

University of Southern Queensland
Faculty of Engineering and Surveying

**Iterative Decoding for Error Resilient Wireless Data
Transmission**

A dissertation submitted by

Tawfiqul Hasan Khan

in fulfillment of the requirements of

Courses ENG4111 and 4112 Research Project

towards the degree of

Bachelor of Engineering (Software)

Submitted: November, 2006

Abstract

Both turbo codes and LDPC codes form two new classes of codes that offer energy efficiencies close to theoretical limit predicted by Claude Shannon. The features of turbo codes include parallel code catenation, recursive convolutional encoders, punctured convolutional codes and an associated decoding algorithm. The features of LDPC codes include code construction, encoding algorithm, and an associated decoding algorithm.

This dissertation specifically describes the process of encoding and decoding for both turbo and LDPC codes and demonstrates the performance comparison between these two codes in terms of some performance factors. In addition, a more general discussion of iterative decoding is presented.

One significant contribution of this dissertation is a study of some major performance factors that intensely contribute in the performance of both turbo codes and LDPC codes. These include Bit Error Rate, latency, code rate and computational resources. Simulation results show the performance of turbo codes and LDPC codes under different performance factors.

University of Southern Queensland
Faculty of Engineering and Surveying

ENG411 & ENG4112 *Research Project*

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Engineering and Surveying, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Engineering and Surveying or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Prof G Baker
Dean
Faculty of Engineering and Surveying

Certification

I certify that the ideas, designs and experimental work, results and analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Tawfiqul Hasan Khan

Student Number: D1231090

Signature

Date

Acknowledgements

I wish to extend my gratitude to Dr. Wei Xiang of the Faculty of Engineering and Surveying, University of Southern Queensland, as my supervisor and for his timely and dedicated assistance throughout the conduct of this project and dissertation. Without his knowledge, experience and supervision, completion of this task would have been much more difficult.

Special thanks are also due to my father Mazharul Hasan Khan and my mother Yasmeen Ahmed for their moral support.

Contents

	Page
Abstract	i
Certification	iii
Acknowledgements	iv
Contents	v
List of Figures	viii
List of Tables	x
List of Appendices	xii
Chapter 1 Introduction	1
1.1 Wireless communications	2
1.1.1 Cellular communication systems	3
1.1.2 Paging systems	6
1.1.3 Wireless data networks	7
1.1.4 Cordless telephony	8
1.1.5 Satellite telephony	9
1.1.6 Third generation systems	10
1.2 Other applications of error control coding	10
1.2.1 Hamming codes.....	11
1.2.2 The binary Golay code	12
1.2.3 Binary cyclic codes	12
1.3 Purpose and structure of thesis.....	13

Chapter 2 Error correction coding.....	15
2.1 History.....	16
2.2 Basic concepts of error correcting coding.....	17
2.3 Shannon capacity limit.....	19
2.4 Block codes	20
2.5 Convolution codes.....	21
2.6 Concatenated codes.....	23
2.7 Iterative decoding for soft decision codes.....	23
2.7.1 Turbo codes.....	24
2.7.2 LDPC codes	26
2.8 Chapter summary	28
Chapter 3 Turbo codes: Encoder and decoder construction	29
3.1 Constituent block codes	29
3.1.1 Encoding of block codes.....	30
3.1.2 Systematic codes.....	31
3.2 Constituent convolution codes	32
3.2.1 Encoding of convolution codes.....	32
3.2.2 State diagram.....	33
3.2.3 Generator matrix	34
3.2.4 Trellis diagram	35
3.2.5 Punctured convolution codes	36
3.2.6 Recursive systematic codes.....	38
3.3 Classes of soft-input, soft- output decoding algorithms	41
3.3.1 Viterbi algorithm.....	41
3.3.2 Soft-output Viterbi algorithm (SOVA)	43
3.3.3 Maximum- a – Posteriori (MAP) algorithm.....	45
3.3.4 Max –Log- MAP and Log- MAP algorithms.....	49
3.4 Encoding of turbo codes.....	53
3.5 Overview of turbo decoding	56
3.5.1 Soft-input, soft- output decoding	56
3.5.2 Turbo decoder schematic	58

3.5.3	Example of turbo decoding	60
3.6	Chapter summary	61
Chapter 4 LPDC codes: Code construction and decoding		62
4.1	Code construction.....	62
4.2	Tanner graphs.....	65
4.3	Encoding algorithm.....	67
4.4	Overview of LPDC decoding.....	69
4.4.1	The sum-product algorithm.....	69
4.4.2	Example of LDPC decoding	73
4.5	Chapter summary	74
Chapter 5 Performance comparison between turbo codes and LDPC codes.....		75
5.1	Performance analysis of turbo codes.....	75
5.1.1	Simulation results.....	78
5.2	Performance analysis of LDPC codes.....	81
5.2.1	Simulation results.....	82
5.3	Performance comparison.....	85
5.4	Chapter summary	87
Chapter 6 Conclusions.....		88
6.1	Future work	89
References:.....		173

List of Figures

	Page
Figure 2.1 A digital communication system.....	18
Figure 2.2 Systematic block encoding for error correction.....	19
Figure 2.3 Encoder Structure of a parallel Concatenated (turbo) code.....	24
Figure 2.4 Block diagram of an iterative decoder for a parallel concatenated code.....	25
Figure 3.1 An encoder of memory-2 rate-1/2 convolutional encoder.....	33
Figure 3.2 State diagram of a memory-2 rate-1/2 convolutional encoder.....	33
Figure 3.3 Six sections of the trellis of a memory-2 rate-1/2 convolutional encoder.....	36
Figure 3.4 An encoder of a memory-2 rate-2/3 PCC.....	37
Figure 3.5 An encoder of a memory-2 rate-1/2 recursive systematic convolutional encoder.....	38
Figure 3.6 State diagram of a memory-2 rate-1/2 recursive systematic convolutional encoder.....	40
Figure 3.7 Six sections of the trellis of a memory-2 rate-1/2 recursive systematic convolutional encoder.....	40
Figure 3.8 Encoder Structure of turbo code.....	54
Figure 3.9 Example of turbo encoder.....	55
Figure 3.10 Turbo decoder schematic.....	59
Figure 3.11 Performance of rate-1/2 parallel concatenated (turbo) code with memory-4, rate-1/2 RSC codes, generators (37, 21) and block size = 1024.....	60

Figure 4.1	Tanner graph of Gallager's (3, 4, 20) code.	66
Figure 4.2	Result of permuted rows and columns.	67
Figure 4.3	Performance of Gallager's (3, 4, 20) code with iterative probabilistic decoding.	74
Figure 5.1	Performance of parallel concatenated (turbo) code for different block interleaver size with memory-4, rate-1/2 RSC codes and generators (37, 21).	78
Figure 5.2	Performance of rate-1/2 and rate- 1/3 parallel concatenated (turbo) code with memory-2, generators (7, 5) block interleaver size = 1024 and iteration = 6.	79
Figure 5.3	Performance of rate-1/2 parallel concatenated (turbo) code with different constraint length.	81
Figure 5.4	Performance of Gallager's (3, 4, 20) code with iterative probabilistic decoding for different iteration number.	83
Figure 5.5	Performance of Gallager's (3, 6, 96) code with iterative probabilistic decoding for different iteration number.	84
Figure 5.6	Performance of Gallager's (3, 6, 816) code with iterative probabilistic decoding for different iteration number.	85
Figure 5.7	Comparison between turbo codes and LDPC codes.	86

List of Tables

	Page
Table 1 Quality of Service for rate 1/2 turbo code in AWGN at $E_b/N_o = 3.0$ dB.	80
Table 2 Quality of Service for rate 1/2 LDPC code in AWGN at $E_b/N_o = 8.0$ dB.	83
Table 3 Performance of a rate-1/2 parallel concatenated (turbo) code with memory- 4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 1024. Iteration = 1.	164
Table 4 Performance of a rate-1/2 parallel concatenated (turbo) code with memory- 4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 1024. Iteration = 2.	164
Table 5 Performance of a rate-1/2 parallel concatenated (turbo) code with memory- 4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 1024. Iteration = 3.	164
Table 6 Performance of a rate-1/2 parallel concatenated (turbo) code with memory- 4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 1024. Iteration = 6.	165
Table 7 Performance of a rate-1/2 parallel concatenated (turbo) code with memory- 4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 1024. Iteration = 10.	165
Table 8 Performance of a rate-1/2 parallel concatenated (turbo) code with memory- 4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 128. Iteration = 6.	165

Table 9 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 256. Iteration = 6.	166
Table 10 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 2056. Iteration = 6.	166
Table 11 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 16384. Iteration = 6.	166
Table 12 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 2048. Iteration = 6.	167
Table 13 Performance of Gallager's (3, 4, 20) code for iteration 1.	167
Table 14 Performance of Gallager's (3, 4, 20) code for iteration 2.	168
Table 15 Performance of Gallager's (3, 4, 20) code for iteration 3.	168
Table 16 Performance of Gallager's (3, 4, 20) code for iteration 6.	169
Table 17 Performance of Gallager's (3, 6, 96) code for iteration 1.	169
Table 18 Performance of Gallager's (3, 6, 96) code for iteration 2.	170
Table 19 Performance of Gallager's (3, 6, 96) code for iteration 3.	170
Table 20 Performance of Gallager's (3, 6, 96) code for iteration 6.	171
Table 21 Performance of Gallager's (3, 6, 816) code for iteration 1.	171
Table 22 Performance of Gallager's (3, 6, 816) code for iteration 2.	172
Table 23 Performance of Gallager's (3, 6, 816) code for iteration 3.	172
Table 24 Performance of Gallager's (3, 6, 816) code for iteration 6.	172

List of Appendices

	Page
Appendix A.....	91
Appendix B.....	93
Appendix C.....	96
Appendix D.....	163

Chapter 1

Introduction

In recent years, the demand for efficient and reliable digital data transmission and storage systems has increased to a great extent. The theory of error detecting and correcting codes is the branch of engineering and mathematics which deals with the reliable transmission and storage of data. Information media is not 100% reliable in practice, in the sense that noise frequently causes data to be distorted. To deal with this undesirable but inevitable situation, the concept of error control coding incorporates some form of redundancy into the digital data that allows a receiver to find and correct bit errors occurring in transmission and storage. Since such coding incurred in the communication or storage channel for error detection or error correction, it is often referred to as channel coding.

Error correcting codes are particularly suitable when the transmission channel is noisy. This is the case of wireless communication. It is well known that wireless links are very vulnerable to channel deficiency such as channel noise, multi-path effect and fading. Nowadays, all digital wireless communications use error correcting codes. And iterative decoding is one of the most powerful techniques to correct transmission errors. Iterative

decoding can be defined as a technique employing a soft-output decoding algorithm that is iterated several times to provide powerful error correcting capabilities desired by very noisy wired and wireless channels. Turbo codes and low-density parity-check codes are two error control codes based on iterative decoding. The features of turbo codes include parallel code concatenation, recursive convolutional encoding, non-uniform interleaving, and an associated iterative decoding algorithm. Low-density parity-check (LDPC) codes are another method of transmitting messages over noisy transmission channels. LDPC code was the first code to allow data transmission rates close to the Shannon Limit, where Shannon limit of a communication channel is the theoretical maximum information transfer rate of the channel.

This chapter begins with a brief overview of wireless personal communications. The basic concepts of some error correcting codes other than iteratively decodable codes are then presented later in this chapter.

1.1 Wireless communications

An Italian electrical engineer, Guglielmo Marconi made the first wireless transmission across water in 1897. Wireless is an old-fashioned term for a radio transceiver. Radio transceiver is a mixed receiver and transmitter device. The original application of wireless was to communicate where wires could not go. Throughout the next century, great strides were made in wireless communication technology. During the First World War, the idea of broadcasting emerged, but broadcast stations were generally too awkward to be either mobile or portable. During the Second World War, the contest for superior battle field communication capabilities gave birth to mobile and portable radio.

In modern usage, wireless is a method of communication that uses low-powered radio waves to transmit data between devices. The term refers to communication without cables or cords, mainly using radio frequency and infrared waves. Low-powered radio

waves are often unregulated. Wireless is now increasingly being used by unregulated computer users.

The development of mobile radio paved the way for personal communications. The first widespread non-military application of land mobile radio was pioneered in 1921 in Detroit, Michigan (Hamming 1950, pp. 147-160). The main purpose of it was to provide police car dispatch service. Until 1946, land mobile radio systems were unconnected to each other or to the Public Switched Telephone Network (PSTN). A major milestone was attained in 1946 with the development of the Radiotelephone in the U.S., which to be connected to the PSTN. At first, high powered transmitter and large tower were used to provide service to each market, and by this these markets could complete only one half-duplex call at a time. In 1950, Technological improvements doubled the number of concurrent calls, and the doubled the number again in the mid 1960's. During that time, automatic channel trunking was introduced and full-duplex auto-dial service became available. However, the auto-trunked markets quickly became saturated. For example, in 1976, only 12 trunked channels were available for the entire New York City market of approximately 10 million people. The system could only support 543 paying customers and the waiting list exceeded 3700 people (Rappaport 1996).

1.1.1 Cellular communication systems

A mobile or cellular telephone is a long-range, portable electronic device for personal telecommunications over long distances. It was quite obvious that a new approach to mobile telephony was necessary due to overcrowding in the radio spectrum. During the 1960's, the concept of cellular telephony was emerged in the U.S. at AT&T Bell Laboratories and also in several other countries by various telecommunication companies as well. The proposal of cellular telephony to the Federal Communications Commission (FCC) was made by AT&T in 1968 (Rappaport 1996). The idea behind cellular telephony is that a simple hexagon is used to represent a complex object. The geographical areas are covered by cellular radio antennas. These areas are called cells. Cells that are located far

apart can be assigned the same frequency. The first operational commercial cellular system in the world was fielded in Tokyo in 1979 by NTT (Mitsishi 1989, pp. 30-39). Service in Europe soon followed with the Nordic Mobile Telephone (NMT), which was developed by Ericsson and began operation in Scandinavia in 1981 (Kucar 1991, pp. 72-85). Service in the United States first began in Chicago in 1983 with the Advanced Mobile Phone System (AMPS), which was placed in service by Ameritech (Brodsky 1995).

By late 1980's, it was clear that the first generation cellular systems, which were based on analog signaling techniques, were becoming outdated. Progress in integrated circuit technology made digital communications not only practical, but more economical than analog technology. Digital communications allow the utilization of advanced source coding techniques that suit for greater spectral efficiency. Besides, with digital communications it is possible to use error correction coding to provide a degree of resistance to interference that plagues analog systems. Digital systems also enable multiplexing of dissimilar data types and more efficient network control.

Worldwide deployment of second generation digital cellular systems began in the early 1990's. The main difference to previous mobile telephone systems was that the radio signals that first generation networks used were analog, while second generation networks were digital. Second generation technologies can be divided into TDMA-based and CDMA-based standards depending on the type of multiplexing used. In the TDMA standard, also known as United States Digital Cellular (USDC)¹, several users can transmit at the same frequency but at different times. In the CDMA standard, known initially as Interim Standard 95 (IS-95) and later as cdmaOne, several users can transmit at both the same frequency and time, but modulate their signals with high bandwidth spreading signals. Users can be separated in CDMA because the spreading signals are either orthogonal or have low crosscorrelation (Farely and Hoek M V D 2006).

Europe led the way in 1990 with the deployment of GSM (TDMA-based), the Pan European digital cellular standard. Before the deployment of GSM, there was no unified

¹ USDC is also known as Interim Standards IS-54 and IS-136

standard in Europe. In Scandinavia there were two versions of NMT (NMT-450 and NMT-900), in Germany there was C-450, and elsewhere there was the Total Access Communication System (TACS) and R-2000 (Rappaport 1996), (Goodman 1997).

The situation in the United States was completely different than in Europe. In the U.S., there was but a single standard, AMPS. Since there was just one standard, there was no need to set aside new spectrum. However, the AMPS standard was becoming obsolete and it was obvious that a new technology would be required in crowded markets. The industry's response was to introduce several new incompatible, but bandwidth efficient, standards. Each of these standards was specified to be dual-mode systems. That is, they supported the original AMPS system along with one of the newer signaling techniques. Second generation systems in the U.S. can be divided into analog systems and digital systems. The second generation analog system is Narrowband AMPS (NAMPS), which is similar to AMPS except that the bandwidth required for each user is 10 kHz, rather than the 30 kHz required by AMPS (Farely and Hoek M V D 2006).

At the same time the 900 MHz band was being transitioned from AMPS to the new cellular standards. New spectrum became available in U.S. in the 1.9 GHz band. The systems that occupied this band used similar technology as their second generation siblings in the 900 MHz band. Collectively, these systems were called Personal Communication Systems (PCS), implying a slightly different range of coverage and services than cellular (Farely and Hoek M V D 2006).

Cellular service has proven to be extremely successful. Cellular phones have evolved from a niche item reserved for the rich to a mass-market consumer product. The exponential growth in customer demand has led to extreme congestion in cellular networks serving large metropolitan areas. In theory, cellular has the potential to provide service to an arbitrary number of customers by dividing cells into smaller and smaller areas through the process of "cell-splitting" (Lee 1991, pp. 19-23).

However, practical and economic factors limit just how much a cell can be emaciated. Each cell must be serviced by a centrally located base station. Base stations are expensive and require unsightly antenna towers. Many local governments block the placement of base station towers for reasons of aesthetics and perceived health risks. In addition, the network architecture is becoming more complex by increasing frequency due to occurrence of more cells.

1.1.2 Paging systems

While cellular telephony was evolving, progress was also made with other wireless devices and services including paging, wireless data, cordless telephony, and satellite telephony. Paging is considered as an important component of the growing wireless market. With paging, messages are sent in one way direction, from a centrally located broadcasting tower to a pocket sized receiver possessed by the end user.

Paging was the first mobile communication service for citywide paging systems and it was operating as early as 1963 in the USA and Europe. In the 1970's, with the emergence of the POCSAG (Post Office Code Standardization Advisory Group) standard, alphanumeric paging became possible. Initially POCSAG supported a simple beep-only pager but later incorporated numeric and alpha text messaging. Although the POCSAG standard was reliable, it operated at extremely low data rates so only short messages were permitted. New high speed standards have recently been installed and that allow faster transmission, and longer messages (Budde 2002).

In 1993, Motorola's FLEX system began its service in the U.S.A. and in several foreign markets, while in Europe the ERNIES (European Radio Messaging System) began its operation. Newer paging system, such as Motorola's ReFLEX, allows two-way paging. It allows the user to send back short responses and even email. Motorola's InFLEXion protocol allows the delivery of voice-mail messages over the paging channel (Brodsky 1995).

1.1.3 Wireless data networks

Another emerging area of wireless communication technology is wireless data networks. The demand for wireless computer connectivity rises because of increasing popularity of both the Internet and laptop computers. Wireless data services are designed for packet-switched operation in contrast to the circuit-switched operation of cellular and cordless telephony. Each wireless data service can be categorized as either a wide-area messaging service or a wireless local area network (WLAN).

Wide-area messaging services use licensed bands, and customers pay operators based on their usage. Paging can be considered as a type of wide-area messaging service, although it is usually considered separately for historical reasons. Examples of messaging services include the Advanced Radio Data Information Service (ARDIS) and RAM Mobile Data (RMD). Both of them use the Specialized Mobile Radio (SMR) spectrum near 800/900 MHz (Padgett, Gunther and Hattori 1995, pp. 28-41).

A wireless LAN or WLAN is a wireless local area network, which is the linking of two or more computers without using wires. Unlike commercial messaging services, WLANs use unlicensed frequency bands such as the ISM (Industrial Scientific and Medical) bands that were allocated by the Federal Communications Commission (FCC) in 1985. In the U.S.A., the IEEE 802.11 spread-spectrum based WLAN standard has become increasingly popular, and in Europe the HIPERLAN standard is gaining acceptance (Pahlavan, Probert and Chase 1995, vol. 33, pp. 85-95). A new technology, named Bluetooth, has been introduced by an industry group to provide low cost, short range (10 meter) ad hoc communication networks in the unlicensed 2.4 MHz band between personal computers and other consumer electronic devices and appliances. Bluetooth is an industrial specification for wireless personal area networks (PANs), also known as IEEE 802.15.1.

1.1.4 Cordless telephony

“A cordless telephone or portable telephone is a telephone with a wireless handset which communicates with a base station connected to a fixed telephone landline (POTS) via radio waves and can only be operated close to (typically less than 100 meters of) its base station”.

(Cordless telephone 2006)

Cordless telephones were first emerged in the early 1980's as a consumer product. The benefit of cordless telephony is straight-forward, it allows the user to move around a house or business while talking on the phone, and it provides telecommunication service in rooms that might not be wired.

There are some limitations in first generation cordless phones. The analog signaling technique is prone to interference, spying, and fraud, especially as the number of users increased. In addition, the user is unable to use the phone when he or she goes out of the range of its base station. Modern cordless telephone standards have addressed these deficiencies. In the U.S.A, there are seven frequency bands that have been allocated by the Federal Communications Commission for uses and there are several proprietary cordless systems operating in the 900 MHz band. These systems use advanced digital signaling techniques such as spread spectrum. These are more robust against interference, and are more secure. In Europe, the DECT (Digital European Cordless Telephone) and CT-2 standards not only offer digital signaling, but allow connectivity beyond the home base station by employing a cellular-like infrastructure. A similar system in Japan, the Personal Handyphone System (PHS), has become extraordinarily successful (Rappaport 1996), (Goodman 1997).

1.1.5 Satellite telephony

Satellite telephony is similar to cellular telephony with the exception that the base stations are satellites in orbit around the earth. Depending on the architecture of a particular system, coverage may include the entire Earth, or only specific regions. Satellite telephony systems can be categorized according to the height of the orbit as either LEO (low earth orbit), MEO (medium earth orbit), or GEO (geosynchronous orbit).

GEO systems have been used for many years to communicate television signals. GEO telephony systems, such as INMARSAT, allow communications to and from remote locations, with the primary application being ship-to-shore communications. The advantage of GEO systems is that each satellite has a large footprint, and global coverage up to 75 degrees latitude can be provided with just 3 satellites (Padgett, Gunther and Hattori 1995, vol. 33, pp. 28-41). The disadvantage of GEO systems is that they have a long round-trip propagation delay of about 250 milliseconds and they require high transmission power (Miller 1998, vol. 35, pp. 26-35).

LEO satellites orbit the earth at high speed, low altitude orbits with an orbital time of 70–90 minutes, an altitude of 640 to 1120 kilometres (400 to 700 miles), and provide coverage cells. With LEO systems, both the propagation time and the power requirements are greatly reduced, allowing for more cost effective satellites and mobile units (Satellite phone 2006). The main disadvantages of LEO systems are that more satellites are required and handoff frequently occurs as satellites enter and leave the field of view. A secondary disadvantage of LEO systems is the shorter lifespan of 5-8 years (compared to 12-15 in GEO systems) because of increasing amount of radiation in low earth orbit (Miller 1998, vol. 35, pp. 26-35).

MEO systems represent a compromise between LEO systems and GEO systems, balancing the advantages and disadvantages of each. Examples of LEO systems include Motorola's Iridium (66 satellites, 1998 startup date), Qualcomm/Loral's Globalstar (48 satellites, 1999 startup), Mobile Communications Holdings' Ellipso (14 satellites, 2000

startup), as well as the proposed Teledesic system (288 satellites, 2002 startup). Examples of MEO systems include ICO (10 satellites, 2000 start up), and TRW's Odyssey (12 satellites, 1998 startup) (Miller 1998, vol. 35, pp. 26-35).

1.1.6 Third generation systems

3G is short for third-generation technology. It is used in the context of mobile phone standards. At the close of the 20th century, mobile communications are characterized by a diverse set of applications using many incompatible standards.

In order for today's mobile communications to become truly personal communications in the next century, it will be necessary to consolidate the standards and applications into a unifying framework. The eventual goal is to define a global third generation mobile radio standard originally termed the Future Public Land Mobile Telecommunications System (FPLNITS), and later renamed for brevity IMT-2000, for International Mobile Telecommunications by the year 2000 (Berruto, Gudmundson, Menolascino, Mohr and Pizarosso 1998, vol. 36, February, pp. 85-95).

1.2 Other applications of error control coding

In this section, some error control codes other than iteratively decodable codes are briefly introduced. The first two subsections introduce ECC concepts of two different linear binary codes and the following section introduces a minimum set of concepts necessary for the understanding of binary cyclic codes. Hamming codes are perhaps the most widely known class of block codes, with the possible exception of Reed-Solomon codes. The binary Golay code is the only nontrivial example of an optimal triple-error correcting code.

1.2.1 Hamming codes

Hamming codes are the first class of linear codes devised for error correction (Hamming 1950, pp. 147-160). These codes and their variations have been widely used for error control in digital communication and data storage system. The fundamental principle embraced by Hamming codes is parity. Hamming codes are capable of correcting one error or detecting two errors but not capable of doing both simultaneously.

For any positive integer $m \geq 3$, there exists a Hamming code with the following parameters.

Code length:	$n = 2^m - 1$
Number of information symbols:	$k = 2^m - m - 1$
Number of parity-check symbols:	$n - k = m$
Error-correcting capability:	$t = 1$, where, $d_{\min} = 3$.

A Hamming code word is generated by multiplying the data bits by generator matrix G using modulo-2 arithmetic. This multiplication's result is called the code word vector (c_1, c_2, \dots, c_n) consisting of the original data bits and the calculated parity bits.

The generator matrix G is used in constructing Hamming codes consists of I (the identity matrix) and a parity generation matrix A :

$$G = [I : A] \tag{1.1}$$

Hamming codes are a good mechanism to catch both single and double bit errors or to correct single bit error. This is accomplished by using more than one parity bit, each computed on the different combination of bits in the data.

1.2.2 The binary Golay code

The binary form of the Golay code is one of the most important types of linear binary block codes. It is of particular significance, since it is one of only a few examples of a nontrivial perfect code. A *t-error-correcting* code can correct a maximum number of t errors. One of the major properties of a perfect *t-error-correcting* code is that every word lies within a distance of t to exactly one code word.

Equivalently, the code has minimum distance, $d_{\min} = 2t + 1$, and covering radius t , where the covering radius r is the smallest number such that every word lies within a distance of r to a codeword.

There are two closely related binary Golay codes. The extended binary Golay code encodes 12 bits of data in a 24-bit word in such a way that any triple-bit error can be corrected and any quadruple-bit error can be detected. The second one, the perfect binary Golay code, has codewords of length 23 and is obtained from the extended binary Golay code by deleting one coordinate position. Conversely, the extended binary Golay code can be obtained from the perfect binary Golay code by adding a parity bit (Morelos-Zaragoza 2002).

1.2.3 Binary cyclic codes

Cyclic codes form an important subclass of linear codes. These codes are attractive for two reasons: first, they can be efficiently encoded and decoded using simply shift-registers and combinatorial logic elements, based on their representation using polynomials; and second, because they have considerable inherent algebraic structure.

Cyclic codes were first studied by Prange in 1957 (Hamming 1950, pp. 147-160). Since then, many algebraic coding theorists are encouraging the study of cyclic codes for both random-error correction and burst-error correction.

BCH codes are family of cyclic codes. A BCH (Bose, Ray-Chaudhuri, Hocquenghem) code is a much studied code within the study of coding theory and more specifically error-correcting codes. In technical terms, a BCH code is a multilevel, cyclic, error-correcting, variable-length digital code that is used to correct multiple random error patterns.

1.3 Purpose and structure of thesis

It is well known that wireless links are very vulnerable to channel imperfection such as channel noise, multi-path effect and fading. Iterative decoding is a powerful technique to correct channel transmission errors, and thus improve the bandwidth efficiency of wireless channels.

The essential idea of iterative decoding is to use two or more soft-in/soft-out (SISO) component decoders to exchange soft decoding information. This soft decoding information is known as extrinsic information. It is the exchange of the extrinsic information that provides the powerful error correcting capabilities desired by very noisy wired and wireless channels.

Turbo codes and low-density parity-check (LDPC) codes are two error control codes based on iterative decoding. These codes are widely used in various international standards, such as W-CDMA, CDMA-2000, IEEE 802.11, and DVB-RCS. The purpose of this study is to investigate the performance of two error control codes, namely, turbo codes and LDPC codes based on iterative decoding and demonstrate the performance comparison between these two codes, in terms of error resilient performance, latency, and computational resources. In the first part of this thesis, the fundamental of wireless networks were presented along with some brief explanation of other error control codes. Chapter 2 begins with the history of turbo codes and LDPC codes, and then gives an overview of block, convolutional and concatenated codes.

Chapter 3 discusses the fundamentals of turbo codes and LDPC codes. Chapter 4 focuses on encoding and decoding strategies that are used to implement turbo processing systems, and describes the major classes of soft-input and soft-output decoding algorithms. Chapter 5 focuses on encoding and decoding strategies that are used to implement LDPC processing systems. Chapter 6 discusses the performance analysis for both turbo codes and LDPC codes along with some simulation results. In the end it demonstrates the performance comparison between these two codes. The conclusions and recommendations are found in Chapter 7.

Chapter 2

Error correction coding

The approach to error correction coding taken by modern digital communication systems started in the late 1940's with the revolutionary work of Shannon (Shannon 1948, vol. 27, pp. 379-423 and 623-656), Hamming (Hamming 1950, pp. 147-160), and Golay (Golay 1949, vol. 37, p. 657). In his paper, Shannon showed that it was possible to achieve reliable communications over noisy channel provided that the source's entropy is lower than the channel's capacity. Shannon did not explicitly state how channel capacity could be practically reached; he just stated that that it was attainable. The aim of this project is to investigate turbo codes and LDPC codes, which share many of the same concepts and terminologies of both block and convolutional codes. For this reason, it would be helpful to first provide an overview of block and convolutional codes before proceeding towards the discussion of turbo codes and LDPC codes. This chapter starts with the history of turbo codes and LDPC codes and then gives the concise history of block codes, convolutional codes and concatenated codes. Later in this chapter, the preliminary description of these two main iteratively decodable codes is presented. The code

construction and the algorithms that are used to decode turbo codes and LDPC codes are detailed in the following chapters.

2.1 History

Iterative decoding technique was first introduced in 1954 by P. Elias when he showed his work on iterated codes in his paper “*Error-Free Coding*” (Elias 1954, vol. PGIT-4, pp. 29-37). Later, in the 1960s, R. G. Gallager and J. L. Massey made an important contribution. In their papers, they referred to iterative decoding as probabilistic decoding (Gallager 1962, vol. 8, no. 1, pp. 21-28), (Massey 1963). The main concept then and as it is today, is to maximize the a-posteriori probability of sending data which is a noisy version of the coded sequence.

Over the past eight years, a substantial amount of research effort has been committed to the analysis of iterative decoding algorithms and the construction of iteratively decodable codes or “turbo-like codes” that approach the Shannon limit. Turbo codes were introduced by C. Berrou, A. Glavieux and P. Thitimajshima in their paper “*Near Shannon Limit Error-Correcting Coding and Decoding: Turbo- Codes*” in 1993 (Berrou, Glavieux & Thitimajshima 1993, pp. 1064-1070).

In 1962, R. G. Gallager in his paper “*Low-Density Parity Check Codes*” introduced a class of linear codes, known as low-density parity check (LDPC) codes, and presented two iterative probabilistic decoding algorithms (Gallager 1962, vol. 8, no. 1, pp. 21-28). Later, R. M. Tanner in his paper “*A Recursive Approach to Low Complexity Codes*” extended Gallager’s probabilistic decoding algorithm to the more general case where the parity-checks are defined by sub codes, instead of simple single parity-check equations (Tanner 1981, vol. IT-27, no. 5, pp. 533-547). V. V. Zyablov and M. S. Pinsker in their paper “*Estimation of the Error- Correction Complexity for Gallager Low-Density Codes*” have showed that LDPC codes have a minimum distance that grows linearly with the code length and that errors up to minimum distance could be corrected with a decoding

algorithm with almost linear complexity (Zyablov & Pinsker 1975, vol. 11, no. 1, pp. 26-36). D. J. C. MacKay and R. M. Neal have showed that LPDC codes can get close to the Shannon limit as turbo codes (MacKay & Neal 1996, vol. 32, pp. 1645-1646). Later in 2001, T. J. Richardson, M. A. Shokrollahi and R. L. Urbanke have showed irregular LPDC codes may outperform turbo codes of approximately the same length and rate, when the block length is large (Richardson, Shokrollahi & Urbanke 2001, vol. 47, no. 2, pp. 619-637).

However, such codes have been studied in detail only for the most basic communication system, in which a single transmitter sends data to a single receiver over a channel whose statistics are known to both the transmitter and the receiver. Such a simplistic model is not valid in the case of wireless networks, where multiple transmitters might want to communicate with multiple receivers at the same time over a channel which can vary rapidly. While the design of efficient error correction codes for a general wireless network is an extremely hard problem and hence the techniques of iterative decoding are still under experimentation and study.

2.2 Basic concepts of error correcting coding

Let us consider a digital communication channel transmitting binary series (0s and 1s) by entering them into the channel input. The channel consists of a modulator, the physical transmission medium and a demodulator. The modulator converts input 0s and 1s to pairs of signals suitable for transmission through the medium. During transmission, these signals are distorted and disturbed by noise. The infinite set of the received signals is then converted back to 0s and 1s by the demodulator using a decision rule. These decisions, however, are not free of errors. The probability of error would certainly be reduced if the transmitted power or the duration of the signals was increased. These methods are not used because neither poor efficiency nor low transfer rate is desirable. Fortunately, there is a procedure called Error Correcting Coding (ECC) to keep the probability of transmission

errors at an acceptably low level. Figure 2.1 shows the representation of a typical transmission by a block diagram.

All error correcting codes are based on the same basic principles. The encoder takes the information symbols from the source as input and adds redundant symbols to it. In a basic form, redundant symbols are appended to information symbols to obtain a coded sequence or codeword. Most of the errors that are introduced during the process of modulating a signal, transmission over a noisy medium and demodulation, can be corrected with the help of those redundant symbols. Such an encoding is said to be systematic. Figure 2.2 shows the systematic block encoding for error correction, where first k symbols are information symbols and remaining $n-k$ symbols are some functions of the information symbols. These $n-k$ redundant symbols are used for error correction or detection purposes. The set of all code sequence is called an Error Correcting Code (ECC) and it is denoted by C .

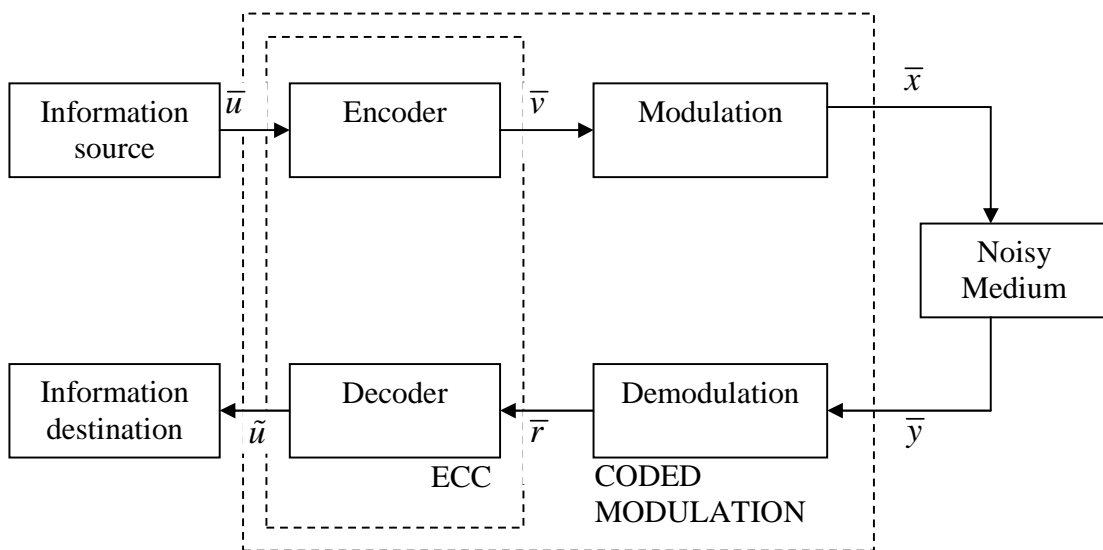


Figure 2.1: A digital communication system.

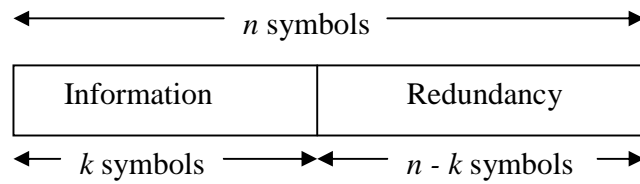


Figure 2.2: Systematic block encoding for error correction.

2.3 Shannon capacity limit

In 1948, Claude E. Shannon published his era making paper (Shannon 1948, vol. 27, pp. 379-423 and 623-656) on the limits of reliable transmission of data over unreliable channels and methodologies of achieving these limits. In that paper, he also formalized the concept of information and investigated bounds for the maximum amount of information that can be transmitted over unreliable channels. In the same paper Shannon introduced the concept of codes as ensembles of vectors that are to be transmitted.

Form Figure 2.1, let's assume the information source produces a message $\bar{u} = \{u_i\}$, $u_i \in \{0,1\}$, $0 \leq i \leq k-1$, of k data bits. The data bits are passed through a channel encoder, which adds structured redundancy by performing a one-to-one mapping of the message \bar{u} to a code word $\bar{x} = \{x_i\}$, $x_i \in \{X_0, X_1\}$, $0 \leq i \leq n-1$ of n code symbols, where X_0 and X_1 are symbols suitable for transmission over the channel. Hence, the code has a rate of $r = k/n$ bits per symbol. The codeword \bar{x} is transmitted over a channel with some capacity bits per channel use.

Given a communication channel, Shannon proved that there exists a number, called the capacity of the channel, such that reliable transmission is possible for rates arbitrarily close to the capacity, and reliable transmission is not possible for rates above the capacity.

The channel capacity depends on the type of channel and it measures the number of data bits that can be supported per channel use (Shokrollahi 2003, pp.2-3).

Now, let's presume the received version of the codeword \bar{y} is passed through a channel decoder, which estimates \tilde{u} of the message. If the channel allows for errors, then there is no general way of telling which codeword was sent with absolute certainty. However, one way to find the most likely codeword is simply list all the 2^k possible codewords, and calculate the conditional probability for the individual codewords. Then find the vector or vectors that yield the maximum probability and return one of them. This decoder is called the maximum likelihood decoder.

Shannon proved a random code can achieve channel capacity for sufficient long block lengths. However, random codes of sufficient length are not practical to implement. Practical codes must have some structure to allow the encoding and decoding algorithms to be executed with reasonable complexity. Codes that approach capacity are very good from a communication point of view, but Shannon's theorems are non-constructive and don't give a clue on how to find such codes (Shokrollahi 2003, pp.2-3).

2.4 Block codes

During the time that Shannon was defining the theoretical limits of reliable communication, Hamming and Golay were busy in developing the first practical error control schemes. Their work gave birth to a successful branch of applied mathematics known as coding theory. In 1946, Richard Hamming, was hired by Bell Labs to work on elasticity theory. However, Hamming found that he spent much of his time working on computers which were highly unreliable at that time. The computers were equipped with error detection capabilities, but they would halt the execution of the program while detecting an error. This led Hamming to search ways to encode the input so that the computer could correct isolated errors and continue running. His solution was to group the data into sets of four information bits and

then calculate three check bits which are a linearly combination of the information bits. The resulting seven bits code word was then fed into the computer. After reading the resulting codeword, the computer ran through an algorithm that could not only detect errors, but also determine the location of a single error. Thus the Hamming code was able to correct a single error in a block of seven encoded bits (Hamming 1950, pp. 147-160).

Although Hamming codes were a great progression, it had some undesirable properties. Firstly, it was not very efficient. It was requiring three check bits for every four data bits. Secondly, it had the ability to correct only a single error within the block. These problems were addressed by Marcel Golay, who generalized Hamming's construction. During his process, Golay discovered two very astonishing codes on which the binary Golay code has already mentioned concisely in Section 1.2.2. The second code is the ternary Golay code, which operates on ternary, rather than binary, numbers. The ternary Golay code protects blocks of six ternary symbols with five ternary check symbols and has the capability to correct two errors in the resulting eleven symbol code word (Golay 1949, vol. 37, p. 657).

The general strategy of Hamming and Golay codes were the same. The ideas were to group q -ary symbols into blocks of k -digit information word and then add $n-k$ redundant symbols to produce an n -digit code word. The resulting code has the ability to correct t errors, and has code rate $r = k/n$. A code of this type is known as a block code, and is referred to as a (q, n, k, t) block code.

2.5 Convolution codes

Although, block codes have attained lots of success, they have several fundamental drawbacks. Firstly, because of their frame oriented characteristic, the entire code word must be received before decoding procedure starts. This can introduce an intolerable latency into the system, particularly for large block lengths. Secondly, the block codes require precise

frame synchronization.² Thirdly, most of the algebraic-based decoders for block codes work with hard-bit decisions, rather than with "soft" outputs of the demodulator. With hard-decision decoding, the output of the channel is taken to be binary, whereas the channel output is continuous-valued with soft-decision decoding. However, in order to achieve the performance bounds predicted by Shannon, continuous-valued channel output are required.

The drawbacks of block codes can be avoided by taking a different approach to coding and one of them was convolutional coding. Convolutional codes were first introduced in 1955 by Elias (Elias 1955, vol. 4, pp. 37-47). Rather than splitting data into distinct blocks, convolutional encoders add redundancy to a continuous stream of input data by using a linear shift register. In convolutional codes, each block of k bits is mapped into a block of n bits but these n bits are not only determined by the present k information bits but also by the previous information bits. This dependence can be captured by a finite state machine. The total number of bits that each output depends on is called the constraint length, and denoted by K_c . The rate of the convolutional encoder is the number of data bits k taken in by the encoder in one coding interval, divided by the number of n coded bits during the same interval. While the data is continuously encoded, it can be continuously decoded with only small latency. In addition, the decoding algorithms can make full use of soft-decision information from the demodulator.

After the development of the Viterbi algorithm, convolutional coding began to be utilized in extensive application of communication systems. The constraint length $K_c = 7$ "Odenwalder" convolutional code, which operates at rates $r = 1/3$ and $r = 1/2$, has become a standard for commercial satellite communication applications (Berlekamp, Peile and Pope 1987), (Odenwalder 1976). Convolutional codes were used by several deep space probes such as Voyager and Pioneer (Wicker 1995).

All of the second generation digital cellular standards incorporate convolutional coding; GSM uses a $K_c = 5$, $r = 1/2$ convolutional code, USDC use a $K_c = 6$, $r = 1/2$ convolutional code, and IS-95 uses a $K_c = 9$ convolutional code with $r = 1/2$ on the downlink and $r = 1/3$

² Frame synchronization means that the decoder has knowledge of which symbol is the first symbol in a received code word.

on the uplink (Rappaport 1996). Globalstar also uses a $K_c = 9$, $r = 1/2$ convolutional code, while Iridium uses a $K_c = 7$ convolutional code with $r = 3/4$ (Costello, Hagenauer, Imai and Wicker 1998, vol. 44, pp. 2531-2560).

2.6 Concatenated codes

Convolutional codes have a key weakness and that is vulnerability to burst errors. This weakness can be eased by using an interleaver, which scrambles the order of the coded bits prior to transmission. At the receiver end, a deinterleaver is used that places the received coded bits back into the proper order after demodulation and prior to decoding. The most common type of interleaver is the block interleaver, which is simply an $M_b \times N_b$ bit array. Data is placed into the array column-wise and then read out row-wise. A burst error of length up to N_b bits can be spread out by a block interleaver such that only one error occurs in every M_b bits. All of the second generation digital cellular standards use some form of block interleaving. A second type of interleaver is the convolutional interleaver, which allows continuous interleaving and deinterleaving and is ideally suited for use with convolutional codes (Ramesy 1970, vol. 16, pp. 338-345).

2.7 Iterative decoding for soft decision codes

Iterative decoding is defined as a technique employing a soft-output decoding algorithm that is iterated several times to improve the error performance of a coding scheme, with the aim of approaching true maximum-likelihood decoding (MLD), with less complexity. After designing the underlying error correcting code, the error performance can be improved by simply increasing the number of iteration. In terms of the application of iterative decoding algorithms, ECC schemes can be generally categorized into two classes, i.e., Turbo codes and LDPC codes. This section presents an overview of these two main iteratively decodable codes.

2.7.1 Turbo codes

Turbo codes are a new class of error correction codes that was introduced in 1993, by a group of researchers from France, along with a practical decoding algorithm. The turbo codes are very important in the sense that they enable reliable communications with power efficiencies close to the theoretical limit predicted by Claude Shannon. Hence, turbo codes have been projected for low-power applications such as deep-space and satellite communications, as well as for interference limited applications such as third generation cellular and personal communication services. Figure 2.3 shows the block diagram of turbo encoder.

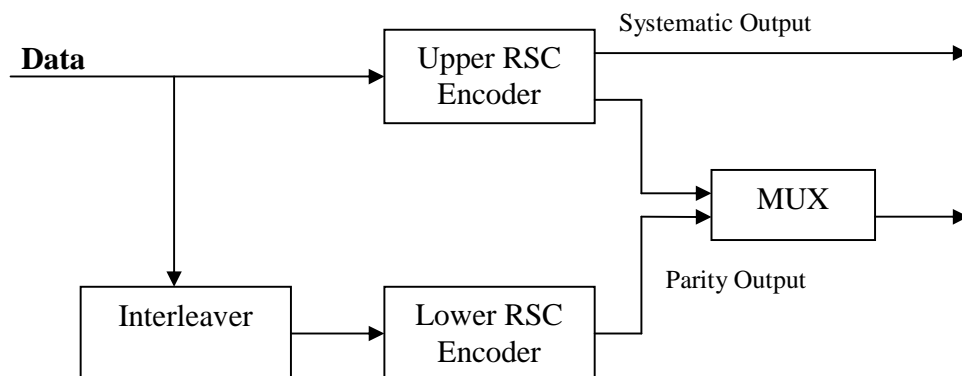


Figure 2.3: Encoder structure of a parallel concatenated (turbo) code.

A turbo code is the parallel concatenation of two or more Recursive Systematic Convolutional (RSC) codes separated by an interleaver. Two identical rate $r = 1/2$ RSC encoders work on the input data in parallel as shown in Figure 2.3. As shown in the figure, the upper encoder receives the data directly, while lower encoder receives the data after it has been interleaved by a permutation function α . In general, the interleaver α is a pseudo-random interleaver. It maps bits in position i to position $\alpha(i)$ according to a prescribed, but randomly generated rule. Because the encoders

are systematic (one of the outputs is the input itself) and receive the same input (although in a different order), the systematic output of the lower encoder is completely unnecessary and does not need to be transmitted. However, the parity outputs of both encoders are transmitted. The overall code rate of the parallel concatenated code is $r = 1/3$, although higher code rate can be achieved by puncturing the parity output bit with a multiplexer (MUX) circuit.

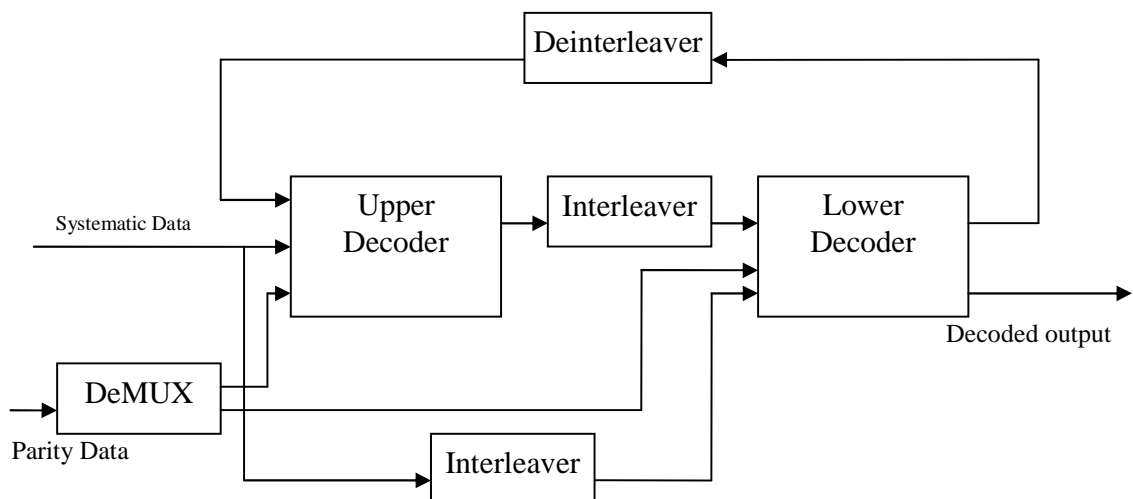


Figure 2.4: Block diagram of an iterative decoder for a parallel concatenated code.

As an interleaver exists between two encoders, optimal decoding of turbo codes is incredibly complex and therefore impractical. However, a suboptimal iterative decoding algorithm was presented in (Berrou, Glavieux & Thitimajshima 1993, pp. 1064-1070) which presents good performance at much lower complexity. The idea behind the decoding strategy is to break the overall decoding problem into two smaller problems with locally optimal solutions and to share information in an iterative fashion. The decoder associated with each of the constituent codes is modified so that it produces soft-outputs in the form of a-posteriori probabilities of

the data bits. The two decoders are cascaded as shown in Figure 2.4. As shown in the figure, the lower decoder receives the interleaved version of soft-output of the upper decoder. At the end of the first iteration, the deinterleaved version of soft-output of the lower decoder is fed back to the upper decoder and used as a-priori information in the next iteration. Decoding continues in an iterative fashion until the desired performance is achieved.

The original turbo code of (Berrou, Glavieux & Thitimajshima 1993, pp. 1064-1070) used constraint length $K_c = 5$ RSC encoders and a 65,536 bit interleaver. The parity bits were punctured in such a way that the overall code was a ($n = 131,072$, $k = 65,532$) linear block code. Simulation results showed that a bit error rate of 10^{-5} could be achieved at an E_b/N_0 ratio of just 0.7 decibels (dB) after 18 iterations of decoding. Thus, the author claimed that turbo codes could come within a 0.7 dB of the Shannon limit. However, it is found that the performance of the turbo codes increases as the length of the codes n increases.

2.7.2 LDPC codes

LDPC codes are one of the hottest topics in coding theory today. Besides turbo codes, low-density parity-check (LDPC) codes form another class of Shannon limit-approaching codes. Unlike many other classes of codes, LDPC codes are very well equipped with very fast encoding and decoding algorithms by now. The design of the codes is such that these algorithms can recover the original codeword in the face of large amount of noise. New analytic and combinatorial tools make LDPC codes not only attractive from a theoretical point of view, but also perfect for practical applications.

In 1962, Robert Gallager (Gallager 1962, vol. 8, no. 1, pp. 21-28) invented low-density parity-check (LDPC) codes in his PhD thesis. Soon after the invention of LPDC codes, they were mostly forgotten, and reinvented several times for the next 30 years. The comeback of LPDC codes were one of the most fascinating aspects of their history. Recently, however,

they have been strongly promoted, and MacKay in his work showed that LDPC codes are capable of closely approaching the channel capacity. In particular, using random coding arguments MacKay showed that LDPC code collections can approach the Shannon capacity limit exponentially fast with increasing code length (MacKay & Neal 1996, vol. 32, pp. 1645-1646).

LDPC codes are codes, specified by a matrix containing mostly 0's and relatively few 1's. There are basically two classes of LDPC codes based on the structure of this matrix, i.e., regular LDPC codes and irregular LDPC codes.

A regular LDPC code is a linear (w_c, w_r, N) code with parity-check matrix \mathbf{H} having the Hamming weight of the columns and rows of \mathbf{H} is equal to w_c and w_r , where both w_c and w_r are much smaller than the code length N . In somewhat artificial sense, LDPC codes are not most favorable of minimizing the probability of decoding error for a given block length. It can be shown that the maximum rate at which they can be used is bounded by the channel capacity. However, the simplicity in the decoding scheme can compensate these disadvantages.

As previously mentioned, a regular LDPC code is defined as the null space of a parity-check matrix \mathbf{H} . This parity-check matrix has some structural properties which are given below:

1. The parity-check matrix \mathbf{H} has constant Hamming weight of the columns and rows equal to w_c and w_r , where both w_c and w_r are much smaller than the code length.
2. The number of 1's in common between any two columns, denoted by λ , is no greater than 1. This implies that no two rows of \mathbf{H} have more than one 1 in common. Because both w_c and w_r are small compared with the code length and the number of rows in the matrix \mathbf{H} , hence the matrix has small density of 1's.

This is the reason why \mathbf{H} is said to be a low-density parity-check matrix, and the code specified by \mathbf{H} is hence called a low-density parity check (LDPC) code.

An irregular LDPC codes are obtained, if the Hamming weights of the columns and rows of \mathbf{H} are chosen in accordance to some nonuniform distribution (Richardson, Shokrollahi & Urbanke 2001, vol. 47, no. 2, pp. 619-637). An irregular LDPC code has a very sparse parity check matrix in which the column weight may vary from column to column. In fact, the best known LDPC codes are irregular; they can do better than regular LDPC codes by 0.5 dB or more (Luby, M G, Miteznmacher, Shokrollahi, M A and Spielman, D A 2001, vol. 47, no. 2, pp. 585-598).

2.8 Chapter summary

This chapter has focused on the basic concepts of two classes of ECC codes, i.e., turbo codes and LDPC codes. Both turbo codes and LDPC codes utilize a soft-output decoding algorithm that is iterated several times to improve the bit error probability, with the aim of approaching Shannon capacity limit, with less complexity.. The basic concepts of error correcting codes and Shannon capacity limit were presented. Turbo codes have many of the common concepts and terminologies of block codes, convolutional codes and concatenated codes. LDPC codes also share some idea of block codes. Hence, an outline for each block codes, convolutional codes and concatenated codes was presented separately.

Presently turbo codes are being utilized in the applications of satellite communications, third generation cellular and personal communication services. . LDPC codes with new analytic and combinatorial tools have also become perfect for practical applications.

Chapter 3

Turbo codes: Encoder and decoder construction

This chapter focuses on the encoder construction and decoding algorithms of turbo codes. Turbo codes share many of the same concepts and terminologies of both block codes and convolutional codes. For this reason, it is helpful to first provide an overview of block codes and convolutional codes before proceeding to the discussion of turbo codes.

3.1 Constituent block codes

Block codes process the information on a block-by-block basis, treating each block of information bits independently from others. Typically, a block code takes a k -digit information word, and transforms this into an n -digit codeword. At first, the encoder for a block code divides the information sequence into message blocks of k information bits each. A message block is then represented by the binary k -tuple $\bar{u} = (u_0, u_1, \dots, u_{k-1})$. So there are total of 2^k different possible messages. The encoder transforms each message \bar{u}

independently into an n -tuple $\bar{v} = (v_0, v_1, \dots, v_{n-1})$. Hence, corresponding to the 2^k different possible messages, there are 2^n different possible code words at the encoder output.

3.1.1 Encoding of block codes

The encoding process consists of breaking up the data into k -tuples u called messages, and then performing a one-to-one mapping of each message u_i to an n -tuple v_i called a code word. Let C denote a binary linear (n, k, d_{\min}) code. Since C is a k -dimensional vector subspace, any codeword $\bar{v} \in C$ can be represented as a linear combination of the elements in the basis:

$$\bar{v} = u_0 v_0 + u_1 v_1 + \dots + u_{k-1} v_{k-1}, \quad (3.1)$$

where $u_i \in \{0,1\}, 0 \leq i < k-1$. Equation (3.1) can be described by the matrix multiplication

$$\bar{v} = \bar{u}G, \quad (3.2)$$

where G is the $k \times n$ generator matrix.

$$G = \begin{pmatrix} \bar{v}_0 \\ \bar{v}_1 \\ \vdots \\ \bar{v}_{k-1} \end{pmatrix} = \begin{pmatrix} v_{0,0} & v_{0,1} & \dots & v_{0,n-1} \\ v_{1,0} & v_{1,1} & \dots & v_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{k-1,0} & v_{k-1,1} & \dots & v_{k-1,n-1} \end{pmatrix}. \quad (3.3)$$

Since C is k -dimensional vector space in V_2 , there is an $(n - k)$ - dimensional dual space C^\top , generated by the rows of a matrix \mathbf{H} , called the parity-check matrix, such that $\mathbf{G}\mathbf{H}^\top = 0$, where \mathbf{H}^\top denotes the transpose of \mathbf{H} .

The Hamming distance $d_H(\bar{v}_1, \bar{v}_2)$ is the number of positions that two code words differ. It can be found by first taking the modulo-2 sum of the two code words and then counting the number of ones in the result. The minimum distance d_{\min} of a code is the smallest Hamming distance between any two words

$$d_{\min} = \min_{\bar{v}_1, \bar{v}_2 \in C} \{d_H(\bar{v}_1, \bar{v}_2) \mid \bar{v}_1 \neq \bar{v}_2\}. \quad (3.4)$$

A code with the minimum distance d_{\min} is capable of correcting all code words with t or fewer errors, where

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor, \quad (3.5)$$

where $\lfloor x \rfloor$ denotes the largest integer less than or equal to x .

The Hamming weight can be found by simply counting the number of ones in the code word. For linear codes, the minimum distance is the smallest Hamming weight of all code words except the all-zeros code word

$$d_{\min} = \min_{i>0} w(v_i). \quad (3.6)$$

3.1.2 Systematic codes

A code is said to be systematic if the message \bar{u} is contained within the codeword \bar{v} . The generator matrix G of a systematic code can be brought to systematic form

$$G_{\text{sys}} = [I_k \mid P], \quad (3.7)$$

where I_k is the $k \times k$ identity matrix and P is a $k \times (n - k)$ parity sub-matrix, such that

$$P = \begin{pmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,n-k-1} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,n-k-1} \\ \vdots & \vdots & \ddots & \vdots \\ p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} \end{pmatrix}. \quad (3.8)$$

Since $GH^T = 0$, it follows that the systematic form, H_{sys} , of parity-check matrix is

$$H_{sys} = [P^T | I_{n-k}]. \quad (3.9)$$

3.2 Constituent convolution codes

3.2.1 Encoding of convolution codes

A convolution encoder has memory, in the sense that the output symbols depend not only on the input symbols, but also on the previous inputs or outputs. The memory m of the encoder measured by the total length of shift registers M . Constraint length $K_c = m + 1$, is defined as the number of inputs ($u[i], u[i-1], \dots, u[i-m]$) that affect the outputs ($v^{(0)}[i], \dots, v^{(n-1)}[i]$) at time i . An encoder with m memory elements will be referred to as a memory- m encoder. Figure 3.1 shows an example of a convolutional encoder. For this encoder, memory $m = 2$, the code rate $r = 1/2$ and the constraint length $K_c = 3$.

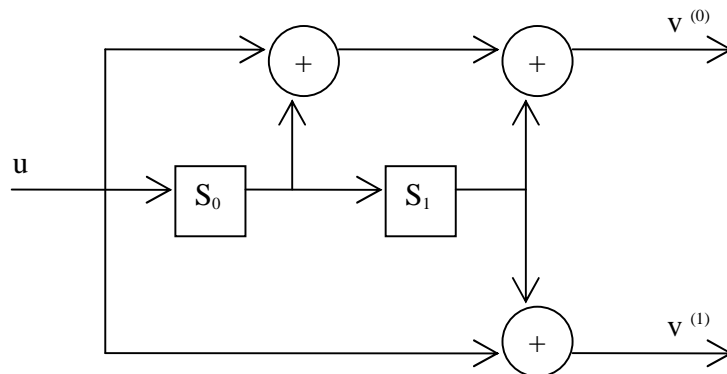


Figure 3.1: An encoder of memory-2 rate-1/2 convolutional encoder.

3.2.2 State diagram

A memory- m rate $1/n$ convolution encoder can be represented by a state diagram. An encoder with m memory elements there are 2^m states. Figure 3.2 shows the state diagram of the memory-2 rate- $1/2$ convolution code. As there is only one information bit, hence two branches enter and leave each state and they are labeled by $u[i] / v^{(0)}[i], \dots, v^{(n-1)}[i]$.

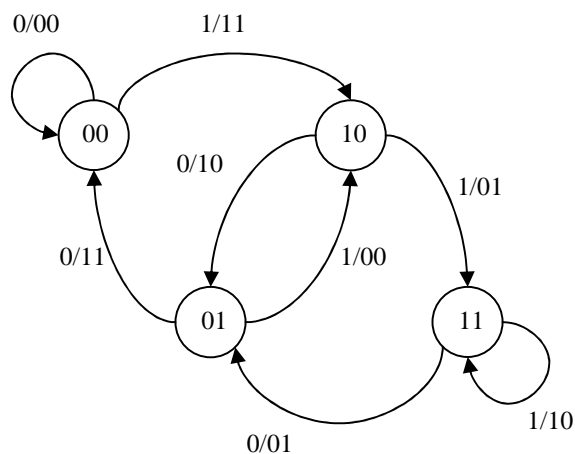


Figure 3.2: State diagram of a memory-2 rate-1/2 convolutional encoder.

3.2.3 Generator matrix

For an encoder with memory $-m$ rate- $1/2$, the impulse responses can last at most $m + 1$ time units and can be written as $\bar{g}^{(0)} = (g_0^{(0)}, g_1^{(0)}, \dots, g_m^{(0)})$ and $\bar{g}^{(1)} = (g_0^{(1)}, g_1^{(1)}, \dots, g_m^{(1)})$. The impulse responses $\bar{g}^{(0)}$ and $\bar{g}^{(1)}$ are called the generator sequences of the encoder. In general, when referring to a convolution code, the generators are expressed in octal form. In order to stress the dynamic nature of the encoder, the indeterminate D (for “Delay”) is used and the generators are written as polynomials in D , $\bar{g}^{(0)}(D), \dots, \bar{g}^{(n-1)}(D)$.

A convolutional encoder is a linear time-invariant system, with impulse responses given by the code generators, $\bar{g}^{(0)}(D), \dots, \bar{g}^{(n-1)}(D)$, where

$$\bar{g}^{(j)}(D) = g_0^{(j)} + g_1^{(j)}D + g_2^{(j)}D^2 + \dots + g_m^{(j)}D^m, \quad (3.10)$$

for $0 \leq j < n$. In term of the generators of a code, the output sequences can be expressed as

$$v^{(j)}[i] = \sum_{l=0}^m u[i-l]g_j[l], \text{ where } 0 \leq j < n \quad (3.11)$$

In particular, for a rate- $1/2$ memory- m encoder, if the generator sequences $\bar{g}^{(0)}$ and $\bar{g}^{(1)}$ are interlaced, then they can be arranged in the matrix in such that

$$G = \left[\begin{array}{cccccc} g_0^{(0)} g_0^{(1)} & g_1^{(0)} g_1^{(1)} & g_2^{(0)} g_2^{(1)} & \dots & g_m^{(0)} g_m^{(1)} & \\ & g_0^{(0)} g_0^{(1)} & g_1^{(0)} g_1^{(1)} & \dots & g_{m-1}^{(0)} g_{m-1}^{(1)} & g_m^{(0)} g_m^{(1)} \\ & & g_0^{(0)} g_0^{(1)} & \dots & g_{m-2}^{(0)} g_{m-2}^{(1)} & g_{m-1}^{(0)} g_{m-1}^{(1)} & g_m^{(0)} g_m^{(1)} \\ & & & \dots & & \dots & \\ & & & & \dots & & \dots \end{array} \right] \quad (3.12)$$

where the blank areas are all zeros. We can then, write the encoding equations in matrix form as

$$\bar{v} = \bar{u}G, \quad (3.13)$$

where all operations are modulo-2. The G matrix is then called the Generator matrix of the encoder.

Hence, the output sequences $\bar{v}^{(j)}(D)$, $0 \leq j < n$, as mentioned, are equal to the discrete convolution between the input sequence $\bar{u}(D)$ and the code generators $\bar{g}^{(0)}(D), \bar{g}^{(1)}(D), \dots, \bar{g}^{(n-1)}(D)$. Let, $\bar{v}(D) = v^{(0)}(D) + Dv^{(1)}(D) + \dots + D^{n-1}v^{(n-1)}(D)$. Then the relationship between input and output can be written as follows:

$$\bar{v}(D) = \bar{u}(D)G(D), \quad (3.14)$$

where, generators of a rate $1/n$ convolutional code are arranged in matrix and referred to as a polynomial generator matrix $G(D)$.

3.2.4 Trellis diagram

A trellis diagram expands the state diagram to show how the states change with time. A trellis diagram is constructed by placing the state diagram of the code at each time interval with branches connecting states between time i and $i + 1$, in correspondence with the encoder table. The inputs and outputs are showed only in the first trellis as any two of the same state-to-state transitions have the same outputs and inputs. Thus, not all outputs are needed to be showed and hence reduce the clutter. Figure 3.3 shows six sections of the trellis of a memory-2 rate-1/2 convolutional encoder.

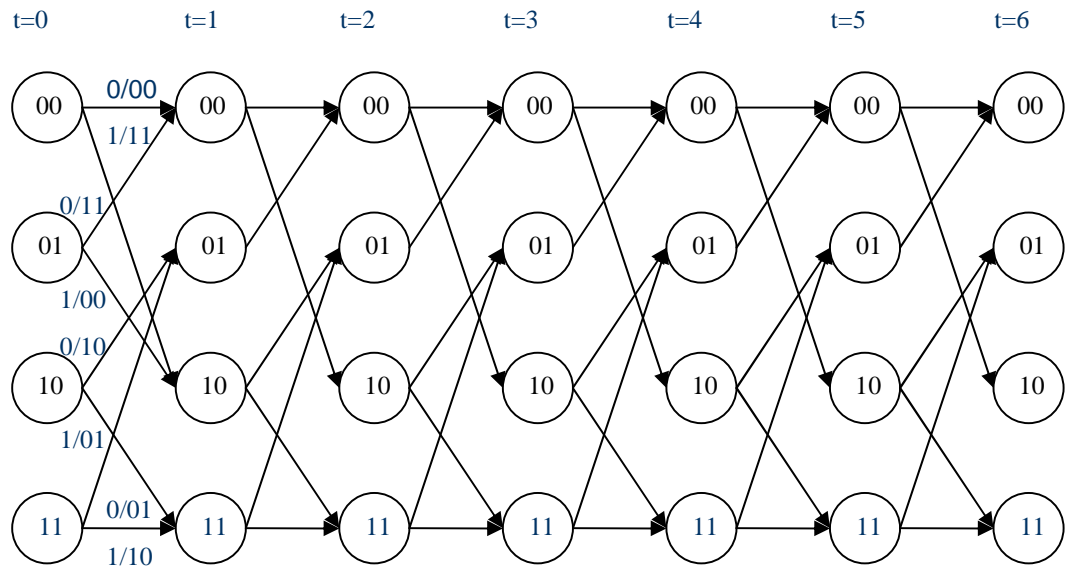


Figure 3.3: Six sections of the trellis of a memory-2 rate-1/2 convolutional encoder.

3.2.5 Punctured convolution codes

If convolutional codes are rate of $1/n$, the highest rate that can be achieved is $r = 1/2$. For many applications it may be desirable to have higher rates such as $r = 2/3$ or $r = 3/4$. One way to attain these higher rates is to use an encoder that can take more than one input stream and thus having rate $r = k/n$, where $k > 1$. However, this increases the complexity of decoder's add-compare-select (ACS) circuit exponentially in the number of input streams (Wicker 1995). Hence, it is desirable to keep the number of input streams at the encoder as low as possible. An alternative of using multiple encoder input streams is to use a single encoder input stream and a process called puncturing.

Puncturing is the process of systematically deleting, or not sending some output bits of a low-rate encoder. The code rate is determined by the number of deleted bits. For instance, one out of every four bits is deleted from the output of a rate $1/2$ convolutional encoder. Then for every two bits at the input of the encoder, three bits remain as outputs after puncturing. Hence, the punctured rate is $2/3$. The convolutional codes of rate $3/4$ can be

generated from the same encoder by deleting two out of every six output bits. One of the main benefits of puncturing is that it allows the same encoder to attain a wide range of coding rates by simply changing the number deleted bits (Hagenauer 1988, vol. 36, pp. 389-400).

When puncturing is used, the location of the deleted bits must be explicitly stated. A puncturing matrix P specifies the rules of deletion of output bits. P is a $k \times n_p$ binary matrix, with binary symbols p_{ij} that indicate whether the corresponding output bit is transmitted ($p_{ij} = 1$) or not ($p_{ij} = 0$). For example, the following matrix can be used to increase a rate 1/2 code to rate 2/3

$$P = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

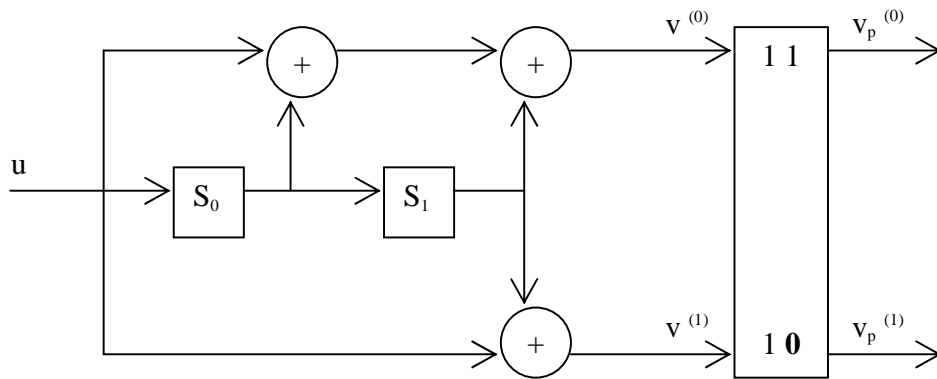


Figure 3.4: An encoder of a memory-2 rate-2/3 PCC.

The corresponding encoder is depicted in Figure 3.4. A coded sequence

$$\bar{v} = (\dots, v_i^{(0)}v_i^{(1)}, v_{i+1}^{(0)}v_{i+1}^{(1)}, v_{i+2}^{(0)}v_{i+2}^{(1)}, v_{i+3}^{(0)}v_{i+3}^{(1)}, \dots),$$

of the rate- 1/2 encoder is transformed into code sequence

$$\bar{v}_p = (\dots, v_i^{(0)}v_i^{(1)}, v_{i+1}^{(0)}v_{i+2}^{(1)}, v_{i+3}^{(0)}, \dots),$$

i.e., every other bit of the second output is deleted (Morelos-Zaragoza 2002).

One of the goals of puncturing is that for a variety of high-rate codes, the same decoder can be used. One way to achieve decoding of a punctured convolution code using the viterbi decoder of the low-rate code is by the insertion of “deleted” symbols in the positions that were sent. This process is known as depuncturing. A special flag is used to mark these deleted symbols.

3.2.6 Recursive systematic codes

If a data sequence being encoded becomes a part of the encoded output sequence, that codes are referred to as systematic.

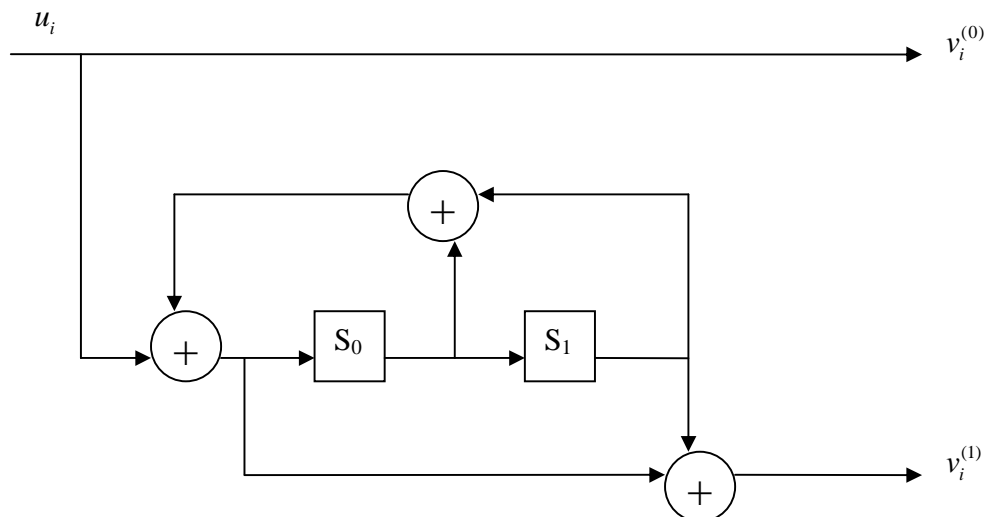


Figure 3.5: An encoder of a memory-2 rate-1/2 recursive systematic convolutional encoder.

Convolutional codes can be made systematic without reducing the free distance. A rate 1/2 convolutional code is made systematic by first calculating the remainder $\bar{y}(D)$ of the polynomial division $\bar{u}(D)/\bar{g}^{(0)}(D)$. The parity output is then found by the polynomial multiplication $\bar{v}^{(1)}(D) = \bar{y}(D)\bar{g}^{(1)}(D)$, and the systematic output is simply $\bar{v}^{(0)}(D) = \bar{u}(D)$. Codes generated in this manner are referred to as recursive systematic convolutional (RSC) codes. RSC encoding proceeds by first computing the remainder variable

$$y[i] = u[i] + \sum_{l=1}^m y[i-l]g_0[l] \quad (3.15)$$

and then finding the parity output

$$v^{(1)}[i] = \sum_{l=0}^m y[i-l]g_1[l] \quad (3.16)$$

A systematic encoder is also an example of discrete-time linear time-invariant system. While conventional convolutional encoders are finite impulse response filters, RSC encoders are infinite impulse response (IIR) filters because the generator matrix contains rational functions. RSC encoders are finite state machines and can be represented by state and trellis diagrams. The state diagram of a memory-2 rate- 1/2 RSC encoder is shown in Figure 3.6 and the trellis diagram for this encoder is shown in Figure 3.7.

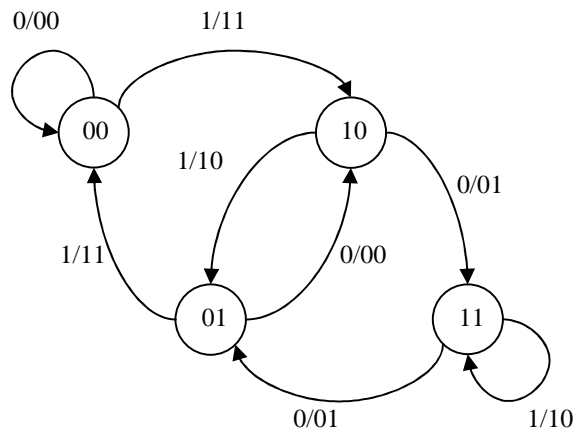


Figure 3.6: State diagram of a memory-2 rate-1/2 recursive systematic convolutional encoder.

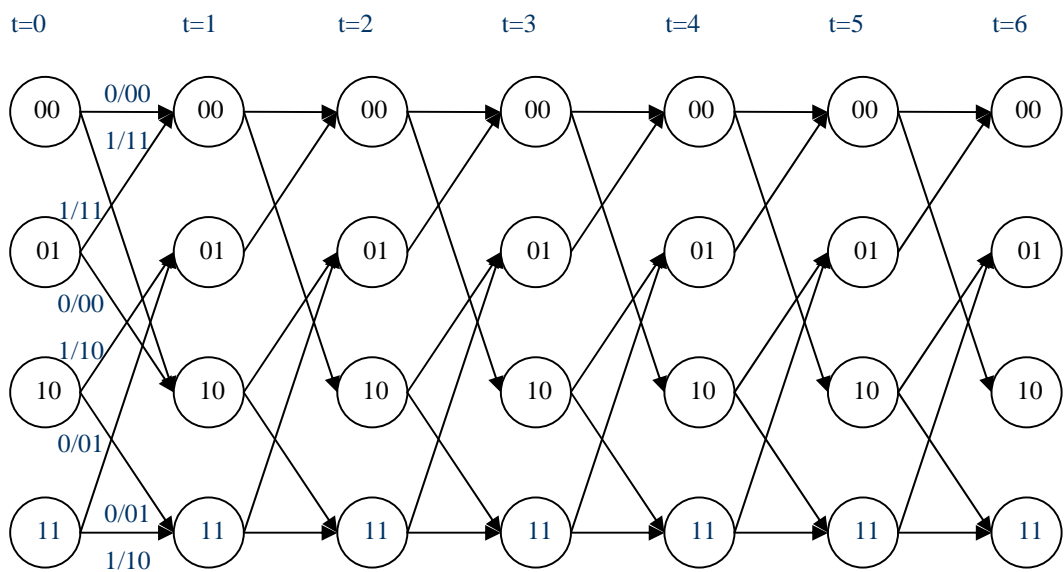


Figure 3.7: Six sections of the trellis of a memory-2 rate-1/2 recursive systematic convolutional encoder.

It can be noticed that the state and the trellis diagrams for the RSC code are almost identical to those for the related non-systematic convolutional code. Actually, the only difference between the two state diagrams is that the input bits labeling the branches leaving nodes (10) and (01) are complements of one another. In conventional convolutional codes, the input bits labeling the two branches entering any node are the same. Conversely, for RSC codes, the input bits labeling the two branches entering any node are complements of one another. Since the structure of the trellis and the output bits labeling the branches remain the same when the code is made systematic and the minimum free distance remain unchanged.

3.3 Classes of soft-input, soft- output decoding algorithms

3.3.1 Viterbi algorithm

Viterbi Algorithm (VA) is the optimum-decoding algorithm for convolutional codes. It finds the sequence of symbols in the given trellis that is closest in distance to the received sequence of noisy symbols. This sequence computed is the global most likely sequence. Therefore, VD is also known as *maximum likelihood decoder*.

The likelihood between the received sequence \bar{r} after transmission over a noisy channel and the actual coded sequence \bar{v} is given by the conditional probability

$$P(\bar{r} | \bar{v}) = \prod_{i=0}^{n-1} P(r_i | v_i) \quad (3.17)$$

To compute the global most-likely sequence, the VA first recursively computes the survivor path entering each state. After the survivor paths entering all states are computed, the survivor path that has the minimum path metric is selected to be the global most likely path.

In VA, the branch metric is defined as the Hamming distance between the received noisy symbol, \bar{r} and the ideal noiseless output symbol, \bar{v} . At stage i , the branch metrics is

$$BM_i^{(b)} = d_H(\bar{r}[i], \bar{v}[i]), \quad (3.18)$$

$b = \sum_{l=0}^{n-1} v_l[i] 2^{n-1-l}$, associated with the n outputs $\bar{v}[i]$ of every branch and the n received bits $\bar{r}[i]$.

For each state $S_i^{(k)}$, $k = 0, 1, \dots, 2^m - 1$, path metrics is defined as the as the distance between the survivor path and the sequence of noisy symbols. It is calculated as follows

$$M(S_i^{(k)}) = \min\{M(S_{i-1}^{(k_1)}) + BM_i^{(b_1)}, M(S_{i-1}^{(k_2)}) + BM_i^{(b_2)}\}. \quad (3.19)$$

After the most likely transition to state at i is computed, the path metric for state at i , $M(S_i^{(k)})$, is updated and the survivor paths is updated as follows, with the output of the winning branch \bar{v}_{k_j} , $j \in \{1, 2\}$,

$$\bar{y}_i^{(k)} = (\bar{y}_{i-1}^{(k_j)}, \bar{v}_{k_j}). \quad (3.20)$$

The path metrics and the survivor paths are updated for all states at each step. In the end, the survivor path with the minimum path metric is selected to be the most likely path.

VA is the optimal maximum likelihood detection method in AWGN channel when Euclidean distance is used as a distance measure. With respect to the VD algorithm with Hamming distance, two changes are required:

1. The branch metric is computed as follows,

$$BM_i^{(b)} = (r_i - m(v_i))^2 \quad (3.21)$$

Let E denote the energy of transmitted symbol. The mapping rule is Binary Phase-Shift Keying (BPSK),

$$m(v_i) = \begin{cases} \sqrt{E} & \text{if, } v_i = 0; \\ -\sqrt{E} & \text{if, } v_i = 1; \end{cases} \quad (3.22)$$

2. Instead of taking the minimum path metrics, the VD takes the maximum path metrics.

3.3.2 Soft-output Viterbi algorithm (SOVA)

A Viterbi algorithm is implemented to minimize the transmission errors of messages through the system by computing the most likely state sequence of a hard decision or soft decision input and then outputting it as a hard decision. The Viterbi Algorithm attempts to minimize bit errors of the output by estimating the original input bits. This is accomplished by calculating the different input possibilities of a specific output symbol and then assigning a confidence level to those inputs that have the most likely probability of actually having occurred. An extension of the classical Viterbi Algorithm is the Soft Output Viterbi Algorithm (SOVA), which attempts to minimize bit errors as well. The SOVA differs from the classical algorithm in that it outputs soft decisions, rather than hard decisions. The SOVA computes soft output of the information bits as a log-likelihood ratio (LLR),

$$\Lambda(u_i) = \log \left(\frac{\Pr\{u_i = 1 | \bar{r}\}}{\Pr\{u_i = 0 | \bar{r}\}} \right) \quad (3.23)$$

Where \bar{r} refers to the received sequence.

In a SOVA decoder, the Viterbi algorithm needs to be executed twice. In the first time, decoding proceeds as with the conventional VA, with the exception that path metrics at each decoding stage are needed to be stored. This part is referred as forward processing. The second part is referred as backward processing. In the second part the VA transverses

the trellis backwards, and computes metrics and paths, starting at $i = N - 1$ and ending at $i = 0$. In this stage, there is no need to store the surviving paths, but only the metrics for each trellis state. Finally for each trellis stage, i , $1 \leq i \leq N$, the soft-outputs are computed (Morelos-Zaragoza 2002).

M_{\max} can be used to denote the metric of the most likely sequence \hat{v} found by the Viterbi algorithm. A-Posteriori Probability (APP) of the associated information sequence \hat{u} given the received sequence \bar{r} , is proportional to M_{\max} , while

$$\Pr\{\hat{u} | \bar{r}\} = \Pr\{\hat{v} | \bar{r}\} \approx e^{M_{\max}}. \quad (3.24)$$

From (3.21) and (3.22), the APP of information bit u_i can be written as

$$\Pr\{u_i = 1 | \bar{r}\} \approx e^{M_i(1)},$$

where $M_i(1) = M_{\max}$. Now, let assume $M_i(0)$ denote the maximum metric of paths associated with the complement of information symbol u_i . Then the soft-output

$$\Lambda(u_i) \approx M_i(1) - M_i(0). \quad (3.25)$$

Hence, at time i , the soft-output can be obtained from the difference between the maximum metric of paths in the trellis with $\hat{u}_i = 1$ and the maximum metric of paths with $\hat{u}_i = 0$.

In the SOVA algorithm, at stage i , the most likely information symbol $u_i = j$, $j \in \{0, 1\}$, is determined and the corresponding maximum metric which is found in the forward pass of the VA is set to $M_i(u_i)$. The path metric of the best competitor, $M_i(u_i \oplus 1)$, can be computed as (Vucetic and Yuan 2000),

$$M_i(v_i \oplus 1) = \min_{k_1, k_2} \{M_f(S_{i-1}^{(k_1)}) + BM_i^{(b_1)}(u_i \oplus 1) + M_b(S_i^{(k_2)})\} , \quad (3.26)$$

where $k_1, k_2 \in \{0, 1, 2, \dots, 2^m - 1\}$,

- $M_f(S_{i-1}^{(k_1)})$ is the path metric of the forward survivor at time $i-1$ and state $S^{(k_1)}$,
- $BM_i^{(b_1)}(u_i \oplus 1)$ is the branch metric at time i for the complement information associated with a transition from state $S^{(k_1)}$ to $S^{(k_2)}$, and
- $M_b(S_i^{(k_2)})$ is the path metric of the backward survivor at time i and state $S^{(k_2)}$.

Lastly, the soft-output is computed as

$$\Lambda(u_i) = M_i(1) - M_i(0). \quad (3.27)$$

3.3.3 Maximum- a – Posteriori (MAP) algorithm

The symbol-by-symbol maximum a-posteriori (MAP) algorithm was formally presented in 1974 by Bahl et al as an alternative to the Viterbi algorithm for decoding convolutional codes (Bahl, Cocke, Jelinek, and Raviv 1974, vol. IT-20, pp. 284-287). The MAP algorithm finds the most likely information bit that has been transmitted in a coded sequence. This is unlike the Viterbi algorithm which finds the most likely sequence that has been transmitted. The difference of error performance between the MAP and Viterbi algorithm is negligible when the decoded bit-error rate (BER) is small. The Map algorithm has been largely ignored as it is significantly more complex than Viterbi algorithm. However, at low signal-to-noise ratio (SNR) or E_b/N_0 and high bit-error rate (BER), MAP can do better than SOVA by 0.5 dB or more. For turbo code it is very important as the output BER's from the first stages of iterative decoding can be very high. Hence, any improvement that can be obtained at these high BERs will directly result in performance increases.

The MAP algorithm calculates the a-posteriori probability (APP) of each state transition, given the noisy observation \bar{r} . The state transitions in the trellis have probabilities

$$\Pr\{S_i^{(m)} | S_{i-1}^{(m')}\}, \quad (3.28)$$

and for the output symbols \bar{v}_i ,

$$q_i(x_i | m', m) \equiv \Pr\{x_i = x | S_{i-1}^{(m')}, S_i^{(m)}\}, \quad (3.29)$$

where $x = \pm 1$, and $x_i = m(v_i) = (-1)^{v_i}$, $0 < i \leq N$.

The sequence \bar{x} is transmitted over AWGN channel and received as a sequence \bar{r} , with transition probabilities

$$\Pr\{\bar{r} | \bar{x}\} = \prod_{i=1}^N p(\bar{r}_i | \bar{x}_i) = \prod_{i=1}^N \prod_{j=0}^{n-1} p(r_{i,j} | x_{i,j}), \quad (3.30)$$

where $p(r_{i,j} | x_{i,j})$ is given by

$$p(r_i | v_i) = p_{n_i}(r_i - m(v_i)). \quad (3.31)$$

Let, $B_i^{(j)}$ be the set of branches connecting state $S_{i-1}^{m'}$ to state $S_i^{(m)}$ such that the associated information bit $u_i = j$, $j \in \{0, 1\}$, Then

$$\Pr\{u_i | j | \bar{r}\} = \sum_{(m', m) \in B_i^{(j)}} \Pr\{S_{i-1}^{(m')}, S_i^{(m)}, \bar{r}\} \equiv \sum_{(m', m) \in B_i^{(j)}} \sigma_i(m', m). \quad (3.32)$$

Here, the value of $\sigma_i(m', m)$ is equal to

$$\sigma_i(m', m) = \alpha_{i-1}(m') \cdot \gamma_i^{(j)}(m', m) \cdot \beta_i(m), \quad (3.33)$$

where the joint probability $\alpha_i(m) \equiv \Pr\{S_i^{(m)}, \bar{r}_p\}$ is given recursively by

$$\alpha_i(m) = \sum_{m'} \alpha_{i-1}(m') \cdot \sum_{j=0}^1 \gamma_i^{(j)}(m', m), \quad (3.34)$$

and is referred to as the forward metric.

The conditional probability $\gamma_i^{(j)}(m', m) \equiv \Pr\{S_i^{(m)}, \bar{r} | S_{i-1}^{(m')}\}$ is given by

$$\gamma_i^{(j)}(m', m) = \sum_x p_i(m | m') \Pr\{x_i = x | S_{i-1}^{(m')}, S_i^{(m)}\} \cdot \Pr\{r_i | x\} \quad (3.35)$$

where $p_i(m | m') = \Pr\{S_i^{(m)} | S_{i-1}^{(m')}\}$, which for the AWGN channel can be put in the form

$$\gamma_i^{(j)}(m', m) = \Pr\{u_i = j\} \cdot \delta_{ij}(m, m') \cdot \exp\left(-\frac{1}{N_0} \sum_{q=0}^{n-1} (r_{i,q} - x_{i,q})^2\right), \quad (3.36)$$

where $\delta_{ij}(m, m') = 1$ if $\{m', m\} \in B_i^{(j)}$, and $\delta_{ij}(m, m') = 0$ otherwise. $\gamma_i^{(j)}(m', m)$ is referred to as the branch metric.

The conditional probability $\beta_i(m) \equiv \Pr\{\bar{r}_f | S_i^{(m)}\}$ is given by

$$\beta_i(m) = \sum_{m'} \beta_{i+1}(m') \cdot \sum_{j=0}^1 \gamma_{i+1}^{(j)}(m', m), \quad (3.37)$$

and referred to as the backward metric.

Combining all the above equations, the soft output (LLR) of information bit u_i is given by

$$\Lambda(u_i) = \log \left(\frac{\Pr\{u_i = 1 | \bar{r}\}}{\Pr\{u_i = 0 | \bar{r}\}} \right) = \log \left(\frac{\sum_m \sum_{m'} \alpha_{i-1}(m') \cdot \gamma_i^{(1)}(m', m) \cdot \beta_i(m)}{\sum_m \sum_{m'} \alpha_{i-1}(m') \cdot \gamma_i^{(0)}(m', m) \cdot \beta_i(m)} \right), \quad (3.38)$$

with the hard-decision output is given by $\hat{u}_i = \text{sgn}(\Lambda(u_i))$ and the reliability of the bit u_i is $|\Lambda(u_i)|$.

The MAP algorithm proceeds as follows

1. Forward recursion.

- (a) Create an array $\alpha(j, i), 0 \leq j \leq 2^m - 1, 0 \leq i \leq N$. This array will store the results of the forward recursion and is initialized as follows:

$$\alpha(j, 0) = \begin{cases} 1 & \text{if } j = 0 \\ 0 & \text{if } j \neq 0 \end{cases}$$

- (b) Begin with time index $i = 1$.
(c) For $m = 0, 1, \dots, 2^m - 1$, compute α according to

$$\alpha_i(m) = \sum_{m'} \alpha_{i-1}(m') \cdot \sum_{j=0}^1 \gamma_i^{(j)}(m', m),$$

- (d) Increment i .
(e) If $i = N$, the end of trellis has been reached - continue to step 2. Otherwise, return to step 1(c).

2. Backward recursion.

- (a) Create an array $\beta(j, i), 0 \leq j \leq 2^m - 1, 0 \leq i \leq N$. This array will store the results of the backward recursion and is initialized as follows:

$$\beta(j, N) = \begin{cases} 1 & \text{if } j = 0 \\ 0 & \text{if } j \neq 0 \end{cases}$$

Otherwise, if the trellis is not terminated

$$\beta(j, N) = \frac{1}{2^m} \forall j$$

- (b) Begin with time index $i = N-1$.
(c) For $m = 0, 1, \dots, 2^m-1$, compute β according to

$$\beta_i(m) = \sum_{m'} \beta_{i+1}(m') \cdot \sum_{j=0}^1 \gamma_{i+1}^{(j)}(m', m),$$

- (d) Decrement i .
(e) If $i = 0$, the end of trellis has been reached - continue to step 3. Otherwise, return to step 2(c).
3. For $i = (0, \dots, N-1)$, compute the LLR according to

$$\Lambda(u_i) = \log \left(\frac{\sum_m \sum_{m'} \alpha_{i-1}(m') \cdot \gamma_i^{(1)}(m', m) \cdot \beta_i(m)}{\sum_m \sum_{m'} \alpha_{i-1}(m') \cdot \gamma_i^{(0)}(m', m) \cdot \beta_i(m)} \right).$$

3.3.4 Max -Log- MAP and Log- MAP algorithms

In principle, the MAP algorithm is able to calculate precise estimates of the a-posteriori probability of each message bit. However, it has two severe practical problems. First, it is computationally intensive. Second it is sensitive to the round-off errors that occur when representing numerical values with finite precision. These two problems can be

assuaged by performing the entire algorithm in the log-domain, rather than waiting until the last step to take the logarithm of the likelihood ratio.

To reduce the computational complexity of the MAP algorithm, the logarithms of the metrics may be used. This result is called log-MAP algorithm. From (3.34) and (3.37) (Robertson, Villeburn and Hoeher 1995, pp. 1009-1013),

$$\begin{aligned}\log \alpha_i(m) &= \log \left(\sum_{m'} \sum_{j=0}^1 \exp(\log \alpha_{i-1}(m') + \log \gamma_i^{(j)}(m', m)) \right), \\ \log \beta_i(m) &= \log \left(\sum_{m'} \sum_{j=0}^1 \exp(\log \beta_{i+1}(m') + \log \gamma_i^{(j)}(m', m)) \right).\end{aligned}\quad (3.39)$$

Taking the logarithm of $\gamma_i^{(j)}(m', m)$ in (3.36),

$$\log \gamma_i^{(j)}(m', m) = \delta_{ij}(m', m) \left\{ \log \Pr\{u_i = j\} - \frac{1}{N_0} \sum_{q=0}^{n-1} (r_{i,q} - x_{i,q})^2 \right\} \quad (3.40)$$

By defining $\bar{\alpha}_i(m) = \log \alpha_i(m)$, $\bar{\beta}_i(m) = \log \beta_i(m)$ and $\bar{\gamma}_i^{(j)}(m', m) = \log \gamma_i^{(j)}(m', m)$, (3.38) can be written as

$$\Lambda(u_i) = \log \left(\frac{\sum_m \sum_{m'} \exp(\bar{\alpha}_{i-1}(m') + \bar{\gamma}_i^{(0)}(m', m) + \bar{\beta}_i(m))}{\sum_m \sum_{m'} \exp(\bar{\alpha}_{i-1}(m') + \bar{\gamma}_i^{(1)}(m', m) + \bar{\beta}_i(m))} \right), \quad (3.41)$$

and an algorithm that works in the log-domain is obtained.

A computationally efficient, suboptimal derivative of the MAP algorithm is the Max-Log-MAP algorithm. The main benefit of executing the algorithm in the Max-Log-MAP algorithm is that multiplication becomes addition. However in this case, addition is not

straight-forward. According to Jacobian logarithm, addition is performed in the log-domain as follows (Robertson, Villeburn and Hoeher 1995, pp. 1009-1013):

$$\begin{aligned}\log(e^{\delta_1} + e^{\delta_2}) &= \max(\delta_1, \delta_2) + \log(1 + e^{-|\delta_1 - \delta_2|}) \\ &= \max(\delta_1, \delta_2) + f_c(|\delta_1 - \delta_2|).\end{aligned}\quad (3.42)$$

In addition, it can be noted that δ_1 and δ_2 are not similar and the correction function $f_c(\cdot)$ is close to zero. Thus, reasonable approximation to the above is found by,

$$\log(e^{\delta_1} + e^{\delta_2}) \approx \max(\delta_1, \delta_2). \quad (3.43)$$

As a result, the log-likelihood ratio of information bit u_i is given by

$$\begin{aligned}\Lambda(u_i) &\approx \max_{m', m} \{ \bar{\alpha}_{i-1}(m') + \bar{\gamma}^{(0)}(m', m) + \bar{\beta}_i(m) \} \\ &\quad - \max_{m', m} \{ \bar{\alpha}_{i-1}(m') + \bar{\gamma}^{(1)}(m', m) + \bar{\beta}_i(m) \}.\end{aligned}\quad (3.44)$$

The forward and backward computation can now be expressed as

$$\begin{aligned}\bar{\alpha}_i(m) &= \max_{m'} \max_{j \in \{0,1\}} \{ \bar{\alpha}_{i-1}(m') + \bar{\gamma}^{(j)}(m', m) \}, \\ \bar{\beta}_i(m) &= \max_{m'} \max_{j \in \{0,1\}} \{ \bar{\beta}_{i+1}(m') + \bar{\gamma}^{(j)}(m', m) \}.\end{aligned}\quad (3.45)$$

Thus, the max-log-MAP algorithm proceeds as follows:

1. Forward recursion.

- (a) Create an array $\bar{\alpha}(j, i), 0 \leq j \leq 2^m - 1, 0 \leq i \leq N$. This array will store the results of the forward recursion and is initialized as follows:

$$\bar{\alpha}(j,0) = \begin{cases} 1 & \text{if } j = 0 \\ -\infty & \text{if } j \neq 0 \end{cases}$$

- (b) Begin with time index $i = 1$.
(c) For $m = 0, 1, \dots, 2^m - 1$, compute $\bar{\alpha}$ according to

$$\bar{\alpha}_i(m) = \max_{m'} \max_{j \in \{0,1\}} \{ \bar{\alpha}_{i-1}(m') + \bar{\gamma}^{(j)}(m', m) \}.$$

- (d) Increment i .
(e) If $i = N$, the end of trellis has been reached - continue to step 2. Otherwise, return to step 1(c).

2. Backward recursion.

- (f) Create an array $\bar{\beta}(j,i), 0 \leq j \leq 2^m - 1, 0 \leq i \leq N$. This array will store the results of the backward recursion and is initialized as follows:

$$\bar{\beta}(j,N) = \begin{cases} 1 & \text{if } j = 0 \\ -\infty & \text{if } j \neq 0 \end{cases}$$

Otherwise, if the trellis is not terminated

$$\bar{\beta}(j,N) = 0 \quad \forall j.$$

- (g) Begin with time index $i = N-1$.
(h) For $m = 0, 1, \dots, 2^m - 1$, compute $\bar{\beta}$ according to

$$\bar{\beta}_i(m) = \max_{m'} \max_{j \in \{0,1\}} \{ \bar{\beta}_{i+1}(m') + \bar{\gamma}^{(j)}(m', m) \}.$$

- (i) Decrement i .
 - (j) If $i = 0$, the end of trellis has been reached - continue to step 3. Otherwise, return to step 2(c).
3. For $i = (0, \dots, N-1)$, determine the LLR according to

$$\Lambda(u_i) \approx \max_{m',m} \{ \bar{\alpha}_{i-1}(m') + \bar{\gamma}^{(0)}(m',m) + \bar{\beta}_i(m) \} \\ - \max_{m',m} \{ \bar{\alpha}_{i-1}(m') + \bar{\gamma}^{(1)}(m',m) + \bar{\beta}_i(m) \}.$$

For binary codes based on rate- $1/n$ convolutional encoders, in terms of decoding complexity, SOVA requires half of that of the max-log-MAP algorithm. The log-MAP algorithm is approximately twice more complex compared to the max-log-MAP algorithm.

3.4 Encoding of turbo codes

Figure 3.8 shows the block diagram of an encoder of a parallel concatenated code, better known as a turbo code. The turbo encoder is composed of two RSC encoders, which are usually identical. The first encoder receives the actual data and the second encoder receives the data after being permuted by an interleaver. Turbo codes appear random only because of this interleaver. As the interleaver must have fixed structure and generally works data in a block-wise manner, turbo codes are by necessity block codes. If the interleaver has a fixed size and both RSC encoders start with the all-zeros state, then turbo codes are linear block code.

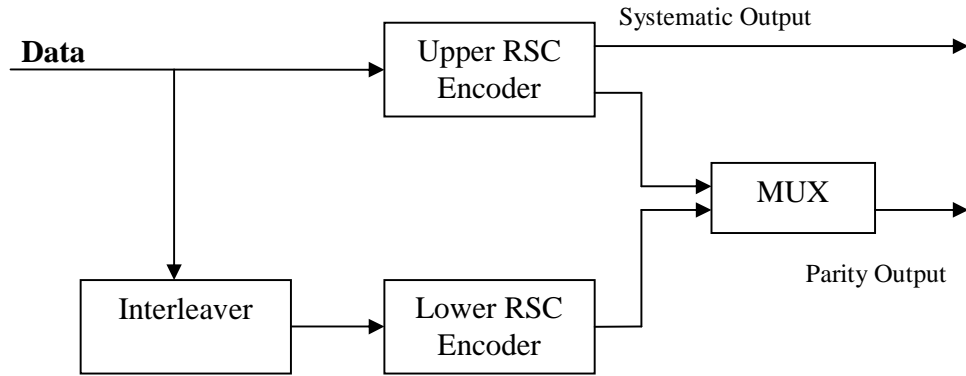


Figure 3.8: Encoder Structure of turbo code.

In Figure 3.8 rate- 1/2 RSC codes are used and the encoders represent the term $\bar{g}_1(D)/\bar{g}_2(D)$ in the polynomial generator matrix $G(D)$. The inputs to the encoders are \bar{u} and $\Pi\bar{u}$, where Π denotes a permutation matrix³ associated with an interleaver.

The systematic output of the turbo encoder, $v_i^{(0)}$ is taken from the first (upper) RSC encoder. The two parity outputs, $v_i^{(1)}$ and $v_i^{(2)}$ are taken from the first (upper) and second (lower) RSC encoders' parity outputs, respectively, for $i = 1, 2, \dots, N$, where N is the block length. The output streams are multiplexed to form the code word $v = (v_0^{(0)}, v_0^{(1)}, v_0^{(2)}, \dots, v_{N-1}^{(0)}, v_{N-1}^{(1)}, v_{N-1}^{(2)})$. In general, code rate of a turbo code is 1/3. As with convolutional codes, this rate can be increased by puncturing. As for instance, a rate 1/2 turbo code can be achieved from the rate 1/3 turbo code by using the following puncturing matrix

³ A permutation matrix is a binary matrix with only one nonzero entry per row and per column.

$$P = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{3.46}$$

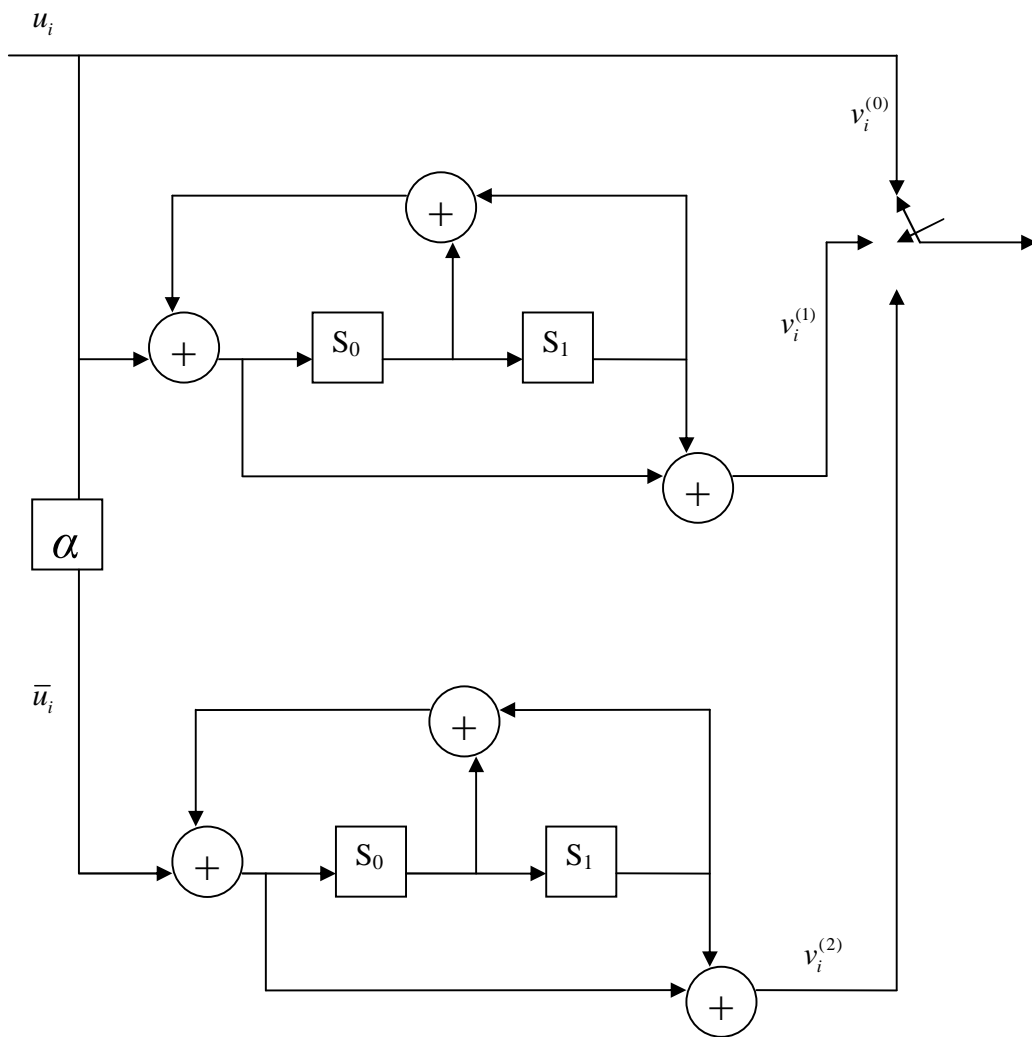


Figure 3.9: Example of turbo encoder.

3.5 Overview of turbo decoding

3.5.1 Soft-input, soft- output decoding

Soft-bit decisions typically are represented as (a-posteriori) log-likelihood ratios (LLRs) of the form

$$\Lambda(u_i) = \log \left(\frac{\Pr\{u_i = 1 | \bar{r}\}}{\Pr\{u_i = 0 | \bar{r}\}} \right). \quad (3.47)$$

A soft-input, soft-output (SISO) decoding algorithm accepts a priori information at its input and produces a-posteriori information at its output. Both the Viterbi algorithm and MAP algorithm can be modified to accept soft-input and produce soft-outputs.

The systems employed by the binary phase shift keying (BPSK) is characterized by the following input/output relationship

$$r' = a\sqrt{E_s}(2v-1) + n', \quad (3.48)$$

where a is a fading amplitude, $\sqrt{E_s}(2v-1)$ is the BPSK modulated code symbol ($v \in \{0,1\}$), E_s ⁴ is the energy per code symbol, and n' is a zero-mean Gaussian random variable with variance $\sigma^2 = N_0/2$. When a is a constant, the channel is said to be an additive white Gaussian noise (AWGN) channel. Otherwise, the channel is said to be a flat-fading channel, i.e. flat Rayleigh fading channel. An alternative and most convenient expression for (3.48) is

$$r = a(2v-1) + n, \quad (3.49)$$

⁴ E_s is related to the energy per bit E_b and the code rate n by $E_s = nE_b$

where the noise variance is now $\sigma^2 = N_0 / 2E_s$.

The log-likelihood at the output of a SISO decoder using any channel model can be expressed as

$$\Lambda(u_i) = \Lambda_{ci} + \Lambda_a(u_i) + \Lambda_{i,e}(C), \quad (3.50)$$

where Λ_{ci} is known as the channel LLR and given by

$$\Lambda_{ci} = \log \left(\frac{p(r_i | x_i = -1)}{p(r_i | x_i = +1)} \right) = \begin{cases} (-1)^{r_i} \log \left(\frac{1-p}{p} \right), & \text{for a BSC channel with parameter } p, \\ \frac{4}{N_0} r_i, & \text{for an AWGN channel } (\sigma_n^2 = N_0/2); \text{ and} \\ \frac{4}{N_0} a_i r_i, & \text{for a flat Rayleigh fading channel,} \end{cases} \quad (3.51)$$

the quantity

$$\Lambda_a(u_i) = \log \left(\frac{\Pr\{x_i = -1\}}{\Pr\{x_i = +1\}} \right) = \log \left(\frac{\Pr\{u_i = 0\}}{\Pr\{u_i = 1\}} \right), \quad (3.52)$$

is the a-priori LLR of the information symbol, and

$$\Lambda_{i,e}(C) = \log \left(\frac{\sum_{\bar{x} \in C_i(1)} \prod_{\substack{l=1 \\ l \neq i}}^N p(r_l, x_l)}{\sum_{\bar{x} \in C_i(0)} \prod_{\substack{l=1 \\ l \neq i}}^N p(r_l, x_l)} \right), \quad (3.53)$$

is the extrinsic LLR, which is specified by the constraints imposed by the code on the other information symbols.

As mentioned previously, in turbo code's generation two binary rate-1/2 RSC encoders are used as components. In an iterative decoding procedure, the extrinsic information provided by $\Lambda_{i,e}(C)$ can be fed back to the decoder as a a-priori probability for a second round of decoding. According to equation (3.38) in section 3.3.3, the extrinsic LLR can be written as

$$\Lambda_{i,e}(C) = \log \left(\frac{\sum_{(m,m') \in B_i(1)} \alpha_{i-1}(m') \xi_i(m',m) \beta_i(m)}{\sum_{(m,m') \in B_i(0)} \alpha_{i-1}(m') \xi_i(m',m) \beta_i(m)} \right), \quad (3.54)$$

where $\alpha_i(m)$ and $\beta_i(m)$ are given by (3.34) and (3.37), respectively.

A modified branch metric is needed to compute the extrinsic LLR,

$$\xi_i(m',m) = \delta_{ij}(m,m') \exp \left(\frac{E}{N_0} \sum_{q=1}^{n-1} r_{i,q}, x_{i,q} \right), \quad (3.55)$$

where, as before, $\delta_{ij}(m,m') = 1$ if $\{m',m\} \in B_i^{(j)}$, and $\delta_{ij}(m,m') = 0$ otherwise.

3.5.2 Turbo decoder schematic

The schematic for turbo decoder is shown in Figure 3.10. Assuming zero decoder delay in the turbo-decoder, the decoder 1 computes a soft-output $\Lambda^{(1)}$ from the systematic data, code information of encoder 1 and a-priori information $\Lambda_{i,e}^{(2)}(C)$. From this output, the systematic data and a-priori information are $\Lambda_{i,e}^{(2)}(C)$ subtracted to get the extrinsic information $\Lambda_{i,e}^{(1)}(C)$ of the decoder 1. The extrinsic information of the decoder 1 is interleaved, and used as a priori information by the decoder 2. Decoder 2 takes as input the interleaved version of $\Lambda_{i,e}^{(1)}(C)$, the code information of second encoder and the interleaved

version of systematic data. Decoder 2 generates a soft output $\Lambda^{(2)}$, from which the systematic data and a-priori information $\Lambda_{i,e}^{(1)}(C)$ are subtracted to get the extrinsic information $\Lambda_{i,e}^{(2)}(C)$ of the decoder 2. The extrinsic information produced by the decoder 2 is deinterleaved and used as the a priori input to the decoder 1 during the next iteration. The multiplications are the scaling factor and called channel reliability L_c to compensate the distortion. After certain number of iterations, the final estimate of the message is found by deinterleaving and hard-limiting the output of the second decoder.

$$\hat{u} = \begin{cases} 1 & \text{if } \Lambda^{(2)} \geq 0 \\ 0 & \text{if } \Lambda^{(2)} < 0 \end{cases}$$

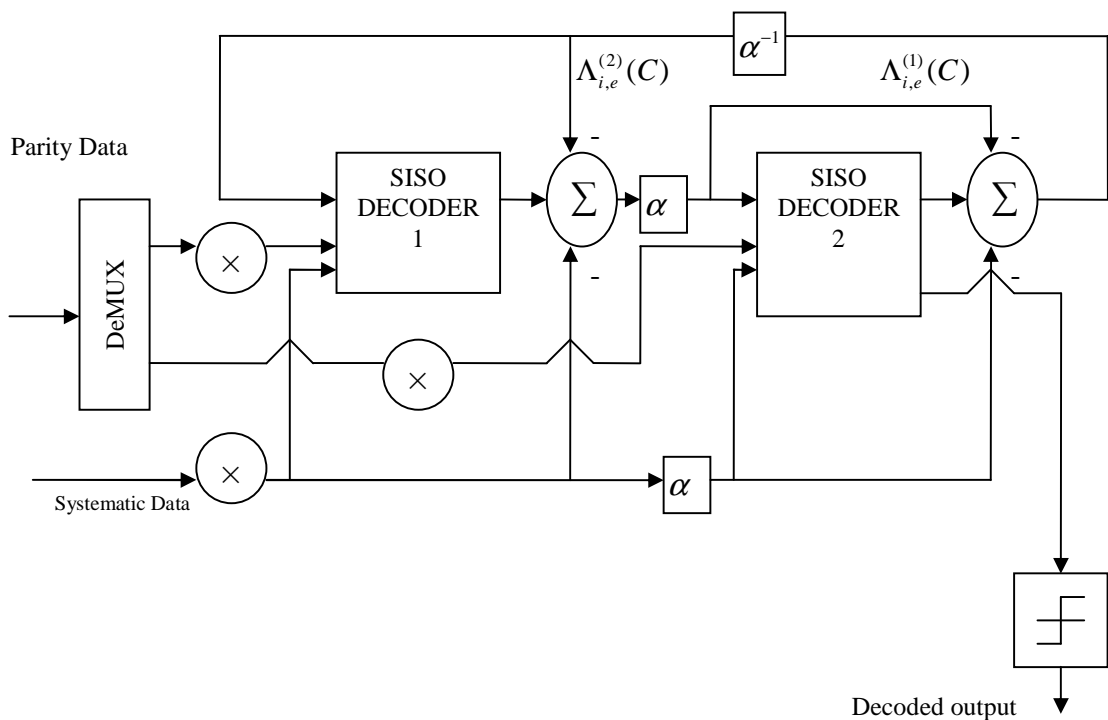


Figure 3.10: Turbo decoder schematic.

3.5.3 Example of turbo decoding

Figure 3.11 shows simulation results of turbo codes for a block size of 1024. It can be noticed in the figure that the BER improves dramatically during the first few iterations. During later iterations, there is still improvement; however the amount of improvement begins to reduce.

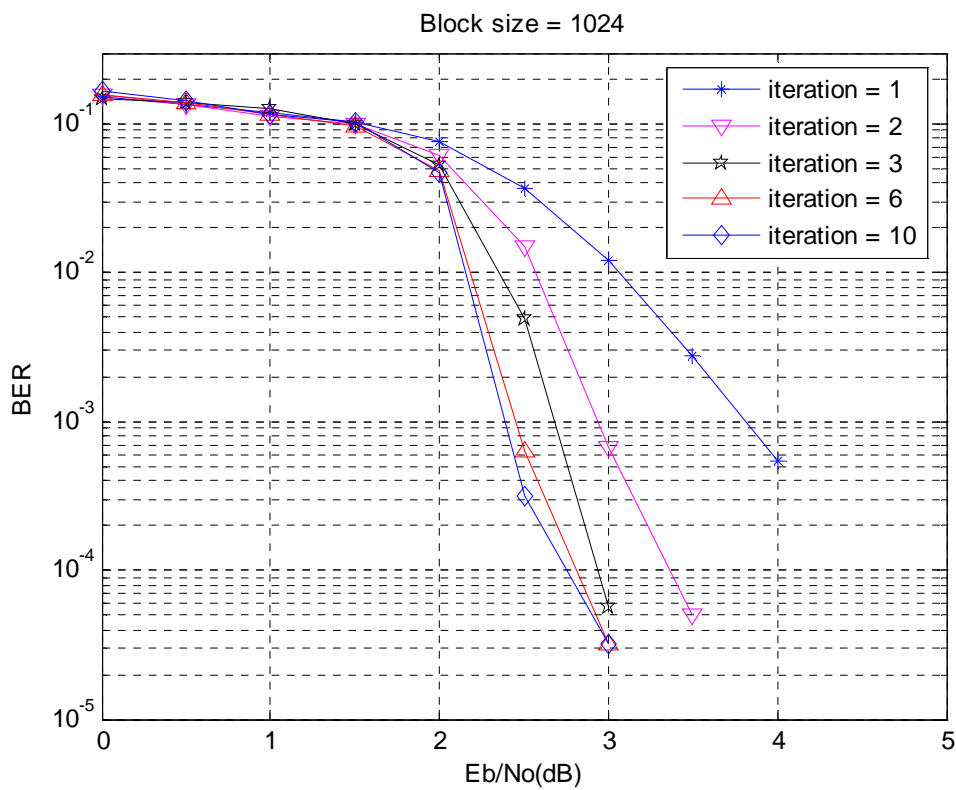


Figure 3.11: Performance of rate-1/2 parallel concatenated (turbo) code with memory-4, rate-1/2 RSC codes, generators (37, 21) and block size = 1024.

3.6 Chapter summary

This chapter has presented an overview of coding theory and of turbo decoding. In order to introduce notation and terminology, a review of block and convolutional codes has been presented. Two representations for convolutional codes have been discussed – the state diagram and the trellis diagram. It has been shown how the rate of the convolutional codes can be increased by using puncturing concept. An overview of the trellis-based soft-input soft-output decoding algorithms has been presented. The algorithms are based on either the Viterbi algorithm or the MAP algorithm. The soft-output Viterbi algorithm (SOVA) is an extension of the Viterbi algorithm that provides the reliability of the bit estimates. The MAP algorithm calculates the a-posteriori probabilities directly. The max-log-MAP and log-MAP algorithms perform the MAP algorithm in the log domain.

Once a foundation of the basic concepts of coding has been presented, turbo codes have been introduced. It has been shown that a turbo code is the parallel concatenation of two RSC codes that are given the same data in permuted order. The importance of soft-input soft-output components decoders is discussed in the context of the decoding of turbo codes. The concept of extrinsic information has been introduced, and the schematic for a typical turbo decoder has been presented.

Chapter 4

LDPC codes: Code construction and decoding

LDPC codes (also referred to as Gallager codes (Gallager 1962, vol. 8, no. 1, pp. 21-28)) have greater performance than turbo codes in some cases. The implementations of iterative decoding algorithms are easy with LDPC codes as the per-iteration complexity much lower than that of turbo decoders. LDPC codes are also parallelizable in hardware. This chapter focuses on the code construction and decoding algorithm of LDPC codes.

4.1 Code construction

Throughout this chapter, N will be used to denote the length of the code and K to denote its dimension and the redundancy $M = N - K$ ⁵. A low density parity check code is a linear block code which has a very sparse parity check matrix \mathbf{H} . Since the parity check matrices

⁵ In previous chapters, n , k , and m are used to describe the code. In this chapter we use n and m as indices, suggesting by them that they are index length and redundancy components.

$$A = \begin{bmatrix} \pi_1(A_0) \\ \pi_2(A_0) \\ \pi_3(A_0) \\ \vdots \\ \pi_{w_c}(A_0) \end{bmatrix} \quad (4.2)$$

where each $\pi_i(A_0)$ denotes a matrix obtained by permuting the columns of A_0 . Obviously, the distance structure of the code is determined by the choice of the permutations. However, a random choice of permutation usually produces a good code. Gallager showed that if each permutation is chosen at random out of the $N!$ possible permutations, then the average minimum distance d_{\min} increases linearly with N . Such codes are called good codes.

An example of an LDPC parity check matrix for a (3, 4)-regular LDPC code in Gallager's paper (Gallager 1962, vol. 8, no. 1, pp. 21-28),

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

It can be noticed, there are 13 linearly independent rows in the above matrix.

The above constructions do not ensure that all the rows of the matrix are linearly independent, so the $M \times N$ matrix created is the parity check matrix of a linear code with rate at least $R \equiv K/N$, where as previously mentioned $K = N - M$.

4.2 Tanner graphs

LDPC codes are linear codes obtained from sparse⁶ bipartite graphs which are also known as Tanner graphs (Morelos-Zaragoza 2002, p 159). For a linear (w_c, w_r, N) code C , there exists a Tanner graph G with N left nodes (called message nodes or code nodes), x_i , associated with code symbols, and at least $M = N - K$ right nodes (called check nodes or parity nodes), z_m , associated with parity check equations. The codewords are those vectors (c_1, \dots, c_n) such that for all check nodes the sum of the neighboring positions among the message nodes is zero. Figure 4.1 illustrates the graph for A from the above example. A graph such as this, consisting of two distinct sets of nodes and having edges only between the nodes in different sets, is called a bipartite graph. For a regular LPDC code, the number edges from the code nodes or the number of degrees are equal to w_c and the degrees of the check nodes equal to w_r .

However, not every binary linear code has a representation by a sparse bipartite graph. If it does, then the code is called a low-density parity-check (LDPC) code.

Tanner graphs can be used to estimate codewords of an LPDC code C by iterative probabilistic decoding algorithms, based on either hard or soft decisions. In Section 4.4 the basic iterative decoding algorithm introduced by Gallager is presented.

⁶ To be more precise, sparseness only applies to sequences of matrices. A sequence of $m \times n$ -matrices is called c -sparse if mn tends to infinity and the number of nonzero elements in these matrices is always less than c , where $c = \max(m, n)$.

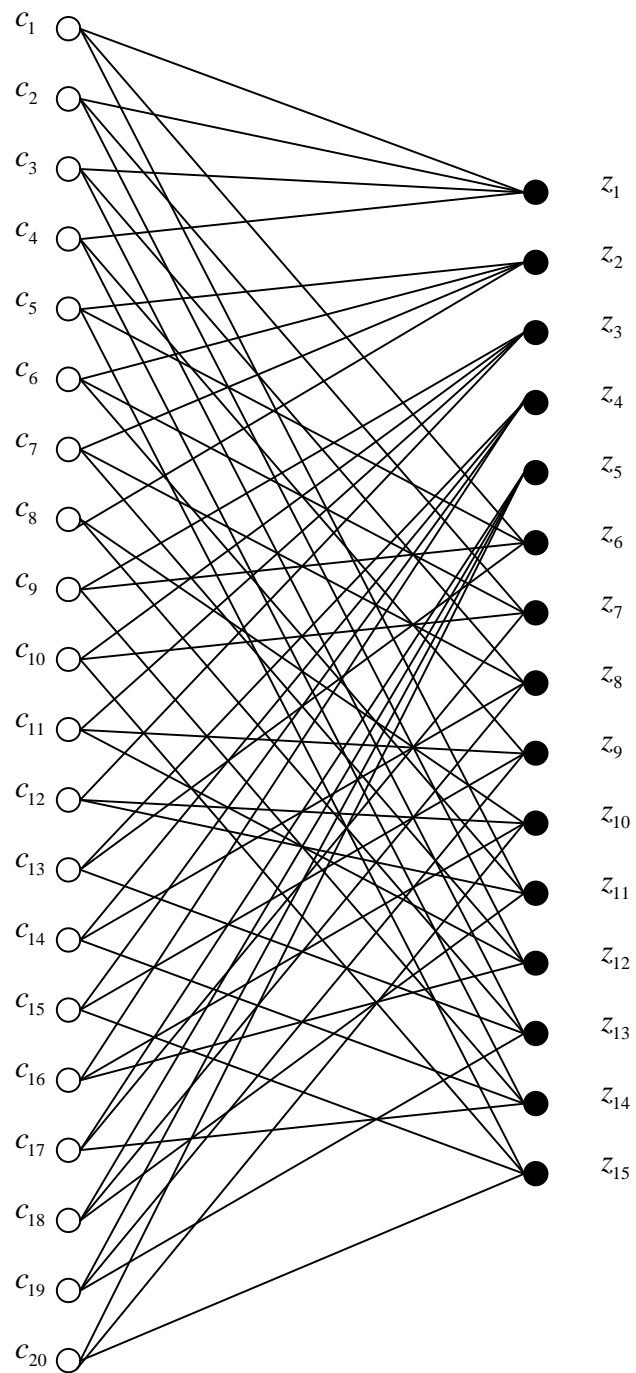


Figure 4.1: Tanner graph of Gallager's (3, 4, 20) code.

4.3 Encoding algorithm

An encoding algorithm for a binary linear code of dimension K and block length N is an algorithm that computes a codeword C from K original bits (x_1, x_2, \dots, x_K) . Although LDPC codes have an efficient decoding algorithm, with linear complexity in the code length, the encoding efficiency is quadratic in the block length, since it requires multiplication by the generator matrix which is not sparse. However, (Richardson T J and Urbanke R L 2001, vol. 47, no. 2, February, pp. 638-656) presents that it is possible to encode with a reasonable complexity, with some previously performed processing prior to encoding.

Before encoding, the following steps are performed. By row and column permutations, the parity check matrix H is brought into the form as showed in Figure 4.2. It can be noticed from the figure, that the upper right corner of the newly formed H is a lower triangular matrix. Because it is obtained only by permutations, the H matrix is still sparse. The newly formed H can be denoted as

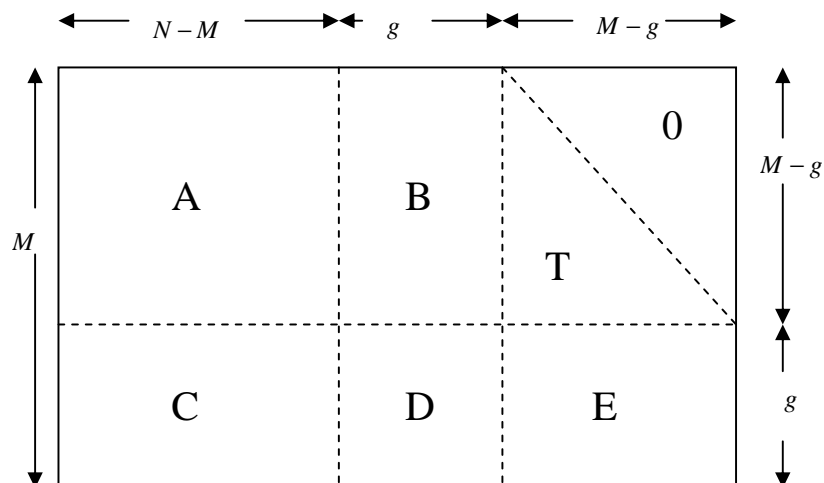


Figure 4.2: Result of permuted rows and columns.

$$H = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix} \quad (4.3)$$

and it can be said that H is in approximate lower triangular form. From the Figure 4.2, g is the gap of this representation. T is a $(M - g) \times (M - g)$ lower triangular matrix with ones along the diagonal and hence is invertible. Now H is multiplied by the matrix

$$H = \begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix} \quad (4.4)$$

to doing Gaussian elimination to clear the matrix E , which produces the form

$$\tilde{H} = \begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix} H = \begin{bmatrix} A & B & T \\ -ET^{-1}A + C & -ET^{-1}B + D & 0 \end{bmatrix}. \quad (4.5)$$

For a message vector \mathbf{u} of length K , the codeword is as followed,

$$c = \begin{bmatrix} \mathbf{u} \\ p_1 \\ p_2 \end{bmatrix} \quad (4.6)$$

where p_1 and p_2 represent parity information. The parity check equation $\tilde{H}c = 0$ gives rise to two equations,

$$A\mathbf{u} + Bp_1 + Tp_2 = 0 \quad (4.7)$$

$$(-ET^{-1}A + C)\mathbf{u} + (-ET^{-1}B + D)p_1 = 0. \quad (4.8)$$

In (4.7), X can be used to denote $(-ET^{-1}B + D)$ by assigning $X = (-ET^{-1}B + D)$. So, from (4.7)

$$\mathbf{p}_1 = -X^{-1}(-ET^{-1}A + C)\mathbf{u}. \quad (4.9)$$

Here, it is assumed that X is nonsingular. If it turns out that X is singular, then columns of \tilde{H} can be permuted to obtain a nonsingular X .

The $g \times (N - M)$ matrix $-X^{-1}(-ET^{-1}A + C)$ can be previously computed and saved, so that \mathbf{p}_1 can be computed with a complexity of $O(g(N - M))$. Once \mathbf{p}_1 is obtained, \mathbf{p}_2 can be calculated as followed,

$$\mathbf{p}_2 = -T^{-1}(A\mathbf{u} + B\mathbf{p}_1). \quad (4.10)$$

The process of computing \mathbf{p}_1 and \mathbf{p}_2 comprises the encoding process. The overall encoding algorithm has the complexity of $O(N + g^2)$. Hence, it is clear that the complexity of the algorithm will be less with smaller g .

4.4 Overview of LPDC decoding

4.4.1 The sum-product algorithm

In this section, an iterative belief-propagation (IBP) decoding algorithm is presented. This algorithm is also known as “sum-product decoder”. Mr. David J. C. MacKay in his paper “Good Error-Correcting Codes Based on Very Sparse Matrices” gave an extensive description of this decoding algorithm (MacKay 1999, vol. 45, no. 2, pp. 399-432).

They referred to the elements z_m corresponding to each row $m = 1 \dots M$ of \mathbf{H} as checks. They assumed the set of bits $\bar{\mathbf{v}}$ and checks $\bar{\mathbf{z}}$ as making up a “belief

network”, in which every bit v_l is the parent of w_c checks z_m , and every check z_m is the child of w_r bits.

This algorithm is appropriate for a binary channel model in which the noise bits are independent – for example, the memory-less binary-symmetric channel, or the Gaussian channel with binary inputs and real outputs. The following notation is convenient in describing algorithm. Let h_{ij} denote the entry of \mathbf{H} in the i -th row and j -th column. Let

$$L(m) = \{l : h_{m,l} = 1\}, \quad (4.11)$$

denote the set of bits l that participate in the m -th parity-check equation. Similarly, the set of checks in which bit l participates,

$$M(l) = \{m : h_{m,l} = 1\}. \quad (4.12)$$

The algorithm has two alternating parts, in which quantities q_{ml}^x and r_{ml}^x associated with each nonzero element in the \mathbf{H} matrix are iteratively updated. The quantity q_{ml}^x is meant to be the probability that the l -th bit of \bar{v} has the value v , given the information obtained via check nodes other than check node m . The quantity r_{ml}^x is meant to be the probability of check m being satisfied when bit l of \bar{v} is considered fixed at v and the other bits are independent with probabilities $\{q_{ml'}, l' \in L(m) \setminus l\}$. The algorithm would produce the exact posteriori probabilities of all the bits after a fixed number of iterations, if the Tanner graph is defined by the matrix \mathbf{H} contained no cycles. In the following, binary transmission over an AWGN channel is assumed. As before, the modulated symbols $m(v_i)$ are transmitted over an AWGN channel and received as $r_i = m(v_i) + n_i$, where n_i is a Gaussian distributed random variable with zero mean and variance $N_0/2E_s$, $1 \leq i \leq N$.

Initialization

Let $p_l^0 = P(v_l = 0)$, (the prior probability that v_l is 0) and let $p_l^1 = P(v_l = 1) = 1 - p_l^0$.

For $l \in \{1, 2, \dots, N\}$, initialize the a-prior probabilities of the code node,

$$p_l^1 = \frac{1}{1 + \exp\left(r_l \frac{4}{N_0}\right)}, \quad (4.13)$$

and $p_l^0 = 1 - p_l^1$. For every (l, m) such that $h_{m,l} = 1$,

$$q_{m,l}^0 = p_l^0, \quad q_{m,l}^1 = p_l^1, \quad (4.14)$$

Horizontal step

In the horizontal step of the algorithm, it runs through the checks m and compute for two probabilities each $l \in L(m)$: first, r_{ml}^0 , the probability of the observed value z_m arising when $v_l = 0$, given that the other bits $\{v_{l'} : l' \neq l\}$ have separable distribution given by the probabilities $\{q_{ml'}^0, q_{ml'}^1\}$, defined by:

$$r_{ml}^0 = \sum_{\{v_{l'} : l' \in L(m) \setminus l\}} P(z_m | v_l = 0, \{v_{l'} : l' \in L(m) \setminus l\}) \times \prod_{l' \in L(m) \setminus l} q_{ml'}^{v_{l'}} \quad (4.15)$$

and second, r_{ml}^1 , the probability of the observed value z_m arising when $v_l = 1$, defined by

$$r_{ml}^1 = \sum_{\{v_{l'} : l' \in L(m) \setminus l\}} P(z_m | v_l = 1, \{v_{l'} : l' \in L(m) \setminus l\}) \quad (4.16)$$

$$\times \prod_{l \in L(m) \setminus l} q_{ml}^{v_l}.$$

The conditional probabilities in these summations are either zero or one, depending on whether the observed z_m matches the hypothesized values for v_l and the $\{v_l\}$.

These probabilities can be computed in various ways based on (4.15) and (4.16). A particular convenient implementation of this computation uses forward and backward passes in which products of the differences $\delta q_{ml} = q_{ml}^0 - q_{ml}^1$ are computed. We obtain $\delta r_{ml} = r_{ml}^0 - r_{ml}^1$ from the identity

$$\delta r_{m,l} = (-1)^{z_m} \prod_{l' \in L(m) \setminus l} \delta q_{m,l'}. \quad (4.17)$$

Finally, $r_{ml}^0 + r_{ml}^1 = 1$, and hence $r_{ml}^0 = (1 + \delta r_{ml})/2$ and $r_{ml}^1 = (1 - \delta r_{ml})/2$.

Vertical Step

The vertical step takes the computed values of r_{ml}^0 and r_{ml}^1 and updates the values of the probabilities q_{ml}^0 and q_{ml}^1 . For each l, m , compute

$$q_{m,l}^0 = p_l^0 \prod_{m' \in M(l) \setminus m} r_{m',l}^0, \quad q_{m,l}^1 = p_l^1 \prod_{m' \in M(l) \setminus m} r_{m',l}^1. \quad (4.18)$$

and normalize, with $\alpha = 1/(q_{ml}^0 + q_{ml}^1)$,

$$q_{m,l}^0 = \alpha q_{m,l}^0, \quad q_{m,l}^1 = \alpha q_{m,l}^1. \quad (4.19)$$

For each l , compute the a-posteriori probabilities

$$q_l^0 = p_l^0 \prod_{m \in M(l)} r_{m,l}^0, \quad q_l^1 = p_l^1 \prod_{m \in M(l)} r_{m,l}^1, \quad (4.20)$$

and normalize, with $\alpha = 1/(q_l^0 + q_l^1)$,

$$q_l^0 = \alpha q_l^0, \quad q_l^1 = \alpha q_l^1. \quad (4.21)$$

These quantities are used to create a tentative decoding \hat{v} , the consistency of which is used to decide whether the decoding algorithm can halt.

Decoding

If the Tanner graph of the code is really without cycles, the values of the pseudo posteriori probabilities q_l^0 and q_l^1 at each iteration would correspond exactly to the posteriori probabilities of bit l given the states of all the checks in a truncated belief network centered on bit l and extending out to a radius equal to twice the number of iterations. The decoding procedure set \hat{v}_i to 1 if $q_l^1 > 0.5$, for $i = 1, 2, \dots, N$.

4.4.2 Example of LDPC decoding

Figure 4.3 shows a simulation result for Gallager's (3, 4, 20) code. For convenience all information data have been made zero. It can be noticed in the figure that the BER improves as the iteration number increases. During later iterations, there is still improvement; however the amount of improvement begins to reduce.

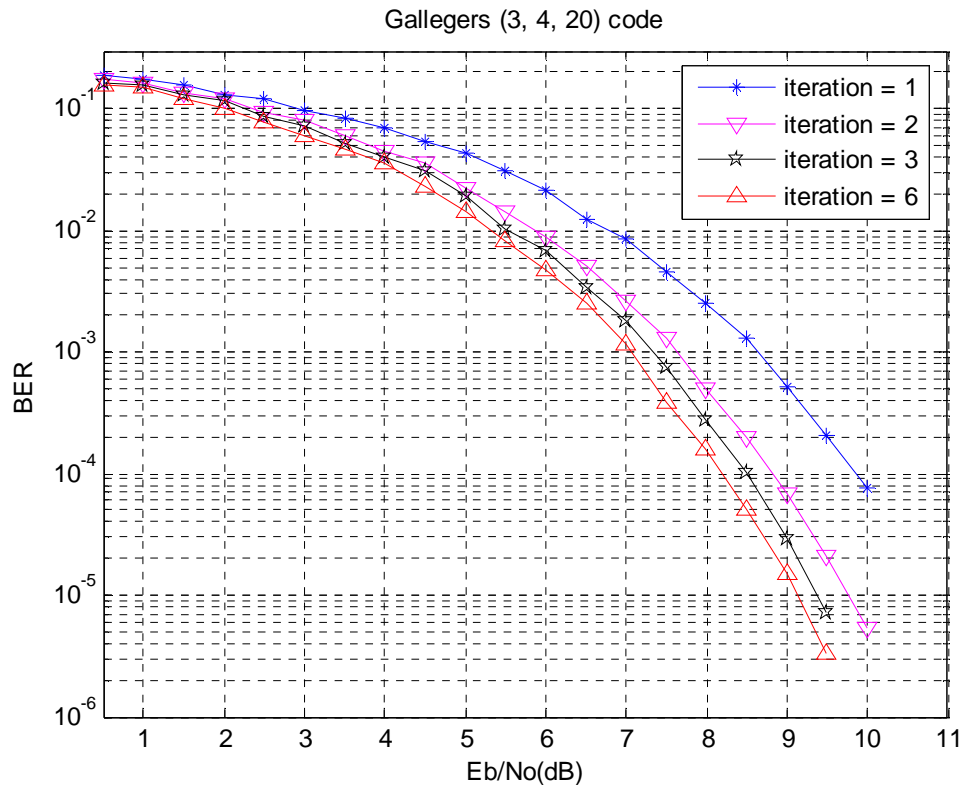


Figure 4.3: Performance of Gallager's (3, 4, 20) code with iterative probabilistic decoding.

4.5 Chapter summary

This chapter has presented an overview of LDPC code construction and decoding. An idea of code construction has been presented, and the concept of Tanner graph has been introduced. In contrast to the turbo code case, which has linear encode complexity, the complexity of the encoding efficiency is quadratic for LDPC codes. Here, an encoding procedure with a reasonable complexity has been presented. One of the potential drawbacks to LDPC codes is the large number of decoding iterations that may be required, resulting in increased latency and decoding hardware complexity. At the end of the chapter, an overview of iterative probabilistic decoding algorithm has been presented.

Chapter 5

Performance comparison between turbo codes and LDPC codes

This chapter focuses on the performance analysis of turbo codes and LDPC codes. It gives an overview of some major performance factors that have impacts on turbo codes and LDPC codes to some extent. Later in this chapter, simulation results are presented along with discussions and comparisons between turbo codes and LDPC codes.

5.1 Performance analysis of turbo codes

The bit error probability of a (n, k) linear block code over an additive white Gaussian noise (AWGN) channel can be bounded as (Proakis 1995)

$$P_b \leq \sum_{i=1}^{2^k-1} \frac{w_i}{k} Q\left(\sqrt{\frac{2d_i RE_b}{N_0}}\right), \quad (5.1)$$

where

$$Q(z) = \frac{1}{\sqrt{2\pi}} \int_z^{\infty} e^{-x^2/2} dx, \quad x \geq 0, \quad (5.2)$$

is the Gaussian Q - function, w_i is the weight of the message sequence of the i th message, d_i is the Hamming weight of the codeword and E_b / N_0 is the signal- to- noise ratio (SNR) per bit.

The computation of (5.1) requires knowledge of the weights of all $2^k - 1$ nonzero codewords and their corresponding messages. The computational complexity of (5.1) prevents its use for all but the smallest values of k . Grouping together codewords of the same Hamming weight, the bound on the probability of bit error can be written as

$$P_b \leq \sum_{d=d_{\min}}^n \frac{W_d}{k} Q\left(\sqrt{d \frac{2RE_b}{N_0}}\right) = \sum_{d=d_{\min}}^n \frac{\tilde{w}_d N_d}{k} Q\left(\sqrt{d \frac{2RE_b}{N_0}}\right), \quad (5.3)$$

where N_d is the number of codewords of weight d and \tilde{w}_d is the average weight of the N_d messages that produce weight d codewords. At high signal to noise ratios, the bit error probability is approximated by the first term of (5.3)

$$P_b \approx \frac{\tilde{w}_{d_{\min}} N_{d_{\min}}}{k} Q\left(\sqrt{d_{\min} \frac{2RE_b}{N_0}}\right), \quad (5.4)$$

which is called the minimum distance asymptote.

In (5.4), the message length k appears in the denominator of the leading coefficient. For sufficiently large interleavers, $k \approx L$. Here, L represents the size of interleaver. Hence, the asymptotic BER of a turbo code is (approximately) inversely proportional to the size of its interleaver. The BER of a turbo code could therefore be lowered by increasing the size of the interleaver or conversely raised by decreasing its size. Undeniably, the size of the interleaver has a significant impact on the performance of turbo codes. Although the most desirable selection is the largest possible interleaver, larger interleavers introduce longer decoding latencies, require more memory in the decoder and increase computational complexities. Hence, the interleaver size must be chosen to match the bit error, latency and decoder memory requirements of the system.

Turbo codes can approach the Shannon limit as their constraint length K_c increases but the complexity of their best known decoding algorithms grows exponentially with the constraint length. In particular, constraint length does not significantly influence the performance of turbo codes at low signal-to-noise ratios. For this reason, turbo codes typically use simple constituent codes with constraint length $3 \leq K_c \leq 5$. However, for a sufficiently large block of frame, constraint length can be a secondary determinant of the location of the “BER floor” region of the performance (Wu 1999).

Another factor that influences the performance of turbo codes is code rate. The performance of turbo codes degrades as code rate increases. If puncturing is used to increase the code rate then the puncturing manner is also a performance factor for turbo codes. The joint optimization of interleaver and puncturing matrix is perhaps the most important aspect of turbo code design (Acikel and Ryan).

The communication delay or “latency” in a turbo coded system is directly proportion to the frame size. As previously mentioned, the more increment will be made in frame or interleaver size the more communication delay will occur.

5.1.1 Simulation results

In this section, the effect of interleaver size, code rate, constraint length and latency are investigated by means of simulation.

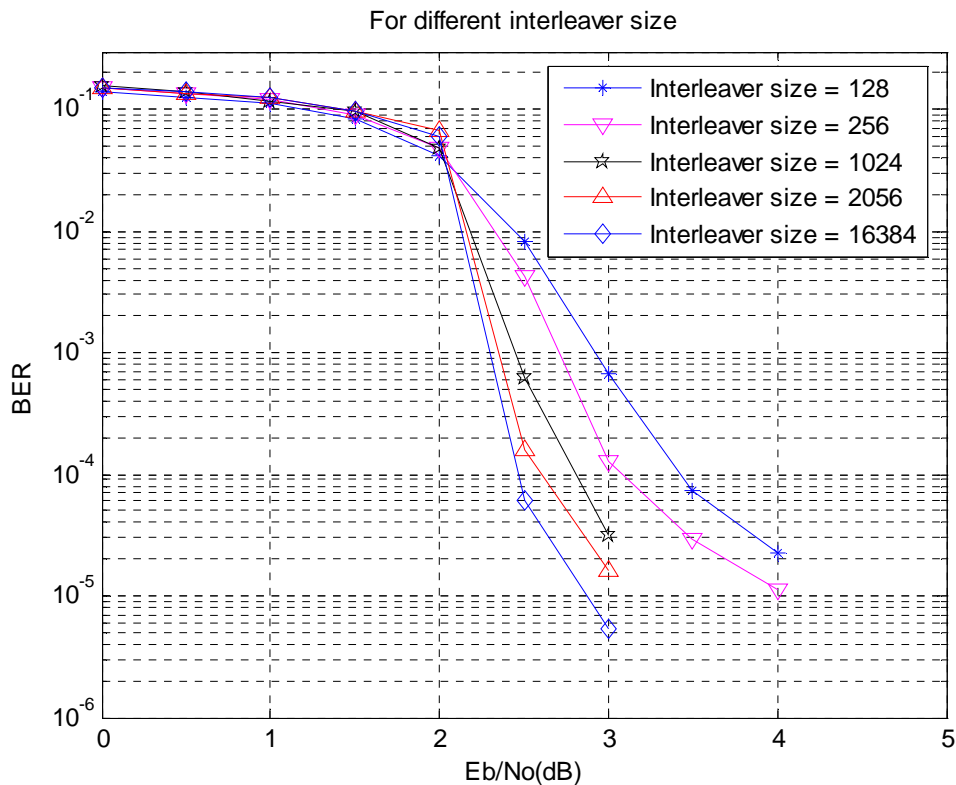


Figure 5.1: Performance of parallel concatenated (turbo) code for different block interleaver size with memory-4, rate-1/2 RSC codes and generators (37, 21).

Figure 5.1 shows the simulated performance for five different interleaver sizes: $L = 128$, $L = 256$, $L = 1024$, $L = 2056$, $L = 16384$. Here, the turbo code is composed of a pair of $K_c = 5$ RSC encoders with feedback generator $(37)_o$ and feed forward generator $(21)_o$. The interleavers are designed at random. The code rate is 1/2 and iteration number is 6. The encoded bits are BPSK modulated and AWGN channel is considered.

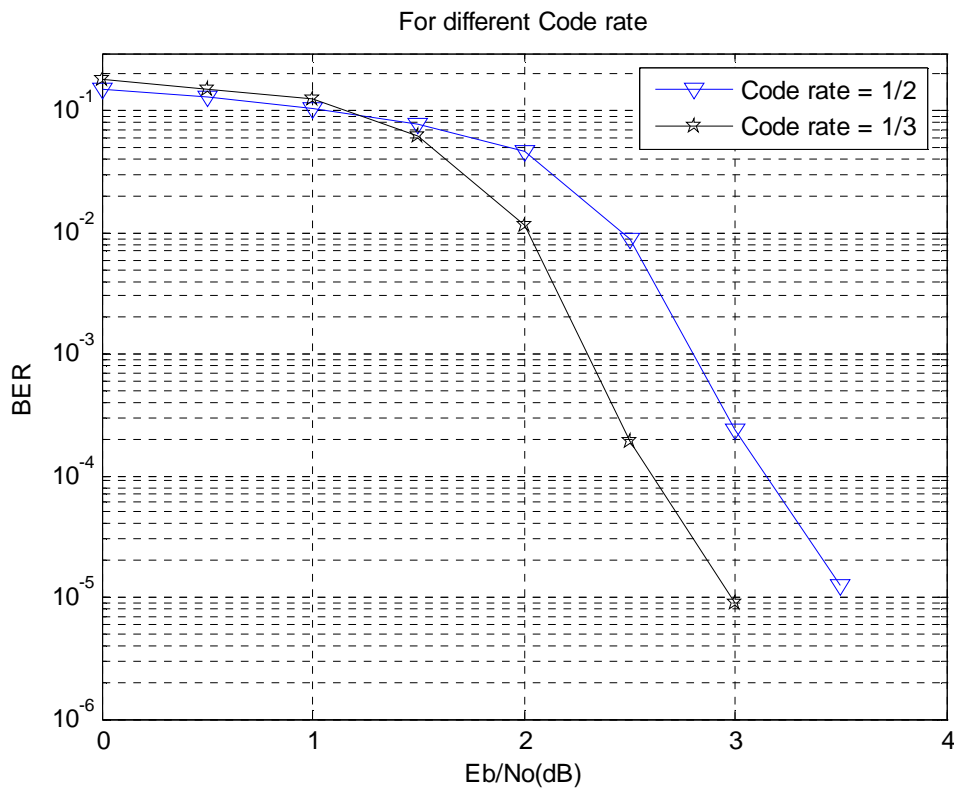


Figure 5.2: Performance of rate-1/2 and rate- 1/3 parallel concatenated (turbo) code with memory-2, generators (7, 5) block interleaver size = 1024 and iteration = 6.

Figure 5.2 shows the simulated performance for two different code rates: $r = 1/2$, $r = 1/3$. Here, in both cases the turbo code is composed of a pair of $K_c = 3$ RSC encoders with feedback generator $(7)_o$ and feed forward generator $(5)_o$. The interleavers are designed at random and the size of the interleaver is $L = 1024$. In both cases, the decoding algorithm was run for 6 times. The encoded bits are BPSK modulated and AWGN channel is considered.

In Figure 5.1, an acceptable range of BER's can be obtained for the rate 1/2 turbo code in an AWGN channel by holding E_b/N_0 to a constant value of 3.0 dB. The BER and latency is listed in Table 1 for the rate 1/2 turbo code in AWGN channel with E_b/N_0 set to 3.0 dB.

Interleaver Size (bits)	Latency	BER
128	31msec	6.72E-04
256	63msec	1.30E-04
1024	235msec	3.190E-05
2056	501msec	1.60E-05
16384	4.06sec	5.39E-06

Table 1 Quality of Service for rate 1/2 turbo code in AWGN at $E_b/N_0 = 3.0$ dB.

Figure 5.3 shows the simulated performance for two different constraint length: $K_c = 3$, $K_c = 5$. For $K_c = 3$, the turbo code is composed of a pair of $K_c = 3$ RSC encoders with feedback generator $(7)_o$ and feed forward generator $(5)_o$. For $K_c = 5$, the turbo code is composed of a pair of $K_c = 5$ RSC encoders with feedback generator $(37)_o$ and feed forward generator $(21)_o$. In all cases, the interleaver size $L = 1024$, code rate is 1/2 and iteration number is 6. The encoded bits are BPSK modulated and AWGN channel is considered.

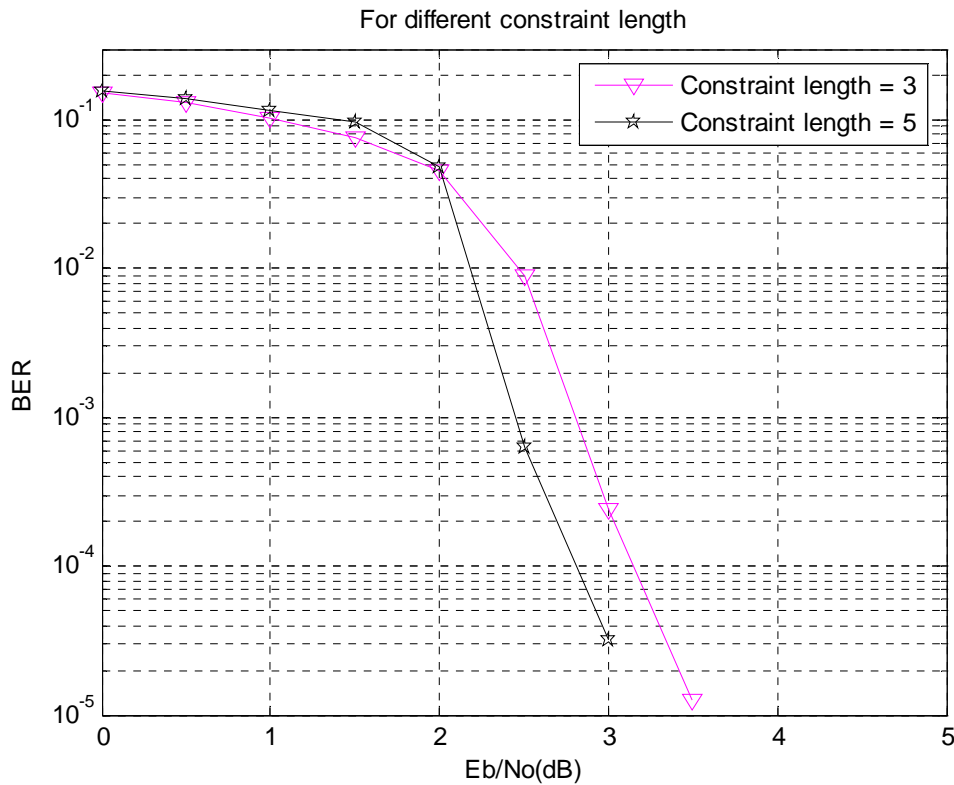


Figure 5.3: Performance of rate-1/2 parallel concatenated (turbo) code with different constraint length.

5.2 Performance analysis of LDPC codes

LDPC codes have excellent distance properties. Gallager showed that for random LDPC codes, the minimum distance d_{\min} between codewords increases with N when column and row weights are held fixed (Moon, T K 2005), and it makes the low density parity check matrix very sparse. The best results can be obtained from LDPC codes by making the weight per column w_c as small as possible. It has been proved that as the block length $N \rightarrow \infty$ and the low-density parity check matrix becomes more sparse, LDPC codes can reach channel capacity. Unsurprisingly, code with large block length improves the performance of LDPC codes.

Because the codes are constructed from very sparse matrices, they have simple and practical decoding algorithms, which work, empirically, at good communication rates. The decoding complexity of LDPC codes has the complexity linearly proportional to the block length.

Another factor that influences the performance of LDPC codes is the number of cycles exists in the matrices. The original matrices shouldn't have cycles with not more than length of 4. This constraint has been found to be beneficial. If there exists cycles with length of 6, 8, ..., the columns should be deleted to further improve the performance. However, it is found that these modifications do not make very impressive difference.

There are, obviously, some potential disadvantages to LDPC codes. As previously mentioned, the best code performance is obtained for very long codes. This long block length of code and the need for iterative decoding introduce latency which is unacceptable in many applications. The latency is the time taken by the decoder to decode one frame of received information block. Another disadvantage is that, since the \mathbf{H} matrix is not necessarily sparse, the encoding operation may have complexity $O(N^2)$. In a brute-force approach, the time to create a Gallager code scales as N^3 , where N is the block length (Shokrollahi 2003, pp.2-3).

5.2.1 Simulation results

Figure 5.4 shows the simulated performance of Gallager's (3, 4, 20) code for different iteration numbers. Here, $N = 20$, $K = 5$, $M = 15$, $w_c = 3$ and $w_r = 4$. For convenience all-zero codeword $C = 0$ is sent. The information bits are BPSK modulated and AWGN channel is considered. Here, the code rate, $R = 0.25$.

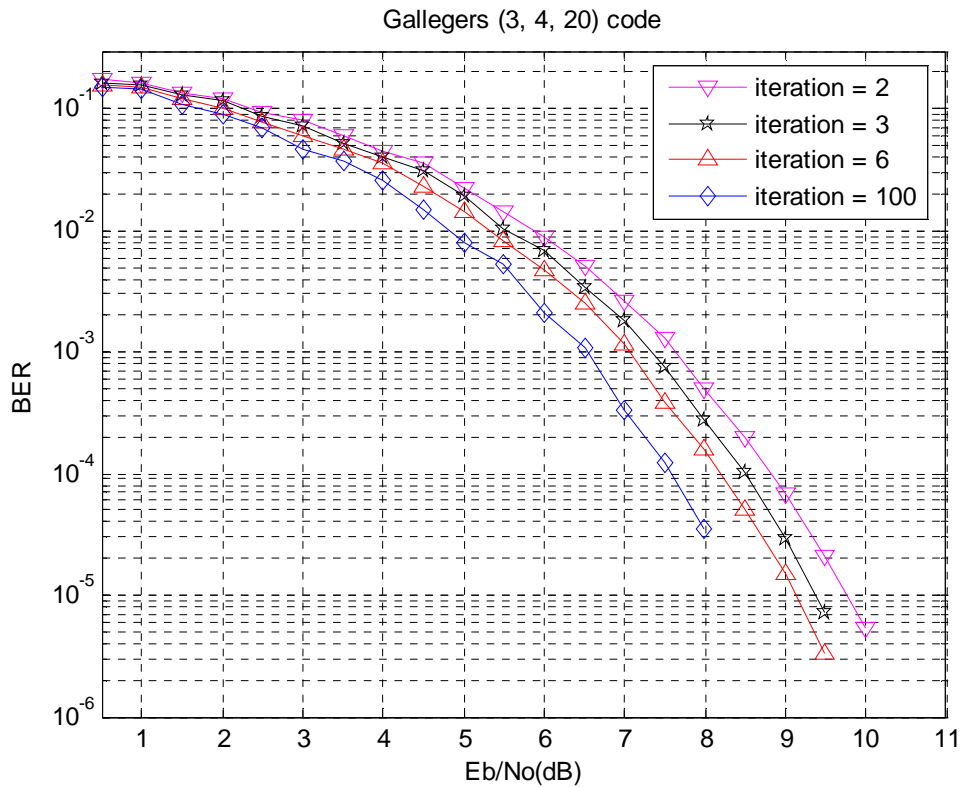


Figure 5.4: Performance of rate-1/4 Gallager's (3, 4, 20) code with iterative probabilistic decoding for different iteration number.

In Figure 5.4, an acceptable range of BER's can be obtained for the rate 1/4 LDPC code in AWGN channel by holding E_b/N_0 to a constant value of 8.0 dB. The latency is listed in Table 2 for the rate 1/4 LDPC code in AWGN channel with E_b/N_0 set to 8.0 dB.

Iteration number	Latency
500	31msec
1000	47msec
1500	63msec
2000	78msec
10000	375msec

Table 2 Quality of Service for rate 1/2 LDPC code in AWGN at $E_b/N_0 = 8.0$ dB.

Figure 5.5 shows the simulated performance of Gallager's (3, 6, 96) code for different iteration numbers. Here, $N = 96$, $K = 48$, $M = 48$, $w_c = 3$ and $w_r = 6$. For convenience all-zero codeword $C = 0$ is sent. The information bits are BPSK modulated and AWGN channel is considered. Here, the code rate, $R = 0.5$.

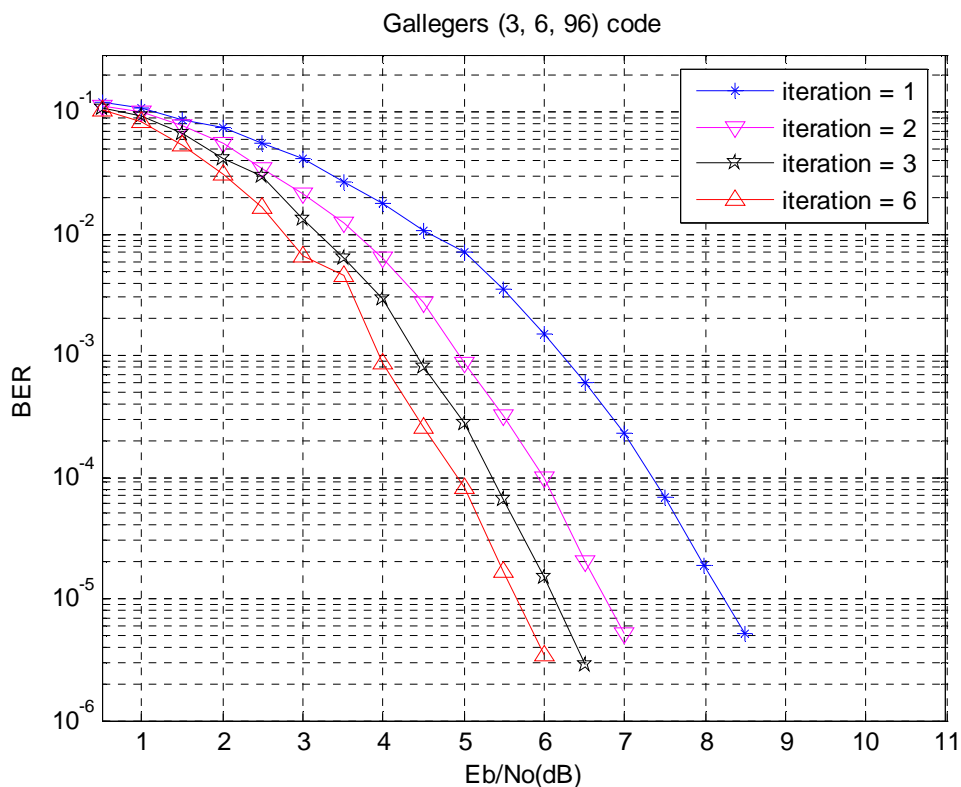


Figure 5.5: Performance of rate-1/2 Gallager's (3, 6, 96) code with iterative probabilistic decoding for different iteration number.

Figure 5.6 shows the simulated performance of Gallager's (3, 6, 816) code for different iteration numbers. Here, $N = 816$, $K = 408$, $M = 408$, $w_c = 3$ and $w_r = 6$. For convenience all-zero codeword $C = 0$ is sent. The information bits are BPSK modulated and AWGN channel is considered. Here, the code rate, $R = 0.5$.

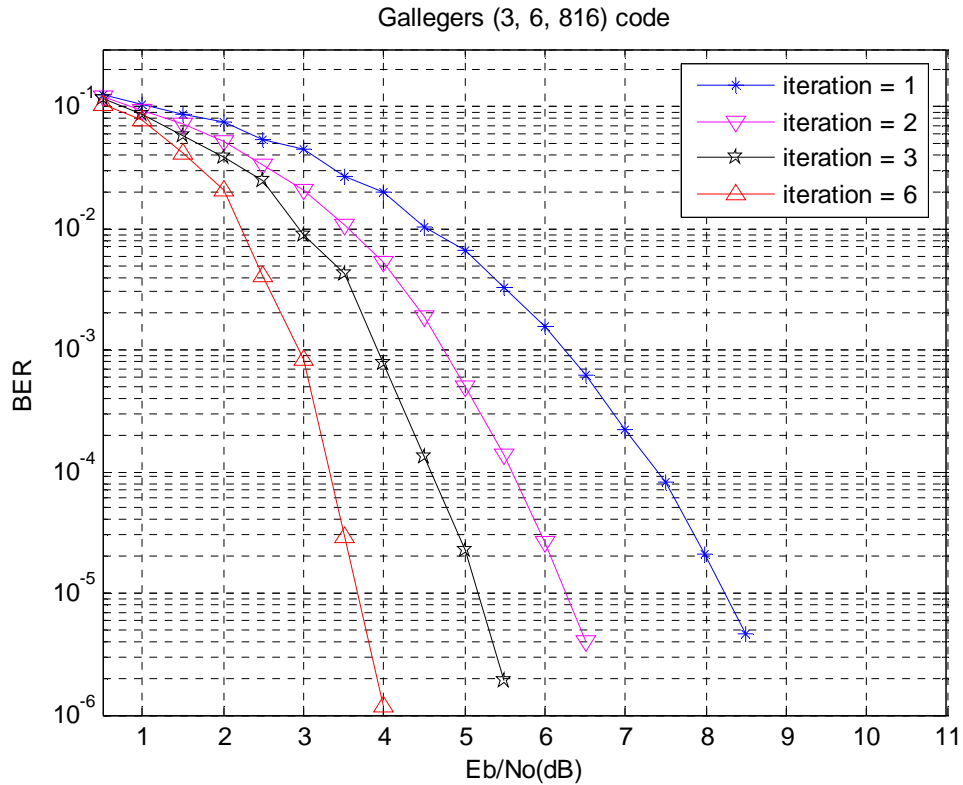


Figure 5.6: Performance of rate-1/2 Gallager's (3, 6, 816) code with iterative probabilistic decoding for different iteration number.

From Figure 5.4, 5.5 and 5.6, it is clearly seen that for LDPC codes as the block length N increases the BER improves.

5.3 Performance comparison

From Figure 5.7, we can see that at error probabilities of about 10^{-5} , LDPC codes are not able to get quite so close to the Shannon limit as turbo codes. However, turbo codes as originally presented are known to have an error "floor" at about 10^{-6} , that implies that the error probability of these turbo codes no longer decreases rapidly with increasing E_b/N_0 below this floor. In LDPC codes, there is no evidence of such error "floor". Hence it is possible that at very low-bit error probabilities, LDPC codes outperform turbo codes.

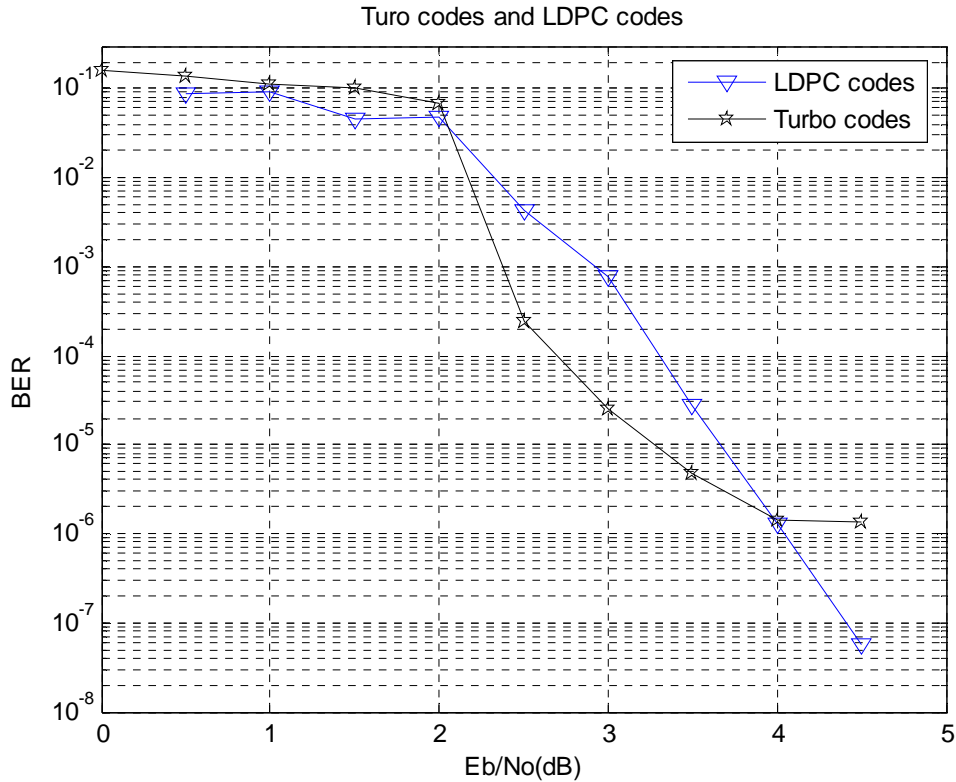


Figure 5.7: Comparison between turbo codes and LDPC codes.

LDPC codes may also have an advantage over turbo codes in terms of their decoding complexity. In some cases, LDPC codes have greater performance than turbo codes with iterative decoding algorithms which are easy to implement and are also parallelizable in hardware. Per-iteration complexity of turbo codes is much higher than the per-iteration complexity of LDPC codes. In addition, for turbo codes the decoding becomes more complex as the interleaver size and the constraint length increases.

LDPC codes of any rate and block length can be created simply by specifying the shape of the parity check matrix, while the rate of turbo codes is dependent on of the structure of puncturing matrix. Hence, flexibility in the code rate for LDPC codes is obtained only through considerable design effort.

On the other hand, LDPC codes have a significantly higher encoding complexity than turbo codes. For turbo codes, the encoding has the complexity linearly proportional to the block length. However, the encoding of LDPC codes is generically quadratic in the code dimension, although this can be reduced somewhat (Moon, T K 2005). Besides, LDPC decoding may require many more iterations than turbo decoding and this may introduce latency and increase decoding hardware complexity.

5.4 Chapter summary

This chapter has presented the performance analysis of turbo codes and LDPC codes individually at first. Then the performance comparison between these two codes has been presented. In order to get the best performance from both turbo codes and LDPC codes some trade-offs have to be made. The impacts on the error performance of both of these codes for some major performance factors have been presented here along with some simulation results.

Chapter 6

Conclusions

In modern days, wireless is a method of communication that uses low-powered radio waves to transmit data between devices. But these wireless links are very vulnerable to channel deficiency such as channel noise, mutli path effect and fading. Iterative decoding is particularly suitable when the transmission channel is noisy. The concept of iterative decoding may be defined as a method of utilizing a soft-output decoding algorithm that is iterated several times to keep the probability of error at an acceptably low level. Turbo codes and low-density parity-check codes are two error control codes based on iterative decoding. In this dissertation, the concepts of both turbo codes and LDPC codes have been presented. This dissertation gives an overview of code construction, iterative decoding procedures and performance analysis for both turbo codes and LDPC codes. Turbo codes are a novel error correction technique that was first introduced by a group of researchers from France in 1993 (Berrou, Glavieux & Thitimajshima 1993, pp. 1064-1070). The essential features of turbo codes are parallel concatenation, recursive systematic convolutional encoding, random interleavers and iterative decoding. Low-density parity-check (LDPC) codes were originally proposed by

Robert Gallager in 1962 (Gallager 1962, vol. 8, no. 1, pp. 21-28). The essential features of LDPC codes are code construction, tanner graphs, encoding codes and iterative belief propagation (IBP) decoding algorithm.

It has been showed that both turbo codes and LDPC codes have the ability to perform at their best but under some conditions. Tradeoffs between some major performance factors have to be made to get the best output from turbo codes and LDPC codes. Turbo codes have been shown to exhibit remarkable performance when the interleaver size is large and a sufficient number of decoding iterations is performed. However, decoding complexity and latency are an invariant to interleaver size. By properly choosing the interleaver size, a tradeoff between performance and efficiency can be made. LDPC codes can approach to Shannon capacity limit, if the block length $N \rightarrow \infty$ and the low-density parity check matrix becomes more sparse. However, this long block length of code and the need for iterative decoding introduce latency. Unsurprisingly, the increasing number of decoding iterations improves the performance of LDPC codes which also introduces communication delay.

In some cases, LDPC codes can outperform turbo codes. For LDPC codes, the decoding complexity of per information bits is linearly proportional to block size, whereas for turbo codes the decoding becomes more complex as the frame size increases. Conversely, the encoding complexity for LDPC codes is quadratic and for turbo codes it is linear.

6.1 Future work

While doing this project I have been able to comprehend and implement two error correcting codes, i.e., turbo codes and LDPC codes. I have also been able to demonstrate the performance comparison between these two codes in terms of bit error rate, latency, code rate and computational resources. However it is shown that for both turbo codes and LDPC codes, the researchers mostly depend on the simulation results to analyze the

performance. In this dissertation, the performance analyses of these two codes under some major performance factors have been presented along with some simulation results. Suggestions for extensions of this work might include the analytical techniques of performance analysis for both turbo codes and LDPC codes; i.e. extrinsic information transfer (EXIT) charts for turbo codes, EXIT charts for LDPC codes, Density evolution.

Another inclusion can be the investigation of irregular LDPC codes. Although a brief overview of irregular LDPC codes has been presented here, an intensive analysis can be made on this type of codes based on this project dissertation. As it is showed that irregular LPDC codes may outperform turbo codes of approximately the same length and rate, when the block length is large.

Appendix A

Project specification

University of Southern Queensland

FACULTY OF ENGINEERING AND SURVEYING

ENG 4111/4112 Research Project
PROJECT SPECIFICATION

FOR: **Tawfiqul Hasan Khan**

TOPIC: Iterative Decoding for Error Resilient Wireless Data Transmission

SUPERVISORS: Dr. Wei Xiang

ENROLMENT: ENG 4111 – S1, D, 2006
ENG 4112 – S2, D, 2006

PROJECT AIM: This project seeks to investigate the performance of two error control codes, turbo codes and low-density parity-check codes based on iterative decoding and demonstrate the performance comparison between these two codes.

PROGRAMME: **Issue A, 15th March 2006**

1. Study the background of Error Control Codes.
2. Implement and analyze Viterbi algorithm, Soft output Viterbi algorithm and Maximum-a-posteriori (MAP) algorithm and Soft-Input Soft-Output (SISO) algorithm.
3. Examine and implement Turbo codes and their iterative decoders.
4. Examine and implement low-density parity check (LDPC) codes and their iterative decoders.
5. Demonstrate the performance comparison between turbo codes and LPDC codes in terms of bit error rate, latency and computational resources.

Time permitted:

6. Ascertain the advantages and disadvantages of turbo codes and LPDC codes, in terms of error resilient performance, latency and computational resources.

AGREED: _____ (student) _____ (Supervisors)

(dated) ___/___/___

Appendix B

List of Acronyms

ACS	Add-Compare-Select
AMPS	Advanced Mobile Phone System
APP	A-Posteriori Probability
ARDIS	Advanced Radio Data Information Service
AWGN	Additive White Gaussian Noise (channel)
BCH	Bose, Ray-Chaudhuri, Hocquenghem
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
CDMA	Code Division Multiple Access
ERNIES	European Radio Messaging System
EXIT	Extrinsic Information Transfer
FER	Finite Impulse Response
FPLNITS	Future Public Land Mobile Telecommunications System
GSM	Global System Mobile
IBP	Iterative Belief Propagation
IEEE	Institute of Electrical and Electronics Engineering
IIR	Infinite Impulse Response
LDPC	Low-density parity check
LEO	Low Earth Orbit
LLR	Log-Likelihood Ratio
MAP	Maximum-a-posteriori
MEO	Medium Earth Orbit
MLD	Maximum-Likelihood Decoding
NMT	Nordic Mobile Telephone
PAN	Personal Area Network
PCC	Punctured Convolutional codes
POCSAG	Post Office Code Standardization Advisory Group
PSTN	Public Switched Telephone Network
RMD	RAM Mobile Data
RSC	Recursive Systematic Convolutional
SISO	Soft-Input, Soft-Output

- SOVA** Soft-Output Viterbi Algorithm
- SMR** Specialized Mobile Radio
- SNR** Signal to noise power Ratio
- TACS** Total Access Communication System
- TDMA** Time Division Multiple Access
- USDC** United States Digital Cellular
- VA** Viterbi Algorithm
- VD** Viterbi Decoder
- WLAN** Wireless Local Area Network

Appendix C

List of codes

The name of all 'C' files and 'text' files are listed below along with their brief descriptions:

File name	Description
1. <code>main_turbo_punc.c</code>	It is a turbo simulation program. The punctured matrix has been used to make the code rate 1/2. It uses two RSC codes encoder to generate codes. It utilizes two Soft-Input Soft-Output MAP decoders to decode the code. A user defined length of random interleaver has been used to make sure that MAP decoder can estimate information symbols independently at each iteration. It writes the output into "turbo_punc.txt" file. Hence, the executable file "main_turbo_punc.exe" of this program has to be stored into the computer first before running.
2. <code>main_turbo.c</code>	It is a turbo simulation program. Here, the code rate is 1/3. It utilizes two Soft-Input Soft-Output MAP decoders to decode the code. A user defined length of random interleaver has been used to make sure that MAP decoder can estimate information symbols independently at each iteration. It writes the output into "turbo.txt" file. Hence, the executable file "main_turbo.exe" of this program has to be stored into the computer first before running.
3. <code>main_ldpc.c</code>	It is an LDPC simulation program. It uses the low-density parity-check matrix and iterative belief propagation (IBP) decoding algorithm for decoding LDPC codes. It writes the output into

“LDPC_output.txt” file. Hence, the executable file “main_ldpc.exe” of this program has to be stored into the computer first before running.

4. gal(3,4,20,5)r1_4.txt It is a text file which contains the information of a specific low-density parity-check matrix. Here, $N = 20$, $K = 5$, $R = 0.25$, $w_c = 3$ and $w_r = 4$. Source: <
<http://www.inference.phy.cam.ac.uk/mackay/codes/data.html> >
5. gal(3,6,96,48)r1_2.txt It is a text file which contains the information of a specific low-density parity-check matrix. Here, $N = 96$, $K = 48$, $R = 0.5$, $w_c = 3$ and $w_r = 6$. Source: <
<http://www.inference.phy.cam.ac.uk/mackay/codes/data.html> >
6. gal(3,6,816,408)r1_2.txt It is a text file which contains the information of a specific low-density parity-check matrix. Here, $N = 816$, $K = 408$, $R = 0.5$, $w_c = 3$ and $w_r = 6$. Source: <
<http://www.inference.phy.cam.ac.uk/mackay/codes/data.html> >

```

/////////////////////////////////////////////////////////////////
//
// Name: main_turbo_punc.c
// Author: Tawfiqul Hasan Khan
// Description: It is a turbo simulation program. The punctured matrix has been
//              used to make the code rate 1/2.It uses two RSC codes encoder to
//              generate codes. It utilizes two Soft-Input Soft-Output MAP
//              decoder to decode the code. An user defined length of random
//              interleaver has been used to make sure that MAP decoder can
//              estimate information symbols independently at each iteration.
// Date of creation: 13/7/2006
/////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>

int ITERATIONS;    // Global variable for number of iterations
int NUM_STATE;    // Global variable for state number
double variance;  // Global variable for variance

/* Declaration of puntured matrix */

int punc_mat[3][2]={ 1, 1,
                    1, 0,
                    0, 1 };

/* Declaration of structure for trllis*/
typedef struct {
    int from_state[2];    // Initial state
    int fin_state[2];    // Final state
    int output[2];       // Output coded symbols (branch label)
}trel;

/* Declaration of structure for decoding*/
typedef struct{

    double alpha;        // Forward metric
    double beta;         // Back ward metric
    double gamma[2];    // Branch metric
    double gamma_I[2];  // Branch metric

}state_metric;

/* Functions prototypes */
int random_generator();
void* create_interleave_deinterleave(unsigned *interleave_mat,
                                     unsigned *deinterleave_mat, int size);
double gaussian_noise(double mean, double variance);
void encoder(trel *trel_state,int *data,int *parity,unsigned size, int force,
            int memory);
void create_encode_table(trel *trel_state,int state_num,int memory);
void interleave(int *data,unsigned size,unsigned *interleave_mat);
void deinterleave(int *data,unsigned size,unsigned *deinterleave_mat);
void decode(trel * trel_state,double *channel_data, double *channel_parity1,
            double *channel_parity2, int size, int *data,
            unsigned *interleave_mat,unsigned *deinterleave_mat);
void deinterleave_double(double *channel_data, int size,
                        unsigned *deinterleave_mat);
void interleave_double(double *channel_data, int size,unsigned *interleave_mat);
void decoder_map(trel * trel_state,double *channel_data, double *channel_parity,

```



```

        int size, state_metric **stage_metric, double *Log[],
        int decode_num, int loop);

int generator[10];

/* Main function*/
int main(int argc, char *argv[]){

    int *data, *parity1,*parity2;          // Variables for data, parity1 &
parity2
    double *channel_data, *channel_parity1,*channel_parity2; // Variables for
channel data, channel parity1, channel parity2
    trel *trel_state;
    unsigned *interleave_mat;             // Iterleaver matrix
    unsigned *deinterleave_mat;          // Deinterleaver matrix
    int size;                             // Interleaver or frame size
    double rate;                          // The code rate

    /* Declaration of some necessary variables*/
    unsigned long seed;
    int i,j;
    int memory;
    int temp1,temp2;
    int total_size;
    int total_error;
    int temp_data;
    double dB,init_dB,fin_dB,inc_dB;
    int max_sim;
    FILE * fp;

    //////////////////////////////////// USER INPUT ////////////////////////////////////
    printf("Enter the data size:");
    scanf("%d",&size);
    printf("Enter seed:");
    scanf("%ld",&seed);
    printf("Enter number of memory:");
    scanf("%d",&memory);
    printf("Enter number of iteration:");
    scanf("%d",&ITERATIONS);
    printf("Enter feedback generator in octal:");
    scanf("%o",&generator[0]);
    printf("Enter feedforward generator in octal:");
    scanf("%o",&generator[1]);

    //////////////////////////////////// INITIALIZATION ////////////////////////////////////

    data= (int*)malloc(size*sizeof(int));
    if(data==NULL){

        printf("ERROR: Could not allocate memory space for data\n");
        exit(0);
    }

    parity1= (int*)malloc(size*sizeof(int));
    if(parity1==NULL){

        printf("ERROR: Could not allocate memory space for parity1\n");
        exit(0);
    }

    parity2= (int*)malloc(size*sizeof(int));
    if(parity2==NULL){

```

```

        printf("ERROR: Could not allocate memory space for parity2\n");
        exit(0);
    }

    channel_data= (double*)malloc(size*sizeof(double));
    if(channel_data==NULL){

        printf("ERROR: Could not allocate memory space for channel_data\n");
        exit(0);
    }

    channel_parity1= (double*)malloc(size*sizeof(double));
    if(channel_parity1==NULL){

        printf("ERROR: Could not allocate memory space for
channel_parity1\n");
        exit(0);
    }

    channel_parity2= (double*)malloc(size*sizeof(double));
    if(channel_parity2==NULL){

        printf("ERROR: Could not allocate memory space for
channel_parity2\n");
        exit(0);
    }

    interleave_mat= (int*)malloc(size*sizeof(int));
    if(interleave_mat==NULL){

        printf("ERROR: Could not allocate memory space for interleave_mat\n");
        exit(0);
    }

    deinterleave_mat= (int*)malloc(size*sizeof(int));
    if(deinterleave_mat==NULL){

        printf("ERROR: Could not allocate memory space for
deinterleave_mat\n");
        exit(0);
    }
    NUM_STATE= pow(2,memory);
    trel_state= (trel *)malloc(NUM_STATE*sizeof(trel));
    if(trel_state==NULL){

        printf("Could not allocate space for trel\n");
        exit(0);
    }

    fp=fopen("turbo_punc.txt","w");
    if(fp==NULL){
        printf("ERROR: Could not open file");
        exit(0);
    }
    ////////////////////////////////// Setting value for simulation //////////////////////////////////

    init_dB=0.0;
    fin_dB=6.0;
    inc_dB=0.5;
    max_sim=LONG_MAX;
    rate=0.5;

```



```

        channel_parity1[i] = channel_parity1[i]+
gaussian_noise(0,variance);
        channel_parity2[i] = channel_parity2[i]+
gaussian_noise(0,variance);
    }
    /* Decoding with two SISO MAP decoder*/
    decode (trel_state,channel_data,channel_parity1,channel_parity2,
            size,data,interleave_mat,deinterleave_mat);
    /* Checking error*/
    for (i=0;i<size;i++){

        temp_data = (channel_data[i] == 1) ? 1 : 0;

        if (data[i] != temp_data)
            total_error++;

    }

    total_size = total_size+size;
}
/* Printing simulation results in the console */
printf ("%5.2lf %10.7e %6ld %2lld %d\n", dB,
        (total_error/(double)total_size), total_error, total_size,
        size);
fprintf(fp,"%5.2lf %10.7e %6ld %2lld %d\n", dB,
        (total_error/(double)total_size), total_error, total_size,
        size);
}
//////////////////////////////////////////////////////////////////DEALLOCATING MEMORY//////////////////////////////////////////////////////////////////

free(data);
free(parity1);
free(parity2);
free(channel_data);
free(channel_parity1);
free(channel_parity2);
free(interleave_mat);
free(deinterleave_mat);
free(trel_state);
fclose(fp);
}

//////////////////////////////////////////////////////////////////
// Function name: random_generator()
// Description: Generate random numbers
//////////////////////////////////////////////////////////////////

int random_generator()
{
    double temp;

    temp=(double)rand()/LONG_MAX;

    if(temp>0.5){
        return 1;
    } else {
        return 0;
    }
}
}

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function name: create_interleave_deinterleave
// Description: Create interleave and deinterleave matrix
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void* create_interleave_deinterleave(unsigned *interleave_mat,unsigned
*deinterleave_mat, int size)
{
    int *temp, temp_pos,i;

    temp= (int*)malloc(size*sizeof(int));
    if(temp==NULL){

        return NULL;
    }

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

        temp[i] = 0;
    }
    /* Writing interleaver */
    for (i=0; (i<size); i++){

        do {
            temp_pos = rand()%size;
        } while ( temp[temp_pos] );

        temp[temp_pos] = 1;
        interleave_mat[i] = temp_pos;
    }
    /* Writing deinterleaver */
    for (i=0; (i<size); i++){

        deinterleave_mat[interleave_mat[i]]=i;
    }
    free(temp);
    return;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function name: gaussian_noise
// Description: Create gaussian noise with 0 mean and given variance
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

double gaussian_noise(double mean, double variance)
{
    double s1, s2, rand_n, r, g_noise;
    do {

        rand_n = (double)(rand())/LONG_MAX;
        s1 = rand_n * 2 - 1.0;

        rand_n = (double)(rand())/LONG_MAX;
        s2 = rand_n * 2 - 1.0;

        r = s1 * s1 + s2 * s2;
    } while( r >= 1 );

    g_noise =mean + sqrt(variance)*(s1 * sqrt( (-2.0*log(r))/r ));

    return g_noise;
}

```

```

/////////////////////////////////////////////////////////////////
// Function name: decode
// Description: Utilizes two SISO MAP decoder to decode the codes
/////////////////////////////////////////////////////////////////

void decode (trel * trel_state,double *channel_data, double *channel_parity1,
double *channel_parity2, int size, int *data,unsigned *interleave_mat,unsigned
*deinterleave_mat)
{
    double *Log[2];
    state_metric **stage_metric1; /* For decoder 1*/
    state_metric **stage_metric2; /* For decoder 2*/

    int i,j,input,state,loop;
    double Lc;

    // Create matrix for LLR

    for (input=0;input<2;input++){

        Log[input] = (double*)malloc(size*sizeof(double));
    }

    stage_metric1 = (state_metric**)malloc(size*sizeof(state_metric*));
    stage_metric2 = (state_metric**)malloc(size*sizeof(state_metric*));

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

        stage_metric1[i]
=(state_metric*)malloc(NUM_STATE*sizeof(state_metric));
        stage_metric2[i]
=(state_metric*)malloc(NUM_STATE*sizeof(state_metric));
    }

    /* Initialization of iteration arrays */
    for (state=0;state<NUM_STATE;state++){

        stage_metric1[0][state].alpha = stage_metric1[size-1][state].beta =
stage_metric2[0][state].alpha = (state==0) ? 1.0 : 0.0;
    }

    /* Initialization of extrinsic information array for decoder 2, that will
    be used in decoder 1 */

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

        Log[1][i] = 0.0;
    }

    Lc = 2.0/variance;
    /* Run the decoding for number of ITERATION times*/
    for (loop=0;loop<ITERATIONS;loop++){

        ////////////////////////////////// 1st decoder //////////////////////////////////

decoder_map(trel_state,channel_data,channel_parity1,size,stage_metric1,Log,1,loo
p);

        //////////////////////////////////

        for (i=0;i<size;i++){
            Log[0][i]=Log[0][i]-Lc*channel_data[i] - Log[1][i];
        }
    }
}

```

```

    }
    /*Interleaving */
    interleave_double(Log[0],size,interleave_mat);
    interleave_double(channel_data,size,interleave_mat);

    ////////////////////////////////// 2nd decoder //////////////////////////////////

decoder_map(trel_state,channel_data,channel_parity2,size,stage_metric2,Log,2,loop);

    //////////////////////////////////

    if(loop<ITERATIONS-1){

        for (i=0;i<size;i++){
            Log[1][i]=Log[1][i]-Lc*channel_data[i] - Log[0][i];
        }

        }
        deinterleave_double(channel_data,size,deinterleave_mat);
        deinterleave_double(Log[1],size,deinterleave_mat);
        deinterleave_double(Log[0],size,deinterleave_mat);
    }

    /* Makeing decisions */
    for (i=0;i<size;i++){

        if ( Log[1][i] > 0)
            channel_data[i] = 1.0;
        else
            channel_data[i] = -1.0;
    }

    ////////////////////////////////// free memory //////////////////////////////////
    for (i=0;i<size;i++){

        free(stage_metric1[i]);
        free(stage_metric2[i]);

    }
    free(stage_metric1);
    free(stage_metric2);

    for ( input=0;input<2;input++)
        free(Log[input]);

}

////////////////////////////////////
// Function name: deinterleave_double
// Description: deinterleave the given matrix
////////////////////////////////////

void deinterleave_double(double *channel_data, int size,unsigned
*deinterleave_mat)
{
    int i;
    double *temp;

    temp = (double*)malloc(size*sizeof(double));
    if(temp==NULL){

        printf("ERROR: Could not allocate place for temp\n");
    }
}

```

```

    }
    for (i=0;i<size;i++){
        temp[i] = channel_data[i];
    }

    for (i=0;i<size;i++){
        channel_data[deinterleave_mat[i]] = temp[i];
    }
    free(temp);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function name: interleave_double
// Description: interleave the given matrix
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void interleave_double(double *channel_data, int size,unsigned *interleave_mat)
{
    int i;
    double *temp;

    temp = (double*)malloc(size*sizeof(double));
    if(temp==NULL){
        printf("ERROR: Could not allocate place for temp\n");
    }
    for (i=0;i<size;i++){
        temp[i] =channel_data[i];
    }

    for (i=0;i<size;i++){
        channel_data[interleave_mat[i]] = temp[i];
    }
    free(temp);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function name: create_encode_table
// Description: Create encoder table based on the memory length and generator
matrices
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void create_encode_table(trel * trel_state,int state_num,int memory)
{
    register int i, j, result, temp, feed;
    int K; //the constraint length
    int data, init_state,temp_output,temp_state;

    for (init_state=0;init_state<state_num;init_state++){
        for (data=0;data<2;data++){

            feed=0;
            K=memory+1;
            temp=init_state<<1 | data ;

            for(i=1;i<=K;i++){

                feed=feed ^ ((generator[0]>>(K-i))&0x01) & ((temp>>(K-i)) &
0x01);

```



```

    }

    temp_output=0;
    temp=init_state<<1 | feed ;

    for(i=1;i<=K;i++){

        temp_output=temp_output ^ ((generator[1]>>(K-i))&0x01) &
        ((temp>>(K-i)) & 0x01);

    }
    trel_state[init_state].output[data]=temp_output;

    temp= pow(2,memory)-1;
    temp_state=init_state;
    trel_state[init_state].fin_state[data] = ((temp_state<<1)| feed) &
temp;
    trel_state[trel_state[init_state].fin_state[data]].from_state[data]
= init_state;
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function name: encoder
// Description: Encode data based on encoder table
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void encoder(trel *trel_state,int *data,int *parity,unsigned size, int force,
int memory)
{
    int i,j;
    int state;

    state=0;

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

        // force the encoder to zero state at the end
        if (i>=size-memory && force){

            if (trel_state[state].fin_state[0]&1){
                data[i] =1;
            }else{
                data[i] =0;
            }
        }
        j = data[i];

        /* Calculate output due to new msg bit */
        parity[i] = trel_state[state].output[j];
        /* Calculate the new state */
        state = trel_state[state].fin_state[j];
    }
}

```

```
////////////////////////////////////  
// Function name: interleave  
// Description: interleave the given matrix  
////////////////////////////////////
```

```
void interleave(int *data,unsigned size,unsigned *interleave_mat)  
{  
    int i,*temp;  
  
    temp = (int*)malloc(size*sizeof(int));  
    if(temp==NULL){  
  
        printf("ERROR: Could not allocate place for temp\n");  
    }  
    for (i=0;i<size;i++){  
  
        temp[i] = data[i];  
    }  
  
    for (i=0;i<size;i++){  
  
        data[interleave_mat[i]] = temp[i];  
    }  
    free(temp);  
}
```

```
////////////////////////////////////  
// Function name: deinterleave  
// Description: deinterleave the given matrix  
////////////////////////////////////
```

```
void deinterleave(int *data,unsigned size,unsigned *deinterleave_mat)  
{  
    int i,*temp;  
  
    temp = (int*)malloc(size*sizeof(int));  
    if(temp==NULL){  
  
        printf("ERROR: Could not allocate place for temp\n");  
    }  
  
    for (i=0;i<size;i++){  
  
        temp[i] = data[i];  
    }  
  
    for (i=0;i<size;i++){  
  
        data[deinterleave_mat[i]] = temp[i];  
    }  
    free(temp);  
}
```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function name: decoder_map
// Description: Decode codes in according to Maximum-a posteriori algorithm
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void decoder_map(trel * trel_state,double *channel_data, double *channel_parity,
int size, state_metric **stage_metric, double *Log[],int decode_num, int loop)
{
    int i,input,state,index,punc_index;
    double soft_input;
    double soft_output;
    double denominator;
    double min,temp;
    double num_deno[2];
    double Lc;

    Lc = 2.0/variance;
    if(decode_num==1){
        index=1;
        punc_index=1;
    }else{
        index=0;
        punc_index=2;
    }

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

        /* Calculate branch metric using punctured matrix*/
        for ( input=0;input<2;input++){

            soft_input = (input == 0) ? -1.0 : 1.0;

            for ( state=0;state<NUM_STATE;state++){

                soft_output = (trel_state[state].output[input] == 0) ? -1.0
: 1.0;

                if (punc_mat[punc_index][i%2]==1){

stage_metric[i][state].gamma_I[input]=exp(0.25*Lc*channel_parity[i]*soft_output)
;
                    }else{
                        stage_metric[i][state].gamma_I[input]=1.0;
                    }
                stage_metric[i][state].gamma[input]=
exp(0.25*soft_input*(Log[index][i]+
Lc*channel_data[i]))*stage_metric[i][state].gamma_I[input];
            }
        }
    }

    /* Calculate Forward metrics */
    for ( i=1;i<size;i++){

        denominator=0;
        /* Calculate denominator */
        for (state=0;state<NUM_STATE;state++){

            denominator+= stage_metric[i-
1][trel_state[state].from_state[0]].alpha

```

```

        * stage_metric[i-
1][trel_state[state].from_state[0]].gamma[0]
        + stage_metric[i-
1][trel_state[state].from_state[1]].alpha
        * stage_metric[i-
1][trel_state[state].from_state[1]].gamma[1];
    }
    /* Using normalization*/
    for (state=0;state<NUM_STATE;state++){

        stage_metric[i][state].alpha = ( stage_metric[i-
1][trel_state[state].from_state[0]].alpha
            * stage_metric[i-
1][trel_state[state].from_state[0]].gamma[0]
            + stage_metric[i-
1][trel_state[state].from_state[1]].alpha
            * stage_metric[i-
1][trel_state[state].from_state[1]].gamma[1] )
            / denominator;
    }
}

/* Calculate Backward metrics */

if (loop==0 && decode_num==2){

    denominator=0;

    /* Calculate denominator */
    for (state=0;state<NUM_STATE;state++){

        denominator += stage_metric[size-
1][trel_state[state].from_state[0]].alpha
            * stage_metric[size-
1][trel_state[state].from_state[0]].gamma[0]
            + stage_metric[size-
1][trel_state[state].from_state[1]].alpha
            * stage_metric[size-
1][trel_state[state].from_state[1]].gamma[1];
    }
    /* Using normalization*/
    for (state=0;state<NUM_STATE;state++){

        stage_metric[size-1][state].beta = (stage_metric[size-
1][trel_state[state].from_state[0]].alpha
            * stage_metric[size-
1][trel_state[state].from_state[0]].gamma[0]
            + stage_metric[size-
1][trel_state[state].from_state[1]].alpha
            * stage_metric[size-
1][trel_state[state].from_state[1]].gamma[1] )
            / denominator;
    }
}

for (i=size-1;i>=1;i--){

    denominator=0;
    /* Calculate denominator */
    for (state=0;state<NUM_STATE;state++){

        denominator +=
stage_metric[i][trel_state[state].from_state[0]].alpha

```

```

        *
stage_metric[i][trel_state[state].from_state[0]].gamma[0]
        +
stage_metric[i][trel_state[state].from_state[1]].alpha
        *
stage_metric[i][trel_state[state].from_state[1]].gamma[1];
    }
    /* Using normalization*/
    for (state=0;state<NUM_STATE;state++){

        stage_metric[i-1][state].beta = (
stage_metric[i][trel_state[state].fin_state[0]].beta
            * stage_metric[i][state].gamma[0]
        +
stage_metric[i][trel_state[state].fin_state[1]].beta
            * stage_metric[i][state].gamma[1] ) /
denominator;
    }

    /* Compute the extrinsic LLRs */
    for (i=0;i<size;i++){

        min=0;

        /* Find the minimum product of forward metics, backward metrics,
branch metrics */
        for ( input=0;input<2;input++){
            for (state=0;state<NUM_STATE;state++){

                temp = stage_metric[i][state].alpha
                    * stage_metric[i][state].gamma_I[input]
                    *
stage_metric[i][trel_state[state].fin_state[input]].beta;

                if ((temp < min && temp != 0) || min == 0)
                    min = temp;
            }
            // if all else fails, make min real small
            if (min == 0 || min > 1)
                min = 1E-100;

            for (input=0;input<2;input++){

                num_deno[input]=0.0;
                for(state=0;state<NUM_STATE;state++){

                    num_deno[input] += (stage_metric[i][state].alpha *
stage_metric[i][state].gamma_I[input]
                    *
stage_metric[i][trel_state[state].fin_state[input]].beta );
                }

                if (num_deno[0] == 0)
                    num_deno[0] = min;
                else if (num_deno[1] == 0)
                    num_deno[1] = min;

                Log[decode_num-1][i] = (log(num_deno[1]/num_deno[0]));
            }
        }
    }
}

```

```
        for (i=0;i<size;i++){
            Log[decode_num-1][i]=Log[decode_num-1][i] + Lc*channel_data[i] +
Log[index][i];
        }
}
////////////////////////////////// END //////////////////////////////////////
```

```

/////////////////////////////////////////////////////////////////
//
// Name: main_turbo.c
// Author: Tawfiqul Hasan Khan
// Description: It is a turbo simulation program. Here, the code rate is 1/3
//              It utilizes two Soft-Input Soft-Output MAP
//              decoder to decode the code. An user defined length of random
//              interleaver has been used to make sure that MAP decoder can
//              estimate information symbols independently at each iteration.
// Date of creation: 10/7/2006
/////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>

int ITERATIONS;    // Global variable for number of iterations
int NUM_STATE;    // Global variable for state number
double variance;  // Global variable for variance

/* Declaration of structure for trllis*/
typedef struct {
    int from_state[2];    // Initial state
    int fin_state[2];    // Final state
    int output[2];       // Output coded symbols (branch label)
}trel;

/* Declaration of structure for decoding*/
typedef struct{
    double alpha;        // Forward metric
    double beta;        // Back ward metric
    double gamma[2];    // Branch metric
    double gamma_I[2];  // Branch metric
}state_metric;

/* Functions prototypes */
int random_generator();
void* create_interleave_deinterleave(unsigned *interleave_mat,
                                     unsigned *deinterleave_mat, int size);
double gaussian_noise(double mean, double variance);
void encoder(trel *trel_state,int *data,int *parity,unsigned size, int force,
            int memory);
void create_encode_table(trel *trel_state,int state_num,int memory);
void interleave(int *data,unsigned size,unsigned *interleave_mat);
void deinterleave(int *data,unsigned size,unsigned *deinterleave_mat);
void decode(trel * trel_state,double *channel_data, double *channel_parity1,
            double *channel_parity2, int size, int *data,
            unsigned *interleave_mat,unsigned *deinterleave_mat);
void deinterleave_double(double *channel_data, int size,
                        unsigned *deinterleave_mat);
void interleave_double(double *channel_data, int size,unsigned *interleave_mat);
void decoder_map(trel * trel_state,double *channel_data, double *channel_parity,
                int size, state_metric **stage_metric, double *Log[],
                int decode_num, int loop);

int generator[10];

```

```

/* Main function*/
int main(int argc, char *argv[]){

    int *data, *parity1,*parity2;          // Variables for data, parity1 &
parity2
    double *channel_data, *channel_parity1,*channel_parity2; // Variables for
channel data, channel parity1, channel parity2
    trel *trel_state;
    unsigned *interleave_mat;             // Iterleaver matrix
    unsigned *deinterleave_mat;          // Deinterleaver matrix
    int size;                             // Interleaver or frame size
    double rate;                          // The code rate

    /* Declaration of some necessary variables*/
    unsigned long seed;
    int i,j;
    int memory;
    int temp1,temp2;
    int total_size;
    int total_error;
    int temp_data;
    double dB,init_dB,fin_dB,inc_dB;
    int max_sim;
    FILE *fp;

    //////////////////////////////////// USER INPUT ////////////////////////////////////
    printf("Enter the data size:");
    scanf("%d",&size);
    printf("Enter seed:");
    scanf("%ld",&seed);
    printf("Enter number of memory:");
    scanf("%d",&memory);
    printf("Enter number of iteration:");
    scanf("%d",&ITERATIONS);
    printf("Enter feedback generator in octal:");
    scanf("%o",&generator[0]);
    printf("Enter feedforward generator in octal:");
    scanf("%o",&generator[1]);

    //////////////////////////////////// INITIALIZATION ////////////////////////////////////

    data= (int*)malloc(size*sizeof(int));
    if(data==NULL){

        printf("ERROR: Could not allocate memory space for data\n");
        exit(0);
    }

    parity1= (int*)malloc(size*sizeof(int));
    if(parity1==NULL){

        printf("ERROR: Could not allocate memory space for parity1\n");
        exit(0);
    }

    parity2= (int*)malloc(size*sizeof(int));
    if(parity2==NULL){

        printf("ERROR: Could not allocate memory space for parity2\n");
        exit(0);
    }

```



```

    }

    channel_data= (double*)malloc(size*sizeof(double));
    if(channel_data==NULL){

        printf("ERROR: Could not allocate memory space for channel_data\n");
        exit(0);
    }

    channel_parity1= (double*)malloc(size*sizeof(double));
    if(channel_parity1==NULL){

        printf("ERROR: Could not allocate memory space for
channel_parity1\n");
        exit(0);
    }

    channel_parity2= (double*)malloc(size*sizeof(double));
    if(channel_parity2==NULL){

        printf("ERROR: Could not allocate memory space for
channel_parity2\n");
        exit(0);
    }

    interleave_mat= (int*)malloc(size*sizeof(int));
    if(interleave_mat==NULL){

        printf("ERROR: Could not allocate memory space for interleave_mat\n");
        exit(0);
    }

    deinterleave_mat= (int*)malloc(size*sizeof(int));
    if(deinterleave_mat==NULL){

        printf("ERROR: Could not allocate memory space for
deinterleave_mat\n");
        exit(0);
    }
    NUM_STATE= pow(2,memory);
    trel_state= (trel *)malloc(NUM_STATE*sizeof(trel));
    if(trel_state==NULL){

        printf("Could not allocate space for trel\n");
        exit(0);
    }
    fp=fopen("turbo.txt", "w");
    if(fp==NULL){

        printf("ERROR: Couldn't open file");
        exit(0);
    }
    ////////////////////////////////// Setting value for simulation //////////////////////////////////

    init_dB=0.0;
    fin_dB=4.50;
    inc_dB=0.5;
    max_sim=LONG_MAX;
    rate=0.3333;

    ////////////////////////////////// CREATE ENCODE TABLE //////////////////////////////////

    create_encode_table(trel_state,NUM_STATE,memory);

```



```

    }
    /* Decoding with two SISO MAP decoder*/
    decode (trel_state,channel_data,channel_parity1,channel_parity2,
           size,data,interleave_mat,deinterleave_mat);
    /* Checking error*/
    for (i=0;i<size;i++){

        temp_data = (channel_data[i] == 1) ? 1 : 0;

        if (data[i] != temp_data)
            total_error++;
    }

    total_size = total_size+size;
}
/* Printing simulation results in the console */
printf ("%5.2lf %10.7e %6ld %21ld %d\n", dB,
        (total_error/(double)total_size), total_error, total_size,
        size);
fprintf (fp,"%5.2lf %10.7e %6ld %21ld %d\n", dB,
        (total_error/(double)total_size), total_error, total_size,
        size);
}
//////////////////////////////////DEALLOCATING MEMORY//////////////////////////////////

free(data);
free(parity1);
free(parity2);
free(channel_data);
free(channel_parity1);
free(channel_parity2);
free(interleave_mat);
free(deinterleave_mat);
free(trel_state);
fclose(fp);
}

//////////////////////////////////
// Function name: random_generator()
// Description: Generate random numbers
//////////////////////////////////

int random_generator()
{
    double temp;

    temp=(double)rand()/LONG_MAX;

    if(temp>0.5){
        return 1;
    } else {
        return 0;
    }
}

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function name: create_interleave_deinterleave
// Description: Create interleave and deinterleave matrix
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void* create_interleave_deinterleave(unsigned *interleave_mat,unsigned
*deinterleave_mat, int size)
{
    int *temp, temp_pos,i;

    temp= (int*)malloc(size*sizeof(int));
    if(temp==NULL){

        return NULL;
    }

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

        temp[i] = 0;
    }
    /* Writing interleaver */
    for (i=0; (i<size); i++){

        do {
            temp_pos = rand()%size;
        } while ( temp[temp_pos] );

        temp[temp_pos] = 1;
        interleave_mat[i] = temp_pos;
    }
    /* Writing deinterleaver */
    for (i=0; (i<size); i++){

        deinterleave_mat[interleave_mat[i]]=i;
    }
    free(temp);
    return;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function name: gaussian_noise
// Description: Create gaussian noise with 0 mean and given variance
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

double gaussian_noise(double mean, double variance)
{
    double s1, s2, rand_n, r, g_noise;
    do {

        rand_n = (double)(rand())/LONG_MAX;
        s1 = rand_n * 2 - 1.0;

        rand_n = (double)(rand())/LONG_MAX;
        s2 = rand_n * 2 - 1.0;

        r = s1 * s1 + s2 * s2;
    } while( r >= 1 );

    g_noise =mean + sqrt(variance)*(s1 * sqrt( (-2.0*log(r))/r ));

    return g_noise;
}

```



```

        Log[0][i]=Log[0][i]-Lc*channel_data[i] - Log[1][i];
    }
    /*Interleaving */
    interleave_double(Log[0],size,interleave_mat);
    interleave_double(channel_data,size,interleave_mat);

    ////////////////////////////////// 2nd decoder //////////////////////////////////

decoder_map(trel_state,channel_data,channel_parity2,size,stage_metric2,Log,2,loop);

    //////////////////////////////////

    if(loop<ITERATIONS-1){

    for (i=0;i<size;i++){
        Log[1][i]=Log[1][i]-Lc*channel_data[i] - Log[0][i];
    }

    }
    deinterleave_double(channel_data,size,deinterleave_mat);
    deinterleave_double(Log[1],size,deinterleave_mat);
    deinterleave_double(Log[0],size,deinterleave_mat);
}

/* Makeing decisions */
for (i=0;i<size;i++){

    if ( Log[1][i] > 0)
        channel_data[i] = 1.0;
    else
        channel_data[i] = -1.0;
}

//////////////////////////////// free memory //////////////////////////////////
for (i=0;i<size;i++){

    free(stage_metric1[i]);
    free(stage_metric2[i]);

}
free(stage_metric1);
free(stage_metric2);

for ( input=0;input<2;input++)
    free(Log[input]);

}

////////////////////////////////
// Function name: deinterleave_double
// Description: deinterleave the given matrix
////////////////////////////////

void deinterleave_double(double *channel_data, int size,unsigned
*deinterleave_mat)
{
    int i;
    double *temp;

    temp = (double*)malloc(size*sizeof(double));
    if(temp==NULL){

```

```

        printf("ERROR: Could not allocate place for temp\n");
    }
    for (i=0;i<size;i++){
        temp[i] = channel_data[i];
    }

    for (i=0;i<size;i++){
        channel_data[deinterleave_mat[i]] = temp[i];
    }
    free(temp);
}

/////////////////////////////////////////////////////////////////
// Function name: interleave_double
// Description: interleave the given matrix
/////////////////////////////////////////////////////////////////

void interleave_double(double *channel_data, int size,unsigned *interleave_mat)
{
    int i;
    double *temp;

    temp = (double*)malloc(size*sizeof(double));
    if(temp==NULL){
        printf("ERROR: Could not allocate place for temp\n");
    }
    for (i=0;i<size;i++){
        temp[i] =channel_data[i];
    }

    for (i=0;i<size;i++){
        channel_data[interleave_mat[i]] = temp[i];
    }
    free(temp);
}

/////////////////////////////////////////////////////////////////
// Function name: create_encode_table
// Description: Create encoder table based on the memory length and generator
matrices
/////////////////////////////////////////////////////////////////

void create_encode_table(trel * trel_state,int state_num,int memory)
{
    register int i, j, result, temp, feed;
    int K; //the constraint length
    int data, init_state,temp_output,temp_state;

    for (init_state=0;init_state<state_num;init_state++){
        for (data=0;data<2;data++){

            feed=0;
            K=memory+1;
            temp=init_state<<1 | data ;

            for(i=1;i<=K;i++){

                feed=feed ^ ((generator[0]>>(K-i))&0x01) & ((temp>>(K-i)) &
0x01);

```

```

    }

    temp_output=0;
    temp=init_state<<1 | feed ;

    for(i=1;i<=K;i++){

        temp_output=temp_output ^ ((generator[1]>>(K-i))&0x01) &
((temp>>(K-i)) & 0x01);

    }
    trel_state[init_state].output[data]=temp_output;

    temp= pow(2,memory)-1;
    temp_state=init_state;
    trel_state[init_state].fin_state[data] = ((temp_state<<1| feed) &
temp;
    trel_state[trel_state[init_state].fin_state[data]].from_state[data]
= init_state;
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function name: encoder
// Description: Encode data based on encoder table
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void encoder(trel *trel_state,int *data,int *parity,unsigned size, int force,
int memory)
{
    int i,j;
    int state;

    state=0;

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

        // force the encoder to zero state at the end
        if (i>=size-memory && force){

            if (trel_state[state].fin_state[0]&1){
                data[i] =1;
            }else{
                data[i] =0;
            }
        }
        j = data[i];

        /* Calculate output due to new mesg bit */
        parity[i] = trel_state[state].output[j];
        /* Calculate the new state */
        state = trel_state[state].fin_state[j];
    }
}

```



```
////////////////////////////////////  
// Function name: interleave  
// Description: interleave the given matrix  
////////////////////////////////////  
  
void interleave(int *data,unsigned size,unsigned *interleave_mat)  
{  
    int i,*temp;  
  
    temp = (int*)malloc(size*sizeof(int));  
    if(temp==NULL){  
  
        printf("ERROR: Could not allocate place for temp\n");  
    }  
    for (i=0;i<size;i++){  
  
        temp[i] = data[i];  
    }  
  
    for (i=0;i<size;i++){  
  
        data[interleave_mat[i]] = temp[i];  
    }  
    free(temp);  
}
```

```
////////////////////////////////////  
// Function name: deinterleave  
// Description: deinterleave the given matrix  
////////////////////////////////////  
  
void deinterleave(int *data,unsigned size,unsigned *deinterleave_mat)  
{  
    int i,*temp;  
  
    temp = (int*)malloc(size*sizeof(int));  
    if(temp==NULL){  
  
        printf("ERROR: Could not allocate place for temp\n");  
    }  
  
    for (i=0;i<size;i++){  
  
        temp[i] = data[i];  
    }  
  
    for (i=0;i<size;i++){  
  
        data[deinterleave_mat[i]] = temp[i];  
    }  
    free(temp);  
}
```



```

    }
    /* Using normalization*/
    for (state=0;state<NUM_STATE;state++){

        stage_metric[i][state].alpha = ( stage_metric[i-
1][trel_state[state].from_state[0]].alpha
            * stage_metric[i-
1][trel_state[state].from_state[0]].gamma[0]
        + stage_metric[i-
1][trel_state[state].from_state[1]].alpha
            * stage_metric[i-
1][trel_state[state].from_state[1]].gamma[1] )
            / denominator;
    }
}

/* Calculate Backward metrics */

if (loop==0 && decode_num==2){

    denominator=0;

    /* Calculate denominator */
    for (state=0;state<NUM_STATE;state++){

        denominator += stage_metric[size-
1][trel_state[state].from_state[0]].alpha
            * stage_metric[size-
1][trel_state[state].from_state[0]].gamma[0]
        + stage_metric[size-
1][trel_state[state].from_state[1]].alpha
            * stage_metric[size-
1][trel_state[state].from_state[1]].gamma[1];
    }
    /* Using normalization*/
    for (state=0;state<NUM_STATE;state++){

        stage_metric[size-1][state].beta = (stage_metric[size-
1][trel_state[state].from_state[0]].alpha
            * stage_metric[size-
1][trel_state[state].from_state[0]].gamma[0]
        + stage_metric[size-
1][trel_state[state].from_state[1]].alpha
            * stage_metric[size-
1][trel_state[state].from_state[1]].gamma[1] )
            / denominator;
    }
}

for (i=size-1;i>=1;i--){

    denominator=0;
    /* Calculate denominator */
    for (state=0;state<NUM_STATE;state++){

        denominator +=
stage_metric[i][trel_state[state].from_state[0]].alpha
            *
stage_metric[i][trel_state[state].from_state[0]].gamma[0]
        +
stage_metric[i][trel_state[state].from_state[1]].alpha
            *
stage_metric[i][trel_state[state].from_state[1]].gamma[1];
    }
}

```

```

/* Using normalization*/
for (state=0;state<NUM_STATE;state++){

    stage_metric[i-1][state].beta = (
stage_metric[i][trel_state[state].fin_state[0]].beta
        * stage_metric[i][state].gamma[0]
    +
stage_metric[i][trel_state[state].fin_state[1]].beta
        * stage_metric[i][state].gamma[1] ) /
denominator;
    }
}

/* Compute the extrinsic LLRs */
for (i=0;i<size;i++){

    min=0;

    /* Find the minimum product of forward metics, backward metrics,
branch metrics */
    for ( input=0;input<2;input++){
        for (state=0;state<NUM_STATE;state++){

            temp = stage_metric[i][state].alpha
                * stage_metric[i][state].gamma_I[input]
                *
stage_metric[i][trel_state[state].fin_state[input]].beta;

            if ((temp < min && temp != 0) || min == 0)
                min = temp;
        }
    }
    // if all else fails, make min real small
    if (min == 0 || min > 1)
        min = 1E-100;

    for (input=0;input<2;input++){

        num_deno[input]=0.0;
        for(state=0;state<NUM_STATE;state++){

            num_deno[input] += (stage_metric[i][state].alpha *
stage_metric[i][state].gamma_I[input]
                *
stage_metric[i][trel_state[state].fin_state[input]].beta );
        }

        if (num_deno[0] == 0)
            num_deno[0] = min;
        else if (num_deno[1] == 0)
            num_deno[1] = min;

        Log[decode_num-1][i] = (log(num_deno[1]/num_deno[0]));
    }
    for (i=0;i<size;i++){
        Log[decode_num-1][i]=Log[decode_num-1][i] + Lc*channel_data[i] +
Log[index][i];
    }
}
////////////////////////////////// END //////////////////////////////////////

```

```

/////////////////////////////////////////////////////////////////
//
// Name: main_ldpc.c
// Author: Tawfiqul Hasan Khan
// Description: It is an LDPC simulation program. It uses the low-density
//               parity-check matrix and iterative belief propagation (IBP)
//               decoding algorithm for decoding LDPC.
// Date of creation: 13/7/2006
/////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <math.h>
#include <float.h>
#include <limits.h>
#include <stdlib.h>

int ITERATIONS;      // Global variable for number of iterations
double variance;     // Global variable for variance

/* Declaration of structure for parent nodes */
typedef struct parent_node {
    int size;          // column weigth in low-density parity
    check matrix
    int *index_child; // indexes of children
}node_parent;

/* Declaration of structure for child nodes */

typedef struct {
    int size;          // row weigth in low-density parity
    check matrix
    int *index_parent; // indexes of parents
    int syndrome;
}node_child;

/* Function prototypes*/
void perl_decoder(int N,int M, double *channel_data, node_parent *node_code,
node_child *node_check, int *decoded_data);
double gaussian_noise(double mean, double variance);

/* Main function*/
main()
{
    int N;          // Transmitted blocklength of a code
    int K;          // Source blocklength of a code
    int M;          // Number of rows in A. M=N-K
    int L;          // Number of columns of A. L=N
    float rate;
    int *data, *decoded_data;
    double *channel_data;

    node_parent *node_code;
    node_child *node_check;

    int max_size_t; // maximam number of 1's per column
    int max_size_tr; // maximum nuber of 1's per row

```

```

/* Declaration of some necessary variables*/
int total_size;
int total_error;
double dB,init_dB,fin_dB,inc_dB;
int max_sim;
long seed;
int error;
FILE *fp, *fp2;
int i,j;
char filename[20];

//////////////////////////////// FILE Input //////////////////////////////////

printf("Enter file of low-density parity check matrix:");
scanf("%s",filename);

fp=fopen(filename,"r");
if (fp == NULL){

    printf("incorrect input file name ...\n");
    exit(0);
}
fp2=fopen("LDPC_output.txt","w");
if(fp2==NULL){
    printf("incorrect input file name ...\n");
    exit(0);
}
/* Reading the matrix*/
fscanf(fp, "%d %d", &N, &K);
M=N-K;
L=N;

//////////////////////////////// INITIALIZATION //////////////////////////////////
data=(int*)malloc(K*sizeof(int));
if(data==NULL){

    printf("Error: cannot allocate spcae for data\n");
}

decoded_data= (int*)malloc(N*sizeof(int));
if(decoded_data==NULL){

    printf("Error: cannot allocate space for decoded_data");
}

channel_data=(double*)malloc(N*sizeof(double));
if(channel_data==NULL){

    printf("Error: cannot allocate space for channel_data");
}

fscanf(fp, "%d %d", &max_size_t, &max_size_tr);

node_code=(node_parent*)malloc(N*sizeof(node_parent));
node_check=(node_child*)malloc(M*sizeof(node_child));

for (i=0; i<L; i++){
    fscanf(fp, "%d", &node_code[i].size);
    node_code[i].index_child=(int*)malloc(node_code[i].size*sizeof(int));
}

```



```

printf("%d %d\n", max_size_t, max_size_tr);
for (i=0; i<L; i++)
    printf("%4d", size_t_mat[i]);
printf("\n");

for (i=0; i<M; i++)
    printf("%4d", size_tr_mat[i]);
printf("\n");

for (i=0; i<L; i++)
{
    for (j=0; j<size_t_mat[i]; j++)
        printf("%4d", set_t_mat[i][j]);
    printf("\n");
}
printf("\n");
for (i=0; i<M; i++)
{
    for (j=0; j<size_tr_mat[i]; j++)
        printf("%4d", set_tr_mat[i][j]);
    printf("\n");
}
printf("\n");
*/
//////////////////////////////// check end //////////////////////////////////
srandom(seed);

//////////////////////////////// STARTING SIMULATION //////////////////////////////////
dB=init_dB;
for (dB=init_dB; dB<fin_dB+1.0e-1; dB += inc_dB){

    total_size=0;
    total_error=0;
    variance = 1.0/(2.0*rate*pow(10.0,dB/10.0));
    //sigma=1.0/(2.0*0.3333*dB);
    //sigma = 1.0/(2.0*3*(dB/));
    // snr_rms = 2.0*rate*(pow(10.0,(snr/10.0))); // SNR per bit
    // snr_rms = 2.0*(pow(10.0,(snr/10.0))); // SNR per symbol
    //snr_rms = 2.0*sqrt(2.0*rate*(pow(10.0,(snr/10.0)))); // SNR per
bit

    while (total_error<=1000 && total_size< max_sim){

        /* FOR CONVENIENCE, MAKE DATA EQUAL TO ZERO */
        for (i=0; i<K; i++){
            data[i] = 0;
        }
        /* BPSK MAPPING: "0" --> +1, "1" --> -1 */
        for (i=0; i<N; i++){
            channel_data[i] = 1.0;
        }
        /* ADDITIVE WHITE GAUSSIAN NOISE CHANNEL */

        for (i=0; i<N; i++){
            channel_data[i] = channel_data[i]+
gaussian_noise(0,variance);
        }

        /* ITERATIVE DECODING BY BELIEF PROPAGATION */

        perl_decoder(N,
M,channel_data,node_code,node_check,decoded_data);

        /* Checking error */

```



```

        for (i=0; i<N; i++){
            if (decoded_data[i]){
                total_error++;
            }
        }

        total_size = total_size+N;
    }
    printf ("%5.2lf %10.7e %6ld %21ld %d\n", dB,
            (total_error/(double)total_size), total_error, total_size,
            N);
    fprintf (fp2,"%5.2lf %10.7e %6ld %21ld %d\n", dB,
            (total_error/(double)total_size), total_error, total_size,
            N);
}
//////////////////////////////////DEALLOCATING MEMORY//////////////////////////////////
free(data);
free(decoded_data);
free(channel_data);
for(i=0; i<L; i++){
    free(node_code[i].index_child);
}
for(i=0; i<M; i++){
    free(node_check[i].index_parent);
}
free(node_code);
free(node_check);
fclose(fp);
fclose(fp2);
}

//////////////////////////////////
// Function name: perl_decoder()
// Description: Iterative decoding by belief propagation in code's B
//              ayesian network Based on Pearl's book and MacKay's paper
//////////////////////////////////

void perl_decoder(int N,int M, double *channel_data, node_parent *node_code,
node_child *node_check, int *decoded_data)
{

    int i,j,l,loop;
    double *p_l1,*p_l0;
    double **q_ml0, **q_ml1;
    double **r_ml0, **r_ml1;
    double **deltar_ml;
    double alpha;
    double *q_l0, *q_l1;
    double Lc;
    // -----
    // INITIALIZATION STEP
    // -----
    p_l1=(double*)malloc(N*sizeof(double));
    if(p_l1==NULL){

        printf("Error: cannot allocate space for p_l1");
    }

    p_l0=(double*)malloc(N*sizeof(double));
    if(p_l0==NULL){

        printf("Error: cannot allocate space for p_l0");
    }
}

```

```

}

q_l0=(double*)malloc(N*sizeof(double));
if(q_l0==NULL){

    printf("Error: cannot allocate space for q_l0");
}

q_l1=(double*)malloc(N*sizeof(double));
if(q_l1==NULL){

    printf("Error: cannot allocate space for q_l1");
}

q_ml0=(double**)malloc(M*sizeof(double*));
if(q_ml0==NULL){

    printf("Error: cannot allocate space for q_ml0");
}

q_ml1=(double**)malloc(M*sizeof(double*));
if(q_ml1==NULL){

    printf("Error: cannot allocate space for q_ml1");
}

deltar_ml=(double**)malloc(M*sizeof(double*));
if(deltar_ml==NULL){

    printf("Error: cannot allocate space for deltar_ml");
}

r_ml0= (double**)malloc(M*sizeof(double*));
if(r_ml0==NULL){

    printf("Error: cannot allocate space for r_ml0");
}

r_ml1=(double**)malloc(M*sizeof(double*));
if(r_ml1==NULL){

    printf("Error: cannot allocate space for r_ml1");
}

for (i=0; i<M; i++){
    q_ml0[i]=(double*)malloc(N*sizeof(double));
    q_ml1[i]=(double*)malloc(N*sizeof(double));
    deltar_ml[i]=(double*)malloc(N*sizeof(double));
    r_ml0[i]= (double*)malloc(N*sizeof(double));
    r_ml1[i]= (double*)malloc(N*sizeof(double));
}

Lc=2.0/variance;

// Prior probabilities

for (i=0;i<N;i++)
{
    p_l1[i] = 1.0 / ( 1.0 + exp(channel_data[i]*Lc) );
    p_l0[i] = 1.0 - p_l1[i];
}

for (i=0; i<M; i++){
    for (j=0; j<node_check[i].size; j++)
    {
        q_ml0[i][j] = 0.0;
    }
}

```

```

        q_ml1[i][j] = 0.0;
    }
}
for (i=0; i<N; i++){ // run over code
nodes
    for (j=0; j<node_code[i].size; j++){ // run over
children nodes
        q_ml0[node_code[i].index_child[j]-1][i] = p_l0[i];
        q_ml1[node_code[i].index_child[j]-1][i] = p_l1[i];
    }
}
for(loop=0;loop<ITERATIONS;loop++){
////////// Horizontal step //////////
    for (i=0; i<M; i++) {
        for (j=0; j<node_check[i].size; j++)
        {
            deltar_ml[i][node_check[i].index_parent[j]-1] = 1.0;
            for (l=0; l<node_check[i].size; l++)
            {
                if (node_check[i].index_parent[l] !=
node_check[i].index_parent[j]){
                    deltar_ml[i][node_check[i].index_parent[j]-1] *= (
q_ml0[i][node_check[i].index_parent[l]-1] -
q_ml1[i][node_check[i].index_parent[l]-1]);
                }
            }
            r_ml0[i][node_check[i].index_parent[j]-1] = 0.5 * ( 1.0 +
deltar_ml[i][node_check[i].index_parent[j]-1] );
            r_ml1[i][node_check[i].index_parent[j]-1] = 0.5 * ( 1.0 -
deltar_ml[i][node_check[i].index_parent[j]-1] );
        }
    }
}
////////// Vertical step //////////
for (i=0; i<N; i++){
    for (j=0; j<node_code[i].size; j++)
    {
        q_ml0[node_code[i].index_child[j]-1][i] = p_l0[i];
        q_ml1[node_code[i].index_child[j]-1][i] = p_l1[i];
        for (l=0; l<node_code[i].size; l++){
            if (node_code[i].index_child[l] !=
node_code[i].index_child[j])
            {
                q_ml0[node_code[i].index_child[j]-1][i] *=
r_ml0[node_code[i].index_child[l]-1][i];
                q_ml1[node_code[i].index_child[j]-1][i] *=
r_ml1[node_code[i].index_child[l]-1][i];
            }
        }
    }
}

```

```

        }

        alpha = 1.0 / (q_ml0[node_code[i].index_child[j]-
1][i]+q_ml1[node_code[i].index_child[j]-1][i]);

        q_ml0[node_code[i].index_child[j]-1][i] *= alpha;
        q_ml1[node_code[i].index_child[j]-1][i] *= alpha;
    }
}

for (i=0; i<N; i++)
{
    q_l0[i] = p_l0[i];
    q_l1[i] = p_l1[i];

    for (j=0; j<node_code[i].size; j++)
    {
        q_l0[i] *= r_ml0[node_code[i].index_child[j]-1][i];
        q_l1[i] *= r_ml1[node_code[i].index_child[j]-1][i];
    }

    alpha = 1.0 / (q_l0[i]+q_l1[i]);

    q_l0[i] *= alpha;
    q_l1[i] *= alpha;

    if (q_l1[i] > 0.5)
        decoded_data[i] = 1;
    else
        decoded_data[i] = 0;
}
}

for (i=0; i<M; i++){
    free(q_ml0[i]);
    free(q_ml1[i]);
    free(deltar_ml[i]);
    free(r_ml0[i]);
    free(r_ml1[i]);
}
free(p_l1);
free(p_l0);
free(q_l0);
free(q_l1);
free(q_ml0);
free(q_ml1);
free(deltar_ml);
free(r_ml0);
free(r_ml1);
}

```

```
////////////////////////////////////  
// Function name: gaussian_noise  
// Description: Create gaussian noise with 0 mean and given variance  
////////////////////////////////////  
  
double gaussian_noise(double mean, double variance)  
{  
    double s1, s2, rand_n, r, g_noise;  
    do {  
  
        rand_n = (double)(rand())/LONG_MAX;  
        s1 = rand_n * 2 - 1.0;  
  
        rand_n = (double)(rand())/LONG_MAX;  
        s2 = rand_n * 2 - 1.0;  
  
        r = s1 * s1 + s2 * s2;  
    } while( r >= 1 );  
  
    g_noise =mean + sqrt(variance)*(s1 * sqrt( (-2.0*log(r))/r ));  
  
    return g_noise;  
}
```

```

/////////////////////////////////////////////////////////////////
//
//      Filename:          gal(3,4,20,5)r1_4.txt
//      Description:      It is a text file which contains the information of a
//                        specific low-density parity-check matrix. Here,  $N =$ 
//                        20,  $K = 5$ ,  $R = 0.25$ ,  $W_c = 3$  and  $W_r = 4$ . The format
//                        of the files specifying the structure of these
//                        matrices (or the Tanner graphs) is the same as that
//                        used by David MacKay, that is, the "alist" (adjacency
//                        list) format. (Mackay, D J C)
//
/////////////////////////////////////////////////////////////////

```

```

20 5
3 4
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
1 6 11
1 7 12
1 8 13
1 9 14
2 6 15
2 7 11
2 8 12
2 10 13
3 6 14
3 7 15
3 9 12
3 10 11
4 6 13
4 8 14
4 9 15
4 10 12
5 7 14
5 8 11
5 9 13
5 10 15
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
17 18 19 20
1 5 9 13
2 6 10 17
3 7 14 18
4 11 15 19

```

8 12 16 20
1 6 12 18
2 7 11 16
3 8 13 19
4 9 14 17
5 10 15 20

8	18	38
4	19	45
6	24	40
9	27	39
13	17	42
11	29	44
8	24	34
6	25	36
9	19	43
1	20	46
14	27	42
7	22	39
13	18	35
4	26	40
16	29	38
15	21	48
11	23	45
3	17	47
5	28	44
12	32	33
2	30	41
10	31	37
10	18	36
4	23	44
9	29	40
2	27	38
8	30	42
12	28	43
11	20	37
1	19	35
15	31	39
16	32	41
5	26	33
3	25	45
13	21	34
14	24	48
7	17	46
6	22	47
7	27	40
11	18	33
2	32	35
10	28	47
5	24	41

12	25	37			
3	19	39			
14	31	44			
16	30	34			
13	20	38			
9	22	36			
6	17	45			
4	21	42			
15	29	46			
8	26	43			
1	23	48			
1	25	42			
15	22	40			
8	21	41			
9	18	47			
6	27	43			
11	30	46			
7	31	35			
5	20	36			
14	17	38			
16	28	45			
4	32	37			
13	23	33			
12	26	44			
3	29	48			
2	24	39			
10	19	34			
8	20	36	56	80	81
6	19	47	52	67	95
10	25	44	60	71	94
9	28	40	50	77	91
2	18	45	59	69	88
15	29	34	64	76	85
12	21	38	63	65	87
13	27	33	53	79	83
7	30	35	51	75	84
1	22	48	49	68	96
11	32	43	55	66	86
4	26	46	54	70	93
16	31	39	61	74	92
14	17	37	62	72	89
5	23	42	57	78	82
3	24	41	58	73	90

6	31	44	63	76	89
3	27	39	49	66	84
5	28	35	56	71	96
13	26	36	55	74	88
12	22	42	61	77	83
4	25	38	64	75	82
14	18	43	50	80	92
7	29	33	62	69	95
16	21	34	60	70	81
10	24	40	59	79	93
9	30	37	52	65	85
15	20	45	54	68	90
8	32	41	51	78	94
1	23	47	53	73	86
11	19	48	57	72	87
2	17	46	58	67	91
8	22	46	59	66	92
6	20	33	61	73	96
10	23	39	56	67	87
9	19	34	49	75	88
15	18	48	55	70	91
4	27	41	52	74	89
3	30	38	57	71	95
1	29	40	51	65	82
16	26	47	58	69	83
7	31	37	53	77	81
11	17	35	54	79	85
12	32	45	50	72	93
2	28	43	60	76	90
14	25	36	63	78	86
5	21	44	64	68	84
13	24	42	62	80	94

46	232	365
135	246	316
21	137	315
73	264	390
134	161	282
25	200	407
7	224	333
113	178	342
122	168	299
92	216	300
52	195	285
32	186	374
82	258	288
104	228	281
49	271	402
130	242	328
89	172	312
2	260	364
63	208	284
131	158	375
83	243	373
16	252	329
35	138	331
18	150	320
56	213	274
120	181	279
43	269	406
88	196	283
71	259	273
111	176	309
85	241	359
4	212	347
101	143	295
107	263	326
13	190	389
30	222	353
65	185	318
98	230	357
67	223	354
97	193	336
84	202	358
55	162	385
68	210	308
51	152	287
116	147	291
100	154	394
102	231	381
81	235	363
99	261	352
117	237	361
93	272	401
10	214	384
94	221	378
26	144	395
69	146	379
6	217	292
14	205	372
125	247	296
62	164	360
112	183	332
9	262	396
74	141	311
106	198	286
19	160	367

28	175	322
11	239	339
8	166	369
15	225	297
29	254	400
27	174	307
48	177	293
126	233	314
3	219	371
87	187	345
90	153	334
59	170	341
54	203	317
119	211	362
41	182	338
129	234	408
37	155	346
31	169	294
22	240	343
91	145	330
53	265	403
95	251	356
124	148	323
79	156	388
115	266	399
105	220	398
80	191	302
60	165	366
77	197	380
121	159	305
61	151	335
109	244	298
17	194	313
128	149	303
96	171	337
70	180	327
38	142	350
103	257	304
24	188	321
40	163	276
20	255	386
114	206	393
86	139	275
36	226	383
127	268	289
5	267	280
110	236	377
123	184	351
45	215	278
64	248	324
47	167	368
76	270	277
39	192	355
136	245	310
44	229	370
1	256	290
136	159	283
18	233	301
119	172	333
101	259	342
81	164	408
13	165	344
74	153	305
35	150	396

2	155	399
86	149	349
67	232	352
92	242	379
129	272	336
22	190	281
111	268	322
46	193	335
91	207	407
95	271	290
118	212	286
10	175	296
107	249	330
11	171	405
14	148	280
52	191	313
131	209	288
24	187	341
113	203	326
44	160	402
115	178	350
6	188	381
66	170	277
134	139	347
1	266	406
80	168	383
9	201	353
50	185	339
121	184	299
130	198	401
41	157	390
37	262	345
85	230	354
79	146	395
8	237	317
76	253	340
27	228	363
93	247	403
54	152	324
33	169	285
97	223	400
5	260	318
36	231	372
127	239	308
98	263	276
83	250	359
84	158	370
100	225	404
42	154	362
124	167	394
88	218	355
71	220	360
72	243	316
112	258	382
47	163	315
82	204	274
123	261	298
99	210	384
114	145	289
65	162	393
106	226	309
40	195	378
23	140	292
62	143	311

60	248	278
108	216	369
34	251	295
53	245	391
56	246	374
120	227	351
43	199	377
61	221	282
25	224	294
55	202	357
125	213	334
73	229	297
29	264	367
19	252	371
59	234	273
90	217	366
21	254	329
12	192	397
68	206	388
78	138	368
75	196	314
28	177	380
39	182	323
128	161	320
38	244	321
20	256	385
109	222	287
63	142	373
94	208	328
49	180	398
64	255	331
122	174	343
58	194	375
7	238	303
103	166	332
51	151	338
26	240	361
77	200	386
126	215	327
69	219	358
15	179	319
105	211	356
17	265	284
16	176	364
117	236	348
110	241	291
45	257	389
116	183	325
57	267	387
4	186	306
3	156	279
32	137	307
133	269	376
70	235	293
96	270	275
87	141	302
89	147	392
135	214	346
132	144	365
31	197	337
104	189	300
30	173	304
48	205	310
102	181	312

31	255	307
12	259	313
46	196	299
8	199	290
74	260	396
79	157	358
55	250	329
24	262	367
131	238	273
99	166	278
111	189	277
6	231	335
77	148	369
57	251	399
58	211	406
62	269	383
7	226	280
101	202	287
36	252	276
100	216	322
78	270	298
129	197	366
21	228	387
121	187	295
26	173	274
3	164	300
92	156	332
72	220	365
81	139	286
96	249	403
76	218	378
19	258	324
22	188	357
41	143	379
11	241	351
17	152	279
63	194	391
47	215	334
33	263	315
122	204	301
126	140	353
10	176	331
34	268	360
38	185	337
119	168	341
1	240	354
68	183	328
83	219	377
104	146	305
69	266	349
18	158	359
136	213	333
51	248	398
49	167	314
70	272	350
39	191	327
113	175	308
9	267	372
90	178	384
97	221	326
125	142	407
30	237	364
23	222	302
16	201	371

45	243	294
28	264	292
94	165	374
85	217	344
128	169	356
88	181	296
112	239	316
5	141	395
95	223	373
84	172	363
66	208	303
67	236	380
87	150	283
44	161	346
133	145	389
120	170	394
64	138	402
105	206	291
110	232	297
91	190	338
43	247	330
25	195	355
116	203	390
107	171	320
60	227	352
123	160	405
93	244	318
89	200	317
13	182	312
115	163	370
80	261	397
134	180	339
132	257	393
50	154	336
52	179	400
37	233	382
103	162	284
98	265	311
124	174	408
56	198	361
14	242	325
48	256	386
29	224	376
15	230	310
114	246	342
53	151	348
109	155	304
130	177	309
65	225	285
102	253	362
117	234	293
135	212	282
75	137	321
20	209	347
118	192	281
42	210	289
54	184	368
106	186	323
4	159	404
86	214	381
35	235	306
71	245	388
40	147	343
2	144	345

59	153	375
127	207	288
27	271	401
61	149	275
82	229	385
108	254	319
32	193	392
73	205	340
86	166	402
59	237	274
3	194	387
92	248	382
129	271	313
71	184	285
132	263	361
102	249	293
46	173	384
22	247	406
135	145	325
44	183	398
77	170	305
114	156	308
121	140	376
63	169	351
17	209	399
19	244	368
117	196	302
69	220	281
124	153	294
84	149	357
81	189	370
56	165	360
58	137	356
24	268	358
36	243	347
5	206	341
123	190	296
23	171	407
31	158	344
96	262	374
97	192	315
15	148	350
119	197	306
111	232	362
116	238	403
95	185	379
122	259	348
54	151	342
118	208	278
20	163	390
108	200	345
6	168	324
32	224	286
16	245	343
78	267	273
80	138	337
39	255	391
65	222	372
115	188	335
48	157	383
2	191	330
76	241	364
93	160	304
28	150	392

126	155	279
62	210	307
67	235	322
120	240	311
51	226	326
1	152	401
85	146	318
70	239	352
125	167	300
7	198	327
127	159	373
37	172	367
113	234	359
74	261	323
14	203	408
34	218	289
101	181	339
53	251	371
105	233	363
35	162	328
82	182	336
104	199	275
75	174	333
57	264	353
68	180	287
79	219	298
26	141	346
73	214	334
33	236	338
131	177	331
42	217	396
60	266	378
98	175	301
83	246	388
128	201	314
38	228	283
13	215	284
90	142	297
47	265	366
87	269	389
11	154	380
66	147	385
4	260	365
55	230	292
130	193	288
30	242	400
88	250	280
91	161	405
103	252	354
12	223	290
43	212	404
10	139	332
40	178	320
25	204	395
45	270	312
72	213	309
29	179	291
9	256	375
94	143	329
107	229	340
100	187	299
112	205	394
136	227	316
64	257	397

52	231	319
41	144	349
21	258	277
134	164	386
49	272	377
110	211	355
109	216	303
99	254	310
8	207	321
61	176	317
106	225	276
133	253	381
89	186	295
50	195	393
18	202	282
27	221	369
42	230	404
131	152	281
70	271	383
21	224	407
135	218	332
96	189	399
79	186	380
108	210	282
2	199	344
24	249	390
136	196	298
33	206	277
67	246	275
71	237	402
37	190	358
10	179	369
93	255	367
18	223	297
113	170	319
104	258	294
41	264	374
14	175	333
95	174	341
73	201	305
44	137	405
64	250	364
38	236	317
46	239	309
49	147	280
15	182	291
25	146	331
35	229	403
63	247	368
76	144	350
88	202	406
123	191	391
62	198	340
60	155	299
40	254	300
109	211	316
85	178	353
98	145	345
22	205	335
19	272	396
99	215	289
118	140	308
50	159	355
1	139	292

66	265	334
56	267	306
12	238	283
121	194	290
68	177	276
132	245	329
84	154	349
59	253	392
133	157	286
27	240	348
16	243	339
122	221	352
87	268	354
106	209	328
61	192	384
94	225	393
74	213	303
51	176	363
13	171	323
72	172	371
54	252	366
11	219	394
58	165	387
82	166	287
65	248	378
119	242	376
91	162	310
124	261	400
31	143	362
26	169	338
20	149	343
48	188	285
28	241	395
100	150	326
3	167	397
77	141	342
78	161	361
111	207	304
5	183	359
101	256	381
92	204	372
52	156	336
130	222	314
4	184	302
110	200	357
23	185	346
6	235	288
45	260	301
107	160	293
80	180	351
57	195	307
112	214	324
102	193	273
89	203	311
134	142	360
7	164	320
117	257	330
86	138	370
53	148	274
29	269	373
127	226	321
105	158	385
39	153	318
36	151	327

126	263	389
75	220	386
17	208	365
120	259	278
103	168	325
90	244	296
8	262	347
47	251	337
116	234	382
9	227	379
97	232	284
69	163	279
81	212	295
43	228	322
129	266	388
30	197	356
128	216	313
32	187	312
55	173	375
114	181	377
34	233	408
115	270	401
125	231	315
83	217	398
105	224	290
3	271	274
113	195	386
61	173	300
111	158	286
79	160	366
106	245	371
52	259	352
84	159	345
107	253	295
108	156	350
34	201	383
28	222	355
90	189	344
97	171	299
44	144	326
80	251	372
48	184	388
83	226	381
54	221	308
95	244	405
47	256	382
96	148	305
88	214	374
73	137	293
29	155	406
128	225	384
98	232	325
2	231	320
57	162	390
114	212	292
21	220	328
62	268	329
85	206	304
17	235	376
19	247	339
18	198	379
66	203	294
75	250	337
129	179	309

50	138	334
7	163	287
32	257	394
20	266	332
39	176	392
103	152	389
136	249	380
15	269	378
55	141	359
30	200	373
9	241	368
26	238	278
89	246	298
10	218	330
69	157	327
93	252	316
38	172	275
64	183	289
112	161	348
126	181	370
1	270	284
100	267	319
74	204	365
82	193	340
22	228	354
42	151	306
131	210	362
71	211	407
77	174	358
123	227	313
60	207	387
70	229	377
135	197	399
99	168	283
118	217	395
78	254	343
5	146	408
41	194	323
124	260	336
117	145	322
8	167	367
37	242	403
16	234	397
25	142	356
31	140	311
109	180	385
35	236	393
101	216	307
133	263	346
56	143	363
132	264	402
53	248	291
127	223	401
120	233	398
110	237	391
36	219	312
134	175	279
23	205	288
81	178	318
58	262	301
63	272	404
122	239	396
13	186	333
40	139	361

130	154	273			
6	240	276			
119	188	375			
72	187	315			
11	191	277			
43	255	317			
125	190	341			
121	208	369			
27	213	351			
115	215	342			
104	147	349			
45	170	364			
102	192	353			
65	182	347			
12	166	400			
86	165	280			
14	261	281			
46	150	302			
49	230	335			
91	209	321			
4	185	314			
51	265	360			
33	243	357			
24	196	285			
59	202	338			
94	149	310			
67	177	297			
92	164	282			
87	169	303			
116	258	331			
68	199	324			
76	153	296			
136	169	318	470	592	741
34	145	400	461	553	709
89	259	298	411	627	682
48	258	395	507	636	805
126	186	344	436	631	757
72	166	284	452	639	786
23	242	289	474	648	722
83	179	276	537	663	761
77	171	330	522	666	731
68	156	314	516	560	734
82	158	307	505	614	789
9	226	274	514	595	799
51	142	365	501	611	783
73	159	377	479	566	801
84	249	380	442	574	728
38	252	336	454	603	763
113	251	308	425	659	715
40	138	323	543	562	717
80	222	304	426	588	716
121	234	390	450	623	724
19	225	295	531	548	712
99	150	305	418	587	745
8	207	335	438	638	778
119	162	280	434	554	808
22	217	358	518	575	764
70	245	297	491	622	732
86	181	403	544	602	793
81	230	338	464	625	693
85	221	379	521	652	706
52	270	334	510	672	730
98	268	273	439	621	765
28	260	407	453	674	723

2	184	311	493	556	807
4	211	315	480	677	692
39	144	397	484	576	767
124	187	291	435	656	776
97	176	372	476	559	762
117	233	316	500	571	737
133	231	328	457	655	725
120	206	399	517	583	784
95	175	306	530	565	758
10	193	392	495	545	746
43	215	357	515	670	790
135	164	350	420	569	696
129	255	337	519	640	796
17	152	275	417	572	802
131	199	310	503	664	702
87	271	378	460	624	698
31	238	326	533	573	803
13	172	370	542	591	721
60	244	325	469	610	806
27	160	371	529	634	688
101	212	382	482	651	772
93	183	393	448	613	700
58	218	279	508	675	729
41	213	376	432	594	770
1	257	286	488	643	710
5	241	287	433	615	780
92	223	401	410	600	809
108	209	361	496	582	751
111	216	404	538	607	684
75	208	288	466	581	713
35	236	309	424	577	781
130	239	353	528	570	738
53	204	385	458	617	798
7	167	347	506	593	718
55	147	348	467	557	811
59	227	319	489	597	815
71	248	322	428	668	735
116	262	327	472	547	752
45	196	398	414	558	748
6	197	300	520	612	788
20	220	408	492	568	705
78	143	277	478	609	743
14	229	389	487	658	719
132	180	303	462	578	816
109	246	285	421	628	749
16	228	293	455	629	756
104	178	278	490	551	686
107	170	367	456	642	697
64	141	301	431	669	779
29	200	405	485	616	744
37	190	320	498	680	699
57	191	346	430	599	689
47	177	340	471	585	714
123	146	396	409	650	800
90	264	349	504	605	813
44	195	342	511	579	704
33	265	364	541	646	733
91	224	331	502	662	694
100	153	356	512	619	804
26	148	299	412	633	812
67	182	363	463	561	736
69	237	339	523	608	810
102	154	345	446	567	701
115	263	302	440	550	703

56	185	332	441	667	695
54	189	374	497	586	708
65	202	282	536	589	754
62	192	292	525	626	742
49	140	290	481	632	768
63	272	386	416	645	797
118	243	373	513	661	726
30	269	321	486	564	795
106	250	354	483	654	681
79	205	394	539	606	687
50	157	360	524	641	690
15	210	406	451	552	691
112	235	383	535	584	766
127	254	355	534	637	775
46	151	283	444	630	685
76	198	343	526	644	739
24	163	329	477	563	683
122	203	381	422	676	711
105	165	366	459	678	794
61	256	359	445	665	814
66	253	387	427	649	760
11	155	391	449	590	755
94	139	317	443	618	787
42	214	352	468	660	774
110	173	296	423	596	792
25	240	312	447	604	782
128	201	362	437	580	750
103	194	375	429	620	759
74	219	333	473	679	791
88	247	313	465	657	740
125	188	402	475	653	773
114	232	341	499	673	707
96	149	294	413	671	720
32	174	384	509	635	785
36	161	281	494	546	747
12	267	369	415	598	771
3	261	351	540	601	769
21	168	368	532	647	777
18	266	388	419	549	753
134	137	324	527	555	727
19	260	389	433	569	705
39	228	353	456	650	721
123	168	301	516	592	784
2	207	313	423	590	765
78	264	344	491	628	729
117	236	333	502	647	764
49	208	306	523	621	770
70	267	400	530	578	696
100	203	351	419	586	760
71	178	321	471	575	757
61	265	399	506	573	795
103	159	285	442	651	703
114	146	404	430	623	810
40	144	349	464	626	802
111	244	382	448	656	746
60	183	308	470	546	726
91	143	401	429	655	816
62	193	370	505	599	785
97	145	383	465	582	706
104	259	299	422	634	691
11	175	278	460	601	735
36	191	323	439	654	685
110	137	395	475	591	689
80	164	362	463	641	686

21	232	350	512	629	739
58	204	373	484	619	710
120	199	366	450	668	722
75	141	298	532	648	812
108	142	339	432	615	800
83	243	282	409	616	799
131	194	326	473	627	761
25	170	317	452	661	754
98	184	341	424	622	813
92	167	352	421	563	796
115	158	360	438	611	695
33	139	346	476	612	737
3	270	297	417	675	684
86	240	375	487	567	749
81	156	329	497	566	777
46	252	314	538	610	725
87	230	384	494	597	811
24	165	331	517	585	779
13	249	371	521	560	720
116	238	368	489	642	766
42	272	342	481	676	740
95	231	365	485	574	798
76	256	319	420	631	738
128	173	393	414	636	698
53	172	316	446	638	805
28	258	394	541	551	783
90	162	296	525	674	788
119	166	305	459	624	787
8	269	283	431	550	694
51	150	356	437	559	791
107	160	328	461	580	789
133	226	391	441	607	797
56	152	407	509	645	744
113	241	309	411	596	758
27	206	358	542	643	683
44	229	275	427	555	808
109	268	294	443	672	753
79	174	376	474	581	717
14	215	276	486	553	815
22	246	364	451	637	730
12	171	336	499	568	692
57	218	290	543	579	809
93	163	359	479	646	718
15	200	312	518	633	743
73	271	408	526	587	778
122	227	354	436	556	714
16	153	402	537	630	751
35	237	347	449	659	792
4	161	390	425	606	804
59	202	392	466	552	747
94	250	287	534	584	748
48	155	388	515	669	711
41	219	324	520	609	793
68	266	396	492	644	704
129	247	310	501	589	794
26	210	292	535	673	768
72	224	340	495	680	755
9	195	303	480	549	734
89	248	320	490	614	776
106	196	300	428	658	712
69	216	332	544	604	700
52	235	335	458	635	693
55	185	345	514	562	773
23	217	379	453	548	681

84	192	385	539	608	707
124	205	289	469	653	699
6	214	361	527	666	750
30	181	295	500	670	745
135	220	405	524	576	752
54	177	380	508	545	803
63	187	284	529	679	709
17	147	355	444	667	708
88	138	372	483	677	774
96	223	387	477	665	763
64	262	397	467	639	715
127	253	348	493	571	767
66	179	334	410	558	775
10	242	281	445	595	732
82	188	343	472	572	782
99	245	318	468	602	786
47	254	307	462	625	731
32	148	377	510	618	762
37	197	337	435	603	807
112	233	363	426	662	701
134	212	398	454	598	687
18	213	381	498	557	733
74	182	357	418	577	716
130	209	325	412	617	772
5	157	302	416	554	727
7	190	279	511	570	719
102	211	286	482	664	697
38	222	291	513	613	736
1	180	386	540	600	690
85	225	406	536	583	756
121	239	273	457	561	790
136	234	378	522	632	702
118	255	369	528	649	723
29	198	304	531	564	814
45	140	274	447	660	688
34	186	277	507	640	759
65	201	367	478	620	801
77	176	280	440	663	780
50	189	311	415	657	769
20	221	338	488	565	771
101	251	374	503	593	806
105	169	322	496	671	724
126	257	330	455	594	742
125	151	315	434	605	713
43	261	288	504	652	728
132	263	293	519	678	741
31	154	403	413	547	682
67	149	327	533	588	781
45	223	281	455	645	785
41	200	297	410	651	682
123	263	404	486	557	737
120	189	291	539	597	786
132	167	283	531	556	789
129	209	282	449	660	732
42	259	308	465	668	777
126	159	289	511	573	800
30	150	391	428	546	801
21	216	388	543	552	812
44	137	349	500	595	754
35	251	373	501	667	741
27	184	385	414	624	808
79	155	301	453	601	685
60	235	290	489	616	722
29	161	402	509	639	778

125	203	392	480	589	738
136	154	276	514	596	681
61	254	354	521	574	772
72	207	338	508	592	711
87	262	387	416	641	705
98	217	337	429	564	718
49	211	296	541	669	690
74	156	342	437	662	816
84	220	355	502	562	811
112	201	293	490	555	733
25	173	275	525	582	695
26	269	298	473	583	684
6	138	312	497	640	780
107	264	335	427	636	802
114	242	347	535	609	813
118	270	383	463	630	714
110	143	321	421	568	703
16	258	397	443	594	746
86	260	273	466	643	768
59	188	329	422	590	700
46	205	384	520	572	720
134	271	380	536	619	810
78	208	374	468	646	765
33	272	365	519	674	776
113	160	274	413	673	750
88	229	326	499	635	805
19	199	311	441	679	788
18	197	343	527	584	736
93	179	364	538	571	790
53	186	363	471	655	779
10	249	406	529	563	742
40	232	360	517	648	709
119	233	389	537	653	804
81	151	292	467	670	760
103	231	394	478	611	758
130	183	304	452	644	815
14	256	377	419	661	708
50	163	332	469	626	696
116	247	328	474	656	735
32	237	319	484	606	712
38	225	279	523	598	713
100	157	357	461	649	734
39	239	314	494	575	814
76	243	299	516	549	724
23	139	324	487	566	783
91	219	310	492	593	721
111	152	284	459	587	803
56	149	370	485	634	759
115	268	316	456	664	719
95	244	356	493	622	809
82	172	368	481	603	716
2	180	408	524	581	744
92	162	317	436	567	791
24	140	381	448	628	794
99	240	399	454	623	756
11	142	340	439	553	694
90	176	400	451	586	689
97	266	350	491	638	769
48	168	390	435	663	798
1	253	382	447	602	739
3	146	322	530	599	795
117	165	327	442	578	691
128	214	307	424	642	793
65	147	361	472	604	688

52	171	313	488	585	797
55	177	318	513	605	745
133	195	358	534	591	693
102	250	341	433	672	764
54	218	305	430	637	807
57	248	278	434	559	749
47	190	323	477	631	729
75	196	315	432	647	806
66	245	376	415	629	784
94	193	386	444	621	747
64	181	346	483	610	770
34	252	334	462	570	796
17	267	300	507	659	743
108	224	294	503	613	686
80	221	280	476	561	761
131	228	393	426	577	731
83	210	285	544	560	792
135	191	366	431	650	740
89	222	336	482	612	687
73	187	330	458	633	697
37	236	345	475	652	730
28	213	339	440	565	704
36	241	401	522	675	787
13	261	379	423	618	715
127	215	320	533	676	752
69	206	303	496	617	728
71	148	306	446	666	717
109	230	348	505	551	727
63	166	396	540	632	699
15	198	372	412	665	702
124	170	288	460	547	692
68	202	331	417	607	707
58	234	405	506	654	766
121	246	378	532	658	683
12	257	295	411	615	751
104	227	398	498	671	698
51	255	351	504	657	726
20	175	359	450	554	710
8	212	309	457	580	775
9	265	407	464	600	725
122	204	369	542	608	767
62	194	352	526	614	723
70	178	344	518	625	755
77	144	277	495	588	782
5	226	367	528	627	763
106	238	325	420	680	774
105	145	286	425	550	753
85	185	371	510	620	799
67	174	403	470	678	773
31	164	353	409	558	771
101	182	302	445	576	762
7	192	395	515	545	781
4	158	362	512	569	701
43	169	287	418	579	706
22	153	333	438	548	748
96	141	375	479	677	757

Appendix D

Tables

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0	1.50E-01	1072	7168	1024
0.5	1.34E-01	1097	8192	1024
1	1.19E-01	1099	9216	1024
1.5	1.03E-01	1058	10240	1024
2	7.59E-02	1011	13312	1024
2.5	3.70E-02	1024	27648	1024
3	1.21E-02	1005	82944	1024
3.5	2.74E-03	1002	365568	1024
4	5.39E-04	1001	1857536	1024

Table 3 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 1024. Iteration = 1.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0	1.56E-01	1117	7168	1024
0.5	1.35E-01	1106	8192	1024
1	1.14E-01	1047	9216	1024
1.5	1.00E-01	1025	10240	1024
2	6.14E-02	1006	16384	1024
2.5	1.49E-02	1007	67584	1024
3	6.63E-04	1002	1512448	1024
3.5	4.96E-05	1001	20193280	1024

Table 4 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 1024. Iteration = 2.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0	1.49E-01	1070	7168	1024
0.5	1.40E-01	1004	7168	1024
1	1.26E-01	1029	8192	1024
1.5	1.00E-01	1028	10240	1024
2	5.24E-02	1020	19456	1024
2.5	4.89E-03	1041	212992	1024
3	5.70E-05	1001	17569792	1024

Table 5 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 1024. Iteration = 3.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0	1.5513393e-01	1112	7168	1024
0.5	1.3992746e-01	1003	7168	1024
1	1.1718750e-01	1080	9216	1024
1.5	9.7034801e-02	1093	11264	1024
2	4.8925781e-02	1002	20480	1024
2.5	6.3006004e-04	1001	1636352	1024
3	3.1897770e-05	1001	31381504	1024

Table 6 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 1024. Iteration = 6.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0	1.66E-01	1020	6144	1024
0.5	1.42E-01	1021	7168	1024
1	1.16E-01	1071	9216	1024
1.5	1.02E-01	1040	10240	1024
2	4.75E-02	1071	22528	1024
2.5	3.13E-04	1001	3203072	1024
3	3.17E-05	1001	31611904	1024

Table 7 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 1024. Iteration = 10.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0	1.43E-01	1004	7040	128
0.5	1.27E-01	1010	7936	128
1	1.14E-01	1010	8832	128
1.5	8.55E-02	1007	11776	128
2	4.19E-02	1009	24064	128
2.5	8.32E-03	1007	121088	128
3	6.72E-04	1001	1488896	128
3.5	7.21E-05	1001	13878656	128
4	2.26E-05	1001	44232832	128

Table 8 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 128. Iteration = 6.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0	1.51E-01	1007	6656	256
0.5	1.34E-01	1032	7680	256
1	1.21E-01	1022	8448	256
1.5	8.89E-02	1001	11264	256
2	4.78E-02	1003	20992	256
2.5	4.26E-03	1004	235520	256
3	1.30E-04	1001	7689216	256
3.5	2.90E-05	1001	34548736	256
4	1.12E-05	1001	89264896	256

Table 9 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 256. Iteration = 6.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0	1.53E-01	1257	8224	2056
0.5	1.38E-01	1132	8224	2056
1	1.28E-01	1054	8224	2056
1.5	9.72E-02	1199	12336	2056
2	6.66E-02	1096	16448	2056
2.5	1.58E-04	1001	6342760	2056
3	1.60E-05	1002	62710056	2056

Table 10 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 2056. Iteration = 6.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0	1.49E-01	2448	16384	16384
0.5	1.40E-01	2288	16384	16384
1	1.26E-01	2071	16384	16384
1.5	9.89E-02	1620	16384	16384
2	6.04E-02	1980	32768	16384
2.5	6.10E-05	1001	16416768	16384
3	5.39E-06	1001	1.86E+08	16384

Table 11 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 16384. Iteration = 6.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.00	1.62E-01	331	2048	2048
0.50	1.39E-01	285	2048	2048
1.00	1.13E-01	232	2048	2048
1.50	1.01E-01	207	2048	2048
2.00	6.74E-02	138	2048	2048
2.50	2.48E-04	101	407552	2048
3.00	2.52E-05	101	4003840	2048
3.50	4.67E-06	101	21606400	2048
4.00	1.40E-06	102	72859648	2048
4.50	1.38E-06	102	74006272	2048

Table 12 Performance of a rate-1/2 parallel concatenated (turbo) code with memory-4 rate-1/2 RSC codes, generators (37, 21). Block interleaver size = 2048. Iteration = 6.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.91E-01	1002	5240	20
1	1.75E-01	1003	5740	20
1.5	1.58E-01	1001	6320	20
2	1.32E-01	1001	7600	20
2.5	1.21E-01	1003	8280	20
3	9.89E-02	1001	10120	20
3.5	8.52E-02	1004	11780	20
4	6.99E-02	1001	14320	20
4.5	5.37E-02	1001	18640	20
5	4.28E-02	1002	23400	20
5.5	3.15E-02	1002	31840	20
6	2.14E-02	1001	46820	20
6.5	1.25E-02	1001	80320	20
7	8.54E-03	1001	117240	20
7.5	4.54E-03	1001	220380	20
8	2.57E-03	1001	389300	20
8.5	1.28E-03	1001	784380	20
9	5.26E-04	1001	1903320	20
9.5	2.04E-04	1001	4899480	20
10	7.48E-05	1001	13383100	20

Table 13 Performance of Gallager's (3, 4, 20) code for iteration 1.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.74E-01	1003	5760	20
1	1.63E-01	1005	6180	20
1.5	1.38E-01	1001	7240	20
2	1.23E-01	1001	8120	20
2.5	9.42E-02	1006	10680	20
3	8.18E-02	1005	12280	20
3.5	6.00E-02	1002	16700	20
4	4.56E-02	1001	21940	20
4.5	3.64E-02	1001	27520	20
5	2.24E-02	1005	44800	20
5.5	1.42E-02	1001	70260	20
6	8.74E-03	1003	114760	20
6.5	5.08E-03	1001	197100	20
7	2.66E-03	1005	377860	20
7.5	1.30E-03	1001	771300	20
8	4.98E-04	1003	2012060	20
8.5	1.96E-04	1001	5108840	20
9	6.89E-05	1001	14532100	20
9.5	2.10E-05	1001	47589000	20
10	5.30E-06	1001	189044260	20

Table 14 Performance of Gallager's (3, 4, 20) code for iteration 2.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.66E-01	1002	6040	20
1	1.56E-01	1002	6420	20
1.5	1.33E-01	1003	7520	20
2	1.17E-01	1002	8580	20
2.5	8.58E-02	1001	11660	20
3	7.27E-02	1001	13760	20
3.5	5.30E-02	1002	18920	20
4	4.07E-02	1002	24640	20
4.5	3.09E-02	1001	32400	20
5	1.89E-02	1004	53220	20
5.5	1.03E-02	1002	97020	20
6	6.87E-03	1001	145640	20
6.5	3.44E-03	1002	291200	20
7	1.79E-03	1001	560540	20
7.5	7.51E-04	1001	1332640	20
8	2.76E-04	1003	3633140	20
8.5	1.03E-04	1001	9758640	20
9	2.93E-05	1003	34234060	20
9.5	7.24E-06	1003	138451040	20

Table 15 Performance of Gallager's (3, 4, 20) code for iteration 3.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.57E-01	1005	6420	20
1	1.52E-01	1003	6620	20
1.5	1.21E-01	1003	8280	20
2	1.01E-01	1002	9940	20
2.5	7.86E-02	1004	12780	20
3	5.97E-02	1001	16760	20
3.5	4.64E-02	1003	21620	20
4	3.66E-02	1001	27320	20
4.5	2.33E-02	1006	43200	20
5	1.41E-02	1003	71240	20
5.5	8.10E-03	1004	123900	20
6	4.65E-03	1002	215520	20
6.5	2.52E-03	1005	399160	20
7	1.15E-03	1005	875100	20
7.5	3.91E-04	1005	2569680	20
8	1.61E-04	1002	6227880	20
8.5	4.99E-05	1003	20104420	20
9	1.49E-05	1002	67291220	20
9.5	3.37E-06	1002	297664340	20

Table 16 Performance of Gallager's (3, 4, 20) code for iteration 6.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.21E-01	302	2496	96
1	1.08E-01	310	2880	96
1.5	8.59E-02	305	3552	96
2	7.49E-02	302	4032	96
2.5	5.52E-02	302	5472	96
3	4.15E-02	303	7296	96
3.5	2.71E-02	302	11136	96
4	1.80E-02	301	16704	96
4.5	1.05E-02	302	28704	96
5	7.13E-03	301	42240	96
5.5	3.53E-03	301	85344	96
6	1.49E-03	301	201408	96
6.5	6.09E-04	302	496128	96
7	2.32E-04	301	1297920	96
7.5	6.94E-05	301	4339776	96
8	1.86E-05	301	16203840	96
8.5	5.07E-06	301	59365344	96

Table 17 Performance of Gallager's (3, 6, 96) code for iteration 1.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.12E-01	311	2784	96
1	1.00E-01	308	3072	96
1.5	7.71E-02	311	4032	96
2	5.63E-02	308	5472	96
2.5	3.50E-02	302	8640	96
3	2.17E-02	304	14016	96
3.5	1.22E-02	303	24768	96
4	6.44E-03	301	46752	96
4.5	2.75E-03	302	110016	96
5	8.56E-04	301	351840	96
5.5	3.22E-04	301	935520	96
6	9.68E-05	301	3108096	96
6.5	2.00E-05	301	15073920	96
7	5.07E-06	302	59564832	96

Table 18 Performance of Gallager's (3, 6, 96) code for iteration 2.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.07E-01	307	2880	96
1	9.53E-02	311	3264	96
1.5	6.82E-02	301	4416	96
2	4.19E-02	302	7200	96
2.5	3.04E-02	301	9888	96
3	1.35E-02	303	22464	96
3.5	6.40E-03	308	48096	96
4	2.95E-03	301	102144	96
4.5	7.94E-04	301	379296	96
5	2.73E-04	304	1115328	96
5.5	6.55E-05	301	4592544	96
6	1.49E-05	301	20140320	96
6.5	2.81E-06	301	107117280	96

Table 19 Performance of Gallager's (3, 6, 96) code for iteration 3.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.04E-01	318	3072	96
1	8.44E-02	316	3744	96
1.5	5.47E-02	315	5760	96
2	3.11E-02	305	9792	96
2.5	1.69E-02	307	18144	96
3	6.59E-03	305	46272	96
3.5	4.64E-03	303	65280	96
4	8.65E-04	306	353952	96
4.5	2.54E-04	302	1188960	96
5	8.33E-05	306	3672480	96
5.5	1.69E-05	303	17885760	96
6	3.42E-06	301	87926592	96
6.5	5.95E-07	302	507929376	96

Table 20 Performance of Gallager's (3, 6, 96) code for iteration 6.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.27E-01	415	3264	816
1	1.05E-01	344	3264	816
1.5	8.87E-02	362	4080	816
2	7.60E-02	310	4080	816
2.5	5.46E-02	312	5712	816
3	4.48E-02	329	7344	816
3.5	2.66E-02	304	11424	816
4	1.96E-02	320	16320	816
4.5	1.04E-02	305	29376	816
5	6.51E-03	303	46512	816
5.5	3.30E-03	302	91392	816
6	1.59E-03	303	190944	816
6.5	6.16E-04	301	488784	816
7	2.20E-04	301	1365984	816
7.5	8.28E-05	301	3635280	816
8	2.08E-05	301	14474208	816
8.5	4.69E-06	301	64220832	816

Table 21 Performance of Gallager's (3, 6, 816) code for iteration 1.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.22E-01	399	3264	816
1	9.31E-02	304	3264	816
1.5	7.31E-02	358	4896	816
2	5.29E-02	302	5712	816
2.5	3.40E-02	305	8976	816
3	2.04E-02	316	15504	816
3.5	1.05E-02	307	29376	816
4	5.27E-03	301	57120	816
4.5	1.86E-03	302	162384	816
5	5.03E-04	301	598944	816
5.5	1.38E-04	302	2192592	816
6	2.64E-05	301	11422368	816
6.5	4.05E-06	301	74281296	816

Table 22 Performance of Gallager's (3, 6, 816) code for iteration 2.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.5	1.16E-01	379	3264	816
1	8.60E-02	351	4080	816
1.5	5.81E-02	332	5712	816
2	3.82E-02	312	8160	816
2.5	2.50E-02	306	12240	816
3	9.00E-03	301	33456	816
3.5	4.26E-03	306	71808	816
4	7.85E-04	301	383520	816
4.5	1.31E-04	301	2305200	816
5	2.26E-05	302	13378320	816
5.5	1.92E-06	301	156603456	816

Table 23 Performance of Gallager's (3, 6, 816) code for iteration 3.

Eb/No(dB)	Bit-error rate (BER)	Total error	Total block size	Interleaver size
0.50	8.58E-02	140	1632	816
1.00	9.19E-02	150	1632	816
1.50	4.58E-02	112	2448	816
2.00	4.82E-02	118	2448	816
2.50	4.21E-03	103	24480	816
3.00	7.52E-04	108	143616	816
3.50	2.72E-05	101	3711984	816
4.00	1.32E-06	101	76603632	816
4.50	5.62E-08	101	1797770400	816

Table 24 Performance of Gallager's (3, 6, 816) code for iteration 6.

References:

- [1] Acikel, O and Ryan, W E, 'Punctured turbo codes for BPSK/QPSK channels', *IEEE trans. Commun.*
- [2] Bahl, L R, Cocke, F, Jelinek, F and Raviv, J 1974, 'Optimal Decoding of Linear Codes for Minimizing Symbol Error rate', *IEEE Trans. Info. Theory.*, vol. IT-20, March, pp. 284-287.
- [3] Berlekamp, E R, Peile, R E and Pope, S P 1987, 'The application of error control to communications', *IEEE Commun. Magazine*, vol. 25, March, pp. 44-57.
- [4] Berrou, C, Glavieux, A & Thitimajshima, P 1993, 'Near Shannon limit Error-Correcting Coding and Decoding: Turbo-Codes', *Proc.1993 IEEE Int. Conf. Comm. (ICC'93)*, Geneva, Switzerland, May, pp. 1064-1070.
- [5] Berruto, E, Gudmundson, M, Menolascino, R, Mohr, W and Pizarosso, M 1998, 'Research activities on UMTS radio interface, network architectures, and planning', *IEEE Commun. Magazine*, vol. 36, February, pp. 85-95.
- [6] Brodsky, I 1995, 'Wireless: The Revolution in Personal Telecommunications', *Boston, MA: Artech House Publishers*.
- [7] Budde, P 2002, *Wireless Technology – Paging*, Viewed 5th August 2006, <<http://www22.verizon.com/about/community/learningcenter/articles/printerfriendly1/0%2C4066%2C1156%2C00.html>>.
- [8] Costello, D J, Hagenauer, J, Imai, H and Wicker, S B 1998, 'Applications of error control coding', *IEEE Trans. Inform. Theory*, vol. 44, October, pp. 2531-2560.
- [9] *Cordless telephone*, Viewed 4th August 2006, <http://en.wikipedia.org/wiki/Cordless_telephone>.
- [10] Davidson, P & Griffin, R W 2003, *Management an Australian Perspective*, 2nd edition, Kyodo printing co, Singapore.
- [11] Divsalar, D Jin, H and McEliece, R J 1998, 'Coding Theorems for 'Turbo-like' Codes', *Proc. 1998 Allerton Conf. on Commun., Control, and Computing*, Univ. Illinois; Urbana-Champaign, pp. 201-210.
- [12] Elias, P 1954, 'Error-Free Coding', *IRE Trans.*, vol. PGIT-4, pp. 29-37.

- [13] Elias, P 1955, 'Coding for noisy channels', *IRE Conv. Record*, vol. 4, pp. 37-47.
- [14] Farely, T and Hoek, M V D 2006, *Cellular Telephones Basics*, Viewed 4th August 2006, <http://www.privateline.com/mt_cellbasics/>.
- [15] Gallager, R G 1962, 'Low-Density Parity-Check Codes', *IRE Trans. Info. Theory*, vol. 8, no. 1, Jan, pp. 21-28.
- [16] Golay, M J E 1949, 'Notes of digital coding', *Proc. IEEE*, vol. 37, p. 657.
- [17] Goodman, D J 1997, *Wireless Personal Communication Systems, MA: Addison Wesley*.
- [18] Hagenauer, J 1988, 'Rate compatible punctured convolutional codes (RCPC-codes) and their application', *IEEE Trans. Commun.*, vol. 36, September, pp. 389-400.
- [19] Hamming, R W 1950, 'Error Detecting and Error Correcting Codes', *Bell Syst. Tech. J.*, 29, pp. 147-160.
- [20] Kucar, A D 1991, 'Mobile radio: An overview', *IEEE Commun. Magazine*, November, pp. 72-85.
- [21] Lee, L N 1977, 'Concatenated Coding Systems Employing a Unit-Memory Convolutional Code and a Byte-Oriented Decoding Algorithm', *IEEE Trans. Commun.*, COM-25, October, pp. 1064-1074.
- [22] Lee, W C Y 1991, 'Smaller cells for greater performance', *IEEE Commun. Magazine*, November, pp. 19-23.
- [23] Lin, S & Costello, D J 2004, *Error Control Coding*, 2nd edition, Person Prentice Hall, New Jersey.
- [24] Luby, M G, Miteznmacher, Shokrollahi, M A and Spielman, D A 2001, 'Improved Low-Density Parity-Check Codes Using Irregular Graphs', *IEEE Trans. Information Theory*, vol. 47, no. 2, February, pp. 585-598.
- [25] MacKay, D J C, *Encyclopedia of Sparse Graph Codes*, Viewed 5th Spetember 2006, < <http://www.inference.phy.cam.ac.uk/mackay/codes/data.html> >
- [26] MacKay, D J C & Neal, R M 1996, 'Near Shannon Limit Performance of Low Density Parity Check Codes', *Electronic Letters*, vol. 32, pp. 1645-1646.
- [27] MacKay, D J C 1999, 'Good Error-Correcting Codes Based on Very Sparse Matrices', *IEEE Trans. Info. Theory*, vol. 45, no. 2, March, pp. 399-432.
- [28] Massey J L 1963, 'Threshold Decoding', *MIT Press*.

- [29] Miller, B 1998, 'Satellites free the mobile phone', *IEEE Spectrum*, vol. 35, March, pp. 26-35.
- [30] Mitsishi, T 1989, 'Automobile and portable telephones in Japan', *NTT review*, vol. 1, May, pp. 30-39.
- [31] Morelos-Zaragoza, R H 2002, *The Art of Error Correcting Coding*, John Wiley & Sons, England.
- [32] Moon, T K 2005, *Error correction coding : mathematical methods and algorithms*, Wiley-Interscience, Hoboken, N.J..
- [33] Odenwalder, J P 1976, 'Error Control Coding Handbook', *Linkabit Corporation*.
- [34] Padgett, J E, Gunther, C G and Hattori, T 1995, 'Overview of wireless personal communications', *IEEE Commun. Magazine*, vol. 33, January, pp. 28-41.
- [35] Pahlavan, K, Probert, T H and Chase, M E 1995, 'Trends in local wireless networks', *IEEE Commun. Magazine*, vol. 33, March, pp. 85-95.
- [36] Proakis, J 1995, *Digital Communications*, 3rd edition, New York, NY: McGraw-Hill, Inc..
- [37] Ramesy, J L 1970, 'Realization of optimum interleavers', *IEEE Trans. Inform. Theory*, vol. 16, May, pp. 338-345.
- [38] Rappaport, T S 1996, 'Wireless Communications: Principles and Practice', *Upper Saddle River, NJ: Prentice Hall PTR*.
- [39] Richardson T J and Urbanke R L 2001, 'Efficient encoding of low-density parity-check codes', *IEEE Trans. InformationTheory*, vol. 47, no. 2, February, pp. 638-656.
- [40] Richardson, T J, ShokroLLahi, M A & Urbanke, R L 2001, 'Design of Capacity Approaching Irregular Low-Density Parity-Check Codes', *IEEE Trans. Info. Theory*, vol. 47, no. 2, February, pp. 619-637.
- [41] Robertson, P, Villeburn, E and Hoeher, P 1995, 'A Comparison of Optimal and Sub-Optimal MAP Decoding Algorithms Operating in the Log Domain', *Proc. 1995 IEEE Int. Conf. Comm.*, pp. 1009-1013.
- [42] *Satellite phone*, Viewed 4th August 2006, <http://en.wikipedia.org/wiki/Satellite_telephone>.

-
- [43] Shannon, C E 1948, 'A mathematical theory of communication', *Bell Sys. Tech. J.*, vol. 27, pp. 379-423 and 623-656.
- [44] Shokrollahi, A 2003, 'LDPC Codes: An Introduction', *Digital Fountain, Inc*, pp.2-3.
- [45] Tanner, R M 1981, 'A Recursive Approach to Low Complexity Codes', *IEEE Trans. Info. Theory*, vol. IT-27, no. 5, September, pp. 533-547.
- [46] Vucetic, B and Yuan, J 2000, *Turbo Codes: Principles and Applications*, Kluwer Academic.
- [47] Wicker, S 1995, *Error Control Systems for Digital Communications and Storage*, Englewood Cliffs, NJ: Prentice Hall, Inc..
- [48] Wu, Y 1999, 'Implementation of parallel and serial concatenated codes', *prelim report*, Virginia Tech, Blacksburg, VA.
- [49] Zyablov, V V & Pinsker, M S 1975, 'Estimation of the Error-Correction Complexity for Gallegor Low-Density Codes', *Prob. Pered. Inform.*, vol. 11, no. 1, pp. 26-36.