A Comparison Study of LDPC and BCH Codes

By

Wong Hui Yin

Dissertation submitted in partial fulfillment of

the requirements for the

Bachelor of Engineering (Hons)

(Electrical & Electronics Engineering)

DECEMBER 2006

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

A Comparison Study of LDPC and BCH Codes

by

Wong Hui Yin

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfillment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRICAL & ELECTRONICS ENGINEERING)

Approved by,

[illegible]

(Mr. Azizuddin Abd. Aziz)
Project Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

December 2006

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

yin.

WONG HUI YIN

840222 – 13 – 5122

Matric ID: 3967

ABSTRACT

The need for efficient and reliable digital data communication systems has been rising rapidly in recent years. There are various reasons that have brought this need for the communication systems, among them are the increase in automatic data processing equipment and the increased need for long range communication. Therefore, the LDPC and BCH codes were developed for achieving more reliable data transmission in communication systems. This project covers the research about the LDPC and BCH error correction codes. Algorithm for simulating both the LDPC and BCH codes were also being investigated, which includes generating the parity check matrix, generating the message code in Galois array matrix, encoding the message bits, modulation and decoding the message bits for LDPC. Matlab software is used for encoding and decoding the codes. The percentage of accuracy for LDPC simulation codes are ranging from 95% to 99%. The results obtained shows that the LDPC codes are more efficient and reliable than the BCH codes coding method of error correction because the LDPC codes had a channel performance very close to the Shannon limit. LDPC codes are a class of linear block codes that are proving to be the best performing forward error correction available. Markets such as broadband wireless and mobile networks operate in noisy environments and need powerful error correction in order to improve reliability and better data rates. Through LDPC and BCH codes, these systems can operate more reliably, efficiently and at higher data rates.

ACKNOWLEDGEMENTS

Completion of this final year project would not have been possible without the assistance and guidance of certain individuals. Their contribution both technically and mentally is highly appreciated.

Firstly, I would like to express my sincere and utmost appreciation to my supervisor, Mr Azizuddin Abd Aziz for his guidance, advice and commitment throughout the process of conducting the final year project.

Thanks are extended to the UTP Electrical and Electronics department FYP committee, Ms Nasreen Badruddin and Ms. Siti Hawa Tahir for their cooperation on guiding me throughout this project period.

Last but not least, I would like to thank all persons who have contributed to this project but have been inadvertently not mentioned.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

AWGN	:	Additive white Gaussian noise
BCH	:	Bose Chaudhuri Hocquenghem
BER	:	Bit Error Rate
BPSK	:	Binary Phase-shift Keying
BMA	:	Berlekamp – Massey algorithm
DNS	:	Domain Name System
EA	:	Euclidean algorithm
GF	:	Galois Field
LDPC	:	Low-density parity-check
JPEG	:	Joint Photographic Experts Group
MPEG	:	Moving Picture Experts Group
NASA	:	National Aeronautics and Space Administration
SNR	:	Signal – to – noise
TCP	:	Transmission Control Protocol
UDP	:	User Datagram Protocol

CHAPTER 1
INTRODUCTION

1.1 BACKGROUND OF STUDY

When binary data are transmitted electronically, in telecommunications, computers, and CD players, errors may occur in the binary digits or bits. In order to correct them, extra digits are sent along with the information digits, so message sequences are encoded to longer codeword sequences. Codeword with errors may be corrected to the original codeword for a small number of errors and then decoded to the original message sequence.

The Bose, Chaudhuri, and Hocquenghem (BCH) codes form a large class of powerful random error – correcting cyclic codes. This class of codes is the generalization of the Hamming codes for multiple error correction. BCH codes were discovered by Hocquenghem in 1959 and independently by Bose and Chaudhuri in 1960. BCH codes were generalized to codes in p^m symbols by Gorenstein and Zierler in 1961. [37]

The first decoding algorithm for BCH codes were devised by Peterson in 1960. After that, Peterson's algorithm was generalized and refined by Gorenstein and Zierler, Chien, Forney, Berlekamp, Massey, Burton and others. Among all the decoding algorithms for BCH codes, Berlekamp's iterative algorithm, and Chien's search algorithm are the most efficient ones.

Low-density parity-check (LDPC) codes were invented by Gallager in 1962, but were nearly forgotten for more than 30 years. Gallager had proposed an interactive decoding scheme, which is now known as belief propagation decoding algorithm and presented the analysis of decoding performance of LDPC codes. After the introduction of turbo codes by Berrou et al, LDPC codes were rediscovered by Mackay in 1996, and were shown to demonstrate the properties of LDPC codes capable of asymptotically approaching Shannon limit. [3]

The Noisy Channel Coding Theorem discovered by C. E. Shannon in 1948 offered communication engineers the possibility of reducing error rates on noisy channels to negligible levels without sacrificing data rates. This respective theorem establishes that however contaminated with noise interference a communication channel may be, it is possible to communicate digital data or information error-free up to a given maximum rate through the channel. The theorectical maximum information transfer rate of the channel is regarded as Shannon limit. [4]

LDPC was the first code to allow data transmission rates close to the theoretical maximum, the Shannon Limit. Impractical to implement when developed in 1963, LDPC was forgotten. The next 30 or so years of information theory failed to produce anything one-third as effective and LDPC remains, in theory, the most effective developed to date (2006).

The explosive growth in information technology has produced a corresponding increase of commercial interest in the development of highly efficient data transmission codes as such codes impact everything from signal quality to battery life. Although implementation of LDPC codes has lagged that of other codes, notably the turbo code, the absence of encumbering software patents has made LDPC attractive to some and LDPC codes are positioned to become a standard in the developing market for highly efficient data transmission methods. In 2003 an LDPC code beat six turbo codes to become the new standard for the satellite transmission of digital television.

1.2 PROBLEM STATEMENT

We study two families of error-correcting codes, which are BCH and LDPC codes. For LDPC codes, the MacKay's codes are recently invented, and "Gallager codes" were first investigate in 1962, but appear to have been largely forgotten, in spite of their excellent properties.

Comparison studies for LDPC and BCH codes are performed. This is because the codes could be implemented in different areas, depending on the application constraints such as near capacity performances (Shannon limit), complexity, time for decoding, cost for implementing the code on hardware, and transmission capacity. Therefore, proper applications for the error correction codes can be implemented in order to fit the project's criteria, such as simulation time and cost. Comparisons study of both codes involves source codes algorithms for LDPC and BCH codes, error performances, simulation time and others.

This project is to study the importance of LDPC and BCH codes, and the difference between both codes. Some of the error correction schemes are computationally intensive, or require excessive redundant data which may be inhibitive for a certain application. In some cases, cost and simulation time are the main constraint for the projects, therefore, the error correction codes are being compared in order to satisfy the relative applications or projects' requirements and constraints.

For example, the satellite data transmissions require more detailed and complex error corrections codes compared with the network transmissions and computer file transfers, therefore, the higher capacity performance code such as LDPC code can be applied to the satellite transmissions. LDPC code can also be applied in digital video broadcasting (DVB-S2), and WiMAX, which is an IEEE standard for microwave communication. The less complex codes such as BCH codes can be implemented in digital television, mobile communications and storage devices such as compact disk in order to save time, reduce cost and reduce application's complexity.

The comparison study of both codes performed involved error performances, simulation time, effect of noise variance, and effect of data rates on the bit error rate (BER). The comparison study is important in order that proper applications for error correction codes could be implemented based on the projects' constraints such as time for decoding, cost, and codes' complexity.

1.2.1 Importance of error correction codes

The need for efficient and reliable digital data communication systems has been rising rapidly in recent years. There are various reasons that have brought this need for the communication systems, among them are the increase in automatic data processing equipment and the increased need for long range communication. The data systems developed through the use of conventional modulation and voice transmission techniques have generally resulted in systems with relatively low data rates and high error probabilities. Therefore, the LDPC codes were developed for achieving more reliable data transmission in communication systems.

The significant application that requires the error correction codes are Internet, deep space communications, and satellite broadcasting.

1.2.2 Applications for error correction codes

Internet

In a typical TCP/IP stack, error detection is performed at multiple levels. Each Ethernet frame carries a CRC-32 checksum. The receiver discards frames if their checksums do not match. Ethernet is a frame-based computer networking technology for local area networks (LANs). A checksum is a form of redundancy check, a very simple measure for protecting the integrity of data by detecting errors in data that is sent through space (telecommunications) or time (storage). A redundancy check is extra data added to a message for the purposes of error detection and error correction. [5]

User Datagram Protocol (UDP) has an optional checksum. Packets with wrong checksums are discarded. Common network applications that use UDP include the Domain Name System (DNS), for example, http://www.elearning.edu.my, streaming media applications, Voice over IP, Trivial File Transfer Protocol (TFTP), and online games.

Transfer Control Protocol (TCP) has a checksum of the payload, TCP header and source and destination addresses of the IP header. Packets found to have incorrect checksums are discarded and eventually get retransmitted when the sender receives a triple-ack or a time-out occurs. Using TCP, applications on networked hosts can create connections to one another, over which they can exchange data or packets. The protocol guarantees reliable and in-order delivery of sender to receiver data. TCP also distinguishes data for multiple, concurrent applications, for instance Web server and email server, where they are running on the same host. TCP supports many of the internet's most popular application protocols and resulting applications, including the World Wide Web, email and Secure Shell.

Deep Space Telecommunications

NASA has used many different error correcting codes. For missions between 1969 and 1977 the Mariner spacecraft used a Reed-Muller code. The noise these spacecraft were subject to was well approximated by a "bell-curve" (normal distribution), so the Reed-Muller codes were well suited to the situation. [5]

The standard normal distribution is the normal distribution with a mean of zero and a standard deviation of one. It is often called the bell curve because the graph of its probability density resembles a bell.

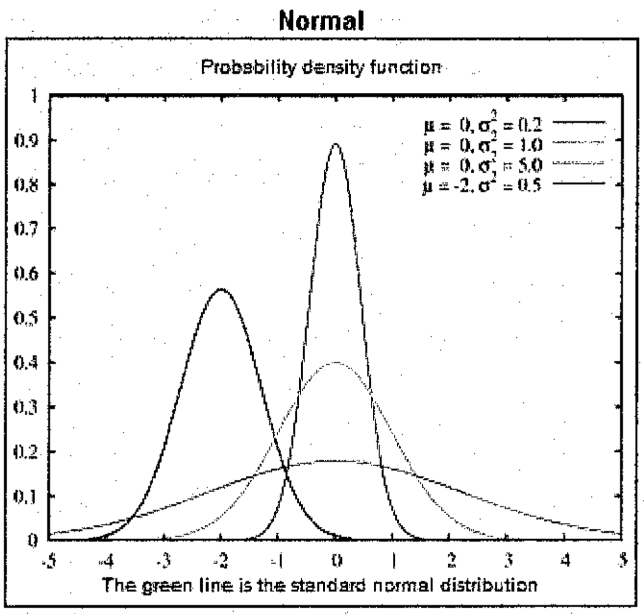


Figure 1 : Bell curve

The Voyager 1 & Voyager 2 spacecraft transmitted color pictures of Jupiter and Saturn in 1979 and 1980.

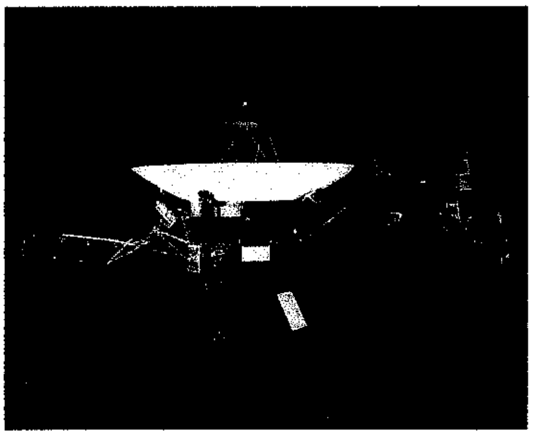


Figure 2 : Voyager Spacecraft

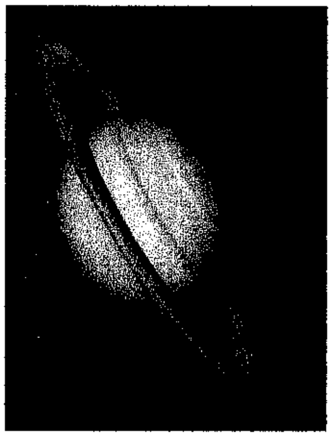


Figure 3 : Saturn taken by Vogager

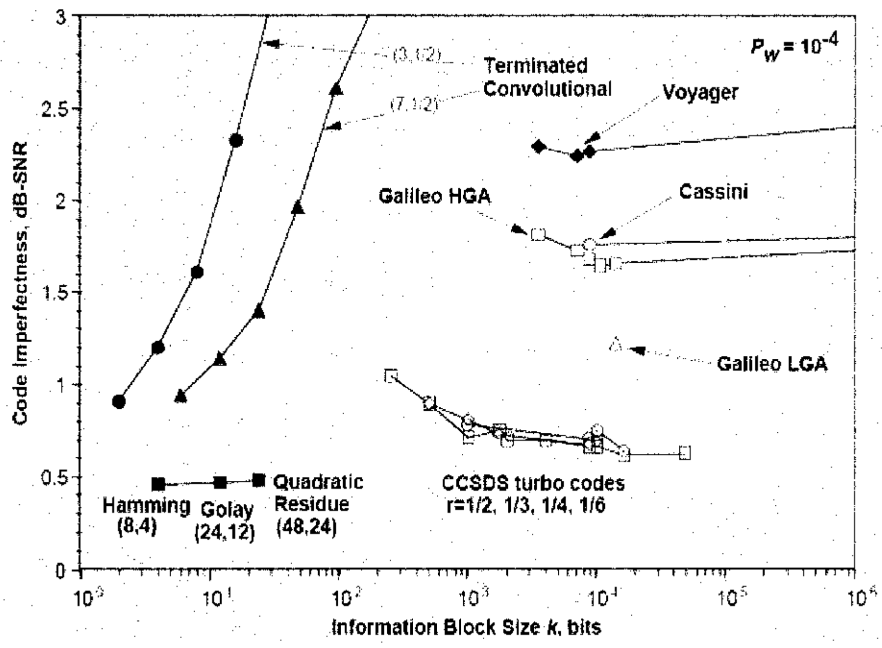


Figure 4 : NASA's Deep Space Missions ECC Codes (code imperfectness)

Color image transmission required 3 times the amount of data, so the Golay (24,12,8) code was used. This Golay code is only 3-error correcting, but it could be transmitted at a much higher data rate. Voyager 2 went on to Uranus and Neptune and the code was switched to a concatenated Reed-Solomon code-Convolutional code for its substantially more powerful error correcting capabilities. Current DSN error correction is done with dedicated hardware. For some NASA deep space craft such as those in the Voyager program, Cassini-Hyugens (Saturn), New Horizons (Pluto) and

Deep Space 1. The use of hardware ECC may not be feasible for the full duration of the mission. A solution to the hardware-software error correction problem exists called "Deep Space Network @ Home". The different kinds of deep space and orbital missions that are conducted suggest that trying to find a "one size fits all" error correction system will be an ongoing problem for some time to come. For missions close to the earth the nature of the "noise" is different from that on a spacecraft headed towards the outer planets. In particular, if a transmitter on a spacecraft far from earth is operating at a low power, the problem of correcting for noise gets larger with distance from the earth. [5]

Satellite Broadcasting

The demand for satellite transponder bandwidth continues to grow, fueled by the desire to deliver television, including new channels and High Definition TV and IP data. An automatic device that receives, amplifies, and retransmits a signal on a different frequency. Transponder availability and bandwidth constraints have limited this growth, because transponder capacity is determined by the selected modulation scheme and Forward Error Correction (FEC) rate. Forward error correction (FEC) is a system of error control for data transmission. [5]

Scientific-Atlanta, which is now part of Cisco Systems, has been evaluating developing products based on Turbo Codes concatenated with minimal complexity Reed-Solomon Codes in its laboratories in Atlanta, Georgia and Toronto, Canada.

1.2.3 Significant of the project

The low-density parity-check (LDPC) and Bose Chaudhuri Hocquenghem (BCH) codes are error correcting codes. These codes are a method of transmitting message over a noisy transmission channel. In computer science and information theory, the issue of error correction and detection has great practical importance. The error detection is the ability to detect errors that are made due to noise or other impairments during the transmission from the transmitter to the receiver. Error correction has the feature of enabling localization of the errors and correcting them.

This project will introduce the comparison study for the LDPC and BCH codes. Moreover, this project also provides the encoding and decoding techniques which would be simulated by using Matlab simulation tool.

The comparison study of both codes performed involved error performances, simulation time, effect of noise variance, and effect of data rates on the bit error rate (BER). The comparison study is important in order that proper applications for error correction codes could be implemented based on the projects' constraints such as time for decoding, cost, and codes' complexity.

1.3 OBJECTIVES

- Investigate and do research on the LDPC and BCH codes.
- Investigate the algorithm for generating the parity check matrix, encoding the message bits, modulation and decoding the message bits for LDPC and BCH codes.
- Implement the encoding and decoding simulations for the LDPC and BCH codes using the Matlab software.
- Conduct and generate the results using Matlab software simulations.
- Conduct a comparison performance study for LDPC and BCH codes.

1.4 SCOPE OF STUDY

This project covers the research about the LDPC and BCH codes. Matlab will be used for encoding and decoding the LDPC and BCH codes. This study would enhance our knowledge on the LDPC and BCH error correcting code, where we learned the method of transmitting a message over a noisy transmission channel. The codes could minimize the probability of lost information transmitted. The Matlab software is used for simulating the encoding and decoding for both the error correcting codes.

CHAPTER 2
LITERATURE REVIEW

2.1 SUPPORTING INFORMATION

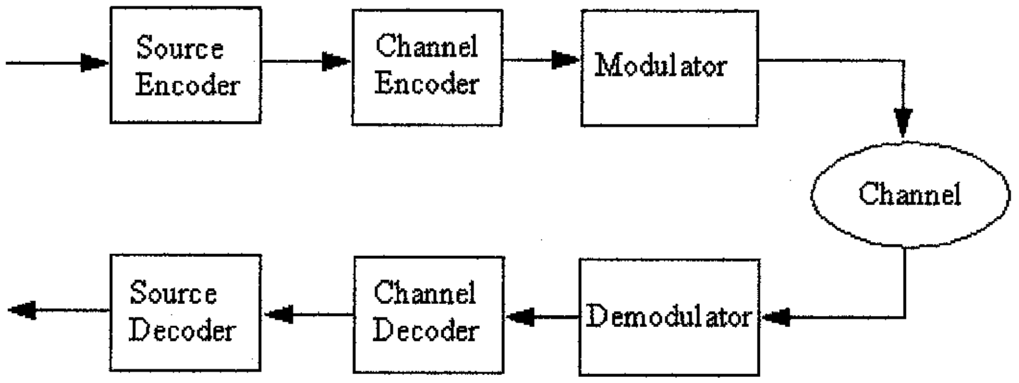


Figure 5 : Block diagram of a general communication system

Coding is the conversion of information to another form. From Figure 5, source coding is conducted for lowering the redundancy in the information, for example ZIP, JPEG, and MPEG2. The channel coding is to defeat the channel noise. The application of redundant symbols to correct data errors could be implemented by channel encoding. Modulation is the conversion of symbols to a waveform for transmission. The conversion of the waveform back to symbols is done by demodulation. The decoding uses the redundant symbols to correct errors. Several parameters for code performance evaluations are code rate (R), Signal – to – noise ratio (Eb/No), and Bit Error Rate (BER). The coding gain is the saving in Eb/No required to achieve a given BER when coding is used vs. that with no coding. Generally, the lower the code rate, the higher the coding gain. Better codes provides better coding gains, however, they are usually more complicated and have higher complexity, for instance LDPC codes. [6]

2.2 LDPC CODES

Low-density parity-check (LDPC) codes are a class of linear block codes. This code contains the parity-check matrix which contains only a few 1's in comparison to the amount of 0's. Basically there are two different possibilities to represent LDPC codes. They can be described either via matrices or using the graphical representation. There are several different algorithms for constructing suitable LDPC codes. The semi-randomly generate sparse parity check matrices was proposed by Mackay. There are several algorithm used to decode LDPC codes, the most common one are belief propagation algorithm, the message passing algorithm and the sum-product algorithm.

The parity-check codes use linear sums of the information bits, called parity symbols or parity bits, for error detection or correction. A single parity check code is constructed by adding a single parity bit to a block of data bits. The parity bit takes on the value of one or zero as needed to ensure that the summation of all the bits in the codeword yields an even or odd result. The summation operation is performed using modulo-2 arithmetic. If the added parity is designed to yield an even result, the method is termed odd parity.

The matrix H is defined as the parity-check matrix, which will enable us to decode the received vectors. For each (k x n) generator matrix G, there exists an (n-k) x n matrix H, such that the rows of G are orthogonal to the rows of H; that is, $GH^T = 0$, where H^T is the transpose of H, and 0 is a k x (n-k) all-zeros matrix. H^T is a matrix whose rows are columns of H and whose columns are the rows of H.

The product UH^T yields the result $UH^T = 0$. Thus, once the parity-check matrix H is constructed to fulfill the foregoing orthogonality requirements, it can be used to test whether the received vector is a valid number of the codeword set U. U is a codeword generated by G if $UH^T = 0$. [7]

2.2.1 Gallager's decoding scheme for LDPC

Maximum-likelihood decoding is a convenient concept for decoding the LDPC codes as it minimizes the probability of decoding error and more effective than other decoding scheme. However, the maximum-likelihood decoder's disadvantage is that the decoder compares the received sequence with all possible code words, which is particularly true for long block lengths, since the size of the code set grows exponentially with block length. A desirable decoder should be relatively simple in terms of equipment, storage and computation, even if it moderately increases the probability of error. If the lower probability of error is required, we can simply increase the block length of the code. [8]

There are two types of decoding scheme which will be described. The first decoding scheme is relatively simple but only applicable to Binary Symmetric Channel (BSC) at rates below capacity. The second decoding scheme, which decodes directly from the a posteriori, or production probabilities at the channel output is a more promising decoder.

In the first decoding scheme, the decoder computes all the parity-checks and changes any digit that is contained in more than some fixed number of unsatisfied parity-check equations. Using these new values, the parity checks are recomputed. This process is repeated until the parity checks are all satisfied.

If the parity-check sets are small, this decoding procedure is reasonable, since most of the parity-check sets will contain either one transmission error or no transmission errors. Therefore, when most of the parity-check equations checking on a digit are unsatisfied, there is a strong indication that that digit is in error.

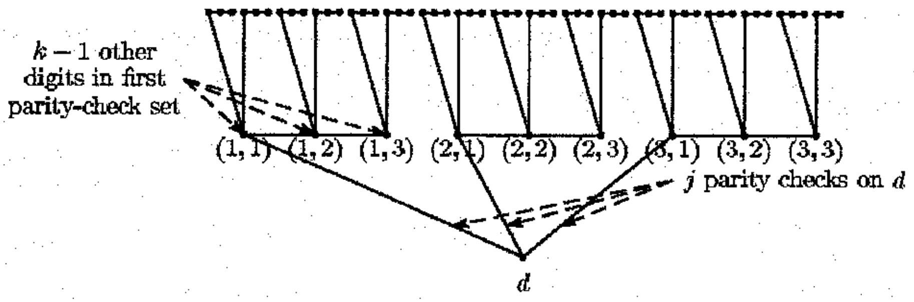


Figure 6 : Parity-check set tree

2.2.2 Mackay's encoding and decoding scheme for LDPC

Mackay reported the empirical which are observation and experiment for performance of Gallager's LDPC codes on Gaussian channels. It was shown that performance successfully better than that of standard convolutional and concatenated codes can be achieved, and the performance is almost as close to the Shannon limit as that of Turbo codes. [9]

A linear code may be described in terms of a generator matrix G or in terms of a parity check matrix H, which satisfies Hx=0 for all codewords x. The Gallager codes were superior for practical purposes.

During the work on Mackay's codes, it was realized that it is possible to create good codes from very sparse random matrices, and to decode them using approximate probabilistic algorithms. The Gallager's decoding algorithm and codes were reinvented. The sparse random parity check matrices [9] were created in the following ways.

i. Construction 1A.

An M by N matrix (M rows, N columns) is created at random with weight per column t, and weight per row as uniform as possible, and overlap between any two columns no greater then 1. The weight of a column is the number of non-zero elements. The overlap between two columns is their inner product.

ii. Construction 2A

Up to M/2 of the columns are designated weight 2 columns, and these are constructed such that there is zero overlap between any pair of columns. The remaining columns are made at random with weight 3, with the weight per row as uniform as possible, and overlap between any two columns of the entire matrix no greater than 1.

iii. Construction 1B and 2B.

A small number of columns are deleted from a matrix produced by constructions 1A and 2A, respectively, so that the bipartite graph corresponding to the matrix has no short cycles of length less than some length *l*.

The constructions stated above do not ensure that all rows of the matrix are linearly independent, therefore the M x N matrix created is the parity matrix of a linear code with rate at least R = M/N, where K = N – M. R denotes the rate. The generator matrix of the code can be created by Gaussian elimination.

The Gaussian channel is simulated with binary input $\pm a$ and additive noise of variance equals to 1. If one communicates using a code of rate R then it is conventional to describe the signal to noise ratio by $\frac{Eb}{No} = \frac{a^2}{2R\sigma^2}$ and this number is reported in decibels as 10 $\log_{10} E_b/N_o$.

iv. Decoding

The decoding problem is to find the most probable vector x such that Hx mod 2 = 0, with the likelihood of x given by $\prod_n f_n^{x_n}$ where $f_n^1 = 1/(1+\exp(-2ay_n/\sigma^2))$ and $f_n^0 = 1 - f_n^1$, and y_n is the channel's output at time n.

Gallager's algorithm may be viewed as an approximate belief propagation algorithm. Moreover, the Turbo decoding algorithm may also be viewed as a belief propagation algorithm.

The elements of x are referred as bits and to the rows of H are referred as checks. We denote the set of bits n that participate in check m by N (m) ≡ {n: H_{mn} =1}. We define the set of checks in which bit n participates, M (n) ≡ {m: H_{mn} =1}. A set N (m) is denoted with bit n excluded by N (m)/n. The algorithm has two alternating parts, in which quantities q_{mn} and r_{mn} associated with each non-zero element in the H matrix are iteratively updated. The quantity q_{mn}^x is meant to be the probability that bit n of **x** is x, given the information obtained via checks other than check m. The quantity r_{mn}^x is meant to be the probability of check m being satisfied if bit n of x is considered fixed at x and the other bits have a separable distribution given by the probabilities {$q_{mn'}$: n' ∈ N(m)\n}. The algorithm would produce the exact posterior probabilities of all the bits if the bipartite graph defined by the matrix H contained no cycles.

2.3 BCH CODES

Bose – Chaudhuri – Hocquenghem (BCH) codes are a generalization of Hamming codes that allow multiple error correction. They are a powerful class of cyclic codes that provide a large selection of block lengths, codes rates, alphabet sizes, and error – correcting capability. At the block lengths of a few hundred, BCH codes could outperform all other block codes with the same block length and code rate. BCH codes employ a binary alphabet and a codeword block length of n = 2m – 1, where m = 3, 4, and etc. [27]

2.3.1 BCH Codes Parameters

The BCH codes have the following parameters for any positive integers 'm' and 't', where $m \geq 3$ and $t < 2^{m-1}$.

Block length: $n = 2^{m} - 1$;

Number of parity – check digits: $n - k \leq mt$;

Minimum distance: $d\min \geq 2t + 1$.

This code [37] is capable of correcting combinations of 't' or fewer errors in a block of $n = 2^{m} - 1$ digits. The generator polynomial of this code is specified in terms of its roots from the Galois field. GF(2^{m}). The generator polynomial g(X) of the t – error – correcting BCH code of length $2^{m} - 1$ is the lowest – degree polynomial over GF(2) that has : "a, a^2, a^3... a^{2t}" as its roots. Let $\phi_i(X)$ be the minimal polynomial of a^i. Then, g(X) must be the least common multiple (LCM) of $\phi_1(X), \phi_2(X), ..., \phi_{2t}(X)$, which is $g(X) = LCM\{\phi_1(X), \phi_2(X), ..., \phi_{2t}(X)\}$.

Hence, every even power of 'a' in the sequence of "a, a^2, a^3... a^{2t}" has the same minimal polynomial as the preceding odd power of 'a' in the sequence. As a result, the generator polynomial g(X) of the binary t – error – correcting BCH code of length $2^{m} - 1$ can be reduced from $g(X) = LCM\{\phi_1(X), \phi_2(X), ..., \phi_{2t}(X)\}$ to $g(X) = LCM\{\phi_1(X), \phi_3(X), ..., \phi_{2t-1}(X)\}$.

Due to the degree of each minimal polynomial is 'm' or less, the degree of g(X) is at most 'mt'; that is, the number of parity – check digits, n – k, of the code is at most equal to 'mt'. 'n' represents the block size, 'n – k' represents the parity – check digits and 't' represents the number of errors that could be corrected with BCH codes. If the value of 't' is small, n – k is exactly equal to 'mt'. The BCH codes defined are usually called primitive BCH codes, where its parameters are code length of $2^m - 1$ with $m \leq 10$. [37]

The single – error – correcting BCH code of length $2^m - 1$ is generated by $g(X) = \phi_1(X)$. Because 'a' is a primitive element of GF(2^m), $\phi_1(X)$ is a primitive polynomial of degree 'm'. Therefore, the single – error- correcting BCH code of length $2^m - 1$ is a Hamming code.

Let $v(X) = v_0 + v_1 a^i + ... + v_{n-1} a^{(n-1)i} = 0$ be a code polynomial in a t – error – correcting BCH code of length $n = 2^m - 1$. This equality can be written as a matrix product as follows:

$$(v_0, v_1, ..., v_{n-1}) \cdot \begin{bmatrix} 1 \\ a^i \\ a^{2i} \\ . \\ . \\ a^{(n-1)i} \end{bmatrix} = 0$$

for $1 \leq i \leq 2t$. The condition given as above shows that the inner product of $(v_0, v_1, ..., v_{n-1})$ and $(1, a^i, a^{2i}, ..., a^{(n-1)i})$ is equal to zero. Therefore, as 'v' is the codeword in the BCH code, then

$$v \cdot H^T = 0$$

2.3.2 Galois Array

Galois array theory is an important for BCH codes encoding and decoding algorithms. In mathematics, more specifically in abstract algebra, Galois Theory, named after Évariste Galois, provides a connection between field theory and group theory. Using Galois Theory, certain problems in field theory can be reduced to group theory, which is in some sense simpler and better understood. Abstract algebra is the field of mathematics that studies algebraic structures, such as groups, rings, fields, modules, vector spaces, and algebras.

Group theory is that branch of mathematics concerned with the study of groups. It has several applications in physics and chemistry. Galois Theory uses groups to describe the symmetries of the equations satisfied by the solutions to a polynomial equation. [10]

A group G is a collection of objects with an operation · satisfying the following rules:

1) For any two elements x and y in the group G we also have x·y in the group G.
2) There is an element, which is usually written 1 or e, but sometimes 0, called the identity in G such that for any x in the group G we have 1·x = x = x · 1.
3) For any elements x, y, z in G we have (x · y) · z = x · (y · z). This property is called associativity; it means we can write x·y·z unambiguously.
4) Every element x in G has a unique inverse y (sometimes written -x or x-1) so that x · y = y · x = 1. [11]

Field theory is a branch of mathematics which studies the properties of fields. A field is a mathematical entity for which addition, subtraction, multiplication and division are well-defined.

A field F is quite similar with a group, but it has two operations, usually written · and +. F is a field if F has elements 0 and 1 such that F with the operation + is a group, for example (F, +), the set F without the element 0 is a group with the operation ·, for example (F {0}, ·). Besides, it also involves relations like (x + y) ·z = x · z + y · z, 0 · x = 0 = x · 0, x · y = y · x. The general definition of a field is that a field is a set in which we can add, subtract and multiply any elements, and can divide by any element other than 0. [11]

Originally Galois used permutation groups to describe how the various roots of a given polynomial equation are related to each other. The modern approach to Galois Theory, developed by Richard Dedekind, Leopold Kronecker and Emil Artin, among others, involves studying automorphisms of field extensions. In mathematics, an automorphism is an isomorphism from a mathematical object to itself. It is, in some sense, a symmetry of the object, and a way of mapping the object to itself while preserving all of its structure. The set of all automorphisms of an object forms a group, called the automorphism group. It is, loosely speaking, the symmetry group of the object. [11]

Galois theory is concerned with symmetries in the roots of a polynomial p(x). for example, if $p(x) = x^2 - 2$ then the roots are $\pm\sqrt{2}$. A symmetry of the roots is a way of swapping the solutions around in a way which does not matter in some sense. Therefore, $\sqrt{2}$ and $-\sqrt{2}$ are the same because any polynomial expression involving $\sqrt{2}$ will be the same if $\sqrt{2}$ is replaced by $-\sqrt{2}$. For example, for the equation $\sqrt{2}^2 + \sqrt{2} + 1 = 3 + \sqrt{2}$, or $\alpha^2 + \alpha + 1 = 3 + \alpha$ when $\alpha - \sqrt{2}$, and this will be true for any expression involving only adding and multiplying $\sqrt{2}$. [11]

2.3.3 Decoding of BCH Codes

There are several decoding scheme available for BCH codes, which would be described as follows:

i. Berlekamp – Massey algorithm (BMA)

   The BMA was invented by Berlekamp and Massey. This is a computationally efficient method to solve the syndrome equation, in terms of the number of operations in GF (2^m). The BMA is important for BCH decoders' implementation in software.

ii. Euclidean algorithm (EA)

Euclidean algorithm involves determining the greatest common divisor (GCD) of two integers of elements of any Euclidean domain by repeatedly dividing the two numbers and the remainder in turns. Due to its regular structure, the EA is widely used in hardware implementations for BCH decoders.

iii. Direct solution

This method was proposed by Peterson. It directly finds the coefficients of error locator polynomial as a set of linear equations. The term Peterson – Gorenstein – Zierler decoder was used in the literature. As the complexity of inverting a matrix grows with eh cube of the error – correcting capability, the direct solution method works only for small values of 't'.

For this project, the Berlekamp decoding scheme would be implemented for decoding the BCH codes.

When the codeword $v(X) = v_0 + v_1X + v_2X^2 + ... + v_{n-1}X^{n-1}$ is transmitted, the transmission errors would result in the following received vector:

$$r(X) = r_0 + r_1X + r_2X^2 + ... + r_{n-1}X^{n-1}$$

In order to decode the BCH codes, the elements $\beta \in GF(2^m)$ to number the positions of a codeword, or the order of the coefficients of the associated polynomial.

By using the $GF(2^m)$ arithmetic, the positions of the errors can be found with solving a set of equations. These equations can be obtained from the error polynomial e(X) and the zeros of the code, a^i.

The equation $r(X) = v(X) + e(X)$ represents the polynomial associated with a received codeword, where the error polynomial is defined as $e(X) = e_{i1}(X^{i1}) + e_{i2}(X^{i2}) + ... + e_{iv}(X^{iv})$ and $v \le t$ is the number of errors. The sets for e(X) and H matrix are known as the error values and error positions respectively, where $e_i \in \{0,1\}$ for binary BCH codes and $a \in GF(2^m)$.

Firstly, the syndrome from the received vector r(X) would be computed for decoding. The syndrome could be represented by: $S = r \cdot H^T$. The ith component of the syndrome is $S_i = r(a^i) = r_0 + r_1 a^i + r_2 a^{2i} + ... + r_{n-1} a^{(n-1)i}$, for $1 \le i \le 2t$. The syndrome components are elements in the field GF (2^m). The syndrome could be computed by dividing the r(X) by the minimal polynomial $\phi_i(X)$ of a^i, which is $r(X) = c_i(X)\phi_i(X) + d_i(X)$, where $d^i(X)$ is the remainder with the degree less than that of $\phi_i(X)$. Because $\phi_i(a^i) = 0$, the syndrome could be written as: $S_i = r(a^i) = d_i(a^i)$.

Let the error locator polynomial be defined as $\sigma(x) = \prod_{l=1}^{v}(1 + a^{il}x) = 1 + \sigma_1 x + \sigma_2 x^2 + ... + \sigma_v x^v$, with roots equal to the inverses of the error locations. Then the following relation between the coefficients of σ (x) and the syndromes holds:

$$\begin{pmatrix} S_{v+1} \\ S_{v+2} \\ \vdots \\ S_{2v} \end{pmatrix} = \begin{pmatrix} S_1 & S_2 & \dots & S_V \\ S_2 & S_3 & \dots & S_{V+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_V & S_{V+1} & \dots & S_{2V-1} \end{pmatrix} \begin{pmatrix} \sigma_V \\ \sigma_{V-1} \\ \vdots \\ \sigma_1 \end{pmatrix}$$

The decoder consists of digital circuits and processing elements to accomplish the following tasks:

- Compute the syndromes, by evaluating the received polynomial at the zeros of the code.

$S_i = r(a^i)$

- Find the coefficients of the error locator polynomial $\sigma(x)$
- Find the inverses of the roots of $\sigma(x)$, for example, the locations of the errors, $a^{i1}, \ldots, a^{iv}$.
- Find the values of the errors $e_{i1}, \ldots, e_{iv}$.
- Correct the received word with the error locations and values found.

CHAPTER 3
METHODOLOGY

3.1 PROCEDURE IDENTIFICATION

In order to perform effectively in this project, the problem statement and objectives are firstly defined. The objective for this project is to do research on the LDPC and BCH codes. Before starting for simulating the LDPC and BCH codes in the Matlab simulation software, detailed understanding of the LDPC and BCH codes characteristics including the H matrix generation, Galois array matrix, encoding the message bits, channel modulation and decoding the received bits are required. Furthermore, the Matlab software would be used for encoding and decoding the codes. The algorithm for the LDPC and BCH source codes would be investigated as well. The Matlab software is used as a simulation tool that helps to generate results of encoding and decoding for both the LDPC and BCH codes.

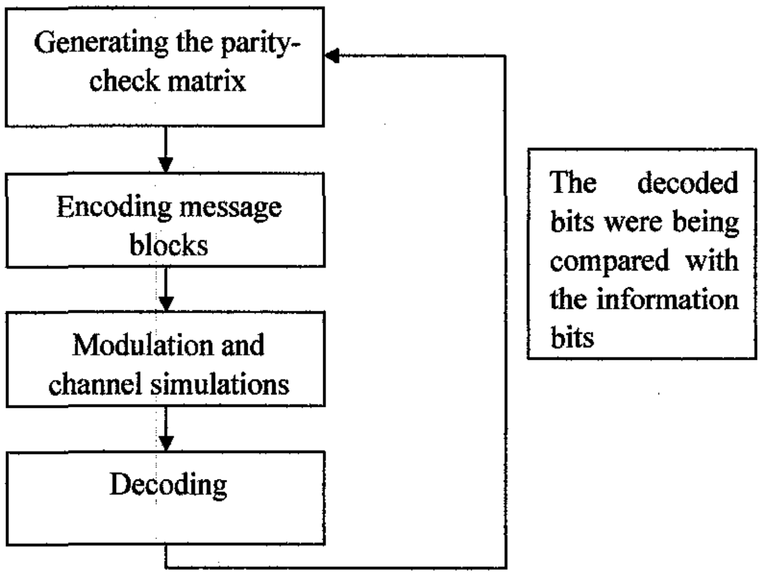

Generating the parity-check matrix
Encoding message blocks
Modulation and channel simulations
Decoding
The decoded bits were being compared with the information bits

Figure 7 : LDPC and BCH codes algorithm

3.1.1 LDPC codes algorithm

Low Density Parity Check codes can be specified by a non-systematic sparse parity-check matrix, H. H matrix would have a uniform column weight greater than 3, and a uniform row weight as well. H is constructed at random subject to these constraints. An (n,j,k) LDPC code is specified by a parity check matrix H, having n-k rows, n columns and j 1's per column. For this Matlab program, the k=3. All the parity check matrices would have 3 ones per column. The code formed from such a parity check matrix is known as a regular Gallager code. [7] The LDPC Matlab source codes could be referred in Appendix A.

Step 1: Generating the parity check matrix

For generating the parity check matrix for a (200,100,3) LDPC code, the function gen_ldpc(rows,cols) could be used as follows:

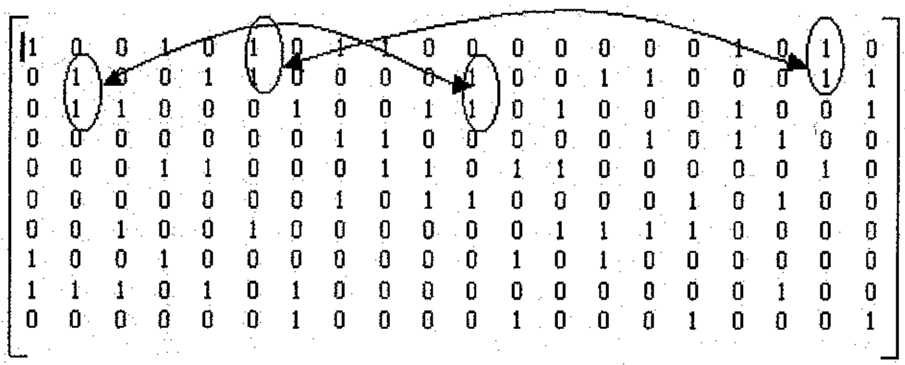
```
h=gen_ldpc(100,200);
```

The algorithm [7] for implementation of this function was shown as follows:

1. An all zero matrix H of dimension rows x cols is created.
2. For each column in H, three 1's are placed in rows chosen at random, subject only to the constraint that the ones be placed in distinct rows.
3. The Matlab software then runs through the matrix searching for a row with zero 1's or just one 1. If a row has no 1's in it then it is a redundant row. So the software chooses 2 columns in the same row at random and places 1's in those columns. If a row just has one 1 in a row, this means that the codeword bit in that column is always zero. Therefore whenever the software find a row with just one 1 in it, it randomly picks another column in the same row and places a 1 there.
4. The software then calculates the number of 1's per row. Number of 1's per row = (cols x bits_per_col)/rows. If this is not an integer, the software rounds

the value to the next higher integer. If the number of 1's per row calculated is not an integer, it is not possible to have a uniform number of ones in each row.

5. The software then runs through the matrix trying to make the number of 1's per row as uniform as possible. For any row containing more number of ones than the value calculated in Step 4, the software picks a column containing a 1 at random and tries to move that 1 to a different row in the same column. The software makes sure that the row chosen does not have a 1 in that particular column. If the software is not able to find such a row, it just tries with a different column containing a 1 in row i.
6. A good parity check matrix for LDPC codes generates a factor graph with no cycles in it. The software runs through the graph trying to eliminate cycles of length 4, for example situations where pairs of rows share1's in a particular pair of columns as shown in Figure 6. The figure 6 shown below is the parity check matrix of a (20,10,3) LDPC code.



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

Figure 8 : Parity check matrix of a (20, 10, 3) LDPC code.

Step 2: Encoding the Message Blocks

The codeword which is by 'u' variable, and an M by N IG parity check matrix H could be related as $u.H^T = 0$. The message bits, s, are located at the end of the codeword and the check bits, c occupy the beginning of the codeword, where the equation is u = [c|s]. [7]

Let H = [A|B], where A is an M by M matrix and B is an M by N-M matrix. If the message bits 's' are located at the end of the codeword, the first part of H is an identity matrix. The equation derived are Ac + Bs = 0, and c = A^{-1}Bs. The check bits could be computed as long as A is non-singular.

For encoding of IG parity check matrix, the first M by M part of the matrix (A) has to be non singular. The IG parity check matrix could be obtained by the gen_ldpc() function. The function rearrange_cols(parity check) rearranges the columns of the IG parity check matrix such that A is non singular. The function does only the column operations of the Gauss-Jordan reduction that can be used to reduce the IG parity check matrix to the systematic form.

Step 3: Conducting Modulation and Channel Simulation

For this program, the Binary Phase Shift Keying (BPSK) and Additive White Gaussian Noise (AWGN) are the constraints for performing the modulation and channel simulation respectively. The Binary Phase Shift Keying uses a single carrier with the phase shifts of 180° to carry the signalIn communications. [7]

The additive white Gaussian noise (AWGN) channel model is one in which the only impairment is the linear addition of wideband or white noise with a constant spectral density a Gaussian distribution of amplitude. The model does not account for the phenomena of fading, frequency selectivity, interference, nonlinearity or dispersion. However, it produces simple, tractable mathematical models which are useful for gaining insight into the underlying behavior of a system before these other phenomena are considered.

The AWGN channel is a good model for many satellite and deep space communication links. It is not a good model for most terrestrial links because of multipath, terrain blocking, interference, etc. However for terrestrial path modeling, AWGN is commonly used to simulate background noise of the channel under study, in addition to multipath, terrain blocking, interference, ground clutter and self interference that modern radio systems encounter in terrestrial operaton.

Step 4: Decoding Received Blocks

The decoding problem is to find the most probable vector x such that Hxmod2 = 0, with the likelihood of x given by $\prod_n f_n^{xn}$ where $f_n^1 = 1/(1+\exp(-2ay_n/\sigma^2))$ and $f_n^0 = 1 - f_n^1$, and y_n is the channel's output at time n. [7]

For the initialization, we first initialize q_{mn}^0 and q_{mn}^1 to the likelihoods of x_n, f_n^0 and f_n^1. For the AWGN channel, $f_n^1 = 1/(1+\exp(-2ay_n/\sigma^2))$ and $f_n^0 = 1 - f_n^1$ where the input to the Gaussian channel is $\pm a$, $\sigma^2 = No/2$ is the variance of the additive noise and y_n is the soft output of the Gaussian Channel.

For the horizontal step calculation, we compute $\delta_{qmn} = q_{mn}^0 - q_{mn}^1$, next $\delta_{rmn} = r_{mn}^0 - r_{mn}^1 = \prod \delta_{qmn'} : n' \in \angle(m) \backslash n$ is computed, Next we would get $r_{mn}^0 = (1+\delta_{rmn})/2$ and $r_{mn}^1 = (1+\delta_{rmn})/2$

Next is the vertical step calculation, for this step we update the values of q_{mn}^0 and q_{mn}^1 using the r's obtained in step 2.

The equations are: $q_{mn}^0 = \alpha_{mn} f_n^0 \prod r_{m'n'}^0 : m' \in M(n) \backslash m$ and $q_{mn}^0 = \alpha_{mn} f_n^0 \prod r_{m'n'}^0 : m' \in M(n) \backslash m$ where α_{mn} is chosen such that $q_{mn}^0 + q_{mn}^1 = 1$. The pseudoposterior probabilities are then obtained as: $q_n^0 = \alpha_n f_n^0 \prod r_{mn'}^0 : m \in M(n)$ and $q_n^1 = \alpha_n f_n^1 \prod r_{mn'}^1 : m \in M(n)$

The final step is the tentative decoding, we set $r_n = 1$ if $q^1{}_n > 0.5$ and $r_n = 0$ otherwise. In the software implementation the algorithm iterates 1000 times. The function **decode_ldpc(rx_waveform,No,amp,h,scale)** does the decoding. The function returns the decoded codeword and not the message bits.

The function can be used along with the modulation and channel transmission functions as shown below:

Once the decoding is done or the termination condition is satisfied or the maximum numbers of iterations are performed, the message bits need to be extracted. At the last step of encoding the message blocks, the reordering done to get a non-singular A was undone. If this reordering was done again, the message bits would be located at the end of the decoded codeword and can be extracted easily. The rearranged_cols array (output by the rearrange_cols( ) function) that holds the reordering information is used to do the reordering. The extract_mesg(vhat,rearranged_cols) works in this manner.

3.1.2 BCH Codes Algorithm

The BCH codes algorithm was divided into several parts for more detailed explanations. The several parts of BCH codes algorithms are: constructing the codeword length, generates a matrix consists of random binary numbers and creates a Galois field array, gets generator polynomial, encodes the message, BPSK modulation, channel simulation, demodulation and decode the received codes. BCH Matlab source could be referred in Appendix C.

Referred to the MATLAB communication toolbox functions, the algorithm for BCH code is described as follows:

Step 1: Construct the codeword

```
m=4;
n=2^m-1;
k=5;
nwords=10;
```

From the codes above, 'n' represents the codeword length, 'k' is the message length, and the 'nwords' represents the number of words to encode for this program.

Step 2: Create Galois field array

```
msg=gf(randint(nwords,k));
```

From the code above, randint (10,5) generates an 10 by 5 matrix of random binary numbers. "0" and "1" occur with equal probability.

'GF' function creates a Galois field array. The msg = gf(randint(nwords,k) creates a Galois field array from the matrix 'randint(nwords,k)'. The Galois field has 2^m elements, where for this program, the value of m is set to default value 1. Each element of x must be 0 or 1. The output for 'msg' is a variable that MATLAB recognizes as a Galois field array, rather than an array of integers. [13]

Step 3: Create generator polynomial

```
[genpoly,t]=bchgenpoly(n,k)
```

The function 'bchgenpoly' gets generator polynomial and error – correction capability. genpoly = bchgenpoly (n,k) returns the narrow – sense generator polynomial of a BCH code with code length 'n' and message length 'k'. The codeword length 'n' must have the form 2^m – 1 for some integer 'm' between 3 and 16. The output 'genpoly' is a Galois row vector that represents the coefficients of the generator polynomial in order of descending powers. The narrow-sense generator polynomial is (X-alpha) * (X-alpha^2) * ... * (X-alpha ^ (N-K)), where alpha is a root of the default primitive polynomial for the field GF (N+1). [14]

Step 4: Encode the message

```
code = bchenc(msg,n,k);
```

CODE = BCHENC (MSG,N,K) encodes the message in MSG using an (N,K) BCH encoder with the narrow-sense generator polynomial. MSG is a Galois array of symbols over GF (2). Each K-element row of MSG represents a message word, where the leftmost symbol is the most significant symbol. Parity symbols are at the end of each word in the output Galois array CODE. [15]

1. Fundamental checks on parameter data types: Firstly, MSG must be a Galois array; secondly, MSG must be in GF (2).
2. Set and check the parity position. Parity position must be either at the beginning or at the end.
3. Check the message length. The message length must equal K.
4. Get the generator polynomial.
5. Get the generator matrix. The function 'cyclgen' is used. CYCLGEN produce parity-check and generator matrices for cyclic code. H = CYCLGEN(N, P) produces the parity-check matrix for a given codeword length N and generator polynomial P. The vector P gives the binary coefficients of the generator polynomial in order of ascending powers. A polynomial can generate a cyclic code if and only if it is a factor of X^N-1. The message length of the code is K = N - M, where M is the degree of P. The parity-check matrix is an M-by-N matrix.
6. Do the coding. Code = msg * gen. (message * generator matrix)
7. Rearrange parity if necessary. [20]

```
y=double(code.x)
```

The above code converts 'code' from Galois array to integers for modulation.

Step 5: BPSK modulation

```
y2=pskmod(y,2);
```

The ‘PSKMOD’ function represents phase shift keying modulation.

Y = PSKMOD(X, M) outputs the complex envelope of the modulation of the message signal X, using the phase shift keying modulation. M is the alphabet size and must be an integer power or 2. The message signal X must consist of integers between 0 and M-1. For two-dimensional signals, the function treats each column as 1 channel. [16]

1. Check that x is a positive integer
2. Check that M is a positive integer. Determine whether is BPSK, QPSK, or OPSK.
3. Check that x is within range. Elements of input X must be integers in [0, M-1].
4. Determine the initial phase. The default value is 0.
5. Evaluate the phase angle based on M and the input value. The phase angle lies between 0 - 2*pi.
6. The complex envelope is (cos (theta) + j*sin (theta)). This can be expressed as exp (j*theta). If there is an initial phase, it is added to the existing phase angle
7. Restore the output signal to the original orientation.

Step 6: AWGN channel simulation

```
channel=awgn(y2,10);
```

AWGN add white Gaussian noise to a signal.
Y = AWGN(X, SNR) adds white Gaussian noise to X. The SNR is in dB.
The power of X is assumed to be 0 dBW. If X is complex, then AWGN adds complex noise. [17]

Step 7: BPSK demodulation

```
r=pskdemod(channel,2);
```

Demodulation is basically the reverse of modulation.
The demodulator, which is designed specifically for the symbol-set used by the modulator, determines the phase of the received signal and maps it back to the symbol it represents, thus recovering the original data. [18]

```
r2=gf(r);
```

The above code converts the signal 'r' to Galois array.

Step 8: Decode the received message

```
[newmsg,err,ccode] = bchdec(r2,n,k)
```

The function 'BCHDEC; represents the BCH decoder.

DECODED = BCHDEC (CODE, N, K) attempts to decode the received signal in CODE using an (N,K) BCH decoder with the narrow-sense generator polynomial. CODE is a Galois array of symbols over GF (2). Each N-element row of CODE represents a corrupted systematic codeword, where the parity symbols are at the end and the leftmost symbol is the most significant symbol.

In the Galois array DECODED, each row represents the attempt at decoding the corresponding row in CODE. A decoding failure occurs if a row of CODE contains more than T errors, where T is the number of correctable errors as returned from BCHGENPOLY. In this case, BCHDEC forms the corresponding row of DECODED by merely removing N-K symbols from the end of the row of CODE.

[DECODED,CNUMERR,CCODE] = BCHDEC(r2, n, k) returns CCODE, the corrected version of CODE. The Galois array CCODE is in the same format as

CODE. If a decoding failure occurs in a certain row of CODE, then the corresponding row in CCODE contains that row unchanged. [19]

1. Fundamental check on parameter data types: Firstly, CODE must be a Galois array. Secondly, code must be in GF (2).
2. Check width of code. CODE must be either an N-element row vector or a matrix with N columns.
3. Set and check the parity position. Parity position must be either beginning or at the end.
4. Get the number of errors we can correct
5. Bring the coded word into the extension field
6. Call to core algorithm Berlekamp
7. Bring back to gf(2).

3.2 IDENTIFICATION OF REQUIRED APPARATUS/TOOLS

This project requires Matlab simulation tool for producing results of encoding and decoding for the error correction codes. Comparison study for LDPC codes with BCH codes will be conducted through the Matlab simulation as well.

CHAPTER 4
RESULTS AND DISCUSSIONS

4.1 PERFORMANCES OF LDPC AND BCH CODES WITH VARYING SNR

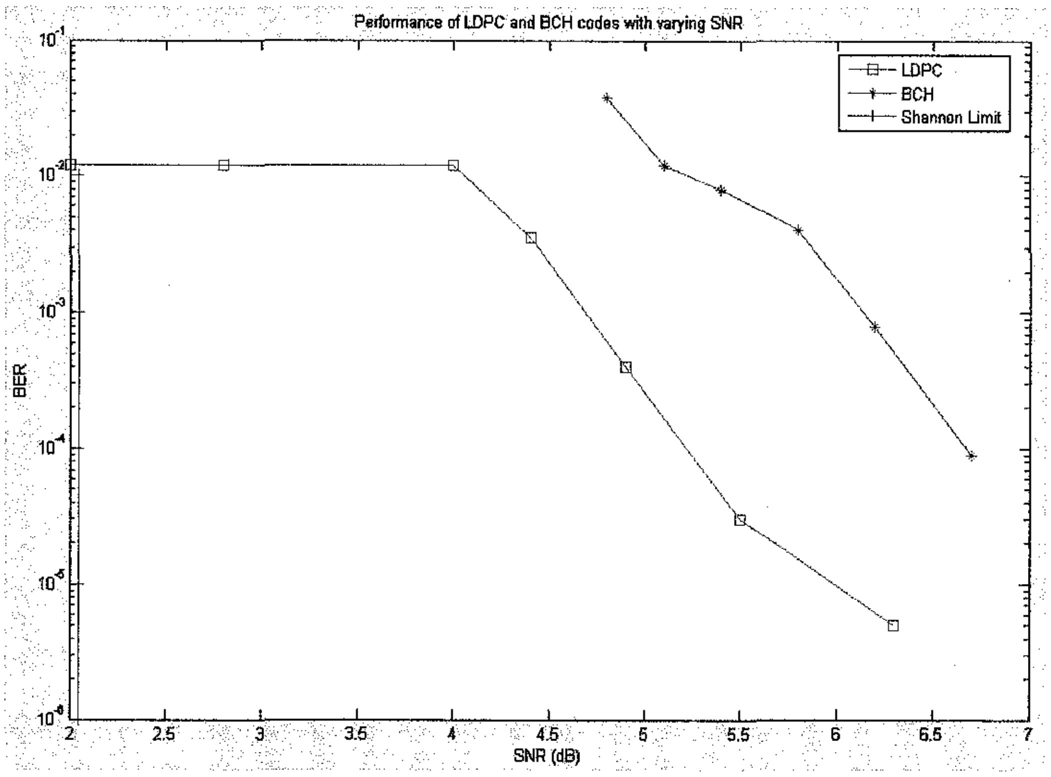


Figure 9 : BER vs. SNR

Figure 9 shows the error performances for both the LDPC and BCH codes. The y – axis represents the bit error rate, which is the ratio of the number of bits incorrectly received to the total number of bits sent during a specified time interval. For a given communication system, the bit error ratio will affected by both the data transmission rate and the signal power margin. The results comparison analysis was performed.

The above figure shows that the higher the value of Signal to Noise Ratio (SNR), the lower the Bit Error Rate (BER). This is because if SNR is higher, the signal power is stronger compared to the noise power, therefore, larger and clearer signal could the detected by the receiver. The BER of the LDPC is lower than the BER for BCH codes. This shows that the LDPC is a more efficient code, where it uses Sum Product Algorithm decoding scheme. The sum-of-product algorithm involves more detailed calculations and iterations on the decoding part for LDPC codes.

From Figure 9 observed, the LDPC code allows the data transmission rates close to the theoretical maximum, the Shannon limit. Although the LDPC cannot guarantee prefect transmission, the probability of error information can be made as small as desired. The results were measured by bit error rate and signal to noise ratio.

Shannon showed the existence of capacity achieving codes but achieving capacity is only part of it. For practical communication, we need fast encoding and decoding algorithms. The LDPC codes are the linear codes associated with sparse bipartite graphs. LDPC code is a very good error correction codes, this is due to the codes are equipped with very fast encoding and decoding algorithms.

For BCH codes, it shows that as the SNR increases, the BER or bit error rate decreases. However, when compared with the LDPC code performance graph, the BCH code shows higher bit error rate. LDPC is a more powerful code, although the decoding algorithm is more complex, it can actually decode more errors, and the bit error rate results also lower compared with BCH code. Signal to noise ratio (SNR) is an engineering term for the power ratio between a signal (meaningful information) and the background noise.

Signal to noise ratio (SNR) [21] is an engineering term for the power ratio between a signal (meaningful information) and the background noise.

$$SNR = \frac{P_{signal}}{P_{noise}}$$

SNR are usually expressed in terms of the logarithmic decibel scale because many signals have a very wide dynamic range. In decibels, the SNR is 20 times the base 10 logarithm of the amplitude ratio or 10 times the logarithm of the power ratio:

$$SNR = 10\log_{10}(\frac{P_{signal}}{P_{noise}})$$

For this project, we relate the SNR with the noise variance (No), which is:

$$SNR = 10\log_{10}(\frac{1}{No})$$

An error ratio is the ratio of the number of bits, or blocks incorrectly received to the total number of bits, or blocks sent during a specified time interval. The error ratio is usually expressed in scientific notation. For example, 2.5 erroneous bits out of 100,000 bits transmitted would be 2.5 out of 10^5 or 2.5 x 10^{-5}.

Moreover, the bit error ratio for the transmission is the number of erroneous bits received divided by the total number of bits transmitted. For the information BER, the number of erroneous decoded bits is divided by the total number of decoded bits.

4.2 THE EFFECT OF NOISE VARIANCE ON THE ACCURACY PERFORMANCE FOR LDPC AND BCH CODES

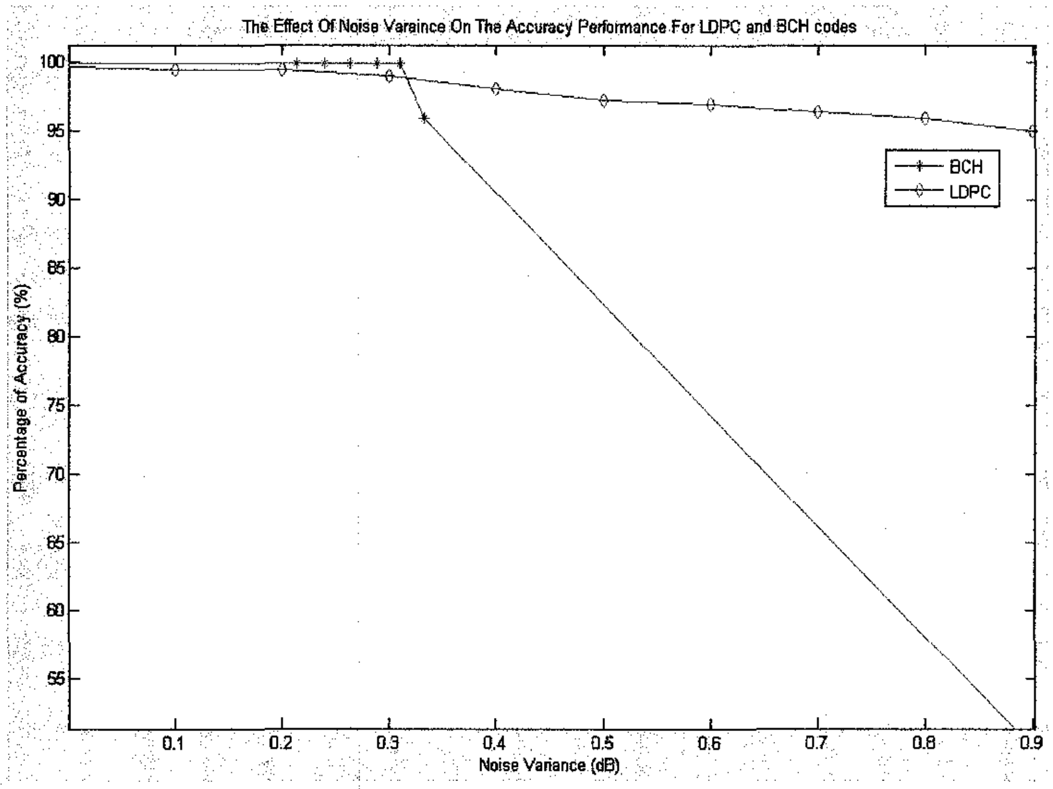


Figure 10 : Percentage of accuracy vs. Noise variance

The percentage of accuracy for this LDPC code program is 95%. For the BCH code, the percentage of accuracy for noise variance range from 0.1 to 0.3 is 99.5%, at noise variance for 0.35, the percentage of accuracy is 95%, and its accuracy decreased simultaneously with the increasing of noise variance. The larger the noise variance, the larger the probability of error occurred in the error correction codes.

For the LDPC codes, the codes were able to be successfully implemented in large range of noise variance, which is from 0.1 to 0.9. However, for the BCH codes, it was only capable for implementation with small noise variance, which ranges from 0.1 to 0.35.

The percentage of accuracy for BCH code depend on the noise variance added to it. The larger the noise variance, the larger the probability of error occurred in the BCH codes. When the noise variance gets more than 0.4, the BCH codes could not decode the received codeword correctly. Thus, LDPC codes were more efficient for implementation in encoding and decoding through a noisy channel.

Generally, Figure 10 shows that the larger the noise variance, the lower the percentage of accuracy for decoding, and the larger the probability of error occurred in the LDPC codes. The implementation of both the LDPC and BCH codes in this MATLAB simulation program were very successful. The programs were able to generate the message bits, encode the message bits with the check bits, send the encoded bits through the AWGN channel and also decoded the received bits successfully with the least probability of error.

4.3 BLOCK LENGTH VS. SIMULATION TIME

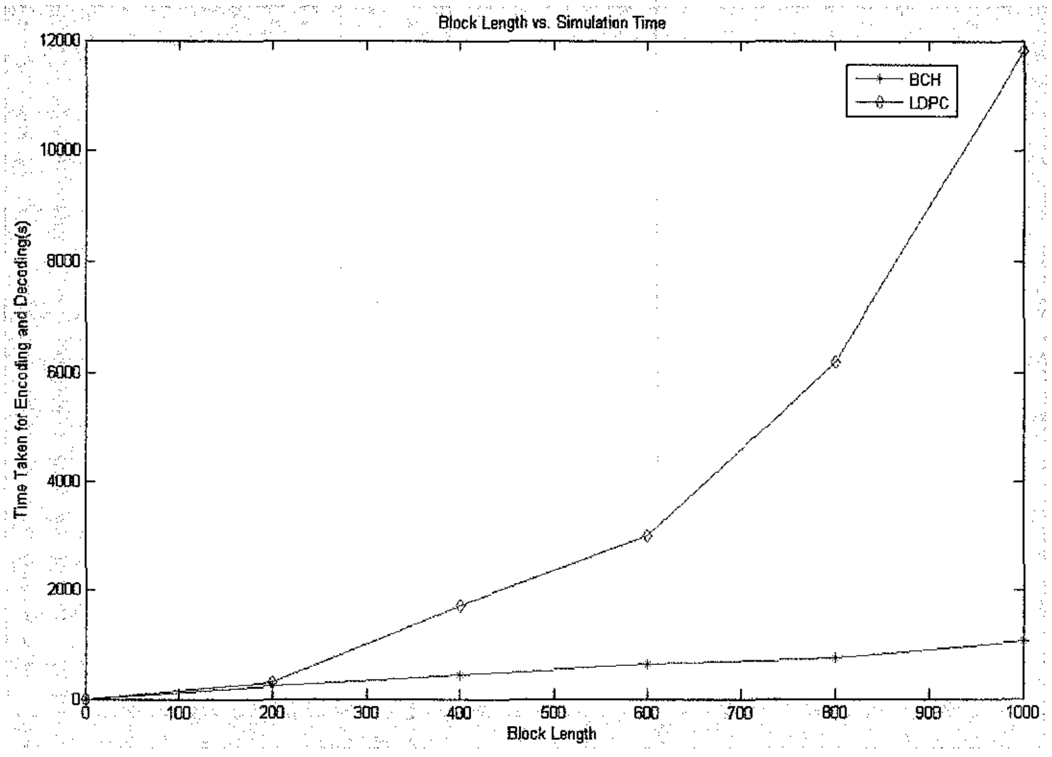


Figure 11 : Simulation time vs. block length

Figure 11 shows that the LDPC codes took longer time for encoding and decoding when compared with the BCH codes. This is due to the LDPC codes are more complicated and have higher complexity.

LDPC codes have much iteration to converge, which takes times. LDPC codes are a way of producing random codes, which is suggested by Shannon's proof of the channel coding theorem; however its decoding algorithm grows linearly with the code length.

LDPC takes a long time to converge to good solutions. The very long code word lengths are producing good decoding efficiency, in other words, the longer the LDPC codeword is sent through the channel modulation, the more accuracy of decoded data received. The iterative convergence is quite slow, which it takes 1000 iterations to converge under standard conditions. Therefore, due to those reasons, the transmission time increases for the information encoding, transmission, and decoding. For the large parity check matrix such as rows = 200 and columns = 500, the LDPC codeword decoding would last for almost 2 hours.

From Figure 11, it shows that the BCH codes takes less time for encoding and decoding to complete. This is due to the encoding and decoding algorithm of BCH codes are much simpler than the encoding and decoding algorithm for LDPC codes. However, BCH codes can only be effectively implemented with small block length code, where LDPC codes can be effectively implemented with unlimited block length code.

4.4 ERROR PERFORMANCES FOR THE CODED AND UNCODED LDPC CODES

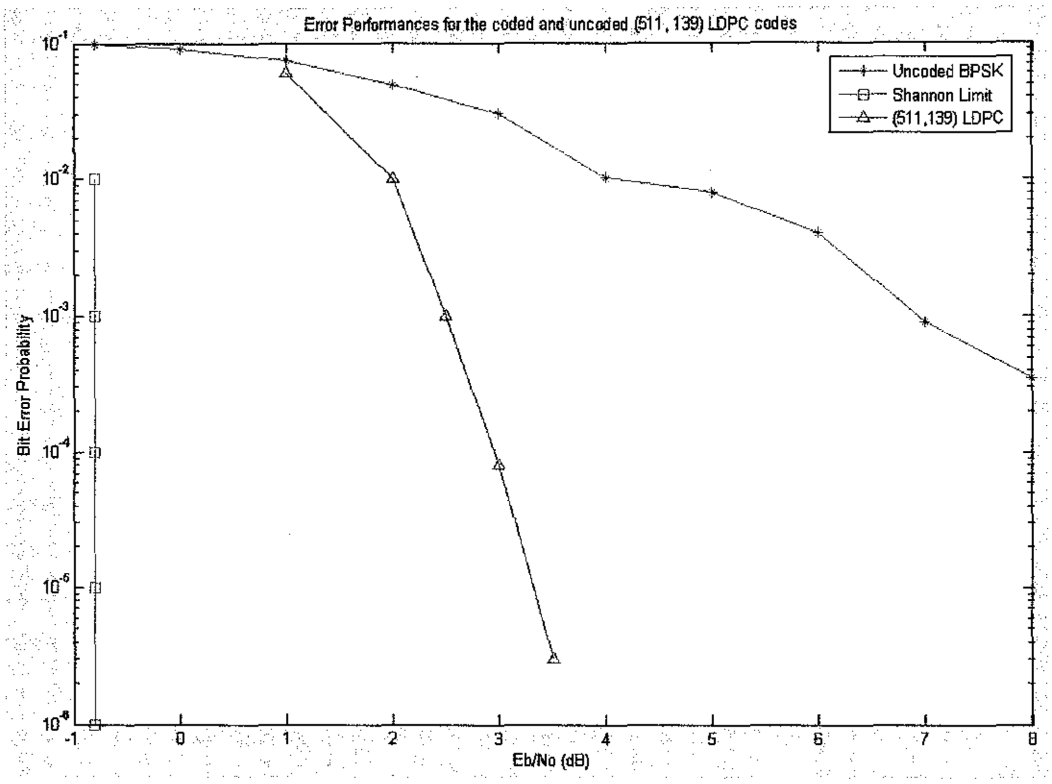


Figure 12 : Error performances for the coded and uncoded codeword [37]

Figure 12 [37] shows that when a message is coded with LDPC codes, the errors contained in the received message could be reduced effectively. The error correction codes could detect errors that are made due to noise or other impairments in the course of the transmission from the transmitter to the receiver. The error correction has the additional feature that enables localization of the errors and correcting them. The error correction schemes are computationally intensive, and require excessive redundant data which may be inhibitive for a certain application.

Error correction in some applications, such as a sender-receiver system, can be achieved with only a detection system in tandem with an automatic repeat request scheme to notify the sender that a portion of the data sent was received incorrectly and will need to be retransmitted, however where efficiency is important, it is possible to detect and correct errors with far less redundant data. [22]

The LDPC code is a powerful code where it transmits a message over a noisy transmission channel. For LDPC code, althought it cannot gurantee perfect transmission, the probability of lost information can be made as small as desired. [23]

From the Figure shown, the Shannon Limit is able to reach bit error probability of 0.000001 at SNR equals to -0.8. The LDPC codes were able to decode and correct the errors up to 0.00003 bit error probability at the SNR of 3.5. For the uncoded BPSK code, it can correct the errors up to 0.00035 at the SNR of 8. Therefore, the message that implements LDPC error correction codes was able to transmit and receive more accurately.

4.5 BLOCK LENGTH VS. BIT ERROR PERCENTAGE FOR LDPC CODES

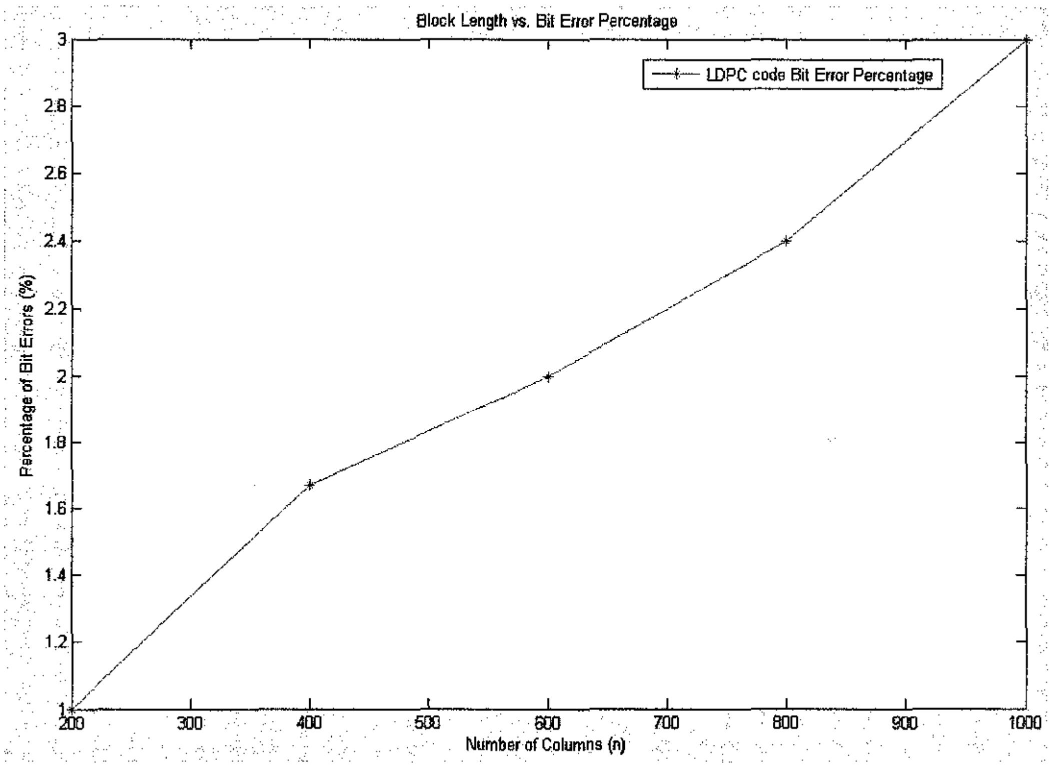


Figure 13 : Block length vs. bit error percentage for LDPC code

From the Figure 13, it was observed that the longer the codeword generated; the higher probability of error would occur. The percentage of accuracy for this program is 99%, which is quite similar with the theoretical values.

The results obtained shows that the LDPC codes is a very efficient coding method of error correction. This error correction coding technique had a channel performance very close to the Shannon limit. Both of the LDPC codes implemented in Matlab simulation achieved the results which are very close to the theoretical values.

4.6 BIT ERROR RATE VS. NUMBER OF ROWS

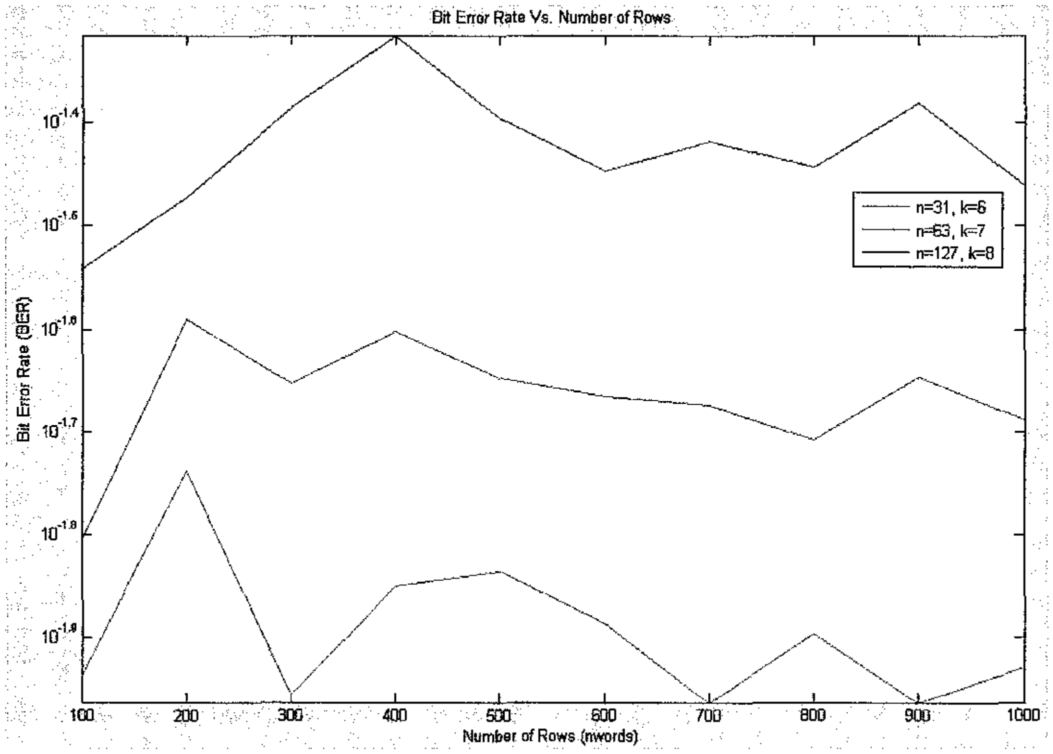


Figure 14 : BER vs. number of rows for bch codes

From Figure 14, it shows that the bit error rate is higher when the code size increases. The n = 127, k = 8 BCH code generated the highest probability of error when compared to the BCH codes for n = 63, k = 7, and n = 31, k = 6. One way to lower the noise density is to reduce the bandwidth. For a given communication system, the bit error ratio is affected by both the data transmission and the signal power margin. The power margin is the difference between available signal power and the minimum signal power needed to overcome system losses and still satisfy the minimum input requirement of the receiver for a given performance level.

4.7 PROBABILITY OF ERROR VS. NUMBER OF ROWS FOR BCH CODES

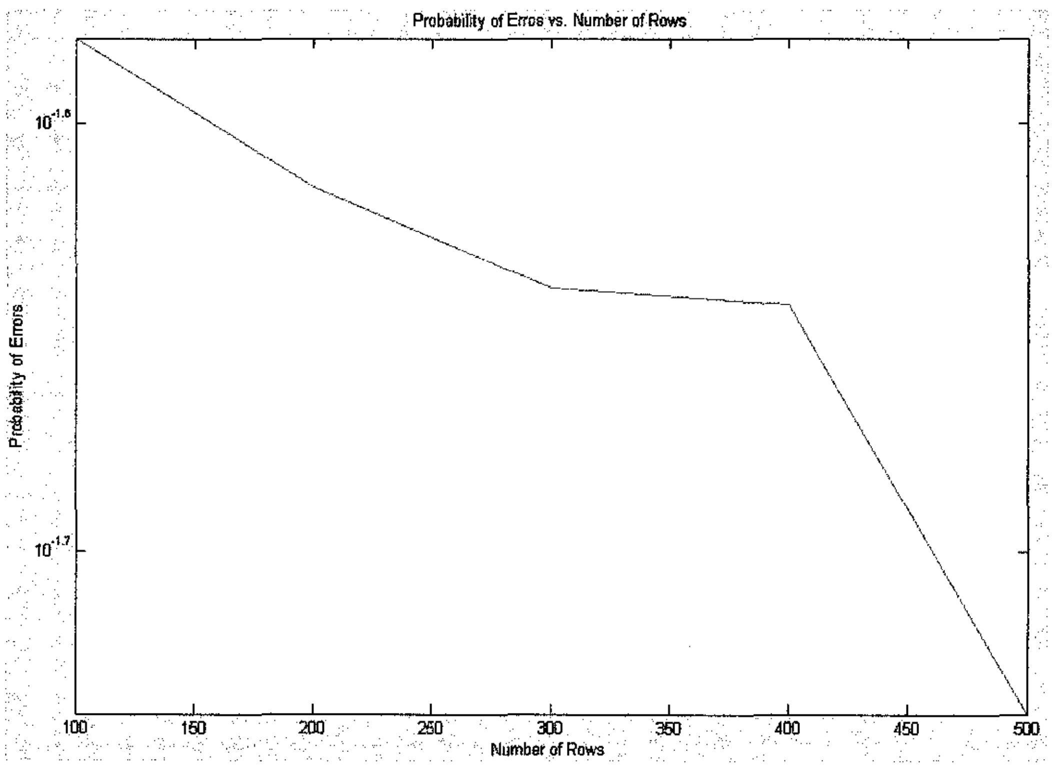


Figure 15 : Probability of errors vs. number of rows for BCH codes

Figure 15 shows that the longer the number of rows for BCH codes, the lower the probability of errors or bit error rate (BER). When a codeword was sent with more number of times, more iterations for decoding would be conducted, which will reduce the Bit Error Rate.

Bit error ratio (BER) [24] is an error ratio of the number of bits incorrectly received to the total number of bits sent during a specified time interval. The error ratio is usually expressed in scientific notation, for instance 2.5×10^{-5}

The bit error ratio would be affected by both the data transmission rate and the signal power margin. Data transmission is the conveyance of information from one space to another.

CHAPTER 5
CONCLUSION AND RECOMMENDATIONS

5.1 CONCLUSION

This project allows me to learn two very interesting and powerful error correcting codes, which are LDPC and BCH codes. The codes are methods of transmitting message over a noisy transmission channel. The codes are practically important for error correction and detection during the transmission of data. This error detection would detect errors that are made due to noise during the transmission from the transmitter to the receiver and eliminate the noise. Compare with BCH codes, the LDPC codes can allow data transmission rate close to the Shannon limit or theoretical maximum.

The LDPC code algorithm could be divided into four parts, which are generating the parity-check matrix, encoding message blocks, modulation channel simulations and decoding the received message bits. Moreover, the several parts of BCH codes algorithms are: constructing the codeword length, generates a matrix consists of random binary numbers and creates a Galois field array, gets generator polynomial, encodes the message, BPSK modulation, channel simulation, demodulation and decode the received codes.

The implementations of error correction codes in Matlab simulation software for this Final Year Project were very successful. The codes enabled us to analyze the error correction codes in further detail and research were conducted successfully.

5.2 RECOMMENDATION

1. Implementation of LDPC code on MIMO architecture and OFDM modulation.

   According to Intel Technology Journal (May 15, 2006) [38], the wireless channels often suffer the problem of interference caused by the reception of a small number of reflections from remote objects. The interference causes the receiver to receive the imperfect signals. The OFDM modulation provides good interference rejection mechanism. The use of OFDM modulation within MIMO structured systems creates a strong system that has the ability to successfully reject fading and interference.

   The MIMO channel can increase its capacity and throughput by using the proper coding prior to transmission. The coding procedure includes adding the protection bits to the transmitted data during transmission. LDPC codes were recommended for this system as they are highly efficient capacity - approaching codes. LDPC codes will fulfill the high - throughput potential of MIMO systems efficiently. [38]

2. Implementation of LDPC and BCH and other error correcting codes on hardware by using FPGA. The hardware description language (HDL) and schematic design should be provided for FPGA implementations. The languages are VHDL and Verilog.

REFERENCES

[1] Madhu Sudan, "Essential Coding Theory", September 27,2004. http://theory.lcs.mit.edu/~madhu/FT04/scribe/lect06.pdf#search=%22history %20of%20BCH%20codes%22

[2] http://en.wikipedia.org/wiki/Berlekamp-Massey_algorithm

[3] http://en.wikipedia.org/wiki/Low_Density_Parity_Check_Codes

[4] Robert H. Morelos - Zaragoza, The Art Of Error Correcting Coding, John Wiley & Sons, LTD, 2002.

[5] http://en.wikipedia.org/wiki/Error_detection_and_correction

[6] Jian Sun, An introduction to low density parity check (LDPC) codes, Wireless communication research laboratory, Lane department of computer science and electrical engineering, West Virgina University, June 3, 2003

[7] http://plaza.ufl.edu/nayagam

[8] Robert G. Gallager, Low - Density Parity - Check Codes, 1963

[9] www.inference.phy.cam.ac.uk/mackay/codes/

[10] http://en.wikipedia.org/wiki/Galois_theory

[11] http://nrich.maths.org/public

[12] http://en.wikipedia.org/wiki/Berlekamp-Massey_algorithm

[13] MATLAB communication toolbox functions for Galois array

[14] MATLAB communication toolbox functions for generator polynomial

[15] MATLAB communication toolbox functions for bchenc

[16] MATLAB communication toolbox functions for pskmod

[17] MATLAB communication toolbox functions for awgn

[18] MATLAB communication toolbox functions for pskdemod

[19] MATLAB communication toolbox functions for bchdec

[20] http://www.comap.com/product/?idx=655

[21] wikipedia.org/wiki/Signal-to-noise_ratio

[22] http://en.wikipedia.org/wiki/Error_correction

[23] http://en.wikipedia.org/wiki/LDPC

[24] http://en.wikipedia.org/wiki/Bit_error_rate

[25] Proakis, John G., "Digital Communications, Singapore", McGraw Hill, 1995.

[26] Forouzan, Behrouz A., "TCP/IP Protocal Suite, 2nd edition", McGraw Hill, 2003.

[27] Bernard Sklar, "Digital Communications Fundamentals and Applications, 2nd edition", Prentice Hall, 2001.

[28] Mackay, D.J.C.; Neal, R.M., "Near Shannon limit performance of low density parity check codes", IEEE Transactions on Information Theory, Volume: 33 Issue 6, 1997.

[29] Mackay, D.J.C., "Good error-correcting codes based on very sparse matrices", IEEE Transactions on Information Theory, Volume: 45 Issue 2, March 1999.

[30] R.G.Gallager., "Low Density Parity Check Codes", IEEE Transactions on Information Theory, 1963.

[31] Kschischang, F.R.; Frey, B.J.; Loeliger, "Factor graphs and the sum-product algorithm", IEEE Transactions on Information Theory Volume: 47 Issue: 2, 2001.

[32] Kavcic, A. Xiao Ma Mitzenmacher, M., "Binary intersymbol interference channels: Gallager codes, density evolution, and code performance bounds", IEEE Transactions on Information Theory, July 2003.

[33] E. R. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg, "On the intractability of certain coding problems," IEE Transactions on Information Theory, vol. 24, May 1978.

[34] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo – codes," IEEE Transactions on Communications, October 1996.

[35] M. C. Davey and D. J. C. Mackay, "Low density parity check codes over GF(q)," IEEE Communications Letter, June 1998.

[36] G. L. Feng and T. R. N. Rao, "Decoding algebraic-geometric codes up to the designed minimum distance," IEEE Transactions on Information Theory, volume 39, January 1993.

[37] Shu Lin, Daniel J. Costello, Jr., "Error Control Coding", Prentice Hall, Second Edition, 2004.

[38] http://www.intel.com/technology/itj/2006/volume10issue02/art07_MIMO_Ar chitecture/p02_intro.htm.

APPENDICES

APPENDIX A
LDPC SOURCE CODES

```
>> rows=100;
>> cols=200;
>> h=gen_ldpc(rows,cols);

variance =

    0.4140

>> [newh,rearranged_cols]=rearrange_cols(h);
>> for i=1:rows
for j=1:rows
A(i,j)=newh(i,j);
end
end
>> for i=1:rows
for j=rows+1:cols
B(i,j-rows)=newh(i,j);
end
end
>> for i=1:cols-rows
s(i)=round(rand);
end
>> s=s.';
>> Ainverse=inv_GF2(A);
d=mul_GF2(Ainverse,B);
>> c=mul_GF2(d,s);
>> u1=c;
>> for i=rows+1:cols
u1(i)=s(i-rows);
end
>> u=reorder_bits(u1,rearranged_cols);
>> tx_waveform=bpsk(u,1);
>> No=0.5;
>> rx_waveform=awgn(tx_waveform,No);
>> scale(1:length(c))=1;
>> length(rx_waveform)

ans =

   200

>> vhat=decode_ldpc_log(rx_waveform,No,1,h,scale)

vhat =

  Columns 1 through 14

     0     1     0     1     1     1     1     1     0     1     1     1     0     0

  Columns 15 through 28

     0     0     1     1     1     0     0     1     1     0     0     1     1     0

  Columns 29 through 42

     1     0     1     0     0     0     1     0     0     1     1     0     0     1

  Columns 43 through 56

     0     0     0     1     1     0     1     0     0     1     0     0     0     0

  Columns 57 through 70

     0     0     0     1     0     0     1     1     0     0     1     1     0     1

  Columns 71 through 84

     0     1     0     1     0     1     1     1     1     0     0     1     1     1
  Columns 85 through 98

     1     0     1     0     1     1     0     1     1     0     1     0     1     1

  Columns 99 through 112

     1     1     0     1     0     0     1     0     1     1     0     0     0     0

  Columns 113 through 126

     1     1     1     1     1     1     1     1     1     0     0     0     1     1

  Columns 127 through 140

     1     1     0     1     0     1     0     1     0     1     0     0     1     1

  Columns 141 through 154

     1     1     0     1     1     1     1     0     1     0     1     0     1     1

  Columns 155 through 168

     1     1     1     1     1     1     0     1     1     0     1     0     1     1

  Columns 169 through 182

     0     0     0     0     1     0     1     1     1     1     0     0     0     1

  Columns 183 through 196

     0     0     1     0     1     0     1     1     0     0     1     1     0     0

  Columns 197 through 200

     1     0     1     1

>> uhat=extract_mesg(vhat,rearranged_cols)

uhat =

  Columns 1 through 14

     0     1     0     1     1     0     1     1     0     0     0     0     1     1

  Columns 15 through 28

     1     1     1     1     1     1     1     0     0     0     1     1     1     1

  Columns 29 through 42

     0     1     0     1     0     1     0     1     0     0     1     1     1     1

  Columns 43 through 56

     0     1     1     1     1     0     1     0     1     0     1     1     1     1

  Columns 57 through 70

     1     1     1     1     0     1     1     0     1     0     1     1     0     0

  Columns 71 through 84

     0     0     1     0     1     1     1     1     0     0     0     1     0     0

  Columns 85 through 98

     1     0     1     0     1     1     0     0     1     1     0     0     1     0

  Columns 99 through 100

     1     1
```

APPENDIX B
LDPC SUBROUTINE MATLAB M – FILES

1. **gen_ldpc.m**

```
function [H]=gen_ldpc(rows,cols)
%H=gen_ldpc(rows,cols)

bits_per_col=3;
for i=1:rows
   row_flag(i)=0;
   for j=1:cols
      parity_check(i,j)=0;
   end
end
%add bits_per_col 1's to each column with the only constraint being that
%the 1's should be placed in distinct rows
for i=1:cols
   a=randperm(rows);
   for j=1:bits_per_col
      parity_check(a(j),i)=1;
      row_flag(a(j))=row_flag(a(j))+1;
   end
end
row_flag;
max_ones_per_row=ceil(cols*bits_per_col/rows);
parity_check;


%add 1's to rows having no 1(a redundant row) or only one 1(that bit in
%the codeword becomes
%zero irrespective of the input)
for i=1:rows
   if row_flag(i)==1
      j=unidrnd(cols);
      while parity_check(i,j)==1
         j=unidrnd(cols);
      end
      parity_check(i,j)=1;
      row_flag(i)=row_flag(i)+1;
   end
   if row_flag(i)==0
     for k=1:2
       j=unidrnd(cols);
       while parity_check(i,j)==1
               j=unidrnd(cols);
       end
       parity_check(i,j)=1;
         row_flag(i)=row_flag(i)+1;
      end
   end
end



%try to distribute the ones so that the number of ones per row is as
%uniform as possible

for i=1:rows
   j=1;
   a=randperm(cols);
   while row_flag(i)>max_ones_per_row;
       if parity_check(i,a(j))==1
            parity_check(i,a(j))=0;
            row_flag(i)=row_flag(i)-1;
            newrow=unidrnd(rows);
            k=0;
            while row_flag(newrow)>=max_ones_per_row | parity_check(newrow,a(j))==1
               newrow=unidrnd(rows);
               k=k+1;
               if k>=rows
                  break;
               end
            end
            if parity_check(newrow,a(j))==0
               parity_check(newrow,a(j))=1;
               row_flag(newrow)=row_flag(newrow)+1;
            else
               parity_check(i,a(j))=1;
               row_flag(i)=row_flag(i)+1;
            end
        end%if loop
         j=j+1;
   end%while loop
end%for loop

row_flag;

parity_check;

parity_check;
%try to eliminate cycles of length 4 in the factor graph
for loop=1:10
ones_position(1)=0;
for r=1:rows
   ones_count=0;
   for c=1:cols
      if parity_check(r,c)==1
         ones_count=ones_count+1;
         ones_position(ones_count)=c;
      end
   end
```

```
for i=1:r-1
  common=0;
   for j=1:ones_count
      if parity_check(i,ones_position(j))==1
         common=common+1 ;
         if common==1
            thecol=ones_position(j);
         end
      end
      if common==2
         common=common-1;
         if(round(rand)==0)
            coltoberearranged=thecol;
            thecol=ones_position(j);
         else
            coltoberearranged=ones_position(j);
         end
         parity_check(i,coltoberearranged)=3; %make this entry 3 so
                                              %that we dont use
                                              %of this entry again
                                              %while getting rid
                                              %of other cylces
         newrow=unidrnd(rows);
         %while ((newrow==i)|(parity_check(newrow,ones_position(j))==1))
         iteration=0;
         while parity_check(newrow,coltoberearranged)~=0
            newrow=unidrnd(rows);
            iteration=iteration+1;
            if iteration==5
               break;
            end
         end
         if iteration==5
            while parity_check(newrow,coltoberearranged)==1
               newrow=unidrnd(rows);
            end
         end
         parity_check(newrow,coltoberearranged)=1;
      end
   end
end

for i=r+1:rows
   common=0;
   for j=1:ones_count
      if parity_check(i,ones_position(j))==1
         common=common+1 ;
         if common==1
            thecol=ones_position(j);
         end
      end
      if common==2
```

```
            common=common-1;
            if(round(rand)==0)
               coltoberearranged=thecol;
               thecol=ones_position(j);
            else
               coltoberearranged=ones_position(j);
            end
            parity_check(i,coltoberearranged)=3;%make this entry 3 so that
                                                %we dont use
                                                %of this entry again
                                                %while getting rid
                                                %of other cylces
            newrow=unidrnd(rows);
            %while ((newrow==i)|(parity_check(newrow,ones_position(j))==1))
            iteration=0;
            while parity_check(newrow,coltoberearranged)~=0
               newrow=unidrnd(rows);
               iteration=iteration+1;
               if iteration==5
                  break
                end
             end
             if iteration==5
               while parity_check(newrow,coltoberearranged)==1
                  newrow=unidrnd(rows);
               end
            end
            parity_check(newrow,coltoberearranged)=1;
         end
      end
   end
end
end;
parity_check;

for i=1:rows
   row_flag(i)=0;
   for j=1:cols
      if parity_check(i,j)==1
            row_flag(i)=row_flag(i)+1;
      end
      if eq(parity_check(i,j),3) %replace the 3's with 0's
         parity_check(i,j)=0;
      end
   end
end

variance=var(row_flag)
H=parity_check;
%Get the Parity Checks

%A=0;
```

```
%B=0;
%for i=1:rows
%   for j=1:rows
%      A(i,j)=parity_check(i,j);
%   end
%end
%for i=1:rows
%   for j=rows+1:cols
%      B(i,j-rows)=parity_check(i,j);
%   end
%end
%ainvb=inv(sparse(A))*sparse(B);
%ainvb=inv(A)*B;
%toc
```

2. **rearrange cols.m**

```
function [b,rearranged_cols]=rearrange_cols(A)
%[b,rearranged_cols]=rearrange_cols(A)
%Rearrange the columns of the parity check matrix to get non singular
%A matrix

dim=size(A);
rows=dim(1);
cols=dim(2);
newA=A;
for i=1:rows
   rearranged_cols(i)=0;
end


   for i=1:rows
      if newA(i,i)==0
         k=i+1;
         if k<cols
               %while A(rows,k)==0
             while newA(i,k)==0
               k=k+1;
               if k==cols
                       break
               end
               end
               %if k~=cols
                       temp=newA(1:rows,k);
                       newA(1:rows,k)=newA(1:rows,i);
               newA(1:rows,i)=temp;

               rearranged_cols(i)=k;

               temp=A(1:rows,k);
                       A(1:rows,k)=A(1:rows,i);
               A(1:rows,i)=temp;
               %end
```

```
          end
        end
        for j=1:rows
          if j~=i
              if newA(j,i)==1
                 newA(j,1:cols)=xor(newA(i,1:cols),newA(j,1:cols));
              end
          end
       end
        A;

end

rearranged_cols;
   b=A;
```

3. **inv Gf2.m**

```
function [b]=inv_GF2(A)
%Ainv=inv_GF2(A)

dim=size(A);
rows=dim(1);
cols=dim(2);

for i=1:rows
   for j=1:rows
      unity(i,j)=0;
   end
   unity(i,i)=1;
end

for i=1:rows
   b(1:rows,i)=gflineq(A,unity(1:rows,i),2);
end
```

4. **mul GF2.m**

```
function [c]=mul_GF2(A,B)
%[c]=mul_GF2(A,B)

dim=size(A);
m=dim(1);%no of rows of the first matrix
n=dim(2);%no of cols of the first matrix and no of rows of 2nd matrix
dim=size(B);
p=dim(2);%no of cols of the second matrix

for i=1:m
   for j=1:p
      temp1=A(i,1:n);
      temp2=B(1:n,j);
      prod=temp1.*(temp2.');
      sum1=0;
      for k=1:n
```

```
            sum1=xor(sum1,prod(k));
         end
         c(i,j)=sum1;
      end
   end
```

5. **reorder_bits.m**

```
function [u]= reorder_bits(c,rearranged_cols)
%v= reorder_bits(c,rearranged_cols)

dim=size(rearranged_cols);
rows=dim(2);
for i=rows:-1:1
   if rearranged_cols(i)~=0
      temp=c(i);
      c(i)=c(rearranged_cols(i));
      c(rearranged_cols(i))=temp;
   end
end
u=c;
```

6. **bpsk.m**

```
function [waveform]=bpsk(bitseq,amplitude)
%waveform=bpsk(bitseq,amplitude)

for i=1:length(bitseq)
   if bitseq(i)==1
      waveform(i)=-amplitude;
   else
      waveform(i)=amplitude;
   end
end
```

7. **awgn.m**

```
function [x]=awgn(waveform,No);
%x=awgn(waveform,No);

NoiseStdDev=sqrt(No/2);
x=waveform + NoiseStdDev*randn(1,length(waveform));
```

8. **decode_ldpc_log.m**

```
function [vhat,iteration]=decode_ldpc(rx_waveform,No,amp,h,scale)
%[vhat]=decode_ldpc(rx_waveform,No,amp,h,scale)

dim=size(h);
rows=dim(1);
cols=dim(2);

vhat(1,1:cols)=0;

zero(1,1:rows)=0;
```

```
prevhat(1:cols)=1;

s=struct('alpha_mn',0,'beta_mn',0,'gamma_mn',0);
%associate this structure with all non zero elements of h




%Initialization : set gamma_n to log-likelyhood ratios for every code
%bit and then initialize the alpha_mns for
% all non -zero elements of the parity_check matrix
for j=1:cols
   gamma_n(j)=(4/No)*rx_waveform(j);
   temp=exp(-abs(gamma_n(j)));
   for i=1:rows
      if (h(i,j)==1)
         newh(i,j)=s;
         newh(i,j).alpha_mn=sign(gamma_n(j))*log((1+temp)/(1-temp));

      end
   end
end

for iteration=1:100
        %%%%%%%%%%%begin horizontal step%%%%%%%%%%%%%%%%%%%%%%%5
        for i=1:rows
      prod_of_alpha_mn=1;
      sum_of_alpha_mn=0;
        for j=1:cols
         if h(i,j)==1
            for k=1:cols
               if ((h(i,j)==1)&(j~=k))
                  prod_of_alpha_mn=prod_of_alpha_mn*newh(i,j).alpha_mn;
                  sum_of_alpha_mn=sum_of_alpha_mn+abs(newh(i,j).alpha_mn);
               end
            end
            temp=exp(-sum_of_alpha_mn);
            newh(i,j).beta_mn=sign(prod_of_alpha_mn)*log((1+temp)/(1-temp));
         end
      end
   end
   %%%%%%%%%%%%%%end horizontal step%%%%%%%%%%%%%%%%%%%%%%%%%


   %%%%%%%%%%%%%%%begin vertical step%%%%%%%%%%%%%%%%%
   for j=1:cols
      sum_of_beta_mn=0;
      for i=1:rows
         if (h(i,j)==1)
               sum_of_beta_mn=sum_of_beta_mn+newh(i,j).beta_mn;
         end
      end
```

```
        for i=1:rows
           newh(i,j).gamma_mn=gamma_n(j)+sum_of_beta_mn-newh(i,j).beta_mn;
           temp=exp(-abs(newh(i,j).gamma_mn));
           newh(i,j).alpha_mn=sign(newh(i,j).gamma_mn)*log((1+temp)/(1-temp));
        end

        %%%%%%calculate pseudo log APP ratios
        lambda_n(j)=gamma_n(j)+sum_of_beta_mn;
        if lambda_n(j)>=0
              vhat(j)=0;
           else
              vhat(j)=1;
           end
     end
     %%%%%%%%%%%%%%%%%%%end vertical step%%%%%%%%%
     %%%%%%%%%%stop if v.hat'=0 or if u get the same codeword 20
     %consequtive times
    iteration;
    if prevhat==vhat
        converge=converge+1;
     else
        converge=0;
        prevhat=vhat;
     end

     if mul_GF2(vhat,h.')==zero
        break;
     end

     if converge==20
        break
     end
end

%decoding
```

9. extract mesg.m

```
function [u]= extract_mesg(c,rearranged_cols)
%u= extract_mesg(c,rearranged_cols)

dim=size(rearranged_cols);
rows=dim(2);
dim=size(c);
cols=dim(2);
for i=1:rows
   if rearranged_cols(i)~=0
      temp=c(i);
      c(i)=c(rearranged_cols(i));
      c(rearranged_cols(i))=temp;
   end
end
u=c(rows+1:cols);
```

APPENDIX C
BCH MATLAB SOURCE CODES

```
>> m = 4;
>> n = 2^m-1;
>> k = 5;
>> nwords = 10;
>> msg = gf(randint(nwords,k));
>> [genpoly,t] = bchgenpoly(n,k);
>> code = bchenc(msg,n,k);
>> y=double(code.x);
>> y2=pskmod(y,2);
>> channel=awgn(y2,10);
>> r=pskdemod(channel,2);
>> r2=gf(r);
>> [newmsg,err,ccode] = bchdec(r2,n,k)

newmsg = GF(2) array.

Array elements =

      0        0        1        1        0
      1        1        0        1        0
      0        1        1        1        1
      1        1        1        1        0
      0        1        0        0        0
      1        1        1        1        1
      0        0        1        0        0
      1        0        0        0        0
      1        0        1        1        0
      1        1        1        1        1


err =

     0       0       0       0       0       0       0       0       0       0


ccode = GF(2) array.

Array elements =

  Columns 1 through 14

      0        0        1        1        0        1        1        1        0        0        0        0
1       0
      1        1        0        1        0        1        1        0        0        1        0        0
0       1
      0        1        1        1        1        0        1        0        1        1        0        0
```

```
1       0
      1       1       1       1       0       1       0       1       1       0       0       1
0       0
      0       1       0       0       0       1       1       1       1       0       1       0
1       1
      1       1       1       1       1       1       1       1       1       1       1       1
1       1
      0       0       1       0       0       0       1       1       1       1       0       1
0       1
      1       0       0       0       0       1       0       1       0       0       1       1
0       1
      1       0       1       1       0       0       1       0       0       0       1       1
1       1
      1       1       1       1       1       1       1       1       1       1       1       1
1       1

  Column 15

      1
      1
      0
      0
      0
      1
      1
      1
      0
      1

>> if ccode==code
disp('All errors were corrected.')
end
All errors were corrected.
>> if newmsg==msg
disp('The message was recovered perfectly.')
end
The message was recovered perfectly.
```

APPENDIX D
BCH SUBROUTINE MATLAB SOURCE CODES

1. **bchgenpoly.m**

```
function [genpoly, t] = bchgenpoly(N,K,varargin);
%BCHGENPOLY  Generator polynomial of BCH code.

% Initial checks
error(nargchk(2,3,nargin));

t = bchnumerr(N,K);
t2 = 2*t;

prim_poly = 1;

m = log2(N+1);

if ~isempty(varargin)
    prim_poly = varargin{1};
    % Check prim_poly
    if isempty(prim_poly)
        if ~isnumeric(prim_poly)
            error('To use the default PRIM_POLY, it must be marked by [].');
        end
    else
        if  ~isnumeric(prim_poly)  ||  ~isscalar(prim_poly)  ||  (floor(prim_poly)  ~=
prim_poly)
            error('PRIM_POLY must be a scalar integer.');
        end

        if ~isprimitive(prim_poly)
            error('PRIM_POLY must be a primitive polynomial.');
        end
    end

end

% Alpha is the primitive element of this GF(2^m) field
if prim_poly == 1
    alpha = gf(2,m);
else
    alpha = gf(2,m,prim_poly);
end

% genpoly = LCM([1 alpha.^k])... for  k = 1 : 2t-1)

% Find all the minimun polynomials, add them to list of minimum
% polynomials, if they're not there yet. Then convolve all the minimum
% polynomials to make the generator polynomial.
```

```
minpol_list = minpol(alpha);

for k=[1:t2-1]
    minpoly = minpol(alpha.^k);

    [len,w] = size(minpol_list);
    minpol_mat = repmat(minpoly, [len 1]);

    eq = (minpol_mat == minpol_list);
    if(~any(sum(eq') == w))
        minpol_list = [minpol_list;minpoly];
    end
end

% convolve all the rows of the minpol_list with each other.
len = size(minpol_list,1);
genpoly  = 1;
for(i = 1:len)
    genpoly = conv(genpoly,minpol_list(i,:));
end

% strip any leading zeros
% the size of the generator polynomial should be N-K+1
genpoly = genpoly( end-(N-K) :end);
```

2. **bchenc.m**

```
function code = bchenc(msg, N, K, varargin)
%BCHENC BCH encoder.

% Initial checks
error(nargchk(3,4,nargin));

% Number of optional input arguments
nvarargin = nargin - 3;

% % Fundamental checks on parameter data types
if ~isa(msg,'gf')
   error('MSG must be a Galois array.');
end

if(msg.m ~=1)
    error('MSG must be in GF(2).');
end

%set and check the parity position
if(nargin>3)
    parityPos = varargin{1};
else
    parityPos = 'end';
end
```

```
if( ~strcmp(parityPos,'beginning') && ~strcmp(parityPos, 'end') )
    error('PARITYPOS must be either ''beginning'' or ''end''  ')
end


[m_msg, n_msg] = size(msg);

if (n_msg ~= K)
    error('The message length must equal K.')
end



% get the generator polynomial
genpoly = bchgenpoly(N,K);

% get the generator matrix
[h, gen] = cyclgen(N, (double(genpoly.x)));

% do the coding
code = msg * gen;

% rearrange parity if necessary
%if(isempty(varargin) || strcmp(lower(varargin{1}), 'beginning'))
if(strcmp(parityPos, 'end'))
    code = [code(:,N-K+1:end), code(:,1:N-K)];
end
```

3. pskmod.m

```
function y = pskmod(x,M,varargin)
%PSKMOD Phase shift keying modulation
% Error checks
if (nargin > 3)
    error('comm:pskmod:numarg', 'Too many input arguments. ');
end

% Check that x is a positive integer
if (~isreal(x) || any(any(ceil(x) ~= x)) || ~isnumeric(x))
    error('comm:pskmod:xreal', 'Elements of input X must be integers in [0, M-1].');
end

% Check that M is a positive integer
if (~isreal(M) || ~isscalar(M) || M<=0 || (ceil(M)~=M) || ~isnumeric(M))
    error('comm:pskmod:Mreal', 'M must be a positive integer. ');
end

% Check that M is of the form 2^K
if(~isnumeric(M) || (ceil(log2(M)) ~= log2(M)))
    error('comm:pskmod:Mpow2', 'M must be in the form of M = 2^K, where K is an
integer. ');
end

% Check that x is within range
```

```
if ((min(min(x)) < 0) || (max(max(x)) > (M-1)))
    error('comm:pskmod:xreal', 'Elements of input X must be integers in [0, M-1].');
end

% Determine initial phase. The default value is 0
if (nargin == 3)
    ini_phase = varargin{1};
    if (isempty(ini_phase))
        ini_phase = 0;
    elseif (~isreal(ini_phase) || ~isscalar(ini_phase))
        error('comm:pskmod:ini_phaseReal', 'INI_PHASE must be a real scalar. ');
    end
else
    ini_phase = 0;
end

% --- Assure that X, if one dimensional, has the correct orientation --- %
wid = size(x,1);
if (wid == 1)
    x = x(:);
end

% Evaluate the phase angle based on M and the input value. The phase angle
% lies between 0 - 2*pi.
theta = 2*pi*x/M;

% The complex envelope is (cos(theta) + j*sin(theta)). This can be
% expressed as exp(j*theta). If there is an initial phase, it is added
% to the existing phase angle
y = exp(j*(theta + ini_phase));

% --- restore the output signal to the original orientation --- %
if(wid == 1)
    y = y.';
end
```

4. awgn.m

```
function y=awgn(varargin)
%AWGN Add white Gaussian noise to a signal.
% --- Initial checks
error(nargchk(2,5,nargin));

% --- Value set indicators (used for the string flags)
pModeSet    = 0;
measModeSet = 0;

% --- Set default values
reqSNR   = [];
sig      = [];
sigPower = 0;
pMode    = 'db';
measMode = 'specify';
```

```
state    = [];

% --- Placeholder for the signature string
sigStr = '';

% --- Identify string and numeric arguments
for n=1:nargin
   if(n>1)
      sigStr(size(sigStr,2)+1) = '/';
   end
   % --- Assign the string and numeric flags
   if(ischar(varargin{n}))
      sigStr(size(sigStr,2)+1) = 's';
   elseif(isnumeric(varargin{n}))
      sigStr(size(sigStr,2)+1) = 'n';
   else
      error('Only string and numeric arguments are allowed.');
   end
end

% --- Identify parameter signatures and assign values to variables
switch sigStr
   % --- awgn(x, snr)
   case 'n/n'
      sig      = varargin{1};
      reqSNR   = varargin{2};

   % --- awgn(x, snr, sigPower)
   case 'n/n/n'
      sig      = varargin{1};
      reqSNR   = varargin{2};
      sigPower = varargin{3};

   % --- awgn(x, snr, 'measured')
   case 'n/n/s'
      sig      = varargin{1};
      reqSNR   = varargin{2};
      measMode = lower(varargin{3});

      measModeSet = 1;

   % --- awgn(x, snr, sigPower, state)
   case 'n/n/n/n'
      sig      = varargin{1};
      reqSNR   = varargin{2};
      sigPower = varargin{3};
      state    = varargin{4};

   % --- awgn(x, snr, 'measured', state)
   case 'n/n/s/n'
      sig      = varargin{1};
      reqSNR   = varargin{2};
      measMode = lower(varargin{3});
      state    = varargin{4};

      measModeSet = 1;

   % --- awgn(x, snr, sigPower, 'db|linear')
   case 'n/n/n/s'
      sig      = varargin{1};
      reqSNR   = varargin{2};
      sigPower = varargin{3};
      pMode    = lower(varargin{4});

      pModeSet = 1;

   % --- awgn(x, snr, 'measured', 'db|linear')
   case 'n/n/s/s'
      sig      = varargin{1};
      reqSNR   = varargin{2};
      measMode = lower(varargin{3});
      pMode    = lower(varargin{4});

      measModeSet = 1;
      pModeSet    = 1;

   % --- awgn(x, snr, sigPower, state, 'db|linear')
   case 'n/n/n/n/s'
      sig      = varargin{1};
      reqSNR   = varargin{2};
      sigPower = varargin{3};
      state    = varargin{4};
      pMode    = lower(varargin{5});

      pModeSet = 1;

   % --- awgn(x, snr, 'measured', state, 'db|linear')
   case 'n/n/s/n/s'
      sig      = varargin{1};
      reqSNR   = varargin{2};
      measMode = lower(varargin{3});
      state    = varargin{4};
      pMode    = lower(varargin{5});

      measModeSet = 1;
      pModeSet    = 1;

   otherwise
      error('Syntax error.');
end

% --- Parameters have all been set, either to their defaults or by
%the values passed in, so perform range and type checks
```

```
% --- sig
if(isempty(sig))
   error('An input signal must be given.');
end

if(ndims(sig)>2)
   error('The input signal must have 2 or fewer dimensions.');
end

% --- measMode
if(measModeSet)
   if(~strcmp(measMode,'measured'))
      error('The signal power parameter must be numeric or ''measured''.');
   end
end

% --- pMode
if(pModeSet)
   switch pMode
   case {'db' 'linear'}
   otherwise
      error('The signal power mode must be ''db'' or ''linear''.');
   end
end

% -- reqSNR
if(any([~isreal(reqSNR) (length(reqSNR)>1) (length(reqSNR)==0)]))
   error('The signal-to-noise ratio must be a real scalar.');
end

if(strcmp(pMode,'linear'))
   if(reqSNR<=0)
      error('In linear mode, the signal-to-noise ratio must be > 0.');
   end
end

% --- sigPower
if(~strcmp(measMode,'measured'))

   % --- If measMode is not 'measured', then the signal power must be specified
   if(any([~isreal(sigPower) (length(sigPower)>1) (length(sigPower)==0)]))
      error('The signal power value must be a real scalar.');
   end

   if(strcmp(pMode,'linear'))
      if(sigPower<0)
         error('In linear mode, the signal power must be >= 0.');
      end
   end

end
```

```
% --- state
if(~isempty(state))
   if(any([~isreal(state)     (length(state)>1)     (length(state)==0)     any((state-
floor(state))~=0)]))
      error('The State must be a real, integer scalar.');
   end
end

% --- All parameters are valid, so no extra checking is required

% --- Check the signal power.  This needs to consider power measurements on matrices
if(strcmp(measMode,'measured'))
   sigPower = sum(abs(sig(:)).^2)/length(sig(:));

   if(strcmp(pMode,'db'))
      sigPower = 10*log10(sigPower);
   end
end

% --- Compute the required noise power
switch lower(pMode)
   case 'linear'
      noisePower = sigPower/reqSNR;
   case 'db'
      noisePower = sigPower-reqSNR;
      pMode = 'dbw';
end

% --- Add the noise
if(isreal(sig))
   opType = 'real';
else
   opType = 'complex';
end

y = sig+wgn(size(sig,1), size(sig,2), noisePower, 1, state, pMode, opType);
```

5. <u>pskdemod.m</u>

```
function z = pskdemod(y,M,varargin)
%PSKDEMOD Phase shift keying demodulation
% Error checks
if (nargin > 3)
    error('comm:pskdemod:numarg', 'Too many input arguments. ');
end

%Check y, m
if( ~isnumeric(y))
    error('comm:pskdemod:Ynum','Y must be numeric.');
end

% Checks that M is positive integer
if (~isreal(M) || ~isscalar(M) || M<=0 || (ceil(M)~=M) || ~isnumeric(M))
```

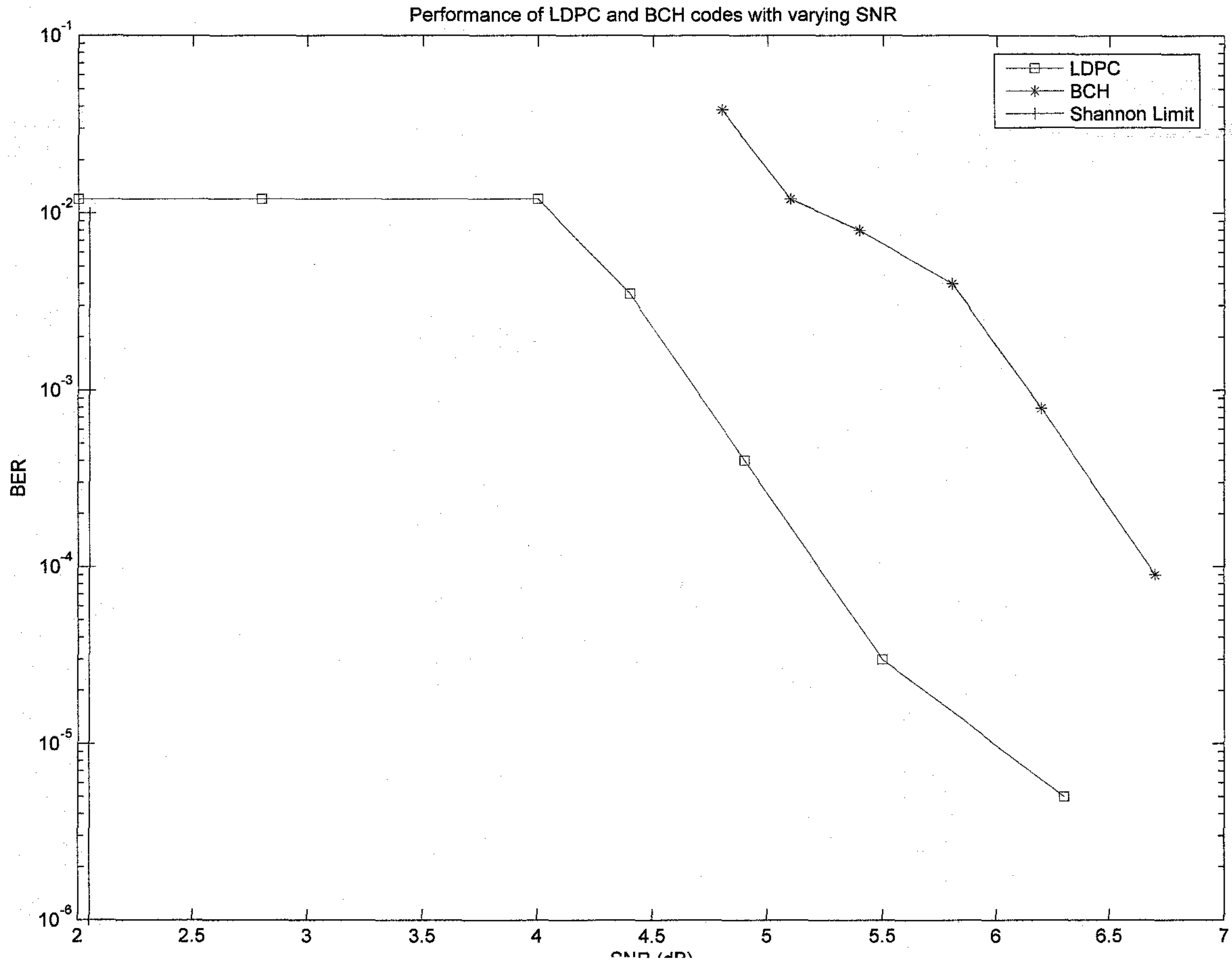
```
    error('comm:pskdemod:Mreal', 'M must be a positive integer. ');
end

% Checks that M is in of the form 2^K
if(~isnumeric(M) || (ceil(log2(M)) ~= log2(M)))
    error('comm:pskdemod:Mpow2', 'M must be in the form of M = 2^K, where K is an
integer. ');
end

% Determine INI_PHASE. The default value is 0
if (nargin == 3)
    ini_phase = varargin{1};
    if (isempty(ini_phase))
        ini_phase = 0;
    elseif (~isreal(ini_phase) || ~isscalar(ini_phase))
        error('comm:pskdemod:Ini_phaseReal', 'INI_PHASE must be a real scalar. ');
    end
else
    ini_phase = 0;
end

% generate a constellation
const = pskmod(0:M-1,M, ini_phase);

%demodulate.
z = genqamdemod(y,const);
```

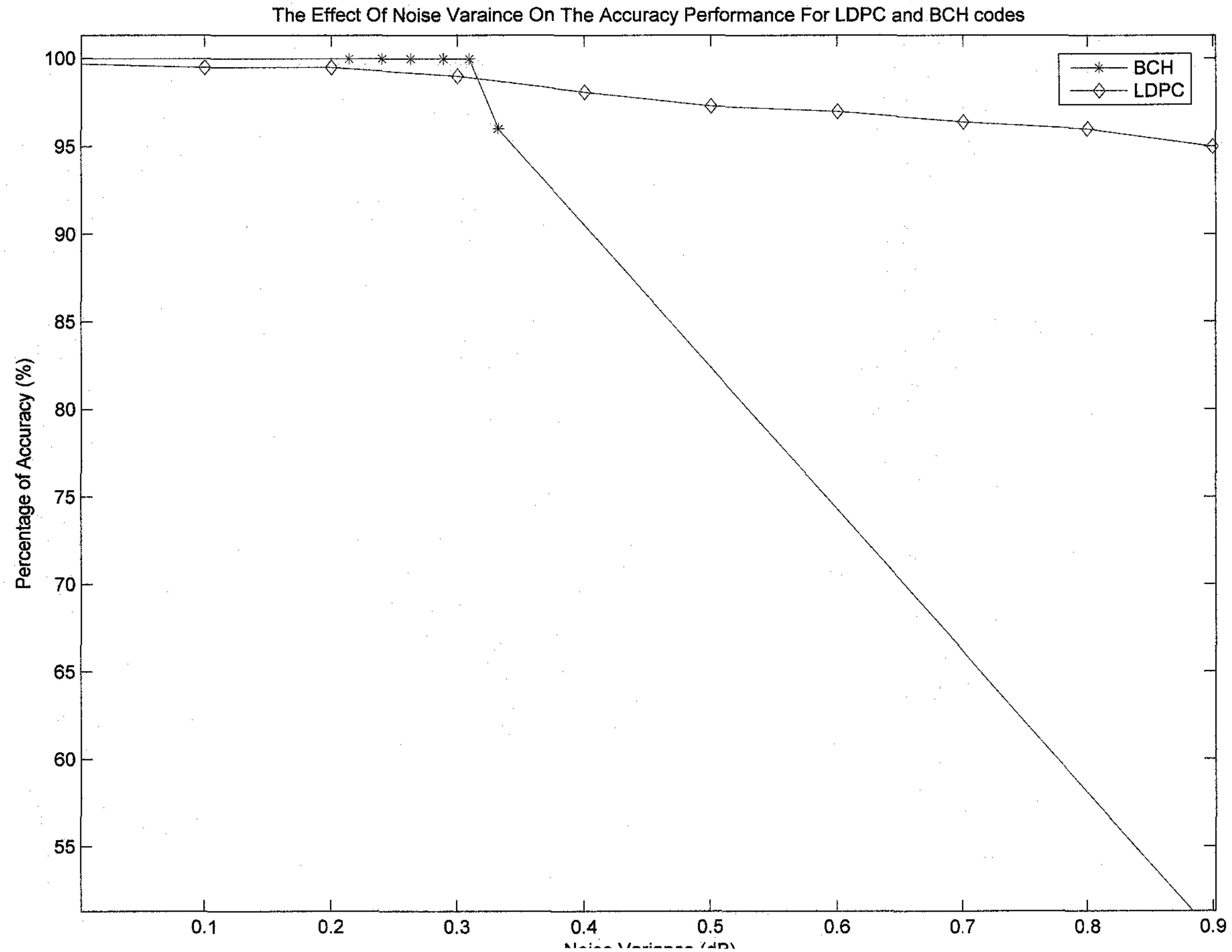
6. bchdec.m

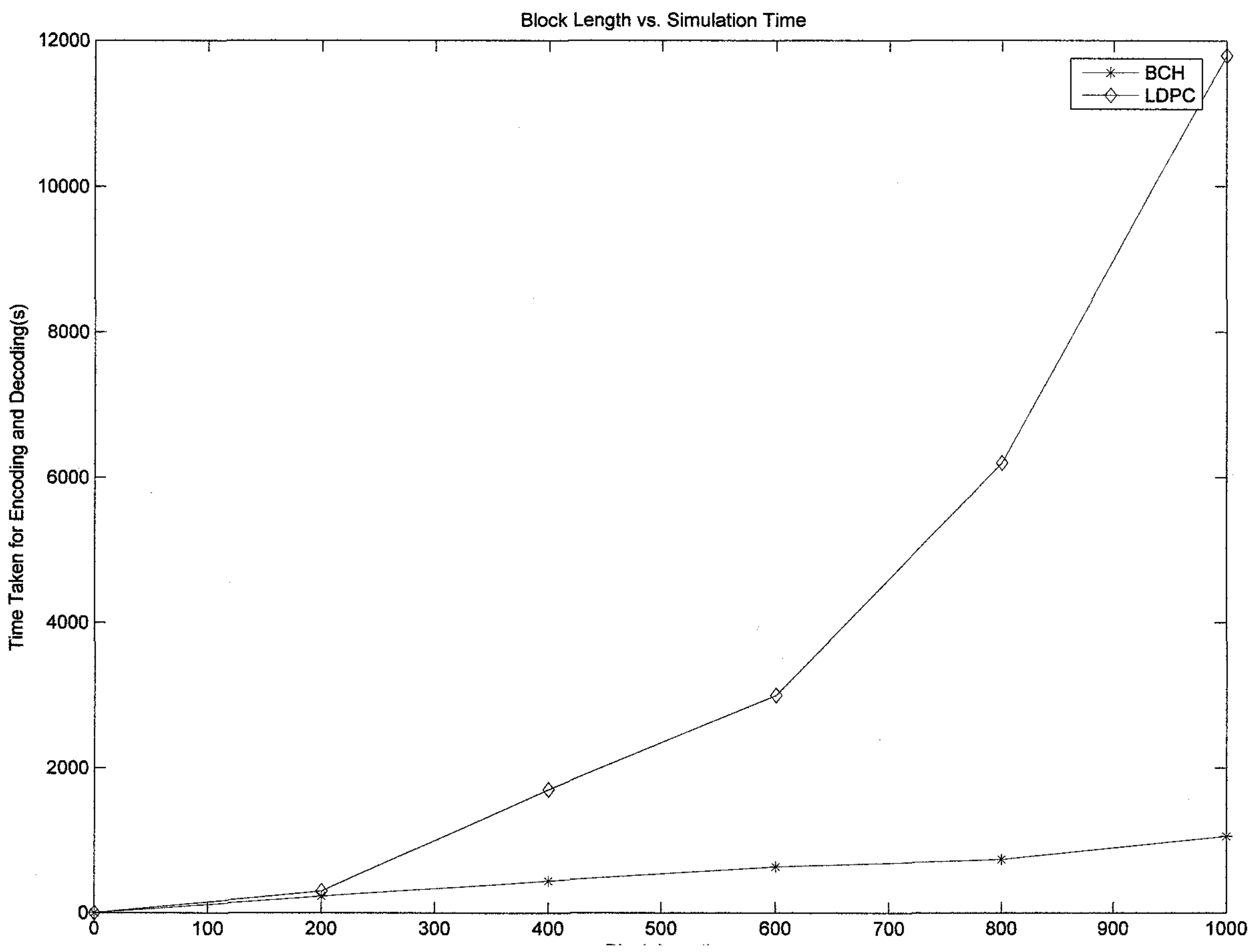
```
function [decoded, cnumerr,ccode] = bchdec(coded,N,K, varargin);
%BCHDEC BCH decoder.

error(nargchk(3,4,nargin));

% Fundamental checks on parameter data types
if ~isa(coded,'gf')
   error('CODE must be a Galois array.');
end

if(coded.m~=1)
    error('Code must be in GF(2).');
end

[m_code, n_code] = size(coded);

% Check mandatory parameters : code, N, K, t

% --- code
if isempty(coded.x)
    error('CODE must be a nonempty Galois array.');
end;

% --- width of code
```

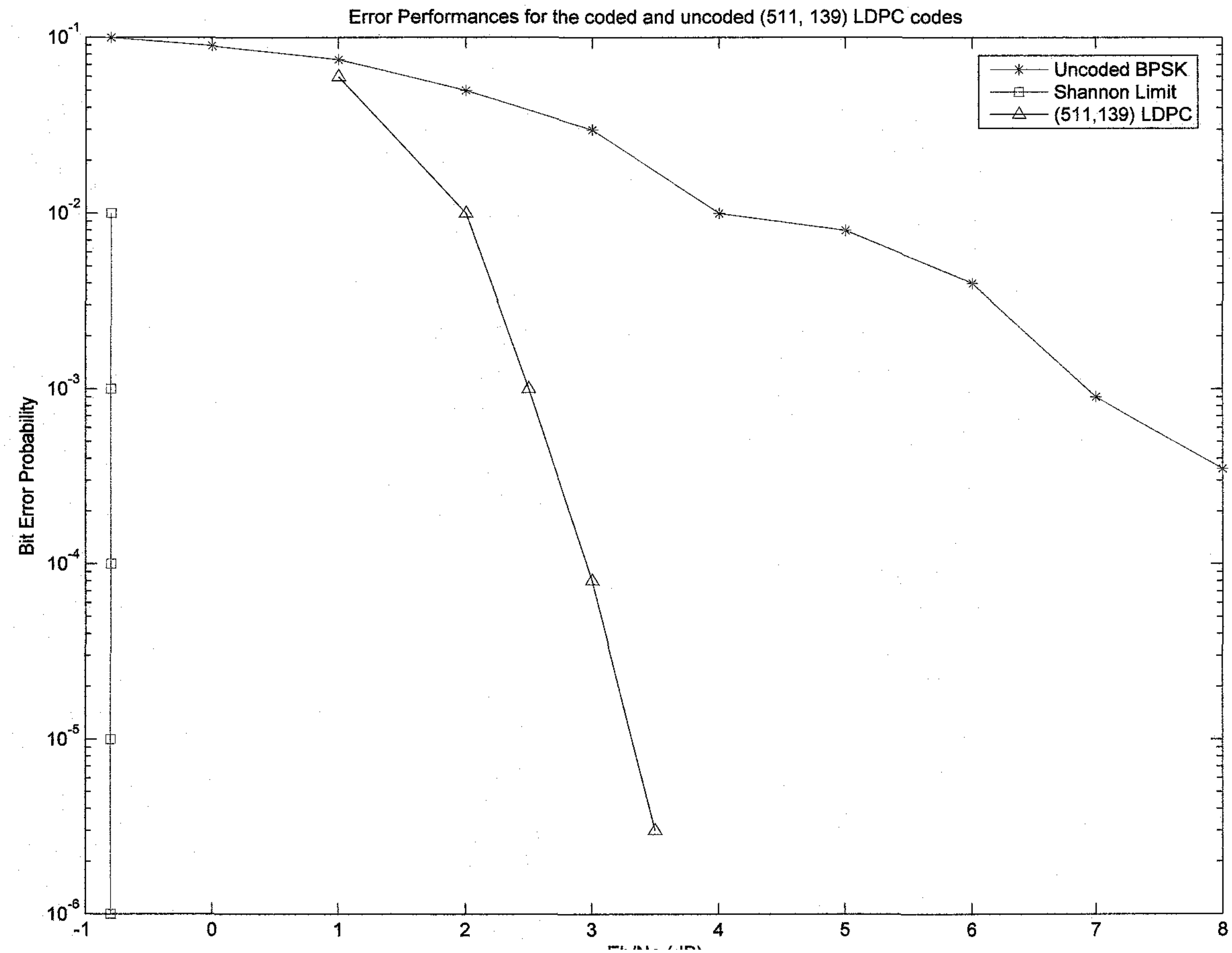
```
if N ~= n_code
    error('CODE must be either a N-element row vector or a matrix with N columns.');
end

%set and check the parity position
if(nargin>3)
    parityPos = varargin{1};
else
    parityPos = 'end';
end

if( ~strcmp(parityPos,'beginning') && ~strcmp(parityPos, 'end') )
    error('PARITYPOS must be either ''beginning'' or ''end''  ')
end

% get the number of errors we can correct
t = bchnumerr(N,K);

M = log2(N+1);
% bring the coded word into the extension field
coded = gf(coded.x,M);
[m_code, n_code] = size(coded);

for j=1:m_code,
    % Call to core algorithm BERLEKAMP
    [coded(j,:) cnumerr(j)] = berlekamp(coded(j,:),N,K,t,1,'bch');
end

% bring back to gf(2)
ccode = gf(coded.x);

switch parityPos
   case 'end'
      decoded = ccode(:,1:K);
   case 'beginning'
      decoded = ccode(:,N-K+1:end);
end;
```

APPENDIX E
SIMULATION RESULTS FOR LDPC AND BCH CODES

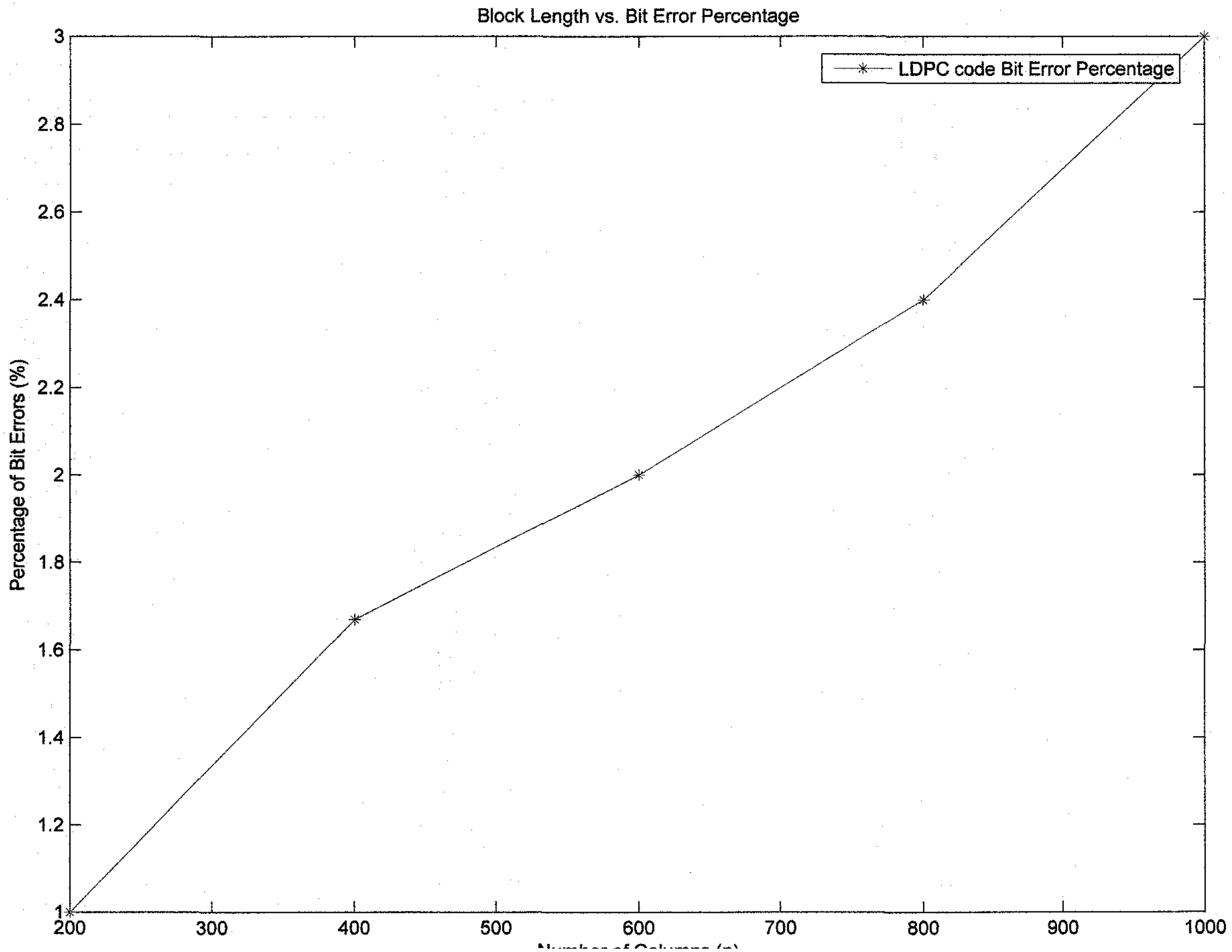

Performance of LDPC and BCH codes with varying SNR
LDPC
BCH
Shannon Limit
BER
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
2
2.5
3
3.5
4
4.5
5
5.5
6
6.5
7
SNR (dB)


The Effect Of Noise Varaince On The Accuracy Performance For LDPC and BCH codes
Percentage of Accuracy (%)
100
95
90
85
80
75
70
65
60
55
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
BCH
LDPC

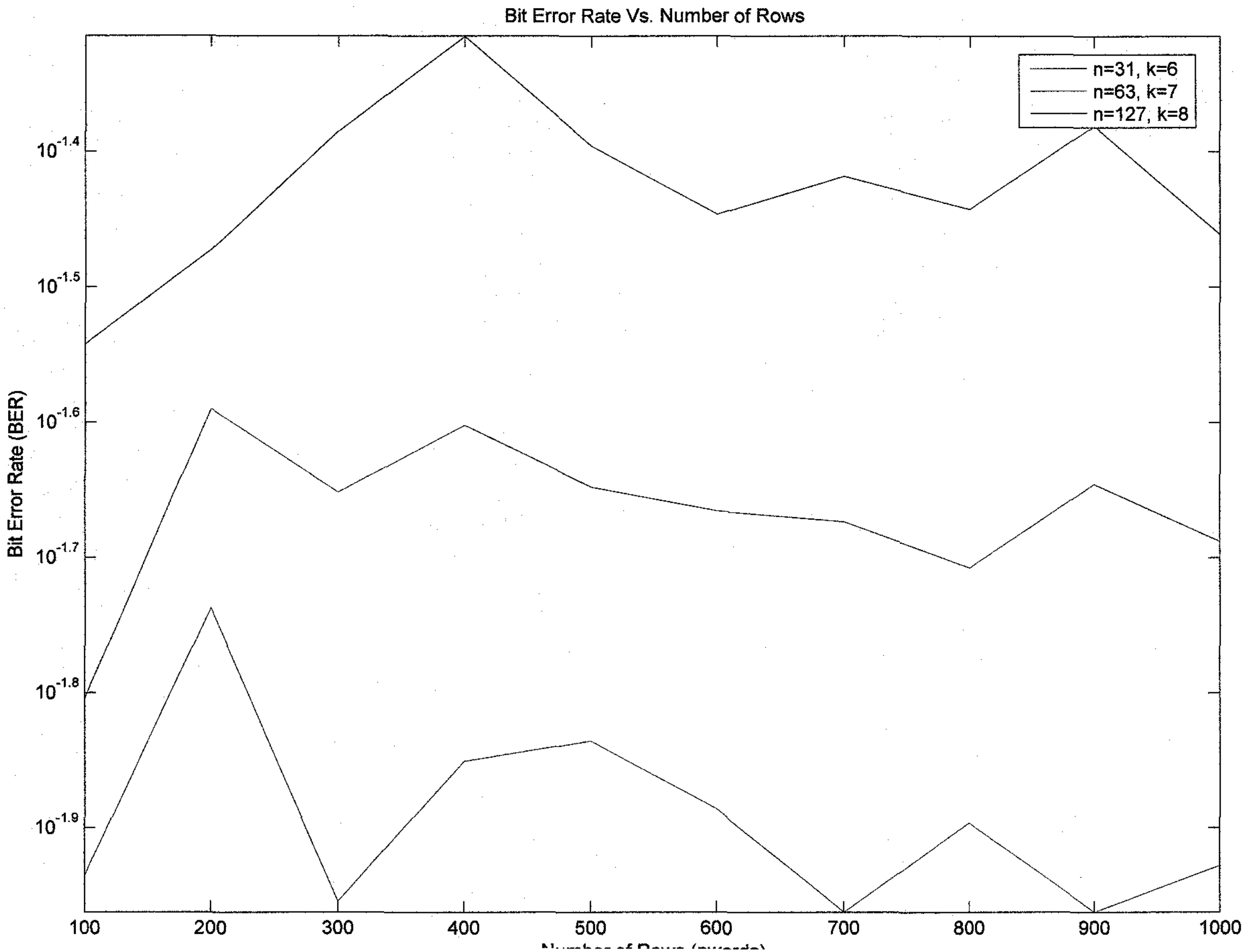

Block Length vs. Simulation Time
Time Taken for Encoding and Decoding(s)
BCH
LDPC
0
2000
4000
6000
8000
10000
12000
0
100
200
300
400
500
600
700
800
900
1000

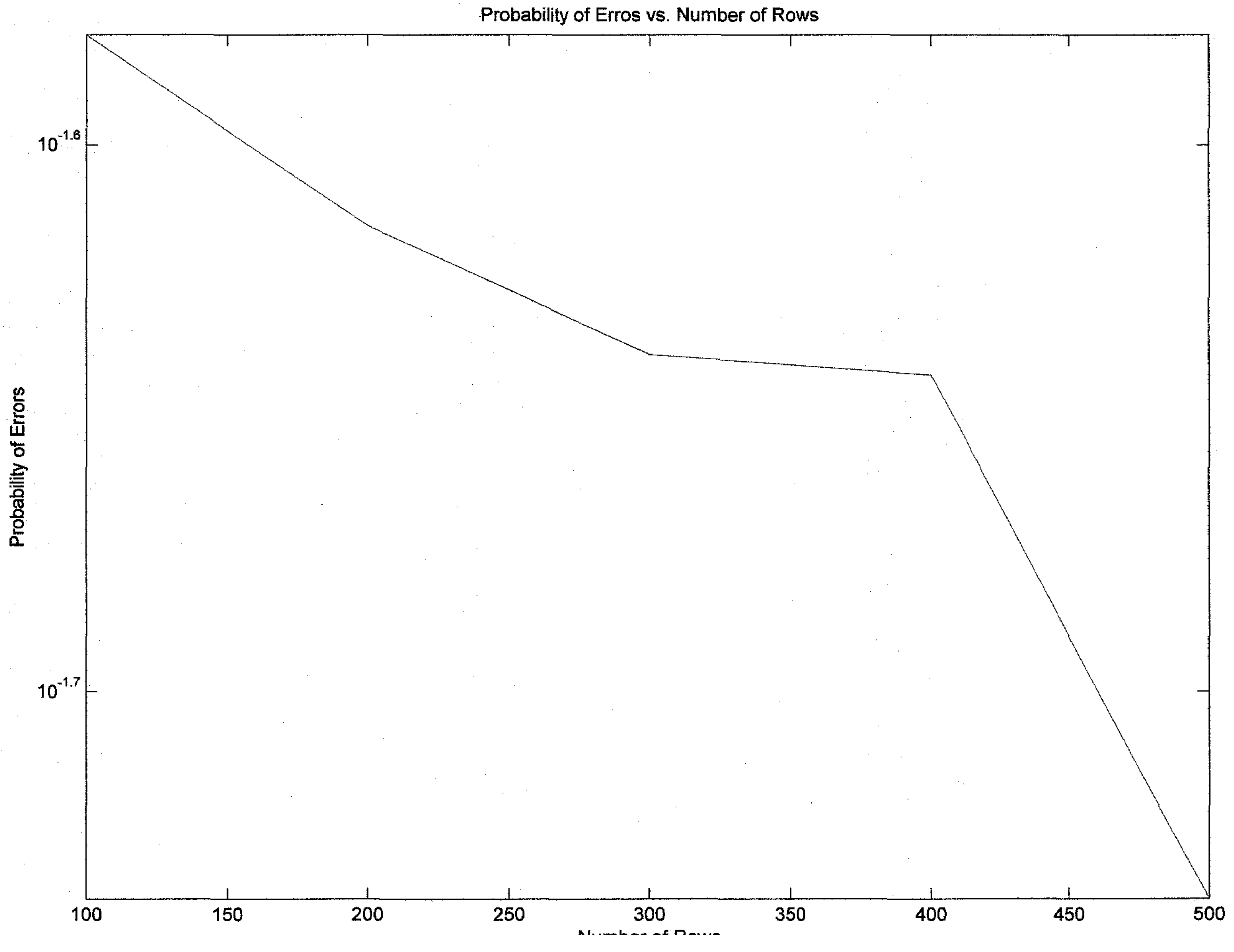


Error Performances for the coded and uncoded (511, 139) LDPC codes
Uncoded BPSK
Shannon Limit
(511,139) LDPC
Bit Error Probability
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
-1
0
1
2
3
4
5
6
7
8



Block Length vs. Bit Error Percentage
LDPC code Bit Error Percentage
Percentage of Bit Errors (%)
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
200
300
400
500
600
700
800
900
1000


Bit Error Rate Vs. Number of Rows
n=31, k=6
n=63, k=7
n=127, k=8
Bit Error Rate (BER)
$10^{-1.4}$
$10^{-1.5}$
$10^{-1.6}$
$10^{-1.7}$
$10^{-1.8}$
$10^{-1.9}$
100
200
300
400
500
600
700
800
900
1000


Probability of Erros vs. Number of Rows
Probability of Errors
$10^{-1.6}$
$10^{-1.7}$
100
150
200
250
300
350
400
450
500