DESIGN OF A REDUCED COMPLEXITY RATELESS SPINAL DECODER

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MURAT TAŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JANUARY 2014

Approval of the thesis:

DESIGN OF A REDUCED COMPLEXITY RATELESS SPINAL DECODER

submitted by **MURAT TAŞ** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen ____________
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Gönül Turhan Sayan ____________
Head of Department, **Electrical and Electronics Engineering**

Assoc. Prof. Dr. Melek Diker Yücel ____________
Supervisor, **Electrical and Electronics Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Yalçın Tanık ____________
Electrical and Electronics Engineering Department, METU

Prof. Dr. Mete Severcan ____________
Electrical and Electronics Engineering Department, METU

Assoc. Prof. Dr. Melek Diker Yücel ____________
Electrical and Electronics Engineering Department, METU

Assoc. Prof. Dr. Ali Özgür Yılmaz ____________
Electrical and Electronics Engineering Department, METU

Sıdıka Bengür, M.Sc. ____________
Manager of HBT-PHSMM, ASELSAN Inc.

Date: January 10, 2014

ABSTRACT

DESIGN OF A REDUCED COMPLEXITY RATELESS SPINAL DECODER

Taş, Murat
M. Sc., Department of Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr. Melek Diker Yücel

January 2014, 94 pages

Wireless communication systems utilize several forward error correcting techniques to cope with changing channel conditions efficiently. One way is selecting from a list of coding/modulation schemes by considering the actual channel state. When this approach of choosing a fixed rate code is not applicable, one can use rateless codes whose rates are dynamically changing with respect to the changing channel conditions. A rateless encoder continues to send its codeword/packet to the destination unless the receiver sends an acknowledgement.

In this thesis, after investigating and evaluating the effects of the main parameters on the performance of the rateless spinal codes; our work focuses on making the spinal decoder algorithm more efficient in terms of the number of computations, while achieving the same rate values. We propose a modification that uses the initial information extracted from path metric distributions to decrease the computational burden at the initial steps of the 'bubble decoder' of Perry et al. The reduction offered by our modified algorithm is up to 70% of the original computational complexity at the cost of negligible rate losses.

Keywords: rateless codes, spinal codes, efficient decoder

ÖZ

DÜŞÜK KARMAŞIKLIKLI, ORANSIZ FİLİZ VEREN KOD ÇÖZÜCÜ TASARIMI

Taş, Murat
Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Melek Diker Yücel

Ocak 2014, 94 sayfa

Kablosuz haberleşme sistemleri, değişen kanal koşullarına uyum sağlamak için çeşitli hata düzeltme kodları kullanırlar. Bir yöntem, değişen kanal koşullarına göre kodlama/modülasyon seçenekleri arasından seçim yapmaktır. Sabit oranlı kodlardan birini seçmeyi gerektiren bu yöntemin uygulanamadığı durumlarda ise, oranı kanalın durumuna göre dinamik olarak değişebilen oransız kodlar kullanılabilir. Oransız kod kullanan haberleşme sistemlerinde, gönderici, paketi alıcı tarafından başarıyla alındığını öğrenene kadar göndermeye devam etmektedir.

Bu çalışma, oransız filiz veren kodların başarımını etkileyen ana parametrelerin incelenip değerlendirilmesinden sonra, kod çözme algoritmasının başarımını bozmadan işlem sayısını azaltmaya odaklanmaktadır. Perry ve arkadaşlarının 'köpük kod çözücü'süne önerdiğimiz değişiklik, algoritmanın metrik dağılımlarından çıkarılan ön bilgiyi kullanarak işlem yükünü daha ilk adımlarda düşürmektedir. Önerdiğimiz kod çözme algoritması, başarımdan küçük bir ödün vererek, oransız filiz veren kod çözücünün karmaşıklığını %70 oranında azaltmaktadır.

Anahtar Sözcükler: oransız kodlar, filiz veren kodlar, etkin kod çözücü

To my family

ACKNOWLEDGEMENTS

I would like to thank my supervisor Assoc. Prof. Dr. Melek Diker Yücel for her guidance and support throughout this work.

I would also like to thank my company ASELSAN for supporting my graduate studies.

I would like to thank my colleagues for their patience and valuable friendship.

Finally, I would like to thank my parents and my sister for everything.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ARQ	Automatic Repeat Request
AWGN	Additive White Gaussian Noise
FEC	Forward Error Correction
LDPC	Low Density Parity Check
LT	Luby Transform
MD4	Message Digest Algorithm 4
ML	Maximum Likelihood
QAM	Quadrature Amplitude Modulation
RNG	Random Number Generator
SHA	Secure Hash Algorithm
SNR	Signal to Noise Ratio

CHAPTER 1

INTRODUCTION

Communication systems aim to transfer information between communicating peers in an error-free manner. To achieve this goal, most digital communication systems use error correction techniques to combat with the effects of noise and interference at the receiver end. The error correction techniques are started to be developed after Claude Shannon's pioneering work that established the information theory field **[Shannon-1948]**. In his paper, Shannon proved that, for a communication channel, there exists a quantity C, called the capacity of the channel, such that for rates smaller than the capacity of the channel, one can achieve arbitrarily low probability of error in transmission by using powerful enough channel codes. Shannon proved that an additive white Gaussian channel of bandwidth W has capacity given as **[Proakis-1995]**

$$C = W \log_2(1 + \frac{P}{WN_0}), \tag{1.1}$$

where P denotes the average transmitted power and N_0 denotes the power spectral density of the noise.

The design of error correcting codes developed with block codes like, Hamming, **[Hamming-1950]**, BCH **[Bose-Chaudhuri-1960]** & **[Hocquenghem-1959]**, RS **[Reed-Solomon-1960]** codes, closely followed by tree and convolutional codes **[Elias-1955]**, **[Wozencraft-1957]**, **[Massey-1963]**, **[Viterbi-1967]**. Since then, more and more powerful channel codes were discovered and implemented in practical communication systems. Two of the most important error correction codes discovered are the LDPC codes of Gallager **[Gallager-1962]** and the Turbo codes **[Berrou-Glavieux-Thitimajshima-1993]**. These two channel codes are shown to operate at rates very close to the channel capacity for AWGN channels. They both have efficient decoding algorithms; hence they are efficiently implemented in practice, including a variety of communication standards.

The error correction codes mentioned above are fixed-rate codes, that is, they take *k* input symbols and generate *n* coded symbols with a fixed-rate of *n*/*k*. The rate of the fixed-rate codes is determined according to the communication channel's conditions either at the design time or at the instant of communication. The communication protocols that use the latter technique should have information about the channel conditions continuously at the transmitter side, so that the transmitter can adapt its coding rate to the environment conditions.

1.1. Rateless Codes

A rateless code is defined as a channel code such that the higher rate codes are prefixes of the lower rate codes **[Erez-Trott-Wornell-2012]**. The transmitter that uses a rateless encoder generates encoded symbols until the receivers acknowledge that the transmission is successful or a timeout for transmission time is reached. This makes the rateless transmission's coding rate subject to change at each transmission. In other words, a higher code rate is possible when the channel conditions are good and a lower rate when the channel conditions are bad. Since the coding rate is not fixed ahead of transmission, the coding process adapts itself nearly perfect to the changing channel conditions.

The first rateless codes that are developed, are designed specifically for erasure channel models. The most important two of them, which are discussed also in Chapter 2, are LT (Luby Transform) **[Luby-2002]** and Raptor **[Shokrollahi-2006]** codes, which are also known as fountain codes. The online codes are also rateless codes developed at the same time. **[Maymounkov-2002]**. The encoder of fountain codes is like a metaphorical fountain that generates an infinite number of encoded symbols **[MacKay-2005]**. This way, the coding rate is determined according to the channel conditions, making the fountain codes rateless.

The LT codes use a specially designed probability distribution to choose the degree of an encoded packet that shows how many packets to exclusive-or with each other. This distribution makes encoding and decoding of LT codes practical to implement with large code block sizes. The Raptor code is the concatenation of a weak LT code with a precoder to compensate for the deficiencies of the weakened LT code. The Raptor codes have lower encoding and decoding complexity as compared to LT codes.

1.2. Rateless Spinal Codes

A recently proposed rateless code is the rateless spinal code **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**. The spinal codes generate encoding symbols or modulation symbols by sequentially applying a hash function to smaller portions of the message to be transmitted. Applying the hash function creates encoded symbols very different from each other even when the two messages differ only in a single bit. This in turn makes spinal codes robust to noise and interference effects. In **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**, it is shown through simulations that the spinal decoders are better performing than the Raptor, LDPC and Strider codes at a large range of SNR values **[Gudipati-Katti-2011]**. In **[Balakrishnan-Iannucci-Perry-Shah-2012]**, it is also shown that spinal codes achieve Shannon capacity in AWGN and binary symmetric channels with a polynomial-time encoder and decoder.

The high rate values achieved with spinal codes can be observed in Figure 1.1 which is reproduced from **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012].** The spinal codes achieve close to capacity rate values for a large range of SNR values.

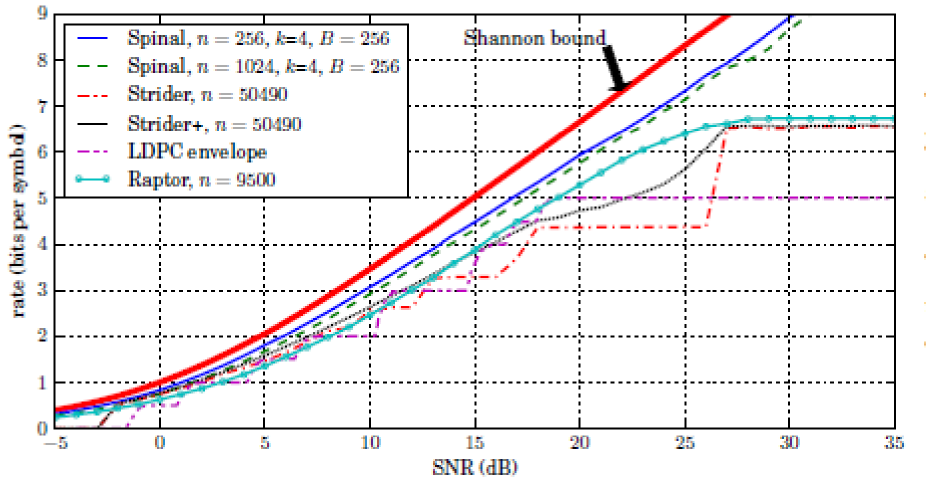


Figure 1.1 –Rate achieved with spinal codes**[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**

Using a hash function to generate encoded symbols makes the decoding of spinal codes harder by inverting the encoder structure. Perry et al. propose an approximate maximum-likelihood decoder, called bubble-decoder **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**, to efficiently decode spinal codes. The bubble decoder traverses the tree of possible transmitted symbols in a breadth-first manner. While traversing the tree, the decoder prunes the least likely paths so that the decoding operation can be completed with reasonable complexity instead of the exponential complexity of the maximum-likelihood decoder.

As mentioned in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**, spinal code's decoder performance gets better when the decoder is more like the maximum-likelihood decoder. This can be realized if the decoder prunes fewer paths at every symbol interval, which in turn increases the complexity and the processing load of the decoder. Since there is a physical limit on the computational complexity of a practical decoder, one cannot continuously increase the number of paths to be maintained at the decoder to get better performance. On the other hand, the speed of the spinal decoder is inversely affected by its computational load, the more the decoder prunes less likely paths, the higher decoding speeds can be achieved.

1.3. Aim and Organization of the Thesis

All practical decoders are limited by the hardware cost, energy expenditure and complexity; hence, more competent ways of decoding spinal codes are highly desirable. In this work, we first investigate the effects of using different values for some parameters of the spinal codes. After that we propose an efficient decoder that improves the performance of bubble decoder by maintaining the possible paths only when it is necessary. Our work focuses on making the spinal decoder algorithm more efficient in terms of computation, while achieving the same rate values for a range of SNR values.

The organization of the thesis is as follows.

Chapter 2 reviews the basics of rateless channel coding and gives brief information about two most important rateless codes, namely the LT and Raptor codes.

In Chapter 3, we review rateless spinal codes. The encoder and decoder of spinal codes are explained in detail. We also give brief information about the puncturing scheme proposed in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** and the hash function used in the encoder and decoder.

In Chapter 4, we propose an algorithm to efficiently decode the spinal codes. Our algorithm builds up on the bubble decoder and saves computational expenditure up to 70% as shown in the following chapter by simulation results

Chapter 5 gives the simulation results that we obtain by investigating the parameters of the spinal encoder and decoder. We also present the results of applying our decoder algorithm proposed in Chapter 4, and compare them with those of the bubble decoder.

In Chapter 6, we comment on the results and discuss future work.

CHAPTER 2

RATELESS CODES

In this chapter, we present a general overview of the concept of rateless channel coding by discussing important rateless codes. In Section 2.1, we review the traditional way of channel coding and its drawbacks in general. In Section 2.2, we introduce the rateless coding concept and point out differences from fixed-rate codes. Section 2.3 covers an important type of rateless codes, called the Luby Transform (LT) code, for which we give the encoding and decoding principles. In Section 2.4 we discuss the Raptor code, which is an improvement over LT codes in terms of the encoder and decoder complexity.

2.1. Fixed-Rate Channel Codes

Traditional channel codes that are found to reach rates very close to Shannon capacity, such as Turbo or Low Density Parity Check (LDPC) codes are examples of fixed rate forward error correction (FEC) codes; i.e., their encoders generate a sequence of n symbols given a sequence of k input symbols. Thus, they have fixed code rates of k/n, which means they have fixed number of additional error correction symbols per source symbol. These codes are shown to operate well when the transmitter and the receiver know the channel conditions. While it is a reasonable assumption in some cases that the communicating peers know the state of the channel, in some situations it can be hard or even impossible to track the channel conditions. This complexity of getting channel information opens up an application area for rateless codes.

To reach high rates in time-varying channels, transmitter-receiver pairs that use traditional fixed-rate codes, should use a reactive approach to choose an encoding method, coding rate and modulation type that is suitable for the actual channel. This reactive approach includes the receiver's sensing the channel state and giving feedback to the transmitter. The transmitter then selects a suitable coding/rate/modulation choice to reach the best possible rate. The reactive method of sensing the channel and sending feedback to the transmitter has two major drawbacks.

First, for the receiver to sense the channel state, the transmitter should send pilot symbols, whose values are known in advance by the receiver. Since some of the channel time is used for sending pilot symbols, this naturally decreases the achieved rate **[Goldsmith-2004]**. Sending feedback from the receiver to the transmitter also uses the channel resources and prevents the system from achieving the maximum possible rate.

Also, in fast time-varying channels or when the estimation of channel state is not of enough quality, the estimated channel state can be different from the actual condition, leading the transmitter to use a suboptimum encoding type/code rate or modulation format. Consequently, the rate achieved by the transmitter will be suboptimum.

The second drawback of using a reactive approach for transmitter parameter selection is that this approach will result in encoders and decoders having large number of alternatives for coding and modulation types, which increases complexity in both the transmitter and the receiver. Also, the need for choosing among different coding/modulation parameters forces the design of related protocols, which also increases the complexity of transmitters and receivers.

2.2. Overview of Rateless Coding

Another approach to reach rates that are close to capacity on rapidly varying channel conditions, or when the reactive approach cannot be applied, is to use rateless codes. A rateless code is defined to be a code such that, the higher rate code is the prefix of the lower rate code **[Erez-Trott-Wornell-2012]**. When the communicating parties are using rateless codes, they are working in a different manner than when they are using fixed-rate codes. The rateless transmitter sends the coded message until the decoder successfully decodes it, or some timeout is reached. This is different from fixed-rate encoding operation where the transmitter should retransmit the message if the receiver could not get the message.

In a receiver using rateless codes, the decoder decodes the message when the rate of the transmission drops below the Shannon capacity value, and it possibly replies with an acknowledgement message. In other words, transmitter initially sends the coded symbols at a rate higher than the channel can carry. By this operation, with each transmitted symbol (or packet), the transmitter sends the information to the receiver incrementally, which resembles the Type II Hybrid ARQ approach, where the parity bits of the encoded message are sent to the receiver incrementally **[Lott-Milenkovic-Soljanin-2007]**. In this concept, the rateless codes can be seen as filling an imaginary bucket with water drops from a fountain (Hence the name fountain codes) **[Mackay-2005]**. The rateless transmission continues until either the transmitter gives up transmitting or the receiver announces the successful decoding of the message.

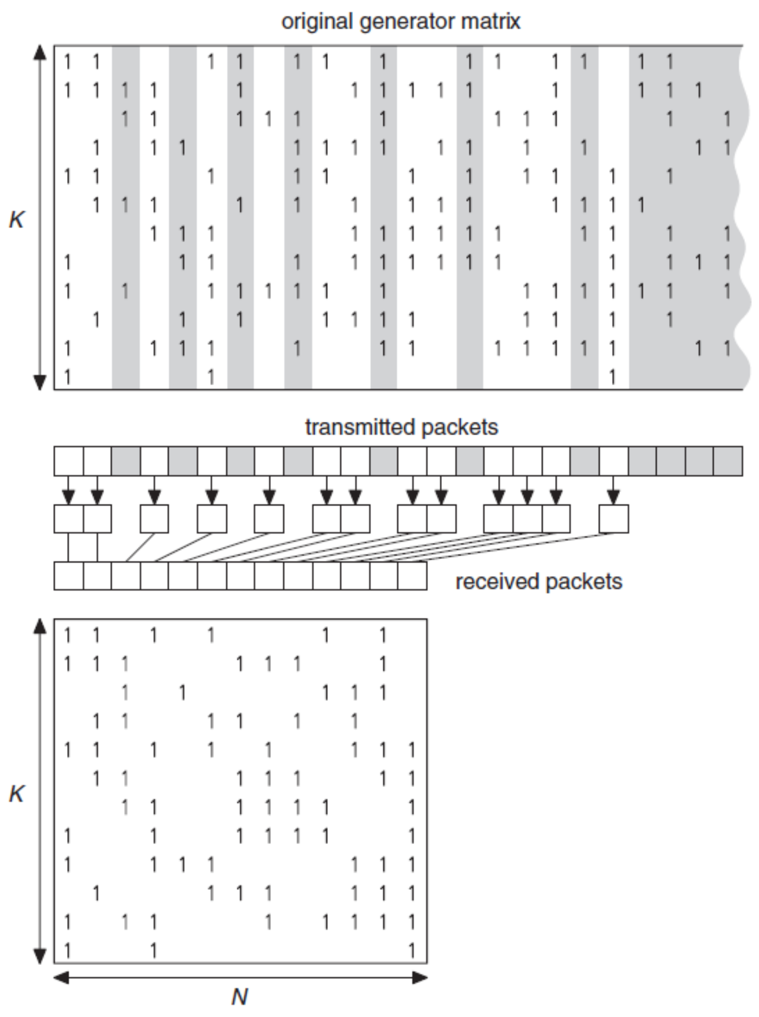


Figure 2.1 – Random linear coding **[Mackay-2005]** (with lost packets shown in gray).

As an example to rateless codes, random linear coding approach is shown in Figure 2.1. At each transmission time, a packet is transmitted which is created according to the following equation

$$c_n = \sum_{k=1}^{K} s_k G_{kn} \tag{2.1}$$

where c_n is the transmitted packet, s_k is the k^{th} message packet and G_{kn} is the generator matrix of the random linear encoder. The transmitted packet consists of the modulo-2 bitwise sum of each message sub-packet where the G_{kn} value is equal to 1. The transmitter can create the generator matrix on the fly or prior to the transmission. The receiver should use the same generator matrix to successfully decode the message, so either the G_{kn} vector is sent at each transmission or the seed value of a random number generator is sent to the receiver before the transmission of packets. The randomness in the coding process is because of the random generation of the code generator matrix.

In case of packet loss at any point of transmission of the message, the transmitter and the receiver are unaware of this and the transmission continues. After getting N packets from the transmitter, the receiver decodes the message by inverting the K-by-N matrix, where N is equal to or larger than K. This is possible when the K-by-N matrix contains a K-by-K sub-matrix, which is invertible. The probability of failed decoding is exponentially decreasing with the number of packets sent in excess, namely $N-K$. The encoding and decoding costs of random linear coding led the researchers invent new coding methods, one of which is the LT codes **[Luby-2002]** discussed below.

2.3. LT Codes

The LT (Luby Transform) codes are the first practical rateless codes, and they are able to generate symbols (packets) endlessly. The encoding principle is similar to the random linear coding method, the difference being in the generator matrix. The generator matrix is sparse; i.e., the number of ones in each column of the generator matrix is significantly smaller than the value column length, K.

The encoding algorithm of the LT encoder can be summarized as follows:

- Randomly choose the degree, d, of the packet that will be transmitted, from the designed degree distribution,
- Uniformly select d packets from the message and calculate modulo-2 bitwise sum of them.

The designed degree distribution is the key component in LT codes. In **[Luby-2002]**, Luby proposed a distribution called Robust Soliton Distribution, which makes the code stable, and be able to be decoded with a simple decoding algorithm. The decoding algorithm is described as follows:

- Find a transmitted packet that is a function of only one source packet,
- Add the packet to all the coded packets that have the same source packet's contribution,
- Repeat the procedure until all source packets are found.

The LT codes have encoder and decoder complexity scaling as $K\, log\, K$ where K is the number of packets in the source message. In the following section, we briefly review Raptor codes **[Shokrollahi-2006]**, which have linear encoder and decoder complexity while having superior performance to the LT codes.

2.4. Raptor Codes

Raptor codes are based on LT codes, instead of using them alone, a Raptor code concatenates the LT code with an outer fixed-rate pre-code. The Raptor codes use a sparse generator matrix like the LT codes, but the degree of each column is constant. This leads Raptor codes to have linear encoding and decoding complexity. As the rateless code that is used at the inner part of the Raptor code is somewhat weaker than an ordinary LT code, some of the source packets of the transmitted message are not included in any of the transmitted packets. To cope with this deficiency, in **[Shokrollahi-2006]**, an irregular low-density parity-check (LDPC) code is used as the outer code. An example for the encoding of Raptor codes is shown in Figure 2.2. The gray-colored circles in Figure 2.2 represent the symbols that are not included at the output of the inner LT code. One can compensate for this effect by using a sufficiently powerful outer code.

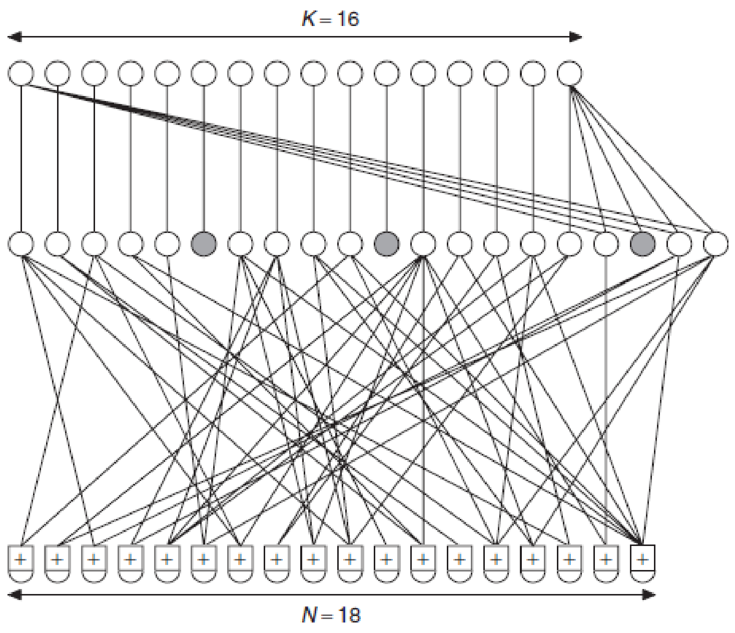


Figure 2.2 – Raptor encoding **[Mackay-2005]**

The decoding of the Raptor codes is the serial decoding of the outer and inner codes, respectively, so the decoder for LT codes and the decoder for the outer code can be used. For the inner part, one can use the algorithm as described in Section 2.3. In **[Shokrollahi-Luby-2009]**, authors describe another decoding algorithm, called inactivation decoding, which combines Gaussian elimination with the belief-propagation algorithm.

In addition to erasure channels, Raptor codes' performance on AWGN and Binary Symmetric channels are investigated in literature. In **[Palanki-Yedidia-2004]** it is shown through simulations that Raptor codes outperform LT codes on noisy channels.

In this chapter we reviewed the rateless coding concept, discussed the differences from the fixed-rate channel codes and briefly introduced two important classes of rateless codes. Asymptotically, the complexity of the LT codes varies with $K \, log \, K$ and that of the Raptor codes is linear in K.

CHAPTER 3

RATELESS SPINAL CODES

In this chapter, we present rateless spinal codes, which is a new class of rateless codes that achieve very high rates at a large range of channel conditions. Spinal codes are non-linear; they have an efficient decoder and reach rates very close to Shannon capacity.

Spinal encoding algorithm encodes a message by applying a hash function to the non-overlapping portions of the message bits sequentially, producing a spine value - output of the hash function - for each non-overlapping message part. It uses this spine value to form the symbols to be transmitted by means of a constellation mapping function. The hash function makes the spine values pseudo-random, which makes the code robust to noise.

Since the encoding function of the spinal encoder uses a hash function, it is not easily invertible. The proposed decoder in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** makes a breadth first search over a tree, replaying the encoder for a portion of all possible messages that can be sent. It then compares the replayed (possible) messages with the received symbols and keeps the most probable (i.e. most likely, with the lower path metrics) messages at each symbol time. The replaying of the encoder only for a portion of the all-possible messages is a practical solution to cope with the exponential complexity of the ML decoder. The proposed decoder approximates the ML decoder as close as desired by setting its parameters.

In Section 3.1 we review the details of the spinal encoder. Section 3.2 discusses the details of the spinal decoder that is presented in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**, together with the channel model. In Section 3.3, puncturing of spinal codes, which is very important for practical applications, is explained. In Section 3.4, we discuss the choice of the hash function and the effect it has on the performance of spinal codes. Finally, in Section 3.5 we give some definitions about spinal codes.

3.1. Spinal Encoder

The spinal encoder can be used in two modes of operation. The transmitter can use it for generating coded bits from source bits and then apply a known constellation function, e.g. 4-QAM, which produces symbols. Or the spinal encoder can generate the modulated symbols to be transmitted from the source bits directly. In this section we will present the spinal encoder working in the latter operation mode **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**, shown in Figure 3.1.

The encoder has three sub-blocks, namely Message Splitter, Spine Generator and Constellation Mapper. It gets the input message, represented by *M* (shown by 1 in Figure 3.1) and produces a sequence of complex symbols (shown by 4 in Figure 3.1) as long as the transmission continues.

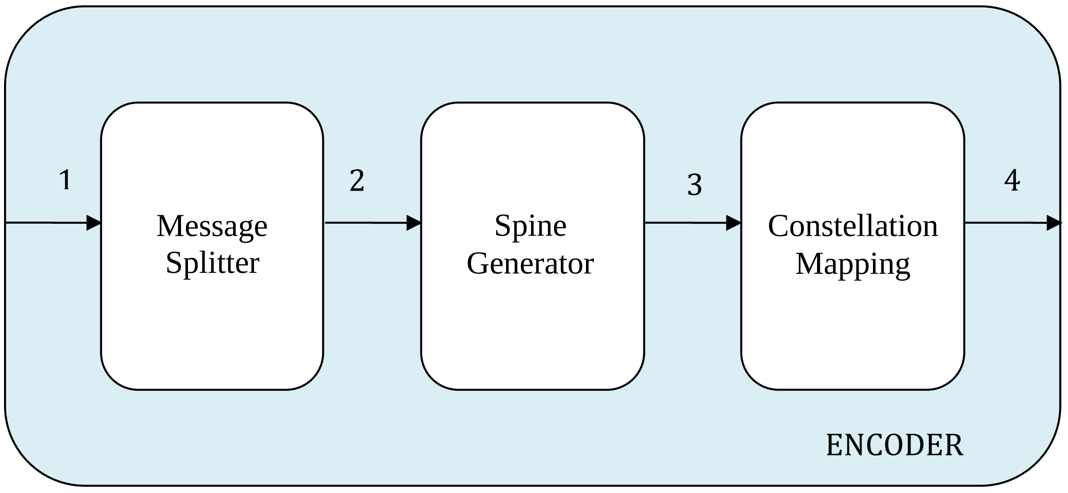


Figure 3.1 – Inner structure of the spinal encoder

3.1.1. Message Splitter

Let the overall message to be transmitted as being composed of *n* bits. Note that since the spinal decoder needs the cyclic redundancy check (CRC) of the message to check its correctness, this *n*-bit message should have CRC data inside. The encoder splits *n*-bit message into *k*-bit chunks, where *k* is a parameter of the encoder and decoder. This step is shown in Figure 3.1 as the message splitting part. The split data shown as “2” is the output of the message splitting process, the vector of *k*-bit chunks which has a length of *n*/*k.* As we will discuss in the following sections, *k* is a parameter that affects the complexity of the spinal decoder, so the value of *k* is generally selected as smaller than 8.

3.1.2. Spine Generator

The second step is the spine generation, which uses each *k*-bit block to generate the spine value - the output of the hash function - to produce the symbol to be transmitted after the constellation-mapping step. The spine generator generates the output by applying a hash function to the corresponding *k*-bit block and the previous spine value. The formula for spine generation is

$$s_0 = \mathbf{0} \qquad (3.1)$$
$$s_i = h(s_{i-1}, m_i) \text{ for } i = 1, \ldots, n/k$$

where s_i is the *v*-bit spine generated for the i^{th} *k*-bit chunk, and *h*(.) is the hash function used to generate the spine.

The hash function takes two arguments, the first one is the spine value calculated at the previous step, and the second is the *k*-bit chunk, which is represented by m_i.

$$h : \{0,1\}^v \times \{0,1\}^k \rightarrow \{0,1\}^v \qquad (3.2)$$

Note that the spine in the first step is selected to be the all zero vector **0**. The initial spine value, s_0, can be selected as any other vector of length *v*, provided that the decoder also knows this initial value.

The spine generator produces spine values for each *k*-bit message segment, therefore a total of *n*/*k* spine values for the entire message are to be transmitted. The vector of *n*/*k* spine values is shown as “3” in the Figure 3.1. The spine that is generated at each symbol time is used for symbol generation at the constellation-mapping step. Since the spine generated at any step is dependent on the spine of the previous step, it can be concluded that a spine value at any step depends on the message portion up to that instant; i.e., the spine s_i depends on the message bits 1 to *ik*. This property that the spine value’s being dependent on every bit up to that instant, gives the spinal encoder a large memory, which can be thought of as a large constraint length in terms of convolutional coding.

The use of the hash function makes the spine values generated for two messages very different from each other, even when there is only one bit difference. So the distance between codewords is made large, which results in a code that is very resilient to noise and errors especially in bad channel conditions. The properties of the hash functions that result in this code behavior are further investigated and explained in Section 3.4.

3.1.3. Constellation-Mapping

After the spine generation phase is over, at the constellation-mapping step, for each pass, the encoder generates a symbol using the spine. At the beginning of the transmission, the constellation-mapper uses the spine value to generate the mapped symbols. If the spine length *v* is not long enough, the constellation mapper sub-block may apply a random number generator (RNG) function to the spine value during the transmission to generate new random bits so that it can map new symbols. The RNG function takes the hash output (spine) as the input parameter and generates a random number. As more symbols are needed to be transmitted, the RNG function takes the previously generated random number as its input and generates a new random number. In other words, the RNG function generates new random numbers recursively and starts with the spine value as its initial input.

After generating the RNG output, the constellation-mapper maps the least significant $2c$ bits of the RNG output as a symbol. The constellation mapping is done by using each c bit-segment for one of the in-phase (I) and quadrature (Q) channels. This mapping can be performed uniformly or using a truncated Gaussian cdf as explained in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**.

The encoder makes this operation for each pass, where a "pass" is defined as the collection of n/k symbols. A constellation symbol is called x_{ij}, where the first subscript i refers to the index of message block m_i or the corresponding the spine s_i to be sent; and the second index j denotes the pass number. So, $2c$ bits of the RNG output picks up a symbol x_{ij} from the constellation map, for i=1,…, n/k in each pass. One pass therefore gives information about all n bits of the message to be transmitted. The spinal encoder details of generating symbols for successive k-bit chunks and for successive passes can be seen in Figure 3.2.

Since spinal codes are rateless, the transmission probably will not finish after one pass, in fact it continues until the receiver acknowledges the transmitter or the transmitter gives up sending. So, the encoder transmits symbols x_{i1} for i=1,…, n/k in the first pass, and continues with the 2nd pass symbols x_{i2} for i=1,…, n/k, followed by the 3rd pass symbols x_{i3} for i=1,…, n/k, etc; hence the rate of transmission falls at low SNR's as the number of passes increases. The generation of new symbols for each new pass requires the constellation-mapping sub-block to work recursively like the hash function does.

The uniform constellation mapping results in a 2^{2c}-QAM constellation (for example c=2 results in 16-QAM constellation). As one can see, selecting a larger value for the parameter c results in a denser constellation, which leads the transmitter to reach higher rates when the channel conditions are good. The best value to use for the parameter c can be determined by simulations of different values. Some findings about the parameter c as well as the other encoder/decoder parameters are given in Chapter 5.

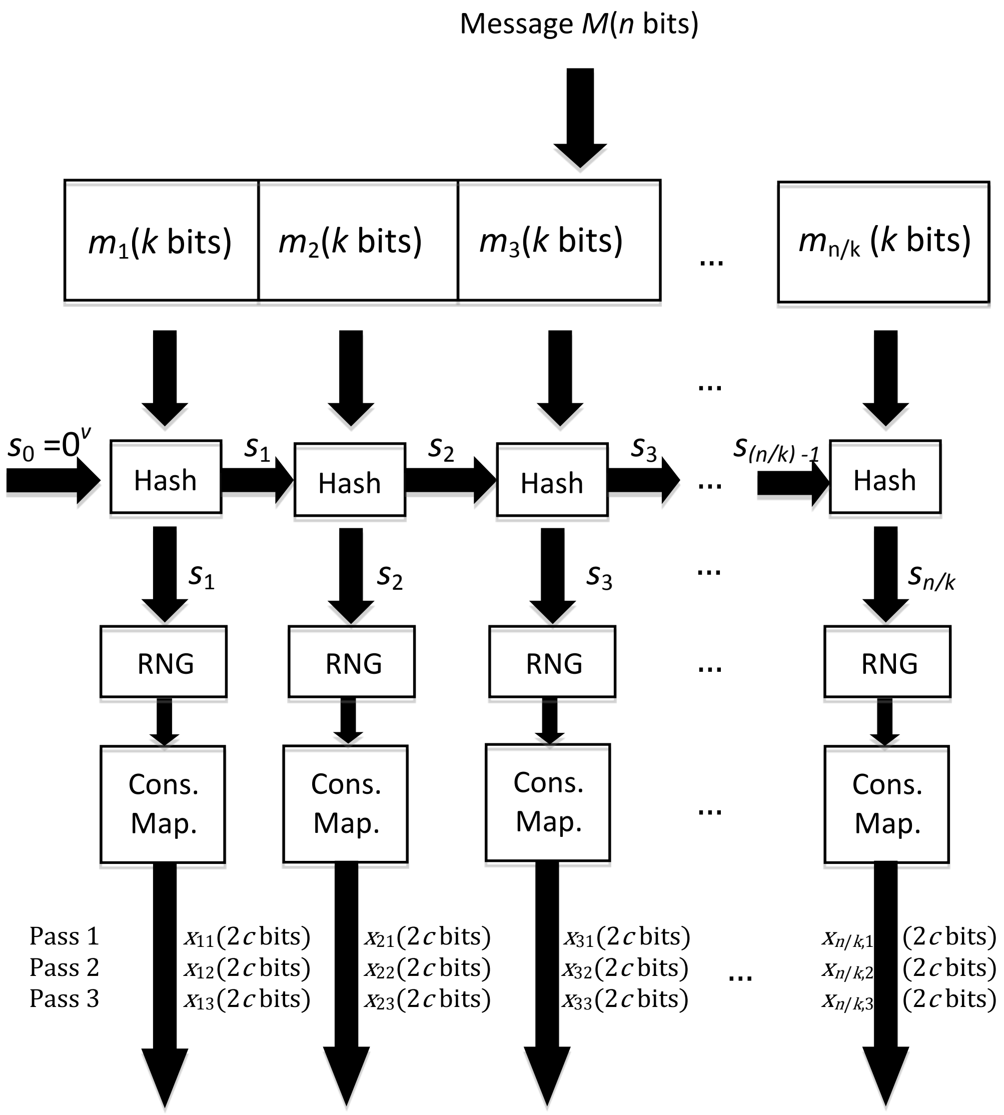


Figure 3.2 – Spinal encoder's generation of symbols in successive passes

3.2. Spinal Decoder

In this section, we review the decoding principles of spinal codes, as presented in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**. Perry et.al call the spinal decoder a “bubble decoder”, using a generalized version of the M-algorithm **[Anderson-Mohan-1984]**. The decoder is said to efficiently decode the encoded symbols by replaying the encoder’s job for a subset of all possible messages.
The decoder mainly replays the encoder’s job for each possible k-bit vector as a new symbol arrives. Since doing that computation is infeasible for 2^n possible messages, the decoder simply maintains B best (i.e., the smallest metric) paths, which are the ancestors of the candidate paths after each symbol transmission. The decoder just does not care the paths that have path metrics larger than the smallest B paths. In this manner the decoder proposed can be viewed as an approximate maximum likelihood decoder. Performance arbitrarily close to that of an ML decoder can be achieved by increasing the value of B.

Although better performance can be achieved by increasing the paths to be maintained at each symbol transmission step, this will make the decoder increasingly complex. Since in practice, the receiver to be designed has complexity and power limitations, we only consider linear-time spinal decoders in this work. In **[Balakrishnan-Iannucci-Perry-Shah-2012]**, Balakrishnan et.al proved that a polynomial-time spinal decoder achieves the capacity on AWGN and binary symmetric channels.

The maximum-likelihood decoder finds the most likely message $\widehat{M}$ among all possible messages M', given the received vector $\bar{y}$. The maximum-likelihood decoder for AWGN channels is the closest candidate symbol vector $\bar{x}(M')$, to the received vector $\bar{y}$:

$$\widehat{M} = argmin_{M' \in \{0,1\}^n} |\bar{y} - \bar{x}(M')|^2 \qquad (3.3)$$

One can write the argument of (3.3) as the sum of each received symbols as follows: **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**

$$|\bar{y} - \bar{x}(M')|^2 = \sum_{i=1}^{n/k} |\bar{y_i} - \bar{x_i}(s_i)|^2 \qquad (3.4)$$

The spinal decoder can make this calculation cumulatively at each pass, thanks to the spinal encoding principle.
So, the ML decoder calculates (3.4) for all possible messages and selects the minimum. However, computation of (3.4) 2^n many times is not feasible in many practical cases.
Instead of keeping all possible paths, the bubble decoder maintains a fixed-number of paths that are most probable up to that instant. And with reception of every encoded symbol, it creates new paths that are children of the current possible ones. The possible paths form a beam, and the parameter *B* is called the beam-width.

Simulation results and hardware experiments in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** show that a moderate *B* value of 256 (for *n*=256, *k*=4, *c*=6) can reach very high rates in AWGN channels. The decoder also accepts another parameter, represented by *d*, which shows how deep up the tree the decoder will traverse when pruning the possible paths. The bubble decoder algorithm is given as follows:

Algorithm of the Bubble Decoder

```
input: $B$, $d$=1, $c$, $n$, $k$
output: Message $M$
Candidates ← [ ]

    • Receive symbol
    • Expand the tree by calculating all possible path metrics
    • Prune candidates according to value of $B$
    • If $n/k$ symbols are received, check CRC of the candidates
        o If the CRC checks for only one candidate, the decoding is
          successful, output the message
        o Else wait for next symbol
    • Else If not enough symbols received, wait for next symbol
```

Operation of the spinal decoder is shown in Figure 3.3. The parameters are chosen as B=2, d=1, k=1 and n=3 for illustration purposes. For the sake of simplicity, the parameter c and the generation of constellation symbols is not considered in Figure 3.3. Notice that each transmitted symbol carries 1 bit of information since k=1. Upon receiving the first symbol, the decoder calculates path metrics for each of the possible paths. After that, since B=2 in this example, both paths are recorded in memory and the decoder waits for the next symbol.

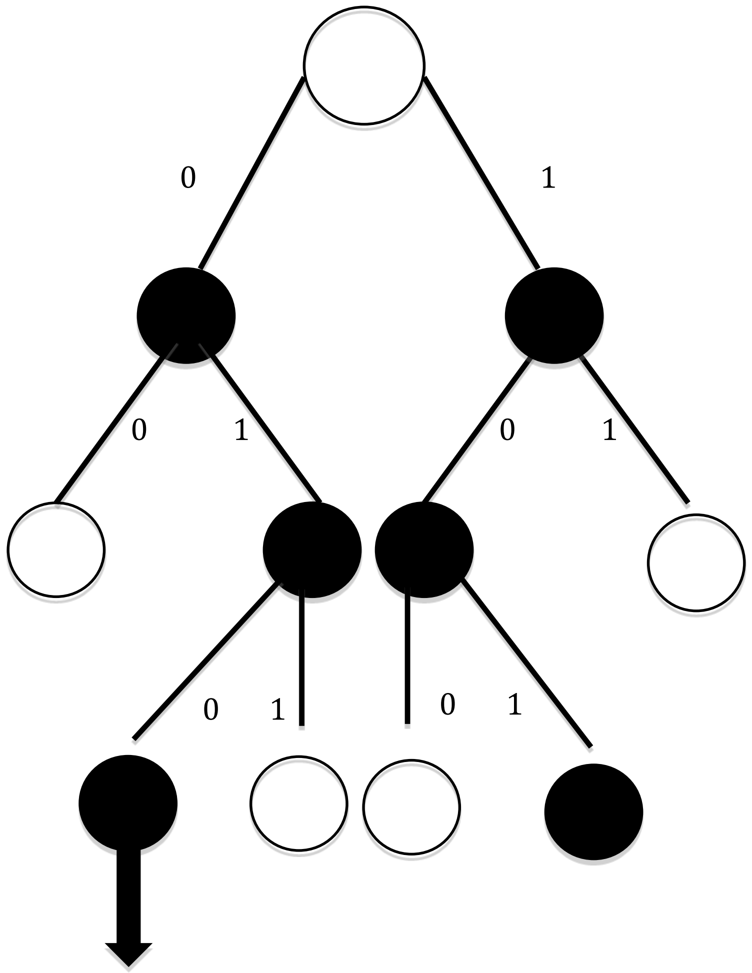


Figure 3.3 – Operation of spinal bubble decoder

Similarly, after getting the second symbol, the decoder calculates all possible metrics, this time generating a total of 4 candidates. Since $B = 2$, the decoder prunes the 2 least likely ones and keeps the 2 paths that have lower path metric values. After getting the third symbol, the decoder selects the best path as its estimation of transmitted message, shown by the black arrowed candidate in Figure 3.3.

Note that in practical use of spinal codes and also in our simulations, the encoded data has CRC information inside it. The decoder uses this information to check whether the message is acceptable or not. If it is not acceptable, a successive pass starts by computing the new metrics in a new tree. In this manner, the computation of the new metrics can be performed cumulatively so that the decoder makes use of previously received symbols.

3.3. Puncturing

The encoder and decoder pair defined above permits a maximum rate of *k* bits/symbol, which is achieved when the message is successfully decoded after the first pass. In fact, only rates of *k* / (# of passes) can be achieved in this setting. This can prevent the transmission to reach high rates when SNR is very high. For example, when *k* = 4, the maximum rate is 4 bits/symbol, which is inefficient above 12 dB SNR where the Shannon capacity (calculated using the formula log(1+SNR)) is approximately 4.074 bits/symbol.

One can achieve higher rates by simply increasing the value of the parameter *k*. But since the complexity of the decoder is exponential in the parameter *k*, rather than increasing the value of *k*, one can apply a puncturing scheme as proposed in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** to achieve higher rates than *k* bits/symbol.

The puncturing scheme defined in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** is shown in Figure 3.4 for *n*/*k*=32. Each pass (which is the collection of *n*/*k* symbols) is divided into eight subpasses. In each subpass, only the symbols corresponding to dark circles are transmitted.

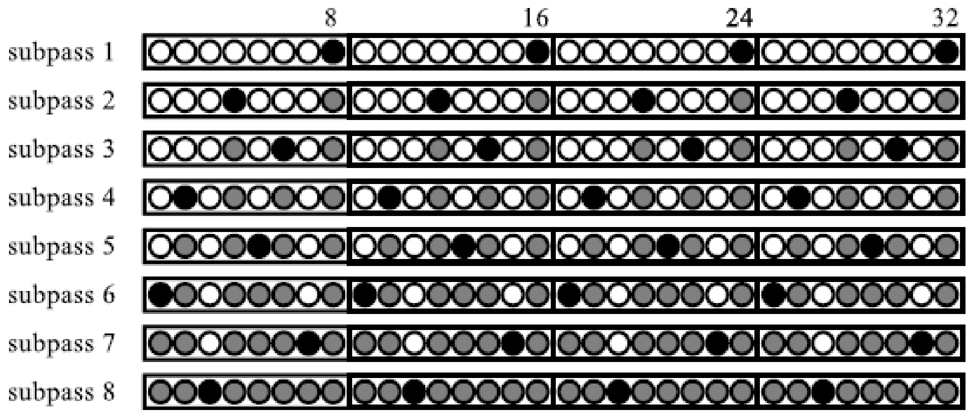


Figure 3.4 – Puncturing schedule offered in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**

The decoder algorithm works in the same manner as the “no puncturing” scheme. For each symbol time, if there is a received symbol, the decoder calculates all possible spines using the paths in the candidate list, and for each possible message it calculates the distance between the received and the generated symbols. Following this step, the decoder prunes the candidate list down to size *B*. The decoder maintains the path metrics of each possible path in *B*, with the corresponding spine values of each one.

If there is no received symbol at the corresponding symbol time (for example the symbols 1 through 7 in Subpass 1 of Pass 1), the decoder only calculates the spines for the possible messages and prunes the candidate list. In the case of ties of the path metrics of different paths, the choice of pruning is made arbitrarily. So the decoder does not make any computation for the symbols not transmitted when using the puncturing scheme.

Using the puncturing scheme, one can achieve both higher maximum rates and rates with finer resolution. Since the decoder tries to decode the message after each subpass, the maximum rate that can be achieved is as high as $8k$. For example, if the message is decoded after the first subpass of the first pass, then the rate will be 32 bits/symbol.

The decoder algorithm does not change when puncturing is used. The decoder expands the *B* paths to $B2^k$ paths as usual, and for any missing symbol, it assumes the path metric of each path is the same as the last step. Using a larger *B* value makes the puncturing scheme more efficient in terms of rate achieved on the channel.

3.4. Hash Function

The spinal encoder uses a hash function to generate spine values, which RNG makes encoded symbols out of. The purpose of the hash function is to magnify the distance between messages, which are close in terms of the Hamming distance. Thus, the hash function to be used should be a function with good mixing properties; i.e., a 1-bit difference in two messages should make approximately half of the bits of the spine value different from each other after that bit's position.

Since the hash function is applied to all k-bit chunks of the message sequentially, the spine value at any point is dependent of the k-bit chunks up to that point. This gives spinal codes a large memory or "constraint length" in terms of convolutional coding concepts.

In **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**, Lookup3 and One-At-A-Time hash functions are used in decoder and encoder. In this work, we use Lookup3 hash function unless otherwise stated. In Chapter 5, where we look at some other cryptographic-strength hash functions, we will compare the performance of Lookup3 hash function with them.

3.5. List of Symbols Used for Spinal Codes

In this section, we give a list of symbols that we use for spinal codes. The list can be used as a quick reference.

M = Message to be transmitted
n = Size of the message to be transmitted, in bits
k = Size of the chunk of each part that is used to generate a spine, in bits
s_i = Spine generated at the output of the hash function for the i^{th} k-bit chunk
v = Size in bits of each spine value
$2c$ = Number of bits to be used when generating a symbol
n/k = Number of symbols at each pass
Pass = Time needed to send/receive n/k symbols, and also the collection of n/k symbols
Subpass = $1/8^{\text{th}}$ of a pass
B = Size of the beam-width that shows how many possible paths are maintained at each step

CHAPTER 4

THE ALGORITHM FOR EFFICIENT DECODING OF SPINAL CODES

While rateless spinal codes reach a high percentage of Shannon channel capacity for a large range of SNR values, the decoding complexity is high. When the bubble decoder proposed in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** is used, instead of the optimum ML decoder, the algorithm still requires excessive number of operations per message especially for channels with small SNR values.

In this work, we state the problem that we want to solve as to design a simple decoding algorithm for spinal codes that needs less number of operations, especially on bad channel conditions, without sacrificing from the rate achieved on the channel. In other words, we aim at a decoding algorithm that is competitive with the original bubble decoder in terms of the achieved rate while having less complexity.

Therefore, we propose an algorithm that decreases the number of decoding operations per message with the help of some observations obtained via offline simulations; and by estimating the channel quality when the transmission is in progress. Our algorithm mainly keeps the value of the beam-width parameter B small when the decoding operation is likely to fail and increases the value of B up to an upper limit otherwise. In Section 4.1, we state and discuss our observations made by analyzing offline simulations of the bubble decoder. Section 4.2 presents our decoding algorithm for spinal codes, which is a modified form of the bubble decoder based on these observations.

4.1. Observations via Offline Simulations

4.1.1. Observation 1

Our first observation while using the bubble decoder is, for every SNR value, the decoding operation reaches 'success' at subpass values that directly depend on the SNR value. By definition, the rate achieved at smaller SNR values (worse channel conditions) is smaller than the rate achieved at high SNR values; i.e., when the channel is good. Since the rate is inversely proportional to the decoding time of the message, one can expect that as the SNR increases, the decoding operation takes less time.

We have obtained the minimum subpass values of successful decoding operations versus channel SNR by computer simulations. The complexity of the spinal decoder is exponential in the value of the parameter k and linear in the value of the parameter *B*. To reach rates close to capacity, *B* values between 64 and 256 are suggested in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** for *k*=4.

In our simulations with *n*=96, *k*=4, c=6 and *d*=1, the subpass values where successful decoding starts are found and given in Table 4.1 for *B*=256 and Table 4.2 for *B*=16 for SNR values between 0 and 30 dB.

Table 4.1 – Subpasses $K(i)$ at which successful decoding starts versus SNR = *i*, for *n*=96, *k*=4, c=6, *B*=256 and *d*=1

SNR	0	1	2	3	4	5	6	7	8	9	10
StartingSubpass	23	19	17	15	13	12	11	11	10	9	8

SNR	11	12	13	14	15	16	17	18	19	20
StartingSubpass	8	7	7	6	6	6	6	6	5	5

SNR	21	22	23	24	25	26	27	28	29	30
StartingSubpass	5	5	5	4	4	4	4	4	4	4

Table 4.2 – Subpasses $K(i)$ at which successful decoding starts versus SNR = i, for n=96, k=4, c=6, B=16 and d=1

SNR	0	1	2	3	4	5	6	7	8	9	10
StartingSubpass	25	21	18	15	15	14	13	11	11	10	9

SNR	11	12	13	14	15	16	17	18	19	20
StartingSubpass	8	8	7	7	7	6	6	6	6	6

SNR	21	22	23	24	25	26	27	28	29	30
StartingSubpass	6	6	6	6	6	6	6	6	6	6

As can be seen in Table 4.1, for SNR values between 0 and 30 dB, the decoding operation cannot reach success before the 4th subpass for B=256, which means the half of the first pass. Also, for small SNR values (for example 0 dB < SNR < 5 dB), a large number of decoding attempts are useless; i.e., the decoding operation will fail until enough number of symbols are received, since the channel is not good enough to achieve high rates. In the light of this observation, we attempt to define two rules that will help us through the design of our algorithm to reduce decoder complexity. The two rules that we obtain based upon Observation 1 are as follows.

Rule 1: The spinal decoder need not use large B values before the symbols belonging to the 4th subpass are received.

Rule 2: Provided that the channel conditions are estimated, the spinal decoder can defer the use of large B values until the receiver receives enough number of symbols for successful decoding.

Note that the first rule is for the parameters of the spinal code that are mentioned above. Similar rules with slight modifications can be obtained for other set of parameters.

To use the above mentioned “Rule 2”, one should estimate the channel quality during transmission. One can get channel state information in a transmission by several methods. In this section we also propose a channel quality estimation method for spinal codes. The method we recommend does not use pilot symbols to estimate channel quality, so the rate is not affected due to pilot symbol overhead. Since we are transmitting messages ratelessly, we do not need to inform the transmitter about the channel state, so the rate is also not affected by feedback transmission to the transmitter. Instead, the method uses the path metric values in the decoder’s candidate list at the end of every subpass, to estimate the SNR, with minor additional computation compared to the bubble decoder. This leads us to the second observation that we make using offline simulations.

4.1.2. Observation 2

The second observation that we make is that the distribution of the best path metrics at the end of each subpass depends on the channel state. Note that the best path metric value is defined to be the path metric whose value is the least among all candidate paths. To gain information about the distribution of the best path metrics versus SNR, we have run a series of simulations for SNR values between 0 and 30 dB. We have obtained the best path metric values at the end of each subpass until successful decoding occurs. The parameters of the spinal code during the simulations are *B*=256, *d*=1, *c*=6, *n*=96.

In Figure 4.1, the histograms of the best path metric values at the end of subpass 1 for the SNR values of 0, 5, 10 and 15 dB are shown. Similarly in Figure 4.2, the histograms of the best path metric values at the end of subpass 6 are shown for the same set of SNR values.

The y-axes in Figures 4.1 and 4.2, show the frequency of each path metric value, the x-axes show the best path metric values for the given SNR values, with the above mentioned spinal code parameters. Note that the actual path metric values seen on these figures are subject to change when another set of parameters is used for encoding and decoding; however the differences will remain similar.

As can be seen from the figures, the distributions of the best path metrics differ for each SNR value. In Figure 4.1, as the SNR increases, the path metric histograms of Subpass 1 become narrower; i.e., they have smaller variance. Also the maximum value of the best path metric gets smaller as the SNR increases. Similarly, in Figure 4.2, at SNR=5 dB, Subpass 6 best path metric can get values between 1200 and 11000; while for SNR=10 dB they are between 1600 and 3200, and for SNR=15 dB, they are getting values between 1200 and 2800. So, in both figures, the best path metrics for smaller SNR values can take values larger than the maximum values obtained for large SNR values.

These differences in the distribution of best path metric values are the key observation in the application of the algorithm we propose.

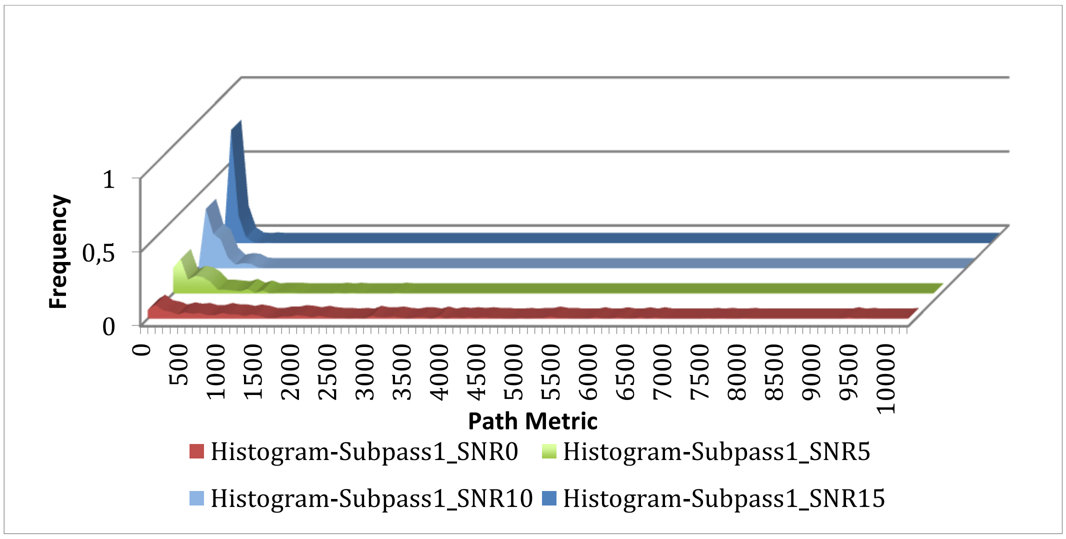


Figure 4.1 – Histogram of Subpass 1 best path metrics at {0, 5, 10, 15} dB for *B*=256

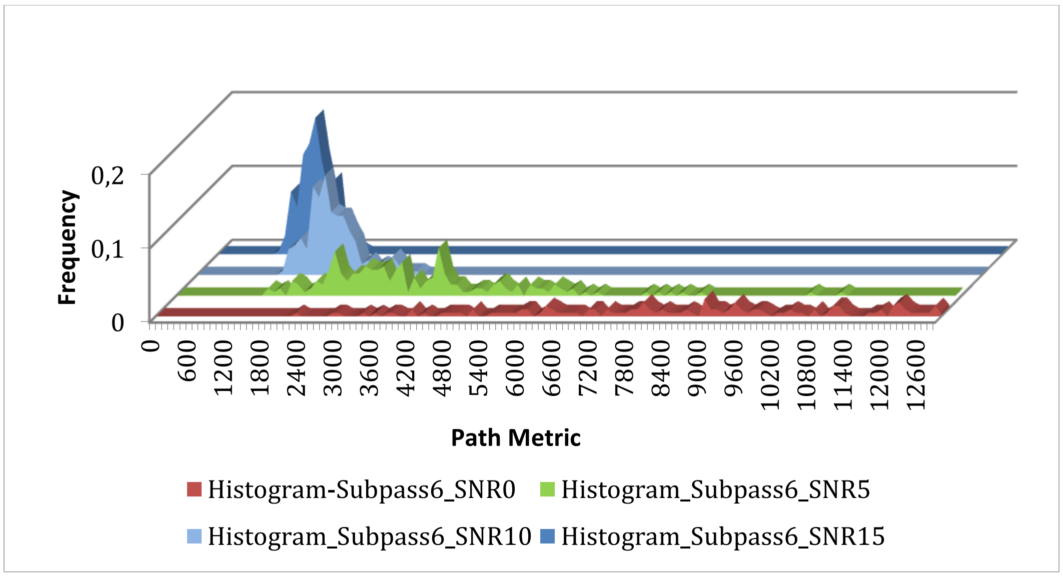


Figure 4.2 – Histogram of Subpass 6 best path metrics at {0, 5, 10, 15} dB for *B*=256

In Appendix A, we give cdf plots of best path metrics at subpasses 1 to 6 for SNR values 0, 5, 10 and 15 dB. Note that the distributions of the best path metrics given in Appendix A are for *B*=1. As expected, these values are different from the ones in Figures 4.1 and 4.2 obtained for *B*=256. The algorithm that we introduce in the next section needs these distributions for *B*=1 since we start with a value of 1 and increase the number of candidate paths when necessary.

If we model the path metric as a random variable *X*, metric distributions found for each subpass and each SNR can be considered as the estimates of the corresponding probability density functions $f_X(x)$, abbreviated as *pdf*. The integral of the *pdf* is called the cumulative distribution function $F_X(x) = \Pr\{X \leq x)$, abbreviated as *cdf*. The *cdf* of the best path metric values at the end of subpass 6 at SNR=10 dB with *B*=1 together with that of *B*=256 is given in Figure 4.3. As one can see, the best path metrics for *B* = 256 take much smaller values than the one for *B* = 1.. Also, the best path metrics for *B* = 256 are confined to a narrower region compared to the one for *B* = 1.

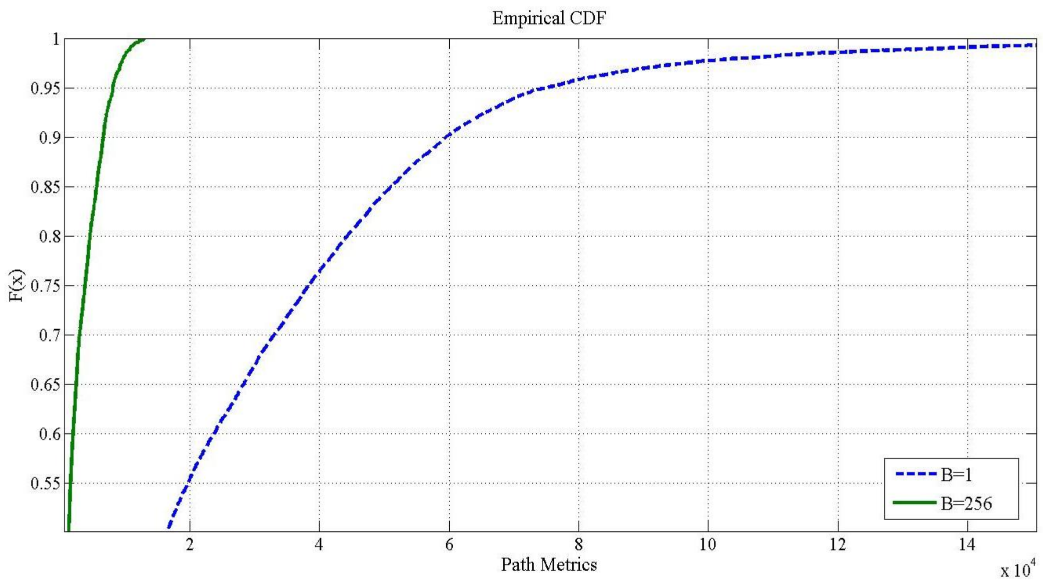


Figure 4.3 – Estimated *cdf* of Subpass 6 best path metrics at SNR=10 dB for *B*=1 and *B*=256

Our decoding algorithm to be presented in the next section strongly depends on a parameter L that we call the *‘LimitValue’*, which shows the value of the path metric for which the cumulative distribution function $F_X(L) = \Pr\{X \leq L)$, abbreviated as *cdf*, reaches a given value a; i.e., $F_X(L) = a$. The choice of a (say as 0.95, 0.8 or 0.6) is crucial since it affects both the rate performance and the computational load of the proposed algorithm. Note that, the value of L is dependent on the SNR value of the channel, the subpass number k_{sp} of the algorithm, the *cdf* parameter a and the size of the candidate list *B*; hence, $L(SNR, k_{sp}, a, B)$ is a list that needs to be initially prepared for all SNR, k_{sp}, a and *B* values of interest.

Using the path metric distributions for *B*=1, we find the *‘LimitValue - L’* of the path metric X such that a given fraction a of the path metric X is smaller than L; in other words, $F_X(x) = \Pr\{X \leq x) = a\}$. We have repeated these simulations for all subpasses and all SNR values in the range [0,30] dB, and obtained all L values corresponding to $F_X(L) = 0.5$, 0.6, 0.7, 0.8, 0.9 and 0.95.

As an example, we have tabulated some of these results in Table 4.3, for the 6th subpass, SNR values 0 to 10 dB and $F_X(L) = 0.95$, 0.8, 0.6. One can see Appendix B for more results with *B*=1.

Table 4.3 – Limit values for different $F_X(x)$ and SNR's at the end of subpass 6 for *B*=1

SNR	$LimitValue\ L$ for $F_X(L) = 0.95$	$LimitValue\ L$ for $F_X(L) = 0.8$	$LimitValue\ L$ for $F_X(L) = 0.6$
0	36364	27266	21219
1	30041	23628	17202
2	25273	19901	16296
3	21775	15578	14007
4	18914	14172	12609
5	16238	12520	11261
6	14264	11898	**9519**
7	13776	**10625**	**9570**
8	12649	**10736**	8783
9	12313	9471	8526
10	11658	9346	8102

Observing Table 4.3, one can deduce that

i) The limit value L, below which say 95% of the path metric values remain, decreases as the SNR increases,

ii) For $F_X(L) = 0.8$ and $F_X(L) = 0.6$, bold entries of Table 4.3 indicate small discrepancies from the behavior stated in (*i*).

iii) Rough estimation of the channel quality (in terms of the SNR) can be done by means of a rough classification of L values in Table 4.3, found by the bubble decoder.

The insertion of the channel estimation into the decoding algorithm in a proper way to speed up computations is the subject of the next section.

4.2. The Algorithm for Efficient Decoding of Spinal Codes

Based upon the above-mentioned Observations 1 and 2, we propose an algorithm that uses the information obtained by guessing the successful decoding time and exploiting the distributions of the best path metrics at the end of subpasses for every SNR value. This information is used for increasing the value of *B* whenever necessary. Since successful decoding is not possible before Subpass 4 for SNR values smaller than 30 dB, we use the starting value of the parameter *B* as 1. By starting with *B* = 1, we minimize the cost of the bubble decoding algorithm while still gaining information about the channel state.

Since for each SNR value, we have found that successful decoding starts after some subpass, we propose a decoding algorithm that basically keeps the size of the candidate list, in other words the value of *B*, small until the estimated channel quality permits successful decoding of the transmitted message.

The algorithm we propose uses the $LimitValue$ - $L(SNR, k_{sp}, a, 1)$ lists obtained for a specific $F_X(L) = a$ and *B*=1 for all SNR and subpass values k_{sp} of interest, while deciding whether to increase the value of *B* or not. At the end of each subpass of decoding, we estimate the channel SNR to decide whether we should increase *B* or not. To estimate the SNR, we compare the best path metric X_{best} obtained at the k_{sp}'th subpass with all *LimitValue*'s belonging to that subpass at different SNR's. Whenever X_{best} falls into the interval $L(i+1, k_{sp}, a, 1) < X_{best} < L(i, k_{sp}, a, 1)$; then the estimated $\hat{SNR} = i$.

For example, suppose the best path metric at the end of Subpass 6 is 18000 and we have chosen to use our algorithm for $F_X(L)$ = 0.95. Table 4.3 that shows Subpass 6 distributions for *B*=1, indicates that for $F_X(L) = 0.95$, the minimum *L* value larger than 18000 is *L*=18914, corresponding to SNR = 4 dB. So, we estimate the SNR to be 4 dB.

We then use Table 4.1 that shows the subpasses $K(i)$ at which the successful decoding starts for each SNR=i value. If the current subpass number k_{sp} is less than the starting subpass number ($K(i) - 1$), we keep B=1. On the other hand, if $k_{sp} = K(i) - 1$, we increase B to B_{max} to get prepared against the possibility of successful decoding at the next subpass. We note that, since for the SNR range of [0,30] dB one cannot decode before the 4th subpass, our algorithm does not increase the value of B before the end of the 3rd subpass regardless of the best path metric values. In other words, we propose an algorithm that starts with a beam-width value B=1 and increases B only when the decoder is close to "successful decoding". Starting with B=1 is due to the fact that a successful decoding is not possible at the first 3 subpasses for the SNR values and spinal code parameters mentioned in Rule 1. The algorithm does not change B when there is not enough chance of successful decoding. The *cdf* value a that will be used to find the $LimitValue$ L, such that $F_X(L) = a$, is determined by running simulations at different parameter values. The results of those simulations and the best choice for the *cdf* values a are explained in Chapter 5, where we present our simulation results.

Our modified decoding algorithm also assigns a maximum value for B, which cannot be exceeded, denoted by B_{max} which can be selected as 256 for example. This limiting is essential since there will always be a limit on the computational complexity in real life scenarios. The modified decoding algorithm is stated as follows:

Modified Decoding Algorithm

```
input: $B_{max}$ , $d, c, n, k, a, K(SNR)$ and $L(SNR, k_{sp}, a, 1)$ for all $SNR$'s & $k_{sp}$'s
output: Message $M$
$B \leftarrow 1$
while (decoding is not successful)
        $DecodingResult$ = bubble_decoder($B, d, c, n, k$)
        if $DecodingResult$ is not successful
                if increase_beamwidth($k_{sp}$, $X_{best}$, $a$, $K$, $L$)
                        true, $B \leftarrow B_{max}$
                        false, $B \leftarrow B$
        else // Decoding is successful
                $M$
```

The function **increase_beamwidth**(.) function compares the *BestPathMetric*, X_{best} with the *LimitValue* - $L(SNR, k_{sp}, a, 1)$ obtained from offline simulations for each subpass and SNR value, and returns 'true' if the next subpass is the starting subpass for successful decoding for the estimated SNR. The algorithm for the **increase_beamwidth**(.) function is summarized as follows:

'increase_beamwidth(.)' Function

```
input: $k_{sp}$, $X_{best}$, $a$, $K(SNR)$ and $L(SNR, k_{sp}, a, 1)$ for all $SNR$'s and $k_{sp}$'s
output: Decision to increase the value of $B$
        $\hat{SNR}$ = snr_estimation($k_{sp}$, $X_{best}$, $a$, $L$)
        StartingSubpass = $K(\hat{SNR})$
        return (StartingSubpass =< ($k_{sp}$+1))
```

The function **snr_estimation**(.) returns the estimate $\hat{SNR} = i$, if the best path metric, X_{best}, satisfies $L(i+1, k_{sp}, a, 1) < X_{best} < L(i, k_{sp}, a, 1)$. For the estimate $\hat{SNR} = i$, the function **starting_subpass**(.) returns the starting subpass number $K(i)$ of successful decoding, according to Table 4.1. Then, the decoder decides whether to increase the value of B or keep it the same according to the following rule:

$$k_{sp} + 1 \geq K(i) \;\Rightarrow\; NewB = B_{max}$$
$$o.w. \;\Rightarrow\; NewB = B \tag{4.1}$$

The reason we compare the current subpass number with the starting subpass value with a '≤' sign (instead of only '=' sign) is a precaution against estimated SNR changes. Suppose that at the end of subpass 4, we have estimated an SNR_1 and concluded that the starting subpass of successful decoding is 7. In this case the decoder will not increase the value of B and continue operation with B = 1. Following this, suppose at the end of subpass 5, we have estimated an $SNR_2 > SNR_1$ and concluded that the starting subpass of successful decoding is 5. In this case, since the starting subpass value (5) is smaller than the next subpass value (6), the decoder should certainly increase the value of B to B_{max}. If the algorithm included '=' sign instead of '≤' sign, the value of B would be stuck at 1 and sudden increases in SNR estimates would not be treated properly.

SNR estimation of our modified decoding algorithm can be better understood with the help of Figure 4.4, where the leftmost distribution is for the largest SNR value and the $L(SNR, k_{sp}, a, 1)$ values at each $SNR = i$ (corresponding to the same subpass k_{sp} and a specific *cdf* value *a*) are shortly denoted by $L(i+1)$, $L(i)$ and $L(i-1)$ from left to right. The plotted distributions are for illustration purposes and not derived from offline simulations.

As explained above, the modified algorithm estimates the SNR to be equal to *i* dB, if the best path metric value X_{best} obtained at the current subpass falls into the interval $L(i+1) < X_{best} < L(i)$ shown in Figure 4.4.

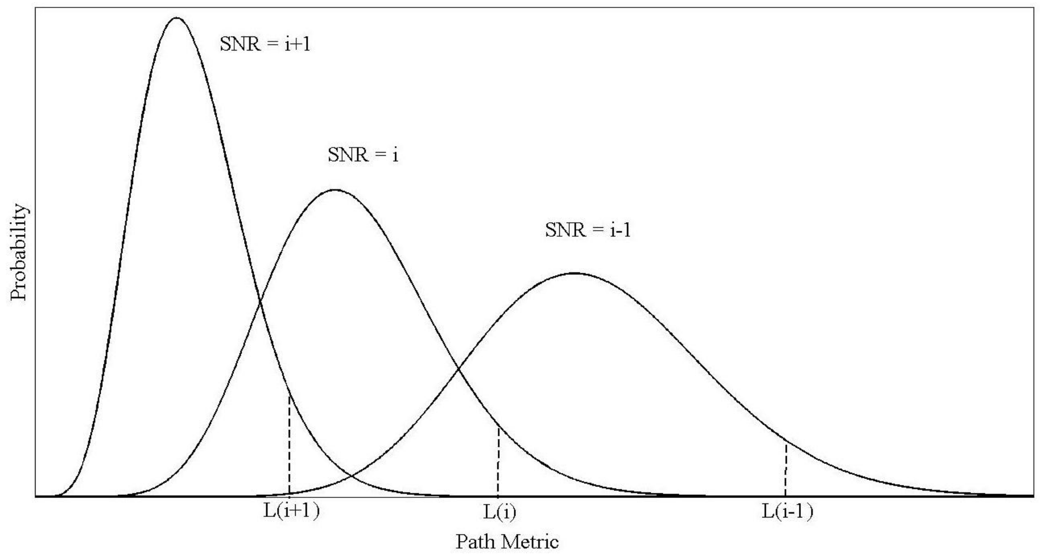


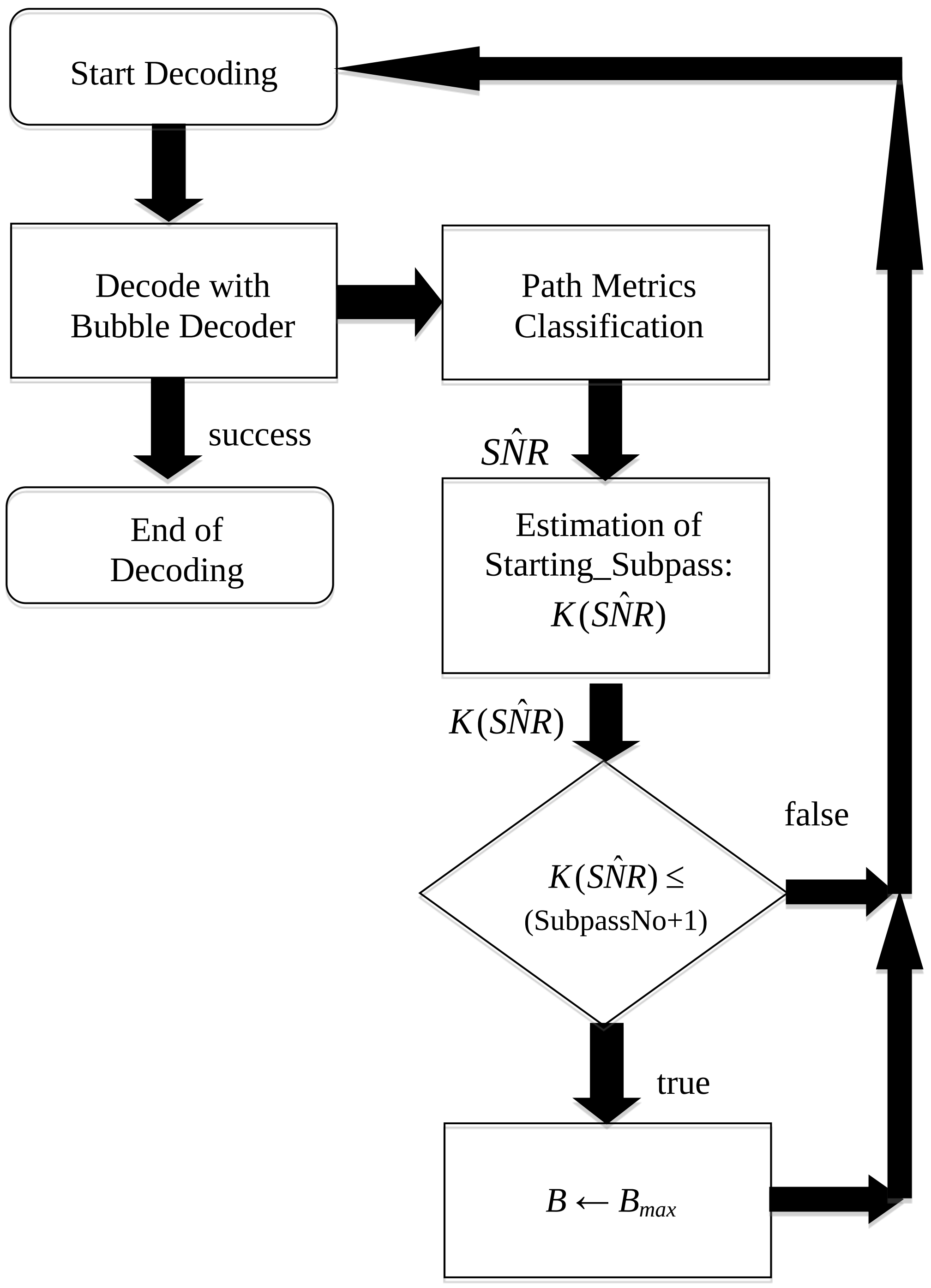
Figure 4.4 – Typical probability densities of best path metrics at different SNR values

Including the details of the **increase_beamwidth**(.) function, the overall decoding algorithm can be stated as follows:

```
input: $B_{max}$ , $d, c, n, k, a, K(SNR)$ and $L(SNR, k_{sp}, a, 1)$ for all $SNR$'s & $k_{sp}$'s
output: Message $M$
$B \leftarrow 1$ (1)
while (decoding is not successful)
        $DecodingResult$ = bubble_decoder($B, d, c, n, k$)
        if $DecodingResult$ is not successful
         $\hat{SNR}$ = snr_estimation($k_{sp}$, $X_{best}$, $a$, $L$)
        StartingSubpass = $K(\hat{SNR})$

        if (StartingSubpass =< ($k_{sp}$+1))
                        true, $B \leftarrow B_{max}$
                        false, $B \leftarrow B$
                end if
        else // Decoding is successful
                $M$
        end if
end while
```

The flowchart version of our modified algorithm is illustrated in Figure 4.5.



Start Decoding
Decode with Bubble Decoder
Path Metrics Classification
success
$S\hat{N}R$
End of Decoding
Estimation of Starting_Subpass: $K(S\hat{N}R)$
$K(S\hat{N}R)$
false
$K(S\hat{N}R) \leq$ (SubpassNo+1)
true
$B \leftarrow B_{max}$

Figure 4.5 – Modified decoding algorithm flowchart

In the next chapter, we discuss the simulation results obtained by using our modified decoding algorithm proposed in this section. Also we will compare spinal coding performance when we use different values for parameters c, k, B, n, or apply different hash functions. Finally we will compare the performance gain obtained when we use the puncturing scheme.

CHAPTER 5

SIMULATION RESULTS

In this chapter, several simulation results about the effect of spinal code parameters, namely the message length *n*, message segment length *k*, constellation size 2^{2c}, and the beam-width *B* are presented. Secondly, we investigate the effect of the hash function on the performance of the spinal code. Then, we discuss the effect of using puncturing during the transmission by comparing it to the case without puncturing.

In the second part, we present the results obtained through computer simulation of our modified decoding algorithm introduced in Chapter 4, and compare its performance with that of the bubble decoder. Our comments on the results of the modified algorithm are given by inspecting two important performance criteria, namely the number of operations made by the decoder to decode a message and the achieved rate. These two performance criteria are subject to change with changing input parameters, namely the channel SNR and the *cdf* value *a* that determines the limit values of the path metrics at each subpass.

In Section 5.1, we describe the implementation details of the simulation environment. Sections 5.2 to 5.5 examine the effects of the parameters *c, k, B, n* respectively, on the performance of spinal codes. Section 5.6 summarizes the effect of the hash function and Section 5.7 discusses the effect of puncturing on the performance. In Section 5.8, we present the results obtained by applying our modified decoding algorithm, and make a comparison with the bubble decoder.

5.1. Simulation Environment

Before going into the details of the results, first we explain the high level structure of the simulation model.

The simulation model is composed of the encoder, channel, decoder and the test manager. The encoder module is implemented in the same way as explained in Chapter 3. For Sections 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7 the decoder is the bubble decoder introduced in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** and explained in Section 3.1. For Section 5.8, the decoder module is the bubble decoder that uses our modified algorithm for decreasing the number of operations per message.

The channel is assumed to be Additive White Gaussian Noise (AWGN) channel with the noise power defined as

$$E_n = E_s \,/\, (2 \times \mathrm{SNR}) \tag{5.1}$$

for I and Q channels, where E_s is the symbol energy.

Throughout the simulations, the energy of the symbol is kept constant and the noise energy is calculated using (5.1).

The puncturing scheme used, whenever it is applicable, is similar to the one mentioned in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** with each pass composed of 8 subpasses as explained in Section 3.3. The two performance criteria; the rate and the number of operations per message are averaged over 1000 simulation runs. In the simulation setting, the transmission continues until the decoder sends the encoder an acknowledgement message. The test manager module controls the messages that are sent between the encoder, channel and decoder processes and logs the result of each simulation run.

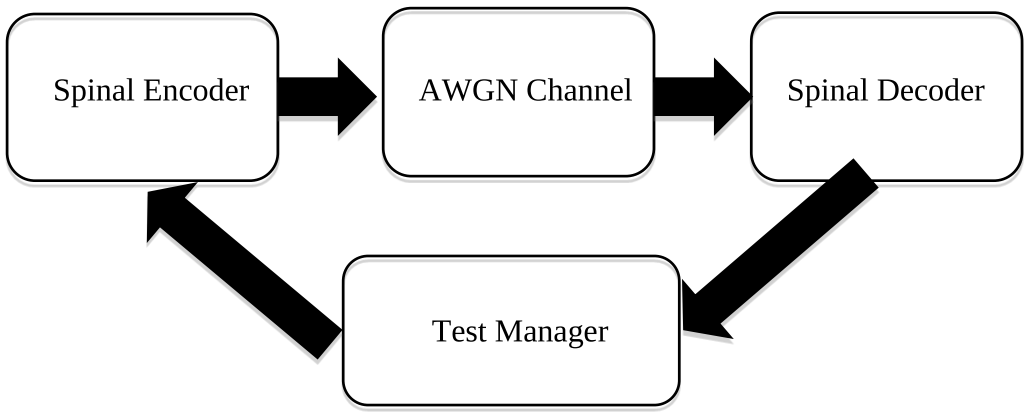


Figure 5.1 – Structure of the simulation software

The structure of the simulation software is shown in Figure 5.1. For each message to be transmitted, the spinal encoder block works as presented in Chapter 3 and generates complex encoded symbols as long as the transmission continues. The generated symbols are sent to the AWGN channel block.

The AWGN channel block generates random samples from a Gaussian distribution with the noise power given as in (5.1). It then adds these noise samples to each of the I and Q channel symbols and sends the output to the spinal decoder block.

The decoder (using our modified algorithm for Section 5.8) uses spinal code parameters given at the beginning of the simulation to decode the message. It then acknowledges the encoder by the help of the test manager module. The simulation loop continues until the given number of messages are transmitted to the receiver. The test manager module also logs the results of each transmission for further inspection.

As mentioned in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**, the linear-time decoder should use B and d parameters as constant, so both for simplicity and linearity of decoder we take d =1 in our simulations.

In Chapter 3, we have considered the RNG function at the constellation-mapping step. In the simulations, we use the hash function that is used to generate the spine value, to also function as the RNG block that is present in the constellation-mapping step of the encoder. We discuss the results obtained by simulations that examine the effect of the parameter c on the performance of the spinal codes in the next section.

5.2. Effect of the Parameter *c* on the Performance

In this section, we present the results obtained by changing the parameter c in the spinal encoding and decoding process. The parameter c is affecting how many bits of the RNG output at the encoder will be used, when the encoder is generating the symbols at the constellation-mapping step. In our implementation of spinal codes simulation software, we simply take the least significant $2c$ bits of the RNG output at the encoder and decoder. Since the output of the RNG function is a finite-length bit sequence, as the transmission continues the encoder needs new coded-bits to generate new modulated symbols. So, the encoder generates more coded-bits using the RNG function recursively.

For example, when one uses Lookup3 hash function for the RNG, the output value is a 32-bit number. If we select $c = 8$, w*e* take each 16-bit segment of the output value to generate symbols. So we can use the RNG output to create 2 symbols. One can use a larger portion of the RNG output; however doing so increases the decoder complexity since more RNG operations should be performed for the same number of encoded symbols.

Table 5.1 – Simulation parameters for Section 5.2

Parameter	*Value*
c	{1,2,3,4,5,6,7,8}
B	256
n	96
k	{2,3,4}
SNR	{5,10,15,20,25,30}
Hash Function	Lookup3 Hash Function
Puncturing	Active

For each different value of parameters *c*, *k* and SNR, 1000 simulations were run and averaged. We set other parameters as having constant values as written in Table 5.1. The simulations were run for SNR values of 5 dB spacing, from 5 dB to 30 dB.

We present the graphical result of simulations with $k = 2$, in Figure 5.2. One observes that as the SNR increases, all of the achieved rate versus SNR curves start to get saturated. The saturation levels are monitored at SNR >20 dB for all values of *c*, except $c = 5$. We conclude that $c = 5$ is the best choice for $k = 2$ and for the parameters in Table 5.1.

Secondly, we give the simulation results for $k = 3$ in Figure 5.3 with the other parameters as in Table 5.1. Saturated curves of Figure 5.3 show the existence of the rate limiting effect for *c* values of 1, 2, 3. However, as *c* gets larger, the saturation disappears and the achieved rate versus SNR curves increase almost linearly for $4 \leq c \leq 8$. Hence, all *c* values greater than 3 can be used in order to maximize the achieved rate.

Finally, the simulation results for $k = 4$ are plotted in Figure 5.4. One can observe that saturated regions exist for $c = 1, 2, 3, 4$ and 5. For the spinal code to work efficiently between 0 and 30 dB, a *c* value of at least 6 should be used not to limit the rate achieved. So, $c = 6$, 7 and 8 can be used for $k = 4$ and in the above mentioned SNR range and for the parameters as in Table 5.1.

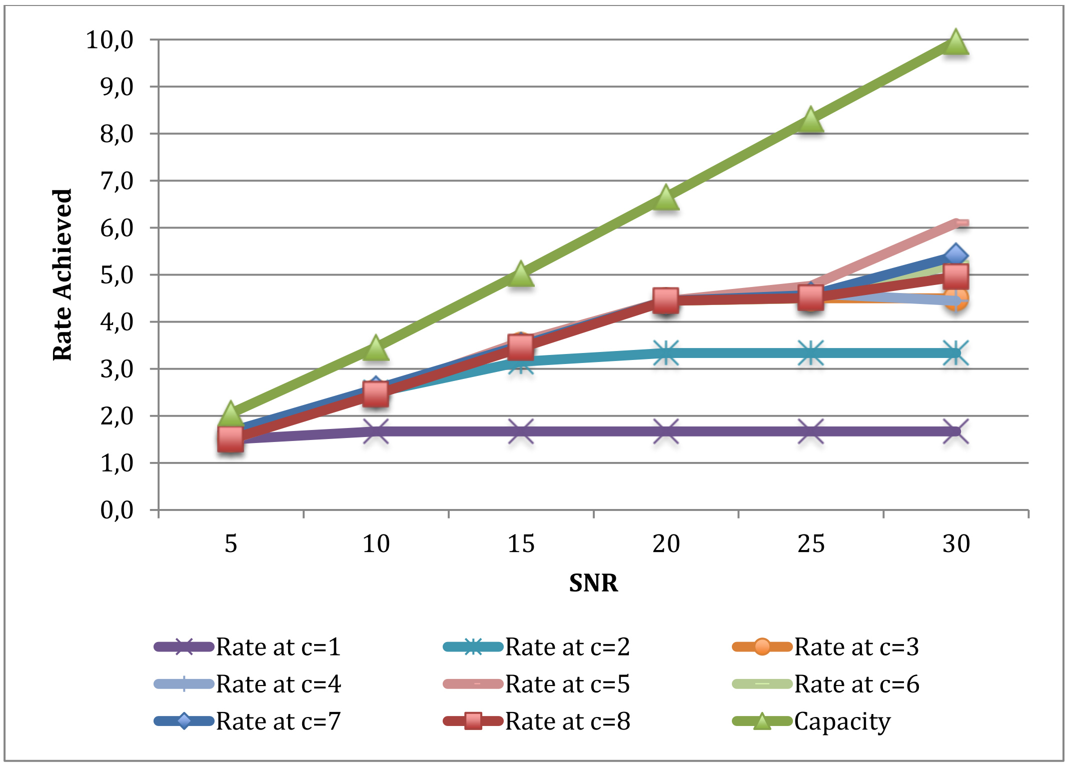


Figure 5.2 – Rate achieved for different *c* values when *k*=2

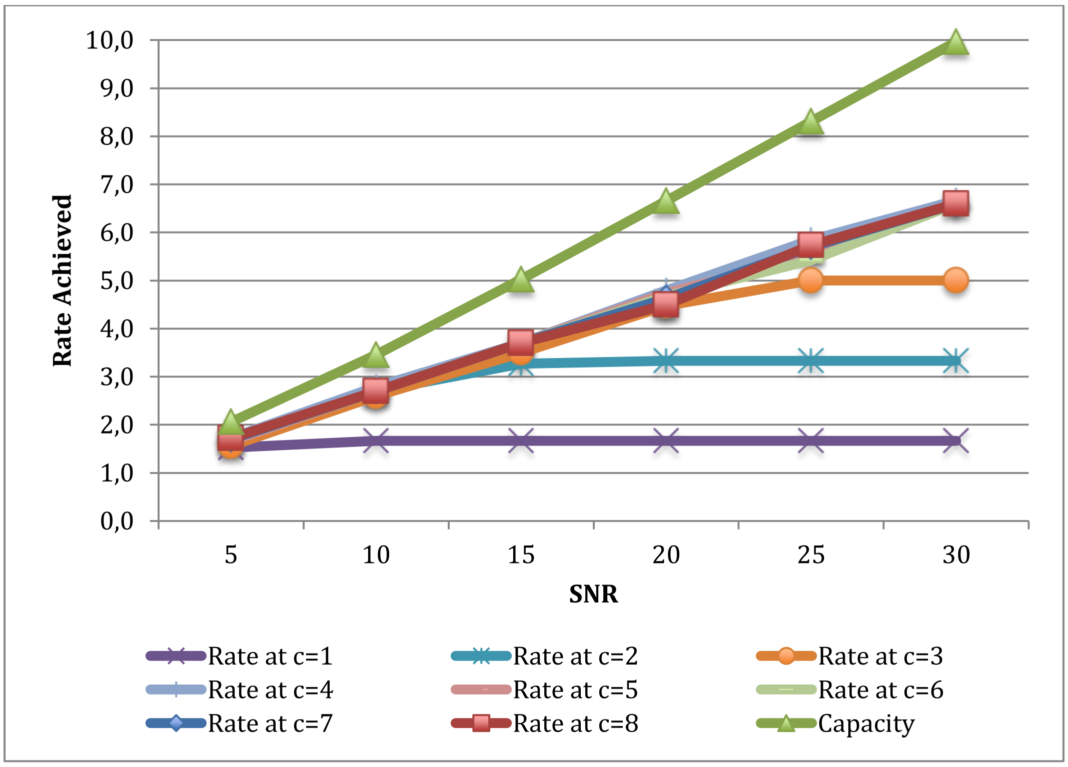


Figure 5.3 – Rate achieved for different *c* values when *k*=3

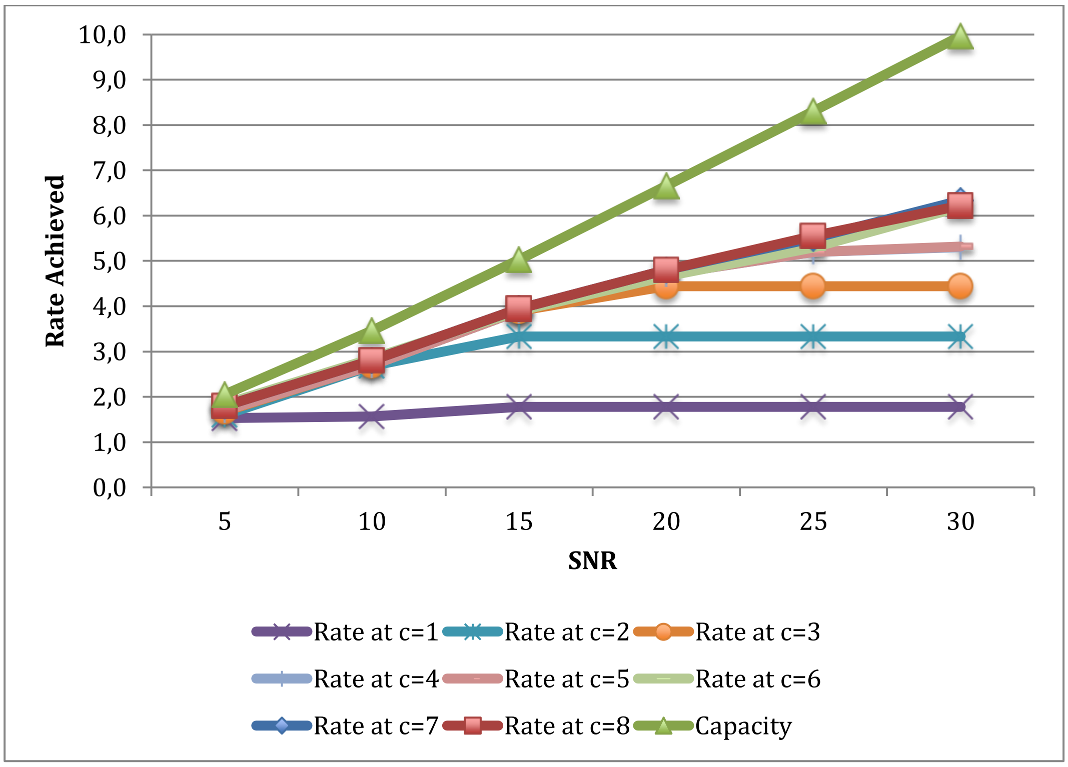


Figure 5.4 – Rate achieved for different *c* values when *k*=4

It seems that for message chunk lengths $2 \leq k \leq 4$, one should avoid small values of *c* like 1, 2 or 3. However, noting that different results can be obtained by using different values especially for the parameter *n*, which is discussed in Section 5.5; we will postpone our comments on the choice of optimum *c* values to a later section. In the next section, we will investigate the effect of different values for *k* on spinal code performance.

5.3. Effect of the Parameter *k* on the Performance

The simulation parameters for this section are summarized in Table 5.2.

Table 5.2 – Simulation parameters for Section 5.3

Parameter	*Value*
c	See Note 1
d	1
B	256
n	96
k	{2,3,4}
SNR	{5,10,15,20,25,30}
Hash Function	Lookup3 Hash Function
Puncturing	Active

Note 1: *Value of the parameter c is chosen such that it is the best selection among the other options; so, it is* 5 *for k* = 2 and 7 *for k* = 3, 4.

We plot the achieved rate values obtained over 1000 simulations in Fig. 5.5. As one can see in Figure 5.5, there is not much difference in the achieved rate for SNR values smaller than 20 dB. For larger SNR values, $k = 3$ is slightly better than $k = 4$, while $k = 2$ is a slightly worse choice for the parameters given in Table 5.2. Since using smaller values of k decreases the complexity of the encoder, one can choose to use $k = 3$ in this setting.

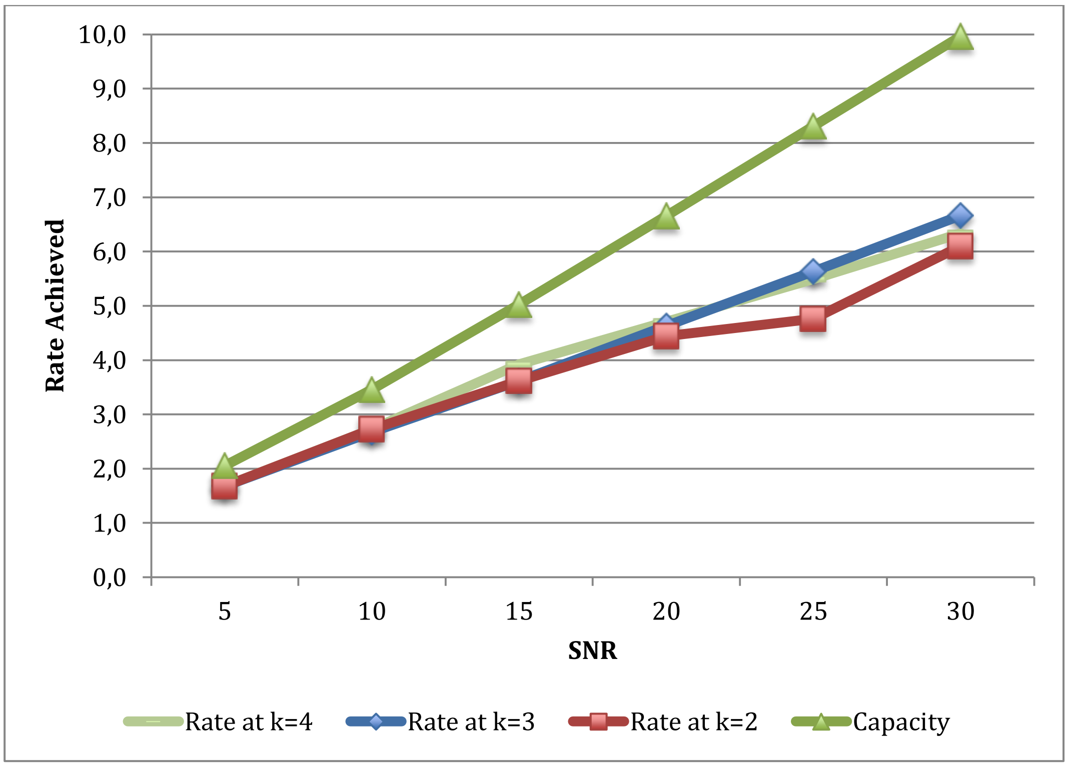


Figure 5.5 – Rate achieved for different *k* values

5.4. Effect of the Parameter *B* on the Performance

In this section, we present simulation results that are obtained by using different beam-width values, *B*, to decode the transmitted message.

Table 5.3 – Simulation parameters for Section 5.4

Parameter	*Value*
$\{k\,,c\}$	{3,4} and {4,6}
B	{2,4,8,16,32,64,128,256,512}
n	96
SNR	{5,10,15,20,25,30}
Hash Function	Lookup3 Hash Function
Puncturing	Active

We tried B values from 2 to 512 and chose $\{k ,c\}$ values of $\{3, 4\}$ and $\{4, 6\}$ with $n = 96$, as shown in Table 5.3. We run the simulations at SNR values of 5 to 30 dB. The other parameter values used in this section are given in Table 5.3.

If one operates on small message chunks like $k = 3$; Figure 5.6 shows that the beam-width value $B = 256$ is as good as $B = 512$. On the other hand, as observed from Figure 5.7 obtained for $k = 4$, the beam-width value $B = 512$ gives the best results. The former case seems to be more advantageous, since the value of B directly affects the number of operations made per message. Moreover, the rate performance versus SNR in the best curves of Figure 5.6 and 5.7 are quite similar, showing that the message segment length $k = 3$ does not bring any weakness with respect to $k = 4$. Hence $k = 3$ should be preferable for the simplicity of the decoding tree.

We also observe that, if one designs a code for an application that restricts the SNR to smaller values, smaller B values can be selected since the value of B directly affects the number of operations made per message.

In the next section, we will discuss the effect of the message length on the spinal code performance.

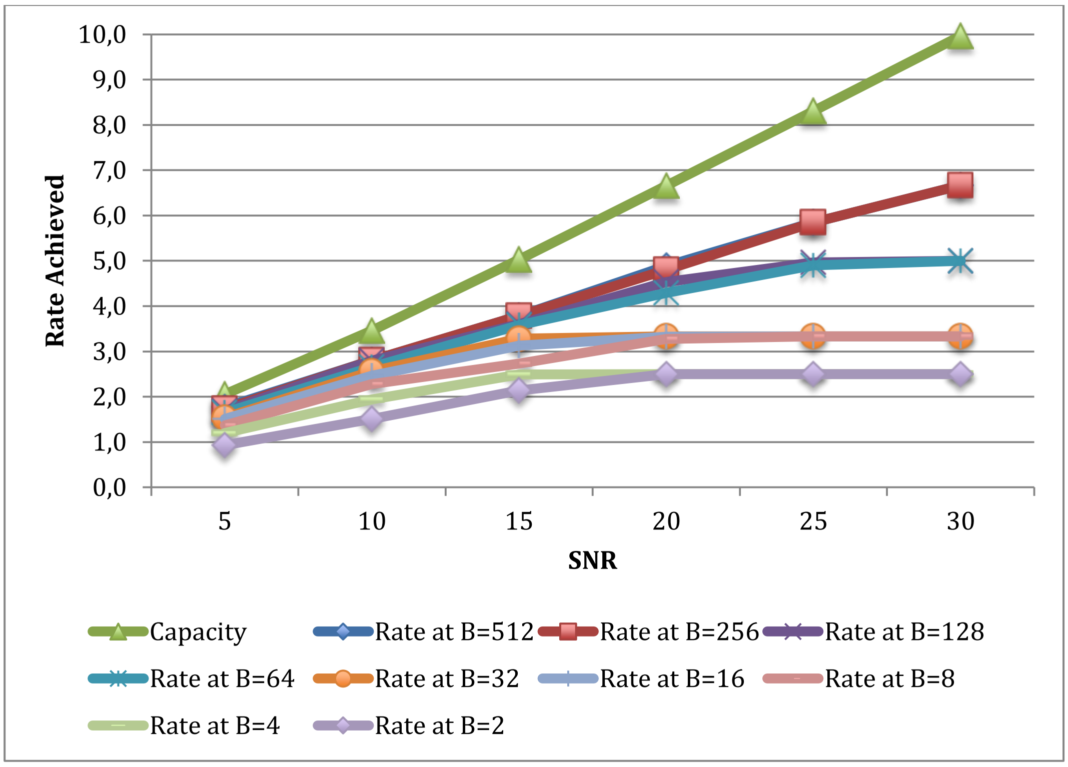


Figure 5.6 – Rate achieved for different *B* values for *k* = 3, *c* = 4

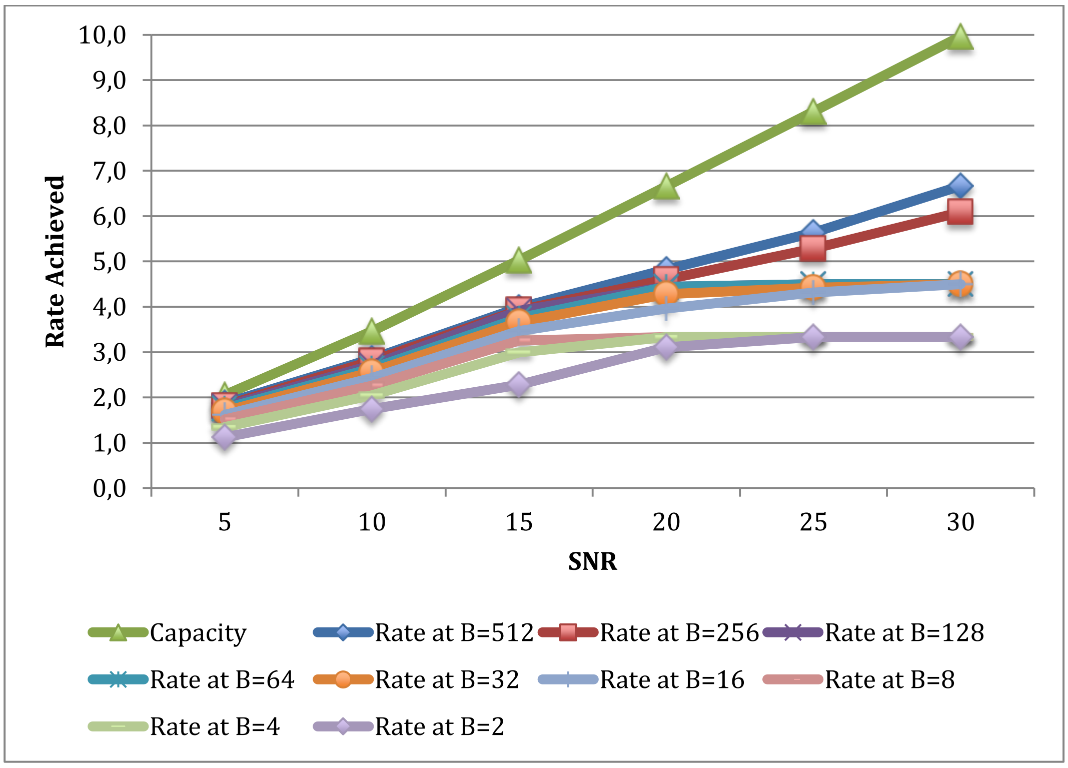


Figure 5.7 – Rate achieved for different *B* values for *k* = 4, *c* = 6

5.5. Effect of the Parameter *n* on the Performance

In this section, we will present the results of simulations when we change the length of the transmitted message. In **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**, it is mentioned that, as the length of the message increases, the probability of the correct path to be lost also increases. So we expect smaller values for the achieved rate for longer messages, which is partially supported by our simulation results.

The simulation parameters for this section are shown in Table 5.4.

Table 5.4 – Simulation parameters for Section 5.5

Parameter	*Value*
{*k* ,*c*}	{3,4} and {4,6}
B	256
n	{64,72,96,120,128,192,240,288}
SNR	{5,10,15,20,25,30}
Hash Function	Lookup3 Hash Function
Puncturing	Active

We plot the results of $k = 3$, $c = 4$ with *n* values of 72, 96, 120, 192, 240 and 288 in Figure 5.8. One observes that only for $n = 96$, the achieved rate is not limited for SNR values larger than 25 dB, so 96 bits seems to be the best choice. Also one can conclude that selecting 72 bits, which is the smallest message length in the given set for $k = 3$, is a bad choice.

In Figure 5.9, we present the results of $k = 4$, $c = 6$ with *n* values of 64, 96, 128, 192, 256 and 288. Although there are no appreciable differences, we observe that $n = 64$ has the smallest rate achieved among these alternatives, followed by $n = 96$ and 128. Also we can say that the best choice for the message length is 192 bits for these set of parameters.

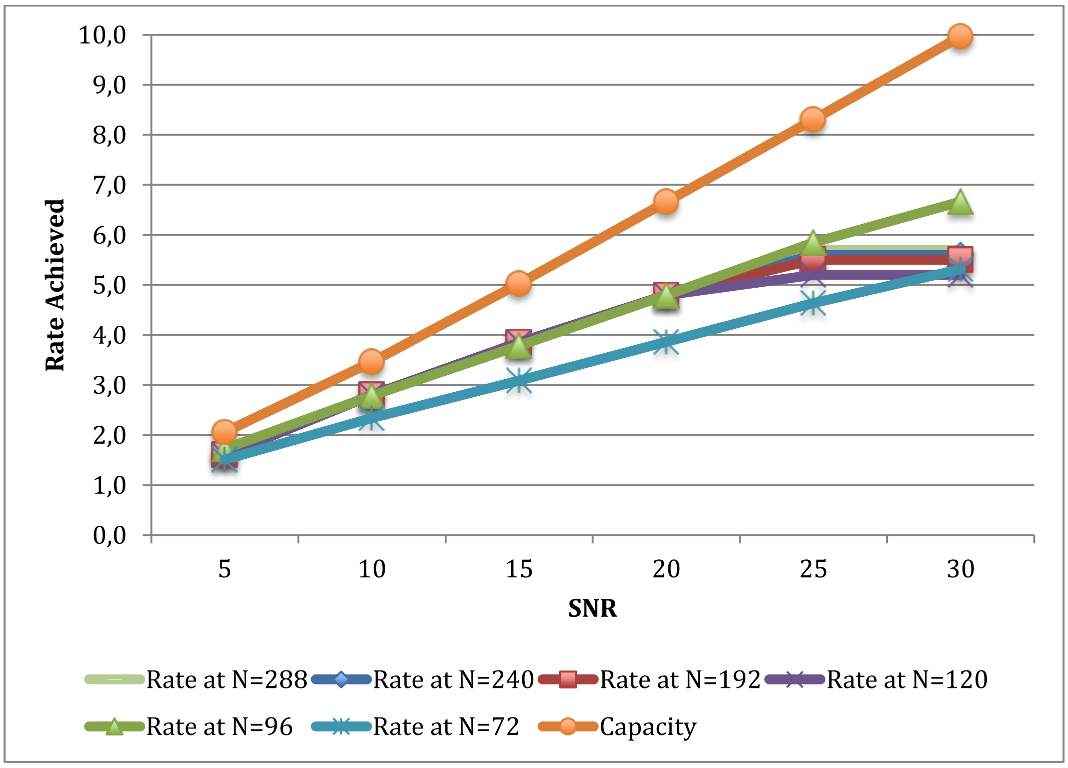


Figure 5.8 – Rate achieved for different *n* values for *k* = 3, *c* = 4

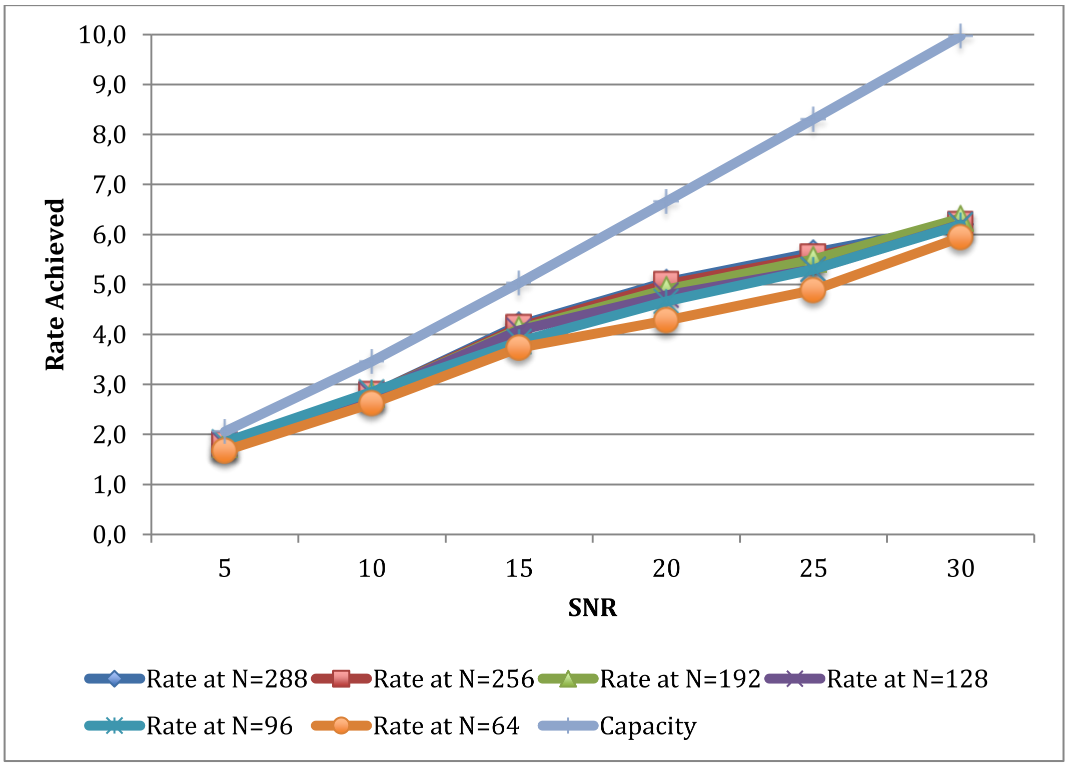


Figure 5.9 – Rate achieved for different *n* values at *k* = 4, *c* = 6

5.6. Effect of the Hash Function on the Performance

In this section we discuss the effect of different hash functions on the performance of the spinal code. The hash function used in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** is “one-at-a-time” hash. We used Lookup3 hash function in our simulations when examining the effect of other spinal code parameters to the performance throughout Chapter 5. In this section we compare the results obtained by using two standard cryptographic hash functions, the MD4 (Message Digest Algorithm 4) and the SHA (Secure Hash Algorithm).

The MD4 (RFC 1320) hash function creates a 128-bit hash value while the SHA (FIPS 180-2) hash function creates a 160-bit hash value. Simulation parameters for this section are given in Table 5.5.

Table 5.5 – Simulation parameters for Section 5.6

Parameter	*Value*
$\{k, c\}$	{4,6} for Lookup3 and SHA, {4,7} for MD4
B	{256}
n	96
SNR	{5,10,15,20,25,30}
Hash Function	Lookup3, MD4 and SHA
Puncturing	Active

As we can see by inspecting Figure 5.10, one can get an average of 7% increase in the rate achieved by using the MD4 hash function instead of using the Lookup3 hash function. The increase is over 10% for some SNR values. We also observe that not much gain can be achieved by using the SHA hash function instead of Lookup3. To comment on the reasons of this difference in relative behaviors of MD4 and SHA as elements of spinal encoder/decoder is quite difficult and beyond the scope of this work.

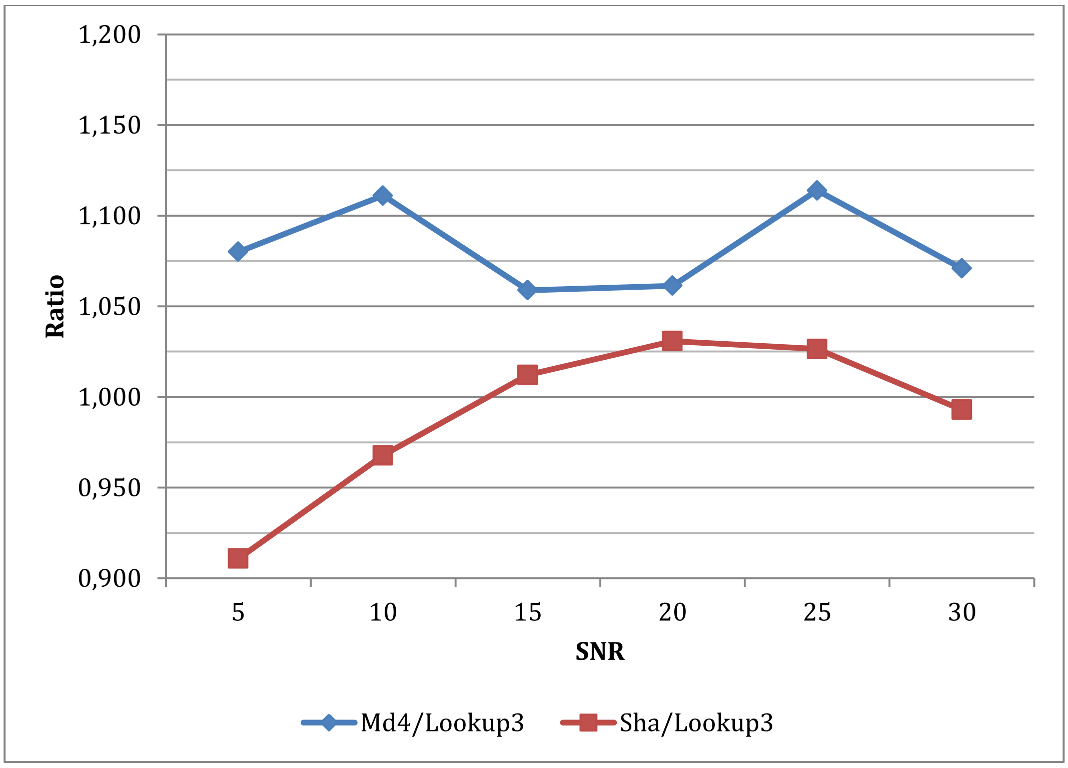


Figure 5.10 – Achieved rate ratio of the MD4 and SHA hash functions to Lookup3

5.7. Effect of the Puncturing on the Performance

In this section, we discuss the gain in the achieved rate of the spinal codes brought by puncturing. The parameters we use in simulations for this section are as given in Table 5.6.

We expect large gains in the achieved rate for large SNR values for which the channel capacity is high, but the information rate is limited by the value of *k* when the puncturing scheme is not used.

Table 5.6 – Simulation parameters for Section 5.7

Parameter	*Value*
$\{k, c\}$	{3,4} and {4,6}
B	{256}
n	96
SNR	{5,10,15,20,25,30}
Hash Function	Lookup3
Puncturing	Active and Not Active

In Figures 5.11 and 5.12 we present the ratio of the achieved rate when puncturing is active to the rate when puncturing is not active, for $\{k, c\}$ pairs of {3,4} and {4,6} respectively. The results are found by averaging over 1000 simulations, similarly to the previous sections.

As indicated by the curves in Figures 5.11 and 5.12, the rate is increased with puncturing at least by 15% for $k = 4$ at SNR = 15 dB, and by 10% for $k = 3$ at SNR = 10 dB. It is interesting to see that puncturing is advantageous at small SNR values as well, as the 35-40% gains of both curves at SNR = 5 dB demonstrate. The maximum rate gains obtained build up to 85% for $k = 4$ and 170% for $k = 3$, both at SNR = 30 dB. Hence, we conclude that puncturing is a practical requirement that brings a lot of rate gain especially for large SNR values.

The local maximum point in Figure 5.12 is due to the low rate resolution of no puncturing scheme. For example at SNR = 10 dB with $k = 4$ and no puncturing, one can decode the message no earlier than the end of the 2nd pass. So, with no puncturing, one can achieve 2 bits/symbol at 10 dB, while with puncturing, because of the increased rate resolution, a much higher rate can be achieved.

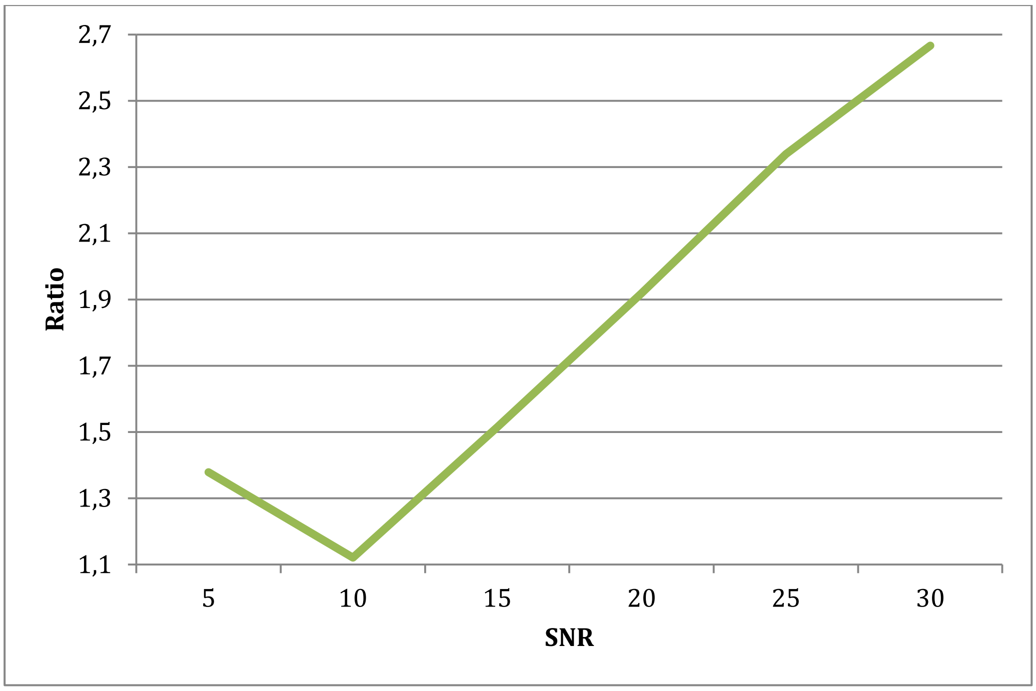


Figure 5.11 – Achieved rate ratio of ‘puncturing to no puncturing’, *k* = 3, *c* = 4

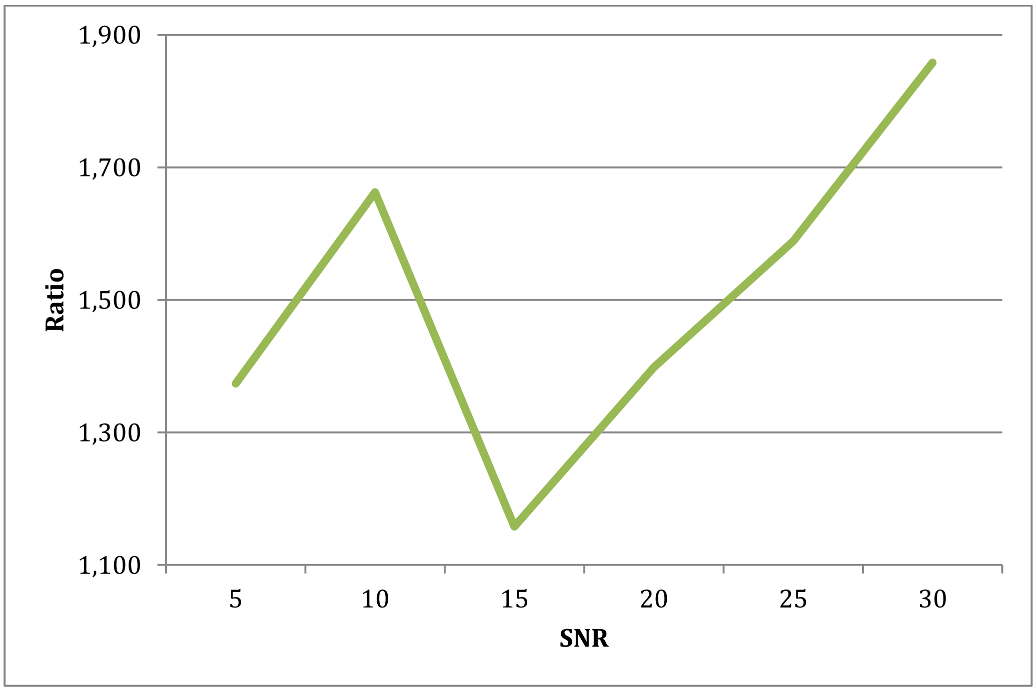


Figure 5.12 – Achieved rate ratio of ‘puncturing to no puncturing’, *k* =4, *c* = 6

5.8. Simulation Results of the Spinal Decoder that Uses Our Modified Algorithm at a Fixed SNR

In this section, we discuss the results obtained using our modified spinal decoder algorithm that is presented in Chapter 4. Simulations have been performed for SNR values between 0 and 30 dB in steps of 1 dB, with spinal code parameters $B_{max} = 256$, $d = 1$, $n = 96$, $c = 6$, and $k = 4$. The *cdf* parameter a can take values of {0.5, 0.6, 0.7, 0.8, 0.9 and 0.95}. The hash function is chosen to be the Lookup3 function and the puncturing is active. For each SNR and *cdf* pair, 1000 simulations are run for the decoder using our modified algorithm. For comparison, reference values of the achieved rate and number of operations per message are created using the bubble decoder with the beam-width value B chosen as 256.

In Figures 5.13 to 5.20, simulation outputs are plotted for various cases. Figure 5.13 gives the ratio of the number of operations made by our modified algorithm to the reference number of operations, per message. Since the modified algorithm uses different L values for each *cdf* value a, the ratio of the number of operations differs with a. More specifically, this ratio increases with increasing values of a, but does not exceed 0.7 even for $a = 0.95$.

In Figure 5.14, we plot the absolute number of operations per message with the reference decoder and with the modified algorithm using different values for the *cdf* parameter a.

Similarly Figure 5.15 gives the ratio of the achieved rate with the modified algorithm for the same set of a values, to the achieved rate of the reference bubble decoder.

Figure 5.16 is the detailed version of Figure 5.15 for better observation of the achieved rates with *cdf* values of 0.8, 0.9 and 0.95.

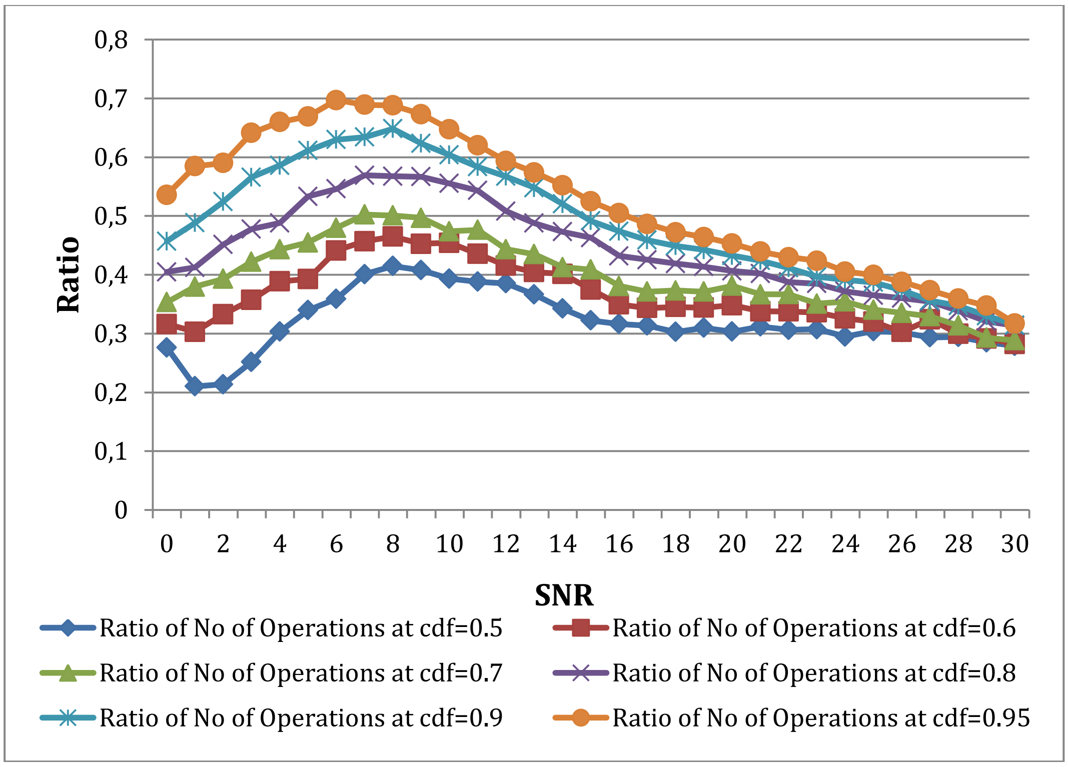


Figure 5.13 - Number of operations ratio of our modified algorithm to the reference decoder, with B_{max} =256.

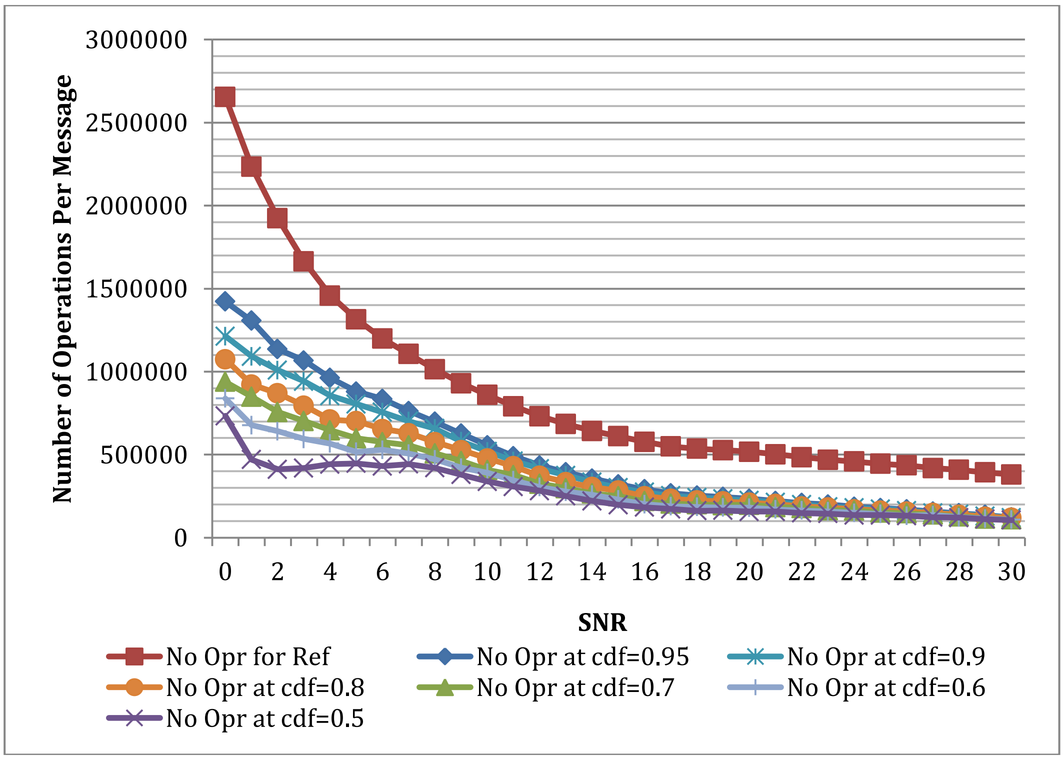


Figure 5.14 - Number of operations per message of our modified algorithm with different *cdf* values and that of the reference decoder, with B_{max} =256

Examining Figure 5.13, we see that for the mentioned SNR range, a reduction on the number of operations per message of at least 30% (for *a* = 0.95 and SNR = 6-7 dB) is possible by using the modified algorithm. This reduction in decoder complexity can be as high as 60% for most values of the SNR in the given range. An interesting observation is that, as the SNR increases beyond 25 dB, the ratio of the number of operations for different *cdf* values all approach to 0.3. This leads one to use larger *a* values for the *cdf,* since higher information rates can be reached with larger values of the *cdf*.

Another observation that can be concluded from Figure 5.13 is that as the SNR increases from 0 to 30 dB, the amount of reduction in the number of operations per message with respect to the reference bubble decoder, which is equal to '1 minus the plotted ratio'; makes a minimum around 6-9 dB and increases almost to 70% at 30 dB as mentioned above. By inspecting Figure 5.14, we can see that the number of operations made per message for the reference decoder decreases faster at SNR values between 0 and 6 dB. Conversely, one can say that the relatively slow decrease in the number of operations of our algorithm in 0-6 dB range is a result of its small initial values at 0 dB, which in turn confirms the power of our algorithm at the lowest SNR values. This difference between initial slopes also explains why the operation reduction capability of our algorithm experiences a minimum around 6-9 dB as mentioned in the previous paragraph, corresponding to the maxima of the curves in Figure 5.13.

We now present the results of simulations that show the ratio of the achieved rate of the modified algorithm to that of the bubble decoder on the given SNR range. We aimed an algorithm that reduces the decoder complexity, while being competitive with the reference decoder in terms of the achieved rate. As can be seen in Figure 5.15, the rate achieved with the modified algorithm for *cdf* values of especially 0.95, 0.9 and 0.8 is very competitive with the reference bubble decoder of **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]**. This effect can also be seen in a closer view in Figure 5.16, where some loss in achieved rate is observed for SNR's larger than 26 dB, only for the *cdf* value of 0.8.

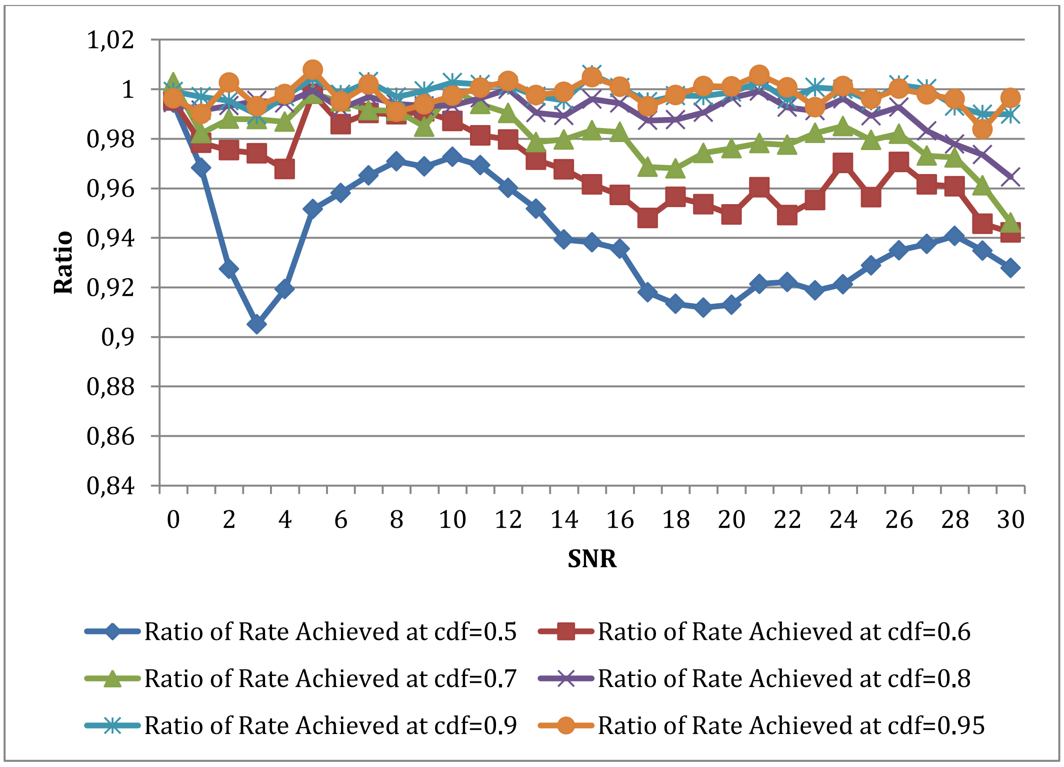


Figure 5.15 - Achieved rate ratio of our modified algorithm to the reference decoder, with B_{max} =256

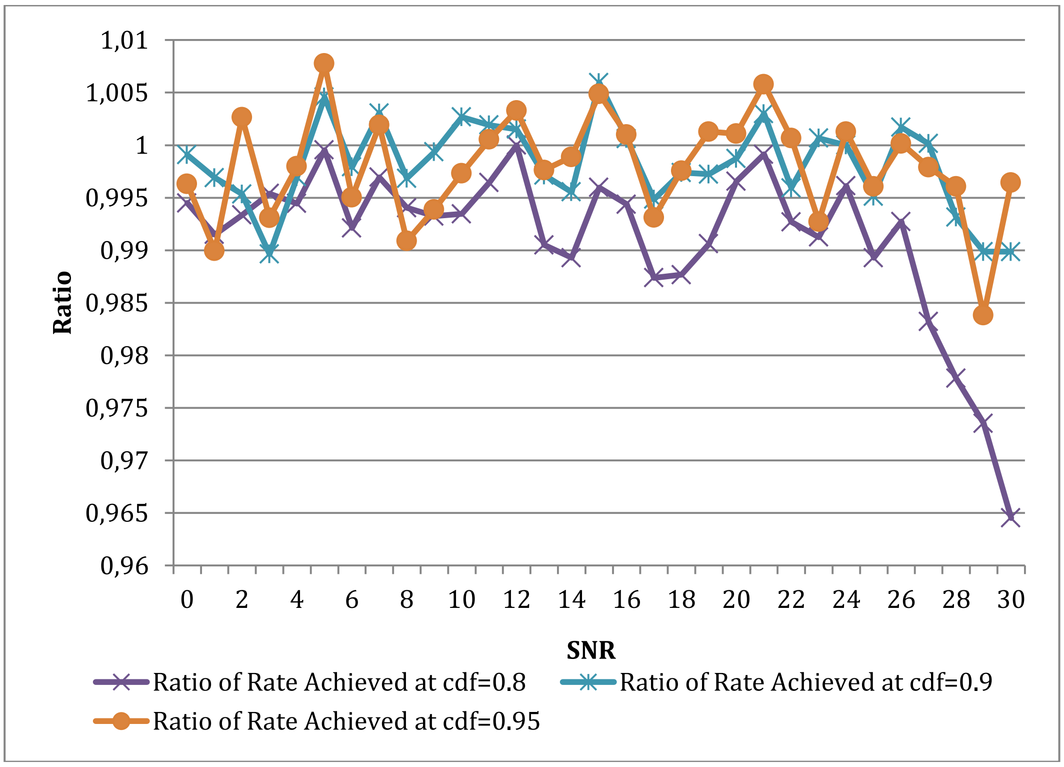


Figure 5.16 - Achieved rate ratio of our modified algorithm to the reference decoder – closer view, with B_{max} =256

It is also of interest to investigate the performance of our algorithm with a different value of the maximum beam-width parameter B_{max}. In the following, we present the simulation results for the decoder that uses the modified algorithm when B_{max} is selected as 16. The remaining parameters are chosen the same, namely d = 1, n = 96, c = 6 and k = 4. The hash function is the Lookup3 function and the puncturing is active. The outputs of the simulations are averaged over 1000 runs of the algorithm at each SNR.

Figure 5.17 gives the ratio of number of operations made on each run of our algorithm with different *cdf* values, to that of the reference decoder with B = 16, per message.

In Figure 5.18, we plot the absolute number of operations made per message by the reference decoder with B = 16, and by our algorithm with B_{max} = 16, using different values for the *cdf* parameter a.

Figure 5.19 gives the ratio of the achieved rate of our algorithm with B_{max} = 16, for different values of the *cdf* parameter a, to that of the reference bubble decoder with B = 16.

Lastly, Figure 5.20 is a detailed version of Figure 5.19, to better observe the achieved rates for the *cdf* values of 0.7, 0.8, 0.9 and 0.95.

Similar observations to the B_{max} = 256 case can be drawn for the B_{max} = 16 case from Figure 5.17. One can reduce the number of operations per message at least by 37% (see a = 0.95 curve), and an average reduction of 58% is possible in the given SNR range.

The difference from the B_{max} = 256 case is emphasized in Figure 5.20, where a wider range of the *cdf* values, namely a = 0.7, 0.8, 0.9 and 0.95, all prove very competitive rates with the reference decoder.

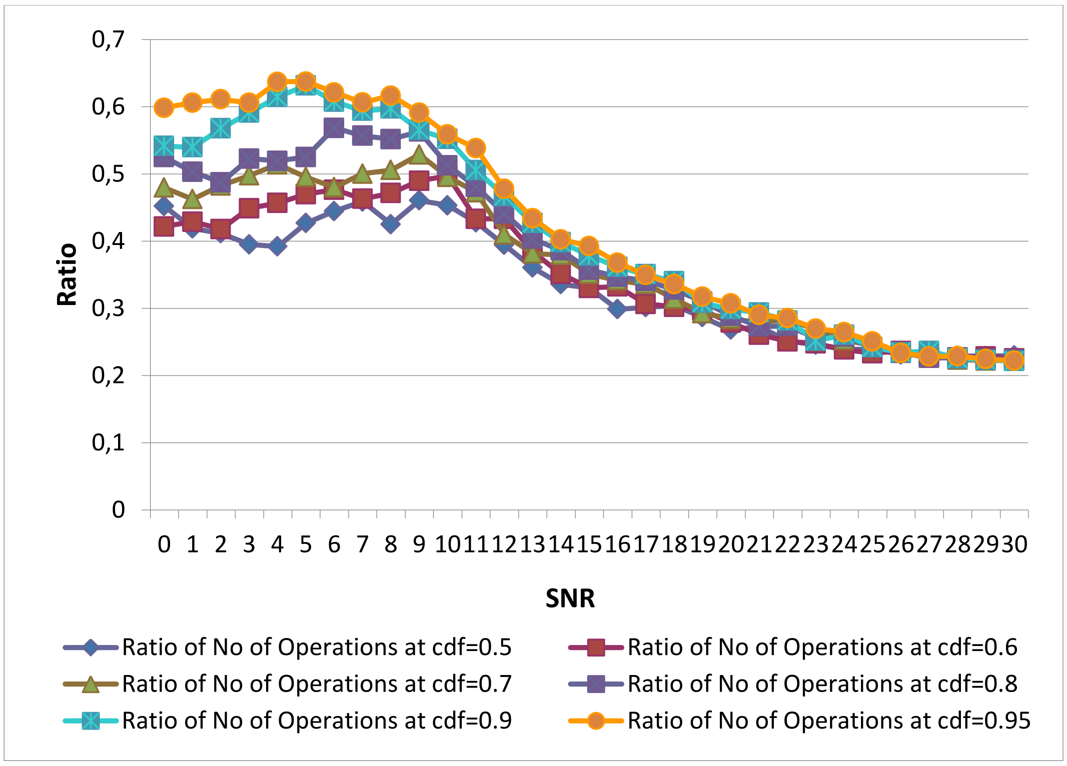


Figure 5.17 - Number of operations ratio of our modified algorithm to the reference decoder, with B_{max} =16

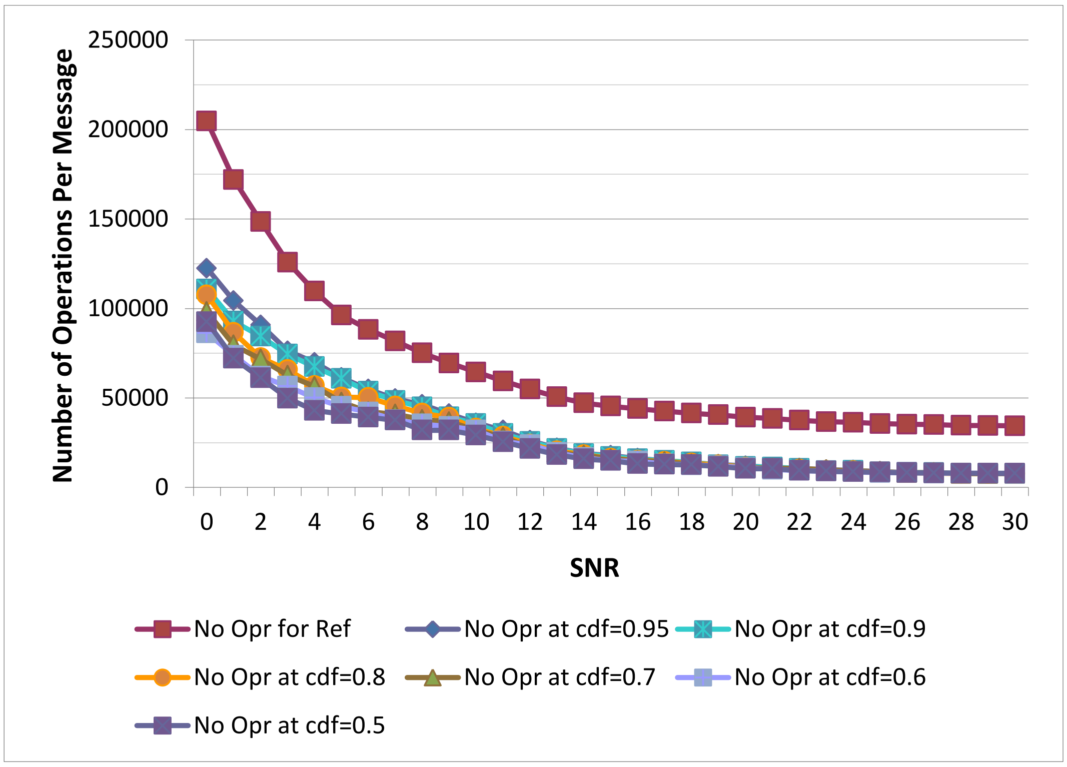


Figure 5.18 - Number of operations per message of our modified algorithm with different *cdf* values and that of the reference decoder, with B_{max} =16

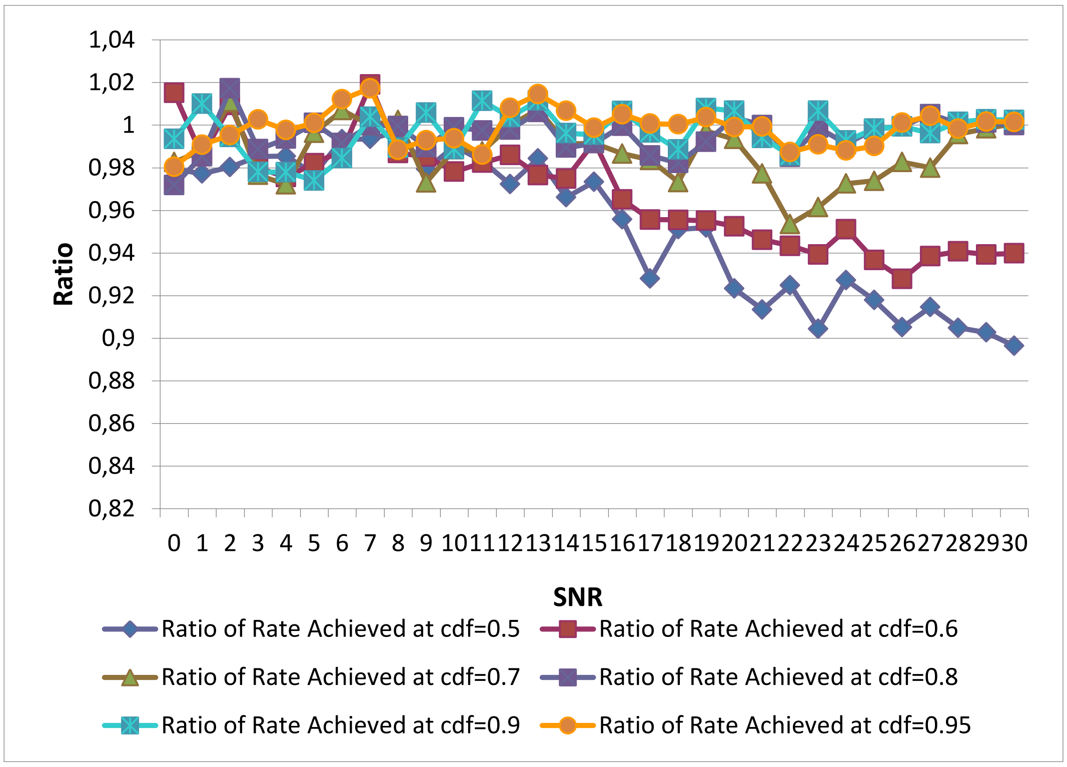


Figure 5.19 - Achieved rate ratio of our modified algorithm to the reference decoder, with B_{max} =16

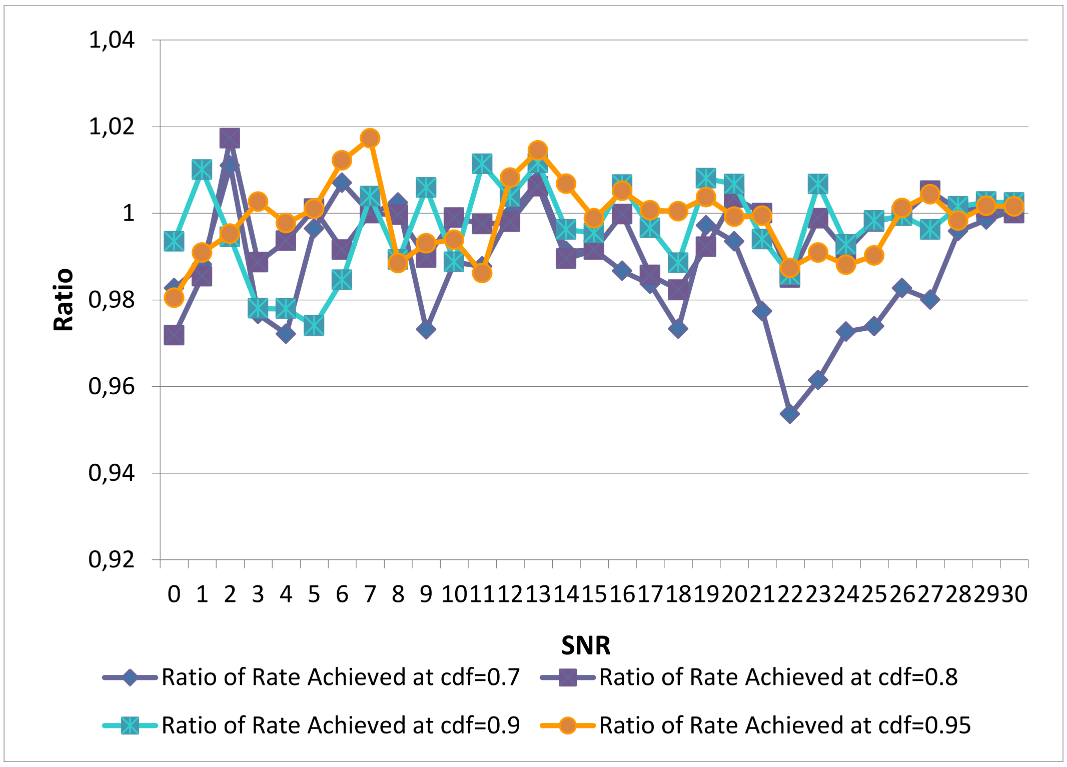


Figure 5.20 - Achieved rate ratio of our modified algorithm to the reference decoder – closer view, with B_{max} =16

In order to obtain a rough idea on the performance of our algorithm, we have averaged the number of operations and the achieved rate values over the specified SNR range. Tables 5.7 and 5.8 summarize the average values for each *cdf* value for the ratio of number of operations and the ratio of achieved rates respectively, when B_{max} is equal to 256. Note that all ratios are with respect to the reference bubble decoder given in **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** with the spinal code parameters mentioned in this section.

Table 5.7 – (Modified to reference) ratio of number of operations per message averaged over the SNR range of [0, 30] dB for B_{max} = 256

cdf	0.5	0.6	0.7	0.8	0.9	0.95
Ratio	0.319	0.363	0.396	0.443	0.486	0.522

Table 5.8– (Modified to reference) ratio of achieved rates averaged over the SNR range of [0, 30] dB for B_{max} =256

cdf	0.5	0.6	0.7	0.8	0.9	0.95
Ratio	0.940	0.968	0.981	0.991	0.998	0.998

Similarly, for B_{max} = 16, we present the average number of operations and achieved rate values over the given SNR range in Tables 5.9 and 5.10.

Table 5.9 – (Modified to reference) ratio of number of operations per message averaged over the SNR range of [0, 30] dB for B_{max} = 16

cdf	0.5	0.6	0.7	0.8	0.9	0.95
Ratio	0.336	0.350	0.370	0.385	0.410	0.423

Table 5.10– (Modified to reference) ratio of achieved rates averaged over the SNR range of [0, 30] dB for B_{max} =16

cdf	0.5	0.6	0.7	0.8	0.9	0.95
Ratio	0.953	0.967	0.987	0.995	0.997	0.998

Tables 5.7 and 5.8 given for B_{max} = 256 show that an average reduction; i.e., (1−ratio), of 52% in the number of operations per message is possible with an average 0.2% loss in the achieved rate, by using the *cdf* value of 0.9. One can gain more reduction in decoder's computational effort by using smaller values of the *cdf* parameter *a*; i.e., by pruning more possible paths when traversing the decoding tree. At the *cdf* value of 0.5, we can get 68% reduction in computations made per message while sacrificing only 6% of the achieved rate with respect to the reference bubble decoder.

Similar observations can be concluded inspecting Tables 5.9 and 5.10. For B_{max} = 16, an average reduction of 58% in the number of operations per message is possible with an average 0.2% loss in the achieved rate, by using the *cdf* value of 0.95. The highest reduction in decoder complexity is obtained with the smallest *cdf* parameter in our simulation set; namely, *a* = 0.5 case provides 67% reduction in the computational load, while losing 4.7% of the rate achieved with the bubble decoder.

Finally, we can compare our results with a related work that aims to decrease the computational load of the decoder for graph-based rateless codes **[Harrison-Jamieson-2012]**. Our work focuses on rateless spinal codes, where we propose a modified decoding algorithm to significantly reduce the decoder complexity, while Harrison et al. present a simpler decoding algorithm for graph-based codes. They compare the performance of the decoding algorithm for graph-based codes to that of spinal codes. The results of **[Harrison-Jamieson-2012]** show complexity reductions of the message decoding cost of graph-based rateless codes compared to spinal codes for SNR values larger than 20 dB, reaching an order of magnitude reduction for SNR values above 25 dB, while for smaller SNR values the spinal codes have a lower message decoding complexity. Our modified algorithm offers a complexity reduction on the order of 40-50% average for the entire range of SNR values from 0 to 30 dB, while achieving the same rate values with the spinal codes.

5.9 Simulation Results of the Modified Spinal Decoder for Changing SNR Conditions

In this section, we compare performances of the reference bubble decoder with our modified decoding algorithm when the SNR changes abruptly from one level to another during the transmission. We run simulations on a simple channel model such that during a portion of the transmission time denoted by *t* subpasses, the SNR is constant at SNR_1; then it changes to a different value denoted by SNR_2 and remains constant until the end of the transmission. We have considered two cases with $\{SNR_1, SNR_2\} = \{20, 10\}$ dB and $\{SNR_1, SNR_2\} = \{10, 20\}$ dB, for several *t* values. The other parameters of spinal encoder and decoder are as given in Table 5.11.

Table 5.11 – Simulation parameters for Section 5.9

Parameter	*Value*
$\{k, c\}$	{4,6}
B	256
n	96
Hash Function	Lookup3
Puncturing	Active
t	{3,4,5,6,7,8,9}

The achieved rates and the number of operations made per message metrics are averaged over 1000 simulation runs. For the first set of runs, SNR_1 = 20 dB in the first *t* subpasses, and then SNR_2 = 10 dB. With *cdf* values changing from 0.5 to 0.95, the achieved rate ratio and the number of operations made per message ratio are given respectively in Figures 5.21 and 5.22, with respect to the bubble decoder for *t* values changing between 3 and 9, which are chosen in this range since the starting subpasses obtained from Table 4.1 are *K*(20dB)=5 and *K*(10dB)=8. We observe that, for *cdf* values of 0.8 and above, the average achieved rate ratios with respect to the bubble decoder for the wide range of *t* values mentioned above is larger than 99%. The corresponding number of operations made per message ratios with respect to the bubble decoder can be as low as 42% (for a *cdf* value equal to 0.8).

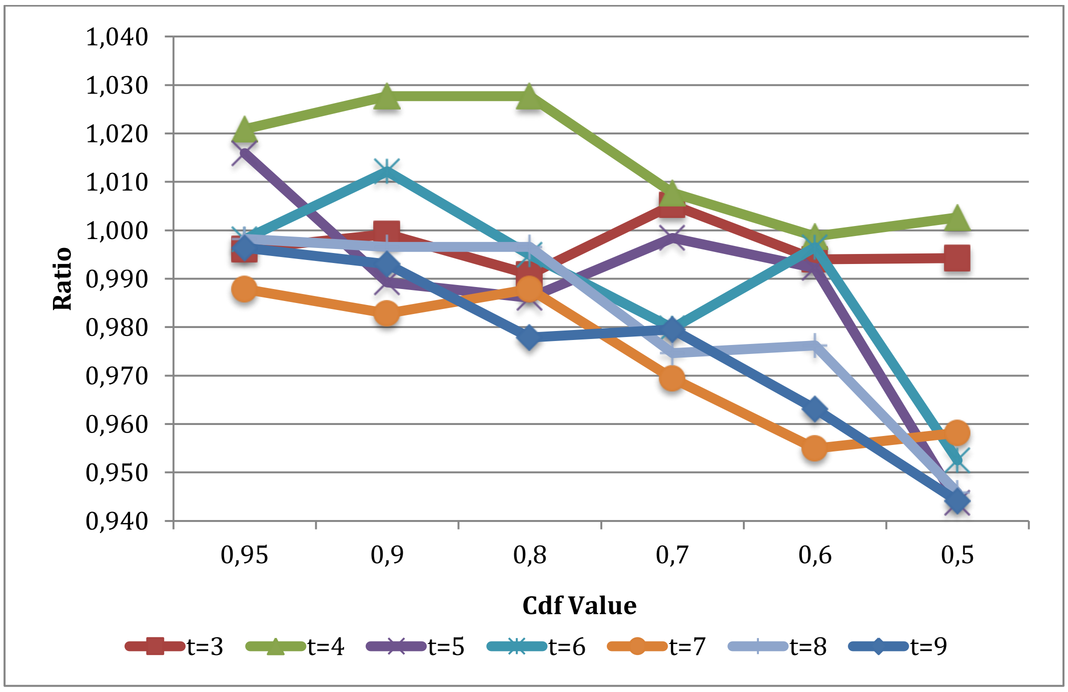


Figure 5.21 - Achieved rate ratio of modified / reference decoders for an abrupt change from SNR_1 = 20 dB in the first *t* subpasses to SNR_2 = 10 dB

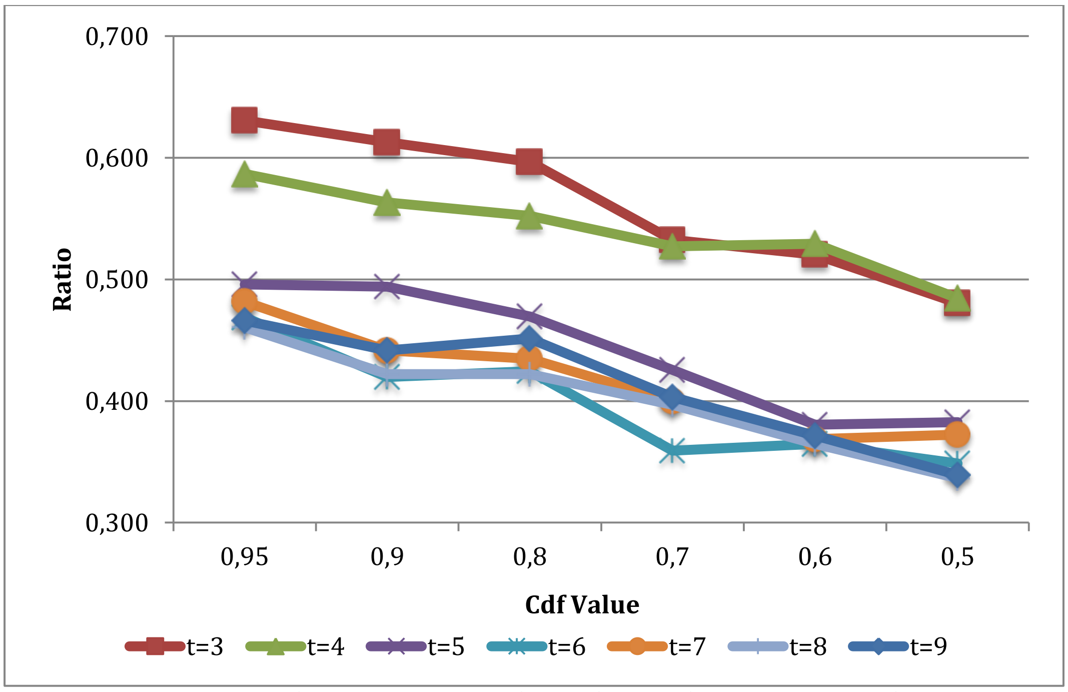


Figure 5.22 - Number of operations ratio of modified / reference decoders for an abrupt change from SNR_1 = 20 dB in the first *t* subpasses to SNR_2 = 10 dB

The results of $\{SNR_1, SNR_2\} = \{10, 20\}$ dB with the same parameters as in Table 5.11 are given in Figures 5.23 and 5.24 for the achieved rate ratio and number of operations made per message ratio respectively, as compared to the bubble decoder.

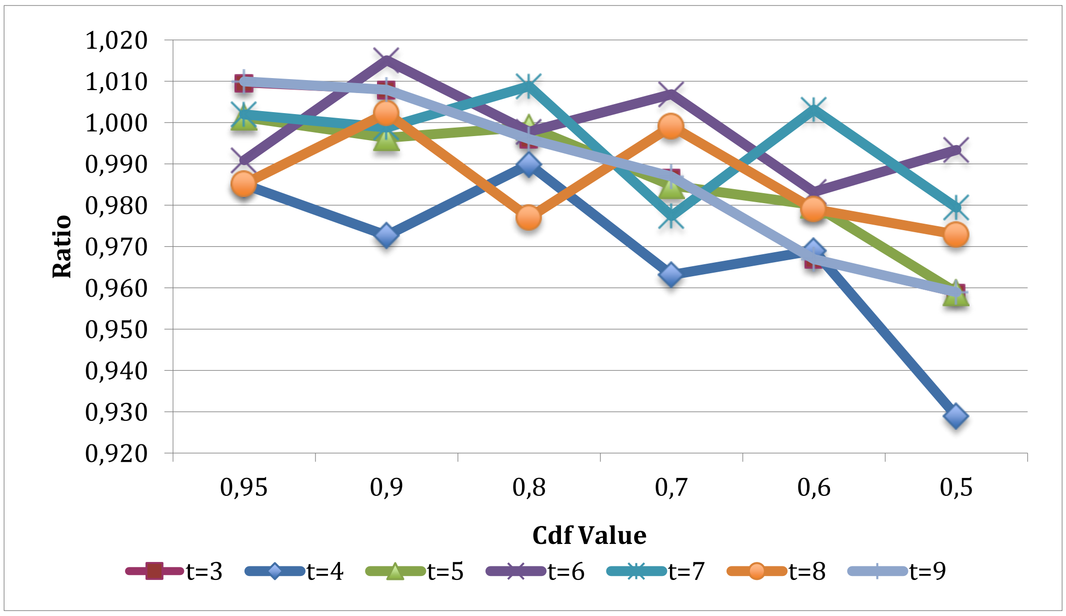


Figure 5.23 - Achieved rate ratio of modified / reference decoders for an abrupt change from $SNR_1 = 10$ dB in the first t subpasses to $SNR_2 = 20$ dB

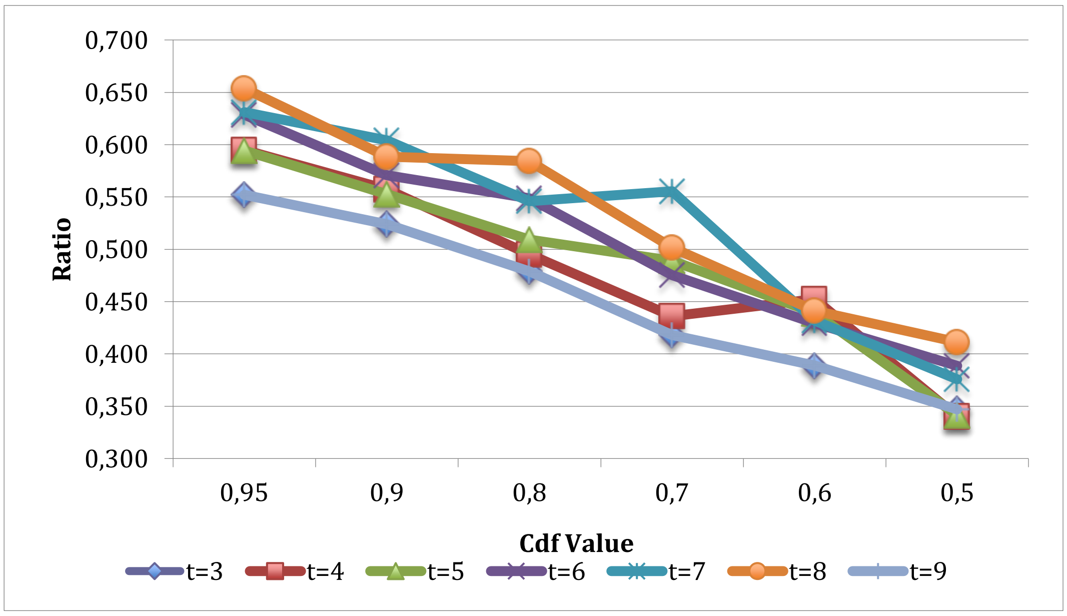


Figure 5.24 - Number of operations ratio of modified / reference decoders for an abrupt change from $SNR_1 = 10$ dB in the first t subpasses to $SNR_2 = 20$ dB

Figures 5.23 and 5.24 show that the case of an abrupt rise from 10 dB to 20 dB gives very similar results to the previous case of an abrupt fall from 20 dB to 10 dB: For *cdf* values of 0.8 and above, the average achieved rate ratios with respect to the bubble decoder for the wide range of *t* values mentioned above is larger than 99%. The corresponding number of operations made per message ratios with respect to the bubble decoder for these *cdf* values can be as low as 48% (for *cdf* = 0.8).

By comparing the results of this section with Tables 5.7 and 5.8 obtained for the case of a fixed SNR during transmission, which indicate an average ratio of 48.6% in the number of operations per message with an average ratio of 99.8% in the achieved rate (for *cdf* =0.9); we conclude that the performance of our modified decoding algorithm is not affected negatively by the presence of an abrupt SNR change. Comparison of this section's results with Table 5.7 may look somewhat unfair since Table 5.7 is obtained by averaging 31 different simulation results obtained for the 1dB-separated SNR values in the range [0, 30] dB. However, the same result is justified in Figure 5.13 as well, where the modified/reference ratio of the number of operations is approximately 60% and 43% for 10 dB and 20 dB respectively (at *cdf* =0.9).

CHAPTER 6

CONCLUSIONS

In this thesis, the performance of rateless spinal codes over an additive white Gaussian noise channel is investigated. Since the decoding complexity of the spinal decoder is high, we have designed an algorithm to reduce the computational load of the decoder. We have made software simulations in order to explore the effects of the code parameters on the performance of the spinal codes and also for evaluating the advantages of our decoding algorithm over the bubble decoder.

Before the performance assessment of our algorithm, we have investigated the effects of parameters *c, k, B, n*; in addition to the consequences of using different hash functions and puncturing on the performance of the original bubble decoder for spinal codes. The simulations are run for the SNR values of 5 to 30 dB in 5 dB steps. Performance is measured in terms of the number of operations per message made by the decoder and the achieved rate, averaged over 1000 simulation runs.

- For the parameter *c* that determines the constellation size, we have found that it should be somewhat larger than *k*, which is the length of the message segments. It seems that $c = 6$ when $k = 4$, and $c = 4$ when $k = 3$, are appropriate choices for the given SNR range.

- For the message length parameter *k* chosen from the set {2, 3, 4}, considering the results presented in Sections 5.3 and 5.5, $k = 4$ is the best choice in the given SNR range. Higher values of *k* are not considered in order to keep the decoding tree complexity at a reasonable level.

- For the beam-width parameter *B*, if the application is over the full SNR range of [0, 30] dB, it is best to use at least *B* = 256 so that the rate is not limited. However, if the application is supposed to work at bad channel conditions all the time, smaller values of *B* can be chosen.

- For the parameter *n*, which is the total message length of one pass, it is reasonable to choose *n* = 192 for *k* = 4, and *n* = 96 is for *k* = 3.

- For the hash functions, we state that using the MD4 hash function instead of the Lookup3 hash function provides an average 7% gain in the achieved rate; however, the decoder's computational work is increased as well.

- For the puncturing, we find that it is essential especially for large SNR values. However, it also provides gains in the achieved rate at worse channel conditions.

Afterwards, the results obtained by our modified decoder algorithm proposed in Chapter 4 are presented and compared with those of the bubble decoder. Our algorithm guesses the channel SNR and the corresponding successful decoding time by exploiting the pre-computed distributions of the best path metrics at the end of each subpass for all SNR's. This information is used for increasing the value of *B* whenever necessary. In other words, our algorithm starts with a beam-width value *B*=1 and increases *B* only when the decoder is close to "successful decoding".

LimitValue's obtained for each SNR value in the range [0,30] dB for each subpass are used to estimate the channel SNR, while deciding whether to increase the value of *B* or not. *LimitValue* lists are prepared by using the metric distributions collected offline. To estimate the channel SNR, we compare the best path metric at the end of each subpass with the *LimitValue*'s of different

SNR's belonging to that subpass. We then predict the SNR as approximately the value that has the closest *LimitValue* larger than the obtained best path metric.

In our simulations, the performance of our decoding algorithm that keeps the beam-width at B=1 and changes it to $B=B_{max}$, is compared with that of the bubble decoder constantly working at $B=B_{max}$. Performance is measured in terms of two ratios; namely, the modified / reference decoder's ratio of 1) the number of operations per message, 2) the achieved rate. Each measurement is made by averaging over 1000 simulation runs.

We have found that our decoder provides a reduction of 52% for B_{max} =256, and a reduction of 58% for B_{max} =16 in the number of operations per message, with nearly the same achieved rates. More precisely, the achieved rate of our decoder is within 0.2% of the rate achieved by the reference bubble decoder. Another conclusion that can be drawn is that, one can use lower values (for example 0.5) for the *cdf* parameter a for less energy expenditure, with some loss in the achieved rate that can be tolerable in some applications. The loss in the achieved rate is 6% and 4.7%, whereas the reduction in the number of operations is 68% and 67% for B_{max} =256 and B_{max} =16 respectively, if the *cdf* parameter a is chosen as 0.5. The choice of the *cdf* value a depends on the specific application and it can be selected among different *cdf* values ranging from 0.5 to 0.95, to fulfill needs of the user.

We also conclude that using our algorithm, one can achieve a reduction of 70% in the number of operations per message made by the decoder as the SNR increases to 30 dB regardless of the value of the *cdf* parameter a when B_{max} =256. The same characteristic is also seen when B_{max} =16, the reduction with increasing SNR converges to 78% independent of the value used for the *cdf* parameter a.

We have also run simulations in case of an abrupt SNR increase or decrease by 10 dB during the transmission. We have compared the performance of our decoding algorithm to the reference bubble decoder and concluded that our algorithm is resistant against changing channel conditions. Even when the SNR changes abruptly during the transmission of the message, our decoder saves a considerable amount of decoder computational resources that reaches 60% with negligible loss in the achieved rate values.

As future research directions, the performance of the proposed decoder on fading channels can be investigated. To implement the proposed decoder in hardware and to test its performance in real-life scenarios is another challenging idea.

REFERENCES

- **[Anderson-Mohan-1984]** J. Anderson and S. Mohan. Sequential coding algorithms: A survey and cost analysis. IEEE Trans. on Comm., 32(2):169–176, 1984.
- **[Balakrishnan-Iannucci-Perry-Shah-2012]** H. Balakrishnan, P. Iannucci, J. Perry, and D. Shah. De- randomizing Shannon: The Design and Analysis of a Capacity-Achieving Rateless Code, June 2012.
- **[Berrou-Glavieux-Thitimajshima-1993]** C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon limit error correcting coding and decoding: Turbo-codes," in *Proc. 1993 IEEE Int. Conf. on Communications* (Geneva, Switzerland), pp. 1064–1070, 1993.
- **[Bose-Chaudhuri-1960]** R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error-correcting binary group codes," Information and Control, vol. 3, no. 1, pp. 68–69, 1960.
- **[Elias-1955]** P. Elias, "Coding for Noisy Channels", The 3rd London Symposium, pp. 61-76, Sep. 1955.
- **[Erez-Trott-Wornell-2012]** Erez, U. and Trott, M. and Wornell, G. Rateless Coding for Gaussian Channels. IEEE Trans. Info. Theory, 58(2):530–547, 2012.
- **[Gallager-1962]** R. G. Gallager, "Low density parity check codes," *IRE Trans. Inform. Theory,* vol. IT-8, pp. 21–28, Jan. 1962.
- **[Goldsmith-2004]** A. Goldsmith, Wireless Communications, 2004.

- **[Gudipati-Katti-2011]** A. Gudipati and S. Katti. Strider: Automatic rate adaptation and collision handling. In *SIGCOMM*, 2011.
- **[Hamming-1950]** R. W. Hamming, “Error detecting and error correcting codes,” Bell System Technical Journal, vol. 29, pp. 147–160, Apr. 1950.
- **[Harrison-Jamieson-2012]** C. Harrison, K. Jamieson, Power-Aware Rateless Codes in Mobile Wireless Communication, *Hotnets’12,* October 29–30, 2012, Seattle, WA, USA.
- **[Hocquenghem-1959]** A. Hocquenghem, “Codes Correcteurs D’erreurs”, Chiffres 2, pp. 147-156, 1959.
- **[Lott-Milenkovic-Soljanin-2007]** C. Lott, O. Milenkovic and E. Soljanin, Hybrid ARQ: Theory, State of the Art and Future Directions, 2007.
- **[Luby-2002]** M. Luby, “LT codes,” in Proc. IEEE Symp. Found. Comp. Sci., Vancouver, pp. 271–280, Nov. 2002.
- **[Mackay-2005]** D.J.C. Mackay, Fountain Codes, IEE Proc.-Commun., vol. 152, No. 6, pp. 1062-1068, December 2005.
- **[Massey-1963]** J.L. Massey, “Threshold Decoding”, M.I.T. Press, Cambridge, MA, 1963.
- **[Maymounkov-2002]** P. Maymounkov, “Online codes", NYU Technical Report TR2003-883, Nov 2002.
- **[Palanki-Yedidia-2004]** R. Palanki and J. Yedidia, “Rateless codes on noisy channels,” in Proc. Int. Symp. Inform. Theory, p. 37, 2004.

- **[Perry-Iannucci-Fleming-Balakrishnan-Shah-2012]** J .Perry, P. Iannucci, K. Fleming, H. Balakrishnan, and D. Shah, “Spinal Codes”, In SIGCOMM, Aug. 2012.
- **[Proakis-1995]** J.G. Proakis, “Digital Communications. 3rd Ed.” New York: McGraw-Hill, 1995.
- **[Reed-Solomon-1960]** I. S. Reed and G. Solomon, “Polynomial codes over certain finite fields,” Journal of the Society for Industrial and Applied Mathematics (SIAM), vol. 8, no. 2, pp. 300–304, June 1960.
- **[Shannon-1948]** C. E. Shannon, “A mathematical theory of communication," Bell System Technical Journal, vol. 27, pp. 379-423, 1948.
- **[Shokrollahi-2006]** A. Shokrollahi, “Raptor codes,” IEEE Trans. Inform. Theory, vol. 52, no. 6, pp. 2551–2567, 2006.
- **[Shokrollahi-Luby-2009]** A. Shokrollahi, M. Luby, “Raptor Codes”, Foundations and Trends in Communications and Information Theory, vol. 6, No. 3–4, pp. 213–322, 2009.
- **[Viterbi-1967]** A. J. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *IEEE Trans. Inform. Theory*, vol. IT-13, pp. 260–269, 1967.
- **[Wozencraft-1957]** Wozencraft, J., “Sequential decoding for reliable communication”, IRE Natl. Convention rec., 5(2), 11–25, 1957.

APPENDIX A

CDF PLOTS OF SUBPASS 1-6 PATH METRIC VALUES FOR *B*=1 AT VARIOUS SNR'S

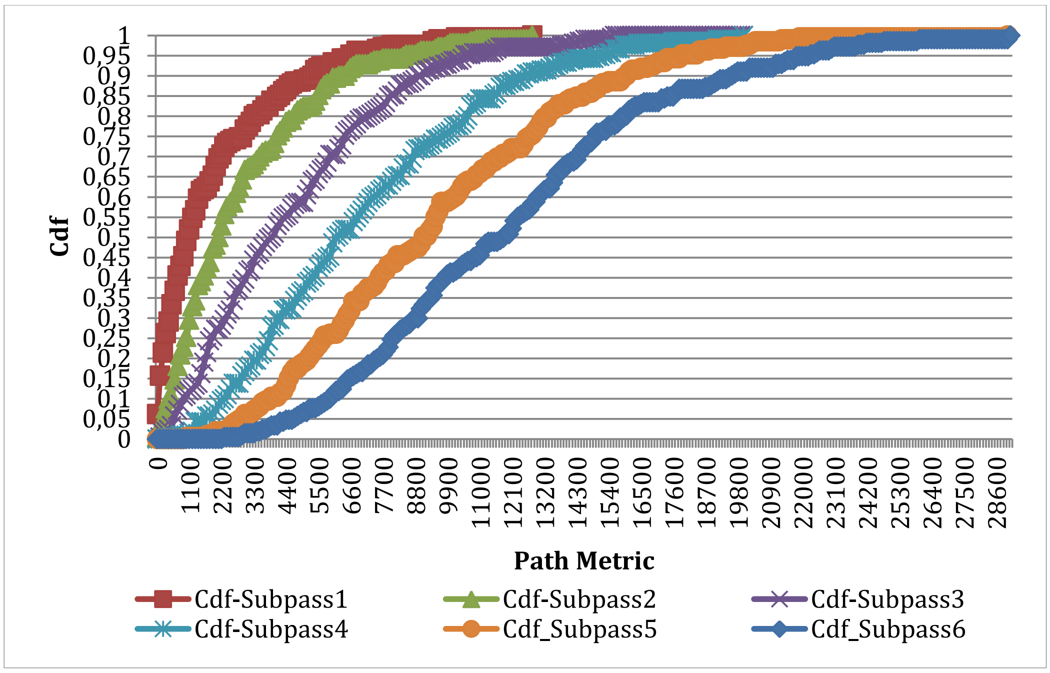


Figure A.1 – Path metric *cdf*'s in Subpasses 1 to 6 for SNR = 0 dB

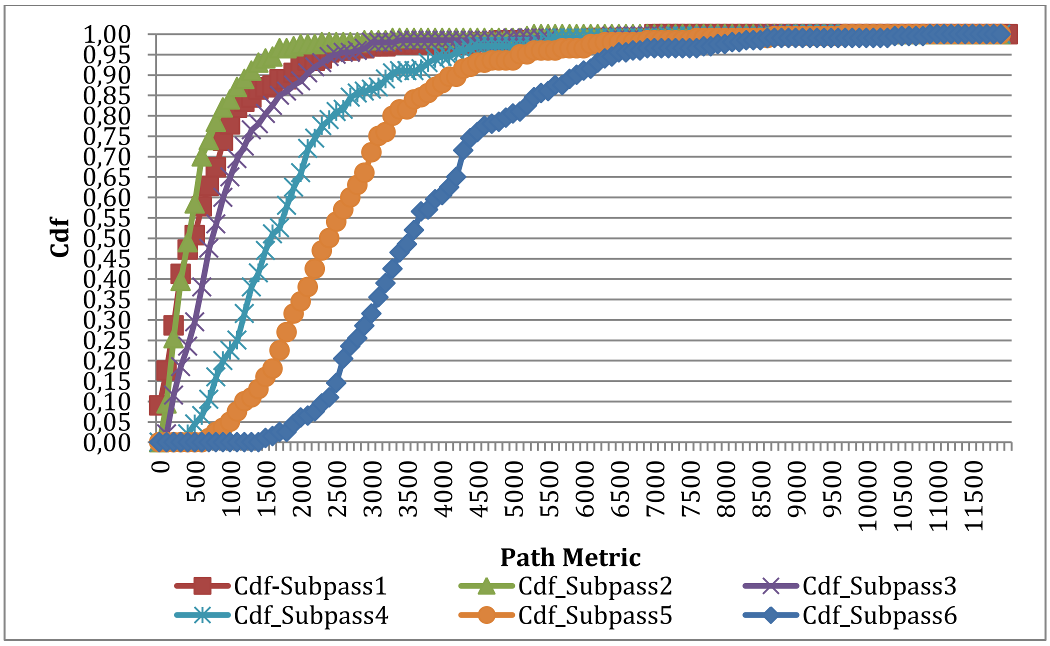


Figure A.2 – Path metric *cdf*'s in Subpasses 1 to 6 for SNR = 5 dB

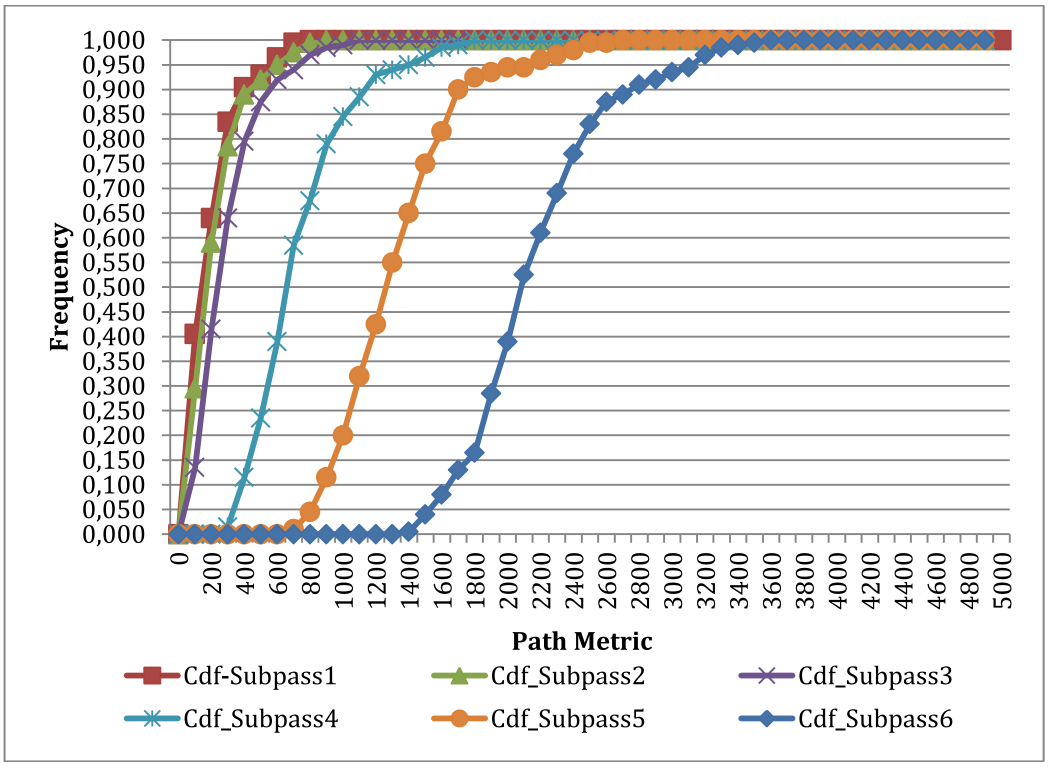


Figure A.3 – Path metric *cdf*'s in Subpasses 1 to 6 for SNR = 10 dB

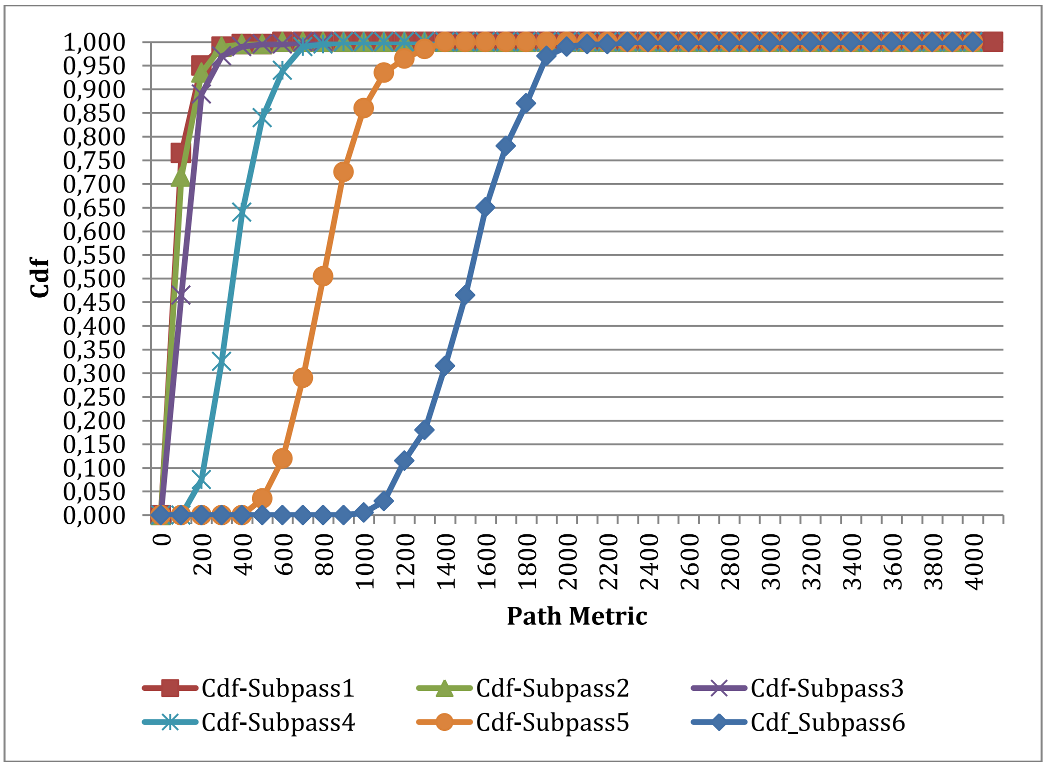


Figure A.4 – Path metric *cdf*'s in Subpasses 1 to 6 for SNR = 15 dB

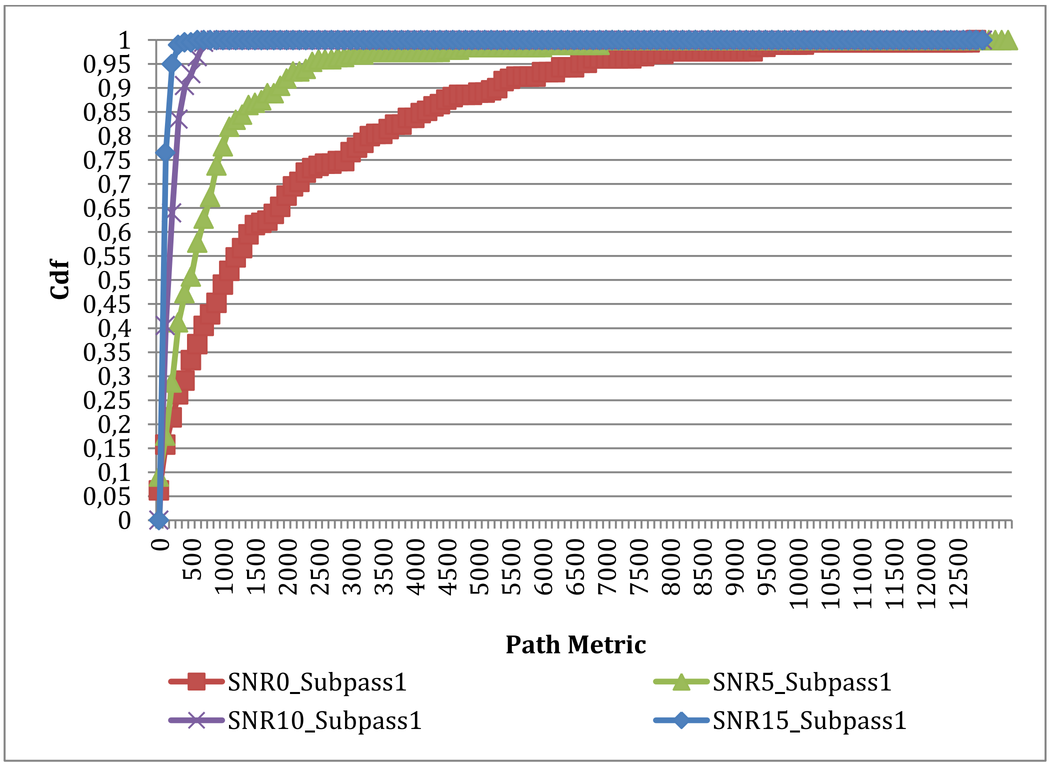


Figure A.5 – Path metric *cdf'*s at {0, 5, 10, 15} dB for Subpass 1

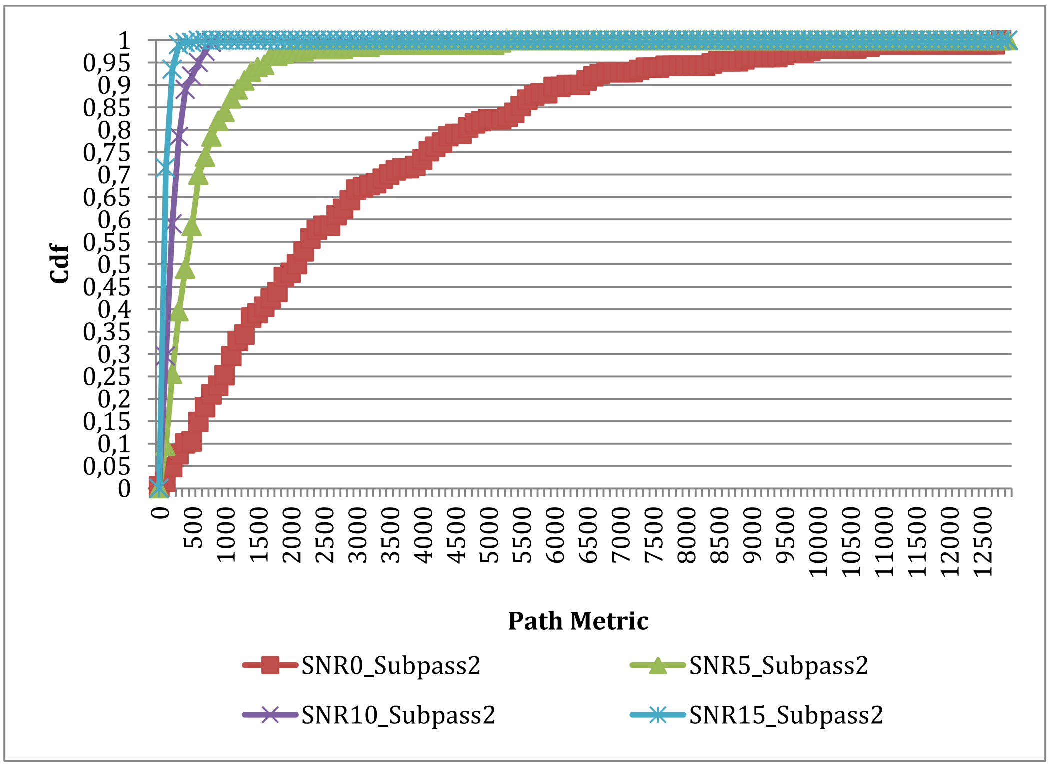


Figure A.6 – Path metric *cdf'*s at {0, 5, 10, 15} dB for Subpass 2

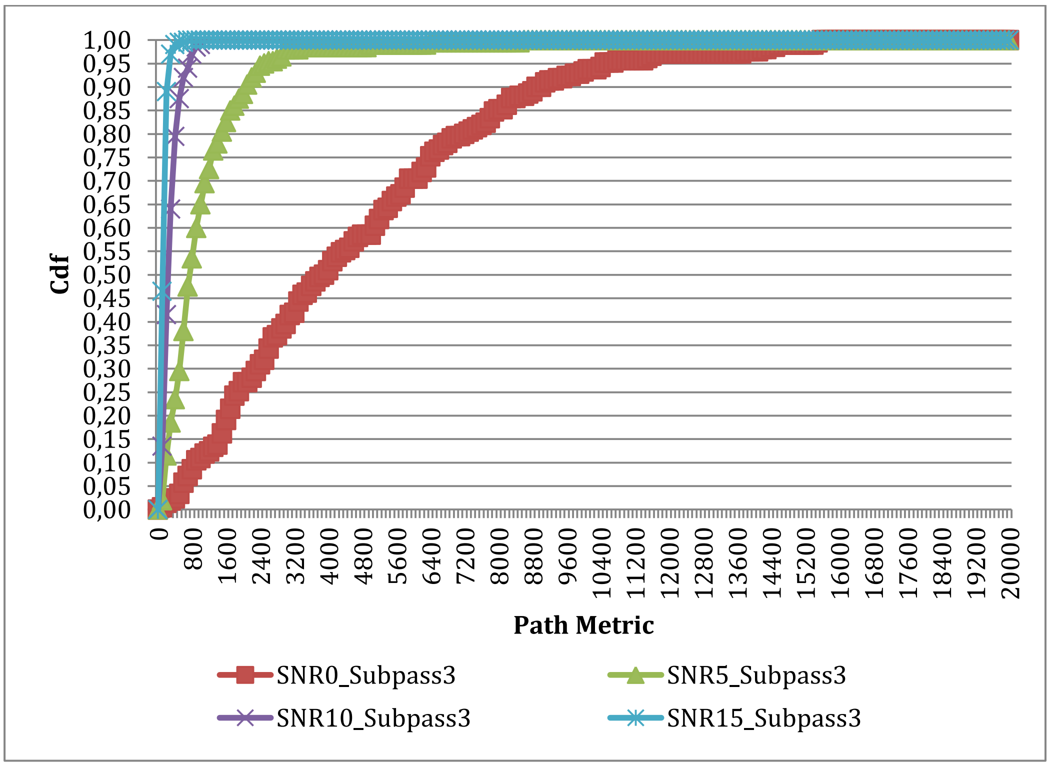


Figure A.7 – Path metric *cdf'*s at {0, 5, 10, 15} dB for Subpass 3

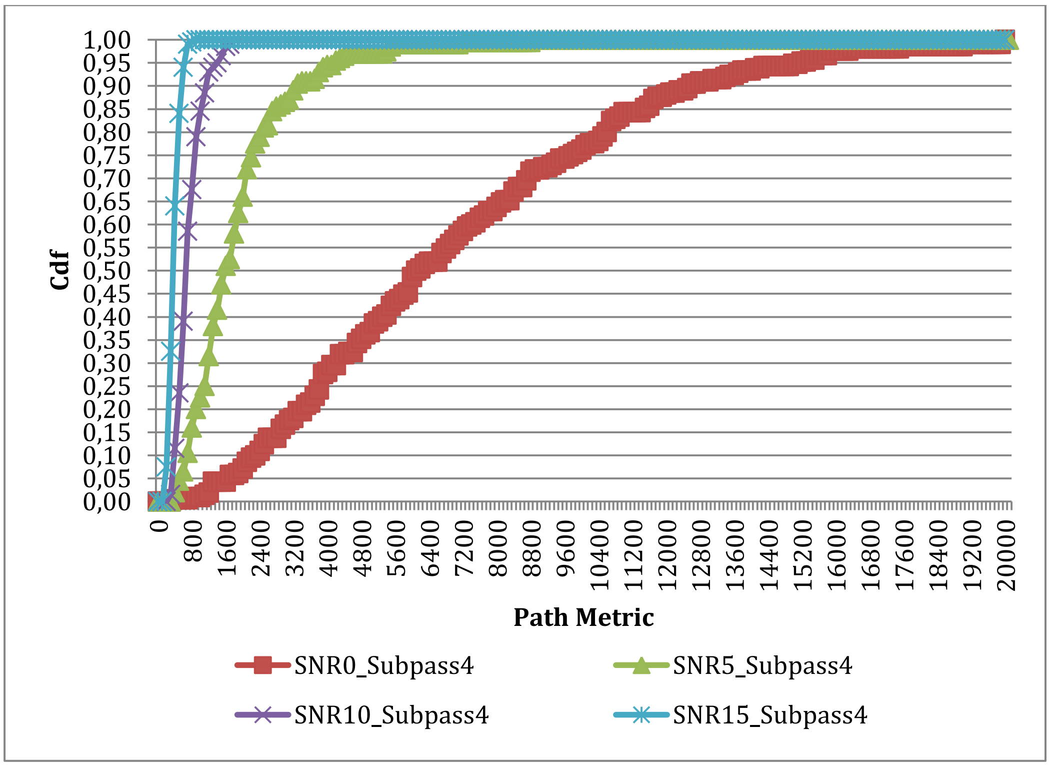


Figure A.8 – Path metric *cdf'*s at {0, 5, 10, 15} dB for Subpass 4

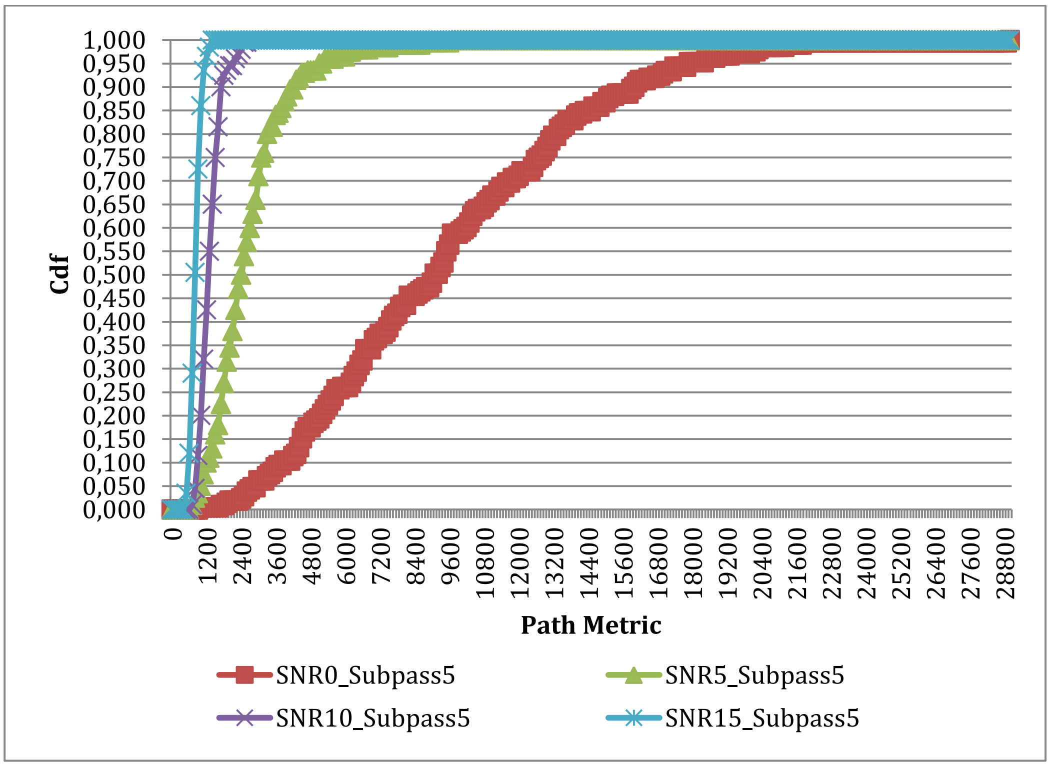


Figure A.9 – Path metric *cdf*'s at {0, 5, 10, 15} dB for Subpass 5

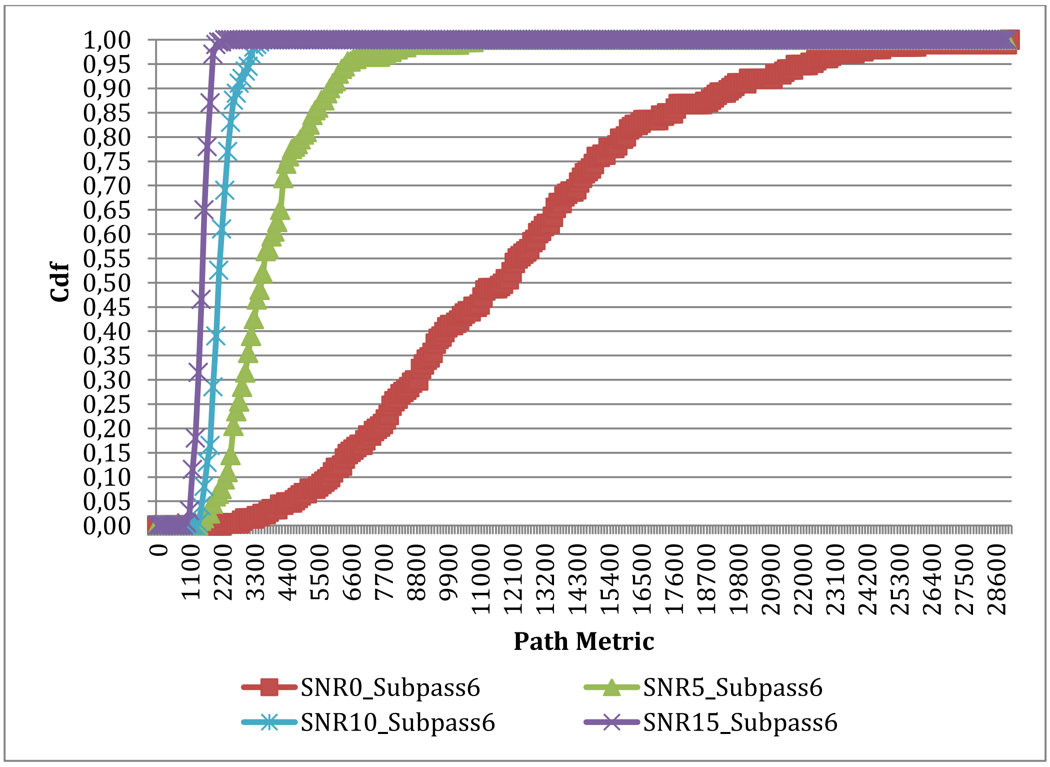


Figure A.10 – Path metric *cdf*'s at {0, 5, 10, 15} dB for Subpass 6

APPENDIX B

LIMIT VALUES OF SUBPASS 1-6 PATH METRIC VALUES FOR *B*=1 AT VARIOUS SNR'S

Table B.1 – Limit values for our modified algorithm over the SNR range of [0, 30] dB for subpasses 1 to 6 when the path metric *cdf* reaches *a* = 0.95

***a* = 0.95**	Subpass1	Subpass2	Subpass3	Subpass4	Subpass5	Subpass6
SNR0	8422	16475	20008	25850	31946	36364
SNR1	8214	12868	16679	22877	25459	30041
SNR2	6128	10637	14134	19387	21602	25273
SNR3	4831	8433	11582	15169	17329	21775
SNR4	4119	7121	10384	14195	16050	18914
SNR5	3212	6042	9969	11539	13379	16238
SNR6	2917	5532	8802	11431	12539	14264
SNR7	2469	4805	8199	10446	11143	13776
SNR8	2112	4436	7524	9593	11232	12649
SNR9	1901	3863	6637	8796	9934	12313
SNR10	1621	3987	6778	8785	9818	11658
SNR11	1478	3703	6752	8007	9138	11322
SNR12	1324	3538	6469	8151	9089	10725
SNR13	1231	3369	6384	7433	8524	10061
SNR14	1157	3244	6011	7295	8533	9574
SNR15	1041	3123	5840	7017	8491	9262
SNR16	933	3228	5627	7794	8411	8788
SNR17	944	3044	5455	7851	7883	9181
SNR18	878	3072	5350	7681	7130	8702
SNR19	834	2956	5349	7278	7121	8144
SNR20	811	3006	5449	6767	7420	7601
SNR21	763	3107	5229	6839	7003	7392
SNR22	751	3000	4640	6174	6934	7875
SNR23	719	3109	5167	6033	6844	7561
SNR24	739	2939	4654	5746	6264	7257
SNR25	719	2994	4562	5359	6241	6986
SNR26	710	3019	3880	3817	6037	6033
SNR27	713	2790	3980	3800	6212	5601
SNR28	714	2754	3797	3796	6173	4984
SNR29	672	2753	3734	3422	6150	4935
SNR30	685	2782	3838	3543	5930	4825

Table B.2– Limit values for our modified algorithm over the SNR range of [0, 15] dB for subpasses 1 to 6 when the path metric *cdf* reaches *a* = 0.9

***a* = 0.9**	Subpass1	Subpass2	Subpass3	Subpass4	Subpass5	Subpass6
SNR0	7006	13753	17661	23657	26965	33336
SNR1	5851	10516	15154	19254	21474	25771
SNR2	4758	8496	13009	16421	18557	21700
SNR3	3631	6941	10517	13804	15952	18684
SNR4	3592	5921	9350	12771	14463	15761
SNR5	2361	4837	8315	10596	12051	15023
SNR6	2310	4542	7475	9350	10450	13084
SNR7	1926	3931	6944	8879	9713	12757
SNR8	1645	3716	6921	8004	9183	11697
SNR9	1585	3435	6028	7974	8322	10424
SNR10	1291	3315	6279	7355	7805	10833
SNR11	1197	3359	5703	7347	7799	9919
SNR12	1155	2968	5469	6797	7924	9888
SNR13	1100	3062	5817	6814	7073	9213
SNR14	965	2975	5026	6675	7766	8912
SNR15	935	2887	5433	6896	7305	8474

Table B.3 – Limit values for our modified algorithm over the SNR range of [0, 15] dB for subpasses 1 to 6 when the path metric *cdf* reaches *a* = 0.8

***a* = 0.8**	Subpass1	Subpass2	Subpass3	Subpass4	Subpass5	Subpass6
SNR0	4209	9606	15577	19412	22086	27266
SNR1	4130	8253	12131	15373	19600	23628
SNR2	3399	6390	10652	13450	15459	19901
SNR3	2417	5403	8423	11019	13030	15578
SNR4	2569	4755	7275	9975	11258	14172
SNR5	1765	4223	6654	8682	10704	12520
SNR6	1752	3527	5993	8302	9406	11898
SNR7	1371	3486	5553	8086	8573	10625
SNR8	1175	3346	5520	7192	8235	10736
SNR9	1273	3009	4837	6398	7647	9471
SNR10	968	2988	4713	6611	7562	9346
SNR11	897	2694	4672	6677	7191	9192
SNR12	826	2669	4478	6126	7160	8246
SNR13	822	2761	4796	5580	6393	8391
SNR14	772	2440	4530	5482	6705	7543
SNR15	733	2644	4602	5748	6369	7826

Table B.4 – Limit values for our modified algorithm over the SNR range of [0, 15] dB for subpasses 1 to 6 when the path metric *cdf* reaches *a* = 0.7

***a* = 0.7**	Subpass1	Subpass2	Subpass3	Subpass4	Subpass5	Subpass6
SNR0	2801	8260	13353	17230	19632	24253
SNR1	3324	7066	10610	13469	15678	21436
SNR2	2732	5322	8288	11941	13911	18060
SNR3	2057	4647	7378	9666	11599	14007
SNR4	1817	4159	7275	8549	11258	14172
SNR5	1476	3626	5819	7717	9360	11261
SNR6	1464	3027	5244	7275	8364	10697
SNR7	1095	3058	4860	7279	7708	10625
SNR8	937	2603	4845	6399	7488	9761
SNR9	953	2576	4234	6398	6882	8526
SNR10	808	2327	4184	5885	6802	8335
SNR11	748	2357	4157	6009	6534	8502
SNR12	662	2374	3978	5447	6510	7420
SNR13	688	2453	4260	4960	6393	7551
SNR14	673	2440	4027	4872	6092	6861
SNR15	628	2404	3761	5162	5660	7164

Table B.5 – Limit values for our modified algorithm over the SNR range of [0, 15] dB for subpasses 1 to 6 when the path metric *cdf* reaches *a* = 0.6

***a* = 0.6**	Subpass1	Subpass2	Subpass3	Subpass4	Subpass5	Subpass6
SNR0	2801	6878	11121	15120	17183	21219
SNR1	2503	5889	9098	11550	15678	17202
SNR2	2045	4259	7093	10458	12353	16296
SNR3	1213	3871	6326	9666	11599	14007
SNR4	1547	3566	6220	7115	9645	12609
SNR5	1181	3024	4984	6753	8037	11261
SNR6	1170	3027	4493	6234	8364	9519
SNR7	822	2620	4164	6468	7708	9570
SNR8	937	2603	4159	5600	6739	8783
SNR9	629	2147	3627	5595	6882	8526
SNR10	648	2327	3662	5146	6048	8102
SNR11	598	2357	3625	5341	5882	7766
SNR12	662	2076	3472	4755	5849	7420
SNR13	549	2142	3725	4960	5684	6713
SNR14	579	2170	3521	4264	5477	6861
SNR15	523	2165	3348	4598	5660	6510

Table B.6 – Limit values for our modified algorithm over the SNR range of [0, 15] dB for subpasses 1 to 6 when the path metric *cdf* reaches $a = 0.5$

***a* = 0.5**	Subpass1	Subpass2	Subpass3	Subpass4	Subpass5	Subpass6
SNR0	1404	5495	8897	12962	17183	21219
SNR1	1672	4700	7567	11550	13695	17202
SNR2	1362	4259	7093	8965	12353	14491
SNR3	1213	3098	5269	8286	10156	12462
SNR4	1031	2960	5199	7115	8042	11026
SNR5	881	2420	4159	6753	8037	10015
SNR6	877	2521	4493	6234	7301	9519
SNR7	822	2184	4164	5663	6856	8507
SNR8	704	2230	3463	5600	6739	7808
SNR9	629	2147	3627	4792	6111	7569
SNR10	648	1994	3662	4413	6048	7500
SNR11	598	2019	3625	4676	5882	7076
SNR12	496	1781	2984	4755	5205	6595
SNR13	551	1841	3197	4343	5684	6713
SNR14	482	1898	3009	4264	5477	6168
SNR15	467	1925	2927	4025	4955	5869