A NEW DISTANCE-BASED ALGORITHM FOR BLOCK TURBO CODES: FROM CONCEPT TO IMPLEMENTATION

Nong Le

A Thesis

In

The Department

Of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science at

Concordia University

Montreal, Quebec, Canada

Fall 2005

© Nong Le, 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-10240-3
Our file Notre référence
ISBN: 0-494-10240-3

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

A New Distance-Based Algorithm for Block Turbo Codes: from Concept to Implementation

Nong Le

List-based algorithms for decoding *Block Turbo Codes* (BTC) have gained popularity due to their low computational complexity. The normal way to calculate the soft outputs involves searching for a decision code word $\boldsymbol{D}$ and a competing codeword $\boldsymbol{B}$. In addition, a scaling factor α and an estimated reliability value β are used. In this thesis, we present a new approach that does not require α and β. Soft outputs are generated based on the Euclidean distance property of decision code words. More importantly, such algorithm has very low computational complexity and is very attractive for practical applications. Based on the synthesis result of FPGA (*Field Programmable Gate Array*) implementations of the new algorithm, significant complexity saving (up to 79%) is achieved compared to commercially available products. In terms of error performance, we observe certain improvement (0.3dB coding gain) for BTCs of large Hamming distance and negligible performance degradation for BTCs of short Hamming distance.

DEDICATION

This thesis is dedicated to my dear wife

Yue Wang

and my lovely Son

Ryan Le

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to my advisors Dr. M. Reza Soleymani and Dr. Y. R. Shayan for their guidance and encouragement during the entire period of this research work. Their technical acumen, precise suggestions and timely discussions is heartily appreciated.

Financial support for this research by the Natural Sciences and Engineering Research Council of Canada (NSERC) is greatly appreciated. Special thanks go to the staff of the Department of Electrical and Computer Engineering at Concordia University for having been so helpful and supportive.

Many thanks to my parents and in-laws, for their endless support during my studies. I wish to accord my special thanks to my darling wife: Yue Wang. Without her support and understanding, I would not have had the persistence to finish my studies. I dedicate this work to my lovely son, Ryan Le, for being a constant source of joy and a tireless companion through the years.

TABLE OF CONTENTS

LIST OF TABLES viii

LIST OF FIGURES ix

LIST OF ABBREVIATIONS xi

1 **Introduction** 1

1.1 Literature Review of Error Control Coding 1

1.2 Main Contributions 5

1.3 Outline of This Thesis 6

2 **Block Turbo Codes** 8

2.1 Introduction 8

2.2 Encoding Structure of a Block Turbo Code 9

2.3 Decoding of Block Turbo Codes 10

2.3.1 Decoding-Based Algorithm 12

2.3.2 Encoding-Based Algorithm 17

2.3.3 Trellis-Based Algorithm 20

3 **Distance-Based Decoding of Block Turbo Codes** 24

3.1 Introduction 24

3.2 Drawbacks of the Traditional Algorithm 25

3.3 The Distance-Based Algorithm 29

3.3.1 Confidence Value 30

3.3.2 Soft-Output Calculation 35

3.3.3 Algorithm Comparison 39

3.4 Monte Carlo Simulation Using C++ 40

3.5 Simulation Results 49

3.6 Discussions and Conclusions 55

4 **FPGA Implementation of Distance-Based Decoding Algorithm** 57

4.1 Introduction 57

4.2 Review of FPGA Technologies 58

4.2.1 Basic Structure of FPGA 58

4.2.2 FPGA Design Flow 59

4.2.3 Introduction of Xilinx Virtex II Family 60

4.3 Fixed-Point Representation of the Algorithm 64

4.4 FPGA Implementation of the Distance-Based Algorithm 66

4.4.1 Elementary Decoder 67

4.4.2 Memory Design 70

4.4.3 Iterative Decoding Architecture 72

4.5 Synthesis Results 74

4.5.1 Minimization of Surface Area 74

4.5.2 Maximization of Data Rate 75

4.5.3 Optimization of Error Performance 76

5 **Conclusions and Future Work** 78

5.1 Conclusions 78

5.2 Future Directions 79

Bibliography 81

Appendix 85

LIST OF TABLES

3.1 Function map between $Dist_{des}$ and Φ 35

3.2 Function descriptions 47

4.1 Virtex-II FPGA family members 61

4.2 Soft outputs lookup table 69

4.3 Comparison to TC3021 (surface area) 75

4.4 Comparison to TC3021 (data rate) 76

4.5 Comparison to TC3024 (performance) 76

LIST OF FIGURES

1.1	Structure of a turbo encoder	4
1.2	Structure of a turbo decoder	4
2.1	Construction of a BTC	9
2.2	Block diagram of a block turbo encoder	10
2.3	Generating a list of candidate codewords for a (5,3) code using the order-i reprocessing algorithms	19
2.4	Trellis structure of a block codes	21
3.1	Examples of augmented list decoding with high SNR	27
3.2	Examples of augmented list decoding with low SNR	28
3.3	Noise distributions for a simple binary block code	32
3.4	Confidence value versus destructive Euclidean distance for BTC $(64,51,6)^2$	34
3.5	Block diagram of the decoding procedure	38
3.6	Extrinsic information structure	39
3.7	Parameter setting for a BTC$(64,51,6)^2$	41
3.8	Block diagram of the software simulation	42
3.9	Simulation of AWGN channel	43
3.10	BCH decoding using Berlekamp algorithm	45
3.11	BER versus Eb/No of BTC$(64,51,6)^2$ on a Gaussian channel using BPSK signaling at iteration 4	52

3.12 BER versus Eb/No of BTC$(32,21,6)^2$ on a Gaussian channel using BPSK signaling at iteration 4 52

3.13 BER versus Eb/No of BTC$(64,57,4)^2$ on a Gaussian channel using BPSK signaling at iteration 4 53

3.14 BER versus Eb/No of BTC$(32,26,4)^2$ on a Gaussian channel using BPSK signaling at iteration 4 53

3.15 Comparison with other algorithms for BTC$(64,51,6)^2$ on a Gaussian channel using signaling at iteration 4 54

3.16 Comparison with other algorithms for BTC$(64,51,6)^2$ on a Gaussian channel using signaling at iteration 4 54

4.1 FPGA Design Flow 59

4.2 Internal structure of Virtex II 62

4.3 Virtex II general slice diagram 63

4.4 4-bit uniformed quantization scheme 64

4.5 Quantization results for BTC$(64,51,6)^2$ after 5 iterations 66

4.6 Elementary decoder (half-iteration) 67

4.7 Interleaver example 71

4.8 Single-elementary-decoder structure 72

4.9 Pipeline structure 73

4.10 Bit rate vs. complexity trade-off 77

LIST OF ABBREVIATIONS

2D	2-Dimensional
3D	3-Dimensional
APP	A Posteriori Probability
AWGN	Additive White Gaussian Noise
BCH	Bose-Chaudhuri-Hocquenghem
BCJR	Bahl-Cocke-Jelinek-Raviv
BER	Bit Error Rate
BMA	Box and Match Algorithm
BPSK	Binary Phase Shift Keying
BTC	Block Turbo Code
CLB	Configurable Logic Block
DCM	Digital Clock Manager
DVB	Digital Video Broadcast
ED	Elementary Decoder
FPGA	Field Programmable Gate Array
LLR	Log-Likelihood Ratio
LRB	Least Reliable Bit
LUT	Look-Up Tables
MAP	Maximum A Posteriori

MLD	Maximum Likelihood Decoder
MRIP	Most Reliable Independent Position
OSD	Ordered Statistic Decoding
PDF	Probability Distribution Function
RM	Reed Muller
RS	Reed Solomon
RSC	Recursive Systematic Convolutional
SISO	Soft-Input Soft-Output
SNR	Signal-to-Noise Ratio
TPC	Turbo Product Code
VHDL	VHSIC Hardware Description Language

Chapter 1

Introduction

1.1 Literature Review of Error Control Coding

Error control coding is an essential ingredient in a high performance digital communications system. Its main function, as the name suggests, is to reduce the number of reception errors by adding redundancy to a signal at the transmitter and correcting errors at the receiver. The inclusion of redundancy in the transmitted signal results in a coded signal consisting of more bits than the original signal. In return, this overhead enables the receiver to detect and correct the errors.

The history of coding theory can be traced back to 1948, when Shannon presented his research work [1] in the fields of information and coding theory. In this famous paper, he introduced the new concept of *"Channel Capacity"*, which denotes the upper bound of any practical communication system. Also Shannon indicated that, ignoring the complexity of the coding scheme, there exist certain systems which can achieve arbitrarily reliable communication as long as the transmitted signal rate do not exceed the channel capacity.

The earlier research in coding theory mainly focused on two main divisions: *Block Codes* and *Convolutional Codes*.

An *(n, k)* block code is defined as a mapping of a k-dimensional information sequence into an n-dimensional data sequence. Most of powerful block codes, like *Bose-Chaudhuri-Hocquenghem* (BCH) codes and *Reed Solomon* (RS) codes, use a mathematical construct known as a finite field or *Galois Field* (GF). A Galois field contains a set of field elements associated with arithmetic operations including addition and multiplication. The block encoder operates on the message symbols over a Galois field and maps them into codeword symbols. In a well-designed coding system, this mapping process spreads the set of codewords evenly into the vector space, resulting in large Hamming distance between the codewords. In the past 50 years, many decoding algorithms have been proposed for block codes. In 1965, a range of powerful decoding algorithms for BCH codes were found by Berlekamp [2] and Massey [3]. Later, Chase [4] announced a series of algorithms which can achieve near maximum likelihood decoding of block codes by using channel measurement information. In 1978, Wolf [5] discovered a new method of constructing trellis diagram for linear block codes. Due to the high complexity, his work did not receive enough attention until 1988 when Forney [6] proved that it is possible to construct relatively simple trellis structure for certain block codes.

Convolutional codes, found by Elias in 1955 [7], constitute another important coding family that is widely used in many communication systems. These codes get their name because the encoding process can be viewed as the convolution of the message symbols

and the impulse response of the encoder. Unlike block codes, convolutional codes operate on serial data, one or few bits at a time. The optimum decoding scheme for convolutional codes is the maximum likelihood decoding where the decoder attempts to find the closest valid sequence to the received data stream. The most popular algorithm is Viterbi algorithm [8], which is implemented by tracking likely paths through a "trellis" structure and choosing the most likely path as the output data sequence.

A more powerful code that is constructed by concatenating an RS code as an outer code and a convolutional code as an inner code can take the advantages of both codes [9]. Due to its excellent performance, this coding scheme is widely used in many applications such as deep space, satellite and wireless communications.

A breakthrough in coding history is the invention of *Turbo Codes,* which were first introduced by Berrou, Glavieux and Thitimajshima [10] in 1993. This new coding scheme consists of two *Recursive Systematic Convolutional* (RSC) encoders arranged in a so-called parallel concatenation along with a pseudorandom interleaver as shown in Figure 1.1. The turbo encoder processes the information vector twice but in a different sequence due to presence of an interleaver. Also, in order to increase the code rate, a puncturer is often used to reduce coding overhead by deleting a few parity bits. The structure of a turbo decoder is depicted in Figure 1.2, where there are two decoders corresponding to the two encoders. The inputs to the first decoder consist of the observed systematic bits, the parity bit stream from the first encoder and the deinterleaved extrinsic information from the second decoder. Similarly, the inputs to the second decoder consist

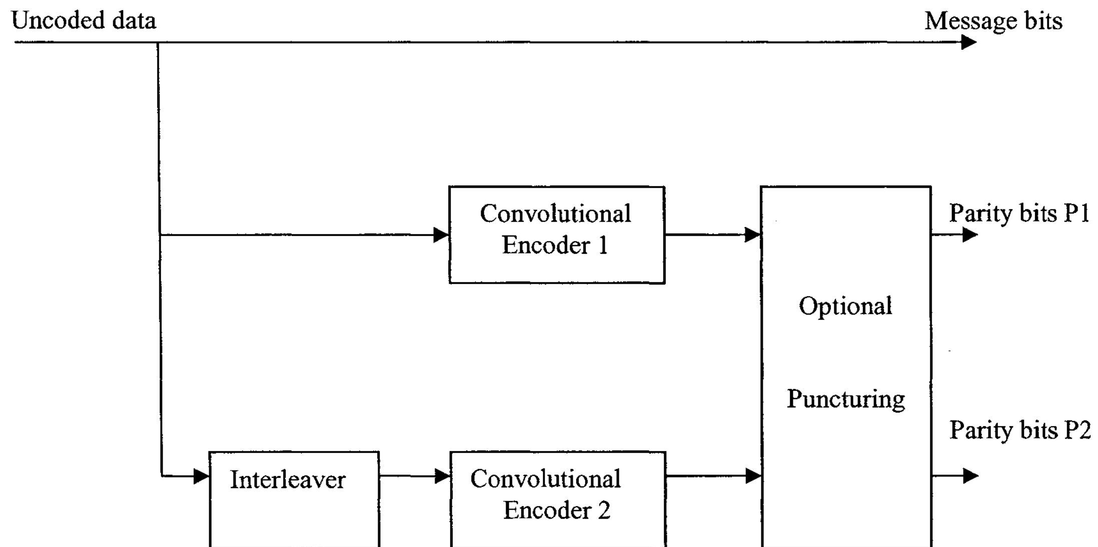


Figure 1.1: Structure of a turbo encoder

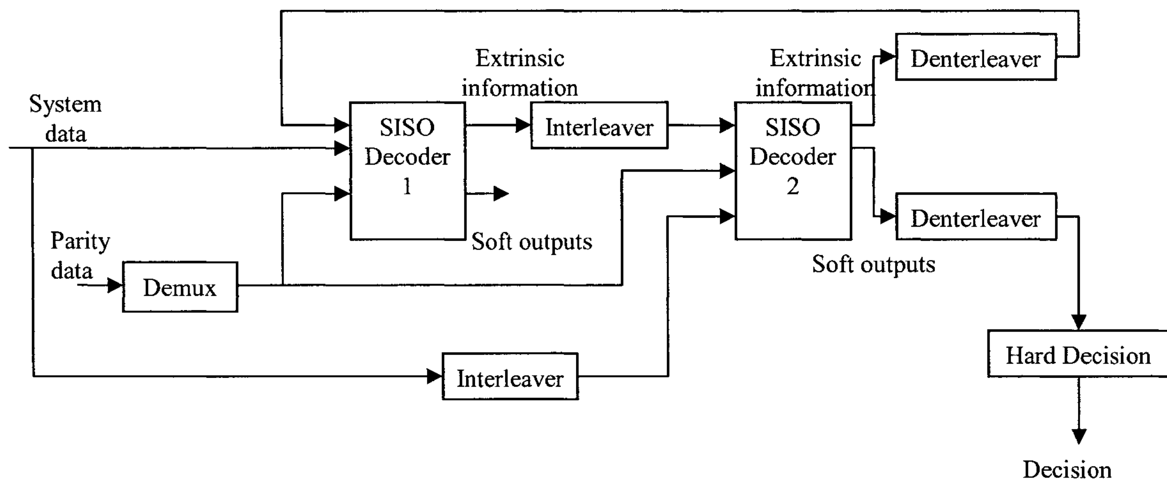


Figure 1.2: Structure of a turbo decoder

of the interleaved systematic bit stream, the observed parity bit stream from the second encoder and the interleaved extrinsic information from the first decoder. Each component decoder computes the *A Posteriori Probability* (APP) of the information symbols which is the reliability value for the information symbols. The sequence of reliability values is

then passed to the other decoder. To improve the correctness of its decisions, each decoder has to be fed with information that does not originate from itself. This decoding scheme is able to achieve very high performance. In [10], it is reported that 0.5dB above the Shannon limit at a *Bit Error Rate* (BER) of 10^{-5} is achievable by using a code rate ½ turbo code with a large block length $N=2^{16}$ and 18 iterations. Inspired by Berrou's work, an intensive research activity was initiated [11] [12] [13].

One year after the birth of the turbo codes, Pyndiah and *et al* proposed a new coding-decoding scheme [14] which extends the turbo concept to block product codes. They call the new codes as *Block Turbo Codes* (BTCs). The algorithm provides an efficient solution for coding scheme requiring high code rates (R>0.8). For investigated codes, the *Signal to Noise Ratio* (SNR) for a BER of 10^{-5} was at 2.5 dB of their Shannon limits. The algorithm is improved by [15] [16] [17] [18] [19] and implementations are shown in [20] [21] [22].

1.2 Main Contributions

The focus of this work is towards developing a low complexity decoding methodology for BTCs. The methodology starts from high level models which can be used for software solution and proceeds towards high performance hardware solutions. During the study, we have created 4 sets of C++ programs for software simulations and 2 sets of VHDL (*VHSIC Hardware Description Language*) codes for hardware implementations. Main contributions and steps taken in this thesis are as follows.

- After deep understanding of related work, numerous methods are tried to improve the decoding algorithm for BTCs. These efforts lead to the discovery of a new algorithm, called *Distance-Based* algorithm. A float point version of the decoding algorithm coded in C++ (see appendix) is designed for investigating the performance of the new algorithm.
- In order to study the hardware complexity of the algorithm, the effect of quantization on the performance is investigated and a set of C++ programs is developed for obtaining the simulation results in the fixed point model.
- Finally, the new algorithm is implemented on a real chip. FPGA is selected as a device platform and VHDL language is used to describe the targeting system. The circuit design process is done through system modeling, functional simulation, VHDL synthesis, place & route, system integration, and device fabrication. These tasks are performed by using several digital design tools including Xilinx XST, ModelSim, and Synplify Pro. In addition, a C++ simulation program is developed to verify the result.

1.3 Outline of this Thesis

In Chapter 2, a fairly comprehensive introduction on block turbo codes is provided. Various attributes of block turbo codes and corresponding decoding technologies, including trellis-based algorithms and list-based algorithms, are discussed.

In Chapter 3, drawbacks of the traditional decoding algorithm for BTCs are explained and a new algorithm which resolves these drawbacks is introduced. The effect of varying

various parameters such as code rate, number of test patterns, and number of iterations is presented. A software program coded in C++ is developed to evaluate the error performance and compared with the traditional algorithm. We demonstrate that the new algorithm is beneficial in terms of error performance, data rate, and especially computational complexity.

Chapter 4 mainly discusses hardware implementation of the new algorithm. In this chapter, we briefly review the digital design technology with a focus on Xilinx Virtex II FPGA platform. Several possible decoder architectures are proposed and synthesis results are compared to existing products in terms of data rate, occupied area and error performance.

In Chapter 5, the conclusion of our work on low complexity decoding algorithm for BTCs is summarized and future research directions are indicated.

Chapter 2

Block Turbo Codes

2.1 Introduction

A *Block Turbo Code* (BTC) is a product code obtained from the concatenation of two or more block codes with iterative decoding technique. In some literature, it is also referred to as a *Turbo Product Code* (TPC). *Soft-Input Soft-Output* (SISO) decoding methods that can be applied to BTCs are divided into two categories: trellis-based algorithms [11] and augmented list algorithms [14] [15] [23] [24] [25].

This chapter is organized as follows. Section 2.2 describes the details of the encoder schematic of BTCs. Section 2.3 discusses the decoding technologies that are available for BTCs: particularly, two categories of decoding methods including augmented list algorithms and trellis-based algorithms will be discussed with detailed mathematical derivations.

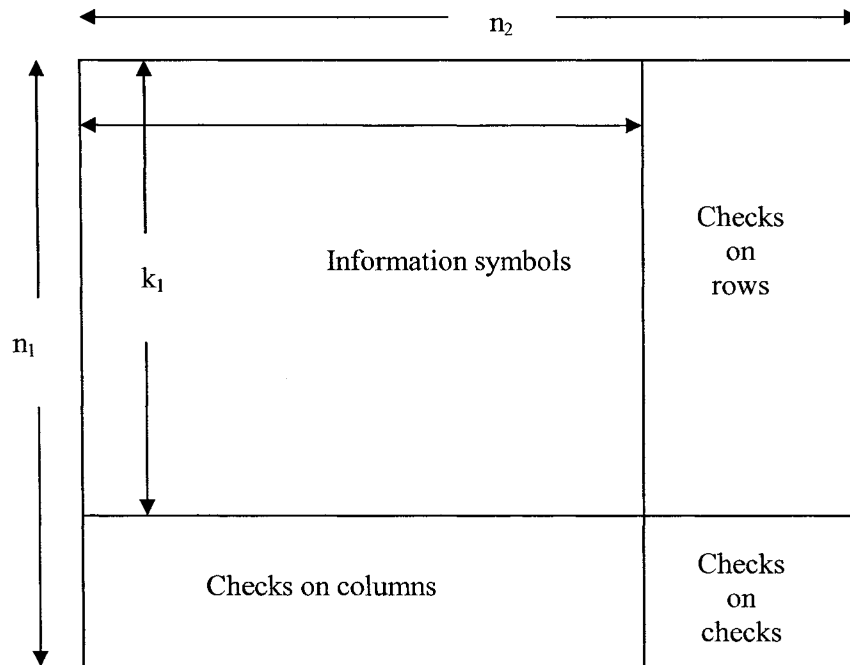


Figure 2.1: Construction of a BTC

2.2 Encoding Structure of a Block Turbo Code

Constructing a BTC requires concatenating several block codes. The most common BTCs in practical use are 2-dimensional (2D) BTCs and 3-dimensional (3D) BTCs, which consist of two or three linear block codes, respectively. Generally speaking, the inner component codes of a BTC can be any block codes like Hamming codes, BCH codes, and RS codes.

Figure 2.1 illustrates a typical code structure of a BTC constructed by serially concatenating two systematic linear block codes C^1 with parameters (n_1, k_1, δ_1) and C^2 with parameters (n_2, k_2, δ_2), where n_i, k_i, δ_i $(i=1,2)$ represent the code length, the number of information bits, and the minimum Hamming distance respectively. Information

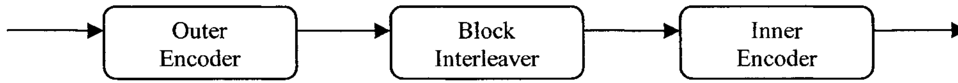


Figure 2.2: Block diagram of a block turbo encoder

symbols are arranged into k_1 rows and k_2 columns. A BTC encoder first processes k_1 rows using code C^2 and then processes n_2 columns using code C^1. Clearly, the resulting codes have a code rate of $(k_1 \cdot k_2)/(n_1 \cdot n_2)$. Figure 2.2 shows the block diagram of such an encoder.

In this thesis, we mainly discuss 2-dimensional BTCs that use BCH codes as the component codes. For simplicity, we assume that the two component codes have the same parameters, denoted as $BTC(n, k, \delta)^2$.

2.3 Decoding of Block Turbo Codes

During the decoding of a BTC, component codes on each row (or column) are always processed separately. Therefore, unless otherwise stated, each decoding stage operates on component codes.

Let us assume that a codeword of a (n, k, δ) linear block code $\boldsymbol{C}=\{c_1, c_2, \ldots, c_n\}$ is transmitted as $\boldsymbol{X}=\{x_1, x_2, \ldots, x_n\}$, where $x_j = +1$ if $c_j = 1$ and $x_j = -1$ if $c_j = 0$, over an *Additive White Gaussian Noise* (AWGN) channel. Denote the received vector $\boldsymbol{R}=\{r_1, r_2, \ldots, r_n\}$, where $r_j = x_j + n_j$, $j=1,2,\ldots,n$, and n_j is a Gaussian random variable with standard deviation σ.

For a *Maximum Likelihood Decoder* (MLD), the optimal decision $\boldsymbol{D}$ is obtained by

$$D = C^i \quad \text{if} \, |R - C^i|^2 < |R - C^\ell|^2 \;\; \ell \in [1, 2^k] \;\; \ell \neq i \tag{2.1}$$

where $\{C^i\}$ is a set of all valid codewords.

The major obstacle in SISO decoding of BTCs is generating soft outputs that are required during the iterations. There are two different SISO decoding methods available for BTCs. One is trellis-based algorithm and the other is augmented list algorithm.

Trellis-based algorithm is an extension of *Bahl-Cocke-Jelinek-Raviv* (BCJR) algorithm [26], which can achieve a performance of an optimal decoding. It is proved that certain BTCs can be represented by trellis diagrams [27] [28], and consequently, state sequence of a discrete-time finite Markov process in a memoryless channel can be estimated. Trellis-based algorithms have similar mathematical derivations to the traditional convolutional turbo codes. However, the major drawback of trellis-based algorithms is their large complexity. The current technology of representing a block code by a trellis diagram is only available for those codes with short code length. As the code length increases, the complexity of trellis-based algorithms usually becomes too large to be practical for implementations.

Augmented list decoding algorithms are low complexity algorithms that can be used for BTCs with no restrictions on their code length. These algorithms calculate soft outputs based on a list of candidate codewords. Depending on the techniques used for generating

the codewords list, augmented list algorithms can be further divided into two categories: decoding-based list algorithms and encoding-based list algorithms.

More details of these decoding algorithms are described in next subsections.

2.3.1 Decoding-based List Algorithm

When Pyndiah [14] introduced the concept of block turbo codes, he initially proposed an augmented list algorithm which uses Chase II algorithm to generate a candidate codeword list. The key idea of Chase II algorithm [4] is to reduce the computational complexity by limiting the code word search procedure within a sphere of radius $(\delta -1)$ centered on a binary sequence $\boldsymbol{Y}=(y_1, y_2, \ldots.. y_n)$ where $y_j= 0.5(1+sgn(r_j)), i=1,2,\ldots,n$. Soft information is computed based on a list of available valid codewords. This algorithm uses two parameters, a scaling factor α and an estimated reliability value β, whose roles are essential in the performance. The author reports that more than 98% of channel capacity can be reached with high code rate BTCs.

For practical considerations, [17] presents a way to reduce the complexity of the turbo decoder by a factor of ten compared to [14] at a cost of 0.7 performance loss. Two practical prototypes of this decoder were presented in [20]. In [18], a fast Chase algorithm was proposed to reduce the computational complexity of the Chase decoding procedure as well.

It is worth mentioning that our new decoding method for BTCs is developed based on the original algorithm [14]. Therefore, in this section, we will focus our discussion on the original algorithm that is proposed by Pyndiah [14].

The original algorithm starts with generating a list of the most possible codewords based on the received sequence $\boldsymbol{R}$. Decoding procedure using Chase II algorithm can be described as follows [4].

1. Determine the position of p *Least Reliable Binary* (LRB) symbols, where reliability value $\Lambda'(r_j)$ is the *Log-Likelihood Ratio* (LLR) normalized by $2/\sigma^2$ as

$$\Lambda'(r_j) = |r_j| \tag{2.2}$$

2. Form 2^p error patterns E defined as all combination of patterns with '0' or '1' in p LRB positions.
3. Form 2^p test patterns as $T=Y\oplus E$, where $\oplus$ denotes modulo 2 addition operation.
4. Decode all test patterns using an algebraic decoder and save the resulting codewords in a set $\mathcal{C}$.
5. Find a decision codeword $\boldsymbol{D}= (d_1, d_2, \ldots, d_n)$ in set $\mathcal{C}$, which has the minimum squared Euclidean distance with $\boldsymbol{R}$.

Given decision codeword $\boldsymbol{D}$, the reliability of the decision bit d_j defined by LLR can be expressed as:

$$\Lambda(d_j) = \ln\left(\frac{P(x_j = +1 \mid R)}{P(x_j = -1 \mid R)}\right) \tag{2.3}$$

where

$$P_r\{x_j = +1 \mid R\} = \sum_{C^i \in S_j^{+1}} P_r\{X = C^i \mid R\} \tag{2.4}$$

and

$$P_r\{x_j = -1 \mid R\} = \sum_{C^i \in S_j^{-1}} P_r\{X = C^i \mid R\} \tag{2.5}$$

where S_j^{+1} and S_j^{-1} are the sets containing codewords in $\mathcal{C}$ such that $c_j^i=+1$ and $c_j^i=-1$, respectively. Assuming that the different codewords are uniformly distributed, we obtain the LLR of d_j as:

$$\Lambda(d_j) = \ln\left(\frac{\sum_{C^i \in S_j^{+1}} p\{R \mid X = C^i\}}{\sum_{C^i \in S_j^{-1}} p\{R \mid X = C^i\}}\right) \tag{2.6}$$

where

$$p\{R \mid X = C^i\} = \left(\frac{1}{\sqrt{2\pi}\sigma}\right)^n \exp\left(-\frac{|R - C^i|^2}{2\sigma^2}\right) \tag{2.7}$$

is the P*robability Density Function* (pdf) of $\boldsymbol{R}$ conditioned on $\boldsymbol{X}$. Let $C^{+1(j)}$ and $C^{-1(j)}$ be the codewords closest to $\boldsymbol{R}$ in S_j^{+1} and S_j^{-1}, respectively. We get

$$\Lambda(d_j) = \frac{1}{2\sigma^2}\left(|R - C^{-1(j)}|^2 - |R - C^{+1(j)}|^2\right) + \ln\left(\frac{\sum_i A_i}{\sum_i B_i}\right) \tag{2.8}$$

where

$$A_i = \exp\left(\frac{|R - C^{+1(j)}|^2 - |R - C^i|^2}{2\sigma^2}\right) \le 1 \quad with\ C^i \in S^{+1(j)} \tag{2.9}$$

and

$$B_i = \exp\left(\frac{|R - C^{-1(j)}|^2 - |R - C^i|^2}{2\sigma^2}\right) \le 1 \quad with\ C^i \in S^{-1(j)} \tag{2.10}$$

For high signal to noise ratio in Gaussian channel, since $\sigma \rightarrow 0$, $\sum_i A_i \approx \sum_i B_i \rightarrow 0$, the second term in (2.8) tends to be zero, and the LLR of d_j could be approximated by

$$\Lambda(d_j) = \frac{1}{2\sigma^2}(|R - C^{-1(i)}|^2 - |R - C^{+1(i)}|^2) \tag{2.11}$$

In Equation (2.11), two codewords are required for computing the reliability of the decision bit d_j. Obviously, one is the decision codeword $\boldsymbol{D}$ and the other one is the *Competing Codeword* $\boldsymbol{B}$ with minimum squared Euclidean distance from $\boldsymbol{R}$ where $b_j \neq d_j$. Thus, Equation (2.11) can also be expressed in terms of $\boldsymbol{B}$ and $\boldsymbol{D}$ as

$$\Lambda(d_j) = \frac{1}{2\sigma^2}(|R - B|^2 - |R - D|^2) \cdot d_j \tag{2.12}$$

This equation can be further simplified as

$$\Lambda(d_j) = \frac{2}{\sigma^2}(r_j + \sum_{\ell=1,\ell \neq j}^{n} r_\ell c_\ell^{+1(j)} p_\ell) \tag{2.13}$$

where

$$p_\ell = \begin{cases} 0 & if \quad c_\ell^{+1(i)} = c_\ell^{-1(i)} \\ 1 & if \quad c_\ell^{+1(i)} \neq c_\ell^{-1(i)} \end{cases} \tag{2.14}$$

Normalizing $\Lambda(d_j)$ by $2/\sigma^2$, we get

$$r_j' = r_j + w_j \tag{2.15}$$

where w_j is the extrinsic information with

$$w_j = \sum_{\ell=1,\ell \neq j}^{n} r_\ell c_\ell^{+1(j)} p_\ell \tag{2.16}$$

Since Chase II algorithm only considers 2^p codeword candidates, it is likely that some of the positions have no competing codeword. Therefore, a reliability factor β is introduced to estimate the average reliability of decision bit according to decoding step t. This parameter is experimentally predetermined where in [14]

$$\beta(t)=[0.2,0.4,0.6,0.8,1.0,1.0,1.0,1.0] \tag{2.17}$$

For each stage, we perform decoding for each row of product code, and calculate the soft output for each bit. Extrinsic information of rows W^{-} can be expressed as

$$W^{-} = R^{-}_{output} - R^{-}_{input} \tag{2.18}$$

Then, W^{-} is passed to the next decoding step using soft input as

$$R^{|}_{input} = R+\alpha W^{-} \tag{2.19}$$

where α is a scaling factor representing the reliability of the extrinsic information and increases as decoding step t increases. They are also decided by experimental simulation [14] as

$$\alpha\,(t)=[0.0,0.2,0.3,0.5,0.7,0.9,1.0,1.0] \tag{2.20}$$

Similarly, the extrinsic information of columns $W^{|}$ is given by

$$W^{|}= R^{|}_{output} - R^{|}_{input} \tag{2.21}$$

This sequence of computations is repeated for each iteration by each of the two decoders. After all iterations are complete, the decoded information bit can be retrieved by simply looking at the sign bit of the soft output: if it is positive the bit is a one, if it is negative the bit is a zero.

2.3.2 Encoding-Based List Algorithm

Decoding-based list algorithms can achieve an error performance close to the channel capacity when working with high code rate BTCs. However, as the Hamming distance of the BTC increases, using the Chase algorithm to construct the candidate list is very inefficient. This results in poor performance when we try to decode low code rate BTCs with decoding-based list algorithms. Recently, some researchers proposed few augmented list algorithms [23] [24] [25] which use encoding-based techniques for generating the candidates list. These algorithms improve performance by examining a large portion of valid codewords. One major disadvantage is that the algorithms are usually much more complex and memory intensive. This limits their use in practical applications.

Here, we only present the most basic algorithm, namely the *order-i reprocessing algorithm*, which is developed in 1998 [23]. Many other encoding-based algorithms, such as the *Box and Match Algorithm (BMA)* [24] and the *Ordered Statistic Decoding (OSD)* algorithm [25], can be viewed as modified versions of the order-i reprocessing algorithm.

In an order-i reprocessing algorithm, the list of candidate codewords is generated by encoding a set of possible information vectors. It is achieved by following steps.

1. Reorder the soft inputs from least to most reliable. This will create a reordered systematic code, where the k *Most Reliable Independent Positions* (MRIPs) are the information bits.
2. Find the generator matrix $\boldsymbol{G'}$ of the new code based on the reordering process.

3. Form test information sequence as $T=Y\oplus E$, where $\oplus$ denotes modulo 2 addition operation, and E is all possible weight-l error patterns in k MRIPs with $1 \leq l \leq i$.
4. Encode all test patterns using generator matrix $\boldsymbol{G'}$ and calculate squared Euclidean distance between the codeword and received sequence $\boldsymbol{R}$.
5. Find decision codeword $\boldsymbol{D'}=(d_1, d_2,.., d_n)$, which has the minimum squared Euclidean distance with $\boldsymbol{R}$.

To get a clear understanding of the algorithm, we may see a simple example which illustrates the list-generating process for a *(5, 3)* linear block code using the order-*2* reprocessing algorithm. As shown in Figure 2.4 (a), all symbols in the received sequence are evaluated based on their reliability values and 3 symbols (x_1, x_3, x_4) are determined as the MRIPs. At the same time, a new generator matrix associated with these MRIPs is calculated and will be used for encoding the test sequences in the next steps. Since $i=2$, both 1-error patterns and 2-error patterns are taken in considerations as shown in Figure 2.4 (b) (c) (d). As a result, by encoding these test sequences we obtain a candidates list consisting of seven valid codewords.

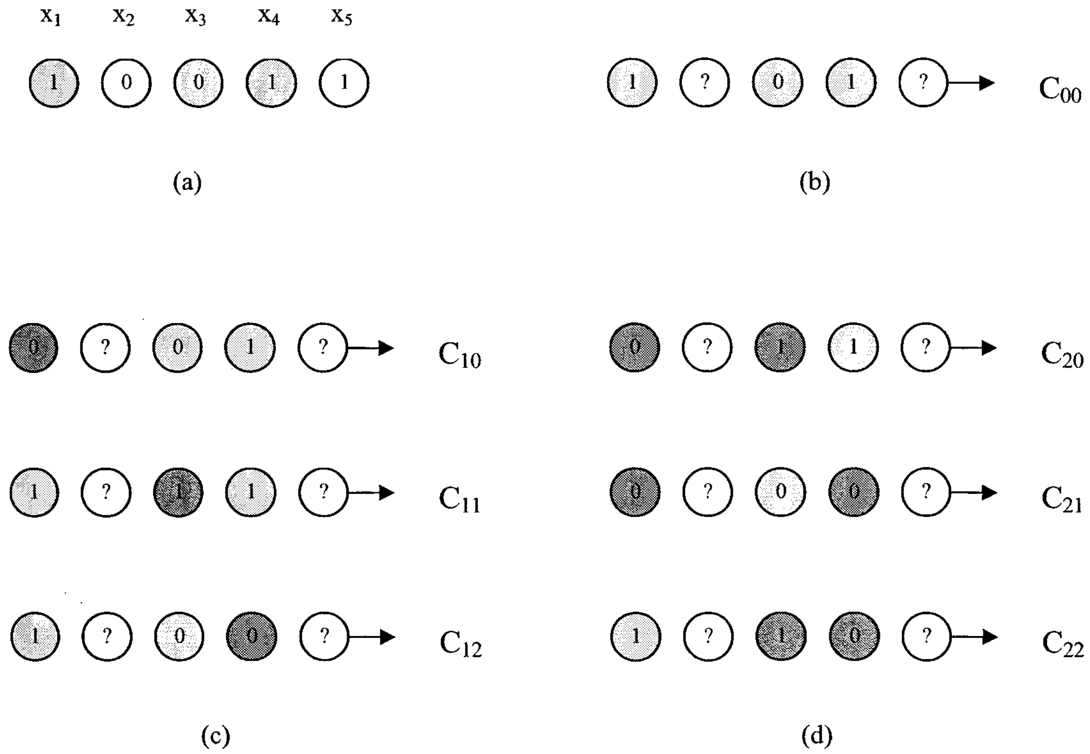


Figure 2.3: Generating a list of candidate codewords for a (5,3) code using the order-*2* reprocessing algorithm (a) received sequence with MRIPs in x_1, x_3, and x_4 (b) encoding the initial k-tuple MRIP (c) encoding all test sequences with 1-error patterns (d) encoding all test sequences with 2-error patterns

Once the candidates list is generated, the rest of decoding process will be the same as the decoding-based algorithm described in the previous section, and the soft output can be calculated by following equation:

$$\Lambda(d_j) = \frac{1}{2\sigma^2}(|R - C^{-1(i)}|^2 - |R - C^{+1(i)}|^2) \tag{2.22}$$

and extrinsic information is found as

$$w_j = \sum_{\ell=1,\ell\neq j}^{n} r_\ell c_\ell^{+1(j)} p_\ell \quad (2.23)$$

with

$$p_\ell = \begin{cases} 0 & if \quad c_\ell^{+1(i)} = c_\ell^{-1(i)} \\ 1 & if \quad c_\ell^{+1(i)} \neq c_\ell^{-1(i)} \end{cases} \quad (2.24)$$

The extrinsic information is fed into next iteration by a scaling factor α, which is also chosen experimentally. However, in an encoding-based algorithm, the estimated reliability β is not always necessary, since a large number of codewords are examined and it is usually guaranteed that each position will have a corresponding competing codeword.

Although the concept of the decoding-based algorithm is very simple, converting it into hardware implementation will raise many problems. From the circuit design point of views, reordering sequence, and creating a new generator matrix require a large amount of computational complexity. Moreover, reviewing such a large number of codewords will increase not only the memory requirement but also the decoding latency.

2.3.2 Trellis-Based Algorithm

Using the trellis-based decoding technique for BTCs can achieve a performance close to a maximum likelihood decoder. The trellis representation can be obtained either by the BCJR algorithm or by the Massey algorithm.

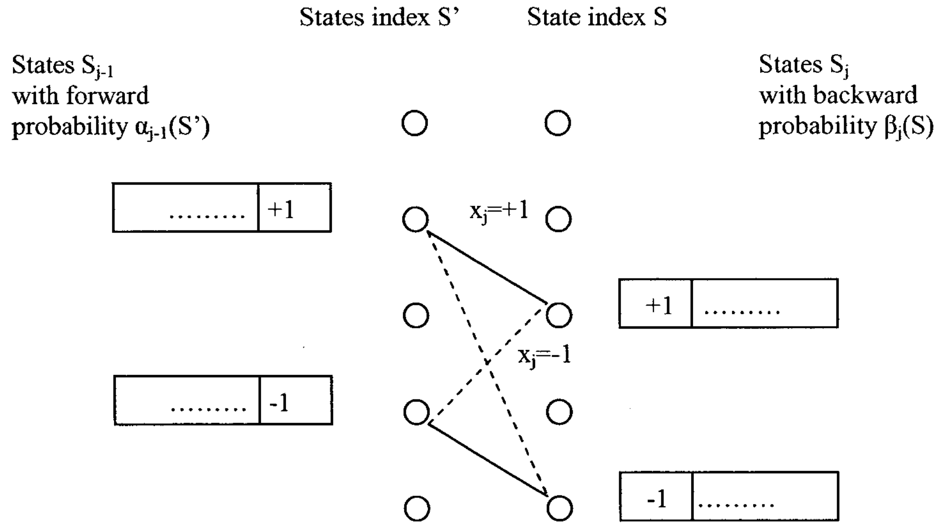


Figure 2.4: Trellis structure of a block code

Figure 2.4 shows the binary trellis of a block code. The soft output for *j*th symbol can be calculated by the following equation:

$$L(\hat{x}_j) = \ln \frac{P(x_j = +1 \mid R)}{P(x_j = +1 \mid R)} = \ln \frac{\sum_{(s',s),x_j=+1} p(s',s,R)}{\sum_{(s',s),x_j=-1} p(s',s,R)} \tag{2.25}$$

Assuming a memoryless transmission channel, the joint probability *p(s',s,R)* can be written as

$$\begin{aligned} p(s',s,R) &= p(s',R_{k<j}) \cdot p(s,R_j \mid s') \cdot p(R_{k>j} \mid s) \\ &= p(s',R_{k<j}) \cdot P(s \mid s') \cdot p(R_j \mid s,s') \cdot p(R_{k>j} \mid s) \\ &= \alpha_{j-1}(s') \cdot \beta_j(s) \cdot \gamma_j(s,s') \end{aligned} \tag{2.26}$$

where $R_{k<j}$ represents the sequence of received symbols r_k from the first bit up to bit $j-1$ and, similarly, $R_{k>j}$ represents the sequence of received symbols r_k from the bit j up to the last bit.

Therefore, in order to obtain the soft output, we need to calculate the forward metrics α_j, the reverse metrics β_j, and the branch metrics γ_j.

The forward metrics α_j and the reverse metrics β_j are found by following two equations:

$$\alpha_j(s) = \sum_{s'} \gamma_j(s,s') \cdot \alpha_{j-1}(s') \tag{2.27}$$

$$\beta_{j-1}(s') = \sum_{s} \gamma_j(s,s') \cdot \beta_j(s) \tag{2.28}$$

Since the starting and the ending of the trellis diagram always merges to zero. We have $\alpha_0(0)=1$ and $\beta_n(0)=1$.

The branch metrics γ_j is given by

$$\gamma_j(s,s') = P(s \mid s') \cdot p(r_j \mid s', s) = p(x_j; r_j) \tag{2.29}$$

Assuming that the information bits are statistically independent, we can find the transition probability as

$$p(x_j; r_j) = \begin{cases} p(r_j \mid x_j) \cdot P(x_j) & 1 \le j \le k \\ p(r_j \mid x_j) & k+1 \le j \le n \end{cases} \tag{2.30}$$

Using log-likelihood, the a prior probability $P(x_j)$ can be expressed as

$$P(x_j = \pm 1) = \frac{e^{\pm L(x_j)}}{1 + e^{\pm L(x_j)}} = \left(\frac{e^{-L(x_j)/2}}{1 + e^{-L(x_j)/2}}\right) \cdot e^{L(x_j) \cdot x_j / 2} = A_j \cdot e^{L(x_j) \cdot x_j / 2} \tag{2.31}$$

and the conditional probability $p(r_j \mid x_j)$ is found by

$$p(r_j \mid x_j) = B_j \cdot e^{L_c \cdot r_j \cdot x_j / 2} \tag{2.32}$$

Observe that A_j and B_j are equal for all transitions from time j-1 to time j and can be cancelled out in the ratio of (2.29). Thus, the branch metrics operation is simplified as

$$\gamma_j(s, s') = e^{x_j \cdot (L_c \cdot r_j + L(x_j))/2} = e^{L(x_j; r_j) \cdot x_j / 2} \tag{2.33}$$

where $L(x_j;r_j)$ is the log-likelihood ratio associated with $p(x_j;r_j)$, written as

$$L(x_j; r_j) = \begin{cases} L_c r_j + L(x_j) & 1 \le j \le k \\ L_c r_j & k+1 \le j \le n \end{cases} \tag{2.34}$$

Combining Equation (2.25) ~ (2.34), the soft output using log-MAP decoder can be expressed as

$$L(\hat{x}_j) = L_c r_j + L(x_j) + \frac{\sum_{(s',s),x_j=+1} \alpha_{j-1}(s') \beta_j(s)}{\sum_{(s',s),x_j=-1} \alpha_{j-1}(s') \beta_j(s)} \tag{2.35}$$

In turbo decoding, the last term in Equation (2.35) will be used as the extrinsic information for the next iteration.

Chapter 3

Distance-Based Decoding Of Block Turbo Codes

3.1 Introduction

In this chapter, we introduce a new algorithm, called *distance-based* algorithm, as it calculates the extrinsic information based on the Euclidean distance property. Although the proposed algorithm is originally considered as a modification of the augmented list algorithm based on Chase II algorithm [14], the concept can be easily extended to other turbo product codes. Compared to the traditional algorithm in [14], when using the proposed algorithm we observed performance improvements for BTCs having large minimum Hamming distance and negligible loss for BTCs having short Hamming distance.

The rest of the chapter is organized as follows. In Section 3.2, the drawbacks of the traditional algorithm are indicated and few suggestions that may improve the algorithm are given. Section 3.3 provides the detailed description of the new algorithm. Section 3.4

describes how the new algorithm is simulated in C++ language. Then the error performance of several BTCs using the new algorithm are investigated in Section 3.5. Finally, conclusion is given in Section 3.6.

3.2 Drawbacks of the Traditional Algorithm

The traditional algorithm [14] described in the previous chapter which uses Chase II algorithm is now widely used in many communication systems, for example, satellite communications and *digital video broadcasting* (DVB). Compared to the turbo convolutional codes using trellis-based algorithms, this algorithm achieves moderate error performance and relatively low complexity for hardware implementations.

As with most of the commonly-used turbo codes, the major drawback of the traditional algorithm for BTCs is its high computational complexity. From the hardware implementation point of view, the overall complexity mainly depends on the number of algebraic decoders that are used for decoding the test sequences. Usually, in order to maximize the data throughput, 2^p test sequences are decoded simultaneously, and consequently, 2^p algebraic decoders are required for each iteration in hardware design. Technically, if we can reduce the observed number of test patterns by half, in return, approximately 50% of complexity saving is achievable. However, the problem is that, for the traditional algorithm, decreasing the number of test patterns will cause serious degradation in the error performance. For example, for BTC$(64,51,6)^2$, using 8 test patterns instead of 16, will result in a 0.5dB performance degradation. Such a penalty is normally not affordable for real applications.

Another design issue is regarding the calculation of the soft outputs. Recall that in the traditional algorithm, the competing codeword ***B*** which is selected from the 2^p codeword candidates may be different depending on the bit position. Thus, to directly realize the algorithm, a process that searches for ***B*** and calculates corresponding soft output has to be repeated for every bit position. This leads to large computational complexity and memory requirement. One solution is provided in [20], which only keep the closest competing codeword in memory. However, the corresponding performance degradation is also severe, which makes the solution less attempting.

Discussion above is mainly focused on computational complexity of the algorithm. Next, we will mention few observations regarding the error performance. Since we know that the traditional algorithm is a sub-optimal method, certainly, it is possible to improve the error performance if we can further increase the accuracy of the feedback extrinsic information. Note that the major step which impacts the final performance is the derivation from Equation (2.6) to Equation (2.12), rewritten as (3.1) and (3.2), respectively.

$$\Lambda(d_j) = \ln\left(\frac{\sum_{C^i \in S_j^{+1}} p\{R \mid X = C^i\}}{\sum_{C^i \in S_j^{-1}} p\{R \mid X = C^i\}}\right) \tag{3.1}$$

$$\Lambda(d_j) = \frac{1}{2\sigma^2}(|R-B|^2 - |R-D|^2) \cdot d_j \tag{3.2}$$

This derivation is done by assuming that the communication channel has a high SNR and the influence of the codewords other than ***B*** and ***D*** can be ignored in the calculation.

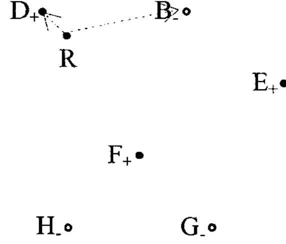


Figure 3.1: Examples of the augmented list decoding with high SNR

This can be seen from a simple example as shown in Figure 3.1. Let us consider a code vector space consisting of six valid codewords (**B, D, E, F, H, G**), where ***R*** is the received sequence, ***D*** is the decision codeword and ***B*** is the competing codeword. To calculate the soft output at *j*th position, we may divide all codewords into two groups, where ***D, E, F*** are valid codewords with corresponding bit taking value of +1, and ***B, H, G*** are valid codeword with corresponding bit taking value of -1, respectively. To achieve the best error performance, soft outputs should be calculated based on Equation (3.1), which can also be expressed as

$$\Lambda(d_j) = \ln\left(\frac{p\{R \mid X = D\} + p\{R \mid X = E\} + p\{R \mid X = F\}}{p\{R \mid X = B\} + p\{R \mid X = H\} + p\{R \mid X = G\}}\right) \tag{3.3}$$

Alternatively, the suboptimal decoding using (3.2) which computes the probability density function for only ***D*** and ***B*** is considered to be sufficient, since the channel noise is very small and other codewords have very litter influence on the calculation of Equation (3.3).

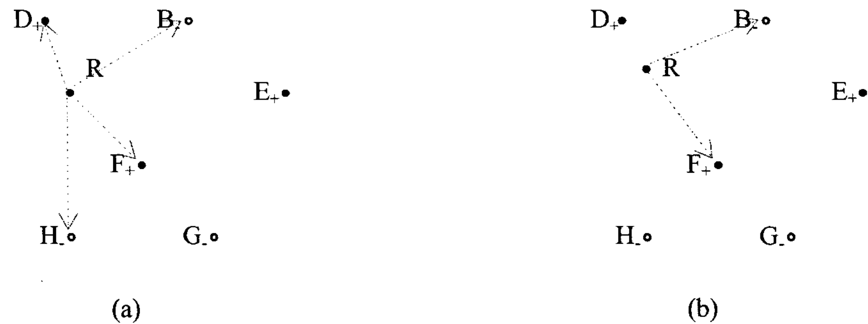


Figure 3.2: Examples of the augmented list decoding with low SNR (a) received sequence is corrupted with large noise (b) the closest codeword is not in the candidate list.

Unfortunately, this approximation does not always retain acceptable accuracy. Evidence is provided from the next two observations.

Case 1: Consider a situation as shown in Figure 3.2(a) where $\boldsymbol{R}$ is corrupted with large noise. Clearly, since the influence of other codewords is comparable to that of the decision codeword $\boldsymbol{D}$ and the competing codeword $\boldsymbol{B}$, the traditional algorithm considering only $\boldsymbol{D}$ and $\boldsymbol{B}$ is not sufficient in this case. At least, few more codewords, such as $\boldsymbol{H}$ and $\boldsymbol{F}$, should be taken into account in the computation of the soft outputs in Equation (3.3). This observation indicates one fact, that is, in list-based algorithms, soft outputs based on a decision codeword $\boldsymbol{D}$ with large Euclidean distance are usually not reliable.

Case 2: As Chase II algorithm reviews only 2^p test sequences, it is possible that the chosen decision codeword may not be the closest codeword to $\boldsymbol{R}$, especially when the number of test sequences is small. Figure 3.2b shows an example, where the closest

codeword $\boldsymbol{D}$ is not in the candidates list, and consequently, the decoder mistakenly chooses the codeword $\boldsymbol{F}$ as the decision. Obviously, soft outputs generated based on $\boldsymbol{F}$ will not give satisfying results. Also, it is predictable that the false decision codeword tends to be having relatively large Euclidean distance from $\boldsymbol{R}$.

From the case study, we have shown that soft output calculation using the augment list algorithm may give out false information under certain situations. More importantly, from the two observations, we find that soft output associated with a decision codeword having large Euclidean distance from $\boldsymbol{R}$ usually has very low reliability. This discovery is a major breakthrough in our study.

In our new approach, all the problems mentioned above are perfectly solved, and both complexity and performance are improved compared to the traditional algorithm.

3.3 The Distance-based Algorithm

The new algorithm is developed and verified through software simulations. In order to compare the error performance, we shall decode the same codes as in the traditional algorithm [14]. Thus, extended BCH codes are used as the component codes to construct BTCs. An extended BCH code can be viewed as an inner block code with an overall parity bit. It is worth mentioning that during the decoding process (section 2.3.1), only inner block code is used in the search of p LRBs. The test sequences for inner code are decoded by an algebraic decoder to produce a list of possible codewords. An overall parity bit for each decoded inner code is then calculated to produce extended codewords.

The traditional algorithm described in Section 2.3.1 can be summarized as two steps, where first step generates codeword list and second step calculates soft outputs. In our approach, Chase II algorithm in step 1 is kept the same. The end of this step gives a decision code word ***D*** that has a minimum squared Euclidean distance to the received sequence ***R***. However, step 2 is replaced with a *"distance-based algorithm"*, which computes extrinsic information based on the distance property of ***D***. In the new algorithm, first the confidence value of the decoded codeword is evaluated and then the soft outputs are computed.

3.3.1 Confidence Value

Define *confidence value* Φ as the probability that the decoder makes correct decision given received sequence ***R***. Assuming that $P(X = C^i) = \frac{1}{2^k}$, $i=1,2,..., 2^k$, then :

$$\phi = P(D = X \mid R) = \frac{p\{R \mid X = D\}}{\sum_{i=1}^{2^k} p\{R \mid X = C^i\}} \tag{3.4}$$

where $\{C^i\}$ is the set of all valid codewords.

Computing the Equation (3.4) may be far too complex for a practical implementation. Thus, approximation has to be used. Obviously, a straightforward way is using squared Euclidean distance, denoted as *Dist*, to estimate the confidence value. By doing this, we may treat confidence value Φ as a function of the squared Euclide/an distance, written as

$$\phi = f(Dist) \tag{3.5}$$

However, this method only works at the first iteration where there is no extrinsic information. Side effects of the extrinsic information have to be removed from the calculation of *Dist* in later iterations. A solution is made based on the study of Euclidean distance property. Consider the *j*th position in ***D*** where the individual contribution to *Dist* is $(r_j + w_j - d_j)^2$. In this case, the extrinsic information w_j plays a positive role in the estimation process (decreasing the distance) when $(r_j - d_j) * w_j < 0$. Likewise, the extrinsic information w_j plays a negative role (increasing the distance) when $(r_j - d_j) * w_j > 0$ and this part of effect should not be taken into account in the estimation process. We define *Destructive Euclidean Distance,* denoted $Dist_{des}$, as the sum of distance where the noise vector has a different polarity than the decision vector, written as

$$Dist_{des} = \sum_{j \in DES} (r_j - d_j)^2 \quad where\ DES = \{j \mid (r_j - d_j) \cdot d_j < 0\} \tag{3.6}$$

Note that in (3.6), we have replaced w_j with d_j since most of the time w_j and d_j have the same sign.

Similarly, we define *Non-destructive Euclidean Distance,* denoted $Dist_{non\text{-}des}$, as the sum of distance where the noise vector has the same polarity as the decision vector, written as

$$Dist_{non-des} = \sum_{j \in NONDES} (r_j - d_j)^2 \quad where\ NONDES = \{j \mid (r_j - d_j) \cdot d_j > 0\} \tag{3.7}$$

For better understanding of the concept of destructive Euclidean distance, here we give a simple example. Assume that a data sequence (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) is transmitted over AWGN channel and is received as (r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8). Using an algebraic

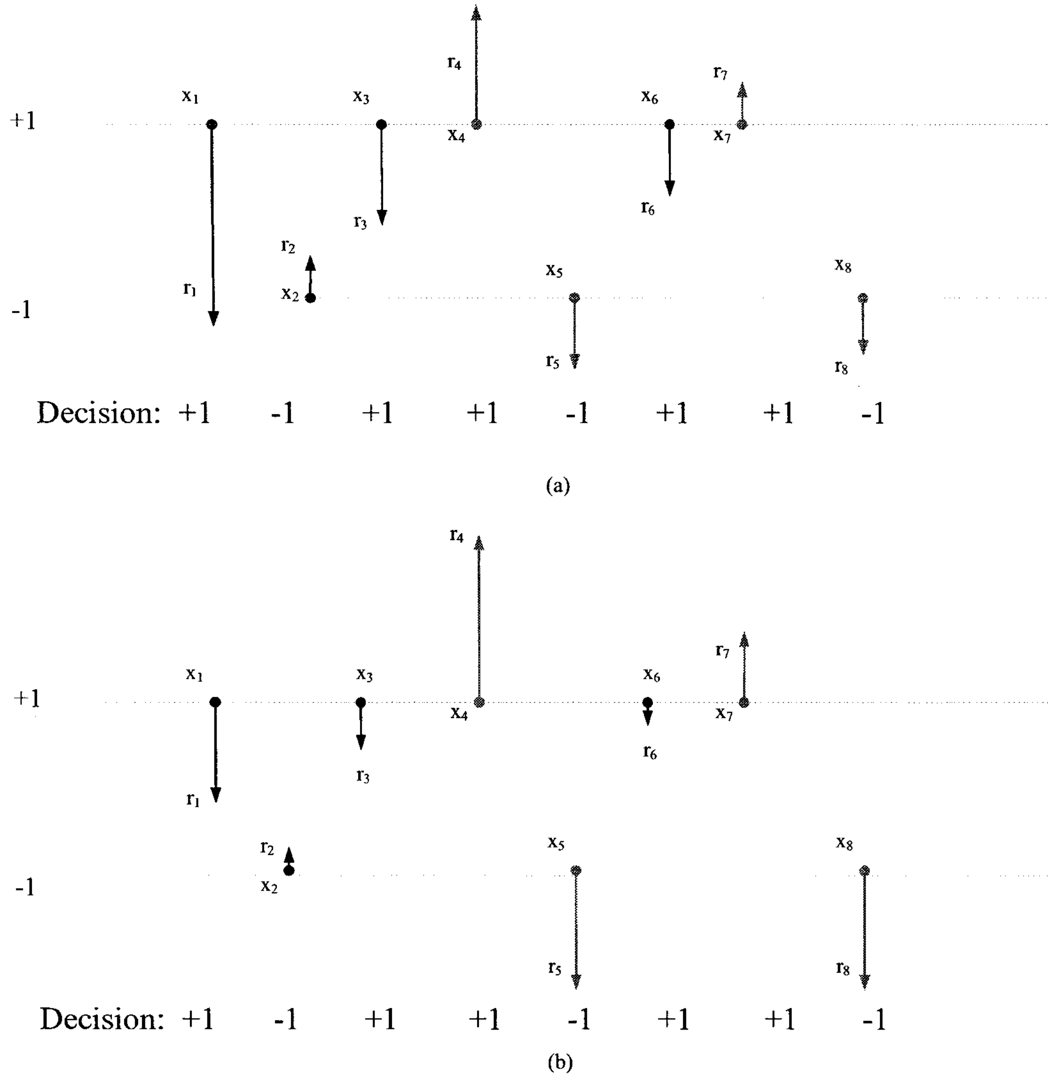


Figure 3.3: Noise distributions for a simple binary block code

(a) iteration 1 (b) iteration 2

decoder we get a decision codeword (d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8). Figure 3.3(a) depicts the situation for the first iteration where there is no extrinsic information presented. According to the definition of destructive Euclidean distance, we get

$$Dist_{des} = n_1^2 + n_2^2 + n_3^2 + n_6^2 \tag{3.8}$$

$$Dist_{non-des} = n_4^2 + n_5^2 + n_7^2 + n_8^2 \tag{3.9}$$

where n_j is an independent noise vector as $n_j = r_j - x_j$. Since noise is randomly distributed over the channel, it is likely that, at the beginning of the decoding, the destructive Euclidean distance $Dist_{des}$ is approximately equal to the non-destructive Euclidean distance $Dist_{non\text{-}des}$. Generally speaking, estimation of confidence value is easy in this stage. Unfortunately, problems usually appear in the next iteration where the input data sequence is added with extrinsic information. The change of the noise distribution in the next iteration is illustrated in Figure 3.3(b). Certainly, we have seen that the destructive Euclidean distance decreases, reflecting the fact that the associated bit is much reliable than the previous iteration. On the contrary, the non-destructive Euclidean distance increases a lot and causes serious problem for the estimation of the confidence value. Therefore, our solution which uses only the destructive Euclidean distance for the estimation procedure is much more efficient than using the whole Euclidean distance.

In our study, we obtained the relationship between the confidence value Φ and the destructive Euclidean distance $Dist_{des}$ through software simulation. Taking $BTC(64,51,6)^2$ as an example, Figure 3.4 was generated by simulation for 10000 samples of decoding results. Since Φ also depends on the iteration step t, signal to noise ratio Eb/No, and the number of LRB p, different curves are plotted for comparison. As can be seen, all resulting curves are similar to each other. Therefore, for practical considerations, we may omit the influence of the variable t, Eb/No, and p, and treat the confidence value Φ as a function of destructive Euclidean distance, written as

$$\phi \approx f(Dist_{des}) \tag{3.10}$$

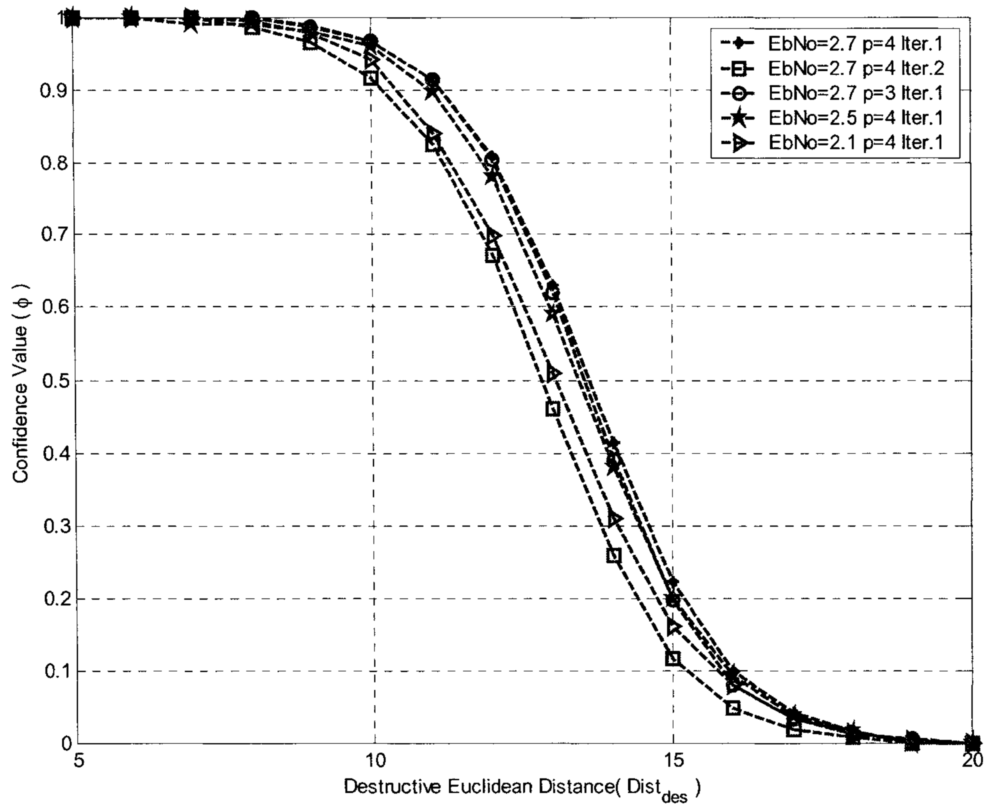


Figure 3.4: Confidence value versus destructive Euclidean distance for BTC $(64,51,6)^2$

The actual values of Φ have to be chosen for the individual code from software simulation as shown in Figure 3.4. The curve with the best performance is selected as shown in Table 3.1 for BTC$(64,51,6)^2$. It is worth mentioning that the values of Φ as a function of $Dist_{des}$ are pre-defined and are used to generate a lookup table of soft outputs as will be discussed later. Consequently, no computational complexity is added for practical implementations in this stage.

Table 3.1: Function map between $Dist_{des}$ and Φ

$Dist_{des}$	<9	9	10	11	12	13	14	>14
Φ	0.99	0.93	0.9	0.82	0.65	0.42	0.21	0

Also observe that confidence value Φ is only sensitive to destructive Euclidean distance in a range from 8 to 15. To simplify hardware design, we may approximate the function map using values in Table 3.1.

3.3.2 Soft-Output Calculation

Consider the information bit x_j that belongs to a codeword with certain confidence value Φ. The probability of x_j can be expressed as:

$$P(x_j = \pm 1 \mid R) = P(x_j = \pm 1, D = X \mid R) + P(x_j = \pm 1, D \neq X \mid R) \tag{3.11}$$

The first term represents the probability value when the decoder gives correct codeword. Applying Bayes' rule to this term will yield:

$$\begin{aligned} P(x_j = \pm 1, D = X \mid R) &= P(x_j = \pm 1 \mid D = X, R) \cdot P(D = X \mid R) \\ &= P(x_j = \pm 1 \mid D = X, R) \cdot \phi \end{aligned} \tag{3.12}$$

Since we know the decision bit d_j, then:

$$P(x_j = \pm 1, D = X \mid R) = \begin{cases} \phi & if \quad d_j = x_j \\ 0 & if \quad d_j \neq x_j \end{cases} \tag{3.13}$$

The second term in Equation (3.11) represents the probability value when the decoder decides in favor of a wrong codeword. In this case, we assume that the reliability of decision bit d_j is the same as an information bit corrupted with Gaussian noise.

$$P(x_j = \pm 1 \mid D \neq X) = \frac{\exp(\pm 2r_j / \sigma^2)}{1 + \exp(\pm 2r_j / \sigma^2)} \tag{3.14}$$

Again, we apply Bayes' rule to the second term of the Equation (3.11), and get

$$\begin{aligned} P(x_j = \pm 1, D \neq X \mid R) &= P(x_j = \pm 1 \mid D \neq X, R) \cdot P(D \neq X \mid R) \\ &= \frac{\exp(\pm 2r_j / \sigma^2)}{1 + \exp(\pm 2r_j / \sigma^2)} \cdot (1 - \phi) \end{aligned} \tag{3.15}$$

Combining Equations (3.11)-(3.15), the a posterior probability of x_j is found as:

$$P(x_j = +1 \mid R) = \begin{cases} \phi + (1-\phi) \dfrac{\exp(+2r_j / \sigma^2)}{1 + \exp(+2r_j / \sigma^2)} & \text{if} \quad d_j = +1 \\ (1-\phi) \dfrac{\exp(+2r_j / \sigma^2)}{1 + \exp(+2r_j / \sigma^2)} & \text{if} \quad d_j = -1 \end{cases} \tag{3.16}$$

and

$$P(x_j = -1 \mid R) = \begin{cases} (1-\phi) \dfrac{\exp(-2r_j / \sigma^2)}{1 + \exp(-2r_j / \sigma^2)} & \text{if} \quad d_j = +1 \\ \phi + (1-\phi) \dfrac{\exp(-2r_j / \sigma^2)}{1 + \exp(-2r_j / \sigma^2)} & \text{if} \quad d_j = -1 \end{cases} \tag{3.17}$$

Similar to the original algorithm described in previous section, we can obtain the extrinsic information w_j by the following equation:

$$w_j = \frac{\sigma^2}{2}\ln\left(\frac{P(x_j = +1 \mid R)}{P(x_j = -1 \mid R)}\right) - r_j \quad (3.18)$$

substituting $P(x_j=+1|R)$ and $P(x_j=-1|R)$ as in Equation (3.16) and (3.17), we get,

$$w_j = \frac{\sigma^2}{2}\ln\left(\frac{\phi + \exp(2r_j d_j / \sigma^2)}{1-\phi}\right) - r_j \quad if \quad d_j = +1 \quad (3.19)$$

and

$$w_j = \frac{\sigma^2}{2}\ln\left(\frac{1-\phi}{\phi + \exp(2r_j d_j / \sigma^2)}\right) - r_j \quad if \quad d_j = -1 \quad (3.20)$$

Equation (3.19) and (3.20) can also be represented as:

$$w_j = d_j\left(\frac{\sigma^2}{2}\ln\left(\frac{\phi + \exp(2r_j d_j / \sigma^2)}{1-\phi}\right) - r_j d_j\right) \quad (3.21)$$

Observing Equations (3.10) and (3.21), we find that the extrinsic information w_j can be viewed as a function of r_j, d_j, and $Dist_{des}$($Dist_{des}$ is used to determine the confidence value Φ as described in previous section), written as $w_j=g(r_j, d_j, Dist_{des})$. Therefore, for practical implementations, soft outputs can be pre-calculated and stored in a lookup-table indexed by r_j, d_j, and $Dist_{des}$. For example, we can quantize r_j into 16 levels (4 bits) and $Dist_{des}$ into 8 levels (3 bits). If the data width of the soft output w_j is 4 bits, then the lookup table can be realized as a 256x4 ROM(Read-Only Memory). Comparing to other list-based algorithms which require large complexity to search and calculate outputs for each bit position, the advantage of using the distance-based algorithm is obvious.

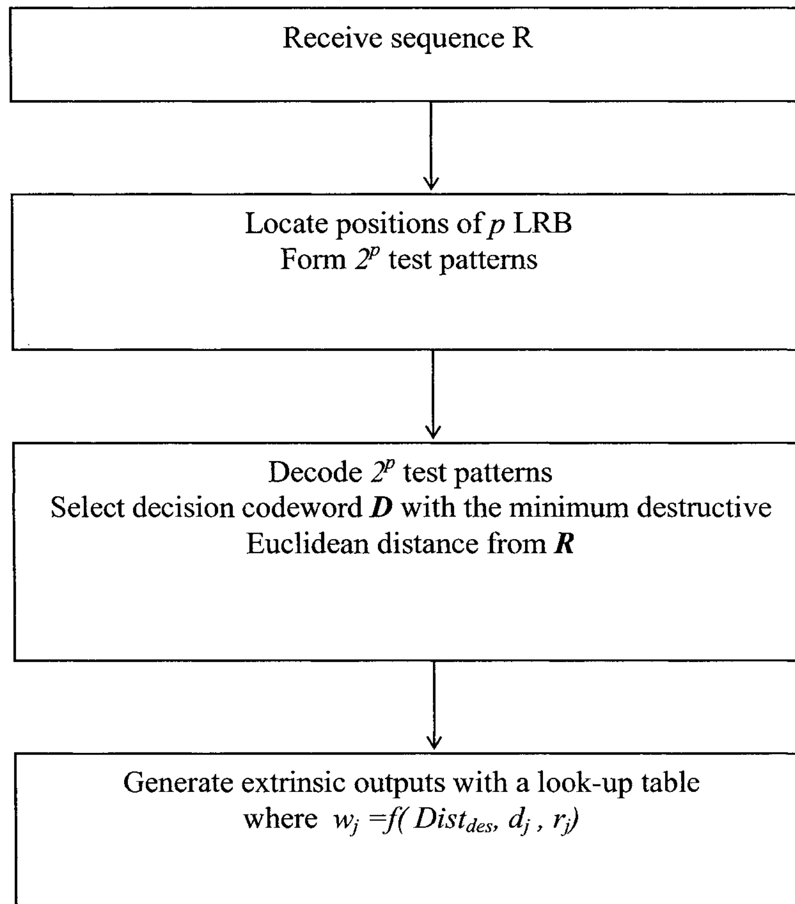


Figure 3.5: Block diagram of the decoding procedure

Another important difference from the traditional algorithm in [14] is that the soft outputs generated by the distance-based algorithm can be directly fed into the next decoding stage without scaling by a weighting factor α. Therefore, computational complexity is also reduced. A flow diagram of the proposed algorithm is shown in Figure 3.5.

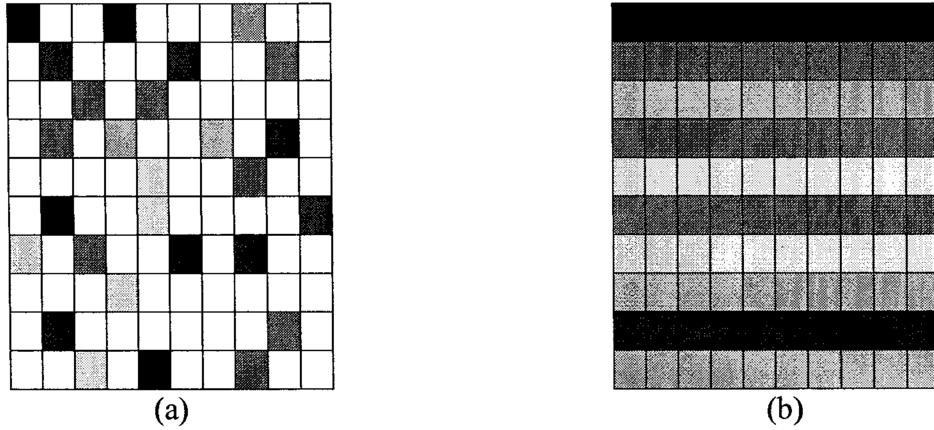


Figure 3.6: Extrinsic information structure (a) traditional algorithm (b) distance-based algorithm

3.3.3 Algorithm Comparison

Both the traditional algorithm and our distance-based algorithm use the same set of candidate codewords for the calculation of soft outputs. However, due to the different techniques that are used for computing the reliability value of the decision bit, the resulting error correcting ability of the two algorithms are also different.

In Figure 3.6, we show a visualization of the soft output structure for the two algorithms. As can be seen in Figure 3.6 (a), the traditional decoding algorithm outputs a star-type information structure since only few positions can have explicit values, and other positions have to use predetermined average reliability value β. Such design may raise several problems. As mentioned in section 3.2, the most likely code word may not exist in the 2^p test patterns, especially, when p is relatively small. Since the selection of the decision $\boldsymbol{D}$ is essential in the calculation of the extrinsic information, results based on a

false codeword $\boldsymbol{D}$ will give out wrong information. This leads to the poor performance when we try to use the traditional algorithm with small number of test patterns.

Now, let us see how this problem is solved in our new approach. As shown in Figure 3.6 (b), the distance-based algorithm calculates extrinsic information from the same codeword by using an estimated average value. Note in Equation (3.20) the extrinsic information decreases as the destructive Euclidean distance increases. For a decision codeword $\boldsymbol{D}$ having large destructive Euclidean distance, the extrinsic information based on Equation (3.20) will tend to be zero, meaning no additional information can be obtained from the iteration. Consequently, low reliability outputs are discarded from the decoding procedure. This results in high performance.

On the other hand, it is worth mentioning that without considering the computational complexity, the traditional algorithm will always outperforms our distance-based approach if given enough test patterns. Theoretically, the traditional algorithm can achieve optimal decoding by setting $p=k$. However, the proposed algorithm is not capable to converge to that point since we eventually lost certain information due to the estimation of confidence value.

3.4 Monte Carlo Simulation using C++

Software languages such as C/C++, MATLAB are efficient and cost-effective tools for algorithm simulation and system modeling. In a turbo decoding system, where complexity is relatively high, simulation speed is usually a major concern. In our study,

we model the system using C++ to take its speed advantage. The source code is available as reference in appendix.

Basically, the program in appendix is capable to simulate the new algorithm for any BTCs using extended BCH component codes. All parameters are defined at the beginning of the file. Therefore, by modifying these parameters, users can simulate the performance for various BTCs using the same program. The settings for BTC$(64,51,6)^2$ is shown in Figure 3.7.

```
const   int     m= 6;
const   int     q= 64;
const   int     n= 64;
const   int     k= 51;
const   int     ITERATIONS = 5;
const   int     SAMPLES = 100000;
const   double  EBNO = 2.5;
const   int     T_CORRECTABLE = 2;
const   int     T_TESTING_BITS = 4;
const   int     LISTLENGTH = 16;
const   int     INPUTLENGTH = 2601;
const   int     OUTPUTLENGTH = 4096;

const   int     GF_generator[]={1,1,0,0,0,0,1};
const   int     code_generator[]={1,0,0,1,1,1,0,0,1,0,1,0,1};
```

Figure 3.7: Parameter setting for a BTC$(64,51,6)^2$

The design of the program is quite straight forward. Figure 3.8 shows the major blocks which work in a similar way to the a real system.

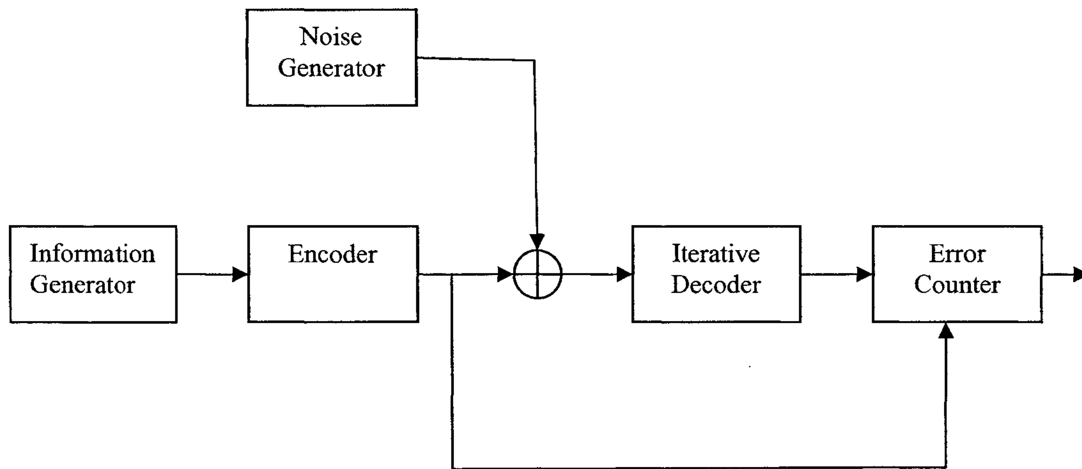


Figure 3.8 Block diagram of the software simulation

As shown in Figure 3.8, the simulation starts with the Information Generator, which is responsible to generate information symbols. In this design, we assume the information symbols are random binary bits with $k*k$ bit length for each block. In C++ program, we can use the built-in random function for this procedure.

The information sequence is then fed into the BTC Encoder. The core of the encoder is an extended BCH encoder, which adds n-k parity check bits to each row (or column). The result of the unit is a block of binary bits with n*n bit length, which is treated as a transmitted data sequence.

The next step is to pass the coded bits through AWGN channel. This is simulated by the Noise Generator, which adds noise to the transmitted data sequence. Again, this will require the use of the built-in random function. In order to assure the resulting random values having a Gaussian distribution, here, we adopt a well-known method, namely the

Box-Muller algorithm, for generation of white Gaussian noise. The mathematical background of the Box-Muller algorithm can be found in [31]. The C++ code is shown in Figure 3.8.

```
// Simulation of a AWGN channel. Noise is generated using Box-Muller method

  void pass_awgn_channel(int *input, double *output, int length,double sigma)
  {
    double x1, x2, w, y1, y2;
    int half=(int)length/2;
    for(int i=0; i<half;i++)
    {
        do {
            x1 = 2.0 * ranf() - 1.0;
            x2 = 2.0 * ranf() - 1.0;
            w = x1 * x1 + x2 * x2;
            } while ( w >= 1.0 );

        w = sqrt( (-2.0 * log( w ) ) / w );
        y1 = sigma * x1 * w;
        y2 = sigma * x2 * w;
        output[i]=y1+input[i]*2.0-1.0;
        output[i+half]=y2+input[i+half]*2.0-1.0;
    }
    if ((double)half!=((double)length/2))
    {
        do {
            x1 = 2.0 * ranf() - 1.0;
            x2 = 2.0 * ranf() - 1.0;
            w = x1 * x1 + x2 * x2;
            } while ( w >= 1.0 );
        w = sqrt( (-2.0 * log( w ) ) / w );
        y1 = sigma * x1 * w;
        output[length-1]=y1+input[length-1]*2.0-1.0;
    }
}
```

Figure 3.9: Simulation of an AWGN channel

On the receiver side, the major functional unit is a BTC Decoder, which is also the most sophisticated and important unit in this design. The decoder is made up of several individual functional blocks, which perform the tasks of locating p LRBs, generating a

list of test sequences, decoding all test sequences, selecting a decision codeword, and outputing extrinsic information. These functions are mostly very easy to implement in software languages like C++. Here, we would like to say a few words on the decoding algorithm that is used for the inner component code. In fact, there are many decoding algorithms available for the inner BCH codes, while some of them, like the trans-table algorithm [22] or chien search algorithm, have been optimized for particular codes. Of course, using a fast algorithm may improve the simulation speed, but this will lose certain flexibility in the program. Considering these facts, we decided to use a more general decoding algorithm, namely the Berlekamp algorithm [2], for the inner BCH decoding. The C++ code is shown in Figure 3.10.

```
// Simulation of a AWGN channel. Noise is generated using Box-Muller method
  bool BCH::do_decode(int *input, int *output,int listpos)
  {
    int paritybit=0;
    for (int i=0; i<n;i++)
    { output[i]=input[i];
    }
    get_syndrome(input);
    bool noerror=true;
    for(i=0;i<T_CORRECTABLE*2;i++)
    { if (syndrom[i]!=0) noerror=false;
    }
    if (noerror)
    { for ( i=0; i<n-1;i++)
      { paritybit+=input[i];
      }
      if (paritybit==(int)(paritybit/2)*2) output[n-1]=0;
      else output[n-1]=1;
    return true;
    }
    for ( i=0;i<2*T_CORRECTABLE+2;i++)
    { u[i]=i-1;
      for (int j=0;j<6;j++)
      { Q[i][j]=0;
      }
    }
    Q[0][0]=1;   Q[1][0]=1;   du[0]=1;
    du[1]=syndrom[0];  lu[0]=0; lu[1]=0;
    for (int j1=2;j1<(T_CORRECTABLE*2+2);j1++)
    { if (du[j1-1]==0)
      { lu[j1]=lu[j1-1];
        for(int i=0;i<lu[j1]+1;i++)
        { Q[j1][i]=Q[j1-1][i];
        }
      }
      else
      { int tempp=-1;     int  tempd=0;      int dp;
        for (int j2=0;j2<(j1-1);j2++)
        { if (du[j2]!=0)
          { dp=u[j2]-lu[j2];
            if (tempp==-1)
            {  tempp=j2;
               tempd=dp;
            }
            else  if (dp>tempd)
                  {
                    tempp=j2;
                    tempd=dp;
                  }
          }
        }
```

Figure 3.10: BCH decoding using Berlekamp algorithm. (see next page)

```
        int tempq1;
        tempq1=GF_multiply_table[du[j1-1]][GF_inverse(du[tempp])];
        int  degree=j1-1-tempp;
        int tempq2[6],tempq3[6];
        for(int i=0;i<degree+1;i++)
        { tempq2[i]=0;
        }
        tempq2[degree]=tempq1;
        int lengthofqp,lengthofq3,lengthoftq,lengthofq;
        lengthofqp=lu[tempp]+1;
        lengthofq3=lengthofqp+degree;
        GF_poly_convolution(tempq2,degree+1,Q[tempp],lengthofqp,tempq3);
        lengthoftq=lu[j1-1]+1;
        if (lengthoftq>=lengthofq3) lengthofq=lengthoftq;
        else lengthofq=lengthofq3;
        GF_poly_add(Q[j1-1],lengthoftq,tempq3,lengthofq3,Q[j1]);
        lu[j1]=lengthofq-1;
      }
      if (j1!=2*T_CORRECTABLE+1)
      {
        int tempdu=syndrom[j1-1];
        for(int j3=1;j3<lu[j1]+1;j3++)
        {
          int temp1;
          temp1=GF_multiply_table[Q[j1][j3]][syndrom[j1-j3-1]];
          tempdu=GF_add_table[tempdu][temp1];
        }
        du[j1]=tempdu;
      }
  }
  int count=0;
  int position;
  if (lu[2*T_CORRECTABLE+1]<=2)
  { for (j1=1;j1<q;j1++)
    {
      if (get_GF_poly_value(Q[1+2*T_CORRECTABLE],lu[2*T_CORRECTABLE]+1,j1)==0)
      { position=GF_inverse(j1)-1;
        output[n-1-1-position]=1-output[n-1-1-position];
        count=count+1;
      }
    }
  }
  if (count==0) return false;
  for ( i=0; i<n-1;i++)
  { paritybit+=output[i];
  }
  if (paritybit==(int)(paritybit/2)*2) output[n-1]=0;
  else output[n-1]=1;
  return true;
}
```

Figure 3.10: BCH decoding using Berlekamp algorithm. (continued)

The last part of the design is an Error Counter which compares the decoding result with the transmitted symbols. The number of error bits is counted and saved in the memory. At the end of the simulation, error performance is calculated and displayed in terms of BER.

Following table describes the major functions of each procedure in the program.

Table 3.2: Function descriptions

Functions	Description
Global functions	
main( )	This function is the main function of the program
generate_message( )	This function generates k random binary bits used for an information vector.
count_errors( )	This function counts the number of errors in the decoded sequence.
pass_awgn_channel( )	This function simulates an AWGN channel by adding noise to the transmitted data sequence.
generate_extrinsic_lookup_table()	This function creates the lookup table for extrinsic information.
generate_ Galois_field()	This function generates a Galois field.
GF_add( )	This function performs addition of two elements in a Galois field
get_GF_poly_value( )	This function calculates the value of a polynomial in a Galois field
GF_inverse( )	This function calculates the inverse value of a element in a Galois field

GF_poly_convolution( )	This function performs multiplication of two polynomials in a Galois field.
GF_poly_add( )	This function performs addition of two polynomials in a Galois field.
Functions in BCH class (used for component encoding and decoding)	
get_syndrome( )	This function computes syndrome of a binary sequence.
do_encode( )	This function encode an information sequence by using BCH codes
do_decode( )	This function performs hard decoding of a sequence using BCH codes
do_soft_decode( )	This function performs soft decoding of a sequence using BCH codes
locate_LRB_bits( )	This function locates p positions of LRBs.
generate_test_sequence( )	This function generates 2^p test sequences
decode_list( )	This function decodes all test sequences and forms a candidate codewords list.
get_distance( )	This function calculates the destructive distance of a codeword.
locate_decision_code( )	This function selects the decision codeword.
Functions in BCHTurboEncoder class	
do_encode( )	This function encode information vector using BTC.
Functions in BCHTurboDecoder class	
initial_extrinsic( )	This function initials all extrinsic information matrixes in the memory.
do_decode( )	This function performs iterative decoding on a data sequence.

3.5 Simulation Results

In this section, simulation results for BTCs over Gaussian channel using BPSK signaling are investigated for both the traditional algorithm and the distance-based algorithm. In addition, the impact of selecting p on error performance will be discussed.

First consider the performance of BTCs encoded by two-error correcting component codes. Figure 3.11 depicts BER of $BTC(64,51,6)^2$ after 4 iterations for both algorithms with p=2,3,4. As can be seen, our proposed algorithm improves performance over the traditional algorithm by approximately 0.3 dB at a BER of 10^{-5} with 16 test patterns (p=4), approximately 0.5 dB at a BER of 10^{-5} with 8 test patterns (p=3), and approximately 0.7 dB at a BER of 10^{-5} with 4 test patterns (p=2). It's interesting to see that the distance-based algorithm has excellent performance when using a small number of test patterns. Observe that the new algorithm with 8 test patterns outperforms, by 0.1 dB, the traditional algorithm with 16 test patterns. This means that at least a 50% reduction in computational complexity is achievable. Moreover, to further reduce the complexity, we may even use 4 test patterns for the new algorithm with only a loss of 0.1 dB at a BER of 10^{-5} compared with the traditional algorithm.

Similar conclusion can also be reached for $BTC(32,21,6)^2$ as shown in Figure 3.12. For all observed p, the new algorithm achieves better performance than the traditional algorithm. Again, as we expected, the new algorithm demonstrates an excellent decoding ability by using a small number of test patterns. For example, instead of using 16 test

patterns in the traditional algorithm, we may use the new algorithm with only 4 test patterns to obtain a similar performance (around 0.2 dB loss).

Next, we will examine the performance of BTC with one-error correcting component codes. Simulation results for $BTC(64,57,4)^2$ and $BTC(32,26,4)^2$ are shown in Figure 3.13 and Figure 3.14. We have observed performance improvement by using the new algorithm when the number of test patterns are relatively small ($p=2, 3$). However, as p increases ($p>=4$), the traditional algorithm shows better error correcting ability and the proposed algorithm cannot improve performance as we expected. This can be explained by studying the minimum Hamming distance of the codes. Recall that during the decoding procedure, 2^p test patterns are reviewed and only few of them may give unique valid code words for computing soft outputs. (the others may be either invalid code words or repeating code words). Because the number of valid candidates decreases as the minimum Hamming distance increases, in the case of BTC encoded by two-error correcting codes, for the traditional algorithm to be close to the performance of a nearly optimal decoder a large number of test patterns is required. Therefore, we may use the distance-based algorithm to improve the performance. However, for BTC with one-error correcting codes, relatively small number of test patterns provides sufficient information for the traditional algorithm and the new algorithm does not have any advantage in this case.

Finally, since we know the new algorithm is a sub-optimal algorithm, it could be useful to compare these results with the iterative decoding simulation performance when the

optimal algorithm is used for the component codes. Normally, we consider that the OSD (ordered statistic decoding) algorithm can achieve a performance very close to an optimal decoder. Here, we will use the results from [25] as reference. Figure 3.15 and 3.16 shows the simulation results for BTC$(64,51,6)^2$ and BTC$(32,21,6)^2$ over AWGN channel. As we can see, a coding gain of 0.2 db at a BER of 10^{-5} is observed by adopting the OSD algorithm. However, such performance improvement is usually worthless when considering the large complexity that is required for the OSD algorithm. Therefore, the new algorithm is advantageous for practical applications due to the low complexity.

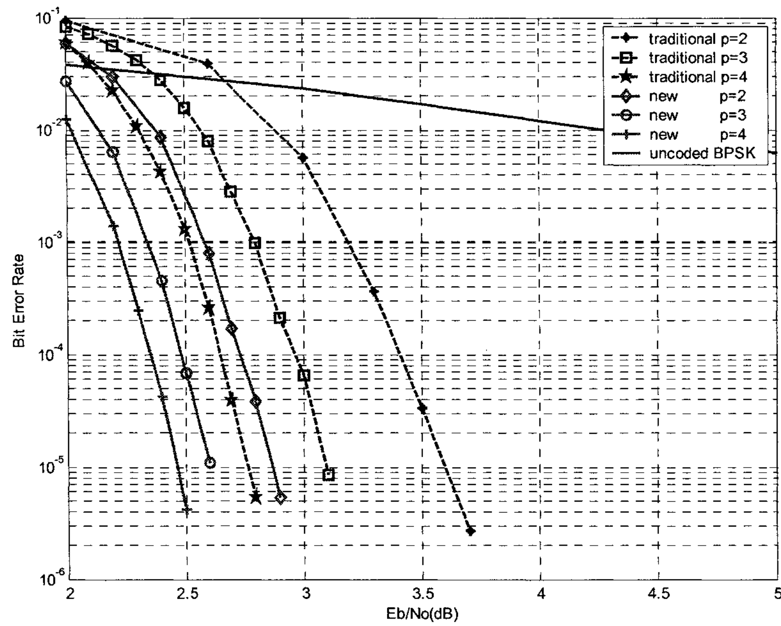


Figure 3.11: BER versus Eb/No of BTC$(64,51,6)^2$ on a Gaussian channel using BPSK signaling at iteration 4.

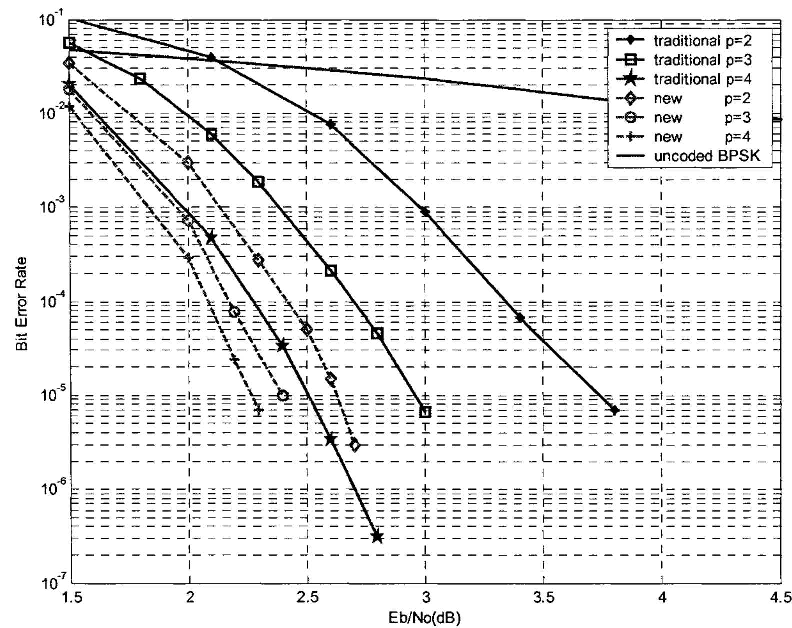


Figure 3.12: BER versus Eb/No of BTC$(32,21,6)^2$ on a Gaussian channel using BPSK signaling at iteration 4.

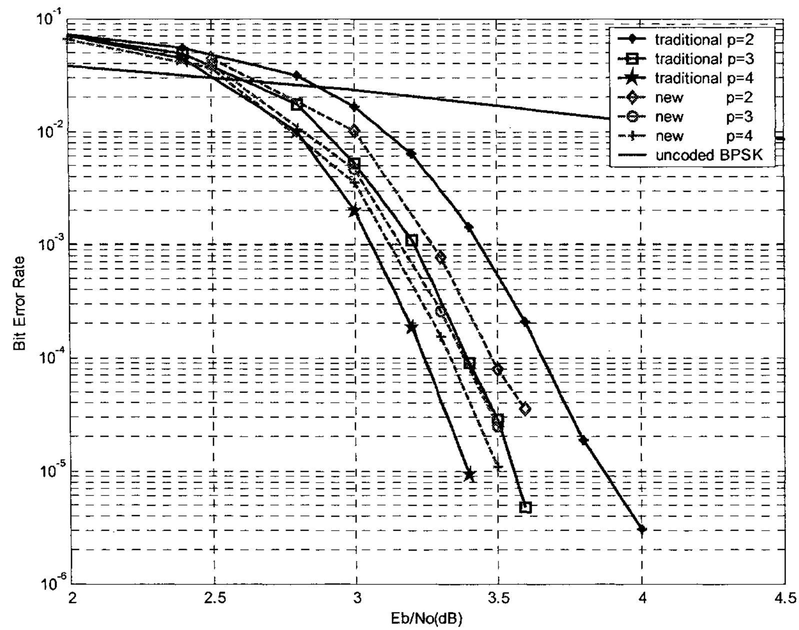


Figure 3.13: BER versus Eb/No of BTC(64,57,4)^2 on a Gaussian channel using BPSK signaling at iteration 4.

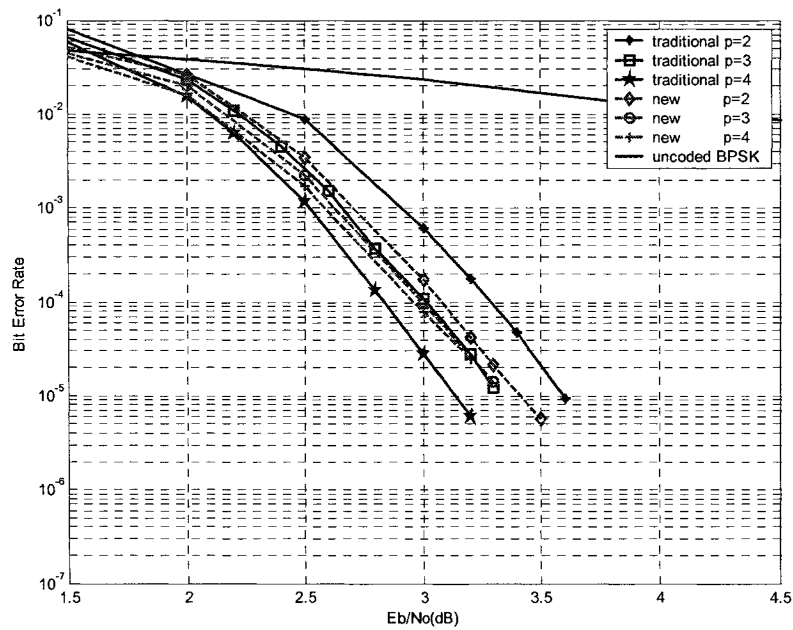


Figure 3.14: BER versus Eb/No of BTC(32,26,4)^2 on a Gaussian channel using BPSK signaling at iteration 4

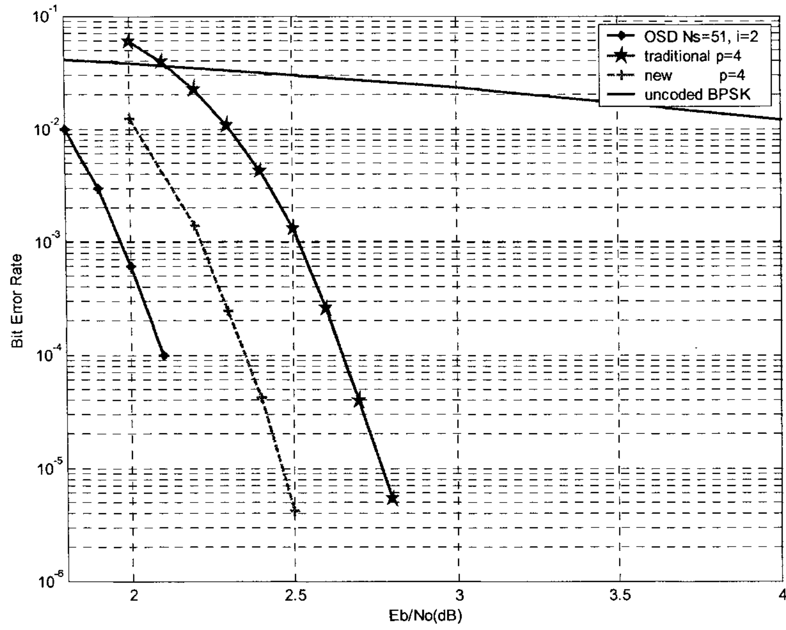


Figure 3.15: Comparison with other algorithms for BTC(64,51,6)^2 on a Gaussian channel using signaling at iteration 4

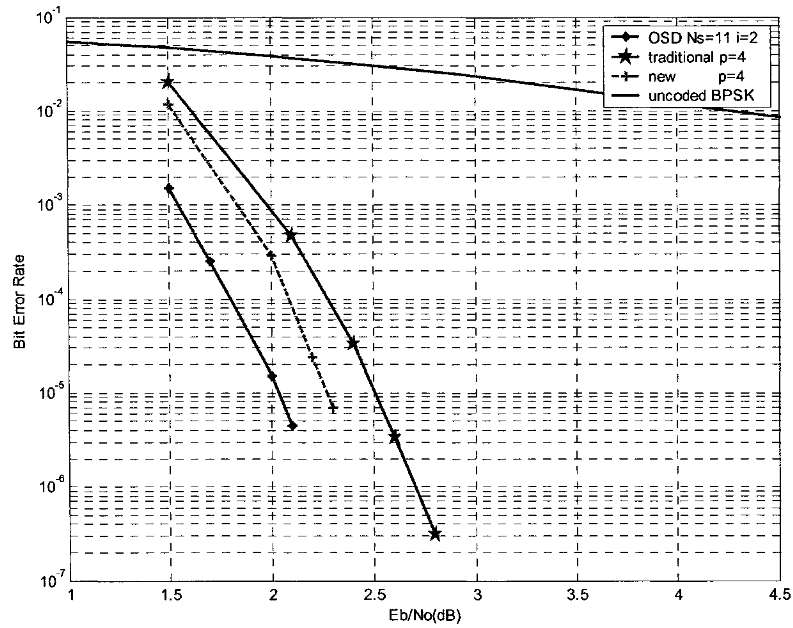


Figure 3.16: Comparison with other algorithms for BTC(32,21,6)^2 on a Gaussian channel using signaling at iteration 4

3.6 Discussions and Conclusions

In this chapter, we introduce a new distance-based algorithm, which performs SISO decoding based on the distance property of the decision codeword. Many exciting simulation results are presented, showing that the new algorithm is advantageous over the traditional algorithm.

In terms of error performance, the distance-based algorithm can improve the error correcting ability for most low code rate BTCs that use 2-error correcting component codes. Compared to commonly-used low code rate BTCs decoded by the traditional algorithm, up to 0.3 dB coding gain is achievable. Also, for high code rate BTCs, by using the new algorithm we observe certain improvement with small number of test patterns ($p<4$) and negligible performance loss with large number of test patterns ($p=4$).

The major advantage of the new algorithm is its very low computational complexity. Compared to the traditional algorithm, complexity reduction is obtained due to the following reasons.

- The major saving is accomplished through decreasing the number of test patterns in the new algorithm. Based on the Monte Carlo simulation results, we show a solution that uses only 4 test patterns and achieves a similar performance to the traditional algorithm with 16 test patterns. This results in a significant reduction in complexity.
- The calculation of soft outputs is much simpler in the new algorithm. In our approach, the competing codeword ***B*** is not necessary any more, and finding soft outputs by a

lookup table is more like a one shoot task. This feature also facilitates the hardware implementation, since no further optimization is required.

- The multiplication operation that is required in the traditional algorithm for scaling extrinsic information by a factor α is not necessary any more in the new algorithm.

Analyses given above only give a rough idea regarding the complexity issue. More precise comparison has to be carried out through hardware implementations. In next chapter, detailed description of FPGA implementation of the new algorithm is studied, and synthesis results will demonstrate the complexity saving in real applications.

Chapter 4

FPGA Design of the New Algorithm

4.1 Introduction

In previous Chapter, a new distance-based algorithm is presented to achieve excellent error performance for BTCs with large Hamming distance. In fact, the new algorithm is also very attractive for practical applications. As we mentioned in Section 3.3, since soft outputs are pre-calculated and saved in a lookup table, computational complexity is significantly reduced compared to the traditional algorithm. Reduction in computational complexity is shown through several FPGA *(Field Programmable Gate Array*) designs targeting on Xilinx Virtex II platform. Synthesis results of this implementation are directly compared to the commercially available product *TC3000* from *Turbo Concept.*

The rest of the chapter is organized as follows. In Section 4.2, FPGA technology and design methodology are reviewed with focus on Xilinx Virtex II products. Section 4.3 deals with some design issues such as quantization, memory arrangement, and hardware structures. Synthesis results are presented in Section 4.4 .

4.2 Review of FPGA Technologies

A field-programmable gate array is a large-scale integrated circuit that can be programmed after it is manufactured. As the name suggest, the term "field-programmable" means it is able to change the operation of the device "in the field," and "gate array" is the basic internal architecture that makes this reprogramming possible.

FPGA is a new technology which is ideal for prototyping systems, or verifying algorithms. From a hardware perspective, the FPGA platforms fill the gap between software programmable systems based on traditional microprocessors, and application-specific platforms based on custom hardware functions. From a software perspective, FPGA-based platforms enable the rapid creation of hardware-accelerated algorithms.

4.2.1 Basic Structure of FPGA

FPGAs come in a wide variety in terms of size, speed and technologies used for internal functions. They have certain key elements in common. A typical FPGA usually composed of a number of programmable logic blocks, where each of the blocks contains a few registers and configurable logic elements. These blocks are tied together with programmable interconnections.

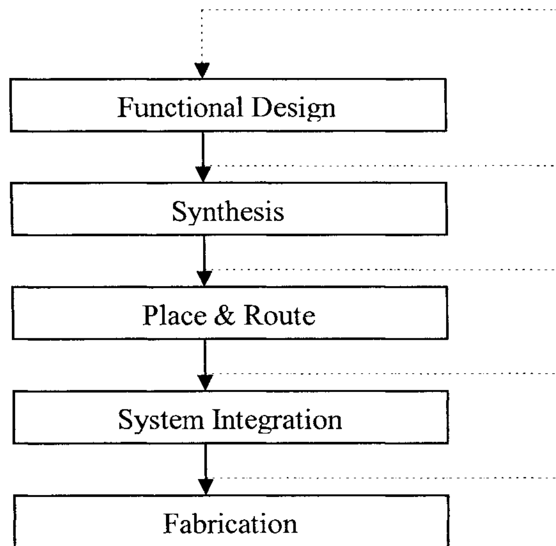


Figure 4.1: FPGA Design Flow

Depending on the technology used for programming, FPGAs in the market today can be divided into two categories: SRAM-based and antifuse-based FPGAs. In the first category, Xilinx and Altera leads in many other competitors, while for antifuse-based FPGAs, Actel , QuickLogic and Cypress are the leading manufactures.

4.2.2 FPGA Design Flow

As shown in Figure 4.1, a typical FPGA design may take following five steps: functional design, synthesis, place & route, system integration, and fabrication. Although in the figure, the design flow is drawn as a downward process, it is actually an iterative process where a designer always can return to any previous stage until required functionality is achieved. In each step, components are verified by simulation program such as ModelSim to ensure the correct functionality before moving to next step.

1. Functional design: Initial system concept is translated into an actual implementation either using schematic entry or a hardware description languages (HDL) such as VHDL or Verilog. For a complex project, the design usually starts with describing the system as several functional blocks in the behavior model, and then these individual blocks are constructed at RTL level.
2. Synthesis: The function design is mapped into a netlist--- a description of logic cells and their connection. The most common netlist formats are EDIF (Electronic Design Interchange Format), VHDL, Verilog, or XNF.
3. Place & route: The blocks of netlist are arranged on the chip and make the connections between cells and blocks. Once the locations of cells are determined, associated time delay can be estimated and used to verify the operating clock speed by a simulation tool.
4. System integration: A complete schematic of the entire system is constructed which includes all FPGAs and other devices such as EPROMs, RAMs, etc. This procedure visually prototypes the actual system and a successful simulation almost ensures correct operation of the physical prototype.
5. Fabrication: The program is downloaded into actual FPGA devices. This is the final process which is performed after entire system is simulated and the correct functionality is achieved.

4.2.3 Introduction of Xilinx Virtex II Family

Virtex II FPGAs from Xilinx are mainly developed for telecommunication, wireless, networking, video, and DSP applications. The Virtex-II architecture uses the leading-

edge 0.15 μm / 0.12 μm CMOS 8-layer metal process and is optimized for high speed applications with low power consumption. Some major features are listed as follows:

- Densities from 40K to 8M system gates
- 420 MHz internal clock speed (Advance Data)
- 3 Mbit of dual-port RAM in 18 Kbit block SelectRAM resources
- Up to 1.5 Mbit of distributed SelectRAM resources
- High-performance interface to external memory such as DRAM, SRAM, and CAM
- Up to 93,184 internal registers / latches with Clock Enable
- Up to 93,184 look-up tables (LUTs) or 16-bit shift registers
- High-Performance clock management circuitry with up to 12 DCM (Digital Clock Manager) modules

Following table shows all members in Vertex II family.

Table 4.1 Virtex-II Field-Programmable Gate Array Family Members

		CLB (1 CLB = 4 slices = Max 128 bits)				SelectRAM Blocks			
Device	System Gates	Array Row x Col.	Slices	Maximum Distributed RAM Kbits	Multiplier Blocks	18 Kbit Blocks	Max RAM (Kbits)	DCMs	Max I/O Pads(1)
XC2V40	40K	8 x 8	256	8	4	4	72	4	88
XC2V80	80K	16 x 8	512	16	8	8	144	4	120
XC2V250	250K	24 x 16	1,536	48	24	24	432	8	200
XC2V500	500K	32 x 24	3,072	96	32	32	576	8	264
XC2V1000	1M	40 x 32	5,120	160	40	40	720	8	432
XC2V1500	1.5M	48 x 40	7,680	240	48	48	864	8	528
XC2V2000	2M	56 x 48	10,752	336	56	56	1,008	8	624
XC2V3000	3M	64 x 56	14,336	448	96	96	1,728	12	720
XC2V4000	4M	80 x 72	23,040	720	120	120	2,160	12	912
XC2V6000	6M	96 x 88	33,792	1,056	144	144	2,592	12	1,104
XC2V8000	8M	112 x 104	46,592	1,456	168	168	3,024	12	1,108

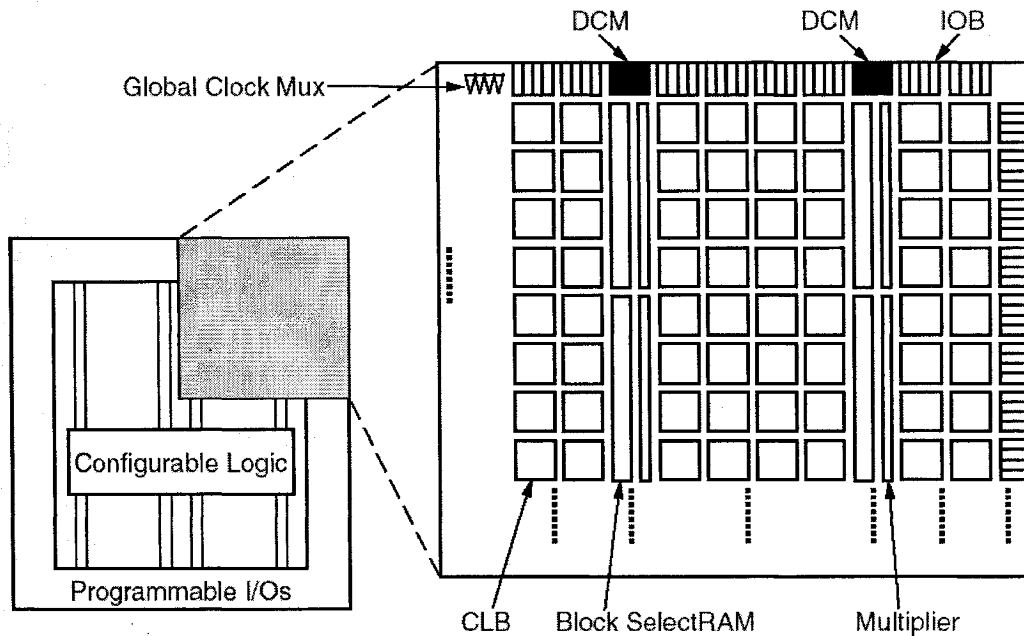


Figure 4.2 Internal structure of Virtex II

The internal structure of a Virtex II FPGA is mainly composed of four major elements as shown in Figure 4.2.

- *Configurable Logic Blocks* (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements.
- Block SelectRAM memory modules provide large 18 Kbit storage elements of dual-port RAM.
- Multiplier blocks are 18-bit x 18-bit dedicated multipliers.
- DCM (*Digital Clock Manager*) blocks provide self-calibrating, fully digital solutions for clock distribution delay compensation, and clock multiplication and division.

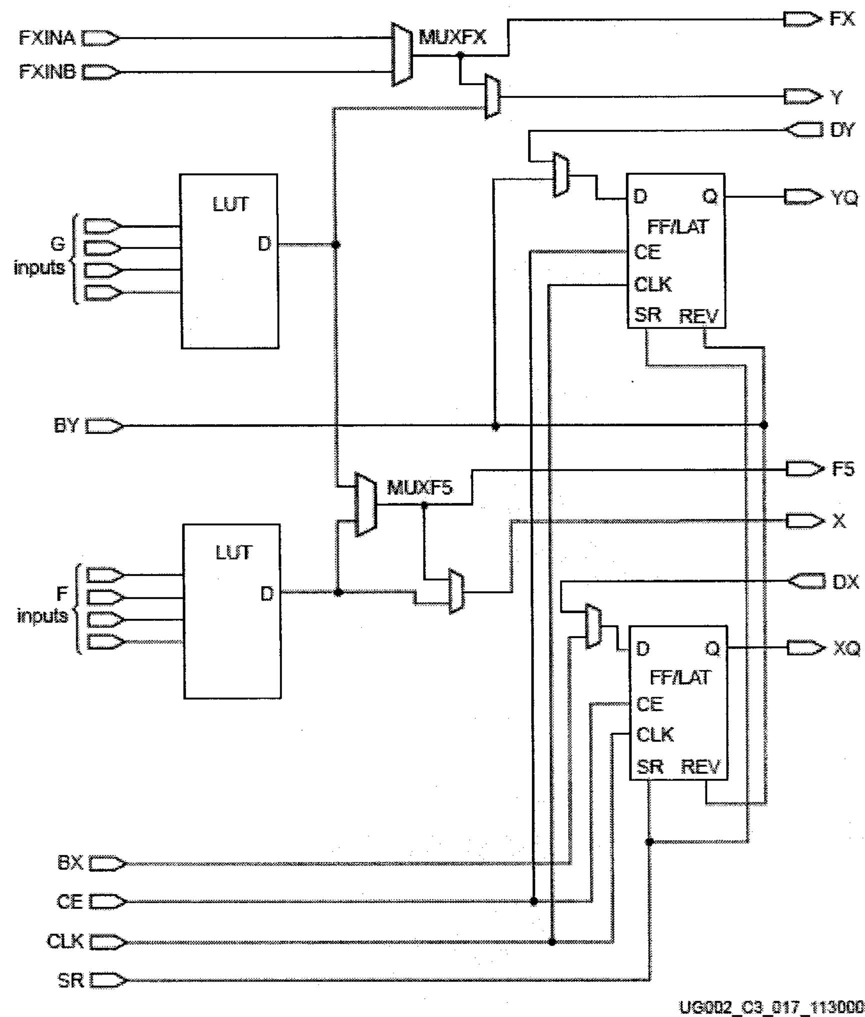


Figure 4.3: Virtex II General Slice

As the most important unit in the device, CLBs are responsible for all logic functions in FPGA design. A close look of a slice diagram is shown in Figure 4.3.

In particular, Virtex II FPGAs use *Look-Up Tables* (LUTs) to implement logic functions. Therefore, the combination function can be implemented with a fix –propagation delay. Moreover, LUTs also can be configured as memory units (distributed RAM) or shift registers.

4.3 Fixed-Point Representation of the Algorithm

The distance-based algorithm described in the previous chapter is investigated in the floating-point domain. Directly implementing such an algorithm is almost impossible for a practical application; thus, fixed-point number representation using a quantization technique is a necessary step towards an actual system implementation.

Quantization strategy for turbo codes is discussed in many literatures [32] [33] [34]. There are two major quantization techniques: uniformed quantization and non-uniformed quantization. In this thesis, we only discuss the uniformed quantization, since it is much easier to implement.

In figure 4.4, we show a typical uniformed 4-bit quantization scheme where D is the size of quantization step.

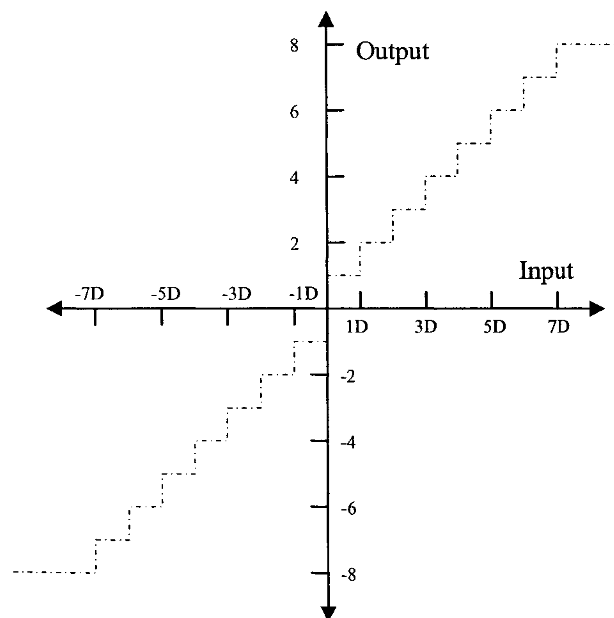


Figure 4.4: 4-bit uniformed quantization

It is predictable that a fixed-point number representation will result in certain performance loss, since values lying anywhere within a quantization interval are assigned the same value for computer processing and the original amplitude value cannot be recovered without error.

In order to study the impact of different quantization bit width on the BER, a C++ program similar to the one in the appendix is developed to simulate the algorithm in a floating point domain with different bit width settings. The effects of quantization bits on performance for a BTC$(64,51,6)^2$ using 4 test patterns after 5 iterations are illustrated in Figure 4.5. It shows that a 4-bit quantization scheme (16 level, D=0.2) provides a reasonable compromise between complexity and performance.

To be consistent, all the hardware implementations shown later in this section will use a 4-bit quantization scheme. As shown in 4.5, the hardware implementation of the distance-based algorithm will result in an approximately 0.1 dB performance loss compared to the floating-point version of the algorithm.

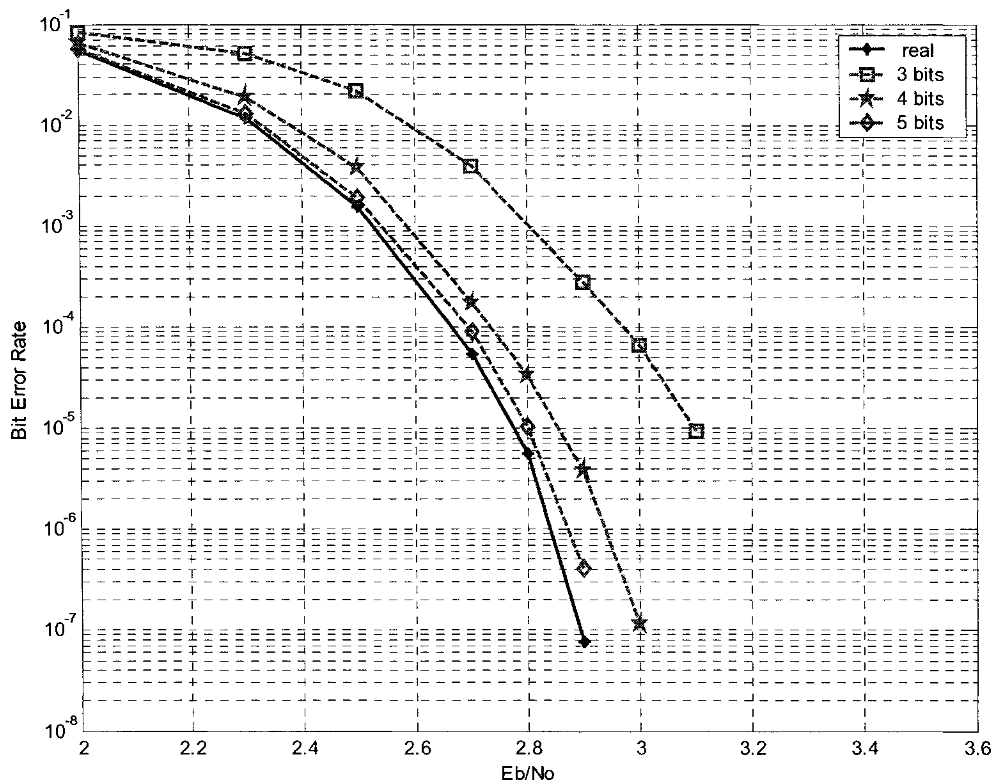


Figure 4.5: Quantization results for BTC$(64,51,6)^2$ using 4 test patterns after 5 iterations

4.4 FPGA Design of the Distance-Based Algorithm

Hardware implementation of the distance-based algorithm on FPGA platform is quite straight forward. In this section, we present the FPGA design for the new algorithm using BTC$(64,51,6)^2$. First, we will introduce the two fundamental units, an elementary decoder and a block interleaver. Then two possible hardware structures are presented.

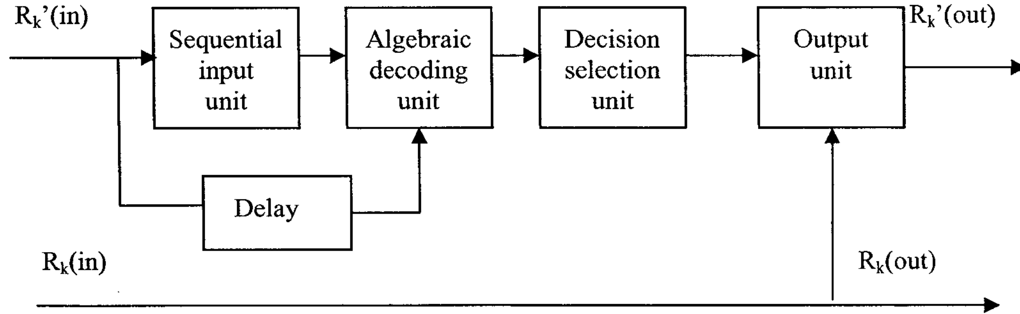


Figure 4.6: Elementary decoder (half-iteration)

4.3.1 Elementary Decoder

The core of the decoder is a half-iteration decoder, namely elementary decoder. As shown in Figure 4.5, it consists of four major units explained as follows:

- Sequential input unit, which operates at the rhythm of input symbols. The main function of this unit is to locate p LRBs, and calculate initial values (including destructive distance, syndrome, and parity bit). The information will be used to resemble 2^p test sequences in the next block.
- Algebraic decoding unit, which performs BCH decoding of all test sequences. In our approach, the unit consists of 2^p BCH decoders, which enables all test sequences to be processed simultaneously. Thus, it is clear that the total complexity of the unit is depending on the number of test patterns used in the design.
- Decision selection unit, which determines the codeword with the minimum destructive Euclidean distance from $\boldsymbol{R}$.

- Output unit, which generates the soft output for next iteration. The unit is realized as a lookup table or a block of memory. The function table $w_j = g(r_j, d_j, Dist_{des})$ is pre-calculated using Equation

$$w_j = d_j\left(\frac{\sigma^2}{2}\ln\left(\frac{\phi + \exp(2r_j d_j / \sigma^2)}{1-\phi}\right) - r_j d_j\right)$$

  where σ is assumed to be known. For practical considerations, we only need to save the absolute values of the possible extrinsic information and the sign bit can be found by observing d_j. The value is rounded to the closest level corresponding to the quantization scheme. An example of the lookup table for $BTC(64,51,6)^2$ with 16 quantization levels is shown as in Table 4.2.

Table 4.2: Soft outputs lookup table (absolute value, 4-bit quantization)

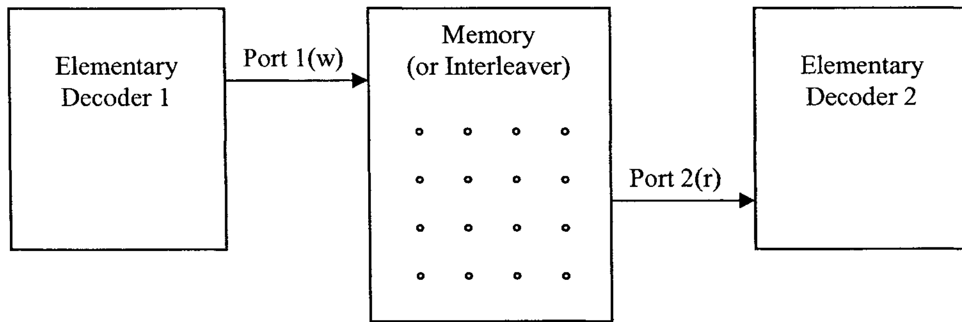
r_j*d_j \ Dist	<9	9	10	11	12	13	14	>14
0.1	1.1	0.7	0.5	0.5	0.3	0.1	0.1	0.1
0.3	1.1	0.7	0.5	0.5	0.3	0.1	0.1	0.1
0.5	0.9	0.5	0.5	0.3	0.3	0.1	0.1	0.1
0.7	0.9	0.5	0.5	0.3	0.3	0.1	0.1	0.1
0.9	0.9	0.5	0.5	0.3	0.3	0.1	0.1	0.1
1.1	0.9	0.5	0.5	0.3	0.3	0.1	0.1	0.1
1.3	0.9	0.5	0.5	0.3	0.3	0.1	0.1	0.1
1.5	0.9	0.5	0.5	0.3	0.3	0.1	0.1	0.1
-0.1	1.1	0.7	0.7	0.5	0.3	0.3	0.1	0.1
-0.3	1.3	0.9	0.9	0.7	0.5	0.3	0.1	0.1
-0.5	1.5	1.1	0.9	0.9	0.7	0.5	0.3	0.1
-0.7	1.7	1.3	1.1	1.1	0.9	0.7	0.5	0.1
-0.9	1.9	1.5	1.3	1.3	1.1	0.9	0.7	0.1
-1.1	2.1	1.7	1.5	1.5	1.3	1.1	0.9	0.1
-1.3	2.3	1.9	1.7	1.7	1.5	1.3	1.1	0.1
-1.5	2.5	2.1	1.9	1.9	1.7	1.5	1.3	0.1

4.4.2 Memory Design

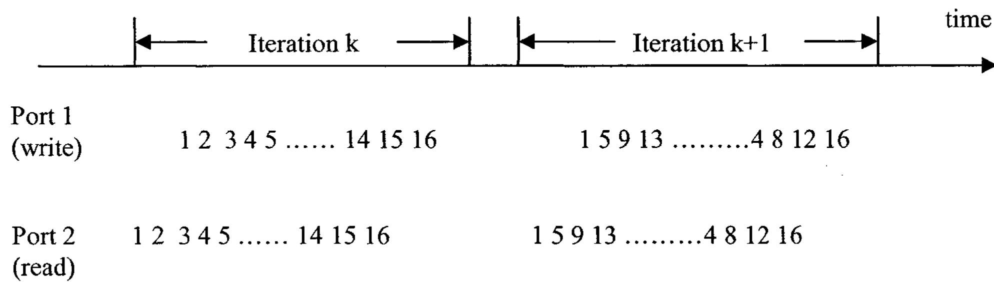
Iterative decoding requires a large amount of memory to store data. As shown in Figure 4.3, two types of data have to be saved during the decoding. They are the original received data sequence $\boldsymbol{R}$ and the output soft data sequence $\boldsymbol{R}^{'}$. Both of them have the same size, which is calculated as bit-width * code length. In our approach, since code length is 4096 and quantization bit width is 4, each matrix requires a 4x4096(16k) memory. The number of matrix is depending on the actual architecture used in the design which will be discussed later in this chapter.

Virtex II devices provide two types of memory resources: Block SelectRAM, and Distributed SelectRAM. In general, Block SelectRAM is a designated unit suitable for storing a large blocks of memory. Therefore, in our design, it is used for saving both data matrixes $\boldsymbol{R'}$ and $\boldsymbol{R}$. Distributed SelectRAM is realized by reconfiguring CLB, and is suitable for relatively small memory blocks such as lookup tables and FIFOs(Fist-in First-out). It is adopted for implementing the extrinsic lookup table as shown in Table 4.2.

The memory blocks also function as a block interleaver during the decoding process. Figure 4.7(a) illustrates a possible connection between memory and decoders. In the figure, elementary decoders are pipelined with an interleaver (memory) in the middle. For each decoder, the memory provides a separate port (this feature is supported by BlockRAM dual-port technology), which can be used independently for either reading or writing operations. Figure 4.7(b) demonstrates a simple interleaver example where 16



(a)



(b)

Figure 4.7: Interleaver example (a) interconnections between interleaver and decoders (b) interleaver operation

symbols are arranged in 4 rows and 4 columns. Let us assume previous data from decoder 1 is saved in column wise. Starting with a new iteration, the decoder 2 reads the data in the memory by rows through port 2. On the other hand, new soft output data from decoder 1 is ready after a short delay. Thus, the used data in the memory will be updated by the new date sequence. Observation from Figure 4.7(b) proves that no conflict happens during the memory assessment from two ports.

4.4.3 Iterative Decoding Architecture

So far, we have introduced two major components, an elementary decoder and a memory block, in the hardware design. Now, we need to assemble the components to form an iterative decoding architecture. Analysis of the algorithm indicates that there are two different types of decoder architectures which can be adopted in the design depending on the number of elementary decoders used.

Prototype A

Figure 4.8 illustrates a possible architecture of a BTC decoder. Notice there is only one elementary decoder in this design. Thus, all iterations are performed in the same component. Obviously, such design does not consume much chip area. Consequently, relatively low-cost chips can be adopted for the algorithm implementation. However, the major drawback of this architecture is the data rate. It is clear that once a block of data sequence enters the decoding process, other blocks of data have to be waiting in the line until the current iterations are completed. It is clear that, as the number of iterations grows, the output data rate becomes very slow.

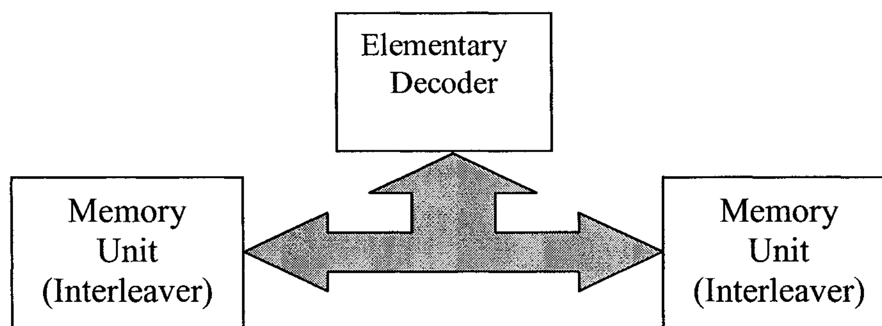


Figure 4.8: Single-elementary-decoder structure

Prototype B

In Figure 4.9, another prototype is introduced where several elementary decoders are concatenated to form a pipeline. Consequently, all decoders are synchronized in the decoding process with a goal to maximize the date rate. The number of elementary decoders required is *2t*, where *t* is the total iteration steps. Of course, the associated chip area is usually very large. The total chip area can be estimated as *2t* times of that of Prototype A.

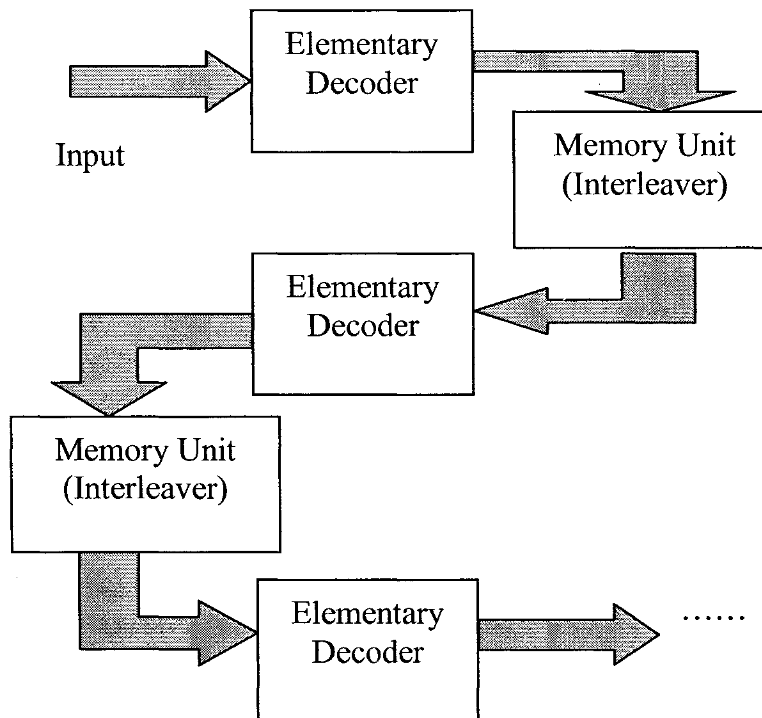


Figure 4.9 Pipeline structure

4.5 Synthesis Results

In this section, we compare the hardware complexity of the distance-based algorithm with the traditional algorithm. A commercially available product TC3000, developed by *Turbo Concept*, is chosen as a reference. TC3000 is a family of BTC decoders customizable with VHDL generics and its data sheet is available in [30].

To be consistent, we should use the same configurations in our approach. Therefore, turbo decoders which perform 5 iterations on $BTC(64,51,6)^2$ with 16 quantization levels(4-bit data width) are investigated. All our designs shown in the thesis are targeting on the Xilinx VirtexII device family. They are implemented by high level VHDL language, and automatically synthesized by Synplify Pro. Functional simulations are performed in ModelSim, where a C++ program was used to validate the results.

Synthesis results are evaluated in terms of surface area, data rate, and performance (BER). Similar to the members in *TC3000* family, several designs with different architectures and parameters are developed for meeting the system requirement.

4.5.1 Minimization of Surface Area

For applications in this category, cost control is usually a main concern. Relatively low date rates and moderate performance is affordable. Therefore, prototype A which uses only one elementary decoder is a suitable solution in this case. In addition, to further decrease the complexity, 4 test patterns are used for generating codeword list. Synthesis results show that the FPGA design using the new algorithm only consumes 550 CLBs,

which can be fit in a very low cost chip XC2V 250. Compared to TC3021 that uses the traditional algorithm, our new approach achieves up to 79% complexity saving with a cost of 0.1 dB degradation in performance. Details are explained in Table 4.3.

Table 4.3: Comparison to TC3021 (chip area)

	Traditional Algorithm (TC3021)	New Algorithm
CLB Slices	2682	550
Data Rate	7Mbits/s	8Mbits/s
EbNo@10^{-5}	2.7dB	2.8dB
Test Patterns	16	4

4.5.2 Maximization of Data Rate

Prototype B which uses pipeline structure can maximize the overall data output throughput. Therefore, it is adopted for the applications in this category. Moreover, if the error performance is not a major concern, we recommend using 4 test patterns for all elementary decoders. The final synthesis result of the new algorithm is compared to the product TC3024 as shown in Table 4.4. Our approach obtained a data rate of 65 Mbits/s, which is 2.5 times faster than that of TC3024. More importantly, by using the new algorithm, only 36% surface area is required.

Table 4.4: Comparison to TC3021 (data rate)

	Traditional Algorithm (TC3024)	New Algorithm
CLB Slices	10000	3600
Data Rate	25Mbits/s	65Mbits/s
EbNo@10^{-5}	2.7dB	2.8dB
Test Patterns	16	4

4.5.3 Optimization of Error Performance

For certain applications that wish to get a better performance, we may increase the number of test patterns in the design. Table 4.5 shows an 8-test-pattern solution, which will provide 0.1dB gain in performance over the traditional algorithm. The result shows that our design is advantageous in terms of area, data rates, and performance.

Table 4.5: Comparison to TC3024 (performance)

	Traditional Algorithm (TC3024)	New Algorithm
CLB Slices	10000	6080
Data Rate	25Mbits/s	65Mbits/s
EbNo@10^{-5}	2.7dB	2.6dB
Test Patterns	16	8

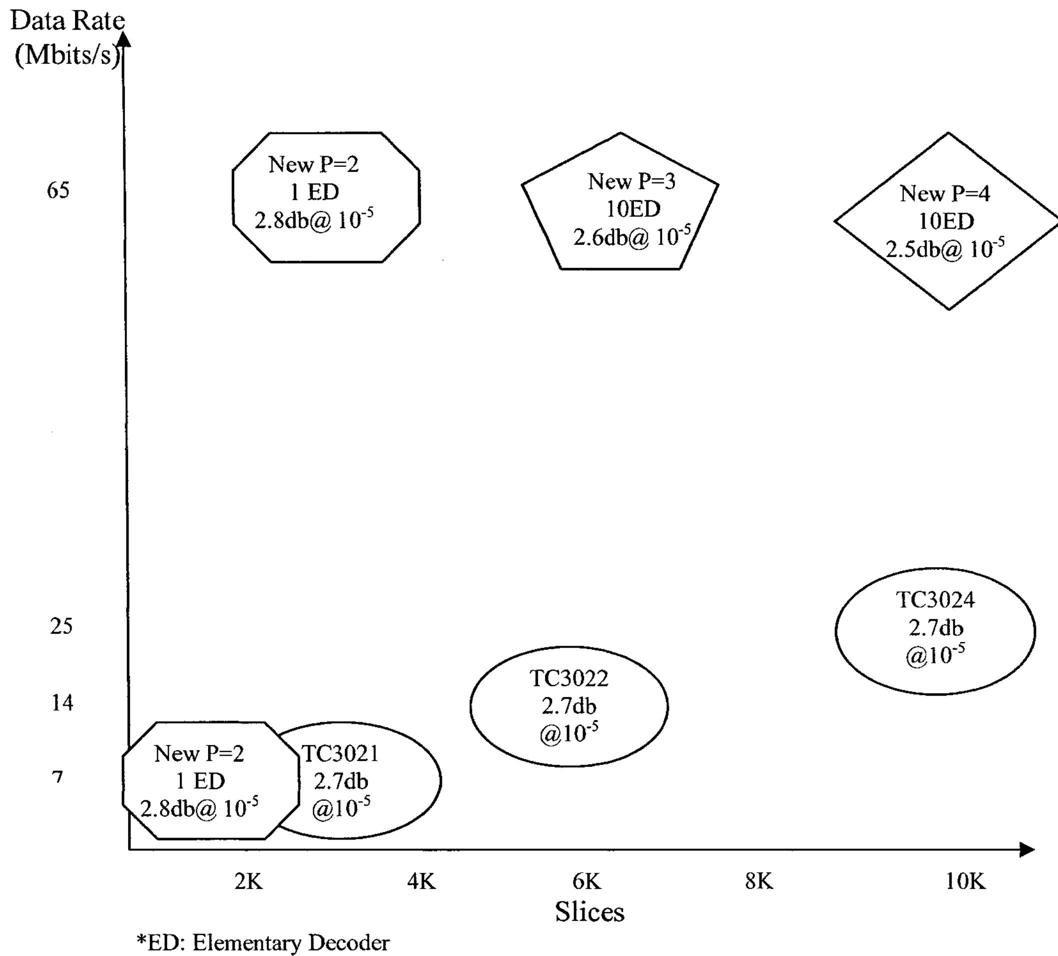


Figure 4.10: Bit rate vs. complexity trade-off

Of course, in some cases where higher performance is required, a decoder using 16 test patterns can also be considered.

Figure 4.10 summarizes our results in a comparison chart and gives a synthetic view of the data-rate/ complexity trade-off.

Chapter 5

Conclusions and Future Works

5.1 Conclusions

As a member of the turbo code family, block turbo codes exhibit excellent error performance and are widely used in many applications including satellite communications and digital video broadcasting. This thesis tries to give a broad view on BTCs, with focus on the decoding algorithms that are available. Both trellis-based algorithms and augmented list algorithms are explained in great detail.

An important contribution of our work is the proposal of a new algorithm that delivers soft outputs using the distance property of a code word. This solution can improve the performance over the traditional algorithm by 0.3 dB for BTCs having large hamming distance and have negligible performance loss for BTCs having short Hamming distance. From the point of view of computational complexity, the implementation of the proposed algorithm is much simpler than the traditional algorithm, since the soft outputs are generated via a lookup-table. Moreover, from the investigation of simulation results, we have noticed the fact that the new algorithm is able to achieve a moderate performance by

using a small number of test patterns. This leads to a large saving in hardware complexity. As a proof, the computational complexity of hardware implementations for the new algorithm is investigated on Xilinx FPGA Virtex II platforms, showing significant saving is achievable. For example , a BTC$(64,51,6)^2$ decoder that has a throughput of 8 Mbps by using 4 test patterns consumes only 550 control logic block (CLB) slices, which achieves about a 79% complexity saving in comparison to the commercially available product *TC3000*. Therefore, for practical considerations, it is suitable for high data rate applications.

5.2 Future Directions

Some of the issues that may be considered for future works are:

- Improvement of the proposed algorithm: The key step in the new algorithm is to find the confidence value by the destructive Euclidean distance of a decision codeword. The way that we use to obtain the confidence value via exhaustive simulations may be replaced by other better strategies. Also, considering the confidence value as a function of the destructive Euclidean distance may not be sufficient. For future research, we should try to improve the accuracy of estimation by using more parameters, such as building a lookup table as $\phi = f(Dist_{des}, t)$, where the confidence value can be adaptive for each decoding stage.
- Evaluation of error performance: As we all know, for most of turbo decoding algorithms, evaluation of error performance is always a tough job since Monte Carlo simulation of a turbo code is usually very time consuming. It is proved that

performance of certain codes can be evaluated by bounding technology with acceptable accuracy. For the distance-based algorithm, as simulation is the only way available for verifying the error performance, we hope that similar analytical method can be developed for evaluating the performance in the near future.

- Reduction of complexity and latency: In fact, the computational complexity may be further reduced. One possible solution is to save the candidate codewords in memory and use them again in the next iterations. This can decrease the total amount of hard decoding and significantly reduce the computational complexity.
- Decoding for other product codes: We have investigated the error performance for BTCs using inner BCH codes. Obviously, the new algorithm can be easily extended to other product codes. One example could be BTCs using Reed-Muller component codes, which also can be decoded by trellis based algorithms with maximum likelihood decoding. This work can be helpful to evaluate the proposed algorithm compared to an optimum decoder.

Bibliography

[1] C. Shannon, "A Mathematical Theory of Communication," *Bell syst. Tech.*, vol. 27, pp. 379-656, July 1948.

[2] E. Berlekamp, "On Decoding Binary Bose-Chadhuri-Hocquenghem Codes," *IEEE Trans. Inform. Theroy*, Vol. IT-11, pp. 577-579, 1965.

[3] J. L. Massey, "Threshold Decoding," Cambridge, MA, MIT Press, 1963.

[4] D. Chase, "A Class of Algorithm for Decoding Block Codes with Channel Measurement Information," *IEEE Trans. On Inform. Theory,* vol. IT-18, pp. 170-182. Jan. 1972.

[5] J. K. Wolf, "Efficient Maximum-Likelihood Decoding of Linear Block Codes," *IEEE Trans. On Inform. Theory*, vol. IT-24, pp. 76-80. Jan. 1978.

[6] G. D. Forney, "Coset codes II: Binary Lattices and Related Codes," *IEEE Trans. On Inform. Theory*, Vol. 34, No. 5, pp. 1152-1187, Sept. 1988.

[7] P. Elias, "Error-Free Coding," *IEEE Trans. Inform. Theroy*, pp. 29-37, Sept. 1954.

[8] A. Viterbi, "Error Bounds for Convolutional codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Trans. On Inform. Theory*, vol. IT-13, pp. 260-269, Apr. 1967.

[9] G. D Forney, "Concatenated Codes," Cambridge, MA, MIT Press, 1966.

[10] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon Limit Error-correcting Coding and Decoding: Turbo Codes," *Proc. of the 1993 Int. Conf. on Commun.*, ICC1993, Geneva, Switzerland, May 1993.

[11] J. Hagenauer, E. Offer, L. Papke, "Iterative Decoding Binary Block and Convolutional Codes," *IEEE Trans. On Inform. Theory*, Vol 42, No. 2, pp. 429-445, Mar. 1996.

[12] C. Berrou, and A. Glavieux, "Near Optimum Limit Error-correcting Coding and Decoding: Turbo Codes," *IEEE Trans. Commun.* Vol. 44, No. 10, pp 1261-1271, Oct. 1996.

[13] S. Benedetto, D. Divalar, G.Montorsi, and F. Pollara, "Serial Concatnation of Interleaved Codes: Performance Analysis, Design, and Iterative Decoding," *IEEE Trans. On Inform. Theory*, Vol 44, No. 3, pp. 909-926, May. 1998.

[14] R. M. Pyndiah, A. Glavieux, A. Picart, and S. Jacq, "Near-optimum decoding of product codes," in Proc. *IEEE GLOBECOM, San Francisco, USA,* pp. 339-343, Nov. 1994.

[15] A. Picart, and R. M. Pyndiah, "Adapted Iterative Decoding of Product Codes," in Proc. *IEEE GLOBECOM'99,* pp. 2357-2362, Nov. 1999.

[16] R. M. Pyndiah, "Near-optimum decoding of product codes: Block turbo codes," *IEEE Trans. Commun.*, vol. 46, pp. 1003–1010, Aug. 1998.

[17] R. M. Pyndiah, Pierre Combelles, P. Adde, " A very Low Complexity Block Turbo Codes," in Proc. *IEEE GLOBECOM,* London , pp. 101-105, Nov. 1996.

[18] S. A. Hirst, B. Honary, and G. Markarian, "Fast Chase algorithm with an application in turbo decoding," *IEEE Trans. Commun.*, vol. 49, pp.1693–1699, Oct. 2001.

[19] S. Dave, J. Kim, and S. C. Kwatra, "An efficient decoding algorithm for block turbo codes," *IEEE Trans. Commun.*, vol. 49, pp. 41–46, Jan. 2001.

[20] S. Kerouedan, and P. Adde, "Block turbo codes: towards implementation", *IEEE Conf.* , Vol. 3, pp. 1219 – 1222, Sept. 2001.

[21] P. Adde, R. M. Pyndiah " Recent Simplifications and improvements Block Turbo Codes," in Proc. *Int. Symp.* On Turbo Codes and Related Topics, Brest, France, pp. 133-136, Sept. 2000.

[22] S. Kerouedan and P. Adde, "Implementation of a Block Turbo Decoder on a Single Chip," in Proc. *Int. Symp.* On Turbo Codes and Related Topics, Brest, France, pp. 243-246, Sept. 2000.

[23] M. P. C. Fossorier and S. Lin, "Soft-decision decoding of linear block codes based on ordered statistics," *IEEE Trans. Inform. Theory*, vol. 41, pp. 1379–1396, Sept. 1995.

[24] P.A.Martin, A.Valembois, M.P.C.Fossorier, and D.P. Taylor, " On soft-input soft-output decoding using box and match techniques", *IEEE Trans. Commun.* Vol. 52, pp. 2033 – 2037, Dec. 2004.

[25] P. A. Martin, D. P. Taylor, and M. P. C. Fossorier, "Soft-input softoutput list-based decoding algorithm," *IEEE Trans. Commun.*, vol. 52, pp. 252–262, Feb. 2004.

[26] L.Bahl, J. Cocke, F. Jelinek, and J. Raviv, " Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 284-287, Mar. 1974.

[27] D. J. Muder, "Minimal Trellises for Block Codes," *IEEE Trans. Inform. Theory*, Vol. 34, No. 5, pp. 1049-1053, Sept. 1988.

[28] F. R. Kschischang and V. Sorokine, "On the Trellis Structure of Block Codes," *IEEE Trans. Inform. Theory*, Vol. 41, No. 6, pp. 1924-1937, Nov. 1995.

[29] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of Error Correcting Group Code," Inf. *Control*, Vol. 3, pp. 68-79, March 1960.

[30] Xilinx Alliance Core-Turbo Concept, "TC3000 data sheet", [*online*], Available: http://www.xilinx.com/products/logicore/alliance/ turboconcept/ turbo concept _tc 3000. pdf

[31] G. E. P. Box, and M. E. Muller, "A Note on the Generation of Random Normal Deviates," *Ann. Math. Stat.* 29, 610-611, 1958.

[32] Y.Wu and B.D. Woerner, "The Influence of Quantization and Fix-Point Arithmetic Upon the BER performance of Turbo Codes," in Proc. *IEEE Inter. Conf.* on Vehicular Technology (VTC'99), Vol. 2 pp 1683-1687, May. 1999.

[33] Y. Wu and B. D. Woerner, "Internal Data Width SISO Decoding Module with Modular Renormalization," in Proc. *IEEE Inter. Conf.* on Vehicular Technology (VTC'00), Tokyo, Japan, May. 2000.

[34] H. Michel, A. Worm and N. When, "Influence of Quantization on the Bit-Error Performance of Turbo Codes," in Proc. IEEE *IEEE Inter. Conf.* on Vehicular Technology (VTC'00), Tokyo, Japan, May. 2000.

[35] S. Lin and D. J. Costello, "Error Control Coding Fundamentals and Applications," Prentice-Hall, Englewood Cliffs, USA, 1983.

[36] M. Reza Soleymani, Yingzi Gao, U. Vilaipornsawai, " Turbo Coding for Satellite and Wireless Communications," Kluwer Academic Publishers ,2002

Appendix Simulation Program for Distance-Based Algorithm

```
//=================================================================
// Designed by Nong Le          Jan.10.2005
// This simulation program is used for BTC(64,51,6)^2
// File name: BCHendecode.h: interface for the BCHTurboEncoder class.
// Created by Visaul C++ 6.0
// m is 6
// n is 64
// k is 51
// input length is 2601
// output length is 4096
// component codes is extended BCH(64,51)
//=================================================================

#if !defined(afx_BCHendecode_h__e3a07593_a2c1_405d_9acc_0c613ce7bebd__included_)
#define afx_BCHendecode_h__e3a07593_a2c1_405d_9acc_0c613ce7bebd__included_


#if _msc_ver > 1000
#pragma once
#endif // _msc_ver > 1000



const   int     m =  6 ;
const   int     q =  64;
const   int     n =  64;
const   int     k =  51;
const   int     ITERATIONS =  5;
const   int     SAMPLES =  100000;
const   double  EBNO =  2.5;
const   int     T_CORRECTABLE =  2 ;
const   int     T_TESTING_BITS =  4;
const   int     LISTLENGTH =  16;
const   int     INPUTLENGTH =  2601;
const   int     OUTPUTLENGTH =  4096;


const   int     GF_generator[]={1,1,0,0,0,0,1};
const   int     code_generator[]={1,0,0,1,1,1,0,0,1,0,1,0,1};
```

```
class BCH
{
public:
	bool locate_decision_code();
	void get_distance(double *input);
	void decode_list();
	void generate_test_sequence(double *inputseq, int *pos);
	void locate_LRB_bits(double *input, int *pos);
	void do_soft_decode(double *input, double *extrinsic_matrixinput,
			double *extrinsic_matrixoutput, int * decision);
	void get_syndrome(int *recievedpoly);
	bool do_decode(int *input, int *output,int listpos);
	void do_encode(int *input, int *output);
	int syndrom[T_CORRECTABLE*2],tempn[n],tempk[k];
	int Q[T_CORRECTABLE*2+2][6];
	int du[T_CORRECTABLE*2+2],u[T_CORRECTABLE*2+2],lu[T_CORRECTABLE*2+2];
	int position_of_decision;
	bool is_valid_code[LISTLENGTH];
	int test_sequence[LISTLENGTH][n],decoded_sequence[LISTLENGTH][n];
	double destructive_distance[LISTLENGTH];
	BCH();
	virtual ~BCH();
};

class BCHTurboEncoder
{
public:
	void do_encode(int *input,int *output);
	void construct(BCH *com1,BCH *com2);
	BCHTurboEncoder();
	virtual ~BCHTurboEncoder();
	BCH *component1,*component2;
	int tempk1[k],tempk2[k],tempn1[n],tempn2[n];
};

class BCHTurboDecoder
{
public:
	void initial_extrinsic();
	double extrinsic_matrix1[n][n],extrinsic_matrix2[n][n];
	void construct(BCH *com1,BCH *com2);
	void do_decode(double *input,int *output);
	BCHTurboDecoder();
	virtual ~BCHTurboDecoder();
	BCH *component1,*component2;
	int tempdecision1[n],tempdecision2[n];
	int ITERATIONSativetimes;
	double temp_extrinsic1[n],temp_extrinsic2[n],tempn1[n],tempn2[n];
};

#endif // !defined(afx_BCHendecode_h__e3a07593_a2c1_405d_9acc_0c613ce7bebd__included_)
```

```
//=====================================================================
// File name: BCHendecode.cpp
//=====================================================================

#include "BCHendecode.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
double  extrinsic__lookup_table[2][41][40];
double  sigma=sqrt((double)OUTPUTLENGTH/(double)INPUTLENGTH*0.5*pow(10,(-
(double)EBNO/10)));
double  sigma27=sqrt((double)OUTPUTLENGTH/(double)INPUTLENGTH*0.5*pow(10,(-
(double)2.7/10)));

int count_errors(int *input,int *output);
int alpha[q][m],GF_index[q],GF_add_table[q][q],GF_multiply_table[q][q];  //use for GF_

//=====================================================================
// generate random message and add gaussian noise
//=====================================================================

void generate_message(int *message,int length)
{
  double r;
        for (int i=0; i<length;i++)
        {
                r=((double)rand())/RAND_MAX;
                message[i]=(int)(r+0.5);
        }
}

double ranf()
{
        return ((double)rand())/RAND_MAX;
}

int count_errors(int *input,int *output)
{
        int result=0;
        for (int i=0;i<k;i++)
        {
                for (int j=0;j<k;j++)
                {
                        if (input[i*k+j]!=output[i*n+j]) result++;
                }
        }
        return result;
}
```

```
void pass_awgn_channel(int *input, double *output, int length,double sigma)
{
        double x1, x2, w, y1, y2;

        int half=(int)length/2;
        for(int i=0; i<half;i++)
        {
              do {
                        x1 = 2.0 * ranf() - 1.0;
                        x2 = 2.0 * ranf() - 1.0;
                        w = x1 * x1 + x2 * x2;
                 } while ( w >= 1.0 );

                 w = sqrt( (-2.0 * log( w ) ) / w );
                 y1 = sigma * x1 * w;
                 y2 = sigma * x2 * w;
                 output[i]=y1+input[i]*2.0-1.0;
                 output[i+half]=y2+input[i+half]*2.0-1.0;


        }
        if ((double)half!=((double)length/2))
        {
                 do {
                        x1 = 2.0 * ranf() - 1.0;
                        x2 = 2.0 * ranf() - 1.0;
                        w = x1 * x1 + x2 * x2;
                 } while ( w >= 1.0 );
                 w = sqrt( (-2.0 * log( w ) ) / w );
                 y1 = sigma * x1 * w;
                 output[length-1]=y1+input[length-1]*2.0-1.0;
        }
}


//==================================================================
// generate extrinsic output lookup table
//==================================================================
void generate_extrinsic_lookup_table()
{
        double relibilty_base_on_distance[40];
        for (int inter=0; inter<=9;inter++)
        {
                 relibilty_base_on_distance[inter]=0.99;
        }

        for (inter=14; inter<40;inter++)
        {
                 relibilty_base_on_distance[inter]=0;
        }
        relibilty_base_on_distance[8]=0.99;
        relibilty_base_on_distance[9]=0.93;
        relibilty_base_on_distance[10]=0.9;
```

```
        relibilty_base_on_distance[11]=0.82;
        relibilty_base_on_distance[12]=0.65;
        relibilty_base_on_distance[13]=0.42;
        relibilty_base_on_distance[14]=0.21;

        for(int dis=0;dis<40;dis++)
        {
                for( int inp=0;inp<41;inp++)
                {
                        double douinp=((double)inp-20)/10.0;
                        double signal=-1;
                        double logp=2*douinp/(sigma27*sigma27);
                        double pplus=(exp(logp))/(exp(logp)+1);
                        double pcorrect=relibilty_base_on_distance[dis];
                        double pafter=((1-pcorrect)*pplus)/(pcorrect+(1-pcorrect)*(1-pplus));
                        double rafter=log(pafter)*sigma27*sigma27/2;
                        extrinsic__lookup_table[0][inp][dis]=rafter-douinp;
                        if( (pplus==0)||(pplus==1)||(pcorrect==0))
                        extrinsic__lookup_table[0][inp][dis]=0;
                }
        }
        for( dis=0;dis<40;dis++)
        {
                for( int inp=0;inp<41;inp++)
                {
                        double douinp=((double)inp-20)/10.0;
                        double signal=1;
                        double logp=2*douinp/(sigma27*sigma27);
                        double pplus=(exp(logp))/(exp(logp)+1);
                        double pcorrect=relibilty_base_on_distance[dis];
                        double pafter=(pcorrect+(1-pcorrect)*pplus)/((1-pcorrect)*(1-pplus));
                        double rafter=log(pafter)*sigma27*sigma27/2;
                        extrinsic__lookup_table[1][inp][dis]=rafter-douinp;
                        if( (pplus==0)||(pplus==1)||(pcorrect==0))
                                extrinsic__lookup_table[1][inp][dis]=0;
                }
        }
}


//=====================================================================
// golias field math
//=====================================================================

int array_to_number(int *array_point)
{
        int result=0;
        int bi=1;
        for(int i=m-1 ; i>=0;i--)
        {
                result=result+array_point[i]*bi;
                bi=bi*2;
        }
        return result;
}
```

```
void GF_add(int *a, int *b,int* result)
{
	for (int i=0;i<m;i++)
	{
		if(a[i]==b[i]) result[i]=0;
		else result[i]=1;
	}
}

int get_GF_poly_value(int *p, int length, int x)
{
	int temp,result,tempm;
	result=0;
	temp=1;
	for (int i=0;i<length;i++)
	{
		if (p[i]!=0)
		{
			tempm=GF_multiply_table[temp][p[i]];
			result=GF_add_table[result][tempm];
		}
		temp=GF_multiply_table[temp][x];
	}
	return result;
}

int get_GF_poly_reversed_value(int *p, int length, int x)
{
	int temp,result;
	result=0;
	temp=1;
	for (int i=length-1;i>=0;i--)
	{
		if (p[i]==1)
		{
			result=GF_add_table[result][temp];
		}
		temp=GF_multiply_table[temp][x];
	}
	return result;
}

int GF_inverse(int x)
{
	int result=q-x+1;
	if (result==q) result=1;
	return result;
}

void GF_poly_convolution(int *poly1, int length1, int *poly2, int length2, int *result)
{
	for (int i=0;i<length1+length2-1;i++)
	{
		result[i]=0;
	}
```

```
for (int j1=0; j1<length1;j1++)
        {
                for (int j2=0; j2<length2;j2++)
                {
                        int temp=GF_multiply_table[poly1[j1]][poly2[j2]];
                        result[j1+j2]=GF_add_table[result[j1+j2]][temp];
                }
        }
}

void GF_poly_add(int *poly1, int length1, int *poly2, int length2, int *result)
{
        int length;
        if (length1>=length2) length=length1;
        else length=length2;
        int tempa,tempb;
        for (int i=0; i<length;i++)
        {
                if (length1>=(i+1)) tempa=poly1[i];
                else tempa=0;
                if (length2>=(i+1)) tempb=poly2[i];
                else tempb=0;
                result[i]=GF_add_table[tempa][tempb];
        }
}

bool generate_golais_field()
{
        for (int i=0;i<q;i++)
        {
                for (int j=0;j<m;j++)
                {
                        alpha[i][j]=0;
                }
        }
        for (int j=0;j<q;j++)
        {
                GF_index[j]=0;
        }
        for (i=0;i<q;i++)
        {

                for (int j=0;j<q;j++)
                {
                        GF_add_table[i][j]=0;
                        GF_multiply_table[i][j]=0;
                }
        }
        GF_index[0]=0;
        for (i=1; i<=m;i++)
        {
                alpha[i][i-1]=1;
                GF_index[array_to_number(alpha[i])]=i;
        }
```

```
	for (i=m+1;i<q;i++)
	{
		for (int j=0; j<m;j++)
		{
			for( int h=0;h<m;h++)
			{
				alpha[i][j]= alpha[i][j]+GF_generator[h]*alpha[i-m+h][j];
			}
			alpha[i][j]= alpha[i][j]-((int)(alpha[i][j]/2))*2;
		}
		int l=array_to_number(alpha[i]);
		GF_index[l]=i;
	}
	bool is_success=true;
	for (i=1;i<q;i++)
	{if (GF_index[i]==0)
	is_success=false;
	}

	if (is_success)
	{
		int temp[m];

		for (int j1=0;j1<q;j1++)
		{
			for (int j2=0;j2<q;j2++)
			{
				GF_add(alpha[j1],alpha[j2],temp);
				GF_add_table[j1][j2]=GF_index[array_to_number(temp)];
				if ((j1==0)||(j2==0)) GF_multiply_table[j1][j2]=0;
				else
				{
				int c=j1+j2-2;
				GF_multiply_table[j1][j2]=c-(int)(c/(q-1))*(q-1)+1;
				}
			}
		}

	}

	return is_success;
}

//===========================================================================
// BCH class
//===========================================================================

BCH::BCH()
{
}

BCH::~BCH()
{
}
```

```
void BCH::get_syndrome(int *recievedpoly)
{
        for (int i=0;i<(2*T_CORRECTABLE);i++)
        {
                syndrom[i]=get_GF_poly_reversed_value(recievedpoly,n-1,i+2);
        }
}

void BCH::do_encode(int *input, int *output)
{

        int regist[n-1-k];
        for (int i=0;i<n-1-k;i++)
        {
                regist[i]=0;
        }
        int in,out;
        for (i=0; i<n-1;i++)
        {
                if (i<k) in=input[i];
                else in=0;
                out=regist[n-1-k-1];
                for (int i2=n-1-k-1;i2>0;i2--)
                {
                        if (code_generator[i2]==0) regist[i2]=regist[i2-1];
                        else
                        {
                                if (out==regist[i2-1]) regist[i2]=0;
                                else regist[i2]=1;
                        }
                }
                if (in==out) regist[0]=0;
                else regist[0]=1;
        }
        int paritybit=0;
        for (i=0; i<k;i++)
        {
                output[i]=input[i];
                 paritybit+=input[i];
        }
        for (i=k;i<n-1;i++)
        {
                output[i]=regist[n-1-i-1];
                paritybit+=output[i];
        }
        if (paritybit==(int)(paritybit/2)*2) output[n-1]=0;
        else output[n-1]=1;


 }
```

```
bool BCH::do_decode(int *input, int *output,int listpos)
{
   int paritybit=0;
         for (int i=0; i<n;i++)
         {
                  output[i]=input[i];
         }
         get_syndrome(input);
         bool noerror=true;
         for(i=0;i<T_CORRECTABLE*2;i++)
         {
                  if (syndrom[i]!=0) noerror=false;
         }
         if (noerror)
         {
                  for ( i=0; i<n-1;i++)
                  {
                           paritybit+=input[i];
                  }
                  if (paritybit==(int)(paritybit/2)*2) output[n-1]=0;
                  else output[n-1]=1;
                  return true;
         }

         for ( i=0;i<2*T_CORRECTABLE+2;i++)
         {
                  u[i]=i-1;
                  for (int j=0;j<6;j++)
                  {
                           Q[i][j]=0;
                  }
         }
         Q[0][0]=1;
         Q[1][0]=1;
         du[0]=1;
         du[1]=syndrom[0];
         lu[0]=0;
         lu[1]=0;
         for (int j1=2;j1<(T_CORRECTABLE*2+2);j1++)
         {
                  if (du[j1-1]==0)
                  {
                           lu[j1]=lu[j1-1];
                           for(int i=0;i<lu[j1]+1;i++)
                           {
                                    Q[j1][i]=Q[j1-1][i];
                           }
                  }
                  else
                  {
                           int tempp=-1;
                           int  tempd=0;
                           int dp;
```

```
                for (int j2=0;j2<(j1-1);j2++)
                {
                        if (du[j2]!=0)
                        {
                                dp=u[j2]-lu[j2];
                                if (tempp==-1)
                                {
                                        tempp=j2;
                                        tempd=dp;
                                }
                                else  if (dp>tempd)
                                {
                                        tempp=j2;
                                        tempd=dp;
                                }
                        }
                }
                int tempq1;
                tempq1=GF_multiply_table[du[j1-1]][GF_inverse(du[tempp])];
                int  degree=j1-1-tempp;
                int tempq2[6],tempq3[6];
                for(int i=0;i<degree+1;i++)
                {
                        tempq2[i]=0;
                }
                tempq2[degree]=tempq1;
                int lengthofqp,lengthofq3,lengthoftq,lengthofq;
                lengthofqp=lu[tempp]+1;
                lengthofq3=lengthofqp+degree;
                GF_poly_convolution(tempq2,degree+1,Q[tempp],lengthofqp,tempq3);
                lengthoftq=lu[j1-1]+1;
                if (lengthoftq>=lengthofq3) lengthofq=lengthoftq;
                else lengthofq=lengthofq3;
                GF_poly_add(Q[j1-1],lengthoftq,tempq3,lengthofq3,Q[j1]);
                lu[j1]=lengthofq-1;
        }
        if (j1!=2*T_CORRECTABLE+1)
        {
                int tempdu=syndrom[j1-1];
                for(int j3=1;j3<lu[j1]+1;j3++)
                {
                        int temp1;
                        temp1=GF_multiply_table[Q[j1][j3]][syndrom[j1-j3-1]];
                        tempdu=GF_add_table[tempdu][temp1];
                }
                du[j1]=tempdu;
        }
}
int count=0;
int position;
if (lu[2*T_CORRECTABLE+1]<=2)
{
```

```
                    for (j1=1;j1<q;j1++)
                    {
          if (get_GF_poly_value(Q[1+2*T_CORRECTABLE],   lu[2*T_CORRECTABLE]+1,j1)==0)
                              {
                                        position=GF_inverse(j1)-1;
                                        output[n-1-1-position]=1-output[n-1-1-position];
                                        count=count+1;
                              }
                    }
          }
          if (count==0) return false;
          for ( i=0; i<n-1;i++)
          {
                    paritybit+=output[i];
          }
          if (paritybit==(int)(paritybit/2)*2) output[n-1]=0;
          else output[n-1]=1;
          return true;
}

void BCH::do_soft_decode(double *input, double *extrinsic_input, double *extrinsic_output, int
*decision)
{
          int pos[T_TESTING_BITS];
          double input_sum[n];
          for(int i=0;i<n;i++)
          {
                    input_sum[i]=input[i]+extrinsic_input[i];
          }
          locate_LRB_bits( input_sum,pos);
          generate_test_sequence( input_sum,pos);
          decode_list();
          get_distance(input_sum);
          if (!locate_decision_code())
          {
                    for (int j=0;j<n;j++)
                    {
                              decision[j]=tempn[j];
                              extrinsic_output[j]=0;
                    }
                    return;
          }

          for( i=0;i<n;i++)
          {
                    decision[i]=decoded_sequence[position_of_decision][i];
          }
          for( i=0;i<n;i++)
          {
                    double inputplus;
                    inputplus=input_sum[i];
                    if(input_sum[i]>2) inputplus=2.0;
                    if(input_sum[i]<-2) inputplus=-2.0;
                    int inp=(int)((inputplus+2)*10);
```

```
			extrinsic_output[i]=extrinsic__lookup_table[decision[i]][inp][(int)destructive_distanc
		e[position_of_decision]];
		}
}

void BCH::locate_LRB_bits(double *input, int *pos)
{
	for (int i=0;i<T_TESTING_BITS;i++)
	{
		pos[i]=i;
	}
	double largestvalue,temp;
	int posoflargest;
	for (i=T_TESTING_BITS;i<n-1;i++)
	{
		largestvalue=0;
		for (int i1=0;i1<T_TESTING_BITS;i1++)
		{
			temp=fabs(input[pos[i1]]);
			if(temp>largestvalue)
			{largestvalue=temp;
			posoflargest=i1;
			}
		}
		temp=fabs(input[i]);

		if (temp<largestvalue)
		{
			pos[posoflargest]=i;
		}
	}
}

void BCH::generate_test_sequence(double *input, int *pos)
{
	for (int i=0;i<n;i++)
	{
		if (input[i]>0) tempn[i]=1;
		else tempn[i]=0;
	}
	for (i=0; i<LISTLENGTH;i++)
	{
		for (int j=0;j<n;j++)
		{
			test_sequence[i][j]=tempn[j];
		}
	}
	for (i=0;i<LISTLENGTH;i++)
	{
		int bit;
		int temp_pattern=i;
		for(int p=0;p<T_TESTING_BITS;p++)
		{
			int a=temp_pattern>>1;
			int b=a<<1;
```

```
                        bit=temp_pattern-b;
                        test_sequence[i][pos[p]]=bit;
                        temp_pattern=a;
                }
        }
}

void BCH::decode_list()
{
        for(int i=0; i<LISTLENGTH;i++)
        {
                is_valid_code[i]=do_decode(test_sequence[i],decoded_sequence[i],i);
        }
}

void BCH::get_distance(double *input)
{
        for (int i=0;i<LISTLENGTH;i++)
        {
                if (is_valid_code[i])
                {
                        double tempDis=0;
                        for( int i1=0;i1<n;i1++)
                        {
                                double sig=((double)decoded_sequence[i][i1]-0.5)*2.0;
                                if (input[i1]*sig<0)  tempDis+=(input[i1]-sig)*(input[i1]-sig);
                                else if((input[i1]<1)&&(input[i1]>-1))
                                {
                                        tempDis+=(input[i1]-sig)*(input[i1]-sig);
                                }
                        }
                        destructive_distance[i]=tempDis;
                        if(destructive_distance[i]>39.9) destructive_distance[i]=39.9;
                }
                else destructive_distance[i]=39.9;
        }
}

bool BCH::locate_decision_code()
{
        double tempdist=999999;
        for(int i=0;i<LISTLENGTH;i++)
        {
                if ((is_valid_code[i])&&(destructive_distance[i]<tempdist))
                {
                        position_of_decision=i;
                        tempdist=destructive_distance[i];
                }
        }
        if (tempdist==999999)
        {
                return false;
        }
        else return true;
}
```

```
//=========================================================================
// BCHTurboEncoder class
//=========================================================================

BCHTurboEncoder::BCHTurboEncoder()
{
}


BCHTurboEncoder::~BCHTurboEncoder()
{
}

void BCHTurboEncoder::construct(BCH *com1, BCH *com2)
{
        component1=com1;
        component2=com2;
}

void BCHTurboEncoder::do_encode(int *input, int *output)
{

        for (int i=0;i<k;i++)
        {
                for (int l1=0;l1<k;l1++)
                {
                        tempk1[l1]=input[i*k+l1];
                }

                component1->do_encode(tempk1,tempn1);
                for ( l1=0;l1<n;l1++)
                {
                        output[i*n+l1]=tempn1[l1];
                }
        }
        for (int i2=0;i2<n;i2++)
        {
                for (int l1=0;l1<k;l1++)
                {
                        tempk2[l1]=output[l1*n+i2];
                }
                component2->do_encode(tempk2,tempn2);
                for (int l2=k;l2<n;l2++)
                {
                        output[l2*n+i2]=tempn2[l2];
                }
        }
}
```

```
//==========================================================================
// BCHTurboDecoder class
//==========================================================================

BCHTurboDecoder::BCHTurboDecoder()
{
}

BCHTurboDecoder::~BCHTurboDecoder()
{
}
void BCHTurboDecoder::construct(BCH *com1, BCH *com2)
{
	component1=com1;
	component2=com2;
}
void BCHTurboDecoder::initial_extrinsic()
{
	for (int i=0;i<n;i++)
	{
		for (int j=0;j<n;j++)
		{
			extrinsic_matrix1[i][j]=0;
			extrinsic_matrix2[j][i]=0;
		}
	}
}

void BCHTurboDecoder::do_decode(double *input,int *output)
{
	initial_extrinsic();
	for(int step=0;step<ITERATIONS;step++)
	  {
		double sumofdist=0;
		for(int row=0;row<n;row++)
		{
			for (int col=0;col<n;col++)
			{
				tempn1[col]=input[row*n+col];
			}
component1->do_soft_decode(tempn1,extrinsic_matrix1[row],temp_extrinsic1,tempdecision1);
			for (int i=0;i<n;i++)
			{
				extrinsic_matrix2[i][row]=temp_extrinsic1[i];
			}
		}
		for(int col=0;col<n;col++)
		{
			for (int row=0;row<n;row++)
			{
				tempn2[row]=input[row*n+col];
			}
component2->do_soft_decode(tempn2,extrinsic_matrix2[col],temp_extrinsic2,tempdecision2);
```

```
                    for (int i=0;i<n;i++)
                    {
                            extrinsic_matrix1[i][col]=temp_extrinsic2[i];
                    }
                    for ( row=0;row<n;row++)
                    {
                            output[row*n+col]=tempdecision2[row];

                    }
            }
        }
}


//==================================================================
// main program
//==================================================================

void main()
{
        cout<<"*********** turbo coding *************"<<endl;
        cout<<" designed by nong le"<<endl<<endl;
        cout<<"this is a program dealing with block turbo code "<<endl;
        BCHTurboEncoder encode;
        BCH bch1,bch2;
        generate_extrinsic_lookup_table() ;
        generate_golais_field();
        FILE *stream;
        stream= fopen("BCHturboresultlog.txt","a+");
        fprintf(stream," \n input length %6d , output length %6d",k*k,n*n);
        fprintf(stream," BCH(%4d,%4d) code with %2d error correction capibility,%2d testing
                bits", n, k,T_CORRECTABLE,T_TESTING_BITS);
        fprintf(stream," \n %2d iterations:   ",ITERATIONS);
        fprintf(stream," %10d observed code words:    \n",SAMPLES);
        fprintf(stream," \n \n *********simulation result*************\n");
        fclose(stream);
        encode.construct(&bch1,&bch2);
        int information[INPUTLENGTH],
                    encodedinfo[OUTPUTLENGTH],decodedinfo[OUTPUTLENGTH];
        double recieved[OUTPUTLENGTH];
        int error=0;
        BCHTurboDecoder decode;
        decode.construct(&bch1,&bch2);
cout<<"starting simulation"<<endl<<"  EBNO          pb         error bits      error words"<<endl;
        int errorword=0;
        for (int tim=0;tim<SAMPLES;tim++)
        {
                if ((tim+1)/1000*1000==tim+1)  cout<<"t="<<tim<<"
"<<"errorbits="<<error<<"pb="<<((double)(error))/((double)(INPUTLENGTH*tim))<<endl;
                generate_message(information,INPUTLENGTH);
                encode.do_encode(information,encodedinfo);
                pass_awgn_channel(encodedinfo,recieved,OUTPUTLENGTH,sigma);
```

```
                decode.do_decode(recieved,decodedinfo);
                int temperror=0;
                temperror=count_errors(information,decodedinfo);
                error=error+temperror;
                if(temperror>0)
                {
                        cout<<endl<<"t="<<tim<<"   "<<" error="<<temperror<<"    ";
                        errorword++;
                }

        }
        stream= fopen("BCHturboresultlog.txt","a+");
        double pb=((double)(error))/((double)(INPUTLENGTH*SAMPLES));
        cout<<setw(6)<<EBNO<<"
"<<setw(14)<<pb<<setw(16)<<error<<setw(16)<<errorword<<endl;
        fprintf(stream,"\n     eb           pb         error bits    error words  \n   ");
        fprintf(stream,"  %5.2f        %7.2e       %5d       %5d     \n",EBNO,pb,error,errorword) ;
        fclose(stream);
}
```