_l, Y_l | u_{l,i} = 1\} &= \sum_{(X_n, Y_n) \in C_1(i)} P\{X_l = X_n + I_l, Y_l = Y_n + J_l\} \\ &= \frac{1}{\sqrt{2\pi}\sigma_N} \sum_{(X_n, Y_n) \in C_1(i)} e^{-\frac{(X_l - X_n)^2 + (Y_l - Y_n)^2}{2\sigma_N^2}} \end{aligned} \tag{3.10}$$

$$\begin{aligned} \tilde{P}\{X_l, Y_l | u_{l,i} = 0\} &= \sum_{(X_n, Y_n) \in C_0(i)} P\{X_l = X_n + I_l, Y_l = Y_n + J_l\} \\ &= \frac{1}{\sqrt{2\pi}\sigma_N} \sum_{(X_n, Y_n) \in C_0(i)} e^{-\frac{(X_l - X_n)^2 + (Y_l - Y_n)^2}{2\sigma_N^2}} \end{aligned} \tag{3.11}$$

Using soft bit demodulation, Gray coding serves as the optimal choice of lattice labeling, since here we are interested in the bits themselves rather than the symbols as is the case we used before in Viterbi and BCJR.

3.2 Simulation

As stated earlier, we used SCCC codes in our testing, The following flowchart diagram in Figure 3.7 represents a simplified version of the actual algorithm used in simulation. As in convolutional codes, we also used 1000 packets of various lengths (1000 and 10000 bits/packet)to run the simulations for each value of E_b/N_0. All packets are initialized from the all-zero state and then the data is randomly generated, encoded using the SCCC encoder described earlier and transmitted over an AWGN channel. The data received is decoded using the iterative decoding algorithm described in section 3.1.2 with 10 iterations (unless specified otherwise).

```
                                          ┌─────────────┐
                                          │    Start    │
                                          └──────┬──────┘
                                                 │
        No                              ┌──────────────────────┐
    ┌─────────────────────────────────→│  For Eb/No = 1 to 15  │
    │                                   └──────────┬───────────┘
    │                                              │
 ╱Last Eb/No?╲                          ┌──────────────────────┐
 ╲           ╱ ←──────────────────────  │  For Packet = 1 to 1000│
    │ Yes                               └──────────┬───────────┘
    │                                              │
    │         ┌──────────────┐          ┌──────────────────────┐
    │         │   Display    │          │ Generate random info │
    │         │ the results  │          │ sequence (ex. 1000   │
    │         └──────────────┘          │ bits) starting from  │
    │                                   │ the all-zero state   │
    │           Yes                     └──────────┬───────────┘
    │        ╱Last packet?╲    No                  │
    │        ╲            ╱ ──────────  ┌──────────────────────┐
    │                                   │ Outer encode,        │
 ┌──────┐                               │ interleave, inner    │
 │ End  │                               │ encode and map to    │
 └──────┘                               │ transmission symbols │
                                        └──────────┬───────────┘
```
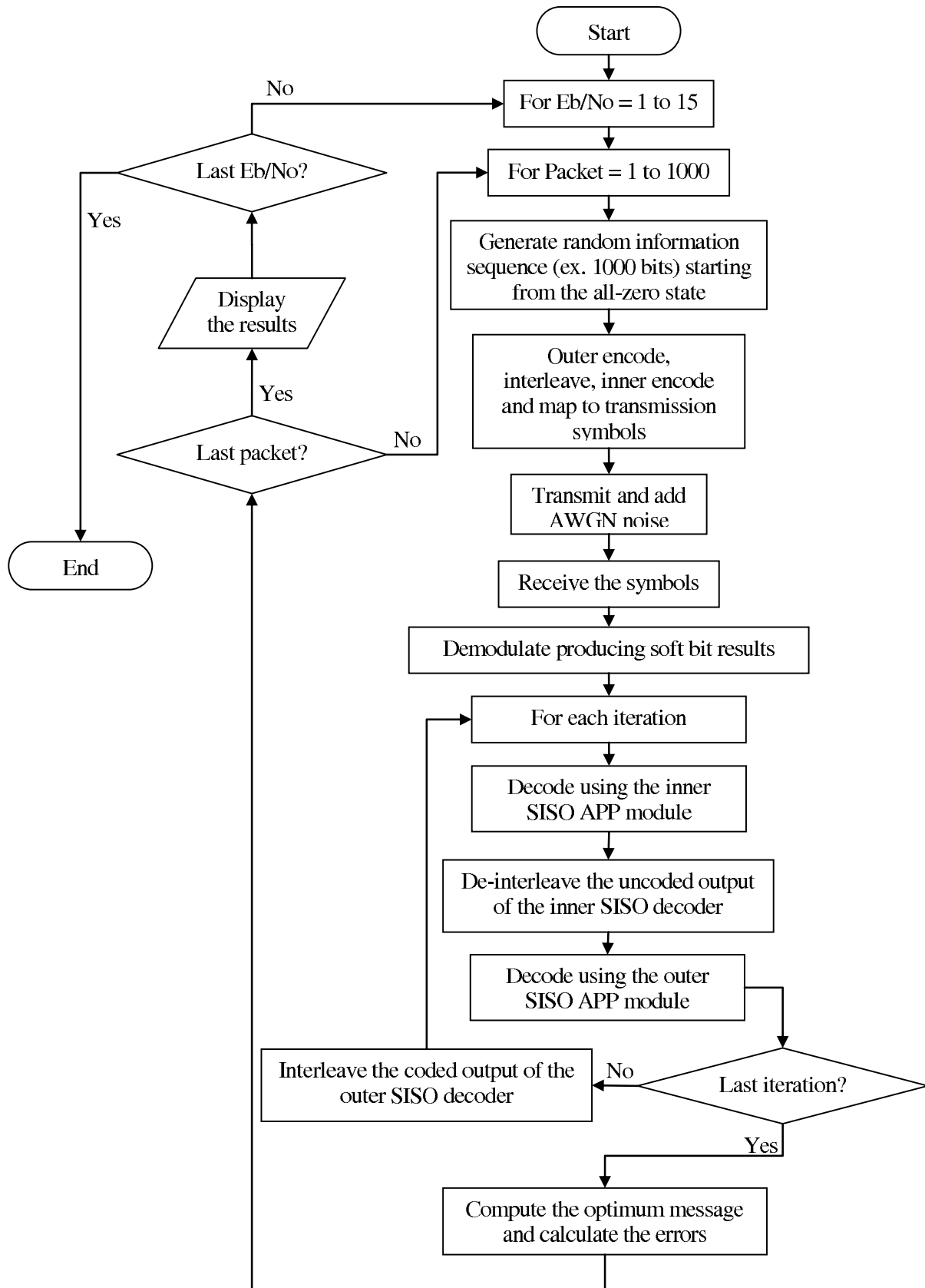
Figure 3.7: A simplified flowchart of the SCCC coding algorithm used in simulations

3.3 Testing Our SCCC Algorithm with MATLAB's Demo

After finishing the program we have used for SCCC. An important step was to check that our results are consistent with performance evaluation done in literature. To make sure we achieve the desired performance, we looked at MATLAB, and found that there is a demonstration of SCCC coding simulation, very similar to our scenario. The following Figures 3.8 and 3.9 show how close our results are to MATLAB's.

Figure 3.8: SCCC comparison between MATLAB demo simulation and our C++ program using an outer code of rate 1/2, G=[1,2], and inner code of rate 2/3, G=[17,6,15]. The results are after 6 iterations

3.4 Using Catastrophic Inner Codes in SCCC

In chapter 2, we have seen that regular catastrophic codes have an interesting error vector, which contains several long burst-like error periods. Burst errors are usually counteracted by using interleavers that separate the errors allowing for better correction. As seen earlier in this chapter, interleaving plays an important role in concatenated codes, so we were motivated to study SCCC having catastrophic inner codes with some regular outer codes. We were hoping that the positioning of a catastrophic code inside an SCCC will increase the sharpness of the BER curve threshold found in catastrophic codes, and move it to some lower SNRs that are in between the high SNR threshold of catastrophic codes and the low SNR threshold of SCCC.

We used the inner code as the catastrophic code along with a regular outer code. The inner code we used is the one whose trellis is shown in Figure 2.8; It is a 2/3 rate code
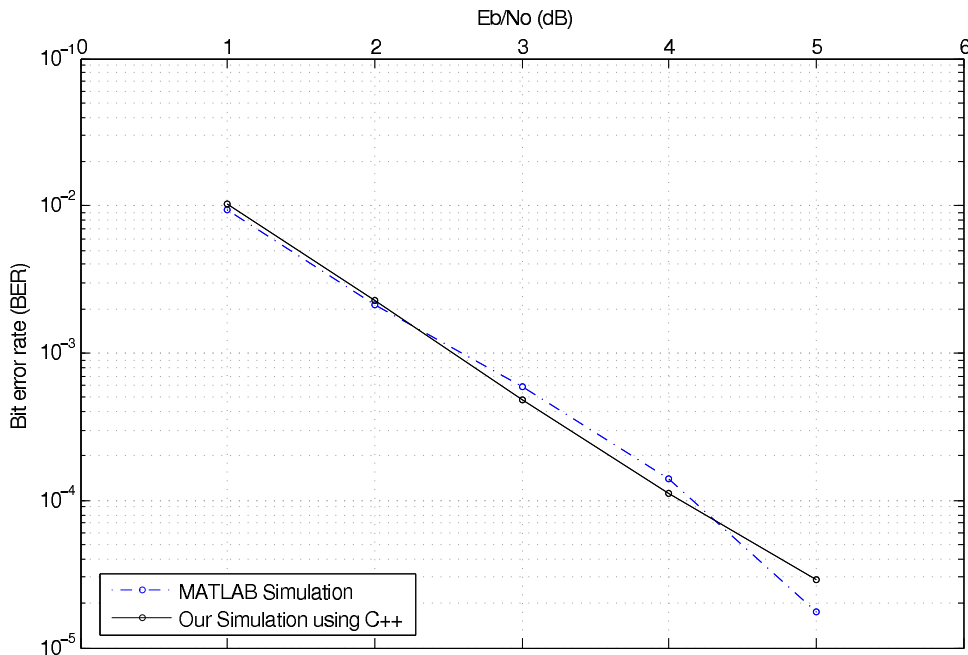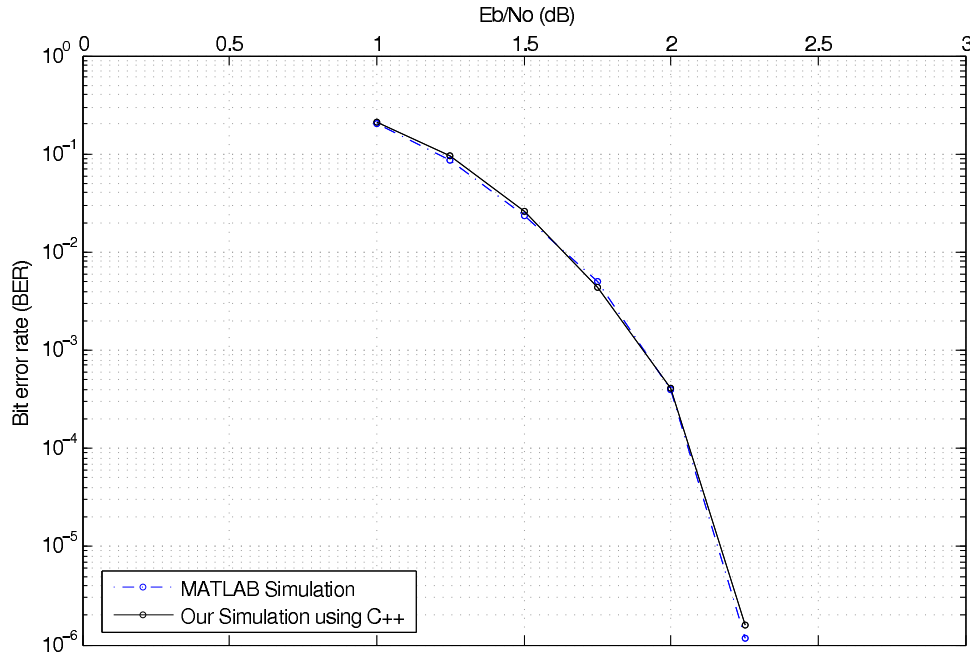
Figure 3.9: SCCC comparison between MATLAB demo simulation and our C++ program using an outer code of rate 1/2, G=[7,5], and inner code of rate 2/3, G=[17,6,15]. The results are after 6 iterations

with G=[5,12,17]. Different rate 1/2 outer codes were used in order to better understand the influence of the outer code on the performance. The outer codes used are shown in Table 3.1. Figure 3.10 shows the results when using packets of 1000-bit lengths. These

Table 3.1: Outer codes used

Code number	Generating Matrix
Code1	[3,2]
Code2	[1,3]
Code3	[7,5]
Code4	[1,2]

results are compared to the performance of the same inner catastrophic code when used alone in a convolutional code and Viterbi decoding in Figure 3.11. The same comparison is done for 10000-bit packets in Figure 3.12

It can be seen from the figures that contrary to what we were hoping to see, SCCC with catastrophic codes did not shift the threshold to lower SNRs, and it did not produce better slopes beyond the threshold, as compared to regular catastrophic codes especially for the 1000-bit per packet case. Increasing the packet length increases the slope in SCCC similar to what happens in catastrophic convolutional codes, but the slope is still less than that of regular catastrophic codes.

In [44], catastrophic codes were studied in turbo codes. One important thing was emphasized that iterative decoding is not effective when catastrophic codes are used due

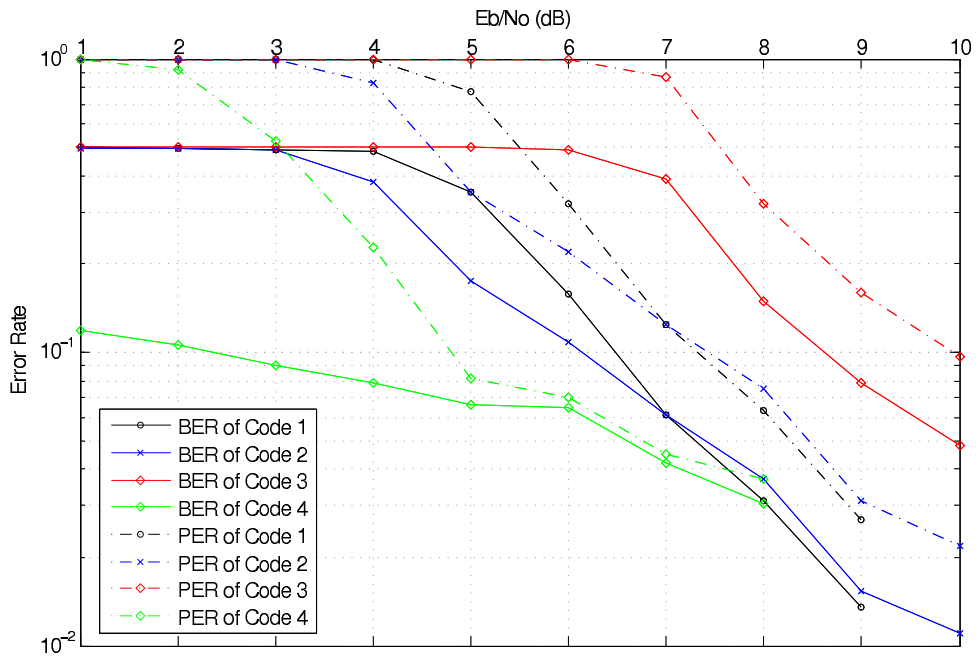Figure 3.10: Comparison between different outer codes used with the same inner catastrophic code, using 1000 packets of 1000-bit length
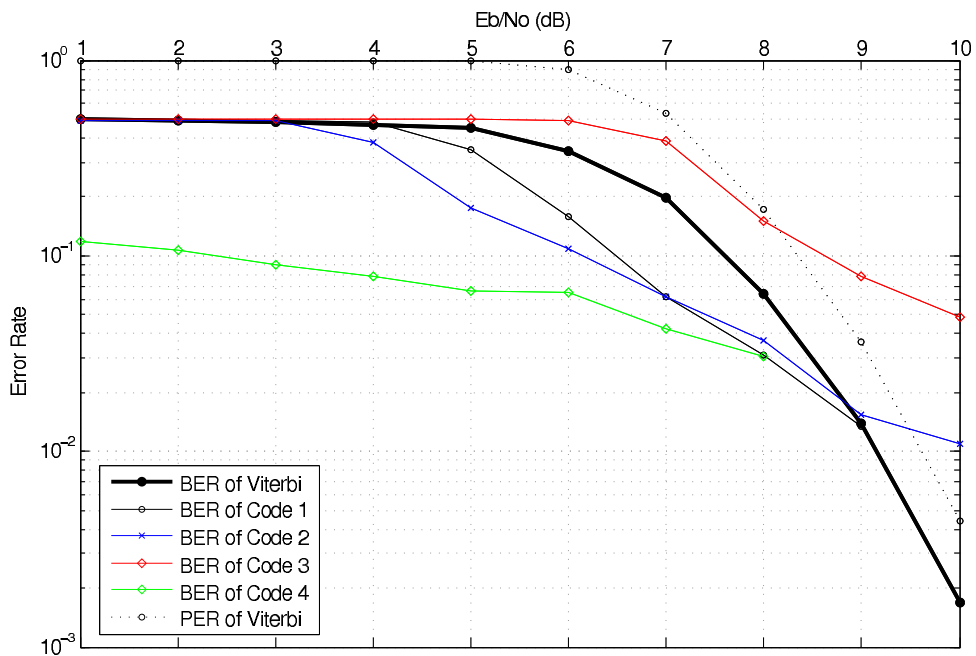

Figure 3.11: Comparison between catastrophic codes in SCCC and convolutional codes, both using 1000 packets of 1000-bit length
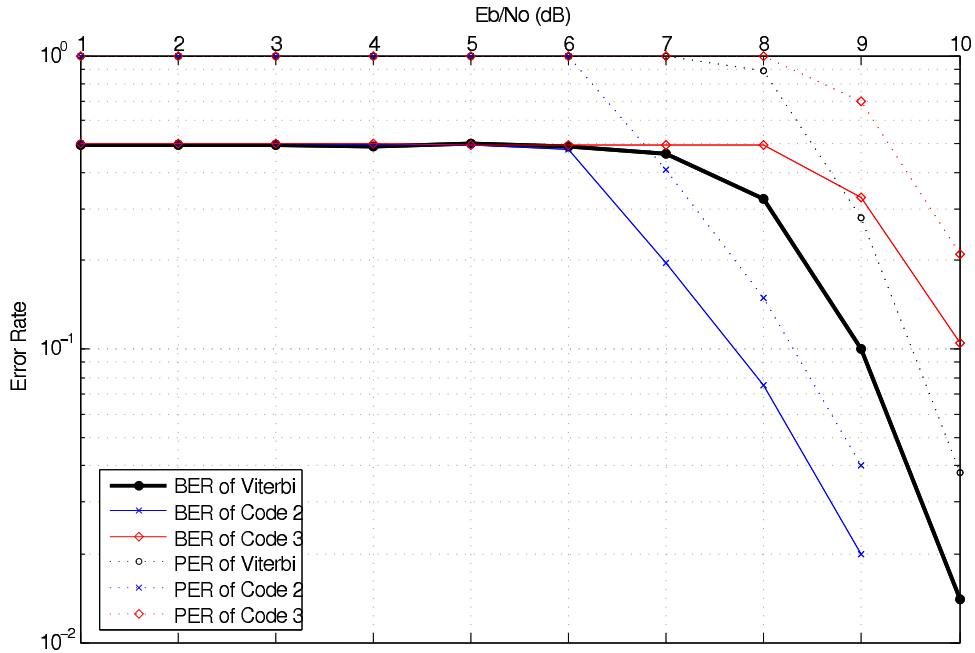
Figure 3.12: Comparison between catastrophic codes in SCCC and convolutional codes, both using 1000 packets of 10000-bit length

to the poor convergence behavior of the constituent decoders. Iterative decoding may succeed if something can be done to start the convergence process in the initial iteration. The author in [44] suggested the use of doping (i.e replace some nonsystematic bits with systematic ones) which takes the code back to the normal behavior of turbo codes, which again is not what we are looking for. However, from Figure 3.12, we can find that selecting appropriate outer codes can help us in maintaining the BER waterfall and shifting around the threshold.

3.5 Controlling Non-Catastrophic SCCC

After the undesirable results we got using catastrophic codes in inner encoders of SCCC for rates of interest, we shift attention to the use of regular SCCC. Since regular SCCC are known to have a sharp BER transition in the low SNR region, we investigated some ways to shift this threshold without destroying the sharpness to some higher SNRs, the following sections describe the different methods used. All the methods were done on the 1/2/3 SCCC code with outer 1/2 rate code with G=[7,5] and inner 2/3 rate code with G=[17,6,15]. A packet length of 1000 is used and decoding is done for 10 iterations.

3.5.1 Introducing Burst Errors

Catastrophic codes resemble normal convolutional codes but with many large bursts of errors. Of course these bursts are the sole cause of the bad performance of catastrophic codes as it is seen earlier. This led us to investigate introducing one intentional burst error of m-bits in some random position within each transmitted packet. The introduction of this burst error is done after encoding; When the whole packet is encoded, a random

number r is picked between 1 and $K - m$, where K is the total number of bits present in a packet, then the bits from r to $r + m$ are flipped and then transmitted. The results of this method are shown in Figure 3.13. As seen in the figure, the introduction of burst
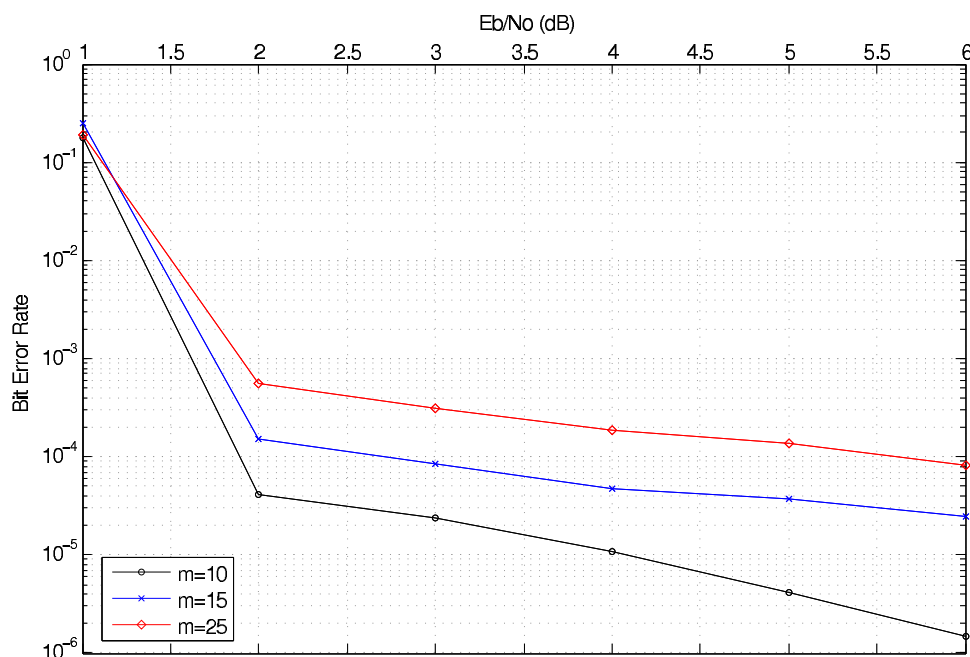
Figure 3.13: Results of introducing random burst errors of different number of bits m

errors did not shift the SNR threshold but rather increased the BER of the error floor for SNRs after the threshold. This of course makes this method useless in our case.

3.5.2 Random BSC-like Errors

Another method tried is the introduction of random BSC-like errors rather than one long burst error. In this way we are hoping to shift the BER curve by increasing the noise. The introduction of this noise is done by passing the coded message through a BSC channel shown in Figure 3.14 with a specific p before the actual transmission. Figure
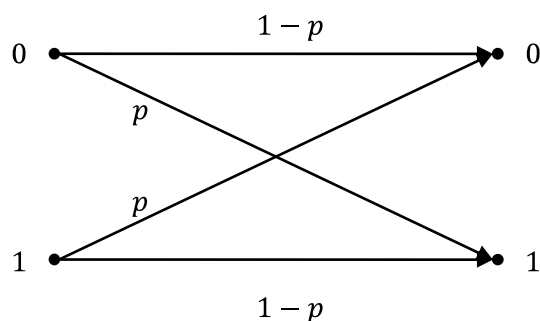
Figure 3.14: The BSC channel

3.15 show the results of this method using different p values. From the figure, it can be seen that introducing BSC-like errors does in fact shift the BER curve to higher SNRs
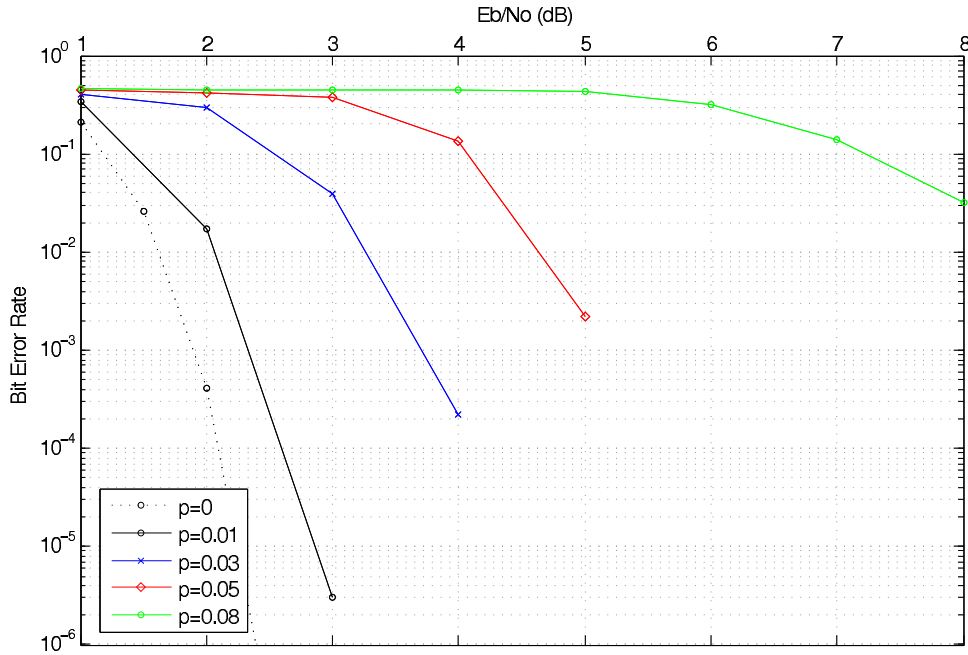
Figure 3.15: Results of introducing random BSC-like errors using different p values

at the expense of flattening the slope as p increases, thus this method cannot be used if shifting the threshold to mid SNRs as the slope will be almost flat, losing the desired sharp threshold feature.

3.5.3 Puncturing

Puncturing is a method usually used to increase the transmission rate without actually changing the code. It is done by removing some pre-specified bits from the coded message before transmission. For a k/n code, puncturing is done by taking successive symbols (say p symbols with n bits each), the rate of which is $\frac{k \times p}{n \times p}$, and removing z bits of the coded message before being transmitted at an equivalent rate of $\frac{k \times p}{n \times p - z}$. The z bits that are removed from each group of p symbols, are specified using a puncturing matrix P which consists of p rows and n columns, each element in P is either 1, for those bits that are transmitted, or 0, for those bits that are removed before transmission. Thus making z equals the number of zeros present in P.

For easiness of representation, we will write P in one line consisting of $p \times n$ bits rather than a $p \times n$ matrix. We will separate each n bits by a semicolon.

Example on puncturing: Suppose we have a 1/3 rate code, and we want to transmitted at a rate of 3/7. We will use a puncturing matrix P with $p = 3$ and $z = 2$ as for example the one shown below:

$$
P = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = [101; 111; 101]
$$

Now suppose that a part of the coded message to be transmitted is the sequence of bits

(100 011 010 111 110 000). This sequence is multiplied in a bitwise operation with P before transmission to yield the actual message to be transmitted.

$$
\begin{aligned}
\text{Coded message} &= \ldots 100\ 011\ 010\ \ 111\ 110\ 000 \ldots \\
P &= \ldots [101; 111; 101][101; 111; 101] \ldots \\
\text{Transmitted message} &= \ldots 10\ 011\ 00\ \ 11\ 110\ 00 \ldots
\end{aligned}
$$

In order to see what puncturing can achieve, we simulated the code using 4 different puncturing matrices changing the 1/3 rate to several new rates as shown in Table 3.2. Some examples of punctured codes are given In [45]. The simulation results are shown

Table 3.2: Puncturing matrices used and the corresponding new rate

New Rate	Puncturing matrix used
2/5	[111;101]
3/8	[111;101;111]
3/7	[101;111;101]
5/11	[101;111;001;111;101]

in Figure 3.16 It is seen that increasing the rate through puncturing does shift the BER
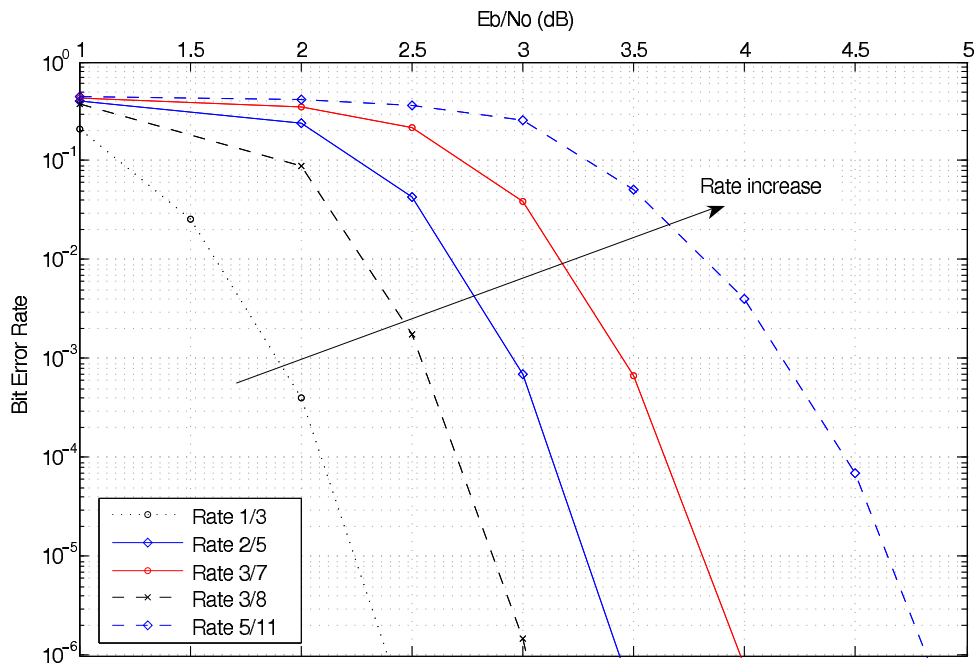
Figure 3.16: Increasing the rate through puncturing and its effects on SCCC

curve to the right. This shift, however, affects the sharpness of the threshold making it less distinct than before. It can be concluded that puncturing is a possible method to be used in shifting the BER curve, it achieves better slopes and threshold sharpness than BSC approach, but still can not be used for shift the BER to high SNRs.

62

3.5.4 Modulation

Our last approach is using different modulations. Up till now, all the simulation in SCCC were done using a QPSK modulation. Using higher order modulations results in less power per transmitted bit, since the average power used to transmit each symbol is kept the same and number of bits per symbol is increased. This reduction of bit power is expected to shift the curve without much affecting the slope or sharpness of the original QPSK curve. We simulated 3 other modulations; 8-PSK, 16-PSK and 16-QAM. As
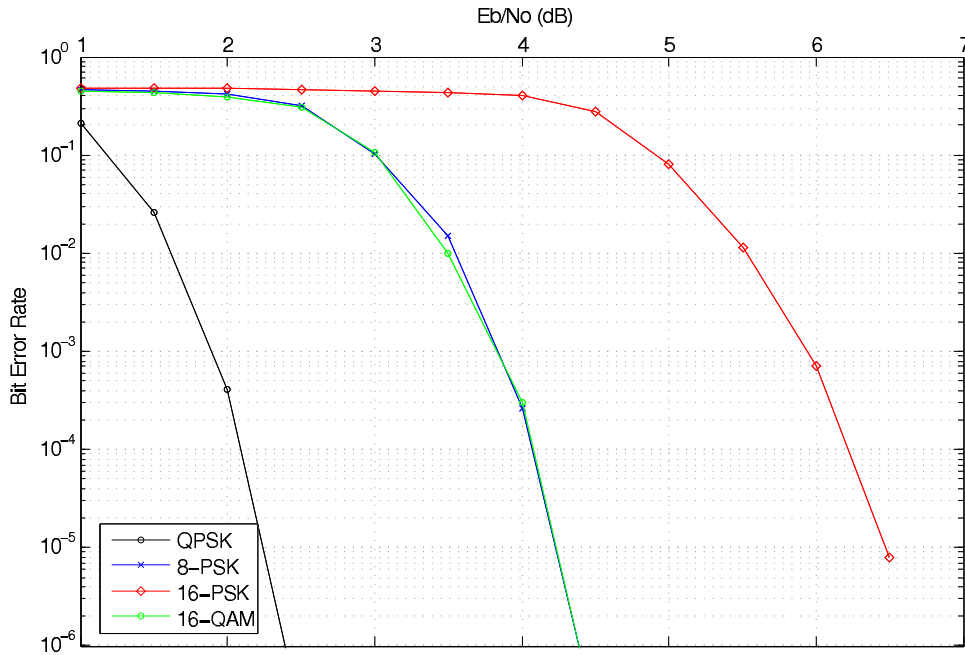
Figure 3.17: Using different modulation schemes on SCCC

shown in Figure 3.17, the results were very much what is expected, the curves shifted to the right with relatively small variations in sharpness and slope. This makes it along with puncturing a possible approach to shift the BER curve to higher SNR thresholds.

As a conclusion, we can see that using non-catastrophic SCCC modified using modulation schemes and/or puncturing, is a very promising approach that can be taken to provide physical layer security in our scenario especially when Bob is working on a low or mild SNR. An additional remark is that both modulation and puncturing vary the equivalent rates and hence change the BER curve, unlike regular catastrophic codes, where the rate is kept constant and only labeling or packet lengths are used in control.

3.6 Second Order Statistics of Information Bit LLRs and Errors

As in the case for BCJR codes, we also investigated the LLRs in the low SNR region around the waterfall: their probability density function and correlation, as well as the correlation of the error sequence. We studied both the normal SCCC and the SCCC with

a catastrophic inner code to see the similarities and differences. The normal SCCC code used is the same one used in the previous section. We investigate different modulations to see the effects since this approach has been shown to have the best way to shift the BER curve.

We start with the normal SCCC modulated using QPSK. The results are shown for SNRs = 1, 1.5, 2 and 2.5 in Figures 3.18, 3.19, 3.20 and 3.21. As shown in the figures,
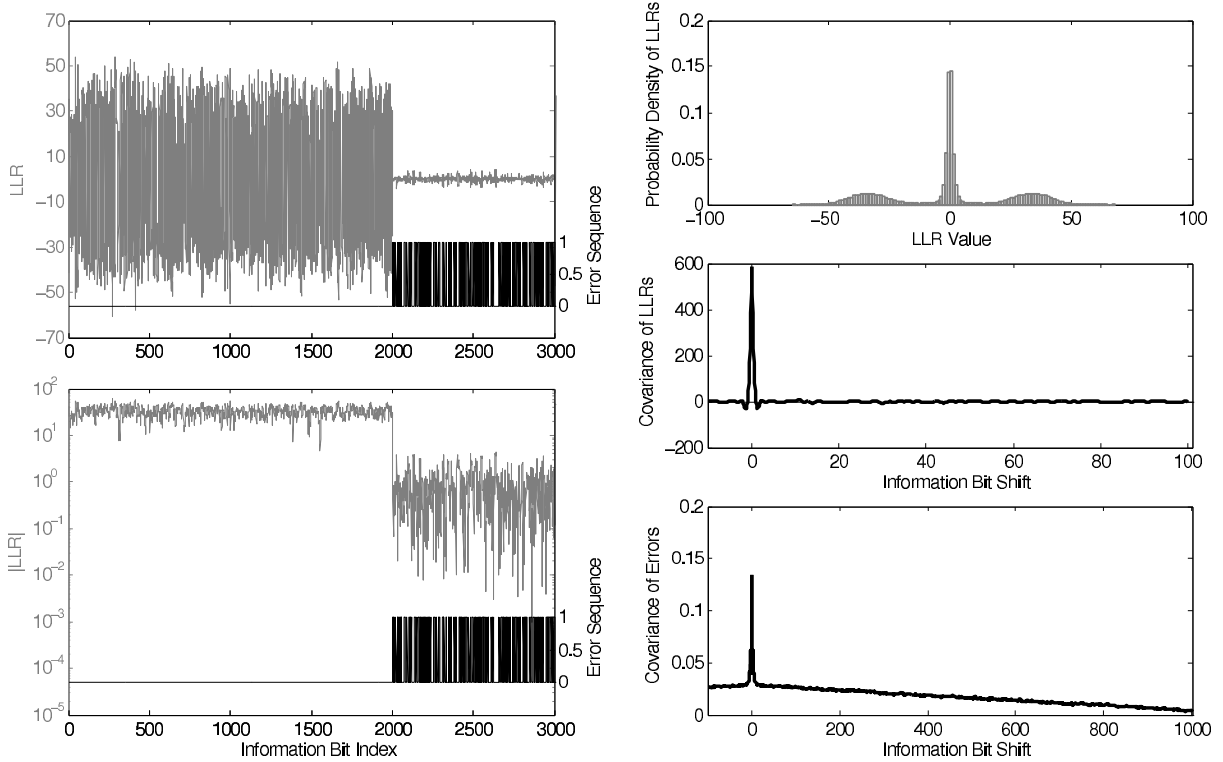
Figure 3.18: SCCC decoding of a code transmitted using QPSK modulation at SNR = 1 dB (BER = 2.05×10^{-1})

the following observations are made:

- The LLRs are not correlated as in the case in convolutional coding.

- Error correlation decreases as SNR increases as in the case of normal non-catastrophic convolutional codes.

- The most interesting observation is the presence of 3 peaks at low SNRs. The 2 side ones are Gaussian-like, while the middle is just around zero. The Gaussian-like figures is an expected result especially when there are many analytical papers that use Gaussian approximations for the study of turbo codes such as [46] and [47]. But the presence of the middle lobe is the interesting part. According to Lee and Blahut in [48],[49] and [50], the Gaussian approximations are useful for the infinite length turbo coding analysis, while on the other hand, finite length codes as in our case have different characteristics than the infinite length codes. One of the main differences is this peak around zero, which is present at low SNRs and disappears

64

Figure 3.19: SCCC decoding of a code transmitted using QPSK modulation at SNR = 1.5 dB (BER = $2.59{\times}10^{-2}$)

Figure 3.20: SCCC decoding of a code transmitted using QPSK modulation at SNR = 2 dB (BER = $4.04{\times}10^{-4}$)
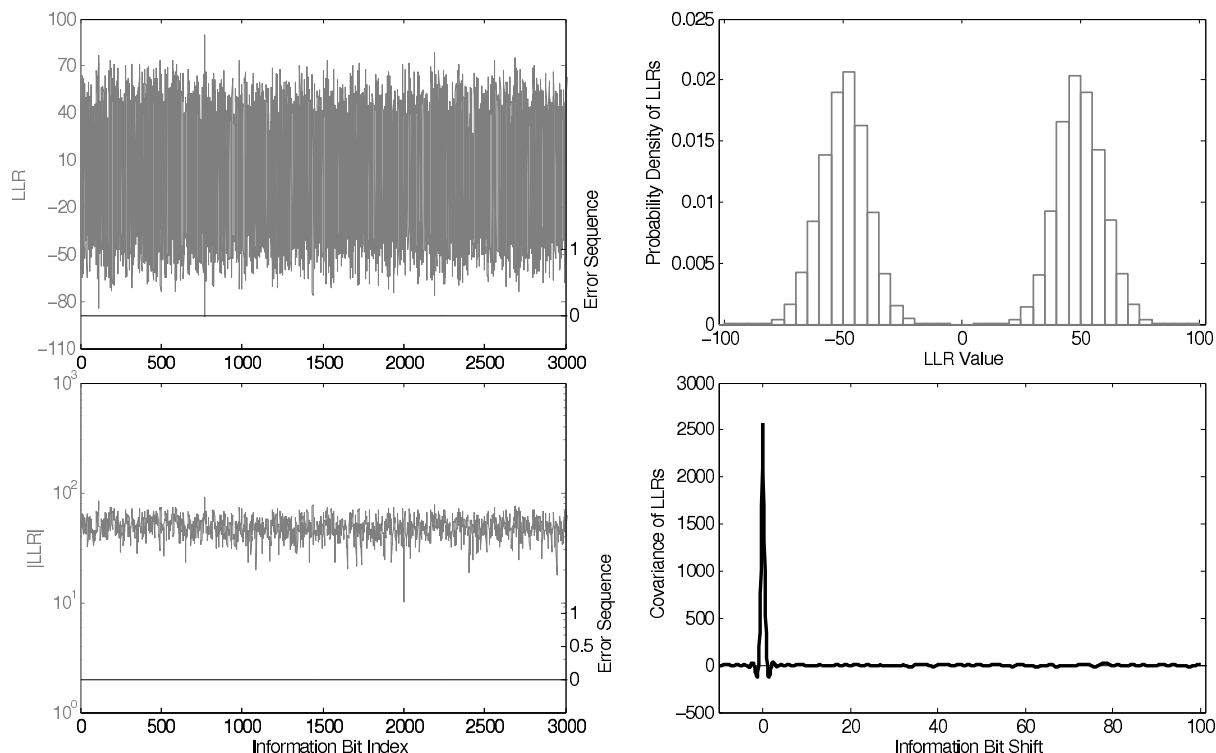
Figure 3.21: SCCC decoding of a code transmitted using QPSK modulation at SNR = 2.5 dB (BER = 2×10^{-7})

gradually as SNR increases beyond the waterfall region, in what seems to be a transition from the middle lobe to the side lobes.

- From the LLRs time domain figures, it is observed that the $|LLR|$ is either at some high value for an *entire* packet (present in one of the side lobes) or at a low value for the *entire* packet (middle lobe around zero). Whenever the $|LLR|$ is at the high value, the bit errors are rare to happen, while on the contrary, if the $|LLR|$ in a packet is at a low value, then that packet has BER almost equal to 0.5.

- As proved recently in [51], optimization of PER or BER are two contradicting requests to any coding scheme transmitting at a rate above the capacity $R > C$ (i.e. at low SNRs). It also proved that codes optimized for PER have the property of transmitting packets either correctly or in such a way that no information is transmitted at all corresponding to a burst error with a probability of 0.5. The SCCC scheme is a PER optimized code, and thus the observations shown above completely agree with [51]. This further implies that we cannot approximate the channel between the information bits and decoded bits as a memoryless channel. As a contrast, in chapter 2 convolutional codes demonstrated low correlation in bit errors.

These observations are seen in the same exact way for higher modulations, but of course of higher SNR values as seen in Figures 3.22 and 3.23 using 8-PSK modulations at SNRs 3 and 4; and Figures 3.24 and 3.25 using 16-PSK modulation at SNRs 5 and 6.

66
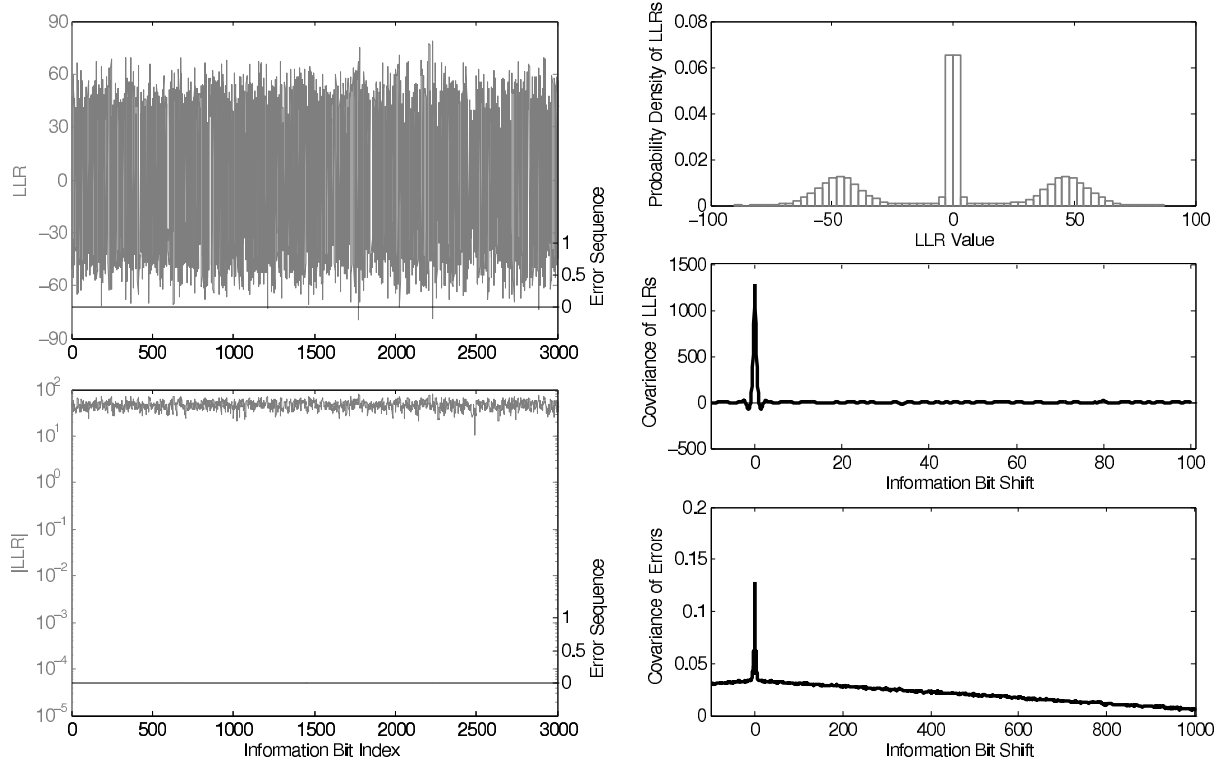
Figure 3.22: SCCC decoding of a code transmitted using 8-PSK modulation at SNR = 3 dB (BER = 1.02×10^{-1})
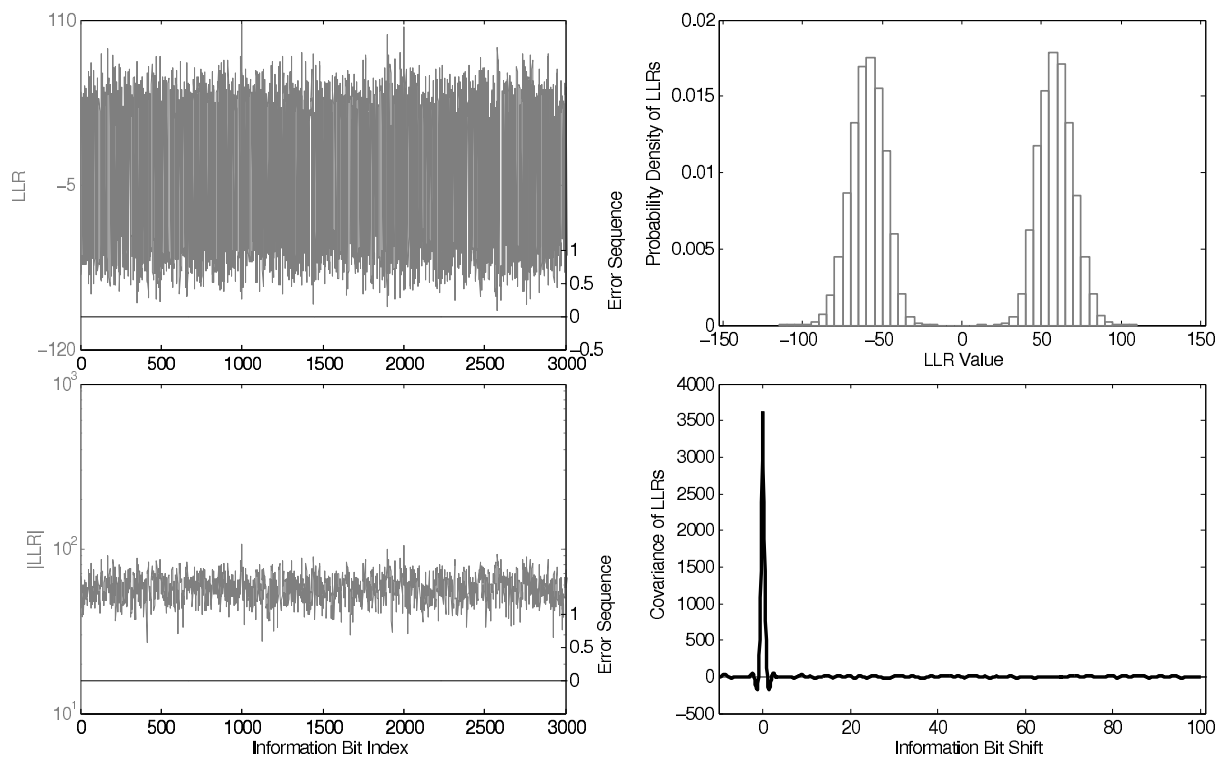
Figure 3.23: SCCC decoding of a code transmitted using 8-PSK modulation at SNR = 4 dB (BER = 2.6×10^{-4})
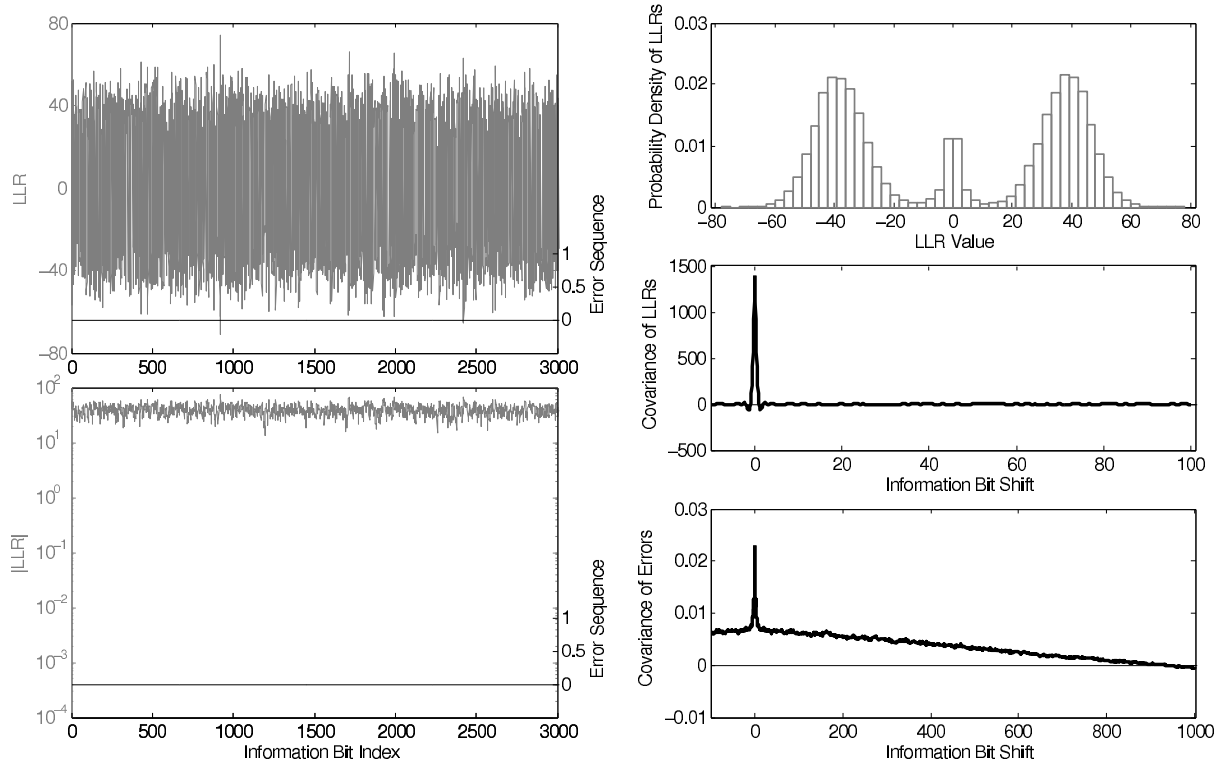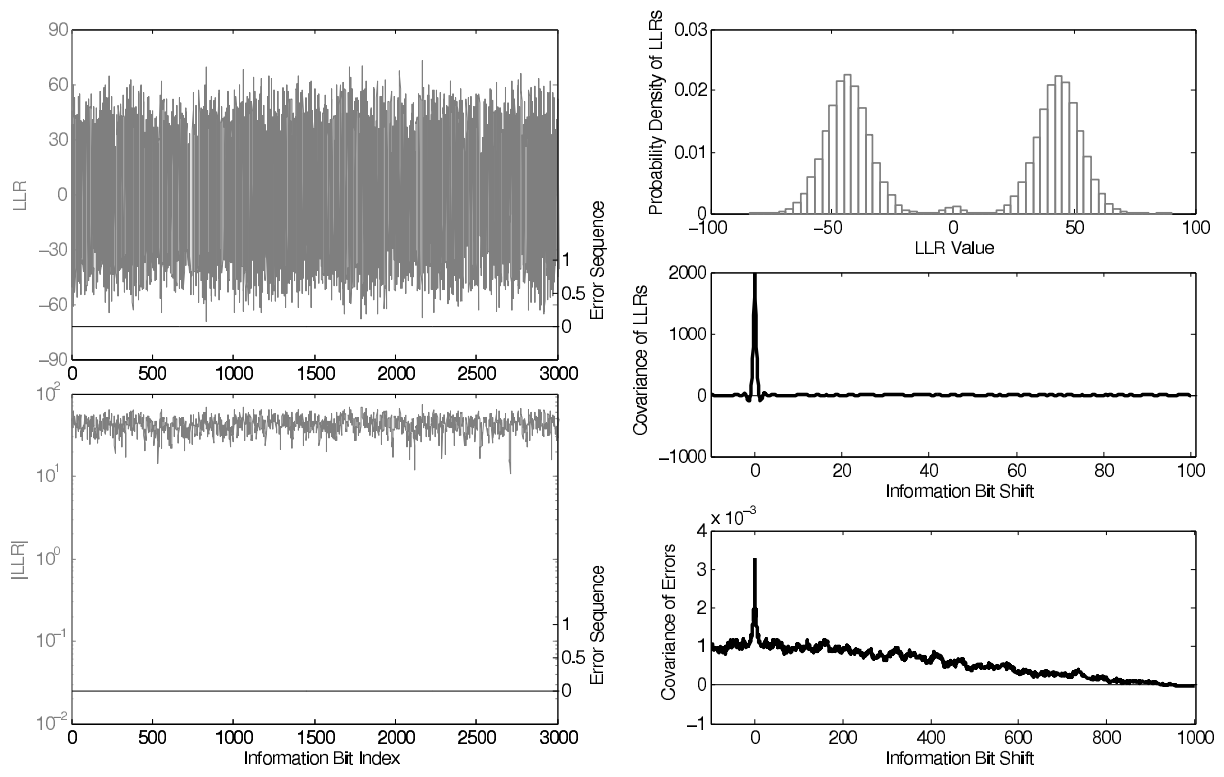
67
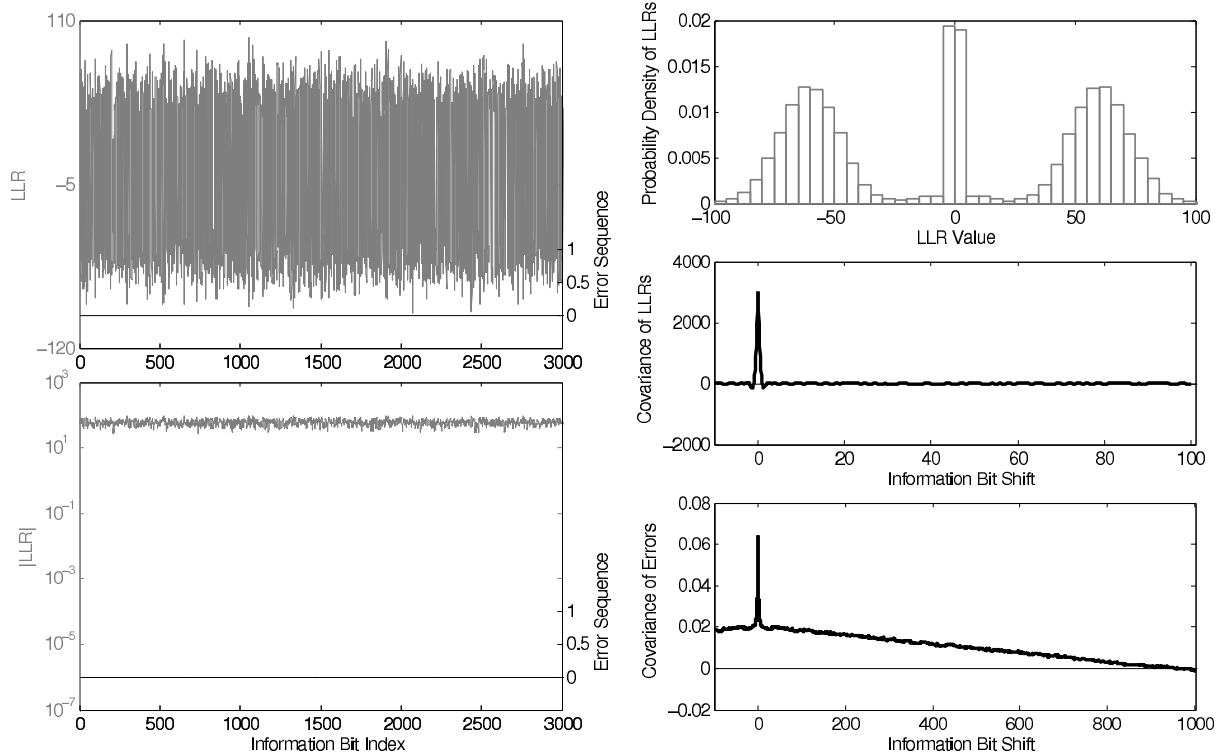
Figure 3.24: SCCC decoding of a code transmitted using 16-PSK modulation at SNR = 5 dB (BER = 8.14×10^{-2})
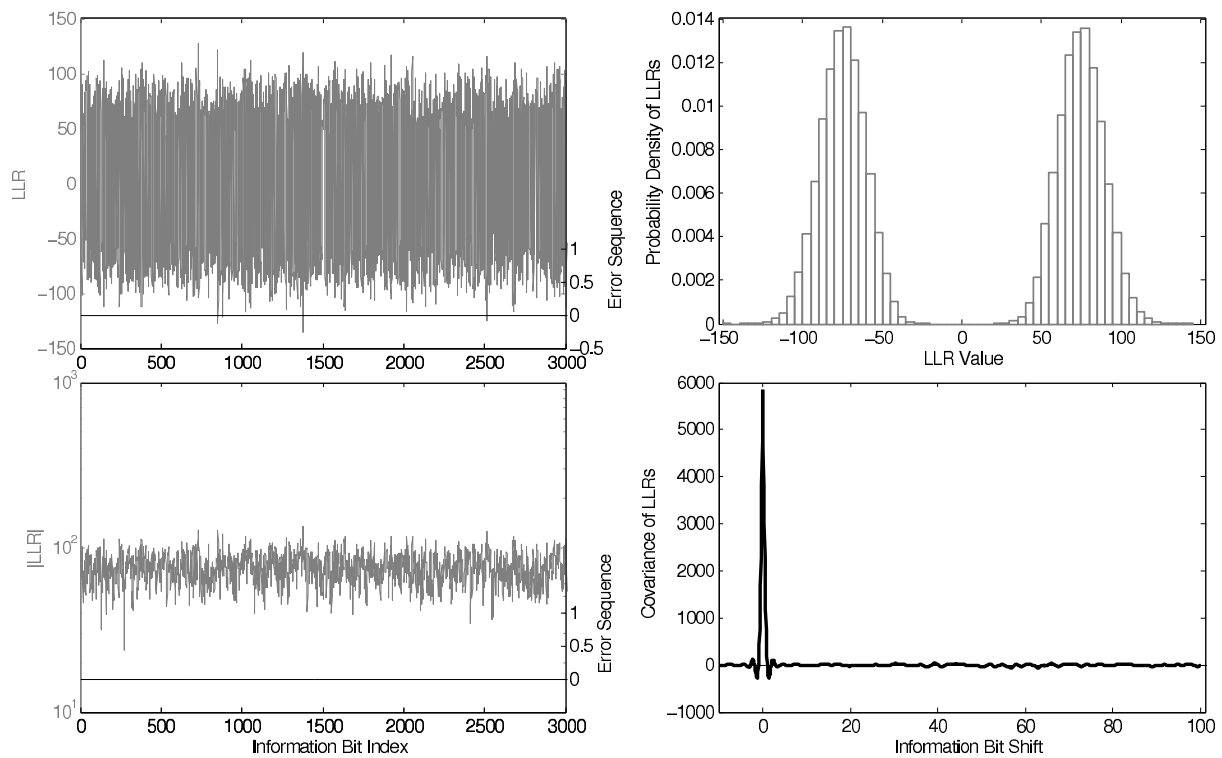
Figure 3.25: SCCC decoding of a code transmitted using 16-PSK modulation at SNR = 6 dB (BER = 7.09×10^{-4})

68

Then, the results of using a catastrophic inner code in SCCC are investigated. We used two different outer codes to see the possible effects of an outer code in decoding. Figures 3.26, 3.27 and 3.28 show the results at 3, 5 and 7 dBs of using an outer code with a generating matrix of $G = [1, 2]$, while figures 3.29 and 3.30 show the results at 7 and 9 dBs of using an outer code with a generating matrix of $G = [7, 5]$. From the
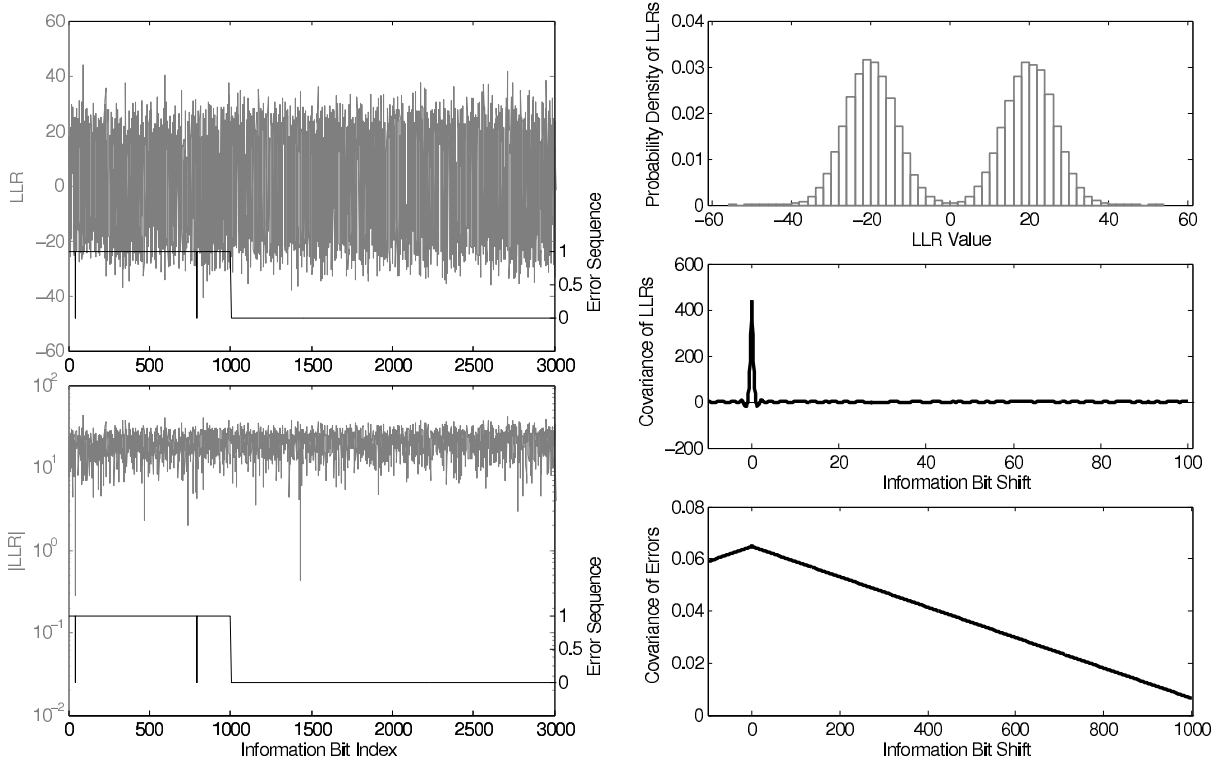
Figure 3.26: SCCC decoding of a catastrophic inner code using an outer code with G = (1,2) at SNR = 3 dB (BER = 9.05×10^{-2}, PER = 5.36×10^{-1})

catastrophic inner codes figures, the following observations can be made:

- As everywhere else, the LLRs are uncorrelated.

- The error correlation in catastrophic SCCC codes does not change much with SNR and it resembles the error correlation in regular SCCC at low SNRs, some packets are received error-free, while others depending on the outer code, have either BER near 0.5 or BER of almost 1.

- The LLRs in the time domain have lost the stair-like that was present in the non-concatenated catastrophic codes before.

- The LLR distribution depends on the outer code too. For some outer codes the LLR distribution has two Gaussian-like lobes, while for some other codes, the LLR distribution is just like catastrophic non-concatenated codes concentrated around zero. In the former case, the error pattern is almost always all-ones or all-zeros in each packet, while in the latter case, the errors are distributed equally on all packets. Both cases, however, lack any interesting feature that can be exploited to further correct the errors.

69
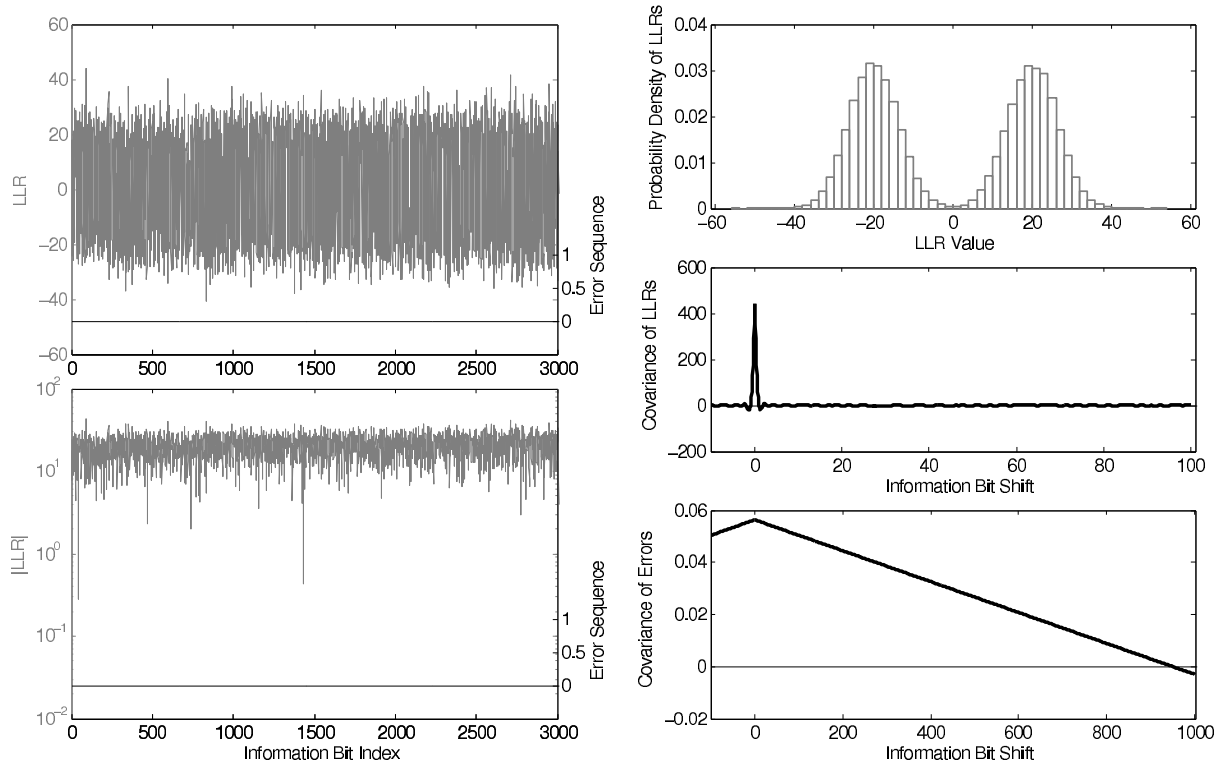
Figure 3.27: SCCC decoding of a catastrophic inner code using an outer code with G = (1,2) at SNR = 5 dB (BER = $6.6{\times}10^{-2}$, PER = $8.1{\times}10^{-2}$)
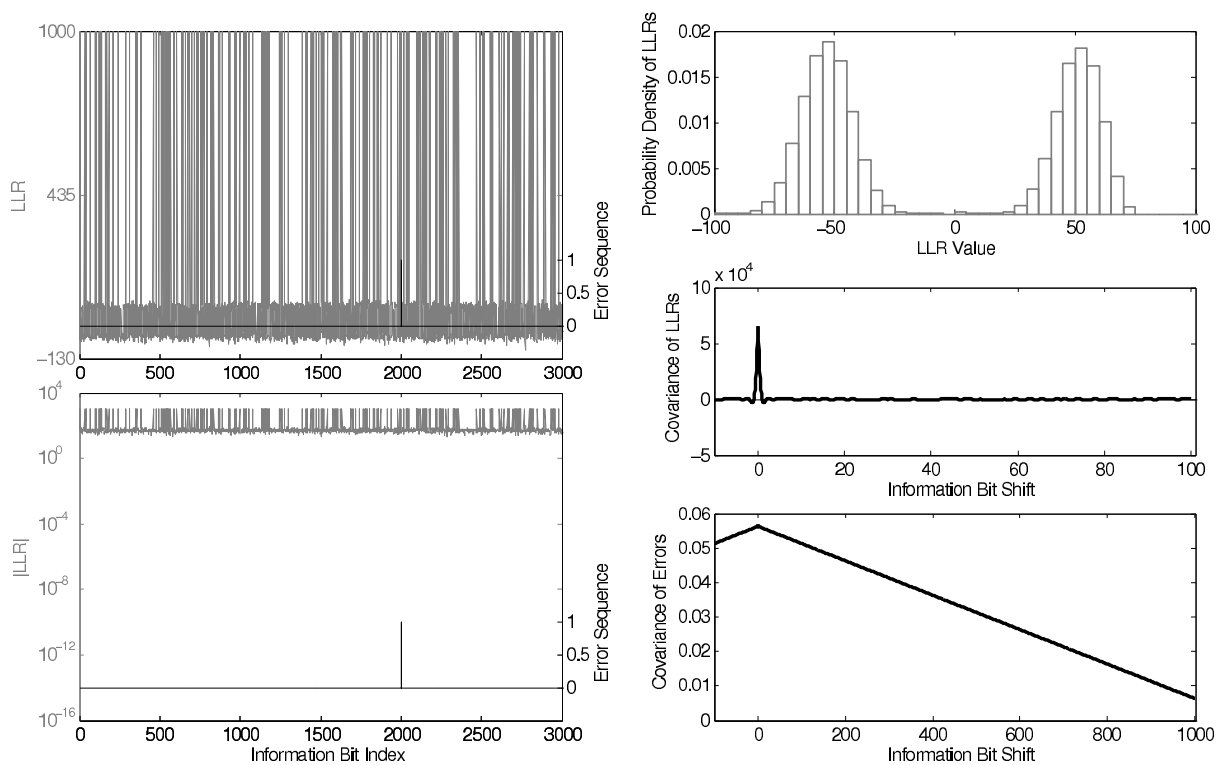
Figure 3.28: SCCC decoding of a catastrophic inner code using an outer code with G = (1,2) at SNR = 7 dB (BER = $4.2{\times}10^{-2}$, PER = $4.5{\times}10^{-2}$)
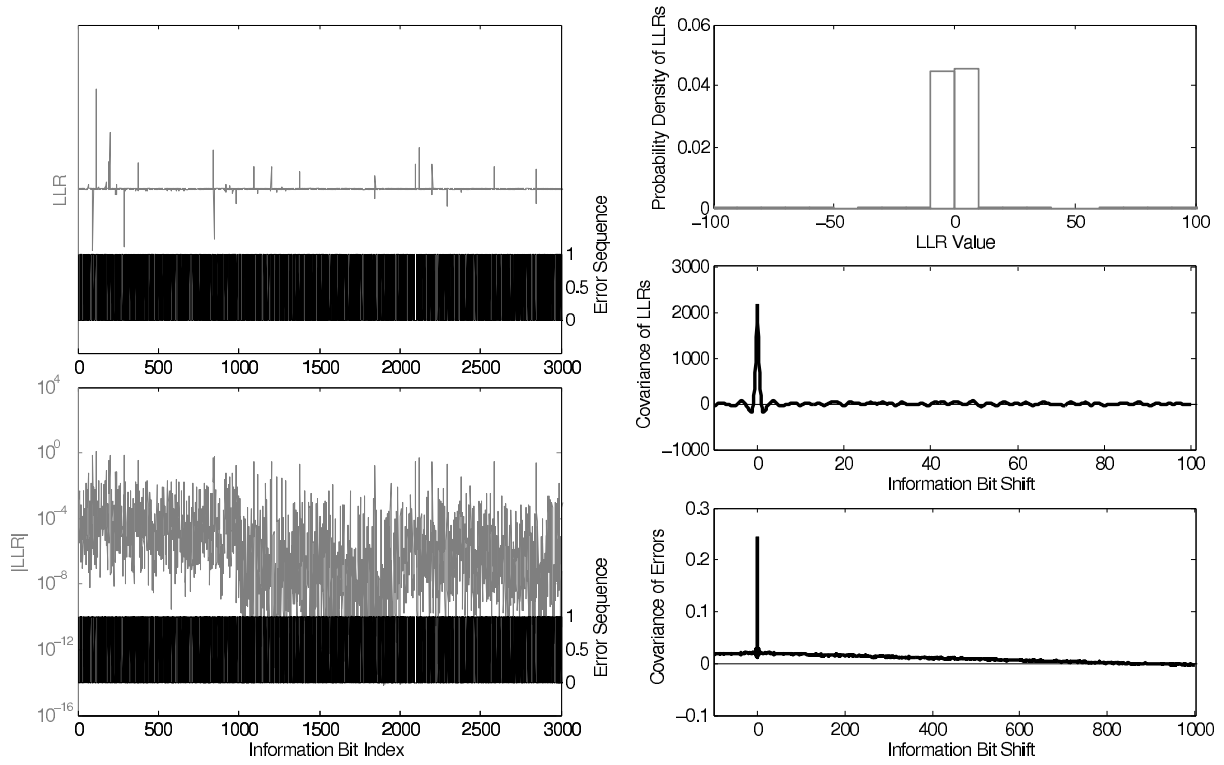
Figure 3.29: SCCC decoding of a catastrophic inner code using an outer code with G = (7,5) at SNR = 7 dB (BER = 8.7×10⁻¹, PER = 3.89×10⁻¹)
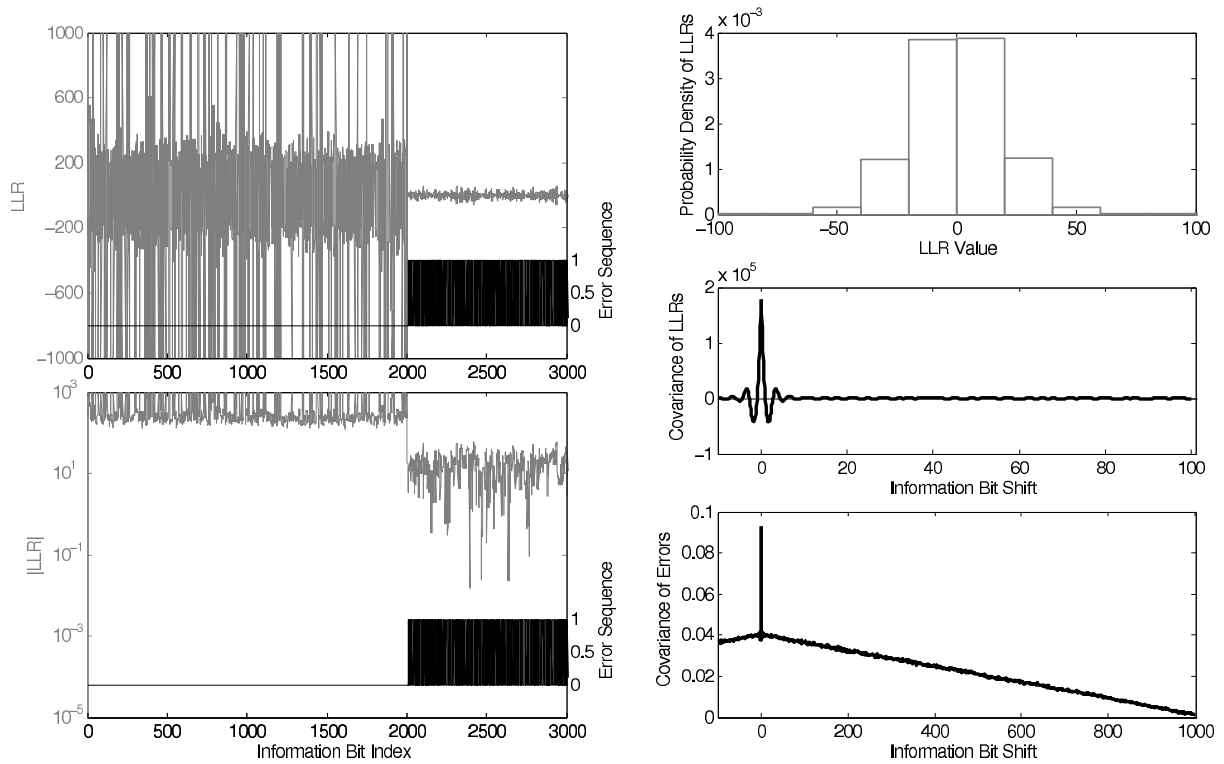
Figure 3.30: SCCC decoding of a catastrophic inner code using an outer code with G = (7,5) at SNR = 9 dB (BER = 7.87×10⁻², PER = 1.59×10⁻¹)

71

It is interesting to note the differences between the behavior of error covariance in regular SCCC as compared to regular convolutional codes. In regular convolutional codes, due to the spread nature of errors, the errors covariance is a fast decaying function to zero, while the SCCC's, due to its bursty packet nature, is a slow decaying function. In both, however, the LLR covariance is similar and very much uncorrelated. This behavior can be interpreted by comparing the LLR distribution. If the LLR distribution is Gaussian-like as in regular convolutional codes, the errors should be almost independent and hence the fast decaying covariance to zero. Therefore, convolutional codes are good for minimizing BER for $R > C$ [51] and result almost like a memoryless BSC channel [52]. While in SCCC, the LLR empirical distribution is not Gaussian because of the middle lobe or peak that occurs at low SNRs. This lobe means that although the LLRs are uncorrelated, they are not independent and thus result in higher correlation of errors that is slowly decaying to zero.

3.7   ARQ Using LLR Values

In this section, we will propose a simple yet very effective method to work as an Automatic Repeat reQuest (ARQ) in normal SCCC. As seen previously, the LLRs in a normal SCCC have 3 lobes before the waterfall, and 2 lobes after the waterfall. The middle lobe concentrated around zero is the lobe that actually produces most of the errors making the BER soaring to high values. When this lobe gradually disappear and only 2 lobes are present then the BER falls to very low values. Another very interesting observation is that always *whole packets* are either in middle zero lobe or divided between the two side lobes. Thus, it can be said that packets are divided into bad packets with so many errors, or good packets with almost no errors at all.

The method is very simple and is explained in the steps below:

1. Compute the average value of $|LLR|$ in each packet of the received sequence. $LLR_{av}^i = \frac{1}{K} \sum_{j=1}^{K} |LLR_j^i|, \forall i = 1, \ldots, N$ where i is the packet number, N is the total number of packets, K is the number of information bits per packet and j is the bit index within a packet.

2. Come up with some threshold T that will divide the bad packets from the good ones. We used a simple direct approach by taking $T = \frac{\max(LLR_{av}) + \min(LLR_{av})}{2}$.

3. Decode all packets with $LLR_{av} > T$ and request a retransmission for packets with $LLR_{av} < T$

4. The BER is calculated only for those packets where $LLR_{av} > T$, and thus the transmission throughput will decrease because of the dropping of all the packets with $LLR_{av} < T$.

The following Figure 3.31 shows the BER before and after applying this method for a simulation of 1000 packets of 1000-bits each, using the QPSK SCCC 1/2/3 code with $G_{outer} = [7,5]$ and $G_{inner} = [17,6,15](K = 1000, N = 1000, T$ is computed in the program). The BER after applying the method is composed of the packets that were decoded only

(i.e. The packets that requests a retransmission were not included in the BER calcula-
tion). The percentage of packets that need retransmission is shown in Figure 3.32. As
seen in the figures, the proposed ARQ method actually does a perfect job in removing
the damaged packets and keeping the good packets.
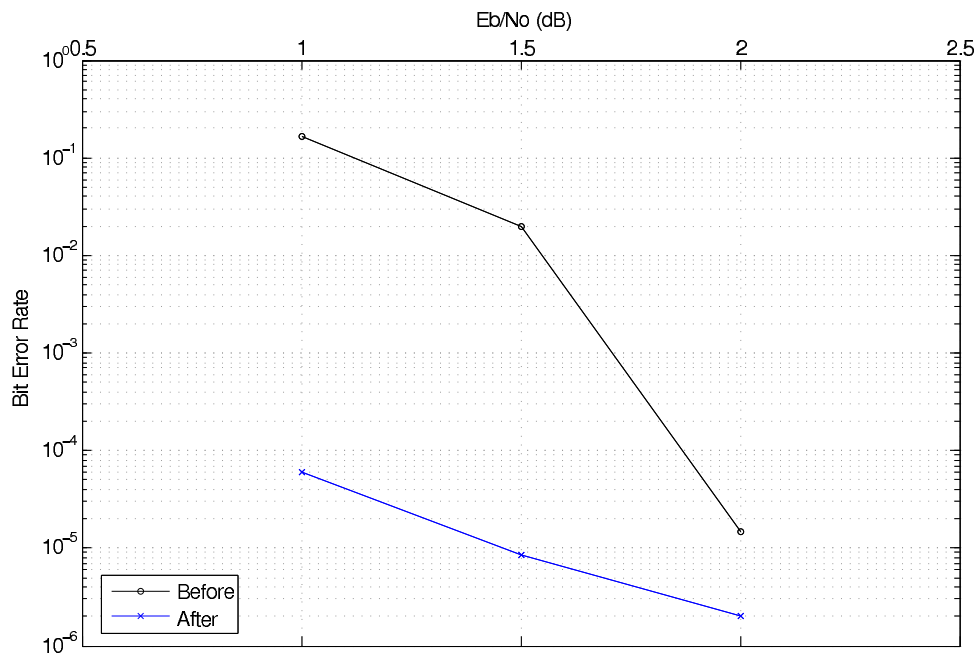
Figure 3.31: Applying the ARQ method
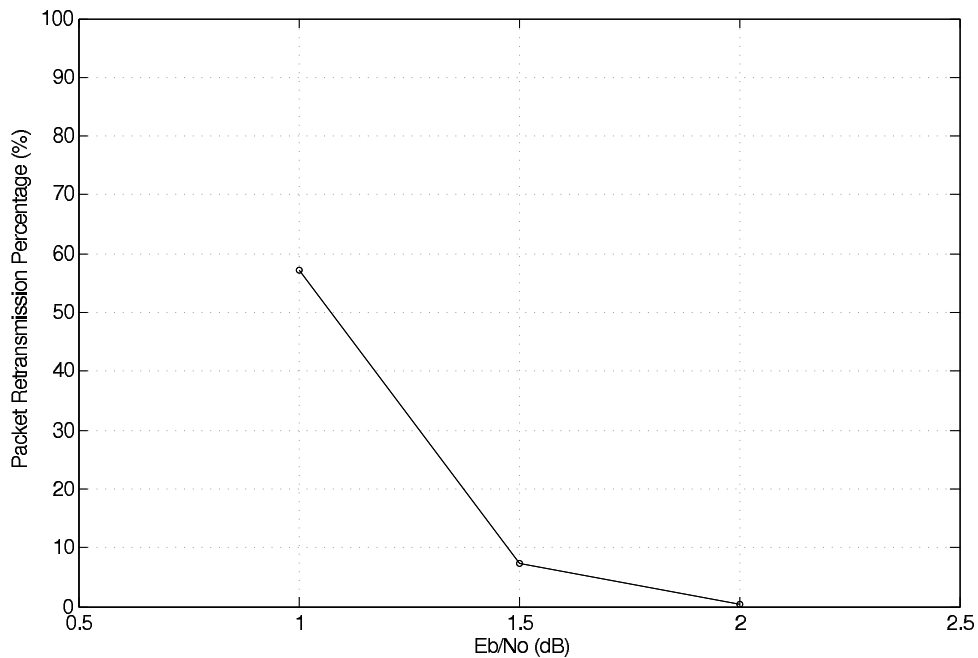
Figure 3.32: Packet percentage that needed retransmission

Chapter 4

Summary

In this concluding chapter, we will summarize our results and compare between the use of catastrophic codes and SCCC in the security context. We will also provide a simple example of the usage of this type of security in a real life scenario. And conclude by proposing some areas that can be investigated in a future work.

4.1 Convolutional Catastrophic Codes vs. SCCC

As seen in the preceding chapters, we investigated the use of catastrophic codes and SCCC codes in a security context where the wiretapper (Eve) is at a disadvantage of a more noisy signal than the legitimate receiver (Bob). Our goal as described in Chapter 1 is to find a code that has a sharp movable threshold in the BER curve, with low error correlation.

From our results we can provide the following comparison between the coding schemes as seen in Table 4.1. An interesting observation found in the codes is the phenomenon of good-bad packets. As seen in catastrophic codes, packets are divided into error-free packets or very bad packets with an average BER of 0.5. The same can be said about the SCCC scheme. In regular SCCC, we were able with a very simple method to eliminate almost all the bad packets and keep the almost error-free packets. The only difference is that in catastrophic codes, the errors within one packet come in long all-error or error-free periods, thus having a high correlation (as seen in Figure 2.28 and Figure 2.29), while bad packets in SCCC have the bits flipping rapidly between error-bits or correct-bits as seen in Figure 3.18, and hence have lower correlation. This makes the bad packets in the regular SCCC scheme almost impossible to correct below the threshold since they have almost independent errors. But on the other hand, this behavior enables us to easily extract all the good packets received. Using the SCCC scheme with a catastrophic inner encoder also has this property, but similar to regular catastrophic codes, there is no correlation between the LLRs and errors (since there is no middle peak in the LLR distribution), which means that there is no way to detect the bad packets as in regular SCCC.

Thus, we can conclude:

- Although regular SCCC provides the best BER slope after the threshold, it cannot

Table 4.1: Comparison between catastrophic codes and SCCC codes.

	Catastrophic Codes	Regular SCCC Codes	Catastrophic SCCC
SNR Threshold	Middle-High range	Low-Middle range	Middle-High range
Threshold	Not sharp as SCCC	Sharp threshold and very steep slope	Similar to regular catastrophic codes
Packet length	Around 1k-10k and even longer packets for a steeper slope	Around 1k-10k	10k or longer
Implementation	Simple Viterbi	Complex iterative SISO	Complex iterative SISO
Error correlation	High	Lower than catastrophic codes	Similar to regular SCCC
Error and LLR Correlation	Low	High	None
Decoding possibility below threshold	Unlikely even when error correlation and $LLRs$ are related	Good packets can be extracted easily	Very hard

be considered secure because of the high correlation between the bad packets and the LLRs. This means that a good amount of the received data can be extracted using a simple ARQ method.

- Catastrophic codes used in regular convolutional codes or SCCC do not show correlation between LLRs and errors, which keeps the system secure in low SNR's. The price that must be paid is the higher value of SNR_2, which makes it unsuitable for use in low SNR regions.

- Regular catastrophic codes provide a simple and easy way to provide some security in mid-high SNR regions through adjusting the BER curves by using packet lengths and lattice labeling.

- Catastrophic codes used in an SCCC provide a promising scenario that could be explored by using a good outer code and adjusting all the other parameters such as: modulation, puncturing, packet length, ..., etc.

The following example demonstrates a possible use of such codes.

Example:
A building-based company needs to implement secure wireless communications between its employees, while forbidding everyone else. Since all the employees are present physically inside the building, the communication routers or hubs can be positioned in places giving all the employees a direct line of sight (LOS) communication ensuring a high SNR

to all employees. The high SNR at the employees will ensure an almost error-free environment inside the company using either codes with some ARQ protocol enabled to ensure that no packets are lost. The walls of the company building will act as a barrier lowering the SNR to anyone present physically outside the building. This low SNR will prevent anyone from getting any useful information of what is being sent inside.

4.2 Future Work

Throughout the course of the study of this thesis, several problems arose and may be investigated further in the future.

- Catastrophic codes are found to have a relation between the $LLRs$ and the catastrophic errors. This relation may be investigated further to see if a wiretapper may be able to correct the errors or not.

- Analyze the low SNR region in SCCC, the correlation between the $LLRs$ and bit errors needs further analysis to better understand the causes and possible corrections, if possible.

- Study and optimize the proposed ARQ method as well as its relation to the resulting throughput.

- Catastrophic codes (in regular convolutional codes or SCCC) should be explored further in the context of physical layer security.

Bibliography

[1] A. Goldsmith. *Wireless Communications*. Cambridge University Press, New York, USA, 2005.

[2] Wireless security. *Wikipedia, the free encyclopedia.* http://en.wikipedia.org/wiki/Wireless_security.

[3] Network security tips. http://www.linksys.com.

[4] Wireless security primer. *WindowSecurity.com.* http://www.windowsecurity.com/articles/Wireless_Security_Primer_Part_II.html.

[5] D. Hardy, G. Malleus, and J.N Mereur. *Networks: Internet, Telephony, Multimedia.* Springer France, 2001.

[6] Introduction to public-key cryptography. *Sun Microsystems Documentation.* http://docs.sun.com/source/816-6154-10/contents.htm.

[7] A. Salomaa. *Public Key Cryptography*. Springer, New York, 1996.

[8] J. F. Humphreys and M. Y. Prest. *Numbers, Groups & Codes*. Cambridge University Press, New York, USA, 2004.

[9] Rsa. *Wikipedia, the free encyclopedia.* http://en.wikipedia.org/wiki/RSA.

[10] Public-key cryptography. *Wikipedia, the free encyclopedia.* http://en.wikipedia.org/wiki/Public_key.

[11] Information-theoretic cryptography. *Advances in Cryptology*, 1666:47–64, August 1999.

[12] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949.

[13] A. D. Wyner. The wire-tap channel. *Bell System Technical Journal*, 54:1355–1387, 1975.

[14] S. K. Leung-Yan-Cheong. On a special class of wire-tap channels. *IEEE Transactions on Information Theory*, 23:625–627, September 1977.

[15] S. K. Leung-Yan-Cheong and Martin E. Hellman. The gaussian wire-tap channel. *IEEE Transactions on Information Theory*, 24:451–456, July 1978.

[16] I. Csiszár and J. Korner. Broadcast channels with confidential messages. *IEEE Transactions on Information Theory*, 24(3):339–348, 1978.

[17] U. M. Maurer. Secret key agreement by public discussion from common information. *IEEE Transactions on Information Theory*, 39(3):733–742, May 1993.

[18] U. Maurer and S. Wolf. Unconditionally secure key agreement and intrinsic conditional information. *IEEE Transactions on Information Theory*, 45(2):499–514, March 1999.

[19] R. Ahlswede and I. Csiszár. Common randomness in information theory and cryprography. i. secret sharing. *IEEE Transactions on Information Theory*, 39(4):1121–1132, July 1993.

[20] C. H. Bennett, G. Brassard, C. Crépeau, and U. Maurer. Generalized privacy amplification. *IEEE Transactions on Information Theory*, 41(6):1915–1923, 1995.

[21] L. H. Ozarow and A. D. Wyner. Wire tap channel ii. *AT&T Bell Laboratories Technical Journal*, 63(10):2135–2157, December 1984.

[22] V. Wei. Generalized hamming weights for linear codes. *IEEE Transactions on Information Theory*, 37(5):1412–1418, September 1991.

[23] J. Muramatsu. Secret key agreement from correlated source outputs using low density parity check matrices. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E89-A(7):2036–2046, July 2006.

[24] M. Bloch, A. Thangaraj, S. W. McLaughlin, and J. M. Merolla. Ldpc-based secret key agreement over the gaussian wiretap channel. *IEEE International Symposium on Information Theory*, pages 1179–1183, July 2006.

[25] A. Thangaraj, S. Dihidar, A. R. Calderbank, S. W. McLaughlin, and J. M. Merolla. On achieving capacity on the wire tap channel using ldpc codes. *Proceedings IEEE International Symposium on Information Theory*, pages 1498–1502, September 2005.

[26] A. Thangaraj, S. Dihidar, A. R. Calderbank, S. W. McLaughlin, and J. M. Merolla. Applications of ldpc codes to the wiretap channel. *arXiv:cs.IT/0411003v3*, January 2007.

[27] M. Bloch, J. Barros, M. R. D. Rodrigues, and S. W. McLaughlin. Wireless information-theoretic security - part i: Theoretical aspects. *arXiv:cs.IT/0611120v1*, November 2006.

[28] M. Bloch, J. Barros, M. R. D. Rodrigues, and S. W. McLaughlin. Wireless information-theoretic security - part ii: Practical implementation. *arXiv:cs.IT/0611121v1*, November 2006.

[29] Y. Liang and H. V. Poor. Generalized multiple access channels with confidential messages. *arXiv:cs.IT/0605084v1*, May 2006.

[30] R. Liu and H. V. Poor. Multi-antenna gaussian broadcast channels with confidential messages. *arXiv:0804.4195v1 [cs.IT]*, April 2008.

[31] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara. Serial concatenation of interleaved codes: Performance analysis, design, and iterative decoding. *IEEE Transactions on Information Theory*, 44(3):909–926, May 1998.

[32] M. P. C. Fossorier, S. Lin, and D. J. Costello. On the wight distribution of terminated convolutional codes. *IEEE Transactions on Information Theory*, 45(5):1646–1648, July 1999.

[33] J. G. Proakis. *Digital Communications*. McGraw-Hill, Inc., New York, NY, USA, 2001.

[34] Shu Lin and D. J. Costello Jr. *Error Control Coding*. Pearson Education, Inc., Upper Saddle River, NJ, USA, 2004.

[35] J. L. Massey and M. K. Sain. Inverses of linear sequential circuits. *IEEE Transactions on Computers*, C-17(4):330–337, April 1968.

[36] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory*, 20(2):284–287, March 1974.

[37] D. F. Yuan and X. H. Shan. Research on the viterbi and bcjr decoding schemes of convolutional codes under different sources. *IEEE Vehicular Technology Conference*, 2:1377–1381, 2001.

[38] The Springer International Series in Engineering and Computer Science. *Fundamentals of Codes, Graphs, and Iterative Decoding*. Springer Netherlands, 2002.

[39] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo codes. *Proceedings of IEEE International Conference on Communications*, pages 1064–1070, May 1993.

[40] C. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, July 1948.

[41] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. Oliver & Boyd, London, third edition, 1948.

[42] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara. A soft-input soft-output app module for iterative decoding of concatenated codes. *IEEE Communication Letters*, 1(1):22–24, January 1997.

[43] X. Gu, K. Niu, and W. Wu. A novel efficient soft output demodulation algorithm for high order modulation. *IEEE International Conference on Computer and Information Theory*, pages 493–498, September 2004.

[44] A. Banerjee, F. Vatta, B. Scanavino, and D. J. Costello. Nonsystematic turbo codes. *IEEE Transactions on Communications*, 53(11):1841–1849, November 2005.

[45] H. Kim and G. L. Stuber. Rate compatible punctured sccc. *IEEE Vehicular Technology Conference*, 4:2399–2403, 2001.

[46] H. El-Gamal and Jr. A. Hammons. Analyzing the turbo decoder using the gaussian approximation. *IEEE Transactions on Information Theory*, 47(2):671–686, February 2001.

[47] S. ten Brink. Convergence behavior of iteratively decoded parallel concatenated codes. *IEEE Transactions on Communications*, 49(10):1727–1737, October 2001.

[48] J. W. Lee and R. E. Blahut. A note on the analysis of finite length turbo decoding. *IEEE International Symposium on Information Theory*, page 83, July 2002.

[49] J. W. Lee and R. E. Blahut. Generalized exitchart and ber analysis of finite-length turbo codes. *IEEE Global Telecommunications Conference*, 4:2067–2072, December 2003.

[50] J. W. Lee and R. E. Blahut. Convergence analysis and ber performance of finite-length turbo codes. *IEEE Transactions on Communications*, 55(5):1033–1043, May 2007.

[51] J. B. Huber and T. Hehn. The lowest-possible ber and fer for any discrete memoryless channel with given capacity. *arXiv:0801.1067v2 [cs.IT]*, May 2008.

[52] S. Huettinger, J. B. Huber, R. F. H. Fischer, and R. Johannesson. Soft-output-decoding: Some aspects from information theory. *Proceedings of the 4th ITG Conference Source and Channel Coding*, pages 81–89, January 2002.

Appendix A: Simulation Code Description

All Simulations were done using C++, the following is a description of the codes used.

Description of the Code Used in Viterbi Simulation

The code is listed in "Viterbi.txt". The top section of the code is the data input section, which must be filled prior to each simulation. The input parameters include the following:

- $modk$: The number of binary bits in each lattice symbol.

- k : The number of binary encoder input bits.

- n : The number of binary encoder output bits.

- m : The length of the encoding register in bits.

- $G[n]$: The n generating polynomials in octal form.

- $Pkts$: The number of packets to be simulated.

- $EbNo\_start$: The first SNR per bit to be simulated.

- $EbNo\_end$: The last SNR per bit to be simulated.

- $limit\_sim\_bits$: The packet bit length.

- $limit\_sim\_errors$: The number of errors needed to stop the simulation.

- $Lattice[2^{modk+1}]$: The position of the 2^{modk} complex lattice points (a+jb) in order. (i.e. $[a_0, b_0, a_1, b_1, \ldots, a_{2^{modk}-1}, b_{2^{modk}-1}]$).

The code will be described in a flowchart. Four important tables are mentioned in the simulation process. These tables are described below:

- "Decoding reference table": This table is filled at the beginning. It includes all the possible transitions that occur within a convolutional code trellis. It associates each starting-state and input, with the corresponding end-state and output.

- "Current state path metric table": This table holds the updated path metric for each state computed to include the last received symbol. The table holds one path metric per state (the surviving path metric).

- "Current state transition table": This table is used at the reception of each symbol. It is filled with the new updated path metrics of all possible transitions between all states. The updated path metrics are computed by taking the path metric value of the starting state from the "Current state path metric table" and adding it to the branch metric computed for each transition.

- "State history table": This table holds the history of the trellis paths leading to each current state (the surviving paths). It is filled during simulation as each received symbol is decoded. After filling the "Current state transition table", the path metrics are compared to find the transition leading to the lowest metric for every state. The transitions are used to determine the preceding state to the new state and eliminate all other paths having higher metrics. The preceding states are added to the table.

The flowchart is shown in Figure A-1 and Figure A-2

Notes regarding the simulation:

- The SNRs simulated in the program start from $EbNo\_start$ and end at $EbNo\_end$ with increment value of 1.

- The lattice symbols $(a_i, b_i), i = \{0, 1, \ldots, 2^{modk} - 1\}$ are scaled down to $(\delta a_i, \delta b_i)$ such that $\frac{1}{2^{modk}} \sum_{i=0}^{2^{modk}-1} \sqrt{(\delta a_i)^2 + (\delta b_i)^2} = 1$

- The binary generating polynomials are computed from the octal input $G[n]$.

- The transmission and reception registers are used to regulate the flow of the symbols especially when $modk \neq n$.

- The AWGN σ is calculated from the input E_b/N_0 as follows:

$$
\begin{array}{rcl}
\frac{E_s}{N_0}(\text{dB}) & = & \frac{E_b}{N_0}(\text{dB}) + 10\log_{10} modk + 10\log_{10} \frac{k}{n} \\
N_0 & = & 10^{\frac{-E_s}{N_0}/10} \quad \text{since } Es = 1 \\
\sigma & = & \sqrt{\frac{N_0}{2}}
\end{array}
$$

- The decoding is done by taking a decoding delay (lookback) of $5 \times \left\lceil \frac{m}{k} \right\rceil$ as suggested in [33].

- The program output the results per SNR on the screen as well as in a file "Summary.txt". Another file "Errors.txt" is generated that contains the error vectors in the simulation.

Description of the Code Used in BCJR Simulation

The listing of the code is found in "BCJR.txt". This code also has the regular Viterbi algorithm embedded, and thus can be used for Viterbi decoding (Description of the Viterbi part is omitted since it is similar to the above section). The top section of the
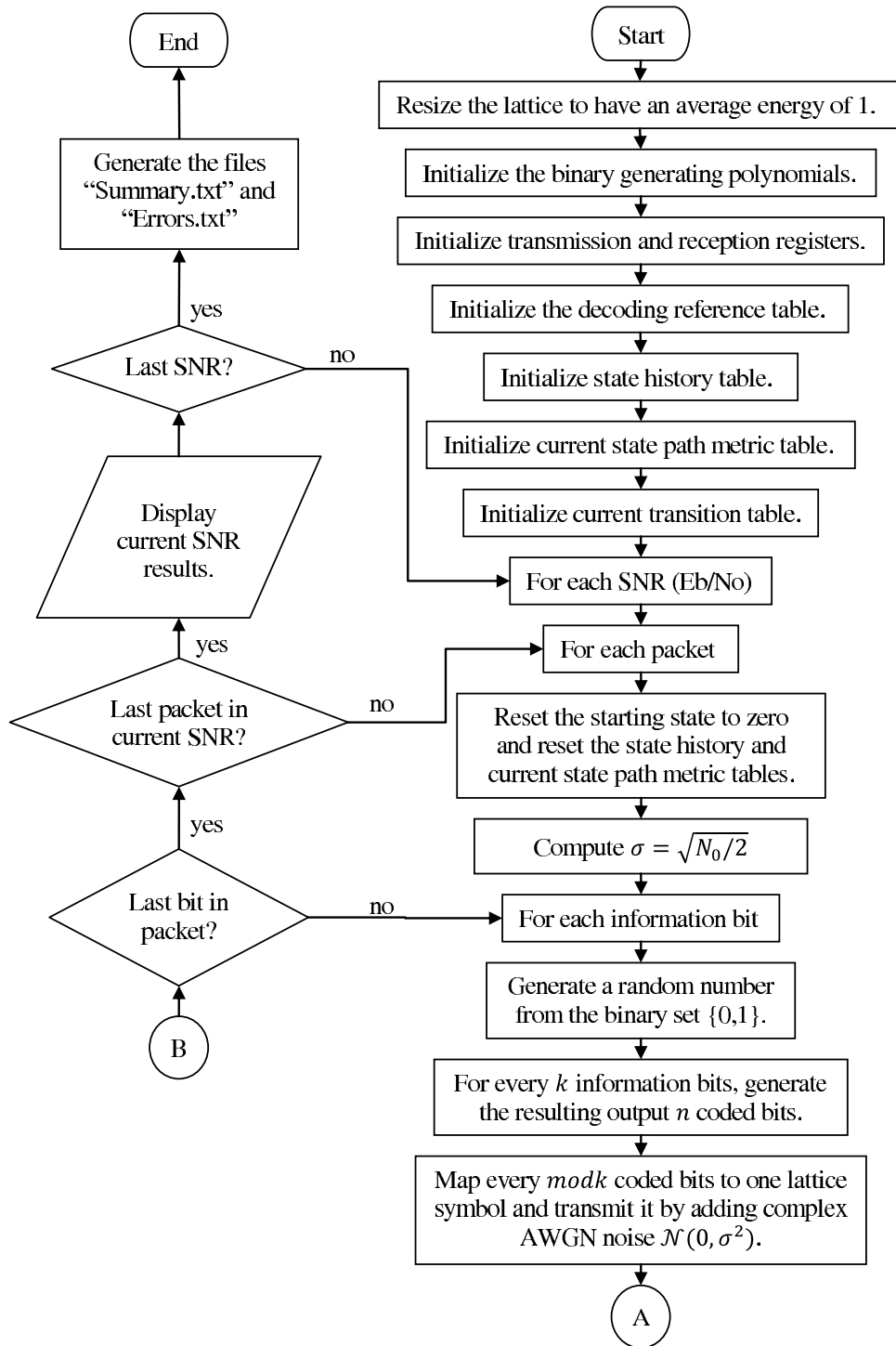
Figure A-1: Flowchart of the Viterbi simulation code part 1
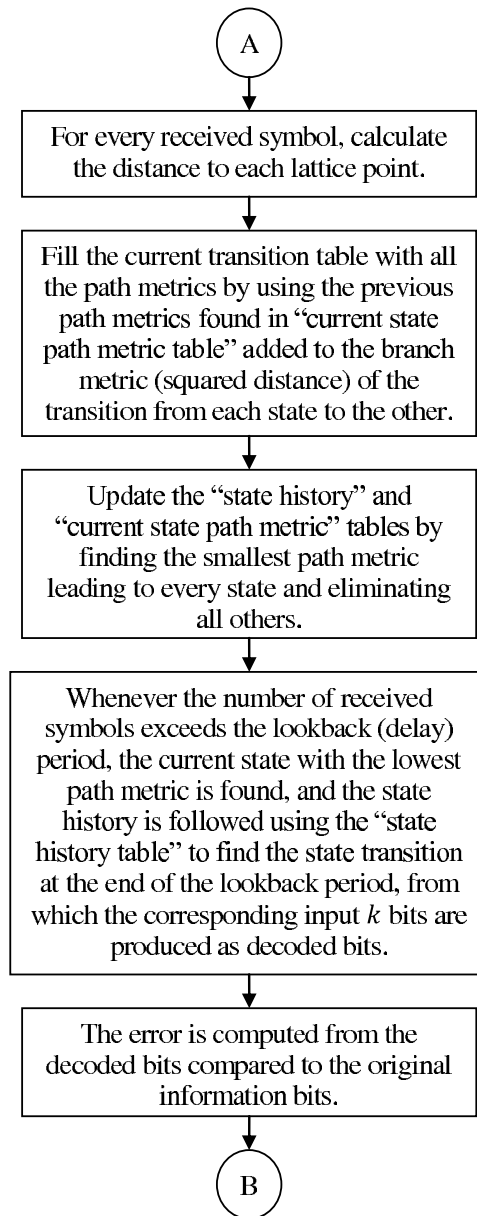
Figure A-2: Flowchart of the Viterbi simulation code part 2

BCJR code is the input section which is similar to the Viterbi code above. There is no need to reiterate the parameters. One extra parameter however is $BCJR$ which is used to select Viterbi decoding if it is reset to 0, while it selects the BCJR algorithm when it is set to 1. The tables however are different, the "state history" and the "current state path metric" tables are replaced by the following three tables:

- "Alpha table": The alpha table is updated at the reception of each new symbol. It stores the forward path metrics $(\alpha(s))$ of all the states and keeps them for the whole delay period to be used in the calculation of the information bit LLRs and decoding decision afterwords.

- "Gamma table": The gamma table is very similar to the alpha table except that it stores the whole branch metrics $(\gamma(s', s))$ of all the state transitions.

- "Beta table": Unlike the "alpha" and "gamma" tables which are updated at the reception of each new symbol, the beta table needs the lookback (delay) period to pass first. When it passes, it updates the *whole* table for every new received symbol (again, unlike the other two tables), and thus helps in making a decision about the state transition at the beginning of the delay period. The beta table holds the computed data for the backward path metrics $(\beta(s))$.

The flowchart of the code is shown in Figure A-3 and Figure A-4.

Notes on the BCJR code:

- The detailed mathematical background of all the computations included is found in section 2.3.1.

- In addition to the generated files in the Viterbi code, the BCJR code also generates "LLRs.txt" file including all the information bits LLRs.

- In both Viterbi and BCJR algorithms, the program uses a look-back window of length $5 \times \left\lceil \frac{m}{k} \right\rceil$ as suggested in [33].

- This BCJR code decodes using the Euclidean distance to the transmitted symbols. Another BCJR code listed in "BCJR_sd.txt" uses the soft demodulation algorithm used in SCCC to compute the probabilities of the coded bits, and a decoding algorithm similar to SCCC outer (or inner) codes. This code does not use windowing (the whole packet is used).

Description of the Code Used in SCCC Simulation

The listing of the code is found in "SCCC.txt". Similar to the Viterbi and BCJR decoding programs, the top portion of the code is for the input parameters. The following list contains all the parameters used in the simulations.

- $modk$: The number of binary bits in each lattice symbol.

The Inner Encoder:

- k_i : The number of binary inner encoder input bits.

- n_i : The number of binary inner encoder output bits.

- m_i : The length of the inner encoding register in bits.

- $G_i[n_i]$: The n_i generating polynomials in octal form.
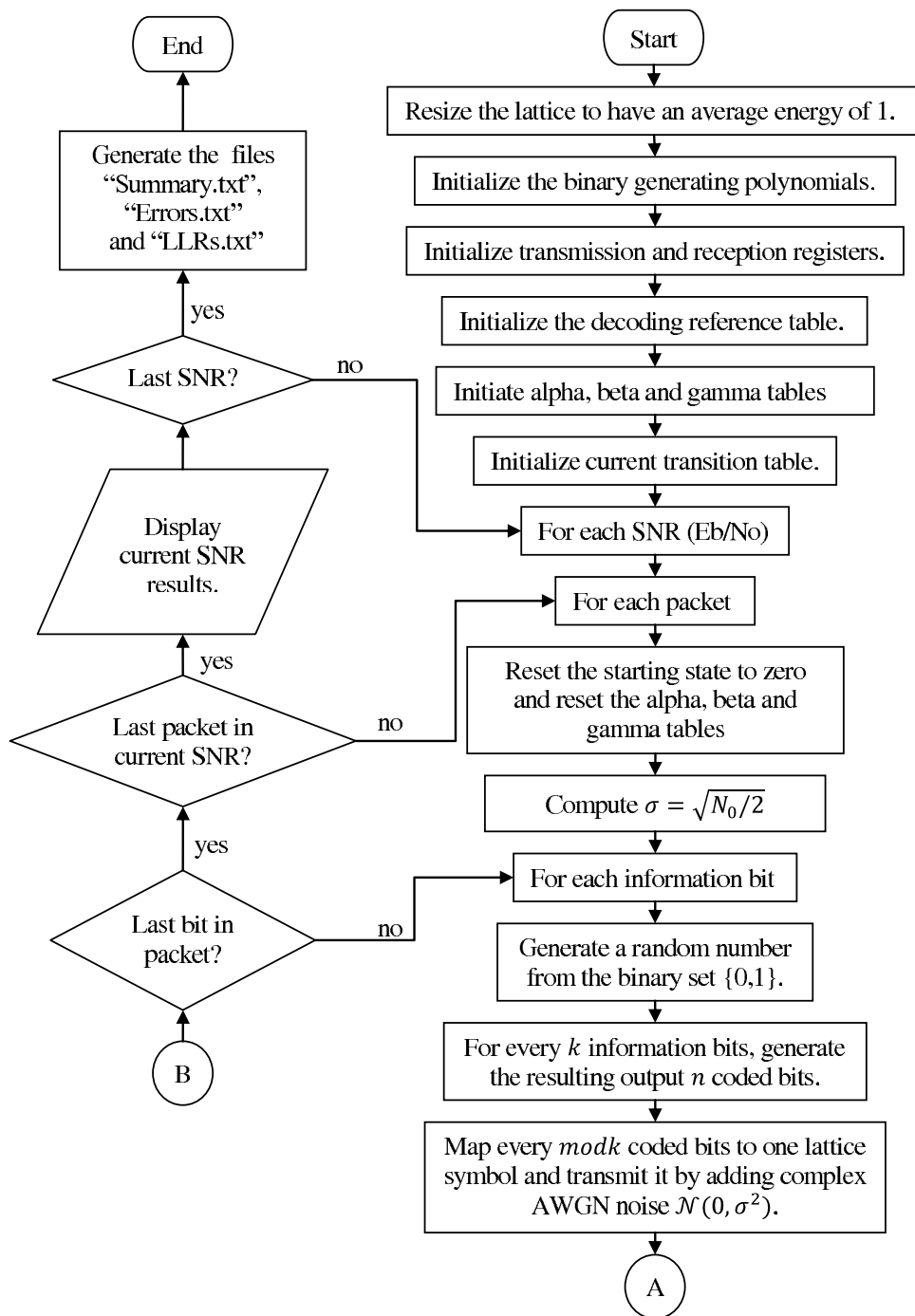
Figure A-3: Flowchart of the BCJR simulation code part 1

A

For every received symbol (r), calculate the branch metrics (γ) corresponding to each lattice point (v). $\gamma = \frac{L_c}{2} r.v$

Fill the current transition table with all the path metrics by using the previous forward path metrics found in the "alpha table" added to the branch metric (γ) of the transition from each state to the other. At the same time update the "gamma table" by including the new γ's.

Update the "alpha table" by including the new forward path metrics (α) for all the states.

Whenever the number of received symbols exceeds the lookback (delay) period, reset the "current transition" and "beta" tables. The initial values of the "beta table" are taken from the "alpha table".

For every state transition, compute the current transition metrics (in reverse) by adding the appropriate already computed branch metrics (γ) and the (β) values found in the "beta table"

Update the "beta table" by computing all the backward path metrics (β) of all the states.

Lookback (delay) state transitions ended? — no / yes

The decoded bits are computed from the three tables (alpha, beta and gamma), and then compared to the original information bits to find the errors.

B

Figure A-4: Flowchart of the BCJR simulation code part 2

The Outer Encoder:

- k_o : The number of binary outer encoder input bits.

- n_o : The number of binary outer encoder output bits.

- m_o : The length of the outer encoding register in bits.

- $G_o[n_o]$: The n_o generating polynomials in octal form.

Some other input parameters:

- K : The interleaver length.

- *iterations* : Number of iterations wanted in decoding.

- $P[c \times n_i]$: The puncturing matrix used. c can be any constant above-zero integer. (All ones for no puncturing).

- BE : The number of consecutive bits to flip before transmission of each packet. (0 for normal).

- PBF : The probability of a random BSC-like flip of the packet bits before transmission. (0 for normal).

- $Pkts$: The number of packets to be simulated.

- $EbNo\_start$: The first SNR per bit to be simulated.

- $EbNo\_end$: The last SNR per bit to be simulated.

- $EbNoinc$: The increment value between successive SNRs.

- $Lattice[2^{modk+1}]$: The position of the 2^{modk} complex lattice points (a+jb) in order. (i.e. $[a_0, b_0, a_1, b_1, \ldots, a_{2^{modk}-1}, b_{2^{modk}-1}]$).

The tables in this program are very similar to the BCJR program tables, with one difference is that there are two alpha tables (one for the inner and the other for the outer code), and two decoding reference tables. The flowchart of the SCCC code is shown in Figures A-5, A-6, and A-7.

Notes on the SCCC code:

- The detailed mathematical background of all the computations included is found in section 3.1.3.

- This code does not use windowing and thus it computes the data for each packet as a whole. No windowing is used here because of the use of an interleaver, which requires the whole sequence of bits to be available.

Figure A-5: Flowchart of the SCCC simulation code part 1

A

Puncture, introduce burst errors or BSC-like errors to the coded sequence

Map every $mod\,k$ coded bits to one lattice symbol and
transmit it by adding complex AWGN noise $\mathcal{N}(0, \sigma^2)$.

Demodulate using soft-bit decisions to produce
the probability distribution of each coded bit.

If puncturing was used, add the probabilities of the missing coded bits as 0.5

C → For each iteration

Reset the current transition table

For every received symbol, update the "gamma table" by computing the branch metrics (γ)
by multiplying the demodulated bits probabililties with corresponding input/output bits of
each branch of the inner code. At the same time fill the current trasition table with the path
metrics of all transitions by using the "inner code alpha table" for the previous metrics.

Update the "inner code alpha" table by including the
new forward path metrics (α) for all the states.

keep repeating until all
received sequence is used.

Set the initial values for the "beta table" using the last values of the "inner alpha table" .

Reset the current transition table

For every state transition, compute the current transition metrics
(in reverse) by adding the appropriate already computed branch
metrics (γ) and the (β) values found in the "beta table"

Update the "beta table" by computing all the
backward path metrics (β) of all the states.

keep repeating until all
received sequence is used.

B

Figure A-6: Flowchart of the SCCC simulation code part 2

B

Compute the inner decoder uncoded bits output probabilities for the whole sequence by using the "inner alpha", "beta" and "gamma" tables. Then deinterleave the result to be used as coded bits input for the outer decoder.

Reset the current transition, "beta" and "gamma" tables.

For every symbol, update the "gamma table" by computing the branch metrics (γ) by multiplying the coded input bits probabililties with corresponding input/output bits of each branch of the outer code. At the same time fill the current trasition table with the path metrics of all transitions by using the "outer code alpha table" for the previous metrics.

keep repeating until all the sequence is used.

Update the "outer code alpha table" by including the new forward path metrics (α) for all the states.

Set the initial values for the "beta table" using the last values of the "outer alpha table".

Reset the current transition table

For every state transition, compute the current transition metrics (in reverse) by adding the appropriate already computed branch metrics (γ) and the (β) values found in the "beta table".

keep repeating until all the sequence is used.

Update the "beta table" by computing all the backward path metrics (β) of all the states.

Compute the updated outer decoder coded bits output probabilities for the whole sequence by using the "outer alpha", "beta" and "gamma" tables. Then interleave the result to be used as uncoded bits input for the inner

C ← no ← Last Iteration?

yes

Compute the outer decoder uncoded bits output probabilities for the whole sequence by using the "outer alpha", "beta" and "gamma" tables. Then take a hard decision for each bit to be 0 or 1. These results are the final decoded sequence, which are compared to the original information sequence to calculate the errors.
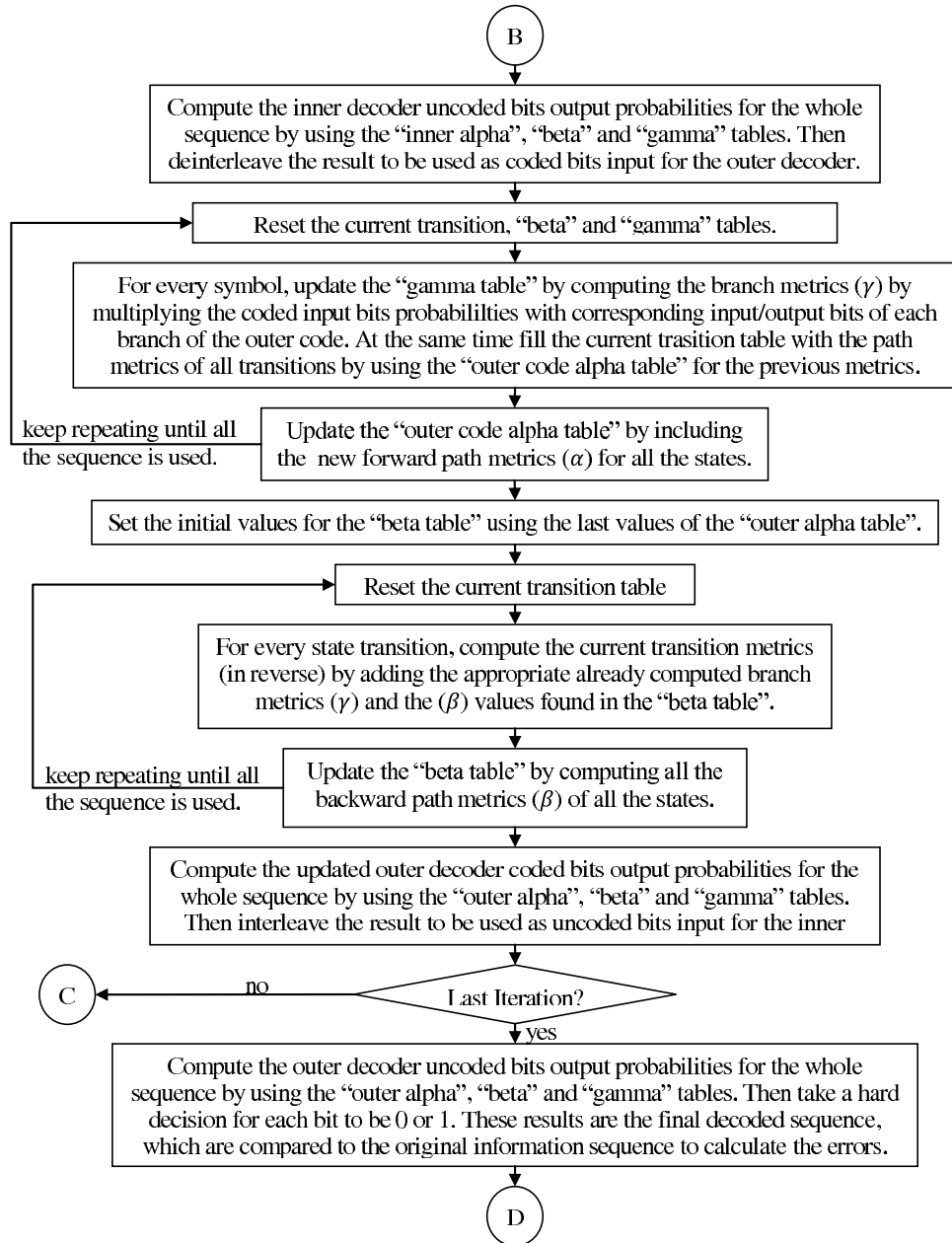
D

Figure A-7: Flowchart of the SCCC simulation code part 3

Appendix B: MATLAB Code

Theoretical Calculation of Regular Catastrophic Codes PER and BER

The following is the MATLAB code used to calculate $P[k]$ for each SNR. The output is then used to calculate the PER as $1 - P[k = 0]$ and the BER according to Equation 2.21.

The top section is the input part including the following parameters:

- n: The code packet length used

- *paths*: These are the exhaustive trellis states paths (of length 2) found in the catastrophic code. Each row represents one path. (We have 16 rows in this case)

- *ppaths*: The paths corresponding to a catastrophic event for each row in *paths*. The full list is found in Table 2.1

- *outputs*: These are 8 rows representing the 8 transitions found in the trellis. Each row contains the starting state, ending state and the corresponding output. Check Figure 2.7.

- *Lattice*: The lattice labeling used for 8-PSK modulation.

- *EbNo*: The SNRs wanted to calculate the probabilities for.

Description of the code is found in comments within the code.

```matlab
n=1000; %packet length
paths = [0 0 0;0 0 2;0 2 1;0 2 3;1 0 0;1 0 2;1 2 1;1 2 3;
        2 1 0;2 1 2;2 3 1;2 3 3;3 1 0;3 1 2;3 3 1;3 3 3];%Correct paths
ppaths = [0 2 3;0 2 1;0 0 2;0 0 0;1 2 3;1 2 1;1 0 2;1 0 0;
          2 3 3;2 3 1;2 1 2;2 1 0;3 3 3;3 3 1;3 1 2;3 1 0];%Parallel paths
outputs = [0 0 0;0 2 5;1 0 6;1 2 3;2 1 3;2 3 6;3 1 5;3 3 0];%Paths outputs
lpp = sqrt(2)/2;
Lattice = [1 0;lpp lpp;-lpp lpp;0 1;-lpp -lpp;-1 0;0 -1;lpp -lpp];%Lattice
Prbcats=0;
EbNo = [1:16];%SNR
%The following calculates sigma for all EbNo.
EsNo = EbNo;
No = 10.^(-EsNo/10);
sigma = sqrt(No/2);
%In the following for loop. All paths and ppaths are compared to find the
%corresponding trellis outputs, that are used to calculate the distance
```

```matlab
%and probability of error.
for i = 1:16 %for each path
    path = paths(i,:);   %get the path
    ppath = ppaths(i,:); %get the corresponding parallel event path
    pathout = [];        %the normal path outputs to be found
    ppathout = [];        %the catastrophic event path outputs to be found
    %The normal path outputs are found below
    for j = 1:length(path)-1
        from = path(j);
        to = path(j+1);
        for k = 1:8
            if from==outputs(k,1) & to==outputs(k,2)
                pathout = [pathout outputs(k,3)];
            end
        end
    end
    %The catastrophic event path outputs are found below
    for j = 1:length(ppath)-1
        from = ppath(j);
        to = ppath(j+1);
        for k = 1:8
            if from==outputs(k,1) & to==outputs(k,2)
                ppathout = [ppathout outputs(k,3)];
            end
        end
    end
    %By knowing the two outputs. The probability of the catastrophic event
    %can be computed for all SNRs.
    c = [Lattice(pathout(1)+1,:);Lattice(pathout(2)+1,:)];
    w = [Lattice(ppathout(1)+1,:);Lattice(ppathout(2)+1,:)];
    C = c(1,1)^2+c(1,2)^2+c(2,1)^2+c(2,2)^2;
    W = w(1,1)^2+w(1,2)^2+w(2,1)^2+w(2,2)^2;
    e = c - w;
    K = (C-W)/2;
    ur = sum(sum(e.*c));
    sr = sqrt(sum(sum(e.*e)).*(sigma.^2));
    %Prbcats is the catastrophic events probability
    Prbcats = 0.5*(1+erf((K-ur)./(sr*sqrt(2))))/16+Prbcats;
end
Prbcats;
%Prbsteperr adjusts the event probability to compute the probability of
%length 1 catastrophic event for easier calculations.
Prbsteperr = (1-sqrt(1-4.*Prbcats))./2
Probnumerr = []; %A table having in each row the probabilities for
                 %a specific number of catastrophic errors
                 %at each SNR
for k = 0:20%Number of catstrophic errors to happen within a packet
    Probnumerr = [Probnumerr;nchoosek(n,k).*(Prbsteperr.^k)
                                        .*((1-Prbsteperr).^(n-k))];
end
```

MATLAB Iterative Decoding of SCCC Simulation

In MATLAB, there is a simulink demo titled "Iterative Decoding of SCCC" that can be found in "Blocksets→Communications→Channel Coding→Iterative Decoding of SCCC". The following Figure B-1 shows the simulink block diagram. The parameters of the convolutional codes and APP decoders are changed to match the used codes as follows: (The following example parameters are used in the simulation results found in Figure 3.9

- The outer convolutional encoder

  Trellis structure: poly2trellis(3, [5 7])

- The inner convolutional encoder

  Trellis structure: poly2trellis([2 2], [3 1 2;3 2 3])

- The inner APP decoder

  Trellis structure: poly2trellis([2 2], [3 1 2;3 2 3])

  Algorithm: True APP

- The inner APP decoder

  Trellis structure: poly2trellis(3, [5 7])

  Algorithm: True APP

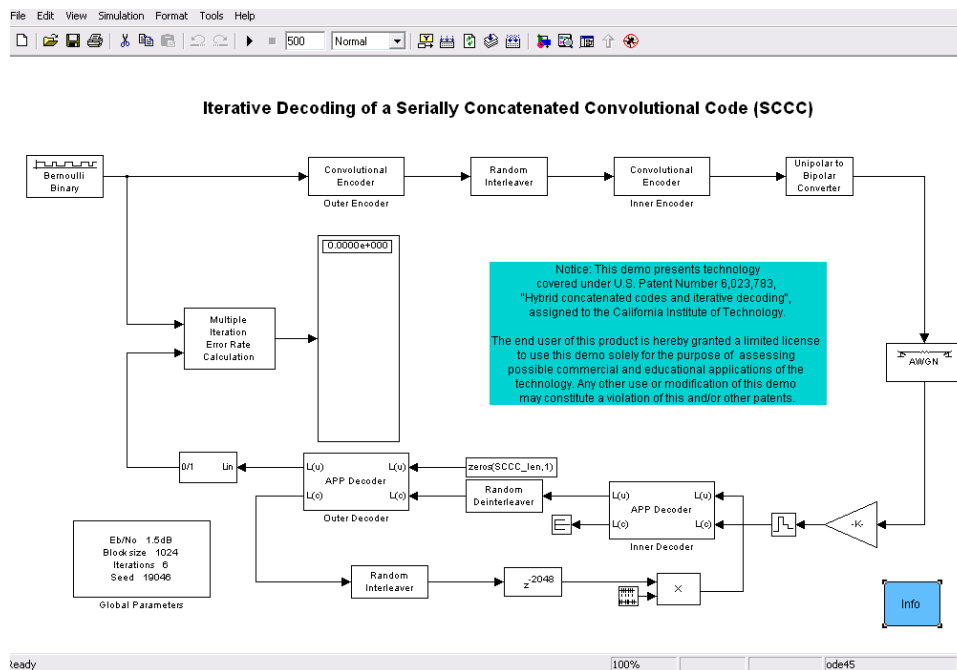- Global parameters Eb/No must also be changed to the desired values.

Figure B-1: MATLAB Simulink Demo "Iterative Decoding of SCCC"

Vita

Bandali Akkawi was born in April 1983, in Amman, Jordan. He enrolled in Princess Sumaya University for Technology, Jordan and earned a Bachelor of Science degree in electronics engineering in February 2005. He then joined Louisiana State University to pursue his graduate studies in communications engineering as a Master of Science candidate in 2006.