UNIVERSITY OF SOUTHERN QUEENSLAND

FACULTY OF ENGINEERING AND SURVEYING

PERFORMANCE EVALUATION OF LOW-DENSITY PARITY-CHECK CODES

A dissertation submitted by

Paul Lauder

in fulfillment of the requirements of

Courses ENG4111 and 4112 Research Project

towards the degree of

Bachelor of Engineering (Software)

Abstract

LDPC codes were first introduced by Robert Gallager in 1960. Due to the complexity of the codes and the limitations of the then rudimentary computer resources the codes were neglected as a viable form of FEC. LDPC codes were rediscovered by Tanner in 1981 when he generalized the codes and provided a means of graphical representation of LDPC codes. LDPC codes were again neglected until the work of MacKay et al in the mid to late 1990's resurrected interest in the codes when they were discovered to out perform the then premium Turbo codes.

This dissertation specifically describes the process of encoding and decoding LDPC codes and demonstrates the performance comparison between the various types of decoders in terms of bit error rate performance factors.

University of Southern Queensland
Faculty of Engineering and Surveying

ENG411 and 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.

Dean
Faculty of Engineering and Surveying

Certification

I certify that the ideas, designs and experimental work, results, 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.

Paul James Lauder
Student Number: 0050014041

_____
Signature

_____
Date

Acknowledgements

I would like to thank my supervisor, Dr Wei Xiang from the Faculty of Engineering and Surveying, for his guidance and support throughout the duration of my project.

I would also like to express a special thank you to my wife Vivienne for her patience and understanding, especially in the final stages of the project.

Paul Lauder

Table of Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

Chapter 1 Introduction

1.1 Overview

The theories and applications of error correcting codes (ECC) originate from the weight and power limitations imposed on the early space vehicles. High power transmitters were not able to be incorporated into the craft and as such the transmissions home would be greatly affected by noise and attenuation. Therefore a method had to be devised to allow the efficient and effective correction of data errors induced into the signals. To counter this unwanted situation, error correcting codes incorporated redundancy into the original data that then enabled the receiver to find and correct bit errors occurring in the transmission.

In modern computer communications there has been a drive by both individual users and especially commercial users to transmit and store increased levels of data. To facilitate this desire new techniques have been employed to increase the data density via compression. Applications such as High Definition television, mobile cellular telephony, satellite communications and Digital Versatile Disks all utilize some form of ECC to enable a high level of data accuracy in order to allow the recreation of the original data at the receiver.

Iterative decoding is one of the most powerful techniques employed in modern ECC algorithms (Dechter et al, 2003). Low-density parity-check (LDPC) codes and Turbo codes are two ECCs based on iterative decoding, this technique will be detailed in Chapter 2. LDPC codes have recently been developed that allow data transmission rates close to the

Shannon Limit (Berou et al, 1993), the theoretical upper limit of the ratio of information to redundancy required for accurate transmission over a particular noisy channel.

1.2 Wireless communications

Wireless communications were born in February 1896 when Guglielmo Marconi journeyed from Italy to England in order to demonstrate to the British telegraph authorities what he had developed in the way of an operational wireless telegraph apparatus. The first wireless systems allowed communications at distances of only a mile or two. The period before the Great War saw small advances in the new technology that allowed for ever greater distances to be covered. Throughout the next century, great strides were made in wireless communication technology.

Table 1: A timeline of significant achievements in wireless communications.

1896	Marconi develops the first wireless telegraph system.
1927	First commercial radiotelephone service operated between Britain and the US.
1946	First automotive mobile telephone set up in St. Louis using push-to-talk technology.
1948	Shannon publishes his seminal paper *A Mathematical Theory of Communication*, establishing the foundations for source and channel encoding.
1950	First terrestrial microwave telecommunications system (TD-2) commences.
1962	The first telecommunication satellite, Telstar, launched into orbit.
1968	Defense Advanced Research Projects Agency – US (DARPA) selected BBN to develop the Advanced Research Projects Agency Network (ARPANET), the foundation of today's Internet.
1977	Mobile cellular network established by Bell Labs with the Advanced Mobile Phone System (AMPS).
1983	Transmission Control Protocol/Internet Protocol (TCP/IP) designated the official protocol for ARPANET.
1992	One millionth host connected to the internet.
1998	Bluetooth wireless data communications developed through collaboration of major electronics manufacturers.

(Dubendorf, 2003)

1.3 Objective of the Project

The objective of this project is to determine the optimal LDPC encoder/decoder combination for any given signal to noise ratio over the Additive Gaussian White Noise (AGWN) channel, in terms of Bit Error Rate (BER) and algorithm efficiency. The project will also determine the applicability of using LDPC codes in wireless systems.

1.4 Structure of the Dissertation

This dissertation is structured in the following way:
Chapter 1 - Introduction. The first chapter introduces the reader to wireless communications and provides the reasoning behind the dissertation.

Chapter 2 - Error Correcting Coding. A detailed overview of the different methods employed in the field of error correcting coding, including block codes, convolutional codes, concatenated codes and turbo codes.

Chapter 3 – LDPC Codes. A detailed analysis of low-density parity-check code encoding and decoding techniques.

Chapter 4 - Performance comparison of LDPC encoders and decoders. Explanation of LDPC simulator software, determination of optimal encoder/decoder combination.

Chapter 5 – Conclusion. Concludes the dissertation and provides recommendations for future work.

Chapter 2

Fundamentals of Error Correcting Coding

Chapter 2 Fundamentals of Error Correcting Coding

2.1 History

In the late 1940s Claude Shannon, a research mathematician at Bell Telephone Laboratories, invented a mathematical theory of communication that gave the first systematic framework in which to optimally design telephone systems (Shannon, 1948). The main questions motivating this were how to design telephone systems to carry the maximum amount of information and how to correct for distortions on the lines.

2.2 Shannon Capacity Limit

Shannon’s ground-breaking approach to telecommunications introduced a simple abstraction of human communication, called the channel. Shannon's communication channel consisted of a sender (the source of information), a transmission medium (with noise and distortion), and a receiver (whose objective is to reconstruct the original message).

In order to quantitatively analyse transmission through the channel he also introduced a measure of the amount of information in a message. For Shannon a message is very

informative if the chance of its occurrence is small. If, in contrast, a message is very predictable, then it contains a small amount of information.

To complete his quantitative analysis of the communication channel, Shannon introduced the concept of entropy, a quantity that measures a source's information production rate and also a measure of the information carrying capacity, called the communication channel capacity.

He showed that if the entropy rate, the amount of information you wish to transmit, exceeds the channel capacity, then there were unavoidable and uncorrectable errors in the transmission. What was truly surprising, however, is that he also showed that if the sender's entropy rate is below the channel capacity, then there is a way to encode the information so that it can be received without errors. This is true even if the channel distorts the message during transmission.

The channel capacity for the AWGN channel is determined by the famous formula developed by Shannon:

$$C = W \log 2(1 + P / N) \tag{2.1}$$

where:

C is capacity (bits per second)

W is the bandwidth (Hertz)

P is transmitter power (Watts)

N is the noise (Watts)

The channel bandwidth establishes a constraint of how fast symbols can be transmitted over the channel. The signal to noise ratio (P/N) determines how much information each symbol can represent. The signal and noise power strengths are calculated at the receiver end of the channel. Thus, the power level is both a function of transmitted power and that of the attenuation of the signal over the transmission channel.

2.3 Basic Concepts Of Error Correcting Coding

Error correcting coding (ECC) (also known as error control coding) is a type of digital signal processing that improves data reliability by introducing a known redundancy into a data sequence prior to transmission or storage. This redundancy enables a receiving system to detect and correct errors caused by corruption through the communication channel. As the name implies, this coding technique enables the decoder to correct errors without requesting retransmission of the original information.

A system employing ECC consists of an information source that produces a digital message sequence u which is k digits in length. The message is passed to a channel encoder that adds the redundant bits, converting the original message into a codeword v of length n digits. Figure 2.1 shows a representation of the structure of a codeword.

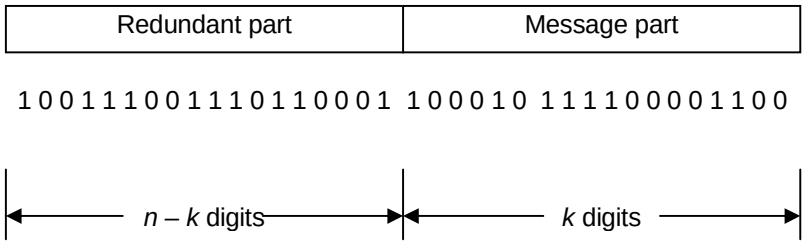

Figure 2.1 Systematic format of a codeword.

The codeword is transmitted through the coding channel by a modulator. Coding channels are grouped into two classes being transmission types or storage types. Typical transmission types include landline telephony, microwave links, fibre optic cabling, satellite communications, high-definition television (HDTV), mobile cellular telephony, and so on. Typical storage types include digital versatile discs (DVD), compact discs (CD), magnetic hard drives, and so on. Every coding channel has an effect on the transmitted codeword due to the presence of noise or attenuation of the signal.

A demodulator converts the received signal r into a readable format for the channel decoder that produces the estimated information sequence $\hat{u}$. Ideally $\hat{u}$ will replicate u if noise has not been introduced by the coding channel. The source decoder utilises the redundancy added to the original information to produce an estimate of the message u which is sent to the destination. Figure 2.2 shows a block diagram of a typical transmission system employing ECC.

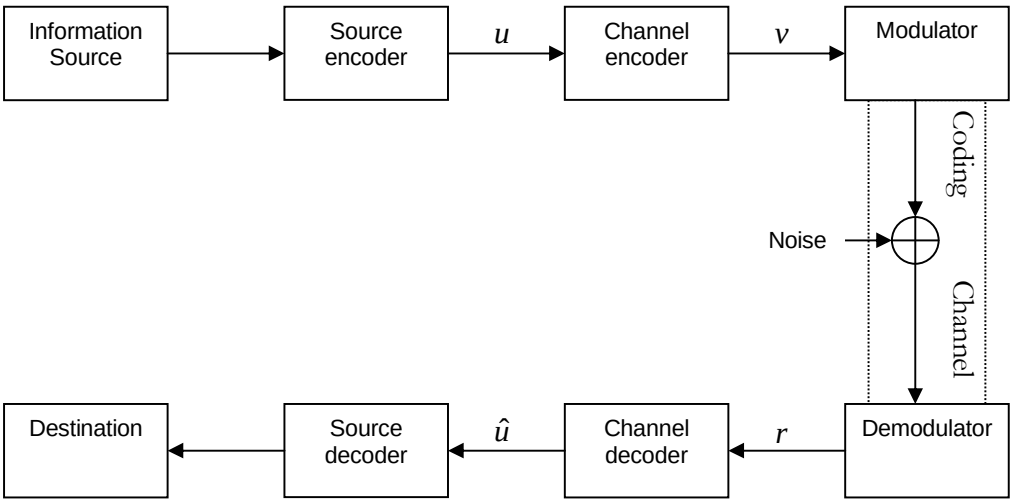

Figure 2.2: Block diagram of a typical data transmission system employing ECC.

2.4 Linear Block Codes

Linear block coding (also known as algebraic coding) was the only type of forward error-correction coding in use when Claude Shannon published his seminal paper *Mathematical Theory of Communication* (Shannon, 1949). With this technique, the encoder intersperses parity bits into the data sequence using particular algebraic algorithms. On the receiving end, the decoder applies an inverse of the algebraic algorithm to identify and correct any errors caused by channel corruption.

In block coding the source information is segmented into messages, represented by u, that have k equal number of information bits. In total there will exist 2^k distinct messages, if the messages are not distinct the code can not be safely implemented without causing ambiguity in determining the original message. The encoder transforms the message u into a n-tuple codeword v, with $n > k$, by applying certain rules. As there are 2^k messages there will also be 2^k distinct codewords having a one-to-one relationship. In order to reduce the complexity of the encoder having to store a large data dictionary when k and n are of a significant length, a special structure is required to represent the set of all codewords. The block code is said to be linear if and only if the resultant modulo-2 sum of any two codewords exists in the set C of all codewords. An example of a linear block code is given in Table 1.

Table 2: (7, 4) Linear block code

Message	**Codeword**
0000	0000000
0001	1010001
0010	1110010
0011	0100011
0100	0110100
0101	1100101
0110	1000110
0111	0010111
1000	1101000
1001	0111001
1010	0011010
1011	1001011
1100	1011100
1101	0001101
1110	0101110
1111	1111111

Because an (n, k) linear block code C is a k-dimensional subspace of the vector space V_n of every binary n-tuple, it is possible to find k linearly independent codewords, g_0,

g_1, K , g_{k-1}, in C such that every codeword v is a linear combination of these k codewords; thus,

$$v = u_0 g_0 + u_1 g_1 + \mathrm{K} + u_{k-1} g_{k-1} \tag{2.2}$$

where $u_1 = \{0,1\}$ for $0 \le i < k$. The k linearly independent codewords can now be arranged into a $k \times n$ matrix in the following form:

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \mathrm{M} \\ g_{k-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \Lambda & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \Lambda & g_{1,n-1} \\ \mathrm{M} & \mathrm{M} & & \mathrm{M} \\ g_{k-1,0} & g_{k-1,1} & \Lambda & g_{k-1,n-1} \end{bmatrix} \tag{2.3}$$

where $g_i = (g_{i,0}, g_{i,1}, \mathrm{K}, g_{i,n-1})$ for $0 \le i < k$. If $u = (u_0, u_1, \mathrm{K}, u_{k-1})$ is the message to be encoded then the resultant codeword can be determined by:

$$\begin{aligned} v &= u \cdot G \\ &= (u_0, u_1, \mathrm{K}, u_{k-1}) \cdot \begin{bmatrix} g_0 \\ g_1 \\ \mathrm{M} \\ g_{k-1} \end{bmatrix} \\ &= u_0 g_0 + u_1 g_1 + \mathrm{K} + u_{k-1} g_{k-1} \end{aligned} \tag{2.4}$$

The matrix G thereby allows the generation of any v codeword from any given message u from the set of codes in C and therefore is called the generator matrix for C.

A linear block code can also be expressed in terms of a parity check matrix where for any $k \times n$ generator matrix with k linearly independent rows, there exists an $(n-k) \times n$ matrix H with $(n-k)$ linearly independent rows such that any vector in the row space of G is

orthogonal to the rows of H, and any vector that is orthogonal to the row space of H is in the row space of G, this provides the following equation:

$$v \cdot H^T = 0 \tag{2.5}$$

The code C is said to be the null space of H. Linear block codes are thereby defined and described in terms of either a generator matrix or a parity check matrix.

2.5 Convolutional Codes

The forward error correction technique known as convolutional coding was first introduced by Elias in his work *Coding for Noisy Channels* (Elias, 1955). Convolutional codes process incoming bits in streams rather than in blocks. The principle feature of such codes is that the encoding of any bit is strongly influenced by the bits that preceded it (that is, the memory of past bits). A convolutional decoder takes into account such memory when trying to estimate the most likely sequence of data that produced the received sequence of code bits. Historically, the first type of convolutional decoding, known as sequential decoding, used a systematic procedure to search for a good estimate of the message sequence; however, such procedures required a great deal of memory, and typically suffered from buffer overflow and nongraceful degradation.

In 1967, Andrew Viterbi developed a decoding technique that has since become the standard for decoding convolutional codes (Viterbi, 1967). At each bit-interval, the Viterbi decoding algorithm compares the actual received code bits with the code bits that might have been generated for each possible memory-state transition. It chooses, based on metrics of similarity, the most likely sequence within a specific time frame. The Viterbi decoding algorithm requires less memory than sequential decoding because unlikely sequences are dismissed early, leaving a relatively small number of candidate sequences that need to be stored.

2.6 Concatenated Codes

In 1966, Forney combined the two previous coding techniques to form a concatenated code (Forney, 1966). In this arrangement, the encoder linked together an algebraic code followed by a convolutional code. The decoder, a mirror image of the encoding operation, consisted of a convolutional decoder followed by an algebraic decoder. Thus, any bursty errors resulting from the convolutional decoder could be effectively corrected by the algebraic decoder. Performance was further enhanced by using an interleaver between the two encoding stages to mitigate any bursts that might be too long for the algebraic decoder to handle. This particular structure demonstrated significant improvement over previous coding systems and is currently being used in the Deep Space Network (Sniffen, 2004) as well as numerous other commercial broadcasting services.

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 (Elias, 1954). After designing the underlying error correcting code, the error performance can be improved by simply increasing the number of iterations. In terms of the application of iterative decoding algorithms, ECC schemes can be generally categorized into two classes, either Turbo codes or Low Density Parity Check (LDPC) codes.

2.7.1 Turbo Codes

Turbo codes are a class of error correcting codes that were first introduced in 1993, by a group of researchers from France, along with a practical decoding algorithm (Berrou *et al*, 1993). 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 used 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.

2.7.2 LDPC Codes

LDPC codes are a form of linear block codes that were first introduced in 1960 by Robert Gallager in his doctoral thesis (Gallager, 1963). Besides turbo codes, low-density parity-check (LDPC) codes form another class of Shannon limit-approaching codes. Due to the complexity of the encoding and decoding of LDPC codes they were not utilised after the time of their discovery for decades. A great deal of research has been conducted recently into LDPC codes with the design of very fast encoding and decoding algorithms. The design of the codes is such that the decoding algorithms have the ability recover the original codeword in the presence of large amounts of noise.

The construction, encoding and decoding of LDPC codes will be presented in detail in the subsequent chapters.

2.8 Summary

In this chapter, we outlined the basic concepts of Error Correcting Codes and the Shannon capacity limit of a communications channel. A basic explanation of different forms of ECC was presented with emphasis on the block coding techniques. LDPC codes have many common concepts of operation as other block coding techniques such as Turbo codes. Both LDPC and Turbo codes utilise iterative soft output decoding algorithms and both can achieve near Shannon limit performance.

Chapter 3

LPDC codes

Chapter 3 LPDC Codes

3.1 Code Description

Gallager defined an LDPC code as the null space of a sparse parity check matrix H (Gallager, 1963). A sparse matrix is one that contains very few 1's when compared to the number of 0's. The original work by Gallager described what is now known as a regular LDPC code, having a parity check matrix H that has constant row weights, denoted by w_r and constant column weights, denoted by w_c. The weighting is determined by the number of 1's contained in the column or row. The values of w_r and w_c are to be minor when compared to the length of the codeword (n) and the number of rows in H (J). Additionally the number of 1's in common between any two columns of H can not be greater than 1. Figure 3.1 demonstrates a regular LDPC code.

If the parity check matrix H contains inconsistent w_r or w_c values it is said to be an irregular LDPC code. In recent times irregular LDPC codes have been demonstrated to achieve remarkable results that were within 0.0045 dB of the Shannon limit (Chung et al, 2001).

$$H = \left[\begin{array}{cccccccccccccccccccc}
1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1\\
\hdashline
1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0\\
0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&1&0&0&0\\
0&0&1&0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0\\
0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1\\
\hdashline
1&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0\\
0&1&0&0&0&0&1&0&0&0&1&0&0&0&0&1&0&0&0&0\\
0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&1&0\\
0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&0&0&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1
\end{array}\right]$$

Figure 3.1: Regular (20, 3, 4) LDPC code.

3.2 Tanner Graphs

In 1981 Robert Tanner rediscovered LDPC codes in his work on the use of recursion to construct error correcting codes (Tanner, 1981). Tanner utilised bipartite graphs to describe the parity check matrix, which are now known as Tanner graphs, which display the incidence relationships between the variable codeword bits and the corresponding check-sum tests.

The graph G representing the parity check matrix H consists of two sets of vertices V and C. The set V consists of n vertices that represent the n codeword bits and are called variable notes, denoted by $v_0, v_1, \mathrm{K}, v_{n-1}$. Variable node indexes correspond to the column number of the parity check matrix. The set C consists of J vertices that represent the J parity check-sums and are called check nodes, denoted by $c_0, c_1, \mathrm{K}, c_{J-1}$. Check node

indexes correspond to the row number of the parity check matrix. An edge is contained in the graph G if and only if the variable node v_n is contained in a parity check sum c_j. The inclusion of a variable node in a check sum is determined by the presence of a 1 in the parity check matrix. The graph G will never have any two variable nodes or any two check nodes connected by an edge. The Tanner graph for the parity check matrix shown in Figure 3.1 is shown in Figure 3.2.

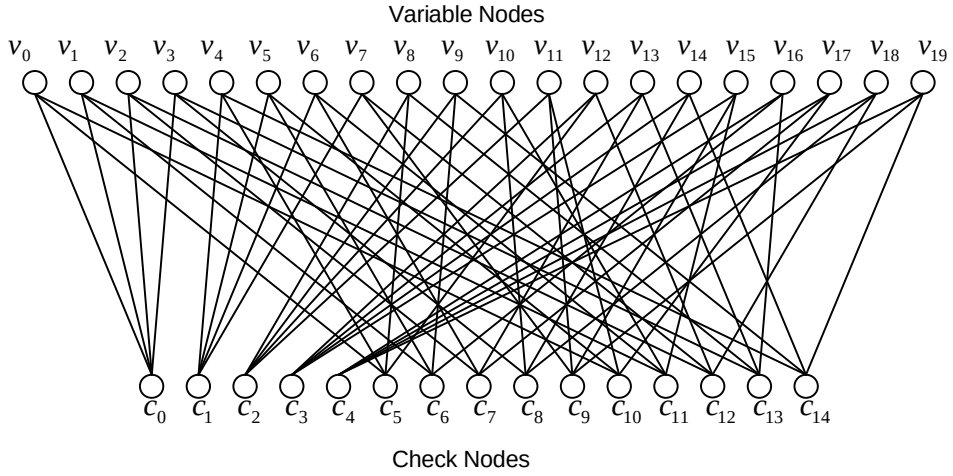

Figure 3.2: Tanner graph for (20, 3, 4) LDPC code

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, this will be examined in section 3.4.3.

3.3 Encoding LDPC codes

There are two methods employed to encode messages into codewords for LDPC codes, both methods require the generation of the parity check matrix H. The algorithms used in the construction of parity check matrix H will be discussed in the following subsections.

The first encoding method is through the use of a generator matrix, denoted by G. The matrix G contains the set of constraints that form the parity check equations of the LDPC code. A codeword c is formed by multiplying a source message u by the matrix G. This is represented by the following equation:

$$c = uG \tag{3. 1}$$

For a binary code with k message bits and length n codewords the generator matrix G is a $(k \times n)$ binary matrix having the form:

$$G = \left[I_k \middle| A^T \right] \tag{3. 2}$$

where the sub-matrix A^T is produced by transforming H into column permuted reduced-row echelon form using Gauss-Jordan elimination (Larson *et al*, 2004) such that:

$$H = \left[A \middle| I_{n-k} \right] \tag{3. 3}$$

where A will be a $(n-k) \times k$ binary matrix and I_{n-k} is an identity matrix. The row space of G will be orthogonal to H so that:

$$GH^T = 0 \tag{3. 4}$$

The process of converting H into the generator matrix G has the effect of causing G to lose the sparseness characteristic that was embodied in H. This has the drawback of driving the encoder complexity towards $\mathrm{O}(n^2)$ (Mackay, 2005).

An alternative approach to encoding LDPC codes was proposed by Richardson and Urbanke based on approximate lower triangulations (Richardson and Urbanke, 2001). This method is also a two step process.

Firstly pre-processing is conducted, using only row and column permutations, the parity-check matrix is put into *approximate lower triangular* form:

$$H_t = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix} \tag{3. 5}$$

such that $-ET^{-1}B + D$ is non-singular.

The g rows of H in sub-matrices C, D, and E is called the gap of the approximate triangular representation. The sub-matrix T is a lower triangular matrix of size $(m-g)\times(m-g)$. If H_t is full rank the sub-matrix B is size $(m-g)\times g$ and sub-matrix A is size $(m-g)\times k$. Keeping the size of the gap low will reduce the encoding complexity for the LDPC code.

Once in lower triangular format, Gauss-Jordan elimination is utilised to clear E, which is the equivalent of multiplying H_t by

$$\begin{bmatrix} I_{m-g} & 0 \\ -ET^{-1} & I_g \end{bmatrix}$$

so that

$$\tilde{H} = \begin{bmatrix} I_{m-g} & 0 \\ -ET^{-1} & I_g \end{bmatrix} \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix} = \begin{bmatrix} A & B & T \\ -ET^{-1}A + C & -ET^{-1}B + D & 0 \end{bmatrix}$$

Finally, to encode the message using $\tilde{H}$ the codeword $c = [c_1, c_2, \mathrm{K}, c_n]$ is divided into three parts being $c = [u, p_1, p_2]$, where $u = [u_1, u_2, \mathrm{K}, u_k]$ is the k -bit message, $p_1 = \lfloor p_{1_1}, p_{1_2}, \mathrm{K}, p_{1_g} \rfloor$ holds the first g parity check bits and $p_2 = \lfloor p_{2_1}, p_{2_2}, \mathrm{K}, p_{2_{m-g}} \rfloor$ contains the remaining parity check bits.

The codeword c must satisfy the syndrome test $c\tilde{H} = 0$ and so

$$Au + Bp_1 + Tp_2 = 0, \tag{3. 6}$$

and

$$\tilde{C}u + \tilde{D}p_1 + 0p_2 = 0 \tag{3. 7}$$

where $\tilde{C} = -ET^{-1}A + C$ and $\tilde{D} = -ET^{-1}B + D$.

Since E has been cleared, the parity bits in p_1 depend only on the message bits, and so can be calculated independently of the parity bits in p_2. If $\tilde{D}$ is invertible, p_1 can be found by $p_1 = \tilde{D}^{-1}\tilde{C}u$.

If $\tilde{D}$ is not invertible the columns of $\tilde{H}$ can be permuted until it is. By keeping g as small as possible the added complexity load of the matrix multiplication to obtain is kept to a minimum.

Once p_1 is known p_2 can be found by $p_2 = -T^{-1}(Au + Bp_1)$. The sparseness of A, B and T can be employed to keep the complexity of this operation low and, as T is upper triangular, p_2 can be found using back substitution.

3.4 Code Construction

3.4.1 Random Generation of Parity Check Matrix

The construction of the parity check matrix can be achieved in a pseudorandom manner using computer searches that are constrained to the characteristics of an LDPC matrix. The parity check matrix H is constructed through the fitting of n columns to an empty matrix,

where n is the length of the LDPC code. Regular and irregular LDPC codes can be constructed using this method.

To generate the matrix H for an LDPC code of length n and rate k/n an appropriate column weight γ and row dimension J must be determined.

3.4.2 Geometric Generation

The generation of LDPC codes can be achieved through the application of finite geometries. A finite geometry is a geometry that has a finite number of points. A geometry G with n points and J lines which has the following fundamental structural properties: 1) every line consists of ρ points; 2) any two points are connected by one and only one line; 3) every point lies on γ lines and 4) two lines are either parallel or they intersect at only one point (Ball & Werner, 2007). The two families of finite geometries which have the above fundamental structural properties are Euclidean and Projective geometries over finite fields. Lin and Costello describe two types of LDPC code constructions for Euclidean and Projective geometries (Lin & Costello, 2004).

3.4.2.1 Euclidian Geometries

A geometry in which Euclid's parallel postulate holds is called Euclidean Geometry. The parallel postulate states that if two lines are drawn which intersect a third in such a way that the sum of the inner angles on one side is less than two right angles, then the two lines inevitably must intersect each other on that side if extended far enough.

Consider $EG(m,2^s)$ as an m-dimensional Euclidean Geometry over the Galois Field $GF(2^s)$ where s and m are two positive integers. Total number of points in this geometry is 2^{ms} with each point representing an m-tuple over $GF(2^s)$. The origin corresponds to the m-tuple that contains all zeros. $EG(m,2^s)$ can be seen as an m-dimensional vector space over $GF(2^s)$. Any two lines in $EG(m,2^s)$ will either intersect at one and only one point or are disjoint. Therefore a line will consist of 2^s points. The total number of lines in $EG(m,2^s)$ can be calculated by:

$$J = \frac{2^{(m-1)s}\left(2^{ms}-1\right)}{2^s-1} \tag{3. 8}$$

Every line will have $2^{(m-1)s}-1$ lines parallel to it. For each point in $EG(m,2^s)$ the number of lines intersecting at this point is:

$$\frac{2^{ms}-1}{2^s-1} \tag{3. 9}$$

a) Type I Euclidean Geometry (EG) LDPC Codes

A type-I EG code that is based on $EG(m,2^S)$ the parity check matrix $H_{EG}^{(1)}$ is formed where each row is the incidence vectors of all the lines $EG(m,2^S)$ and each column corresponds to all the points in $EG(m,2^S)$. The number of rows in $H_{EG}^{(1)}$ can be determined using (3.8) and the number of columns will be $n=2^{ms}$, corresponding to the number of points in the geometry. Each row weight in $H_{EG}^{(1)}$ will equate to the number of points contained in a line in $EG(m,2^S)$ this being 2^s. The number of lines intersecting a point, given by (3.9), will determine the column weight of $H_{EG}^{(1)}$. The density r of $H_{EG}^{(1)}$ is:

$$r=\frac{\rho}{n}=\frac{2^s}{2^{ms}} \tag{3. 10}$$

When $m\geq 2, s\geq 2$ and $r\leq 1/4$ then $H_{EG}^{(1)}$ will be a low density parity check matrix. The null space of $H_{EG}^{(1)}$ provides an LDPC code of length $n=2^{ms}$, denoted by $C_{EG}^{(1)}(m,0,s)$, which is call an m-dimensional type-I $(0,s)$th-order EG-LDPC code. The Tanner graphs of type-I EG codes do not contain any 4-cycles, though there are many cycles of length 6 (Kou, Lin & Fossorier, 2001).

b) Type II Euclidean Geometry LDPC Codes

The construction of a type-II EG LDPC code is carried out by taking the transpose of $H_{EG}^{(1)}$ so that:

$$H_{EG}^{(2)}=\left[H_{EG}^{(1)}\right]^T \tag{3. 11}$$

Then $H_{EG}^{(2)}$ has both the values of J and n, along with ρ and γ interchanged when compared to $H_{EG}^{(1)}$. Then null space of $H_{EG}^{(2)}$ gives an LDPC code, denoted by $C_{EG}^{(2)}(m,0,s)$, having length $n = 2^{(m-1)s}\left(2^{ms}-1\right)/\left(2^{s}-1\right)$. The minimum distance will be of the order of $2^{s}+1$. The Tanner graphs for $C_{EG}^{(1)}(m,0,s)$ and $C_{EG}^{(2)}(m,0,s)$ are dual, both the codes have identical cycle distributions. The $C_{EG}^{(2)}(m,0,s)$ type-II EG LDPC code is not cyclic, but it can be put into a quasi cyclic form (Kou, Lin & Fossorier, 2001).

Figure 3.3 shows a type-I (0,2)th-order EG LDPC code parity check matrix with its associated Tanner graph. Figure 3.4 illustrates a type-II (0,2)th-order EG LDPC code that is the transpose of the matrix in Figure 3.3. It can be seen that the Tanner graph of the type-II matrix is a mirror image of the Tanner graph of the type-I matrix.

$$H_{EG}^{(1)} = \begin{bmatrix}
0&0&0&0&0&0&0&1&1&0&1&0&0&0&1\\
1&0&0&0&0&0&0&0&1&1&0&1&0&0&0\\
0&1&0&0&0&0&0&0&0&1&1&0&1&0&0\\
0&0&1&0&0&0&0&0&0&0&1&1&0&1&0\\
0&0&0&1&0&0&0&0&0&0&0&1&1&0&1\\
1&0&0&0&1&0&0&0&0&0&0&0&1&1&0\\
0&1&0&0&0&1&0&0&0&0&0&0&0&1&1\\
1&0&1&0&0&0&1&0&0&0&0&0&0&0&1\\
1&1&0&1&0&0&0&1&0&0&0&0&0&0&0\\
0&1&1&0&1&0&0&0&1&0&0&0&0&0&0\\
0&0&1&1&0&1&0&0&0&1&0&0&0&0&0\\
0&0&0&1&1&0&1&0&0&0&1&0&0&0&0\\
0&0&0&0&1&1&0&1&0&0&0&1&0&0&0\\
0&0&0&0&0&1&1&0&1&0&0&0&1&0&0\\
0&0&0&0&0&0&1&1&0&1&0&0&0&1&0
\end{bmatrix}$$

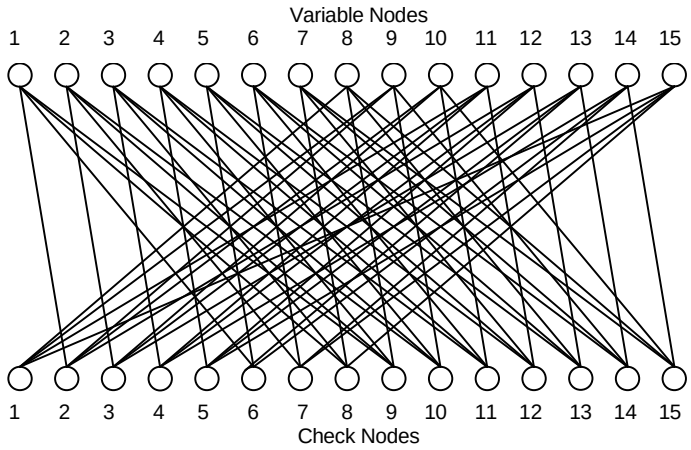

Figure 3.3: Type-I EG LDPC matrix and Tanner graph

$$H_{EG}^{(2)} = \begin{bmatrix}
0&1&0&0&0&1&0&1&1&0&0&0&0&0&0\\
0&0&1&0&0&0&1&0&1&1&0&0&0&0&0\\
0&0&0&1&0&0&0&1&0&1&1&0&0&0&0\\
0&0&0&0&1&0&0&0&1&0&1&1&0&0&0\\
0&0&0&0&0&1&0&0&0&1&0&1&1&0&0\\
0&0&0&0&0&0&1&0&0&0&1&0&1&1&0\\
0&0&0&0&0&0&0&1&0&0&0&1&0&1&1\\
1&0&0&0&0&0&0&0&1&0&0&0&1&0&1\\
1&1&0&0&0&0&0&0&0&1&0&0&0&1&0\\
0&1&1&0&0&0&0&0&0&0&1&0&0&0&1\\
1&0&1&1&0&0&0&0&0&0&0&1&0&0&0\\
0&1&0&1&1&0&0&0&0&0&0&0&1&0&0\\
0&0&1&0&1&1&0&0&0&0&0&0&0&1&0\\
0&0&0&1&0&1&1&0&0&0&0&0&0&0&1\\
1&0&0&0&1&0&1&1&0&0&0&0&0&0&0
\end{bmatrix}$$

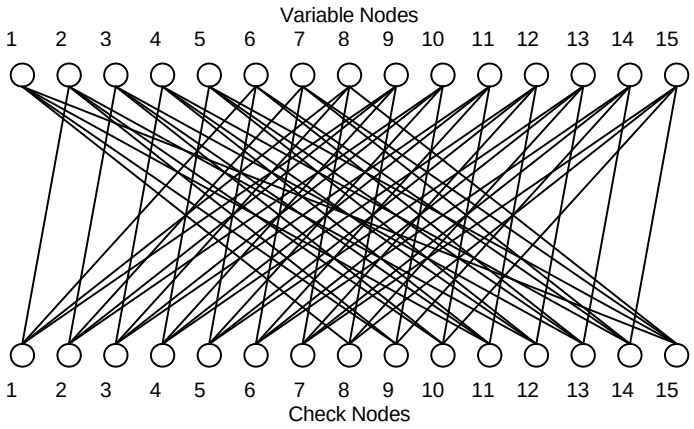

Figure 3.4: Type-II EG LDPC matrix and Tanner graph

The origin is removed from the geometry when constructing the type-I and type-II EG-LDPC codes.

3.4.2.2 Projective Geometries

A projection is the transformation of points and lines in the one plane onto another plane by linking corresponding points on the two planes with parallel lines. The branch of geometry dealing with the properties and invariants of geometric figures under projection is called projective geometry.

c)Type I Projective Geometry LDPC Codes

Let α to be a primitive element of $GF\left(2^{(m+1)s}\right)$, considered to be an extension to the $GF\left(2^s\right)$ field. An m-dimensional projective geometry, $PG\left(m,2^S\right)$, over $GF\left(2^s\right)$ is defined as:

$$\left(\alpha^0\right),\left(\alpha^1\right),\left(\alpha^2\right),\text{K},\left(\alpha^{n-1}\right)$$

where the number of elements is given by:

$$n=\frac{2^{(m+1)s}-1}{2^2-1}$$

Each α^i element corresponds to the points of the $PG(m,2^S)$ geometry. The quantity of lines in $PG(m,2^S)$ is given by:

$$J = \frac{(1+2^s+\Lambda\ +2^{ms})(1+2^s+\Lambda\ +2^{(m-1)s})}{1+2^s}$$

where each line consists of 2^2-1 points. Every α^i element in $PG(m,2^S)$ is intersected by:

$$\gamma = \frac{2^{ms}-1}{2^s-1}$$

lines. The lines in $PG(m,2^S)$ will be either disjoint or intersect at one and only one point. The incidence vector of a line $\mathcal{L}$ in $PG(m,2^S)$ is an n-tuple such that:

$$v_{\mathcal{L}} = (v_0, v_1, \Lambda\ , v_{n-1})$$

where:

$$v_i = \begin{cases} 1 & \text{if } \mathcal{L} \text{ contains the point } (\alpha^i), \\ 0 & \text{otherwise.} \end{cases}$$

The incidence vector will have a weight of 2^S+1. The incidence vectors of the lines in $PG(m,2^S)$ will form the rows of the parity check matrix $H_{PG}^{(1)}$ with the columns of the matrix corresponding to the points of $PG(m,2^S)$. The matrix $H_{PG}^{(1)}$ will have dimensions of $J = \frac{(2^{(m-1)s}+\Lambda\ +2^s+1)(2^{ms}+\Lambda\ +2^s+1)}{2^s+1}$ rows and $n = \frac{2^{(m+1)s}-1}{2^s-1}$ columns. $H_{PG}^{(1)}$ will the following properties:

(i) a row weight of $\rho = 2^S+1$,

(ii) a column weight of $\gamma = \frac{2^{ms}-1}{2^s-1}$,

(iii) a density of $r = \frac{\rho}{n} = \frac{(2^s - 1)(2^s + 1)}{2^{(m+1)s} - 1}$,

(iv) no two rows or columns having more than one 1 in common, and

(v) when m is large r is approximately $2^{-(m-1)s}$ so the matrix will be very sparse.

The null space of $H_{PG}^{(1)}$ provides a cyclic LDPC code of length $n = \frac{2^{(m+1)s} - 1}{2^s - 1}$ having a minimum distance of at least $\gamma + 1 = \frac{2^{ms} - 1}{2^s - 1} + 1$. This LDPC code is denoted $C_{PG}^{(1)}(m, 0, s)$, being an m-dimensional type-I $(0, s)$ th order Projective Geometry LDPC code.

3.5 LPDC Decoding

As stated previously an LDPC code represents the null space of a sparse parity check matrix H. When a codeword v is received the decoder will conduct a test by computing the following $(n-k)$-tuple:

$$s = v.H^T$$

where $s = (s_0, s_1, \mathrm{K}, s_{n-k-1})$ and is said to be the syndrome of v. The syndrome s will be a zero vector if and only if v is a legitimate codeword. If s is not a zero vector then the received codeword contains one or more errors. All LDPC decoders employ syndrome testing to detect the presence of errors.

3.5.1 Bit-flipping Algorithm

The bit-flipping algorithm was implemented by Gallager in his original work into LDPC codes (Gallager, 1963). The bit-flipping algorithm is a hard-decision classed message-passing algorithm. A binary (hard) decision about each received bit is made by the detector and this is passed to the decoder. For the bit-flipping algorithm the messages passed along the Tanner graph edges are also binary: a bit node sends a message declaring if it is a one or a zero, and each check node sends a message to each connected bit node, declaring what value the bit is based on the information available to the check node. The check node determines that its parity-check equation is satisfied if the modulo-2 sum of the incoming bit values is zero. If

the majority of the messages received by a bit node are different from its received value the bit node changes (flips) its current value. This process is repeated until all of the parity-check equations are satisfied, or until some maximum number of decoder iterations has passed and the decoder gives up. The bit-flipping decoder can be immediately terminated whenever a valid codeword has been found by checking if all of the parity-check equations are satisfied. This is true of all message-passing decoding of LDPC codes and has two important benefits. Firstly additional iterations are avoided once a solution has been found, and secondly a failure to converge to a codeword is always detected.

The bit-flipping algorithm is based on the principal that a codeword bit involved in a large number of incorrect check equations is likely to be incorrect itself. The sparsity of H helps spread out the bits into checks so that parity-check equations are unlikely to contain the same set of codeword bits.

3.5.2 Weighted Majority Logic Algorithm

The bit-flipping algorithm can be improved by the inclusion of some form of reliability information for the codeword received that will provide enhanced decoding decisions (Lin and Costello, 2004). Decoding is carried out by calculating weighted check sums for each bit by:

$$E_l \stackrel{\Delta}{=} \sum \left(2s_j^{(l)} - 1\right)\left|y_j\right|_{\min}^{(l)} \tag{3. 12}$$

where $\left|y_j\right|_{\min}^{(l)} \stackrel{\Delta}{=} \left\{\min\left\{\left|y_i\right|\right\}: 0 \le i \le n-1, h_{j,i} = 1\right\}$

The bit position that has the highest E_l value is then flipped. The process is carried out until either the syndrome test is passed or the maximum value of iterations is reached.

3.5.3 Sum-Product Algorithm

The sum-product algorithm is a soft decision message-passing algorithm. It is similar to the bit-flipping algorithm described in the previous section, but with the messages representing each decision (check met, or bit value equal to 1) now probabilities. Whereas bit-flipping decoding accepts an initial hard decision on the received bits as input, the sum-product algorithm is a soft decision algorithm which accepts the probability of each received bit as input.

The input bit probabilities are called the *a priori* probabilities for the received bits because they were known in advance before running the LDPC decoder. The bit probabilities returned by the decoder are called the *a posteriori* probabilities. In the case of sum-product decoding these probabilities are expressed as log-likelihood ratios.

For a binary variable x it is easy to find $p(x=1)$ given $p(x=0)$, since $p(x=1)=1-p(x=0)$ and so there is only the need to store one probability value for x. Log likelihood ratios are used to represent the metrics for a binary variable by a single value:

$$L(x)=\log\left(\frac{p(x=0)}{p(x=1)}\right) \tag{3. 13}$$

If $p(x=0)>p(x=1)$ then $L(x)$ will be positive, the greater the difference between $p(x=0)$ and $p(x=1)$, that there is more confidence that $p(x)=0$, then the larger the positive value for $L(x)$. Conversely, if $p(x=1)>p(x=0)$ then $L(x)$ will be negative and the greater the difference between the larger the negative value of $L(x)$. Therefore the sign of the log likelihood ration provides a hard decision on value of x and the magnitude the absolute value of $L(x)$ determines the reliability of the decision. It is possible to calculate the probabilities from the log likelihood ratios by:

$$p(x=0)=\frac{p(x=0)/\,p(x=1)}{1+\,p(x=0)/\,p(x=1)}=\frac{e^{L(x)}}{1-e^{L(x)}}$$

and

$$p(x=1)=\frac{p(x=1)/\,p(x=0)}{1+\,p(x=1)/\,p(x=0)}=\frac{e^{-L(x)}}{1-e^{-L(x)}}$$

The benefit of the logarithmic representation of probabilities is that when probabilities need to be multiplied log-likelihood ratios need only be added, reducing implementation complexity.

The aim of sum-product decoding is to compute the maximum *a posteriori* probability for each codeword bit, $P_i = P\{c_1 = 1 | N\}$, which is the probability that the i-th codeword bit is a 1 conditional on the event N that all parity-check constraints are satisfied. The extra information about bit i received from the parity-checks is deemed extrinsic information for bit i.

The sum-product algorithm iteratively computes an approximation of the maximum a posteriori value for each code bit. However, the a posteriori probabilities returned by the sum-product decoder are only accurate values if the parity check matrix is cycle free, as can be determined by the Tanner graph. The extrinsic information obtained from a parity check constraint in the first iteration is independent of the a priori probability information for that bit, though it does depend on the a priori probabilities of the other codeword bits. The extrinsic information provided to i-th bit in subsequent iterations remains independent of the original a priori probability for bit i until the original a priori probability is returned back to bit i via a cycle in the Tanner graph. The association of the extrinsic information with the original a priori bit probability is what prevents the resulting posteriori probabilities from being exact.

3.6 Summary

In this chapter, we discussed the general principles of LDPC codes including code construction, message encoding and codeword decoding algorithms.

In the early 1960's when first discovered, the limited computing resources prevented Gallager from demonstrating the full capabilities of message passing LDPC decoders. Codeword lengths were limited to no longer than around 500 bits as the computation required for encoding and decoding made the codes impractical for use. This resulted in his work being ignored for over 30 years except by all but a handful of researchers. It has only been in recent times that the re-discovered by researchers after the emergence of turbo decoding has shown the true benefits of LDPC codes.

Chapter 4

Performance comparison of LDPC Codes

Chapter 4 Performance comparison of LDPC encoders and decoders

4.1 LDPC Encoder Implementation

Generator Matrix Encoding

The simulator function MakeGenerator is used to create a generator matrix from the supplied parity check matrix. The function is passed pointers to the parity check matrix and the allocated space for the generator matrix, along with the row and column dimensions. A working matrix is required to manipulate the parity check matrix data without amending the original matrix which will be required in the decoding functions. The working matrix is required to have the same form and data as the parity check matrix. This is accomplished by the following code:

```
temp_matrix = malloc(rows * sizeof(int *));
if(temp_matrix == NULL){
	printf("Error: failed to allocate memory for temp_matrix matrix\n");
		exit(1);
}
for(i = 0; i < rows; i++){
	temp_matrix[i] = malloc(n * sizeof(int *));
	if(temp_matrix[i] == NULL){
		printf("Error: failed to allocate memory for temp_matrix row\n");
		exit(1);
```

```
	}
}
//Initialise temp matrix to parity check matrix
for(i = 0; i < rows; i++){
	for(j = 0; j < n; j++){
		temp_matrix[i][j] = pchk_matrix[i][j];
	}
}
```

The first operation is put the parity check matrix H into row-echelon form (i.e. so that in any two successive rows that do not consist entirely of zeros, the leading 1 in the lower row occurs further to the right than the leading 1 in the higher row).

The matrix H is put into this form by applying elementary row operations in GF(2), which are; interchanging two rows or adding one row to another modulo 2. From linear algebra it is known that by using only elementary row operations the modified parity-check matrix will have the same null space and therefore the same codeword set as the original.

To achieve this process is broken into two steps. Firstly to make the matrix lower triangular, which achieved by the following code:

```
//Make temp matrix lower triangular
for(i = 0; i < rows; i++){
	//Ensure curent matrix data starts with a 1
	if(temp_matrix[i][i] == 0){
		//Find the next row with a 1 in the column and swap
		rowIdx = FindRowdata(temp_matrix, 1, i+1, rows, i, 1);
		if(rowIdx < 0){
			printf("Error: unable to make generator matrix from parity
			check matrix\n");
			printf("Reason: no row to swap to make H lower
			triangular\n");
			return state ;
		}
		else {
			SwapRows(temp_matrix, i, rowIdx, n);
		}
	}
	//Ensure each row below current row does not start have a 1 for this
	column
	for(j = i+1; j < rows; j++){
		if(temp_matrix[j][i] == 1){
			AddRows(temp_matrix, i, j, n);
		}
	}
}
```

The next step requires the clearing of the upper triangular portion which then makes a reduced row-echelon form matrix.

```
//Make temp matrix upper triangular
for(i = rows-1; i >= 0; i--){
//Ensure each row below current row does not start have a 1 for this column
       for(j = i-1; j >= 0; j--){
              if(temp_matrix[j][i] == 1){
                     AddRows(temp_matrix, i, j, n);
              }
       }
}
```

The working matrix is now in the correct form to make the generator matrix. The parity submatrix is taken from the working matrix, inverted and copied into the generator.

```
//Take Parity submatrix and transpose into generator matrix
for(i = 0; i < rows; i++){
       for(j = rows; j < n; j++){
              gen_matrix[j-rows][i] = temp_matrix[i][j];
       }
}
```

The final step to complete the generator matrix is to add an identity matrix.

```
//Add Identity matrix to generator matrix
for(i = 0; i < k; i++){
       gen_matrix[i][i+rows] = 1;
}
```

The generator matrix is now complete. In order to encode the message into a codeword the generator matrix is multiplied by the message vector, the result is the codeword to be transmitted.

All of this processing can be done off-line and just the matrices G and H' provided to the encoder and decoder respectively. However, the drawback of this approach is that, unlike H, the matrix G will most likely not be sparse and so the matrix multiplication at the encoder will have complexity in the order of n^2 operations. As n is large for LDPC codes, from thousands to hundreds of thousands of bits, the encoder can become prohibitively complex. Later we will see that structured parity-check matrices can be used to significantly lower this implementation complexity, however for arbitrary parity-check matrices a good approach is to avoid constructing G at all and instead to encode using back substitution with H as is demonstrated in the following approximate lower triangular form matrix encoding.

4.2 LDPC Decoder Implementation

Two types of decoders were implemented successfully, the sum-product algorithm based on belief propagation and the standard bit-flipping algorithm.

The bit-flipping decoder required the number of failed checksums to be calculated. To achieve this each bit is checked in turn by determining the variable bits that make up the checksum from the check nodes. The following code fragment carries out the check and compares the received bit to the calculated bit. If the comparison is incorrect then a counter for that bit is incremented.

```
for(j = 0; j < ncols; j++){
      //Get the current message bit
      currentBit = decoded[j];
      //Find a link to a check node
      for(i = 0; i < nrows; i++){
            if(matrix[i][j] == 1){
                  //Find the links to the variable nodes
                  for(k = 0; k < ncols; k++){
                        if(k != j && matrix[i][k] == 1)
                              //Add the message bits
                              sum = sum + decoded[k];
                  }
                  //Compare sum to original bit
                  if(currentBit != fmod(sum , 2)){
                        //Sum is incorrect increment the failure count
                        check_fails[j]++;
                  }
                  sum = 0;
            }
      }
}
```

The vector check_fails is then checked to determine the value of the greatest number of checksum failures. Once this is determined it is a simple matter to find which bits of the received codeword is required to be changed to the opposite information bit. The following code fragment carries out these requirements:

```
//Find the bits with largest failed checksums
for(j = 0; j < ncols; j++){
      if(check_fails[j] > maxFailed)
            maxFailed = check_fails[j];
}

//Flip the bits having the largest failure rate
```

```
for(j = 0; j < ncols; j++){
      if(check_fails[j] == maxFailed)
            decoded[j] = (int)fmod((decoded[j] + 1), 2.0);
}
```

The decoded vector is now checked to determine if the syndrome test is satisfied. If the syndrome test passes the algorithm terminates, otherwise the algorithm will continue up to the maximum iterative count.

The bit-flipping algorithm is a relatively quick decoder but is has the drawback of inducing faults into the codeword. A problem that can occur is that one or more bits will be caught in a loop where the bits will be constantly flipped but the codeword is never closer to being decoded. When this happens the algorithm will require termination through the maximum iteration constraint.

The sum product algorithm required the implementation of a number of working matrices of the same dimension as the parity check matrix. This results in the SPA decoder having greater memory utilization than the BF decoder.

After initializing the memory requirements of the algorithm the Gaussian density probability is calculated.

```
//Calculate the Guassian density probabilities
for(i = 0; i < ncols; i++)
      f1[i] = 1 /(1+exp((-2*codeword[i])/pow(ch_dev,2)));

for(i = 0; i < ncols; i++)
      f0[i] = 1 - f1[i];
```

The *a priori* probabilities are then calculated.

```
/*calculate the q1ij probabilities*/
for(i = 0; i < nrows; i++){
      for(j = 0; j < ncols; j++){
            if(q1[i][j] != 0)
                  q1[i][j] *= f1[j];
      }
}

//calculate the q0ij probabilities
for(i = 0; i < nrows; i++){
      for(j = 0; j < ncols; j++){
```

```
            if(q0[i][j] != 0)
                  q0[i][j] *= f0[j];
      }
}
```

The algorithm is now ready to start the decoding iterations. Upon entering the iteration loop the soft codeword is hard decoded and then the syndrome test is conducted. If the test passes the loop is terminated and the codeword has been decoded, else the algorithm continues. The next step is to calculate the probability difference equations.

```
//Calculate the probability deltas
for(i = 0; i < nrows; i++){
      for(j = 0; j < ncols; j++){
            deltaQ[i][j] = q0[i][j] - q1[i][j];
      }
}

for(i = 0; i < nrows; i++){
      for(j = 0; j < ncols; j++){
            if(deltaP[i][j] != 0){
                  for(k = 0; k < ncols; k++){
                        if(j != k && deltaP[i][k] != 0){
                              deltaP[i][j] *= deltaQ[i][k];
                        }
                  }
            }
      }
}
```

The bit probability values are then calculated by:

```
//Calculate p values
for(i = 0; i < nrows; i++){
      for(j= 0; j < ncols; j++){
            if(p0[i][j] != 0){
                  p0[i][j] *= (0.5 * (1 + deltaP[i][j]));
                  p1[i][j] *= (0.5 * (1 - deltaP[i][j]));
            }
      }
}
```

From the bit probability the coefficients are then calculated:

```
//calculate the coefficients
for(i = 0; i < ncols; i++){
      for(j = 0; j < nrows; j++){
            if(matrix[j][i] != 0){
                  coeff0 = 1.0;
                  coeff1 = 1.0;
                  for(k = 0; k < nrows; k++){
                        if(j != k && matrix[k][i] != 0){
                              coeff0 *= p0[k][i];
```

```
                        coeff1 *= p1[k][i];
                    }
                }
                coeff0 *= f0[i];
                coeff1 *= f1[i];
                q0[j][i] = coeff0 /(coeff0 + coeff1);
                q1[j][i] = coeff1 /(coeff0 + coeff1);
            }
        }
}
```

A new soft codeword is generated by scaling the calculated bit values. This new codeword is now hardcoded and a syndrome test is conducted.

The one step majority logic decoder was not completed.

4.3 Simulation Results

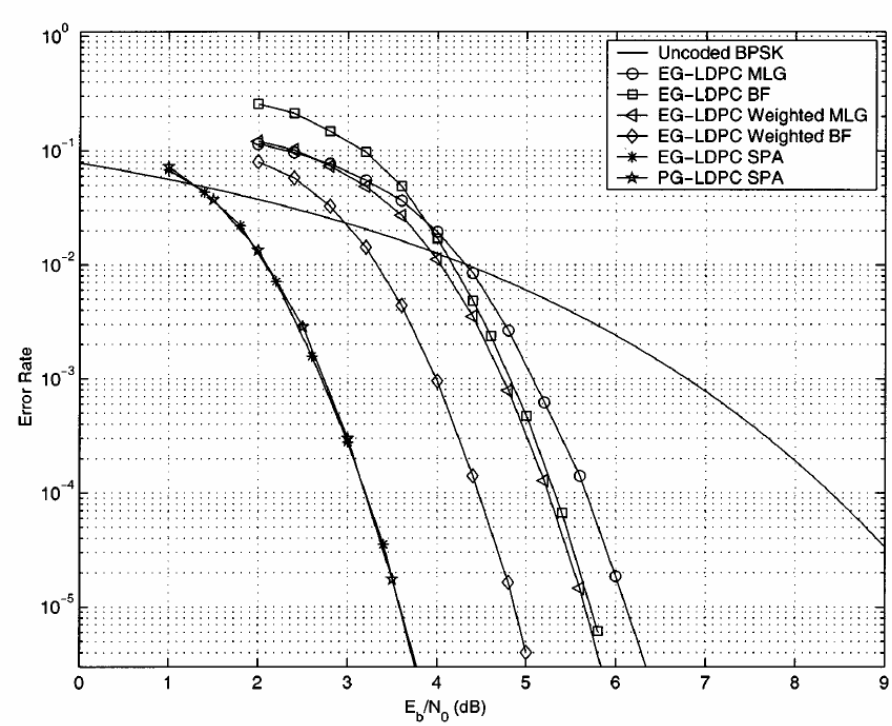

Figure 4.1 Bit-error probabilities of the type-I 2-D (255; 175) EG-LDPC code and (273; 191) PG-LDPC code based on different decoding algorithms. (Kou *et al*, 2001)

Figure 4.1 shows the comparison between geometric codes utilizing various decoding techniques. As can be seen the SPA decoder outperforms all other decoding algorithms for the same error rate by more than 1 dB.

4.4 Summary

In this chapter, the different techniques for encoding and decoding LDPC were simulated. It was shown that the Sum Product algorithm has the most potential to be able to correct a codeword that has been subjected to large amounts of noise. The drawback of the SPA decoder is the additional complexity required to implement which has a great effect on the latency of the decoder, this is especially true when very large code lengths are utilised.

The technique of using the approximate lower triangular method of fast encoding LDPC code was not implemented.

Chapter 5

Conclusions

Chapter 5 Conclusions

5.1 Concluding Remarks

LDPC codes have been proven to be one of the most powerful ECC codes discovered. Today, design techniques for LDPC codes exist which enable the construction of codes which approach within hundredths of a decibel of the Shannon limit of a communications channel.

In addition to the strong theoretical interest in LDPC codes, such codes have already been adopted in satellite-based digital video broadcasting and long-haul optical communication standards, could be adopted in the IEEE wireless local area network standard, and are under consideration for the long term evolution of third-generation mobile telephony.

This project has outlined various techniques for constructing LDPC codes, encoding messages for transmission and the decoding of received messages over an AWGN channel. The simulation of the various algorithms did not achieve their intent due to my inability to correctly simulate the channel environment. Analysis of the different decoding techniques therefore relied on work conducted by other researchers.

5.2 Future Work

LDPC codes will be utilised more often in future in all forms of wireless communications. During this project I have discovered that the main flaw of the Bit Flipping algorithm is its inability to terminate when caught in a loop. This algorithm could be further improved to provide a learning algorithm that would evolve a strategy to flip only a dedicated bit rather than the entire set of bits with the most checksum failures.

An efficient technique for computer simulations of the approximate lower triangular encoding technique should be investigated.

References

Ball, S & Weiner, Z 2007, '*An Introduction to Finite Geometry*', Department of Applied Mathematics IV, Technical University of Catalonia.

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, Vol 2, pp. 1064-1070.

Chung, S, Forney, DG, Richardson, TJ & Urbanke, R (2001), '*On the Design of Low-Density Parity Check Codes within 0.0045dB of the Shannon Limit*', IEEE Communications Letters, Vol. 5, No. 2, pp. 58-60.

Dechter, R & Mateescu, R (2003), '*A Simple Insight Into Properties of Iterative Belief Propagation'*, In proceedings of Uncertainty in Artificial Intelligence.

Dubendorf, VA (2003), '*A History of Wireless Technologies*', Wireless Data Technologies, John Wiley & Sons.

Elias, P 1954, '*Error-free Coding*', IRE Trans. Information Theory, Vol. 4, No. 4, pp. 29-37.

Elias, P 1955, '*Coding for Noisy Channels*', IRE Convention Record pt 4, pp. 37-46.

Forney, GD 1966, '*Concatenated Codes*', MIT Press, Cambridge.

Gallager, RG 1963, '*Low-Density Parity-Check Codes*', No. 21 in MIT Research monograph series, MIT Press, Cambridge, MA.

Larson, R, Edwards, BH & Falvo, DC 2004, *Elementary Linear Algebra*, 5th edn, Houghton Mifflin Company, New York.

Lentmaier, M , Truhachev, DV, Zigangirov, KS & Costello, DJ 2005, '*An Analysis of the Block Error Probability Performance of Iterative Decoding*', IEEE Transactions on Information Theory, Vol. 51, No. 11, pp. 3834-3855.

Lin, S & Costello, DJ 2004, *Error Control Coding*, 2nd edn, Person Prentice Hall, New Jersey.

Kou, Y, Lin, S & Fossorier, M 2001, '*Low Density Parity Check Codes Based on Finite Geometries: A Rediscovery and New Results*', IEEE Transactions on Information Theory, Vol 47, No. 7, pp. 2711-2736.

MacKay, DJC 1999, '*Good Error-Correcting Codes Based on Very Sparse Matrices*', IEEE Transactions on Information Theory, Vol. 45, No. 2, pp. 399-431.

MacKay, DJC 2005, '*Information Theory, Inference, and Learning Algorithms*', Ver. 7.2, Cambridge University Press.

Min-Ho, S, Joon-Sung, K & Hong-Yeop S 2005, '*Generalization of Tanner's Minimum Distance Bounds for LDPC Codes*', IEEE Transactions on Information Theory, Vol. 9, No. 3, pp. 240-242.

Pishro-Nik, H & Fekri, F 2004, '*Decoding LDPC codes Over the Binary Erasure Channel*', IEEE Transactions on Information Theory, Vol. 50, No. 3, pp. 439-454.

Richardson, TJ & Shokrollahi, MA 2001, '*Design of Capacity-Approaching Irregular Low-Density Parity-Check Codes*', IEEE Transactions on Information Theory, Vol. 47, No. 2, pp. 619-637.

Richardson, TJ & Urbanke RL 2001, '*Efficient Encoding of Low-Density Parity-Check Codes*', IEEE Transactions on Information Theory, Vol. 47, No. 2, pp. 638-656.

Shannon, CE 1948, '*A Mathematical Theory of Communication*', The Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656.

Sniffen, RW 2004, '*DSN Telecommunications Link Design Handbook*', Jet Propulsion Laboratory, California Institute of Technology, 810-005, Rev. E Change 1.

Tanner, RM 1981, '*A Recursive Approach to Low Complexity Codes*', IEEE Transactions on Information Theory, Vol. 27, No. 5, pp. 533-547.

Viterbi, AJ 1967, '*Error Bounds for Convolutional Codes and an Asymptotically Optimal Decoding Algorithm*', IEEE Transactions on Information Theory, Vol. 13, No. 2, pp. 260-269.

Appendix A

Project specification

ENG 4111/4112 Research Project
PROJECT SPECIFICATION

FOR: **Paul James Lauder**

TOPIC: Performance evaluation of Low Density Parity Check (LPDC) codes.

SUPERVISOR: Dr. Wei Xiang

SPONSORSHIP: University of Southern Queensland

PROJECT AIM: This project aims to evaluate the performance of LDPC codes. The scope of the project involves building a computer experimental platform for code performance evaluation, plotting bit error rate (BER) curves for the codes, and discussing the encoding and decoding complexity of LDPC codes.

PROGRAMME: **Issue A, 20th March 2007**

1. Study the background of Error Control Coding.
2. Implement and analyse LPDC encoding algorithms.
3. Implement and analyse LPDC decoding algorithms.
4. Ascertain the advantages and disadvantages of regular and irregular LPDC codes, in terms of error resilient performance, latency and computational resources.
5. Determine the practicality of utilising LPDC codes in modern wireless networks.

As time permits:

6. Compare LPDC coding to turbo coding.

AGREED: _____(Student) _____(Supervisor)

Date: / / 2007 Date: / / 2007

Co-examiner: _____

Appendix B

List of codes

```
// LDPC_Simulator.cpp : Defines the entry point for the console
application.
//
#include "stdafx.h"
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "limits.h"
#include "malloc.h"
#include "time.h"

#define MAX_ITERATION 10
#define FALSE 0
#define TRUE 1
//Simulation channel signal to noise parameters
#define DB_MAX 2
#define DB_INC 0.25
#define DB_INIT -2
//Channel message bit parameters
#define HIGH 1          //BSC = +1        BKSP = +1
#define MID 0           //BSC = +0.5      BKSP = 0
#define LOW -1          //BSC = +0        BKSP = -1

int MakeGenerator(int **pchk_matrix, int **gen_matrix, int rows, int n,
int k);
void SwapRows(int **matrix, int rowIdx1, int rowIdx2, int cols);
void AddRows(int **matrix, int rowIdx1, int rowIdx2, int cols);
int FindRowdata(int **matrix, int data, int rowSt, int rows, int colIdx,
int dir);
int BitFlipping(int **pchk_matrix, double *codeword, int *decoded, int
nrows, int ncols);
int SumProduct(int **pchk_matrix, double *codeword, int *decoded, int
nrows, int ncols, double ch_dev);
int OneStepMajorityLogic(int **pchk_matrix, double *codeword, int
*decoded, int nrows, int ncols);
void PrintIntMatrix(int **matrix, int nrows, int ncols);
void PrintMatrix(double **matrix, int nrows, int ncols);
void FileOutput(FILE *fp, char type, double *codeword, int length);
void PrintCodeword(double *codeword, int ncols);
void PrintIntCodeword(int *codeword, int ncols);
void FreeMatrix(double **matrix, int nrows);
void FreeIntMatrix(int **matrix, int nrows);
double BitErrorRate(int *original, int *current, int length);
int SyndromeTest(int **matrix, int *codeword, int nrows, int ncols);
void AddWhiteGaussianNoise(int *codeword, double *received, int length,
double variance, double mean);
void HardDecision(int *hardcode, double *softcode, int ncols);
void ZeroMatrix(double **matrix, int nrows, int ncols);
void ZeroVector(double *vector, int ncols);
void ZeroIntVector(int *vector, int ncols);
void ResetMatrix(int **pchk_matrix, double **matrix, int nrows, int
ncols);
void BSCtoBKSP(int *codeword, int ncols);
int XOR(int a, int b);
```

```
void main(){
	//variables
	int **gen_matrix = NULL;	/*Pointer to Generator matrix*/
	int **pchk_matrix = NULL;	/*Pointer to Parity Check matrix*/
	int *codeword = NULL;		/*Channel codeword vector*/
	int *original = NULL;		/*Original codeword vector*/
	int *decoded = NULL;		/*Final decoded vector*/
	double *received = NULL;	/*Codeword + noise vector received*/
	FILE *fp = NULL;		/*Input file pointer*/
	int pchk_rows = 0;			/*Parity Check matrix row
dimension*/
	int pchk_cols = 0;			/*Parity Check matrix column
dimension*/
	int n = 0;				/*Codeword and Generator
matrix column dimension */
	int k = 0;				/*Message and Generator matrix
row dimension*/
	double code_rate = 0.0;		/*Code rate*/
	double bit_error = 0.0;		/*Bit Error Rate*/
	double variance = 0.0;		/*Channel variance*/
	double deviation = 0.0;		/*Channel standard deviation*/
	double dB = 0.0;		/*Signal to Noise ratio*/
	double mean = 0.0;			/*Signal mean*/
	int i, j;				/*Local counter variables*/
	int row_wt = 0;
	int status = 0;				/*Decoder status bit*/
	int type_decoder = 0;		/*Decoder choice*/
	int type_encoder = 0;		/*Encoder choice*/
	char filename[200];			/*Input filename containing
parity check matrix*/

	system("cls");

	//Simulator Engine
	printf(" ************* LDPC Simulator ***************\n");

	//Get encoder options from user
	printf("Encoding Options:\n");
	printf("  1. Load Parity Check Matrix from file\n");
	printf("  2. Generate Parity Check Matrix");
	while (type_encoder < 1 && type_encoder > 2){
		printf("Enter a number between 1-2: ");
		scanf("%d", &type_encoder);
	}
	switch(type_encoder){
		case 1:
			while(fp =
fopen("C://Project//ldpc//Debug//pchk_matrix_8_12.txt","r");
			printf("Enter filename: ");



	//Get decoder option from user
	printf("Available Decoders:\n");
	printf("  1. Sum Product Algorithm\n");
```

```
      printf("  2. Bit Flipping Algorithm\n");
      printf("  3. Majority Logic Algorithm\n");
      printf("Enter Decoder number: ");
      scanf("%d", &type_decoder);
      while (type_decoder < 1 && type_decoder > 3){
            printf("Enter a number between 1-3: ");
            scanf("%d", &type_decoder);
      }

      //Load parity check matrix from file
      //File contents to be:
      //nrows ncols
      //H00.......H0ncols
      // .           .
      //Hnrows0...Hnrowsncols
      //fp = fopen("C://Project//ldpc//Debug//pchk_matrix_3_7.txt","r");
      //fp =
fopen("C://Project//ldpc//Debug//pchk_matrix_6_12.txt","r");
      fp = fopen("C://Project//ldpc//Debug//pchk_matrix_8_12.txt","r");
      //fp =
fopen("C://Project//ldpc//Debug//pchk_matrix_816_408.txt","r");

      //fp =
fopen("C://Project//ldpc//Debug//pchk_matrix_15_15.txt","r");
      if (fp == NULL){
            printf("Error: Could not open input file ...\n");
            exit(0);
      }

      //Get matrix dimensions
      fscanf(fp, "%d %d", &pchk_rows, &pchk_cols);

      //Calculate code dimensions
      n = pchk_cols;
      k = n - pchk_rows;
      code_rate = (double)k / (double)n;

      //Allocate Parity Check matrix memory space
      pchk_matrix = malloc(pchk_rows * sizeof(int *));
      if(pchk_matrix == NULL){
            printf("Error: failed to allocate memory for pchk_matrix
matrix\n");
            exit(1);
      }
      for(i = 0; i < pchk_rows; i++){
            pchk_matrix[i] = malloc(n * sizeof(int *));
            if(pchk_matrix[i] == NULL){
                  printf("Error: failed to allocate memory for
pchk_matrix row\n");
                  exit(1);
            }
      }

      //Populate Parity Check matrix data
      for(i = 0; i < pchk_rows; i++){
```

```
            for(j = 0; j < n; j++){
                  fscanf(fp, "%d", &pchk_matrix[i][j]);
            }
      }
      //Allocate Parity Check matrix memory space
      gen_matrix = malloc(k * sizeof(int *));
      if(gen_matrix == NULL){
            printf("Error: failed to allocate memory for gen_matrix
matrix\n");
            exit(1);
      }
      for(i = 0; i < k; i++){
            gen_matrix[i] = malloc(n * sizeof(int *));
            if(gen_matrix[i] == NULL){
                  printf("Error: failed to allocate memory for
gen_matrix row\n");
                  exit(1);
            }
      }

      //Initialise the generator matrix
      ZeroMatrix(gen_matrix, k, n);



      //Allocate memory for codeword vector
      codeword = (int *)malloc(sizeof(int) * n);
      if(codeword == NULL){
            printf("Error: failed to allocate memory for codeword
vector\n");
            exit(1);
      }
            //Allocate memory for codeword vector
      original = (int *)malloc(sizeof(int) * k);
      if(original == NULL){
            printf("Error: failed to allocate memory for codeword
vector\n");
            exit(1);
      }
      //Allocate memory for decoded vector
      decoded = (int *)malloc(sizeof(int) * n);
      if(decoded == NULL){
            printf("Error: failed to allocate memory for decoded
vector\n");
            exit(1);
      }

      //Allocate memory for received codeword vector
      received = (double *)malloc(sizeof(double) * n);
      if(received == NULL){
            printf("Error: failed to allocate memory for received
vector\n");
            exit(1);
      }
```

```
	//For testing set codeword to zero vector
	ZeroIntVector(codeword, n);
	ZeroIntVector(original, k);
	PrintIntCodeword(codeword, n);

	//Convert the codeword to bpsk
	BSCtoBKSP(codeword, n);

	PrintIntCodeword(codeword, n);
	PrintIntMatrix(pchk_matrix, pchk_rows, n);	/*DELETE AFTER
TESTING*/

	for(dB  = DB_INIT; dB < DB_MAX; dB += DB_INC){
		printf("db: %lf\n", dB);
		//Initialise received vector
		ZeroVector(received, n);

		//Calculate variance for current SNR
		variance = 1.0/(2.0*code_rate*pow(10.0,dB/10.0));

		//Calculate standrad deviation
		deviation = sqrt(variance);

		//Add White Guassian Noise
		AddWhiteGaussianNoise(codeword, received, n, variance,
mean);
		printf("Sent:     ");
		PrintIntCodeword(codeword, n);
		//printf("Received: ");
		//PrintCodeword(received, n);

		//Run Decoders
		switch(type_decoder){
			case 1: status = SumProduct(pchk_matrix, received,
decoded, pchk_rows, n, deviation);
				break;
			case 2: status = BitFlipping(pchk_matrix, received,
decoded, pchk_rows, n);
				break;
			case 3: status = OneStepMajorityLogic(pchk_matrix,
received, decoded, pchk_rows, n);
				break;
			default:
				printf("Error: Unable to resolve decoder
type.\n");
		}


		printf("Last:     ");
		PrintIntCodeword(decoded, n);
		if(status == 1)
			printf("Decoded\n");
		else
			printf("Unresolved\n");
```

```
            bit_error = BitErrorRate(original, decoded, n);

            printf("BER: %lf\n", bit_error);
      }

      //Close open file pointers
      fclose(fp);

      //Free used memory
      FreeIntMatrix(pchk_matrix, pchk_rows);
      free(codeword);
      free(decoded);
      free(received);
}

/* ************* Generator Matrix Production Methods ************/
int MakeGenerator(int **pchk_matrix, int **gen_matrix, int rows, int n,
int k){
      int **temp_matrix = NULL;     //working matrix
      int state = 0;                      //Function state variable fail
= 0 success = 1
      int rowIdx = 0;                     //Row index of a matrix
      int i, j;                           //Local counter variables

      temp_matrix = malloc(rows * sizeof(int *));
      if(temp_matrix == NULL){
            printf("Error: failed to allocate memory for temp_matrix
matrix\n");
            exit(1);
      }
      for(i = 0; i < rows; i++){
            temp_matrix[i] = malloc(n * sizeof(int *));
            if(temp_matrix[i] == NULL){
                  printf("Error: failed to allocate memory for
temp_matrix row\n");
                  exit(1);
            }
      }
      //Initialise temp matrix to parity check matrix
      for(i = 0; i < rows; i++){
            for(j = 0; j < n; j++){
                  temp_matrix[i][j] = pchk_matrix[i][j];
            }
      }

      //Make temp matrix lower triangular
      for(i = 0; i < rows; i++){
            //Ensure curent matrix data starts with a 1
            if(temp_matrix[i][i] == 0){
                  //Find the next row with a 1 in the column and swap
                  rowIdx = FindRowdata(temp_matrix, 1, i+1, rows, i, 1);

                  if(rowIdx < 0){
                        printf("Error: unable to make generator matrix
from parity check matrix\n");
```

```
                        printf("Reason: no row to swap to make H lower
triangular\n");
                        return state      ;
                  }
                  else {
                        SwapRows(temp_matrix, i, rowIdx, n);
                  }
            }
            //Ensure each row below current row does not start have a 1
for this column
            for(j = i+1; j < rows; j++){
                  if(temp_matrix[j][i] == 1){
                        AddRows(temp_matrix, i, j, n);
                  }
            }
      }

      //Make temp matrix upper triangular
      for(i = rows-1; i >= 0; i--){
            //Ensure each row below current row does not start have a 1
for this column
            for(j = i-1; j >= 0; j--){
                  if(temp_matrix[j][i] == 1){
                        AddRows(temp_matrix, i, j, n);
                  }
            }
      }

      //Take Parity submatrix and transpose into generator matrix
      for(i = 0; i < rows; i++){
            for(j = rows; j < n; j++){
                  gen_matrix[j-rows][i] = temp_matrix[i][j];
            }
      }

      //Add Identity matrix to generator matrix
      for(i = 0; i < k; i++){
            gen_matrix[i][i+rows] = 1;
      }
      state = 1;

      return state;
}

void SwapRows(int **matrix, int rowIdx1, int rowIdx2, int cols){
      int temp;
      int i;

      for(i = 0; i < cols; i++){
            temp = matrix[rowIdx1][i];
            matrix[rowIdx1][i] = matrix[rowIdx2][i];
            matrix[rowIdx2][i] = temp;
      }
}
```

```
//Add content of row1 and row2 and put result into row2
void AddRows(int **matrix, int rowIdx1, int rowIdx2, int cols){
      int i;
      //Only 2 canges to make 1+1->0 & 1+0->1
      //no changes for 0+0->0 & 0+1->1
      for(i = 0; i < cols; i++){
            if(matrix[rowIdx1][i] == 1 && matrix[rowIdx2][i] == 1){
                  matrix[rowIdx2][i] = 0;
            } else if(matrix[rowIdx1][i] == 1 && matrix[rowIdx2][i] ==
0){
                  matrix[rowIdx2][i] = 1;
            }
      }
}

//Find data in a matrix given a starting position
int FindRowdata(int **matrix, int data, int rowSt, int rows, int colIdx,
int dir){
      int i;
      int rowIdx = -1;

      if(dir > 0){
            for(i = rowSt; i < rows; i++){
                  if(matrix[i][colIdx] == data){
                        rowIdx = i;
                        break;
                  }
            }
      } else {
            for(i = rowSt; i >= 0; i--){
                  if(matrix[i][colIdx] == data){
                        rowIdx = i;
                        break;
                  }
            }
      }

      return rowIdx;
}



/* ***************** Decoding Methods ************************ */
int BitFlipping(int **matrix, double *codeword, int *decoded, int nrows,
int ncols){
      int *check_fails = NULL;
      double currentBit = 0.0;
      double sum = 0.0;
      int syndrome = 0;
      int iteration = 1;
      int maxFailed = 0;
      int i, j, k;

      //Create Check Failure vector
```

```
      for(i = 0; i < ncols; i++)
            check_fails = (int *)malloc(sizeof(int) * ncols);
      if(check_fails == NULL){
            printf("Error: failed to allocate memory for
check_fails\n");
            exit(1);
      }

      //Convert to hard decision quantized form
      HardDecision(decoded, codeword, ncols);

      /*Commence decoding iterations*/
      do{
            //Check if codeword passes syndrome test
            syndrome = SyndromeTest(matrix, decoded, nrows, ncols);

            //If sydrome test failed - find and replace incorrect
message bits
            //For each message bit
            for(j = 0; j < ncols; j++){
               //Get the current message bit
               currentBit = decoded[j];
               //Find a link to a check node
               for(i = 0; i < nrows; i++){
                     if(matrix[i][j] == 1){
                           //Find the links to the variable nodes
                           for(k = 0; k < ncols; k++){
                                 if(k != j && matrix[i][k] == 1)
                                       //Add the message bits
                                       sum = sum + decoded[k];
                           }
                           //Compare sum to original bit
                           if(currentBit != fmod(sum , 2)){
                                 //Sum is incorrect increment the
failure count
                                 check_fails[j]++;
                           }
                           sum = 0;
                     }
               }
            }

            //Find the bits with largest failed checksums
            for(j = 0; j < ncols; j++){
               if(check_fails[j] > maxFailed)
                     maxFailed = check_fails[j];
            }

            //Flip the bits having the largest failure rate
            for(j = 0; j < ncols; j++){
               if(check_fails[j] == maxFailed)
                     decoded[j] = (int)fmod((decoded[j] + 1), 2.0);
            }
            maxFailed = 0;
```

```
            iteration++;
          //Reset failure counter
          for(i = 0; i < ncols; i++){
               check_fails[i] = 0;
          }
          //Reset maximum failure counter
          maxFailed = 0;
   }while (iteration < MAX_ITERATION && syndrome != 1);

   //Free allocated memory
   free(check_fails);

   return syndrome;
}

int SumProduct(int **matrix, double *codeword, int *decoded, int nrows,
int ncols, double ch_dev){
      //Variables
      double **p0 = NULL;           /*Pointer to P0ij probabilities*/
      double **p1 = NULL;           /*Pointer to P1ij probabilities*/
      double **deltaP = NULL; /*Pointer to P deltas*/
      double **deltaQ = NULL; /*Pointer to Q deltas*/
      double **q0 = NULL;           /*Pointer to Q0ij probabilities*/
      double **q1 = NULL;           /*Pointer to Q1ij probabilities*/
      double *f0 = NULL;            /*Gaussian probability function for
0 bit*/
      double *f1 = NULL;            /*Gaussian probability function for
1 bit*/
      double *soft0 = NULL;   /*soft codeword probability for bit 0*/
      double *soft1 = NULL;   /*soft codeword probability for bit 1*/
      int i, j, k;                  /*local variables*/
      int syndrome = 0;       /*syndrome test flag*/
      int iteration = 1;            /*number of iterations*/
      int num_data = 0;       /*number of data bits*/
      double coeff0 = 0.0;    /*scaling coefficient for bit 0*/
      double coeff1 = 0.0;    /*scaling coefficient for bit 1*/
      double column_sum = 0.0;/*matrix column data sum*/

      /*Create message bit vectors */
      soft0 = (double *)malloc(sizeof(double) * ncols);
      if(soft0 == NULL){
            printf("Error: failed to allocate memory for softc\n");
            exit(1);
      }
      soft1 = (double *)malloc(sizeof(double) * ncols);
      if(soft1 == NULL){
            printf("Error: failed to allocate memory for softc\n");
            exit(1);
      }

      /*Create Gaussian probability density function vectors*/
      f0 = (double *)malloc(sizeof(double) * ncols);
      if(f0 == NULL){
            printf("Error: failed to allocate memory for f0\n");
            exit(1);
```

```
      }
      f1 = (double *)malloc(sizeof(double) * ncols);
      if(f1 == NULL){
            printf("Error: failed to allocate memory for f1\n");
            exit(1);
      }

      /*Allocate space for algorithm variables*/
      /*prior probabilities*/
      p0 = (double **)malloc(sizeof(double *) * nrows);
      if(p0 == NULL){
            printf("Error: failed to allocate memory for p0\n");
            exit(1);
      }
    for(i = 0; i < nrows; i++){
            p0[i] = (double *)malloc(sizeof(double) * ncols);
          if(p0[i] == NULL){
               printf("Error: failed to allocate memory for p0\n");
               exit(1);}
      }

      p1 = (double **)malloc(sizeof(double *) * nrows);
      if(p1 == NULL){
            printf("Error: failed to allocate memory for p1\n");
            exit(1);
      }
    for(i = 0; i < nrows; i++){
            p1[i] = (double *)malloc(sizeof(double) * ncols);
            if(p1[i] == NULL){
                  printf("Error: failed to allocate memory for p1\n");
                  exit(1);
            }
      }

      /*difference equations*/
      deltaP = (double **)malloc(sizeof(double *) * nrows);
      if(deltaP == NULL){
            printf("Error: failed to allocate memory for deltaP\n");
            exit(1);
      }
    for(i = 0; i < nrows; i++){
            deltaP[i] = (double *)malloc(sizeof(double) * ncols);
            if(deltaP[i] == NULL){
                  printf("Error: failed to allocate memory for
deltaP\n");
                  exit(1);
            }
      }

      deltaQ = (double **)malloc(sizeof(double *) * nrows);
      if(deltaQ == NULL){
            printf("Error: failed to allocate memory for deltaQ\n");
            exit(1);
      }
    for(i = 0; i < nrows; i++){
```

```
            deltaQ[i] = (double *)malloc(sizeof(double) * ncols);
            if(deltaQ[i] == NULL){
                  printf("Error: failed to allocate memory for
deltaQ\n");
                  exit(1);
            }
      }

      /*conditional probabilities*/
      q0 = (double **)malloc(sizeof(double *) * nrows);
      if(q0 == NULL){
            printf("Error: failed to allocate memory for q0\n");
            exit(1);
      }
    for(i = 0; i < nrows; i++){
            q0[i] = (double *)malloc(sizeof(double) * ncols);
            if(q0[i] == NULL){
                  printf("Error: failed to allocate memory for q0\n");
                  exit(1);
            }
      }

      q1 = (double **)malloc(sizeof(double *) * nrows);
      if(q1 == NULL){
            printf("Error: failed to allocate memory for q1\n");
            exit(1);
      }
    for(i = 0; i < nrows; i++){
            q1[i] = (double *)malloc(sizeof(double) * ncols);
            if(q1[i] == NULL){
                  printf("Error: failed to allocate memory for q1\n");
                  exit(1);
            }
      }

      /* Initialise all algorithm matrices to parity check matrix */
      ResetMatrix(matrix, p0, nrows, ncols);
      ResetMatrix(matrix, p1, nrows, ncols);
      ResetMatrix(matrix, q0, nrows, ncols);
      ResetMatrix(matrix, q1, nrows, ncols);
      ResetMatrix(matrix, deltaP, nrows, ncols);
      ResetMatrix(matrix, deltaQ, nrows, ncols);


      //Calculate the Guassian density probabilities
      for(i = 0; i < ncols; i++)
            f1[i] = 1 /(1+exp((-2*codeword[i])/pow(ch_dev,2)));

      for(i = 0; i < ncols; i++)
            f0[i] = 1 - f1[i];

            /*calculate the q1ij probabilities*/
      for(i = 0; i < nrows; i++){
            for(j = 0; j < ncols; j++){
                  if(q1[i][j] != 0)
```

```
                q1[i][j] *= f1[j];
        }
}

//calculate the q0ij probabilities
for(i = 0; i < nrows; i++){
        for(j = 0; j < ncols; j++){
                if(q0[i][j] != 0)
                        q0[i][j] *= f0[j];
        }
}

/*Commence decoding iterations*/
do{

        //Convert to hard decision quantized form
        HardDecision(decoded, codeword, ncols);

        //Check if codeword passes syndrome test
        syndrome = SyndromeTest(matrix, decoded, nrows, ncols);

        //Calculate the probability deltas
        for(i = 0; i < nrows; i++){
                for(j = 0; j < ncols; j++){
                        deltaQ[i][j] = q0[i][j] - q1[i][j];
                }
        }

        for(i = 0; i < nrows; i++){
                for(j = 0; j < ncols; j++){
                        if(deltaP[i][j] != 0){
                                for(k = 0; k < ncols; k++){
                                        if(j != k && deltaP[i][k] != 0){
                                                deltaP[i][j] *= deltaQ[i][k];
                                        }
                                }
                        }
                }
        }

        //Calculate p values
        for(i = 0; i < nrows; i++){
                for(j= 0; j < ncols; j++){
                        if(p0[i][j] != 0){
                                p0[i][j] *= (0.5 * (1 + deltaP[i][j]));
                                p1[i][j] *= (0.5 * (1 - deltaP[i][j]));
                        }
                }
        }

        //calculate the coefficients
        for(i = 0; i < ncols; i++){
                for(j = 0; j < nrows; j++){
                        if(matrix[j][i] != 0){
                                coeff0 = 1.0;
```

```
                        coeff1 = 1.0;
                        for(k = 0; k < nrows; k++){
                              if(j != k && matrix[k][i] != 0){
                                    coeff0 *= p0[k][i];
                                    coeff1 *= p1[k][i];
                              }
                        }
                        coeff0 *= f0[i];
                        coeff1 *= f1[i];
                        q0[j][i] = coeff0 /(coeff0 + coeff1);
                        q1[j][i] = coeff1 /(coeff0 + coeff1);
                  }
            }
      }

      //Generate new soft codewords for each bit
      for(i = 0; i < ncols; i++){
            soft0[i] = 1;
            soft1[i] = 1;
            for(j = 0; j < nrows; j++){
                  if(p0[j][i] != 0){
                        soft0[i] *= p0[j][i];
                        soft1[i] *= p1[j][i];
                  }
            }
            soft0[i] *= f0[i];
            soft1[i] *= f1[i];
            //Hard code the new codeword
            if(soft1[i] > soft0[i])
                  decoded[i] = 1;
            else
                  decoded[i] = 0;
      }

      //Reset matrices
      ResetMatrix(matrix, deltaP, nrows, ncols);
      ResetMatrix(matrix, deltaQ, nrows, ncols);
      ResetMatrix(matrix, p0, nrows, ncols);
      ResetMatrix(matrix, p1, nrows, ncols);

      iteration++;
} while( iteration <= MAX_ITERATION && syndrome != 1);


//Free allocated memory
FreeMatrix(p0, nrows);
FreeMatrix(p1, nrows);
FreeMatrix(deltaP, nrows);
FreeMatrix(deltaQ, nrows);
FreeMatrix(q0, nrows);
FreeMatrix(q1, nrows);
free(soft0);
free(soft1);

return syndrome;
```

```
}

int OneStepMajorityLogic(int **matrix, double *codeword, int *decoded,
int nrows, int ncols){
 //   int iteration = 1;            /*Iteration counter*/
      //int syndrome = 0;           /*Syndrome test status flag*/
      //int col_wt = 0;             /*Parity check matrix column
weight*/
      //int err_wt = 0;             /*Error column weight*/
      //int pass = 0;               /*Orthogonal pass flag*/
      //int ***Errors = NULL; /*3d Error matrix*/
      //int **Used = NULL;          /*Tracking matrix*/
      //int *Erows = NULL;          /*Number of rows in each Error
matrix*/
      //int i, j, k, l, m, n; /*Local counter variables*/

      //printf("**********Entering One Step Majority Logic
Decoder**********\n");       /*DELETE AFTER TESTING*/

      ////Create Error matrix
      //Errors = (int ***)malloc(sizeof(int **) * ncols);
      //if(Errors == NULL){
      //    printf("Error: failed to allocate memory for Error
matrix\n");
      //    exit(1);
      //}
      //for(i = 0; i < ncols; i++){
      //    Errors[i] = (int **)malloc(sizeof(int *) * nrows);
      //    if(Errors[i] == NULL){
      //          printf("Error: failed to allocate memory for Error
Matrix row\n");
      //          exit(1);
      //    }
      //    for(j = 0; j < nrows; j++){
      //          Errors[i][j] = (int *)malloc(sizeof(int) * ncols);
      //          if(Errors[i][j] == NULL){
      //                printf("Error: failed to allocate memory for
Error Matrix column\n");
      //                exit(1);
      //          }
      //    }
      //}

      ////Create tracking matrix
      //Used = (int **)malloc(sizeof(int *) * nrows);
      //if(Used == NULL){
      //    printf("Error: failed to allocate memory for Used
matrix\n");
      //    exit(1);
      //}
 //   for(i = 0; i < nrows; i++){
      //    Used[i] = (int *)malloc(sizeof(int) * ncols);
      //    if(Used[i] == NULL){
      //       printf("Error: failed to allocate memory for Used matrix
row\n");
```

```
      //       exit(1);}
      //}

      ////Initialise tracking matrix to the parity check matrix
      //for(i = 0; i < nrows; i++){
      //    for(j = 0; j < ncols; j++)
      //          Used[i][j] = matrix[i][j];
      //}

      ////Create Error rows vector
      //Erows = (int *)malloc(sizeof(int) * ncols);
      //if(Erows == NULL){
      //    printf("Error: failed to allocate memory for Error rows
vector\n");
      //    exit(1);
      //}

      ////Transfer Parity check matrix base vectors to Error matrix
      //for(i = 0; i < ncols; i++){
      //    col_wt = 0;
      //    for(j = 0; j < nrows; j++){
      //          if(matrix[j][i] == 1){
      //                col_wt++;
      //                for(k = 0; k < ncols; k++){
      //                      Errors[i][col_wt-1][k] = matrix[j][k];
      //                }
      //          }
      //    }
      //    Erows[i] = col_wt;
      //}

      ////Make Error vectors orthogonal
      //for(i = 0; i < ncols; i++){  //for each bit in the codeword
      //    do{
      //          for(j = 0; j < Erows[i]; j++){
      //for each row in the error matrix
      //                for(k = 0; k < ncols; k++){
      //for each bit in the Error matrix row
      //                      if( k != i){
      //do not check the same as the current codeword bit
      //                            if(Errors[i][j][k] == 1){
      //check if the bit is a 1
      //                                  err_wt++;
      //keep the total of ones in this Error matrix column
      //                            }
      //                            if(err_wt > 1){
      //if more than one 1
      //                                  for(l = 0; l < nrows; l++){
      //Find a row in the Used matrix where
      //                                        if(Used[l][i] == 0 &&
Used[l][k] == 1){ //the row has not been used and has a 1 in the current
column
      //                                              //Add the Used row
to the Error row using binary addition
```

```
      //                                                for(m = 0; m <
ncol; m++){
      //
      Errors[i][j][m] = XOR(Errors[i][j][m],Used[l][m]);
      //                                                }
      //                                                //Set the Used
matrix current codeword bit to a 1 to indicate the row has been used
      //                                                Used[l][i] = 1;
      //                                            }
      //                                        }
      //                                }

      //                        }
      //                }
      //                err_wt = 0;
      //          }
      //    }while(pass != 1);
      //}

      ////Display Error Matrix DELETE AFTER TEST
      //for(i = 0; i < ncols; i++){
      //    printf("\n\n%d:\n", i+1);
      //    for(j = 0; j < Erows[i]; j++){
      //          for(k = 0; k < ncols; k ++){
      //                printf("%d ", Errors[i][j][k]);
      //          }
      //          printf("\n");
      //    }
      //}
      //


      //do{
      //
      //iteration++;
      //}while( iteration <= MAX_ITERATION && syndrome != 1);

      //
      //return syndrome;
return 0;
}

/*************** Common Methods **********************/
//Display the matrix
void PrintIntMatrix(int **matrix, int nrows, int ncols){
   int i, j;

   for(i = 0; i < nrows; i++){
      for(j = 0; j < ncols; j++){
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
   }
}
```

```
//Display a matrix
void PrintMatrix(double **matrix, int nrows, int ncols){
   int i, j;

   for(i = 0; i < nrows; i++){
      for(j = 0; j < ncols; j++){
         printf("%2.4f ", matrix[i][j]);
      }
      printf("\n");
   }
   printf("\n\n");
}
//Output code to file
void FileOutput(FILE *fp, char type, double *codeword, int ncols){
   int i;
   fprintf(fp, " %c: ", type);
   for(i = 0; i < ncols; i++){
      fprintf(fp, "%d", (int)codeword[i]);
   }
}
//Display the message
void PrintCodeword(double *codeword, int length){
   int i;

   for(i = 0; i < length; i++){
      printf("%2.4lf ", codeword[i]);
   }
   printf("\n");
}

void PrintIntCodeword(int *codeword, int length){
   int i;

   for(i = 0; i < length; i++){
      printf("%d      ", codeword[i]);
   }
   printf("\n");
}

//Free matrix memory
void FreeMatrix(double **matrix, int nrows){
   int i;

   for(i = 0; i < nrows; i++)
      free(matrix[i]);

   free(matrix);
}

void FreeIntMatrix(int **matrix, int nrows){
   int i;

   for(i = 0; i < nrows; i++)
      free(matrix[i]);
```

```
   free(matrix);
}

//Calculate the BER for the signal
double BitErrorRate(int *original, int *current, int length){
   int errors = 0;
   int i;

   for(i = 0; i < length; i++){
         if(current[i] != original[i])
                  errors++;
   }

   return((double)errors / (double)length);
}

//Conduct syndrome test for cH = 0
int SyndromeTest(int **matrix, int *codeword, int nrows, int ncols){
      int pass = -1;
      int rowsum = 0;
      int multi = 0;
      int i, j;

   for(i = 0; i < nrows; i++){
      for(j = 0; j < ncols; j++){
               multi += (codeword[j] * matrix[i][j]);
        }
        rowsum = (multi % 2);

        if(rowsum > 0)
              pass = 1;
   }
      return pass;
}


//Add AWGN noise using Box-Muller method to codeword
void AddWhiteGaussianNoise(int *codeword, double *received, int length,
double variance, double mean){
     double s1, s2, rand_n, r, noise;
     int i;

       srand((int) time(NULL));

     for(i = 0; i < length; i++){
            do {
                  rand_n = (rand() / (double)RAND_MAX);
                  s1 = rand_n * 2.0 - 1.0;
                  rand_n = (rand() / (double)RAND_MAX);
                  s2 = rand_n * 2.0 - 1.0;

                  r = s1 * s1 + s2 * s2;

            } while( r >= 1.0 || r == 0.0);
```

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

            received[i] = codeword[i] + noise;
      }
}

//Convert message bits to either 0 or 1
void HardDecision(int *hardcode, double *softcode, int length){
   int i;

   for(i = 0; i < length; i++){
         if (softcode[i] <= MID)
               hardcode[i] = LOW;
         else
               hardcode[i] = HIGH;
   }
}

//Zeroise matrix data
void ZeroMatrix(double **matrix, int nrows, int ncols){

      int i, j;

      for(i = 0; i < nrows; i++){
            for(j = 0; j < ncols; j++)
                  matrix[i][j] = 0;
      }
}
//Zero a double type vector
void ZeroVector( double *vector, int ncols){
      int i;

      for(i = 0; i < ncols; i++)
            vector[i] = 0.0;
}
//Zero an int vector
void ZeroIntVector( int *vector, int ncols){
      int i;

      for(i = 0; i < ncols; i++)
            vector[i] = 0;
}

void ResetMatrix(int **pchk_matrix, double **matrix, int nrows, int
ncols){
      int i, j;

      for(i = 0; i < nrows; i++){
            for(j = 0; j < ncols; j++)
                  matrix[i][j] = (double)pchk_matrix[i][j];
      }
}

void BSCtoBKSP(int *codeword, int ncols){
```

```
	int i;

	for(i = 0; i < ncols; i++){
		if(codeword[i] < 0.5)
			codeword[i] = -1;
		else
			codeword[i] = 1;
	}
}

void BKSPtoBSC(int *codeword, int ncols){
	int i;

	for(i = 0; i < ncols; i++){
		if(codeword[i] < 0)
			codeword[i] = 0;
		else
			codeword[i] = 1;
	}
}

int XOR(int a, int b){
	return (a == b) ? 0 : 1;
}
```

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>



// TODO: reference additional headers your program requires here


//random.c: Initial code to generate random LDPC codes


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

#define MAX_CYCLE 100

void printMatrix(int **matrix, int rows, int cols);
```

```
void zeroMatrix(int **matrix, int rows, int cols);

int main(void){
   int i, j;
   int a, b, c;
   int n, k;
   int code_len = 0;
   int message_len = 0;
   int col = 0;
   int row = 0;
   int col_weight = 0;
   int row_weight = 0;
   int rows = 0;
   float rate = 0.0;
   int numcodes = 1;
   int remainder = 0;
   int result = 0;
   int cur_colweight = 0;
   int cur_rowweight = 0;
   int commonbit = 0;
   int index = 0;
   int random_idx = 0;
   int type = 0;  //0 = regular 1 = irregular
   int addcol = 0;
   int cycle = 0;
   int chkcol = 0;
   int chkrow = 0;

   printf("Enter code length:");
   scanf("%d", &code_len);

   printf("Enter message length:");
   scanf("%d", &message_len);

   //calculate rate
   rate = message_len / code_len;

   printf("Enter column weight:");
   scanf("%d", &col_weight);

   //Calculate the number of rows required
   rows = code_len - message_len;

   //Ensure column weight is less than message length
   if(col_weight > rows){
      printf("Error: colweight can not be larger than message length");
      exit(0);
   }

   //Determine if H will be regular or irregular
   if((code_len % rows) == 0)
      type = 0;
   else
      type = 1;
```

```
// Calculate row_weight
//If regular  all rows = row_weight else
// Top half of H = row_weight + 1
// Bottom half of H = row_weight
row_weight = (int)(col_weight * code_len)/rows;
printf("Calc row weight = %d\n", row_weight);
//Allocate memory space for parity check matrix
int **pchk_matrix = malloc(rows * sizeof(int *));
for(i = 0; i < rows; i++){
   pchk_matrix[i] = malloc(code_len * sizeof(int *));
}

//Determine candidate submatrices of H
//Create an array for the candidate binary tuple
float *candidate = malloc(rows * sizeof(float));

//Determine the total number of possible codewords
numcodes = pow(2,rows);

//Create a matrix of candidate tuples
float **candidate_matrix = malloc(numcodes * sizeof(float *));
//Check each n tuple as a candidate
for(i = 0; i < numcodes; i++){
   result = i;
   //Convert int to binary array
   for(j = rows - 1; j >= 0; j--){
      remainder = result % 2;
    result = result / 2;
    //Load candidate array with binary representation
    candidate[j] = remainder;
    //Count the number of information bits
    if(remainder == 1){
       cur_colweight++;
    }
   }
   //Reset remainder
   remainder = 0;
   //If the number of info bit = column weight n tuple is a candidate
   if(cur_colweight == col_weight){
      //Add candidate tuple to the candidate matrix
      candidate_matrix[index] = malloc(rows * sizeof(float *));
    for(j = 0; j < rows; j++){
       candidate_matrix[index][j] = candidate[j];
    }
    index++;
   }
   //Reset column weight counter
   cur_colweight = 0;
}
printf("There are %d rows\n", rows);
//Ensure all nodes are zero
zeroMatrix(pchk_matrix, rows, code_len);

//Seed random number
srand(time(NULL));
```

```
   //random code construction
   for(i = 1; i <= code_len; i++){
      //First column all zeros
      //For each column choose a number that has a binary representation
equal to colweight as a candidate
      do{

        random_idx = rand() % index;
       chkcol = 1;
       chkrow = 0;
       addcol = 0;

       //Check that the candidate has only one 1 in common with any
other column
       for(a = 0; a < i; a++){
          commonbit = 0;
          for(b = 0; b < rows; b++){
             if(pchk_matrix[b][a] != 0 &&
candidate_matrix[random_idx][a] != 0)
                commonbit++;
          }
          if(commonbit > 1){
             chkcol = 0;
             break;
          }else{
             chkcol = 1;
          }
       }

       //Check matrix row weight does not exceed required row weight
       for(b = 0; b < rows; b++){
          cur_rowweight = 0;
          //printf("Row: %d ", b);
          for(a = 0; a < i; a++){

             cur_rowweight +=(int)pchk_matrix[b][a];
            // printf("%d ", (int)pchk_matrix[b][a]);

          }
          cur_rowweight += (int)candidate_matrix[random_idx][a];
          //printf(" mrw = %d ", cur_rowweight);

         // printf(" cm %d ", (int)candidate_matrix[random_idx][a]);
          //printf(" crw = %d ", cur_rowweight);
          if(cur_rowweight > row_weight){
            // printf("no\n");
             chkrow = 0;
             break;
          } else {
            //printf("ok\n");chkcol != 1 &&
             chkrow = 1;
          }
       }
       if (chkrow == 1)
```

```
            addcol = 1;

         } while (addcol == 0);

         for(j = 0; j < rows; j++){
            pchk_matrix[j][i-1] = candidate_matrix[random_idx][j];
         }
         printf("add col: %i\n", i);
         printMatrix(pchk_matrix, rows, code_len);

      }

      for(i = 0; i < rows; i++){
         free(pchk_matrix[i]);
      }
      for(i = 0; i < index; i++){
         free(candidate_matrix[i]);
      }
      free(candidate);
      exit(0);
}

      void zeroMatrix(int **matrix, int rows, int cols){
         int i, j;

         for(i = 0; i < rows; i++){
            for(j = 0; j < cols; j++){
               matrix[i][j] = 0;
            }
         }
      }


void printMatrix(int **matrix, int rows, int cols){
      int i, j;

      for(i = 0; i < rows; i++){
         for(j = 0; j < cols; j++){
            printf("%d ", matrix[i][j]);
         }
         printf("\n");
      }

}
```