

On the Improvement of the Existing Error-Control Coding
Techniques for Deep Space Communications

By

Ms. Koju MATSUZAWA

Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in Telecommunications Science
Assumption University

April, 2003

On the Improvement of the Existing Error-Control Coding Techniques for Deep Space Communications

By

Ms. Koji MATSUZAWA



**Submitted in Partial Fulfillment of the
Requirement for the Degree of
Master of Science in
Telecommunications Science
Assumption University**

April, 2003

The Faculty of Science and Technology


Master Thesis Approval

Thesis Title On the Improvement of the Existing Error-Control Coding
Techniques for Deep Space Communications

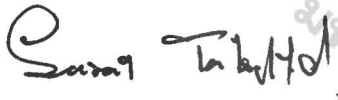
By Mr. Koju MATSUZAWA
Thesis Advisor Asst.Prof.Dr. Dobri Batovski
Academic Year 2/2002


The Department of Telecommunications Science, Faculty of Science and Technology of Assumption University has approved this final report of the **twelve** credits course. **TS7000 Master Thesis**, submitted in partial fulfillment of the requirements for the degree of Master of Science in Telecommunications Science.

Approval Committee:



(Asst.Prof.Dr. Dobri Batovski)
Advisor

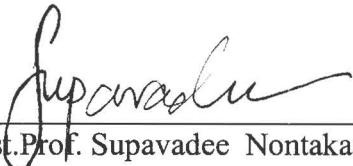

(Asst.Prof.Dr. Chanintorn J. Nukoon)
Committee Member


(Dr. Surat Tanterdtid)
Committee Member


(Asst.Prof.Dr. Surapong Auwatanamongkol)
Representative of Ministry of
University Affairs

Faculty Approval:

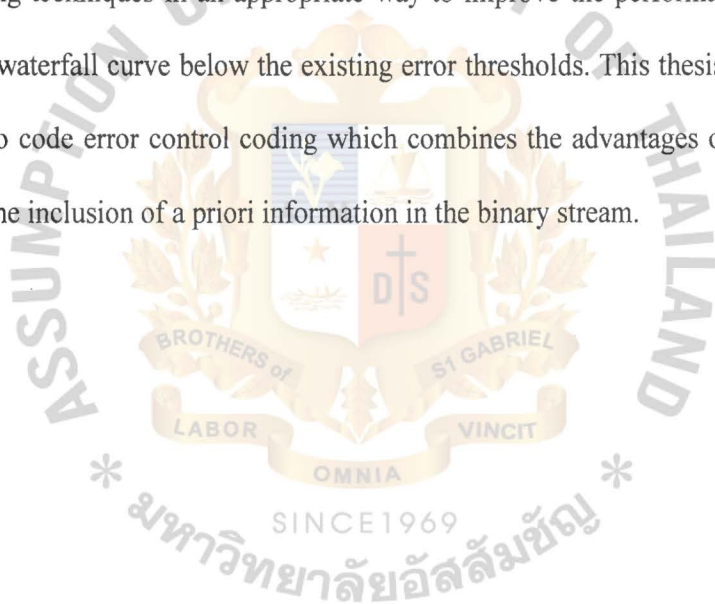

(Asst.Prof.Dr. Chanintorn J. Nukoon)
Program Director


(Asst.Prof. Supavadee Nontakao)
Dean

April / 2003

ABSTRACT

The further success in the exploration of the deep space requires reliable communications. The error-control coding plays an important role in recovering the transmitted binary information over a very long distance. An important aspect of coding in deep space is that due to the significant delay in the communication process, the automatic-repeat-request (ARQ) schemes become rather inefficient. The requirement for the strongest possible protection of the data against severe attenuation and noise raises the problem of modifying the existing coding techniques in an appropriate way to improve the performance of the bit error rate (BER) waterfall curve below the existing error thresholds. This thesis proposes the methods for turbo code error control coding which combines the advantages of the existing techniques with the inclusion of a priori information in the binary stream.



ACKNOWLEDGEMENT

The research would not have been possible without several people who have provided assistance and support. First of all, I would like to express my appreciation to Asst. Prof. Dr. Dobri Atanassov Batovski, my advisor, for his guidance, recommendation and unlimited patience: giving me the background in my limited C Language which is used for the simulation software and also the mathematical proofs.

I would like to express my special thanks to the thesis committee members, especially, to Asst. Prof. Dr. Chanintorn Jittawiriyankoon, Dr. Surat Tanterdtid and Asst. Prof. Dr. Surapong Auwatanamongkol for their constructive comments and invaluable suggestions.



TABLE OF CONTENTS

| | Page |
|---|------|
| ABSTRACT | I |
| ACKNOWLEDGEMENTS | II |
| LIST OF FIGURE | |
| LIST OF TABLES | |
| CHAPTER 1 INTRODUCTION | 1 |
| 1.1 Overview | 1 |
| 1.2 Importance of the turbo code in Deep Space Communication | 4 |
| 1.3 Use of priori redundancy | 4 |
| 1.4 Proposed Framework | 4 |
| CHAPTER 2 OBJECTIVES | 6 |
| CHAPTER 3 LITERATURE REVIEW | 7 |
| 3.1 Source-Controlled Channel Decoding: Estimation of Correlated Parameters | 7 |
| 3.2 Combined Source/Channel (De-) Coding: Can A Priori Information Be Used Twice? | 10 |

| | |
|--|-----------|
| CHAPTER 4 BACKGROUND | 13 |
| 4.1 History of Turbo Code | 13 |
| 4.2 Architecture of Turbo Code | 14 |
| 4.3 Block Diagram of Turbo Codec | 14 |
| 4.4 Turbo Encoder Structures | 16 |
| 4.4.1 Recursive Systematic Convolutional (RSC) | 16 |
| Encoder | |
| 4.4.2 Interleaver | 17 |
| 4.4.3 Turbo Encoder | 17 |
| 4.5 Turbo Decoder Structures | 20 |
| 4.6 Synchronization of Turbo Code | 21 |
| CHAPTER 5 Implementation Method | 23 |
| 5.1 Turbo Encoder Implementation | 23 |
| 5.2 Turbo Decoder Implementation | 24 |
| CHAPTER 6 Priori Pattern Evaluation | 25 |
| 6.1 Simulation of a priori bit vs. Error bits Matches | 25 |
| 6.2 Distribution of a Priori Bits | 25 |
| 6.2.1 Equal Distance of a Priori Bits | 25 |
| 6.2.2 Gaussian Distribution of a Priori Bits | 26 |
| 6.2.3 Uniformly Distributed a Priori Bits | 28 |
| 6.3 Generation of Random Bits | 30 |
| 6.4 A Simulation of a Priori Bit Distributions | 31 |
| Simulation I: A Priori Bit vs. Random Single Error Bits | 33 |
| Simulation II: A Priori Bit vs. Random Multiple Error Bits | 35 |
| Simulation III: A priori Bit vs. Random Single Burst Bits | 37 |

| | |
|---|-----------|
| 6.5 Conclusion | 40 |
| CHAPTER 7 Implemented Turbo Codec Simulation | 41 |
| 7.1 Simulation Setup | 41 |
| 7.1.1 A priori Bits Generation | 41 |
| 7.1.2 A priori Bits Insertion | 42 |
| 7.2 Synoptic scheme of the software implementation | 45 |
| 7.3 Results Analysis | 47 |
| Simulation I: Performance Comparison Between Different Block Sizes without a Priori Bits at the Decoding Process | 48 |
| Simulation II: Performance Comparison Between Different a Priori Bits Distributions | 50 |
| CHAPTER 8 CONCLUSION | 55 |
| REFERENCES | 57 |
| APPENDIX A: BCJR ALGORITHM | 61 |
| APPENDIX B: DIFFERENTIATION ENTROPY OF UNIFORM DISTRIBUTION | 64 |
| APPENDIX C: CODE USED TO SIMULATE THE TURBO CODEC | 66 |

LIST OF FIGURES

| | Page |
|------------|--|
| Figure 3.1 | Transmission System 7 |
| Figure 3.2 | The bit error rate of the 1st bit of a parameter quantized with 3 bits and natural binary mapping 8 |
| Figure 3.3 | Bit error rate after channel decoding using a priori information (AWGN-channel at $1 = -3$ dB, folded binary mapping) 9 |
| Figure 3.4 | Transmission System 10 |
| Figure 3.5 | Bit error rate for natural binary mapping after channel decoding using a priori information (AWGN channel at -3 dB) 11 |
| Figure 3.6 | Parameter SNR for natural binary bit-mapping with and without a priori information 11 |
| Figure 4.1 | Turbo Code Communication System Block Diagram 15 |
| Figure 4.2 | Conventional convolutional encoder with $r=1/2$ and $K=3$ 16 |
| Figure 4.3 | The RSC encoder obtained from Figure 3.2 with $r=1/2$ and $K=3$ 17 |
| Figure 4.4 | Generic rate $1/3$ turbo encoder 18 |
| Figure 4.5 | A half rate turbo code 19 |
| Figure 4.6 | Low rate turbo encoder 19 |
| Figure 4.7 | Generic turbo decoder 20 |
| Figure 4.8 | Turbo Code Structure in the Information Channel 22 |

| | | |
|------------|--|-------|
| Figure 5.1 | Rate 1/3 turbo encoder with a priori bits | 22 |
| Figure 5.2 | Turbo decoder with a priori bits | 23 |
| Figure 6.1 | Equal Distance Distribution (Block_number=1,000) | 25 |
| Figure 6.2 | Gaussian Distribution (Block_number=1,000) | 27 |
| Figure 6.3 | Uniform Distribution (Block_number=1,000) | 28 |
| Figure 6.4 | A priori Bits Distributions | 31 |
| Figure 6.5 | A Priori Bit vs. Random Single Error Bits | 32-33 |
| | Figure 6.5 (a): 10 Bits | 32 |
| | Figure 6.5 (b): 20 Bits | 32 |
| | Figure 6.5 (c): 30 Bits | 32 |
| | Figure 6.5 (d): 40 Bits | 32 |
| | Figure 6.5 (e): 80 Bits | 33 |
| | Figure 6.5 (f): 100 Bits | 33 |
| Figure 6.6 | A Priori Bit vs. Random Multiple Error Bits | 34 |
| | Figure 6.6 (a): 10 Bits | 34 |
| | Figure 6.6 (b): 100 Bits | 34 |
| Figure 6.7 | A Priori Bit vs. Random Multiple Error Bits in term of Percentage of Error | 35 |
| | Figure 6.7 (a): Error 10 % | 35 |
| | Figure 6.7 (b): Error 80 % | 35 |
| Figure 6.8 | A priori Bit vs. Random Single Burst Bits | 36 |
| | Figure 6.8 (a): 10 bits | 36 |
| | Figure 6.8 (b): 100 bits | 36 |
| Figure 6.9 | A priori Bit vs. Random Single Group of Block Size = 10 Bits | 37 |

| | | |
|-------------|--|----|
| | Figure 6.9 (a): Group of 1, 10, and 20 bits | 37 |
| | Figure 6.9 (b): Group of 30 and 40 bits | 37 |
| | Figure 6.9 (c): Group of 50 to 100 bits | 37 |
| Figure 6.10 | A priori Bit vs. Random Single Group of Block Size=100 bits | 38 |
| | Figure 6.10 (a): Group of 1 and 10 bits | 38 |
| | Figure 6.10 (b): Group of 20 to 40 bits | 38 |
| | Figure 6.10 (c): Group of 50 to 100 bits | 38 |
| Figure 7.1 | Histogram of Equal Distribution a priori Bits | 40 |
| Figure 7.2 | Histogram of Gaussian Distribution a priori Bits | 41 |
| Figure 7.3 | Histogram of Uniform Distribution a priori Bits | 41 |
| Figure 7.4 | A priori Bit Insertion at the Encoding Process | 42 |
| Figure 7.5 | A priori Bit Insertion at the Decoding Process | 43 |
| Figure 7.6 | Software synoptic | 44 |
| Figure 7.7 | BER without a priori bits | 48 |
| Figure 7.8 | FER without a priori bits | 48 |
| Figure 7.9 | BER at Block Size=128 bits | 50 |
| Figure 7.10 | BER at Block Size=256 bits | 50 |
| Figure 7.11 | BER at Block Size=512 bits | 51 |
| Figure 7.12 | BER at Block Size=1024 bits | 51 |
| Figure 7.13 | FER at Block Size=128 bits | 52 |
| Figure 7.14 | FER at Block Size=256 bits | 52 |
| Figure 7.15 | FER at Block Size=512 bits | 53 |
| Figure 7.16 | FER at Block Size=1024 bits | 53 |

LIST OF TABLES

| | | Page |
|-----------|---|------|
| Table 3.1 | Bit redundancy, mutual information and mean square error for different bit-mappings of 3 bit quantized parameters | 8 |



CHAPTER 1 INTRODUCTION

1.1 OVERVIEW

Deep space communications are far more challenging than other kind of communications like mobile phone, Earth-orbiting satellites, because it involves radio propagation over very long distances. Signals must travel over millions or billions of kilometers between Earth station and the spacecraft. In the radio channel, it results noisy that is described as Additive White Gaussian Noise (AWGN), burst of errors produced by atmospheric, multi-path fading, interferences from other users of the frequency band. Therefore, to attain reliable communication in deep space, the use of very powerful error control technique is required.

The radio signals for deep space communications consists of carrying instructions to the spacecraft and data between Earth station and the spacecraft. The data needs to be delivered to a distant spacecraft safely and successfully, because if data is incorrectly interpreted, it would potentially cause spacecraft to take undesirable actions, including some that could result in a critical situation for the spacecraft. To prevent against that kind of possibility, the signal is coded with additional redundant data that allows the system to detect or correct any corruption in that signal.

Real communication channels contain a mixture of independent and burst error statistics and those channels are described as compound-error channels, and Automatic – Repeat - Request (ARQ) schemes are often used in packet data communication system. It has been most widely used as an effective method for error protection over compound-error channels for error control in data communication systems. ARQ system utilizes redundancy for error detection by using a start and stop strategy where the transmitter stops and waits until it receives acknowledgement of correct reception of code word or request for retransmission, which necessitates the use of a return path feedback channel path. However,

ARQ scheme only includes error detection strategy rather than both error detection and correction, and also requires the feed back channel to be practically noiseless. For long distance communications in deep space, transmitter can not wait for an acknowledgement from receiver to take another action, therefore ARQ system is not suitable for deep space communications.

The best codes to be used for reliable data transfer have been studied by many researchers and is still making progress.

At the time of Voyager which was launched in 1977, consists of a short convolutional code that is combined with a large block-size Reed-Solomon code with $k=7$ was chosen as a compromise between performance and decoding complexity were used. Evolution of the use of codes and decoding equipment has been paced by the evolution of digital processing capability.

Convolutional codes are one of the most widely used channel codes in practical communication systems. These codes are developed with a separate strong mathematical structure and are primarily used for real time error correction. Convolutional codes convert the entire data stream into one single codeword. The encoded bits depend not only on the current k input bits but also on past input bits. The main decoding strategy for convolutional codes is based on the widely used Viterbi algorithm. As a result of the wide acceptance of convolutional codes, there have been many advances to extend and improve this basic coding scheme. This advancement resulted in two new coding schemes, namely, trellis coded modulation (TCM) and turbo codes. TCM adds redundancy by combining coding and modulation into a single operation. The unique advantage of TCM is that there is no reduction in data rate or expansion in bandwidth as required by most of the other coding schemes.

Turbo code was focused in the late 90s as a new and very powerful error control technique. This error correcting code is able to transmit information across the channel with arbitrary low bit error rate. This code is a parallel concatenation of two component convolutional codes separated by a random interleaver. Random coding of long block lengths may also perform close to channel capacity, but this code is very hard to decode due to the lack of code structure. The performance of a turbo code is partly due to the random interleaver used to give the turbo code a random appearance. The big advantage of a turbo code is that there is enough code structure from the convolutional codes to decode it efficiently.

There are two primary decoding strategies for turbo codes. They are based on a maximum a posteriori (MAP) algorithm and a soft output Viterbi algorithm (SOVA). Regardless of which algorithm is implemented, the turbo code decoder requires the use of two component decoders that operate in an iterative manner. For further improvement of turbo code, researchers are focusing the issues of improving decoder performance and reducing the decoder complexity.

To improve the decoder performance, channel decoders usually aim at minimizing the frame, symbol or residual bit error rate of the source bits. Due to the fact, that coding this usually done by frame and also in a time correlation of successive frames. The redundancy should be used as a priori information at the receiver to improve the decoding result.

In this thesis, turbo code was used to determine if the residual bit error rate can be reduced by using a priori knowledge about the source statistics within the channel decoding, and if it helps the source decoder to perform better process by measuring improvement of Bit Error Rate (BER) and Frame Error Rate (FER).

1.2 IMPORTANCE OF THE TURBO CODE IN DEEP SPACE COMMUNICATIONS

In deep-space communications, turbo code is the most suitable coding technique, because using very large interleavers can maximize the turbo coding gain, and compare to short distance communications like mobile phone, the delay is not a big issue in deep space communications. Also from the view of implementation, use of more than two elementary encoders can be combined in many efficient ways to create a very powerful low rate turbo code at a slight increase in complexity.

1.3 USE OF A PRIORI BITS

The coded data stream of today's powerful source codecs still contains residual redundancy. Hence many techniques have been developed, which use the properties of the respective source signal to reduce its redundancy and irrelevancy. Source coding is usually done frame by frame, but due to requirements of complexity and time delay, source encoders usually work only suboptimal. Consequently, the compressed data stream still contains residual redundancy, which remains both inside a frame and also in a time correlation of successive frames. The redundancy should be used by the source and/or channel decoder as a priori information to improve the decoding result. These techniques are known as robust source coding or source controlled channel decoding.

1.4 PROPOSED FRAMEWORK.

This part describes the required processes for implementing existing error control coding techniques for deep space communication.

The first section introduces the approaching procedure of using a priori bit inside turbo codec.

The second section introduces the statistical approach to calculate the possibility of matches between random noise and a priori bit with different combinations of size and pattern (Equal distance distribution, uniform distribution and Gaussian distribution). The conclusion with the criteria is used to identify the best pattern of a priori bit used at the turbo decoder to improve its performance.

The last section presents the simulation software specification and the result of the simulation. To evaluate the performance of the proposed implementation, the simulation software is used to evaluate the performance in the following criteria.

- The throughputs of error correction in BER and FER with and without a priori information
- The throughputs where the size and pattern of a priori bit is concerned

Finally, the result of the simulation is analyzed in terms of the bit error rate and the frame error rate.

The conclusion chapter concludes the advantages and the drawbacks of implementation of a turbo code with a priori information.

CHAPTER 2: OBJECTIVES

The objectives of the research are to implement and study the performance of the existing turbo coding techniques for deep space communications. The main subject of the research focuses on methods combining the advantages of turbo code with the inclusion of a priori information in the binary stream.

A statistical simulation software is used to evaluate the effect of the turbo code implementation by using a priori knowledge about the source during the channel decoding, and as it will be shown, the source decoder performs better by observing an improvement of the Bit Error Rate (BER) and the Frame Error Rate (FER).



CHAPTER 3: LITERATURE REVIEW

3.1 Source-Controlled Channel Decoding: Estimation of Correlated Parameters [2]

In this paper, an approach to improve channel and source decoding by using the redundancy remains inside one block frame as in time correlation of subsequent frames is described. Also described is how the end-to-end quality of parameters can be improved by choosing the bit mapping and the protection of bits in channel coding.

The idea was born in a discussion of 2 universities' common project "Source controlled channel decoding" (SCCD) and "Soft bit source decoding"(SBSD): Is it better to exploit the a priori information in channel decoding or in source decoding or is it possible to use it twice?

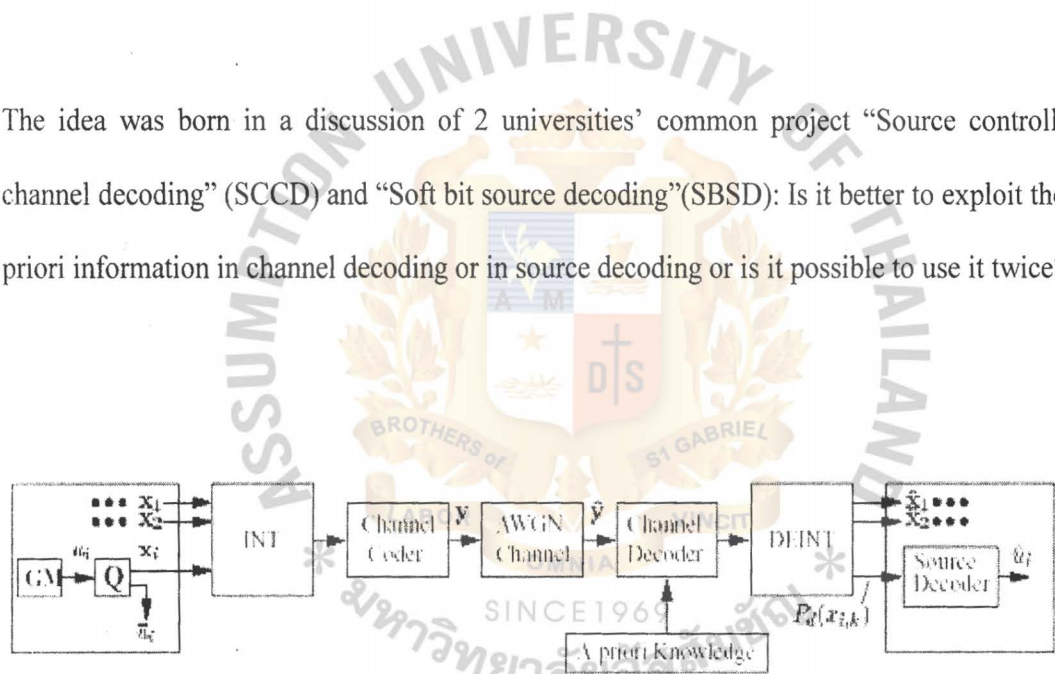


Figure 3.1: Transmission System

In this paper, an effective bit rate of 4, 6 or 8 bit per parameter and an overall blocklength of 120 bit is considered. This leads to 30, 20 or 15 parameters respectively within one block. Furthermore, not only the BER after channel decoding but also the parameter SNR as a quality measurement after source decoding is investigated.

TABLE 3.1: Bit redundancy, mutual information and mean square error for different bit-mappings of 3 bit quantized parameters.

| Bit-mapping | | $1 - H(X)$ | $I(X; \mathbf{X}_{\setminus k})$ | $I(X; \mathbf{X}_{-1})$ | MSE |
|------------------|-------|------------|----------------------------------|-------------------------|-------|
| natural bin. | x_0 | 0.0 | 0.170 | 0.313 | 5.029 |
| 000,001,010,011, | x_1 | 0.0 | 0.146 | 0.029 | 1.506 |
| 100,101,110,111 | x_2 | 0.0 | 0.049 | 0.0 | 0.377 |
| folded bin. | x_0 | 0.0 | 0.0 | 0.313 | 3.871 |
| 011,010,001,000, | x_1 | 0.125 | 0.021 | 0.127 | 1.506 |
| 100,101,110,111 | x_2 | 0.028 | 0.021 | 0.012 | 0.377 |
| Gray code | x_0 | 0.0 | 0.0 | 0.313 | 3.871 |
| 000,001,011,010, | x_1 | 0.125 | 0.045 | 0.127 | 1.876 |
| 110,111,101,100 | x_2 | 0.003 | 0.045 | 0.015 | 0.377 |
| max. dist. | x_0 | 0.038 | 0.142 | 0.012 | 5.029 |
| 110,000,111,001, | x_1 | 0.0 | 0.146 | 0.0 | 4.915 |
| 010,100,011,101 | x_2 | 0.0 | 0.146 | 0.029 | 1.506 |
| low change | x_0 | 0.0 | 0.111 | 0.313 | 5.047 |
| 101,100,110,111, | x_1 | 0.021 | 0.089 | 0.066 | 1.124 |
| 011,001,000,010, | x_2 | 0.021 | 0.089 | 0.066 | 1.124 |

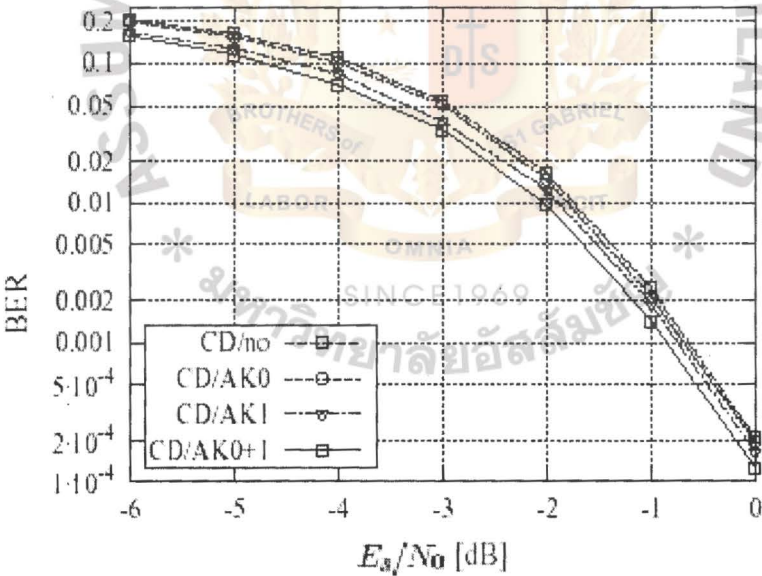


Figure 3.2: The bit error rate of the 1st bit of a parameter quantized with 3 bits and natural binary mapping.

Figure. 3.2 shows that the bit error rate decreases depending on how a priori information is used. A comparison to Table 3.1 shows that the bit redundancy and the two mutual

informations are a good measurement for the gain in bit error rate by using a priori information.

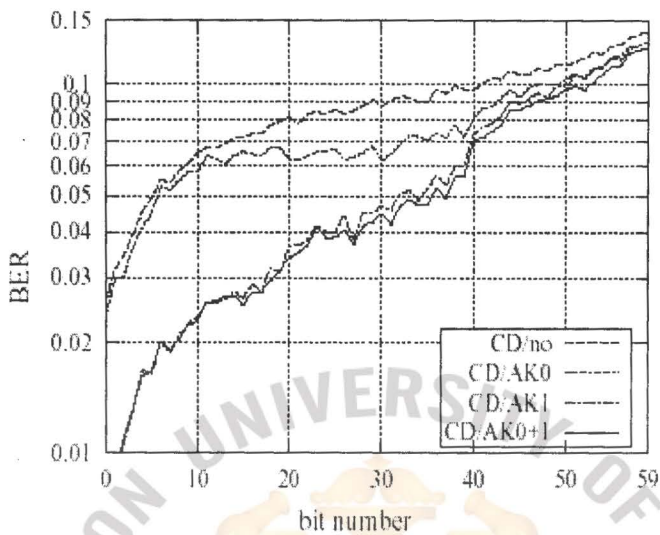


Figure 3.3: Bit error rate after channel decoding using a priori information (AWGN-channel at 1 = -3 dB, folded binary mapping).

In Fig. 3.3 the bit error rate is shown depending on the bit-position in the trellis. The interleaver works in that way that the first bit of each parameter index x_i is ordered in the first part of the trellis, the second bits are in the second part and the least significant bits are at the end of the frame. The unequal error protection here is achieved just by not terminating the convolutional code (see CD/no in Figure 3.5).

This paper examined the bit-mapping of quantized parameter indices and have shown the influence on channel and source decoding. Dependent, on which criteria a system is optimized – either BER or parameter SNR or others – different bit-mappings could deliver the best results.

Due to the properties of convolutional codes, the channel decoding also improves uncorrelated parameters if its bits are ordered close to the correlated ones within a block.

3.2 Combined Source/Channel (De-) Coding: Can A Priori Information Be Used Twice? [3]

This paper works on combined source and channel decoding, trying to answer the following question: Can a priori information that models the source parameters be used twice? : first at the channel decoder and then at the source decoder.



Figure 3.4: Transmission System

The figure 3.4 shows the transmission system. The channel decoder uses the a priori information that models the bit stream generated by the source coder. This does not capture all the details of the source parameter level statistics. By exploiting the a priori knowledge of parameters (once more) at the source decoder, it shows that it is possible to achieve better reconstruction than if this information was used at either of the decoders.

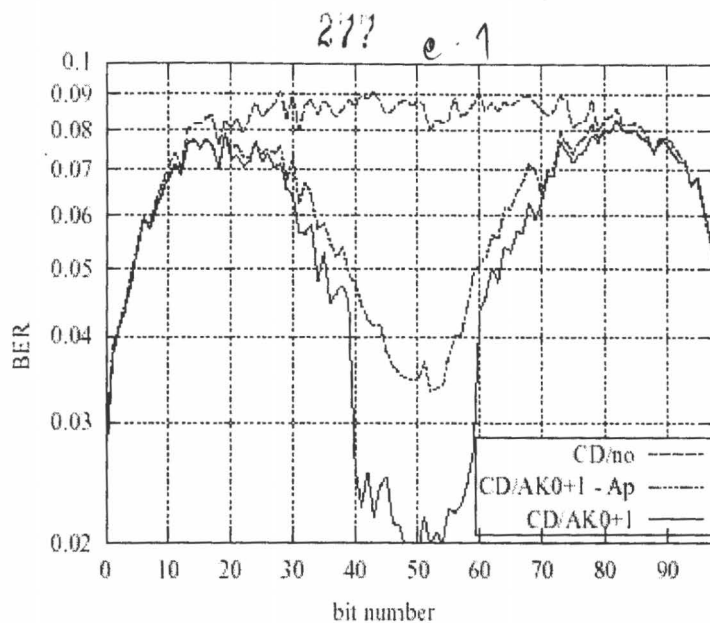


Figure 3.5: Bit error rate for natural binary mapping after channel decoding using a priori information (AWGN channel at -3 dB).

Figure 3.5 shows that the gain in bit error rates for decoding with (CD/AK0+1) is very high, especially for the MSBs (pos. 40..59). After “subtracting” the a priori knowledge of the parameter itself (CD/AK0+1 2 Ap) the BER increases compared to (CD/AK0+1), but there is still a big gain in BER.

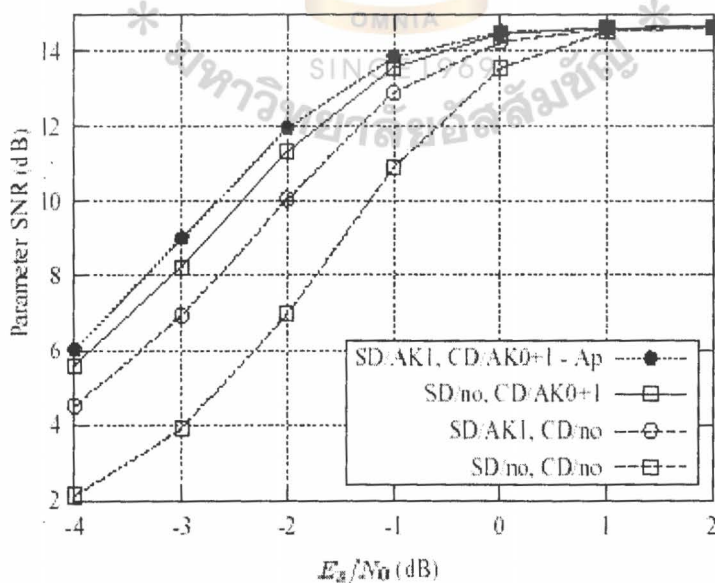


Figure 3.6: Parameter SNR for natural binary bit-mapping with and without a priori information.

Figure 3.6 shows that the curve with (SD/no, CD/AK0+1) now delivers a better quality. That means, if the information is used only once, it should be used in the channel coder. After “subtracting”, we get another gain of up to 1 dB in the parameter SNR which gives the answer to the question “Can A Priori Information be used twice?”. This gain can also be transmitted to systems where some parameters are redundant and others are not. It can be seen in Figure 3.5, where the dummy bits gain also (pos. 0..19, 80..98).

The above result shows that a priori information can be used twice if it is subtracted after channel decoding and not only one correlated parameter is considered. That means, for the source decoder, it does not see any a priori information of one parameter itself in the reliability information delivered by the channel decoder. The results can be extended to a complete speech transmission system, where some parameters are highly correlated and some are not. The bit-mapping is quite important in determining what kind of a priori information delivers the highest gain in channel decoding.

Also the mean square error, one possible optimization criterion, changes with the bit-mapping. Therefore, the bit-mapping has to be chosen dependent on the optimization criterion.

CHAPTER 4: BACKGROUND

4.1 HISTORY OF TURBO CODES

Turbo codes have been known since 1993 as a new and very powerful error control technique after it is introduced by C. Berrou, A. Glavieux, and P. Thitimajshima, in “Near Shannon limit error correcting coding and decoding: Turbo-Codes”[13]. This error correcting code is able to transmit information across the channel with arbitrary low bit error rate. The basic principle of the turbo code concept is that message bit is encoded in two different ways by two encoders. The decoder is correspondingly divided into two separate decoders, where each decoder decodes its concatenated codeword. By using the sophisticated algorithms, the decoders can exchange information on their decoding results and find the correct codeword. This is one of the key ideas that allow a continuous improvement in correction capability when the decoding process is iterated. In the traditional approach, the demodulator block makes a hard decision of the received symbol and passes it to the error control decoder block. This is equivalent to deciding which of two logical values 0 and 1 was transmitted. No information was passed about how reliable the hard decision was. This led to the development of “soft” input decoding algorithms. This was the best solution if only one code was used. For the same reason, in the case of combining more codes as explained above, new Soft In/Soft Out(SISO) algorithms were developed in order to pass more information from the output of one decoder to the input of the next decoder. Soft output decision algorithms provide as an output a real number which is a measure of the probability of error in decoding a particular bit. This can also be interpreted as a measure of the reliability of the decoder’s hard decision. This extra information is very important for the next stage in an iterative decoding process. The name of turbo code reflects this iterative decoding process. There are two important categories of soft output decision algorithms. The first category includes the

maximum likelihood decoding algorithms, which minimize the probability of symbol error, such as the maximum a posteriori (MAP) algorithm. The second category includes the maximum likelihood decoding algorithms, which minimize the probability of word or sequence error, such as the Viterbi algorithm or soft output Viterbi algorithm (SOVA). The performance of a turbo code is partly due to the random interleaver used to give the turbo code a random appearance. The big advantage of a turbo code is that there is enough code structure from the convolutional codes to decode it efficiently. Regardless of which algorithm is implemented, the turbo code decoder requires the use of two component decoders that operate in an iterative manner. For further improvement of turbo code, researchers are focusing the issues of improving decoder performance and reducing the decoder complexity.

4.2 Architecture of Turbo Codes

Turbo codes use the parallel concatenated encoding scheme. However, the turbo code decoder is based on the serial concatenated decoding scheme. The serial concatenated decoders are used because they perform better than the parallel concatenated decoding scheme due to the fact that the serial concatenation scheme has the ability to share information between the concatenated decoders whereas the decoders for the parallel concatenation scheme are primarily decoding independently.

4.3 Block Diagram of the Turbo Codec

The turbo codec architecture for a parallel concatenated scheme is shown in Figure 4.1. The incoming message bits, **m0**, are encoded by a Rate 1/3 systematic turbo encoder. The outputs from the turbo encoder are the systematic message bits **m0**, the parity bits **p1** and the parity bits **p2**. The puncturing block determines the actual coding rate. The puncturing function is a simple deletion of some parity bits from the **p1** and the **p2** streams. No puncturing is applied

to the message bits m_0 . The m_0 and the punctured parity bits p_0 signals are then modulated and sent to the channel.

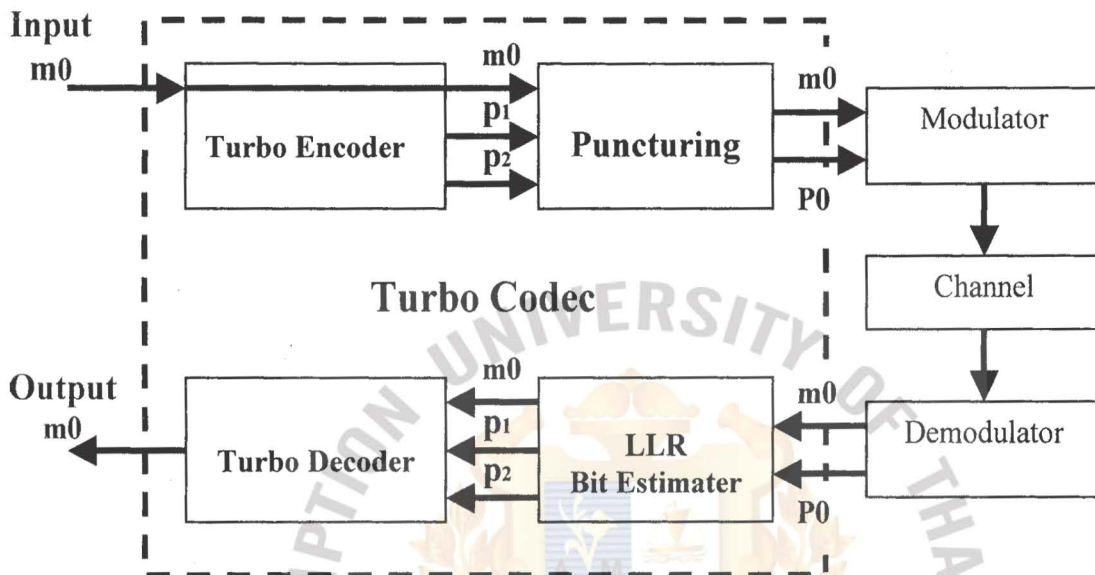


Figure 4.1: Turbo Code Communication System Block Diagram


At the receiver, after soft decision demodulation, the received m_0 and p_0 signals are input to the bit estimator block. The function of this block is to compute the log-likelihood ratios (LLR) for each bit received. The outputs from this block are punctured streams of the estimated m_0 , p_1 and p_2 bits. The turbo decoder does not need to know anything about what mapping or modulation scheme was transmitted. The same decoding engine is used to produce the decoded bits m_0 regardless of the modulation or puncturing scheme used.

4.4 Turbo Encoder Structures

The fundamental turbo code encoder is built using two identical recursive systematic convolutional (RSC) codes with parallel concatenation. An RSC encoder is typically $r = 1/2$ shown in Figure 4.3.

4.4.1 Recursive Systematic Convolutional (RSC) Encoder

The recursive systematic convolutional (RSC) encoder is obtained from the nonrecursive nonsystematic (conventional) convolutional encoder by feeding back one of its encoded outputs to its input. Figure 4.2 shows a conventional convolutional encoder. The symbol

 Represents a D flip-flop and the symbol \oplus represents an exclusive-OR gate.

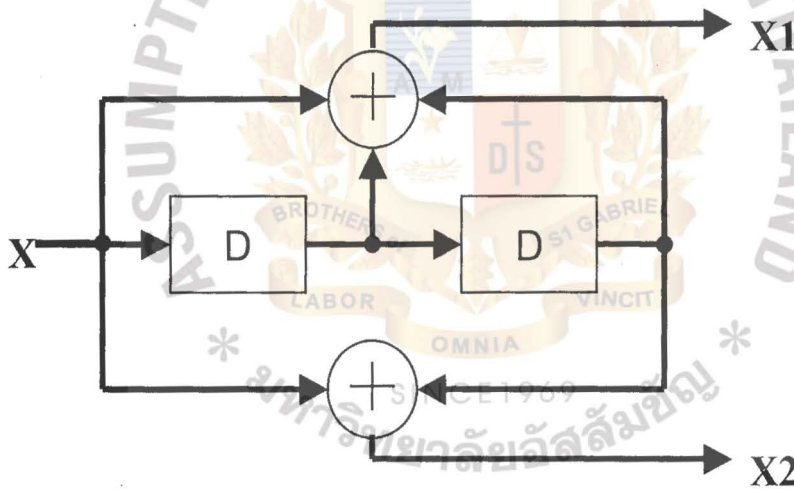


Figure 4.2: Conventional convolutional encoder with $r=1/2$ and $K=3$.

The conventional convolutional encoder is represented by the generator sequences $\mathbf{g}_1=[111]$ and $\mathbf{g}_2=[101]$ and can be equivalently represented in a more compact form as $\mathbf{G}=[\mathbf{g}_1, \mathbf{g}_2]$. The RSC encoder of this conventional convolutional encoder shown in Figure 4.3 is represented as $\mathbf{G}=[1, \mathbf{g}_2 / \mathbf{g}_1]$ where the first output (represented by \mathbf{g}_1) is fed back to the input. In the above representation, 1 denotes the systematic output, \mathbf{g}_2 denotes the feedforward output, and

g_1 is the feedback to the input of the RSC encoder. Figure 4.3 shows the resulting RSC encoder.

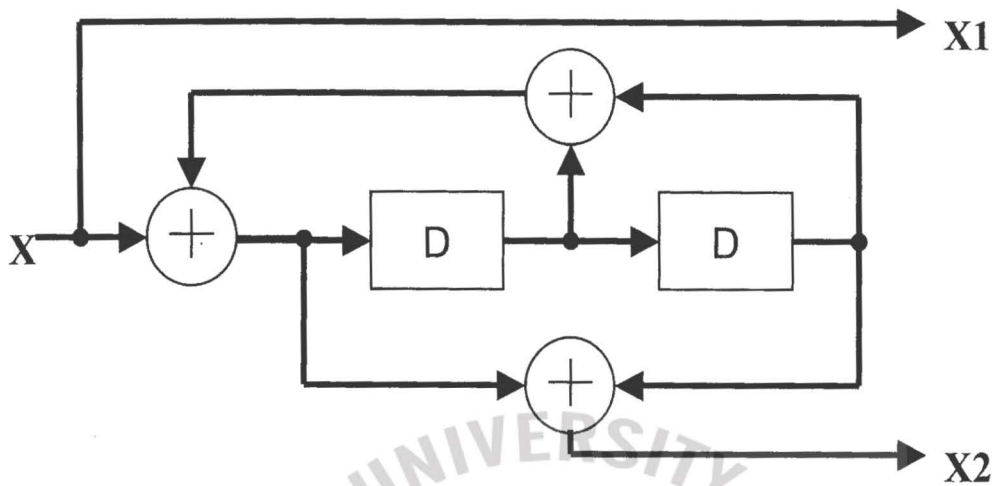


Figure 4.3: The RSC encoder obtained from Figure 3.2 with $r=1/2$ and $K=3$.

4.4.2 Interleaver

An interleaver is used between the two component encoders. The interleaver is used to provide randomness to the input sequences. The interleaver affects the performance of turbo codes because it directly affects the distance properties of the code. The interleaver design is a key factor which determines the good performance of a turbo code.

4.4.3 Turbo Encoder

Figure 4.4 shows a generic turbo encoder. The input sequence of the message bits is organised in blocks of length N . The first block of data will be encoded by the ENC1 block which is a rate half recursive systematic encoder. The same block of message bits is interleaved by the interleaver INT, and encoded by ENC2 which is also a rate half systematic recursive encoder.

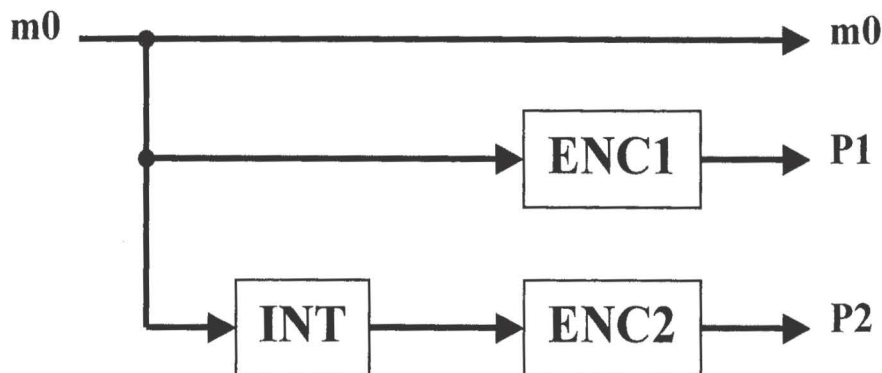


Figure 4.4: Generic rate 1/3 turbo encoder.

The interleaver block, INT, arranges the order of the message bits for input to the second encoder. The main purpose of the interleaver is to increase the minimum distance of the turbo code such that after correction in one dimension the remaining errors should become correctable error patterns in the second dimension. The outputs of the turbo encoder are the message bits sequence **m0**, together with the corresponding parity sequence **p1** produced by one encoder block, say **ENC1**, and the parity sequence **p2** produced by the second encoder block, say **ENC2**. These sequences are modulated and sent through the channel. The interleaved data sequence is not sent because it can be regenerated at the receiver by interleaving the received sequence corresponding to **m0**. The parity bits **p1** and **p2** can be “punctured” as in Figure 4.5 where puncturing is implemented by a multiplexing switch in order to obtain higher coding rates. A rate 1/2 turbo code can be implemented by alternatively selecting the outputs of the two encoders.

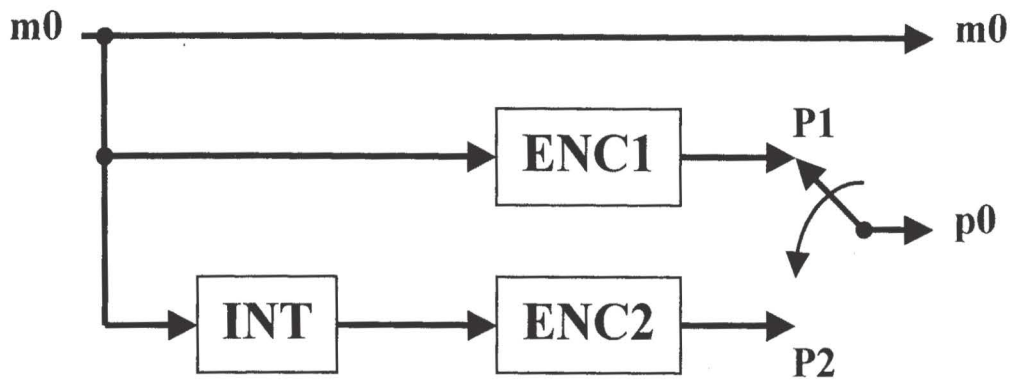


Figure 4.5: A half rate turbo code.

Figure 4.6 shows a particular implementation of turbo code using recursive systematic codes (RSC). Lower coding rates can be achieved using either less puncturing or more interleaver and encoder blocks as shown in Figure 4.6.

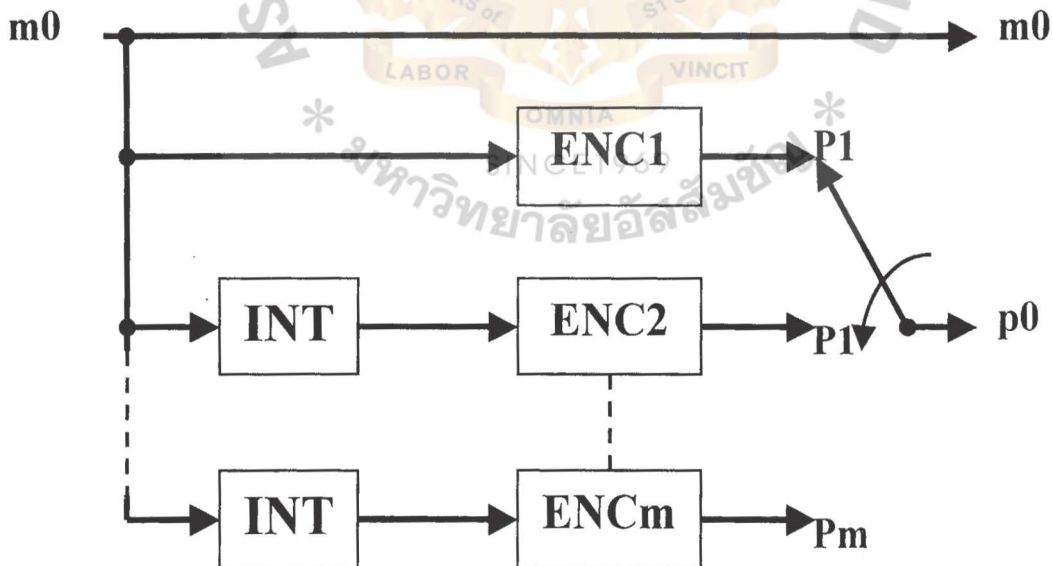


Figure 4.6: Low rate turbo encoder.

4.5 Turbo Decoder Structures

At the receiver, decoding is performed in an iterative process as shown in Figure 4.7.

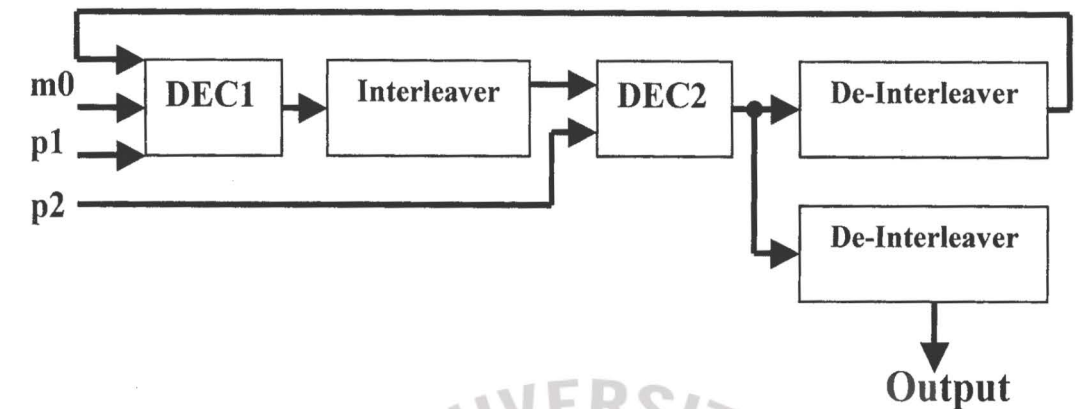


Figure 4.7: Generic turbo decoder.

Decoder **DEC1** provides a soft output which is a measure of the reliability of each decoded bit. From this reliability information, the extrinsic information is produced, which does not depend on the current inputs to the decoder. This extrinsic information, after interleaving, is passed on to **DEC2** which uses this information to decode the interleaved bit sequence. From the soft outputs of **DEC2**, the new extrinsic information is fed back to the input of **DEC1** and so on. If an error occurs at the output of the first decoder due to a very noisy input, it might be corrected by the second decoder. A soft decision decoder outputs a real number which is a measure of the probability of a correct decision. This real number is called the a posteriori probability (APP). There are two types of soft decision decoding algorithms which are typically used, the first being a modified Viterbi algorithm which produces soft outputs and hence is called a soft output Viterbi algorithm (SOVA). A second is the maximum a posteriori (MAP) algorithm. Estimates put the complexity of the MAP algorithm at two times that of the Viterbi algorithm. However, the MAP algorithm results in better performance at low SNR due to a more accurate evaluation of the APP. The performance of a turbo coding scheme improves as the number of decoder iterations is increased. However, the coding gain

from one iteration to another, decreases with the number of iterations. Each iteration involves two decoding stages. Therefore, the overall complexity of a turbo decoder depends on how efficient the decoding algorithm is implemented.

The concept of iterative decoding relies on the use of soft-input/soft-output decoders, which calculates a posteriori probabilities (APP) based on the received channel sequences and a priori information. There are optimal algorithms for computing APP, the BCJR-algorithm[14], also called the MAP-algorithm and soft-output versions of the Viterbi algorithm (SOVA). The MAP algorithm is the ultimate approach, which minimizes the probability of an error by maximizing the probability of a symbol occurrence, based on the received signal. The decoder processes information one block at a time, there is a decoding delay and this latency is further increased by the iteration time of the decoders and the increase of block size.

The algorithm used for the simulations shown in following chapter 7 is based on BCJR-algorithm, and a full derivation is given in Appendix A.

4.6 Synchronization of Turbo Code

In any digital wireless communication, good synchronization is essential and must be established between transmitter and receiver. The synchronization is accomplished by preceding each code block or transfer frame with a fixed-length attached synchronization marker. This known bit pattern can be recognized to determine the start of the code blocks or transfer frames. The turbo code block synchronization is necessary for proper decoding of turbo codes, because the decoding operation needs to know the code block boundaries before it can iterate between the unpermuted and permuted data domains. Unlike the frame synchronization for Reed-Solomon codes, which is performed after convolutional decoding, synchronization for turbo codes must be done in the channel-symbol domain. This requires a

rate-dependent attached synchronization marker and synchronization algorithms that operate at the symbol rate as opposed to the data rate. Operation of the synchronizer is anticipated to be similar to that of the RS frame synchronizer and involves recognition of the marker in the coded symbol stream and anticipation of a recurring marker at an interval corresponding to the length of the turbo code block, the trellis termination sequence, and the synchronization marker. The trellis termination sequence is filled in the symbol output with zeroes in anticipation of processing the next block of data during the encoding process and provides a known final state (zero) to the turbo decoder. Figure 4.8 shows the structure of the turbo code as it appears in the information channel.

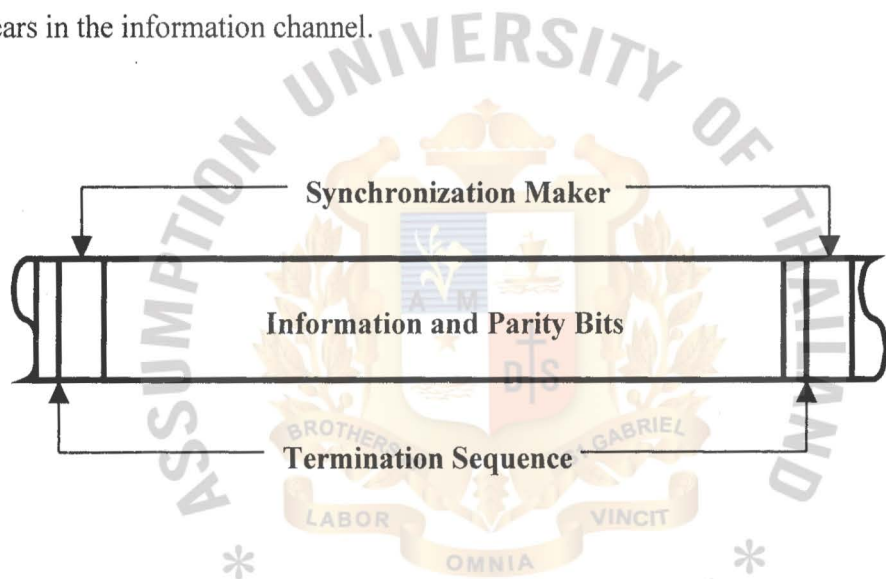


Figure 4.8: Turbo Code Structure in the Information Channel

CHAPTER 5: IMPLEMENTATION METHOD

The method of implementation is using a priori knowledge about the source statistics within the channel decoding. In source controlled channel coding schemes the decoder uses a priori information about the source bit probability in addition to the channel output information in order to achieve better performance. The distinction between a priori and a posteriori is that a priori is an independent of experience and a posteriori is based on experience. In this thesis Turbo Code was implemented to evaluate the bit error rate performance under these conditions using BCJR decoder.

5.1 Turbo Encoder Implementation

The implementation was done by adding same a priori bits at encoding process and decoding process as in figure 5.1 shown below.

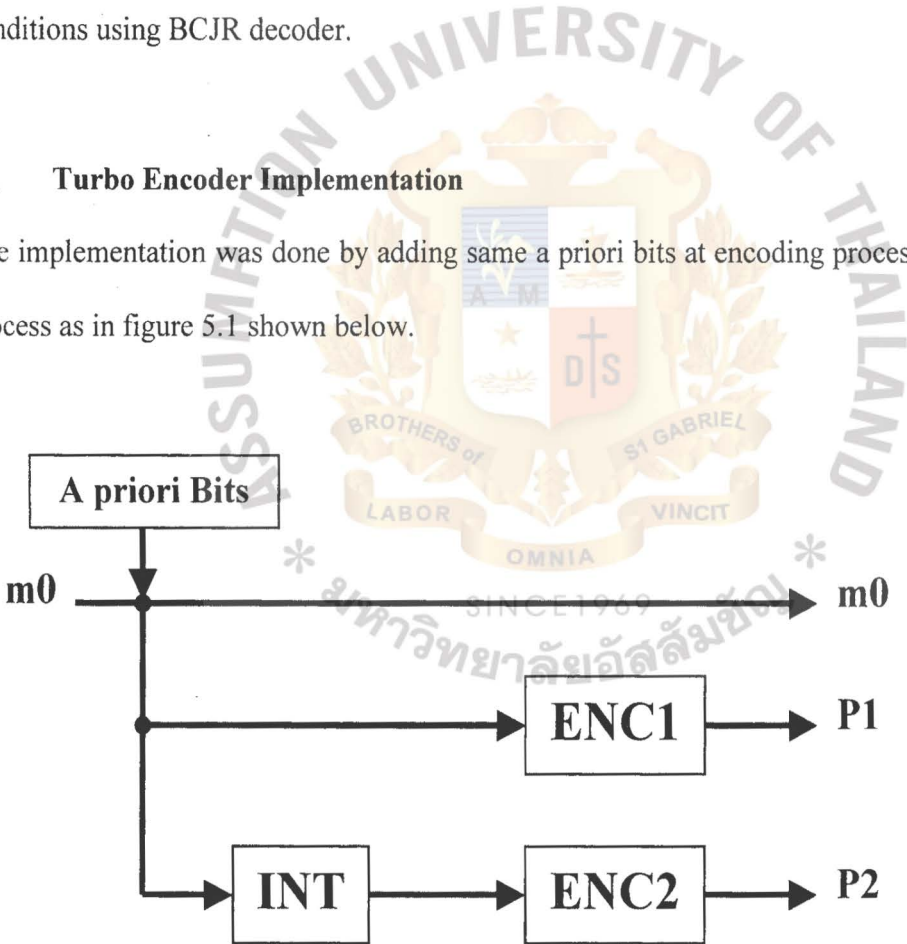


Figure 5.1: Rate 1/3 turbo encoder with a priori bits.

Figure 5.1 shows a priori bits added with different distribution pattern in the message bits before the process of interleave and encode.

5.2 Turbo Decoder Implementation

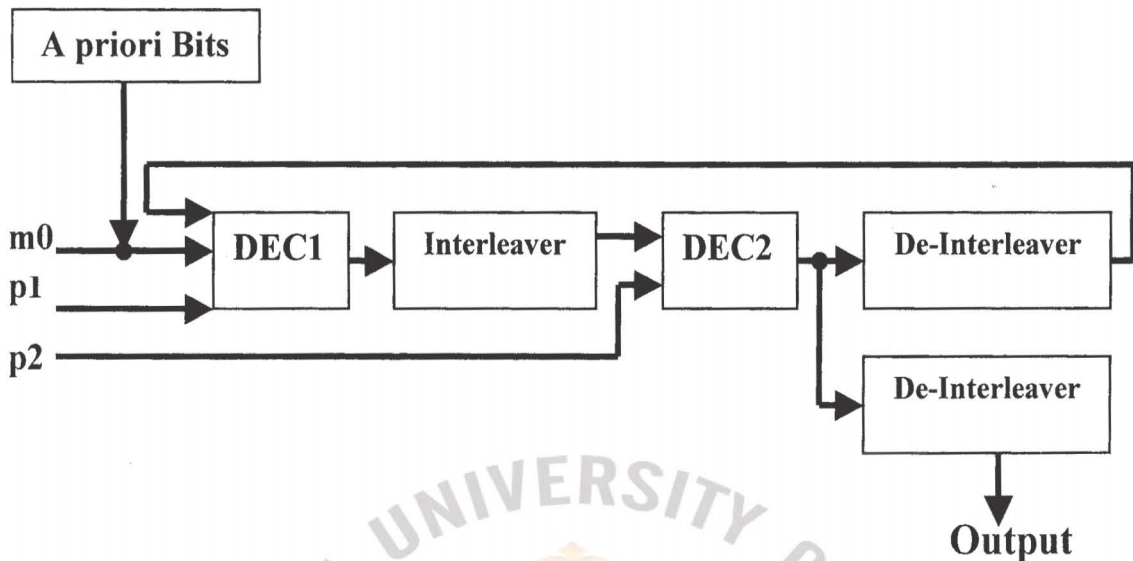


Figure 5.2: Turbo decoder with a priori bits.

Figure 5.2 shows a priori bits added in the received message bits before decoding process. This method adds the same a priori information at decoding process as in encoding process as it is assumed to improve the decoding performance as received noisy message bits are corrected by replacing received a priori bits with original a priori bits.

In Chapter 6, the relations between occurrences of a priori bit and different patterns of error bits by finding matches are studied. This simulation is performed to consider which type of a priori distribution can catch more error bits, and to apply its a priori distribution to the simulation in chapter 7 to determine if these different distribution patterns of a priori bits can effect the performance of turbo code.

CHAPTER 6: A PRIORI PATTERN EVALUATION

6.1 Simulation of a priori bit vs. Error bit Match Probability

This chapter introduces the simulations of the statistical approach to find the relations between occurrences of the different distribution types of a priori bit and a random error bit by finding match probability. This simulation is performed to consider the design of turbo decoder to estimate the error pattern when it performs the decoding to improve the error rate. There are three error bit patterns, single random error bit, multiple random error bits and single burst error bits compared to three different distribution of single a priori bit with several conditions finding average match probability.

6.2 Distribution of a Priori Bits

To catch the uncertain noise pattern, three kinds of distribution are used for a priori bits. These a priori bits are compared with three kinds of randomly generated error bits for number of block, as figures shown in following sections. In the histogram of each distribution, sum of a priori bits must be adjusted to have an exact number of blocks.

6.2.1 Equal Distance of a Priori Bits

This distribution generates a priori bits in every equal distance, middle of each section. The following part shows the procedure for making the histogram of equal distance distribution and Figures 5.1 shows an example of the histogram of equal distance a priori bits at section = 100 bits, block number = 1000.

```

void Equal_Distribution ( long int block_size, long int apriori, long int *block,
                        long int bit_number, long int block_number, long int *distribution)
{
    long int i;
    i=1;while (i<=block_size) {block[i]=0;i++;}
    i=1;while (i<=bit_number)
    {
        //Generate a priori bit for Equal distance distribution

        distribution[i]=block_number/block_size;
        i=i++;
    }
}

```

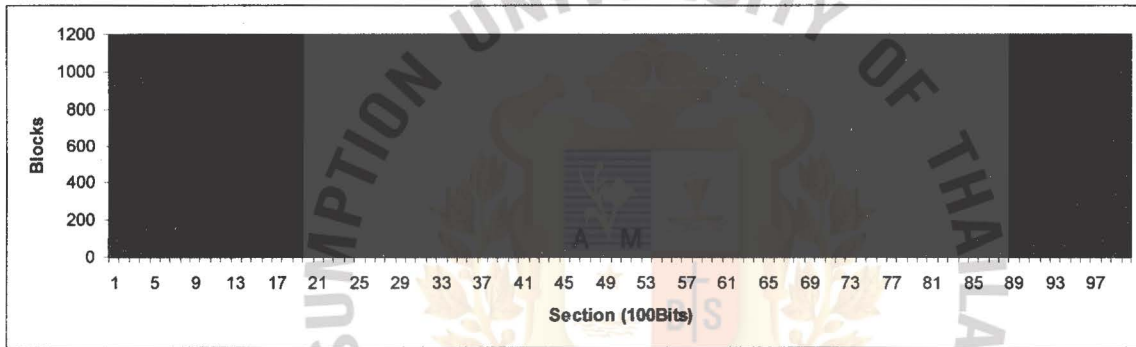


Figure 6.1: Equal Distance Distribution (Block_number=1,000)

6.2.2 Gaussian Distribution of a Priori Bits

To generate the Gaussian distribution a priori bits, following equations is used.

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

$$\int_{-\infty}^{\infty} P(x) dx = 1.$$

In this simulation, mean μ , where the peak of the density occurs, is set to the middle of block, and the standard deviation σ , which indicates the spread or girth of the bell curve, is used for the value of $\sigma=10,20,30,40,50,60,70,80,90$.

The following part shows the procedure for making the histogram of Gaussian distribution and Figures 5.2 shows an example of the histogram at section=100, $\mu=100/2=50$, $\sigma=10$. As equations shown above, sum of all values $P(x)$ generated becomes 1, however to simulate same number as block size specified, during the generation process, those generated values were multiplied until sum becomes equal or greater than block number. In the case of that sum becoming greater than block number, the histogram was adjusted to be same value as block number.

```
void gaussian_Distribution(long int priori_interval, long int block_number,
                          long int dispersion, long int mean, long int *distribution,
                          long int N, long int *block, long int sumdist, long int multiply,
                          long int priori_no)
{
    long int i,j,sum;
    multiply=1;
    sum = 0;

    // Make Gaussian Histogram which sum is same or equal to block number

    while(sum <= block_number)
    {
        sum = 0;
        i=1;while (i<=priori_interval) {distribution[i]=0;i++;}

        // Generate Gaussian Distribution Vaues

        i=1;while (i<=priori_interval)
        {
            distribution[i]=long int(0.9999+multiply*(((1.0/dispersion)*
            (sqrt(2*3.141592654))))*(exp((-i-mean)*
            (i-mean))/(2.0*dispersion*dispersion))));

            sum += distribution[i];
            i++;
        }
        multiply ++;
    }

    // Calculate the total area of histogram
    sumdist=0; i=1;while (i<=priori_interval)
    {
        sumdist=sumdist+distribution[i];
        i++;
    }
}
```



```

//Adjust sum of distribution = block_number
i=0;
while(sumdist - block_number>0)

{ if(distribution[i]>1)
  {
    distribution[i]--;
    sumdist--;
  }
  if(i == priori_interval)
    i = 0;
  else
    i++;
}
}

```

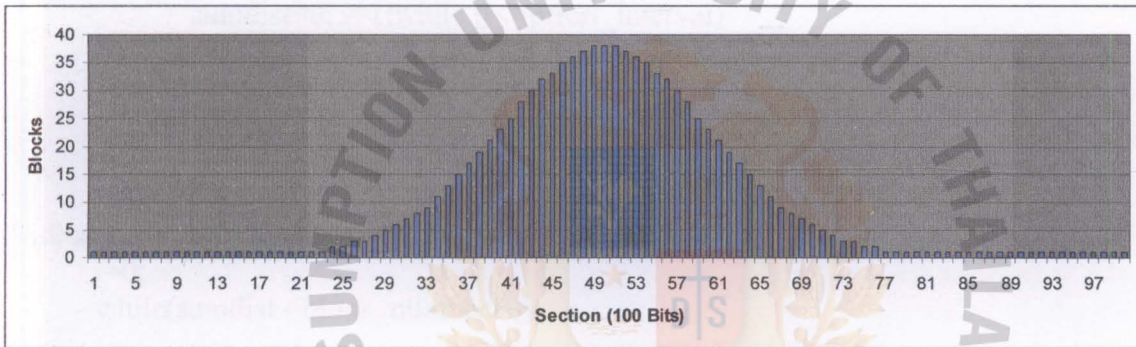


Figure 6.2: Gaussian Distribution (Block_number=1,000)

6.2.3 Uniformly Distributed a Priori Bits

This distribution has same value of a priori bits in each bit position making uniformly distributed a priori bits. The following part shows the procedure for making the histogram of uniform distribution and Figures 5.3 shows an example of the histogram of uniformly distributed a priori bits at section = 100 bits, block number = 1000.

```

void uniform_Distribution ( long int priori_interval, long int block_number,
                           long int dispersion, long int mean, long int *distribution,
                           long int N, long int *block, long int sumdist, long int multiply,
                           long int priori_no)
{

```



```
// Make Uniform Histogram which sum is same or equal to block number
```

```
long int i,j,sum;
multiply=1;
sum = 0;

while(sum < block_number)
{ sum = 0;
  i=1;while (i<=priori_interval) {distribution[i]=0;i++;}
  i=1;while (i<=priori_interval)
  { distribution[i]=multiply;
    sum += distribution[i];
    i++;
  }
}
```

```
// Calculate the total area of histogram
sumdist=0; i=1;while (i<=priori_interval)
```

```
{ sumdist=sumdist+distribution[i];
  i++;
}
```

```
//Adjust sum of distribution = block_number
```

```
i=0;
while(sumdist - block_number>0)
```

```
{ if(distribution[i]>1)
  { distribution[i]--;
    sumdist--;
  }
}
```

```
if(i == priori_interval)
```

```
i = 0;
```

```
else
```

```
i++;
```

```
}
```

```
}
```

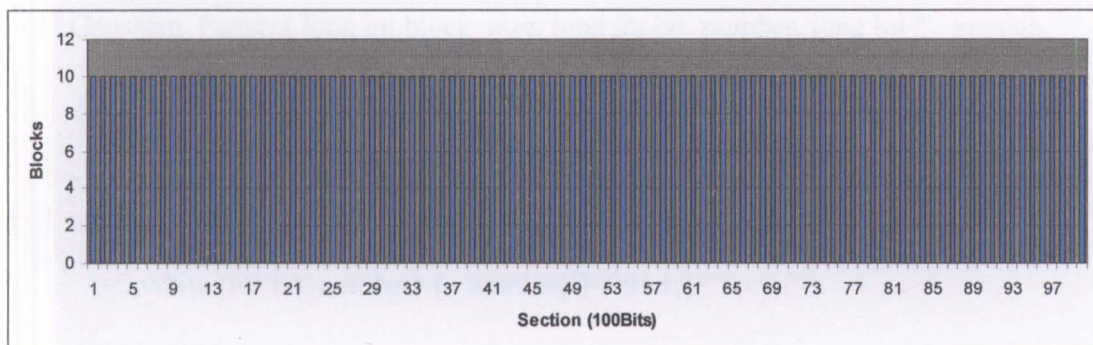


Figure 6.3: Uniform Distribution (Block_number=1,000)

6.3 Generation of Random Bits

During all the simulations, following random bits generation procedure were used. This generator was used for both error bits and a priori bits. The number of a priori bits in a block can be changed by random_no.

```
void Random_distribution(long int block_size,long int random_no,long int *random)
{
    long int I,j;
    i=1;while (i<=block_size) {random[i]=0;i++;}

    // number of random bits in block can be changed by random_no
    i=1;while (i<=random_no)
    {
        j=(rand( )%(block_size))+1;
        if (j==0) {j=1;}
        if (j>block_size) {j=block_size;}
        while (apriori[j]!=0)
        {
            j=(rand( )%(block_size))+1;
            if (j==0) {j=1;}
            if (j>block_size) {j=block_size;}
        }
        random[j]=1;
        i++;
    }
    i=0;
}
```

For example, for Gaussian distribution, after making randomly generated distribution, the Gaussian histogram is deducted at those randomly generated bits as procedure follows.

```
Void Gaussian_Pattern( long int block_size, long int bit_number, long int * aussian,
    long int *block, long int block_number, long int dispersion,
    long int mean, long int *distribution)
{
    //randomly generate a priori bit in each distribution
    long int I,j;

    i=0;while (i<=bit_number) { aussian[i]=0;i++;}

    I = 0;
    while (I < random_no)
    {
```

```

        j=rand()%(bit_number+1);

        if (distribution[j]==0)
            {continue;}

        if ( aussian[j]==1)
            {continue;}

// Deduct from the aussian
distribution[j]--;
aussian[j] = 1;
i++;
}

```

6.4 A Simulation of a Priori Bit Distributions

The purpose of this simulation is to find the relations between occurrences of the different distribution types of a priori bit and a random error bit by finding match probability. The match is counted when a priori bit catches a error bit where a priori bit and the error bit are in the same bit position in a block. The match probability is calculated by total number of match divided by total number of blocks simulated. The simulations are classified into three different cases as

1. A priori bit with a random single error bits.
2. A priori bit with a random multiple error bits.
3. A priori bit with a random single burst error bits.

A priori bits distributions, equal distance, Gaussian and uniform, used for the simulation is show in Figures 6.4.

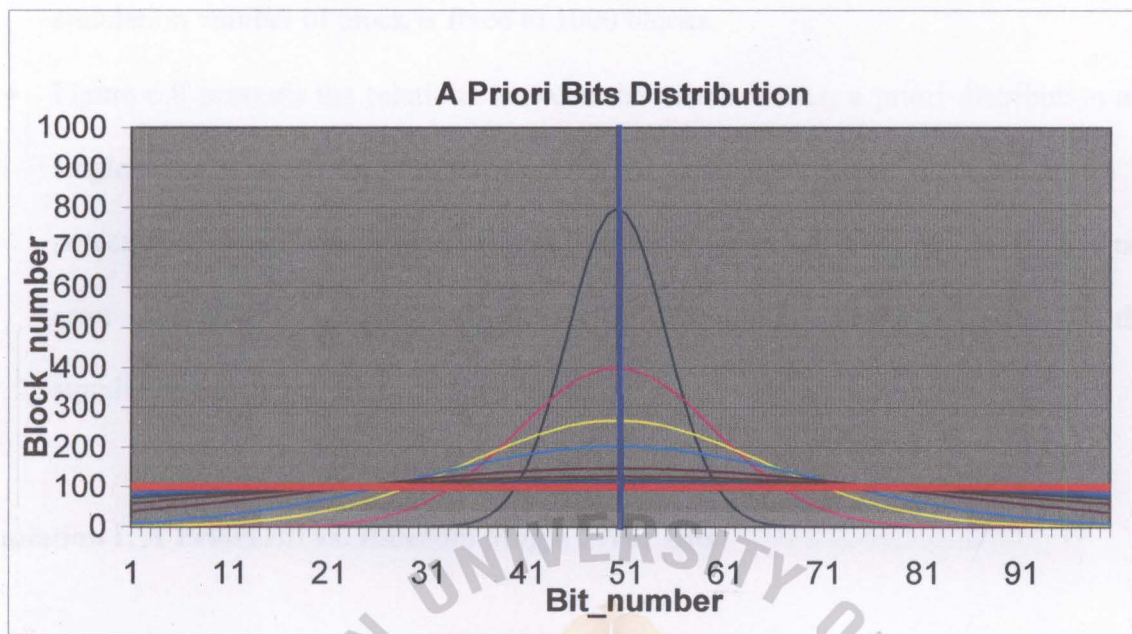


Figure 6.4: A priori Bits Distributions

The result from the simulation related to the purposed 1-3 are shown in Figures 6.5 to Figures 6-9, where

- For the values from 1 to 100 shown as dispersion present the distributions of a priori bits. The distributions expressed by the values are
 1 = Equal distance distribution
 10-90 = Gaussian distribution
 100 = Uniform distribution
- Figure 6.5 presents the relations between the block size, a priori distribution and number of blocks.
- Figure 6.6 presents the relations between the block size, a priori distribution and multiple random a priori bits. The multiple random error bits ratio in a block is shown by percentages from 1 to 100%. In Figure 6.7, the relations are compared by block

size and a priori bits distributions at fixed multiple random error percentage. For this simulation number of block is fixed to 1000 blocks.

- Figure 6.8 presents the relations between the match values, a priori distribution and single burst a priori bit. The single burst error bits ratio in a block is shown by percentages from 1 to 100%. In Figure 6.9, the relations are compared by single burst error ratio in a block and a priori bits distributions at fixed block size. For this simulation number of block is fixed to 1000 blocks.

Simulation I: A Priori Bit vs. Random Single Error Bits

The purpose of the simulation is to evaluate match probability of different a priori bits distribution and single random bit error.

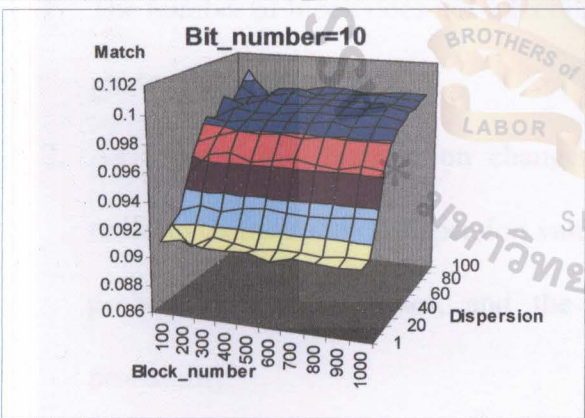


Figure 6.5 (a): 10 Bits

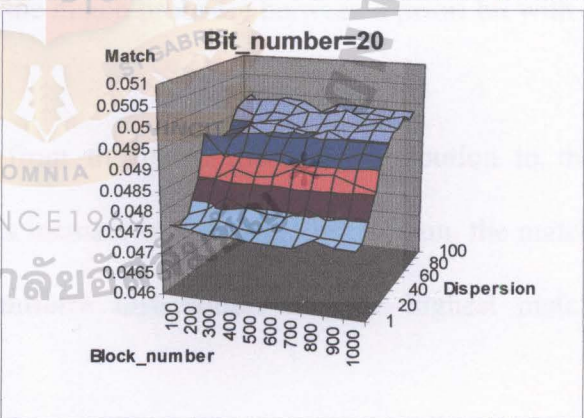


Figure 6.5 (b): 20 Bits

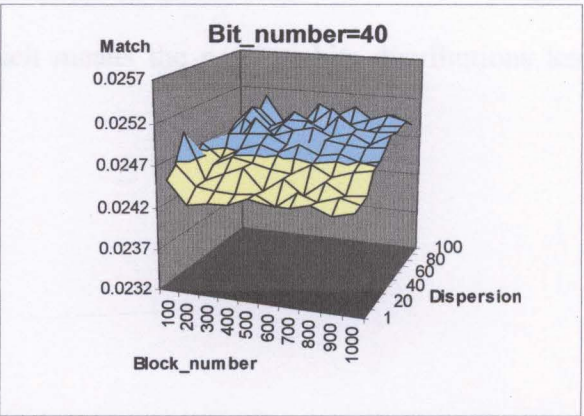
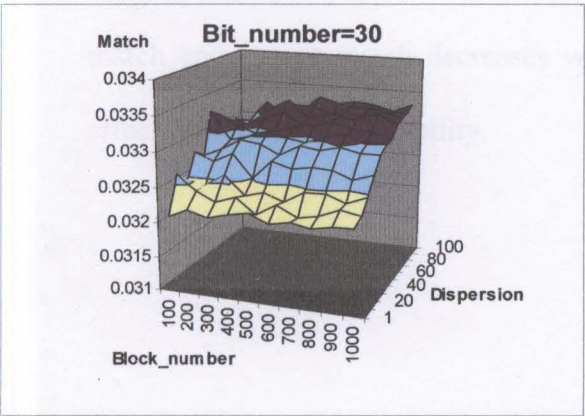


Figure 6.5 (c): 30 Bits

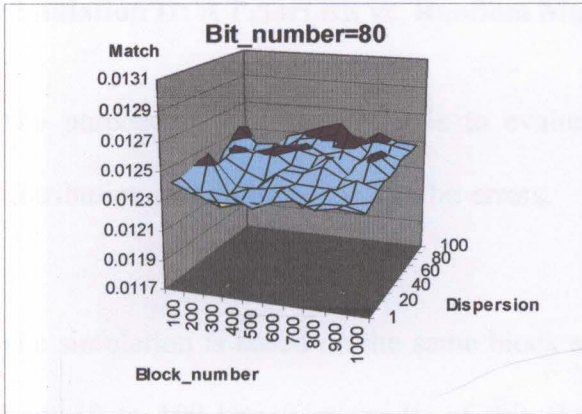


Figure 6.5 (d): 40 Bits

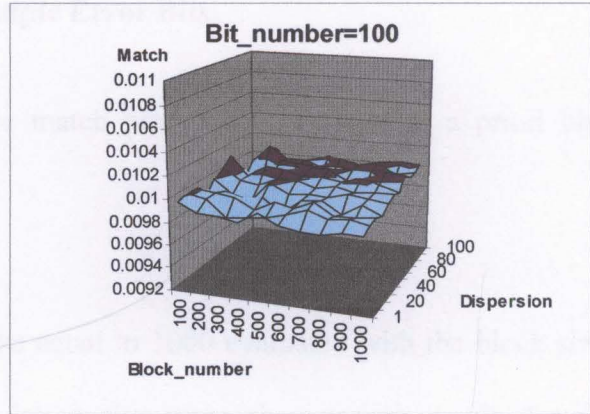


Figure 6.5 (e): 80 Bits

Figure 6.5 (f): 100 Bits

Figure 6.5: A Priori Bit vs. Random Single Error Bits

From the results shown in figure 6.5, the relations between the block size, a priori distribution and number of blocks are concluded as

1. The number of block does not effect on the match probably between a priori bit with a random single error bits.
2. As a priori bits distribution changes from the equal distance distribution to the uniform distribution or dispersion values increase of Gaussian distribution, the match probability also increases, and the uniform distribution has the highest match probability.
3. As block size increases, the difference of match probability between the highest match and lowest match decreases which means the a priori bits distributions less effect on the match probability.

Simulation II: A Priori Bit vs. Random Multiple Error Bits

The purpose of the simulation is to evaluate match probability of different a priori bits distribution and multiple random bit errors.

The simulation is based on the same block size equal to 1000 evaluated with the block size from 10 to 100 bits. The results of this simulation give same characteristic results for all block size, therefore, the result of smallest block size 10 bits and largest block size 100 bits are shown in figure 6.6.

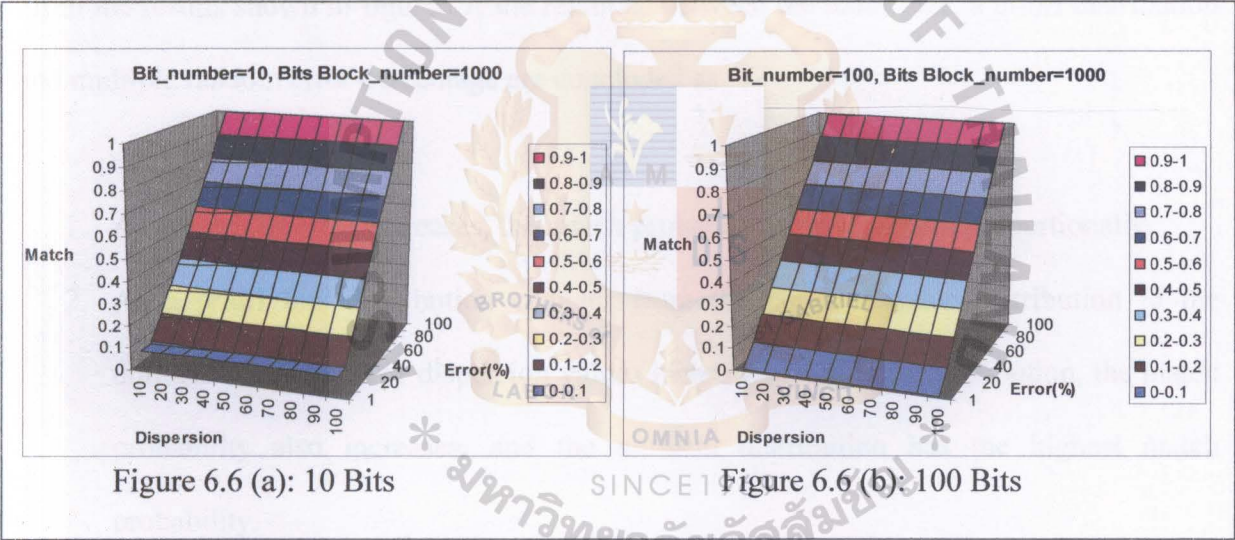


Figure 6.6: A Priori Bit vs. Random Multiple Error Bits

From the result graph shown in figure 6.6, the relations by a priori bits distribution and difference by block size can not be evaluated because of scale, so the following graph 6.7 was made to evaluate them.

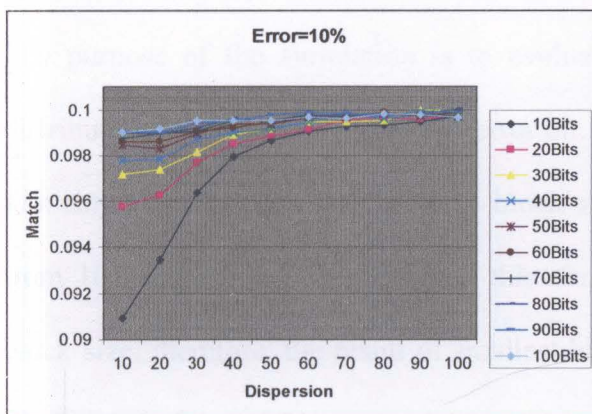


Figure 6.7 (a): Error 10 %

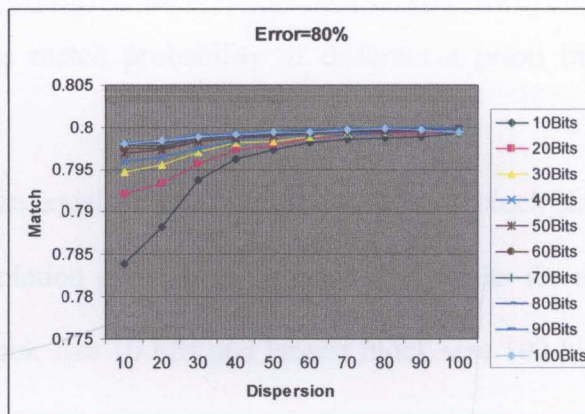


Figure 6.7 (b): Error 80 %

Figure 6.7: A Priori Bit vs. Random Multiple Error Bits in term of Percentage of Error

From the results shown in figure 6.7, the relations between the block size, a priori distribution and multiple random error percentage are concluded as

1. As number of error increases, the match probability also increases proportionally.
2. As a priori bits distribution changes from the equal distance distribution to the uniform distribution or dispersion values increase of Gaussian distribution, the match probability also increases, and the uniform distribution has the highest match probability.
3. As block size increases, the difference of match probability between the highest match and lowest match decreases which means a priori bits distributions have less effect on the match probability, however, at smaller dispersion, as block number increases the match probability also increases.

Simulation III: A priori Bit vs. Random Single Burst Bits

The purpose of the simulation is to evaluate match probability of different a priori bits distribution and single random burst error bit.

The simulation is based on the same block size equal to 1000 evaluated with the block size from 10 to 100 bits. The results of this simulation give same characteristic results for all block size, therefore, the result of smallest block size 10 bits and largest block size 100 bits are shown in figure 6.8.

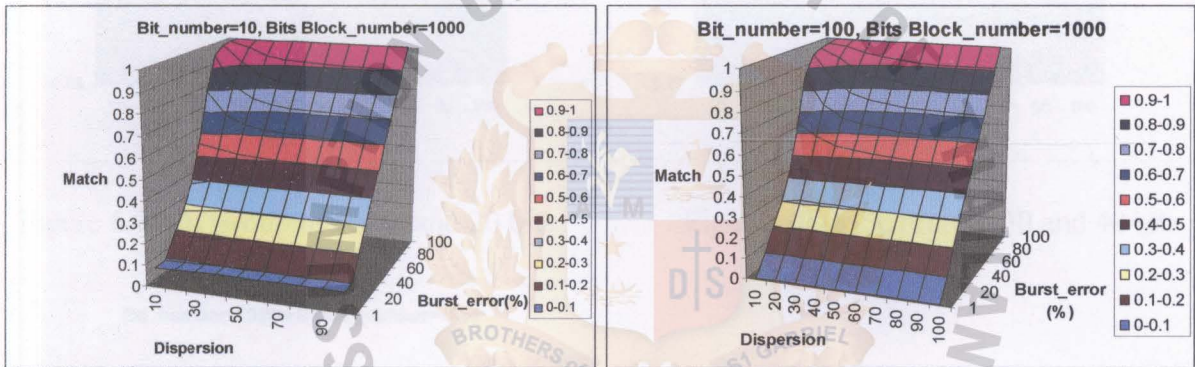


Figure 6.8 (a): 10 bits

Figure 6.8 (b): 100 bits

Figure 6.8: A priori Bit vs. Random Single Burst Bits

From the result graph shown in figure 6.8, the relations by a priori bits distribution and difference by the burst error size can not be evaluated because of scale so the following graph 6.9-6.10 were made to evaluate them.

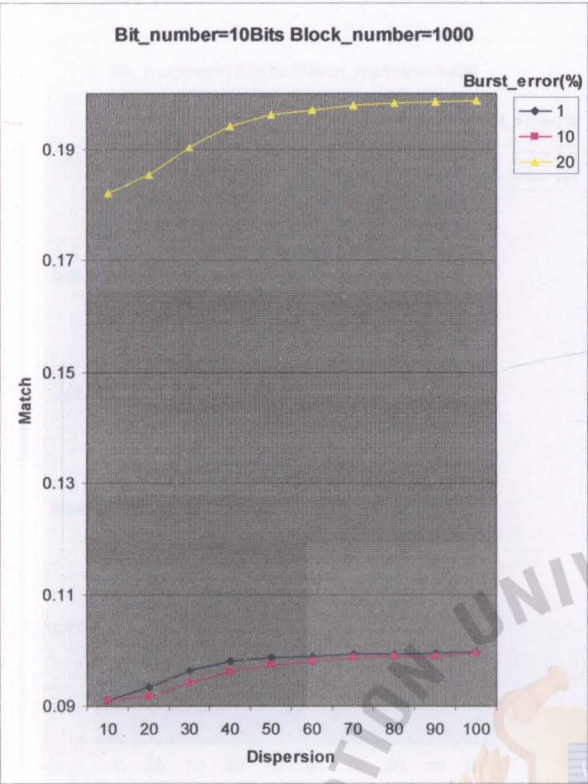


Figure 6.9 (a): Group of 1, 10, and 20 bits

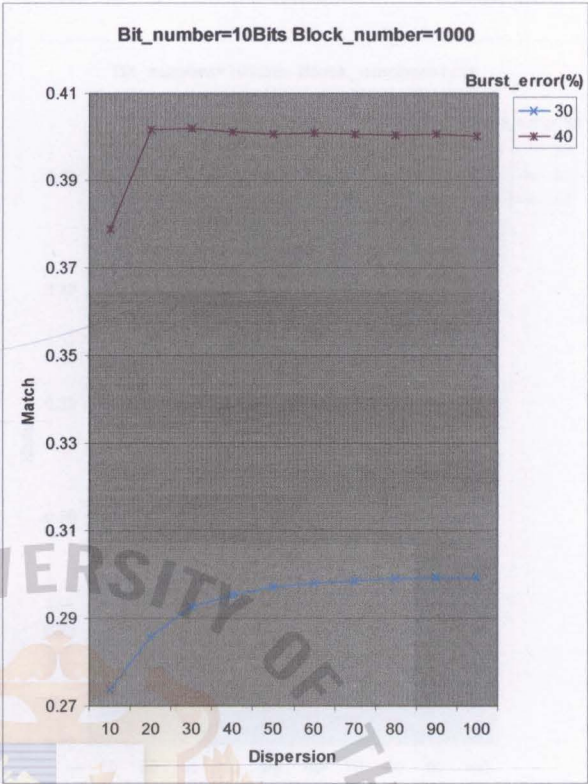


Figure 6.9 (b): Group of 30 and 40 bits

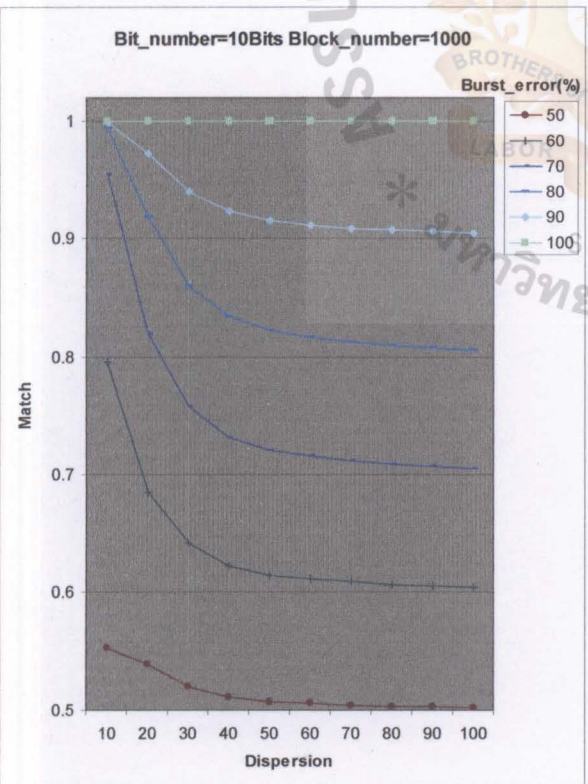


Figure 6.9 (c): Group of 50 to 100 bits

Figure 6.9: A priori Bit vs. Random Single Group of Block Size = 10 Bits

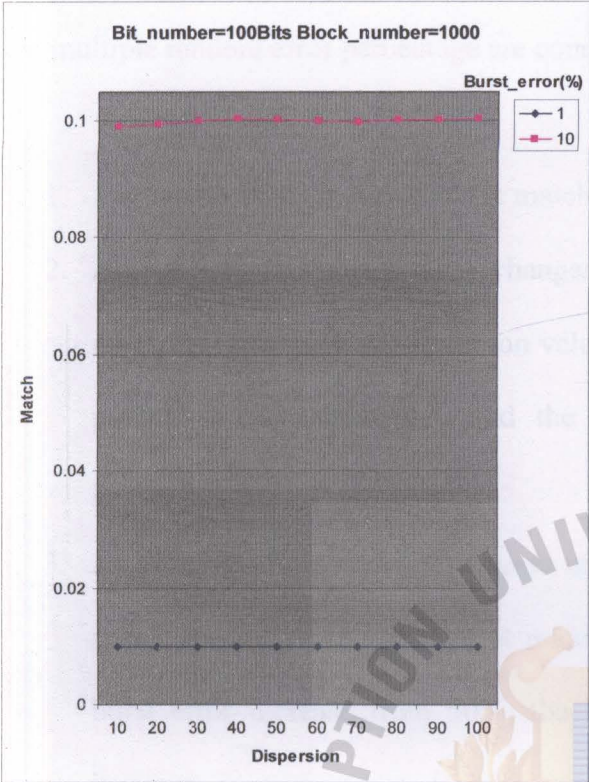


Figure 6.10 (a): Group of 1 and 10 bits

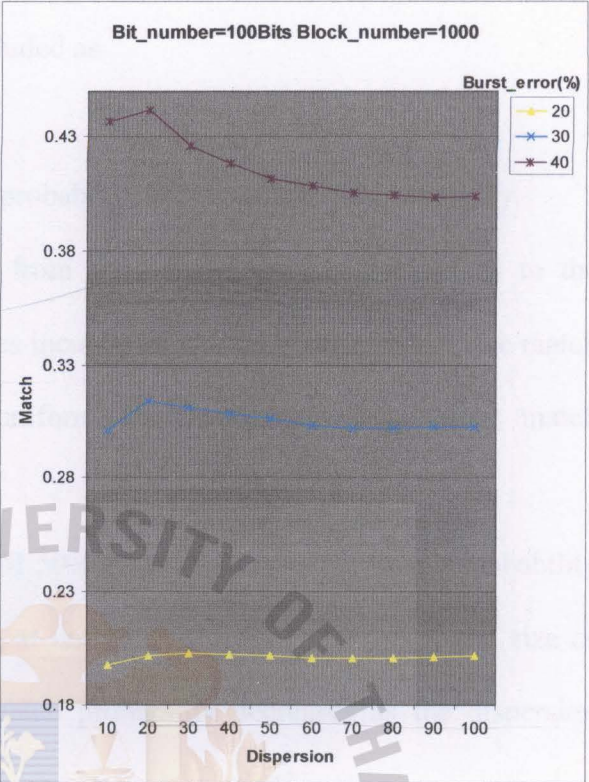


Figure 6.10 (b): Group of 20 to 40 bits

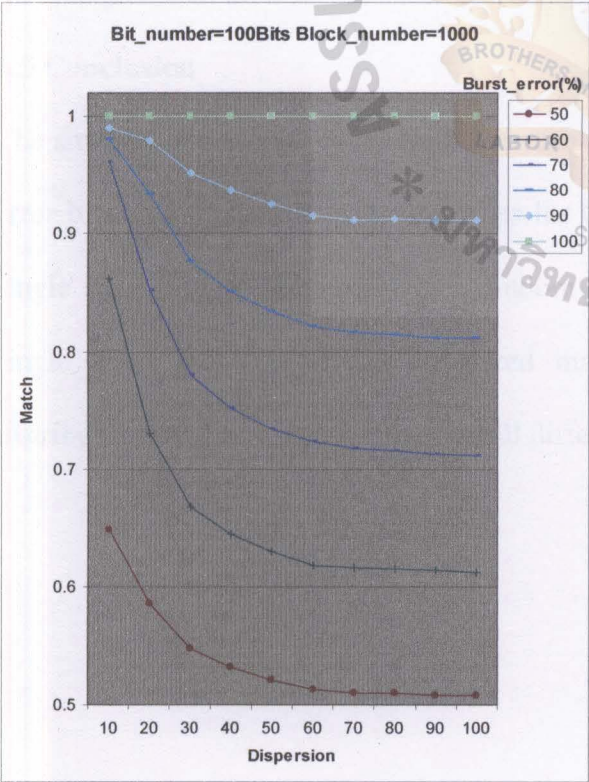


Figure 6.10 (c): Group of 50 to 100 bits

Figure 6.10: A priori Bit vs. Random Single Group of Block Size=100 bits

From the results shown in figure6, the relations between the block size, a priori distribution and multiple random error percentage are concluded as

1. As number of error increase, the match probability also increases proportionally.
2. As a priori bits distribution changes from the equal distance distribution to the uniform distribution or dispersion values increase of Gaussian distribution, the match probability also increases, and the uniform distribution has the highest match probability.
3. As the size of burst error increases until 50% of a block size, the match probability also increases with the highest match at uniform distribution, however the size of burst error increase from 50%, the match probability decreases as the dispersion increase.

6.5 Conclusion

The simulation result gives relations between occurrences of a priori bit and different types of error bits pattern by finding the matches for three different cases, a priori bit with a random single error bits, a priori bit with a random multiple error bits, a priori bit with a random single burst bits though the compared match probability result between a priori bits distributions for each case has very small differences.

CHAPTER 7: IMPLEMENTED TURBO CODEC SIMULATION

This chapter presents the main results of the research. The program simulates a turbo codec with different distributions of a priori bits. The simulations were performed with a rate 1/3 turbo decoder. The results gave the performance changes with BER and the FER for each case.

7.1 Simulation Setup

The simulation setup is composed of two distinct parts, a priori bits generation and a priori bits insertion. For simulations, block size of 128 bits, 256 bits, 512 bits and 1024 bits were used for 1E5 operation times.

7.1.1 A priori Bits Generation

Three kinds of a priori bits (Equal, Gaussian and Uniform) were distributed by using same code described in chapter 6. In each block, a priori bits were added with different a priori inclusion ratio by percentages, 1%, 5%, 10%, 15%, 20%, of block bits as figures 7.1, 7.2 and 7.3 shown below for each distribution pattern.

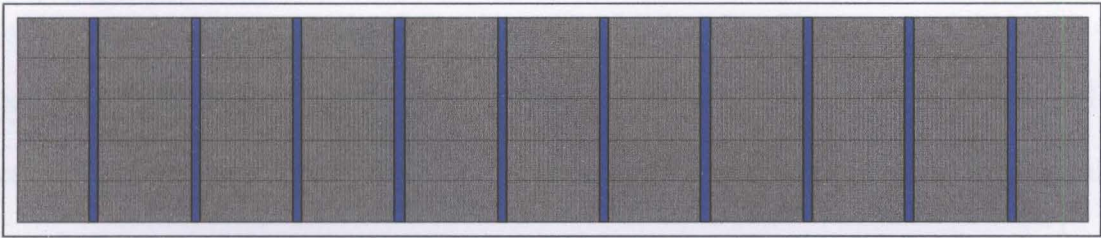


Figure 7.1: Histogram of Equal Distribution a priori Bits

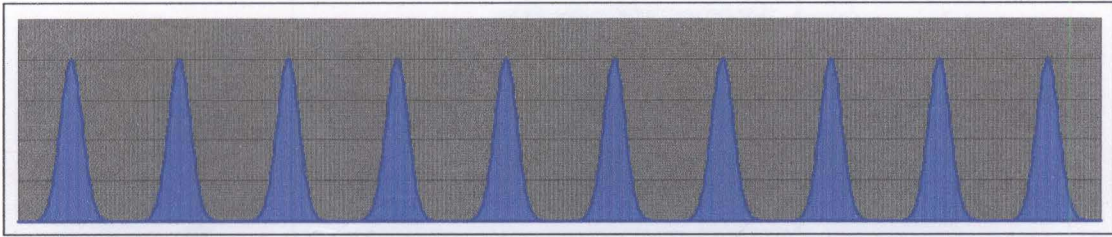


Figure 7.2: Histogram of Gaussian Distribution a priori Bits



Figure 7.3: Histogram of Uniform Distribution a priori Bits

7.1.2 A priori Bits Insertion

The simulation of the turbo code encoder is based on the literature [14]. The simulated turbo code encoder is composed of two identical RSC component encoders. These two component encoders are separated by a random interleaver. The random interleaver is a random permutation of bit order in a bit stream. This random permutation of bit order is stored so that the interleaved bit stream can be deinterleaved at the decoder. The output of the turbo code encoder is described by three streams, one systematic message (uncoded) bit stream and two coded parity bit streams. The first a priori bits before transmission were added into one of the output of the turbo encoder, systematic message bits as shown in figure 7.4.

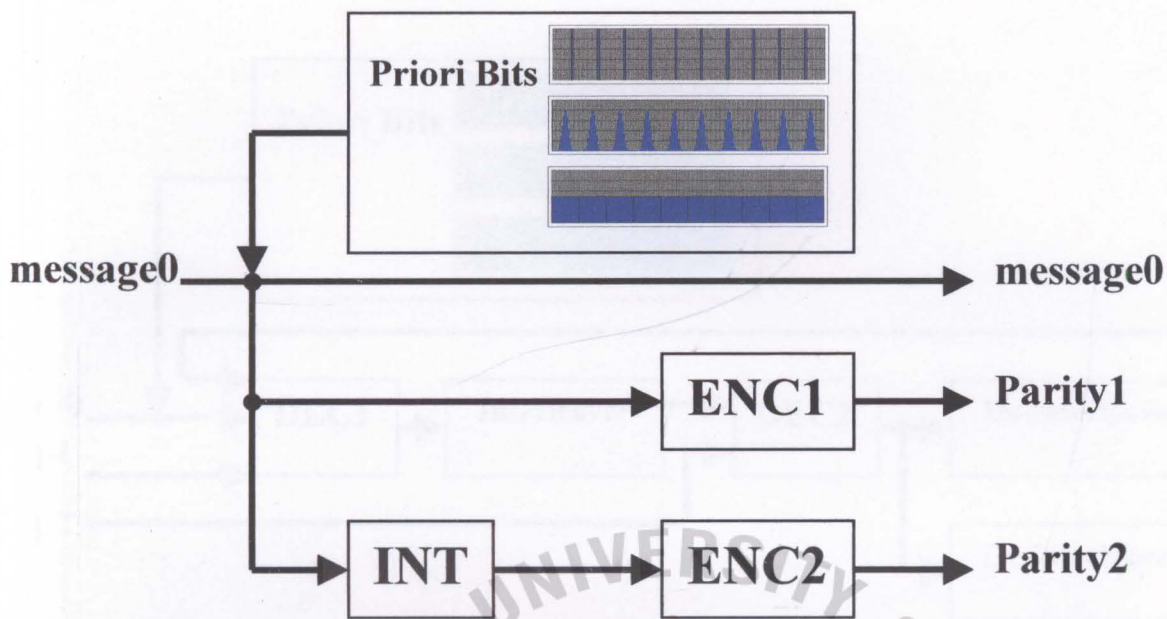


Figure 7.4: A priori Bit Insertion at the Encoding Process

In the simulation, the Gaussian distribution was used as the channel noise, because it is a fairly good model for different transmission mediums including deep space communication. The Gaussian distribution was constructed with mean of zero and standard deviation of one. In order to use this model, the turbo code encoder output bit streams must be mapped from $\{0, 1\}$ to $\{-1, +1\}$ domain.

The simulation of the turbo code decoder is based on literature [14]. For the received systematic message bit stream, the noisy a priori bits are replaced with the same a priori bits at encoding process before passing to the component of turbo decoder as shown in figure 7.5.

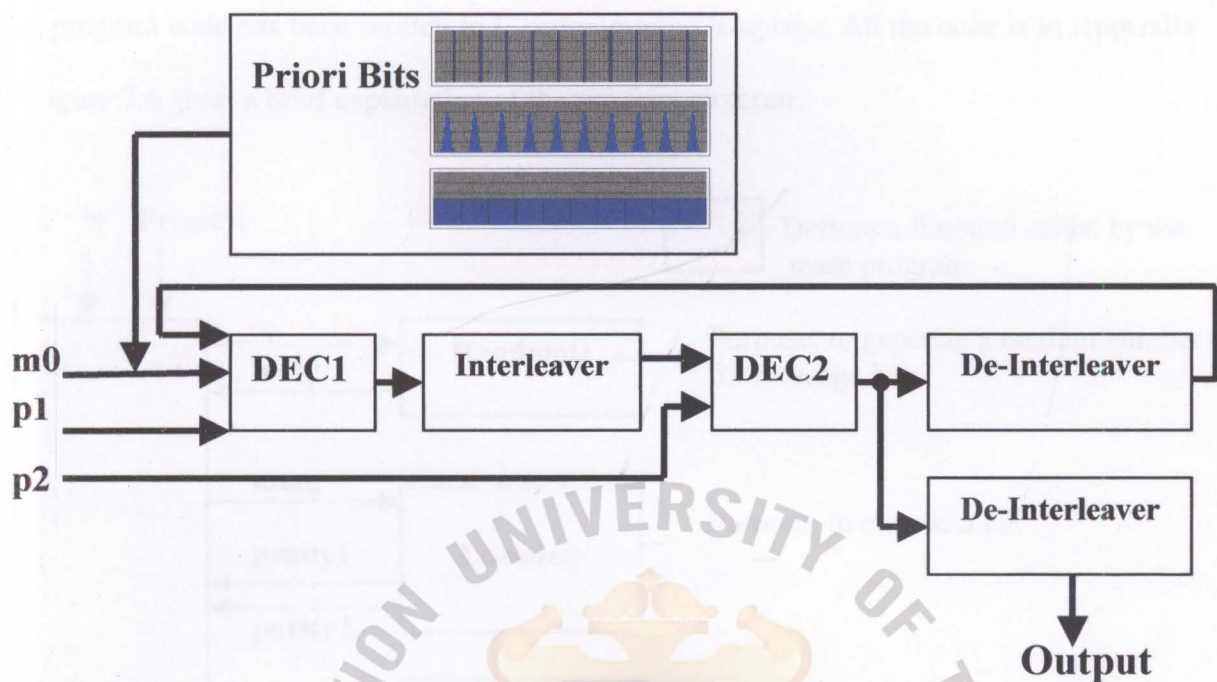
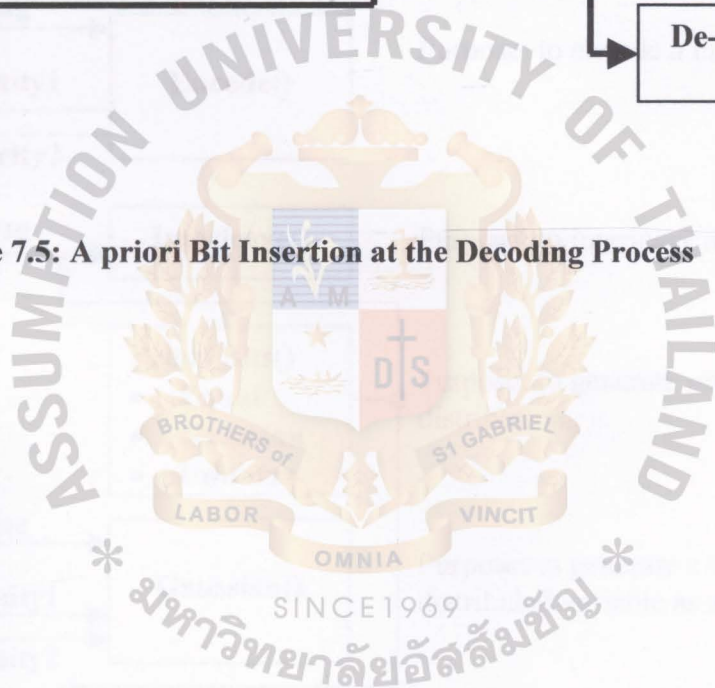


Figure 7.5: A priori Bit Insertion at the Decoding Process



7.2 Synoptic scheme of the software implementation

The program code has been written in C programming language. All the code is in Appendix C. Figure 7.6 gives a brief explanation of the program structure.

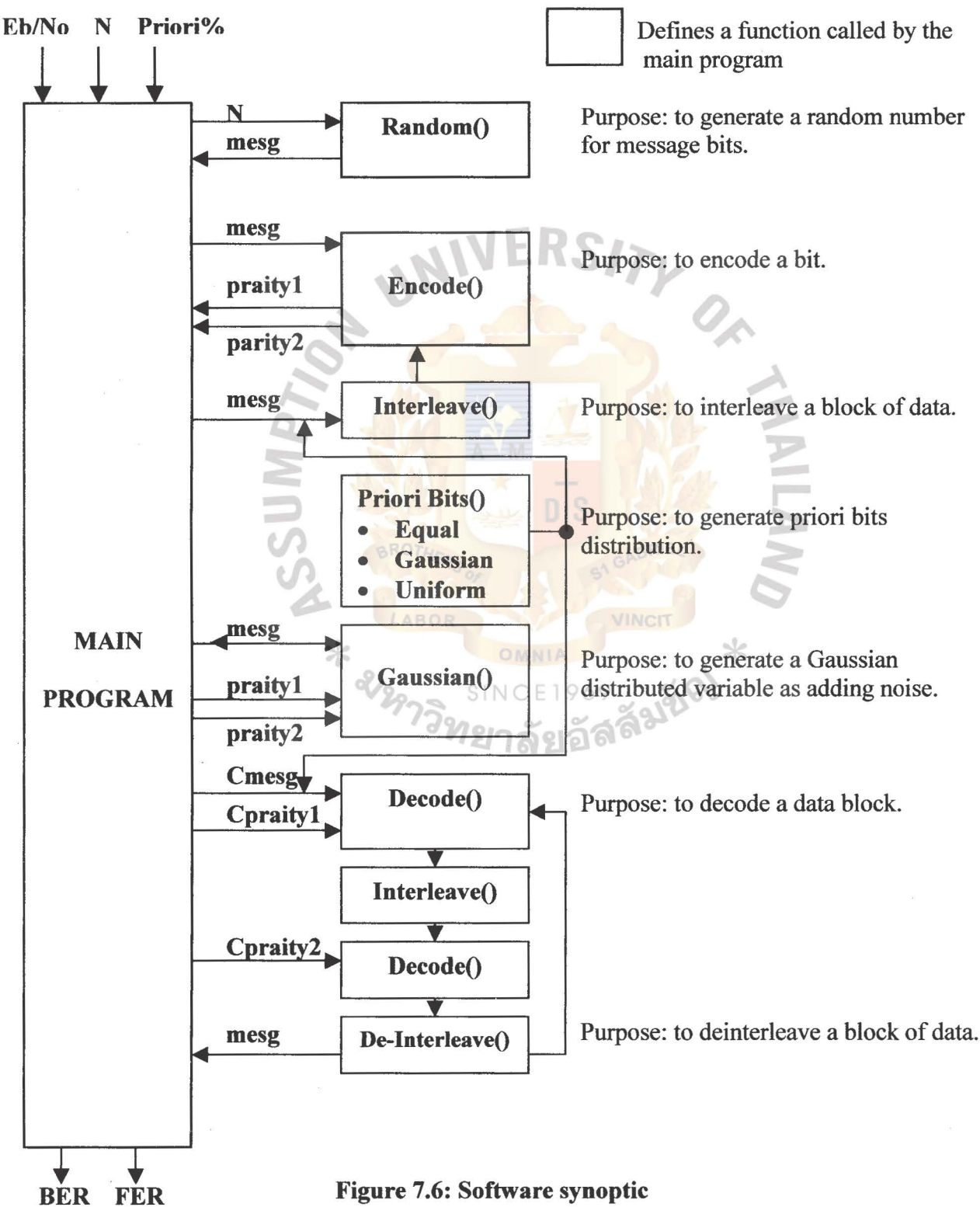


Figure 7.6: Software synoptic

This program includes the code so that the simulation can start where it left off if the program is stopped for some reason. After each turbo code run, the important variables are saved in a file.

* mesg is the original message that is compared after each iteration to the decoded message.

* channelmesg is the AWGN corrupted message.

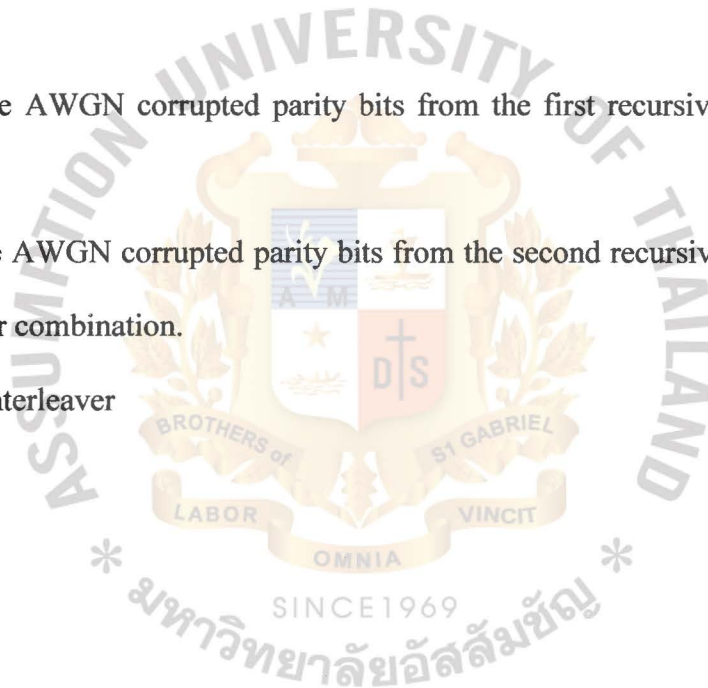
* parity1 is the parity bits from the first recursive convolutional encoder.

* parity2 is the parity bits from the second recursive convolutional encoder and interleaver combination.

* channelparity1 is the AWGN corrupted parity bits from the first recursive convolutional encoder.

* channelparity2 is the AWGN corrupted parity bits from the second recursive convolutional encoder and interleaver combination.

* N is the size of the interleaver



7.3 Results Analysis

The purposes of simulation are classified into two different areas such as

1. To evaluate the error correcting performance with different block size by BER and FER.
2. To evaluate the relationship between a priori bits distribution patterns at defined range of SNR.

The results from the simulation related to the purposes 1 is shown in Figures 7.5 – Figures 7.6, where

1. Figures 7.5 presents the BER performance comparison between different block sizes, 128, 256, 512, 1024 Bits without a priori bits at decoding process.
2. Figures 7.6 presents the FER performance comparison between different block sizes, 128, 256, 512, 1024 Bits without a priori bits at decoding process.

The results from the simulation related to the purposes 2 is shown in Figures 7.7 – Figures 7.16, where

1. Figures 7.7 - Figures 7.8 present the BER performance comparison between different a priori bits distributions. The ratio of a priori bits in a block is changed by percentages
2. Figures 7.9 - Figures 7.16 present the FER performance comparison between different a priori bits distributions. The ratio of a priori bits in a block is are changed by percentages

The next section will describe the simulation detail, in addition with the performance analysis for each model.

Simulation I: Performance Comparison Between Different Block Sizes without a Priori Bits at the Decoding Process.

The purpose of the simulation is to evaluate the error correcting performance with different block size by BER and FER. The evaluation is completed by

- 1. Setting up the simulation configuration as follows :

Simulation I Configuration:

block size = 128

SNR= (double dB=0; dB<=2.5; dB+=0.5)

Number of blocks = 1E5

- 2. Running the simulation software with the different block size such as 128, 256, 512, 1024 Bits to compare the throughput of the simulation by BER and FER.
- 3. Investigating the simulation result

The simulation result is shown in Figure 7.5. The simulation results shown are focused on the range of signal to noise ratio between 0dB and 2.5dB which is the most considerable range for the deep space communication with turbo code.

From the results obtained in Figure 7.5-7.6, the conclusion for the performance evaluation without a priori bits is that as the block size increases the error correcting performance also increases.

This result can be used as a reference to compare the simulation results with different distributions of a priori bits during the decoding process.

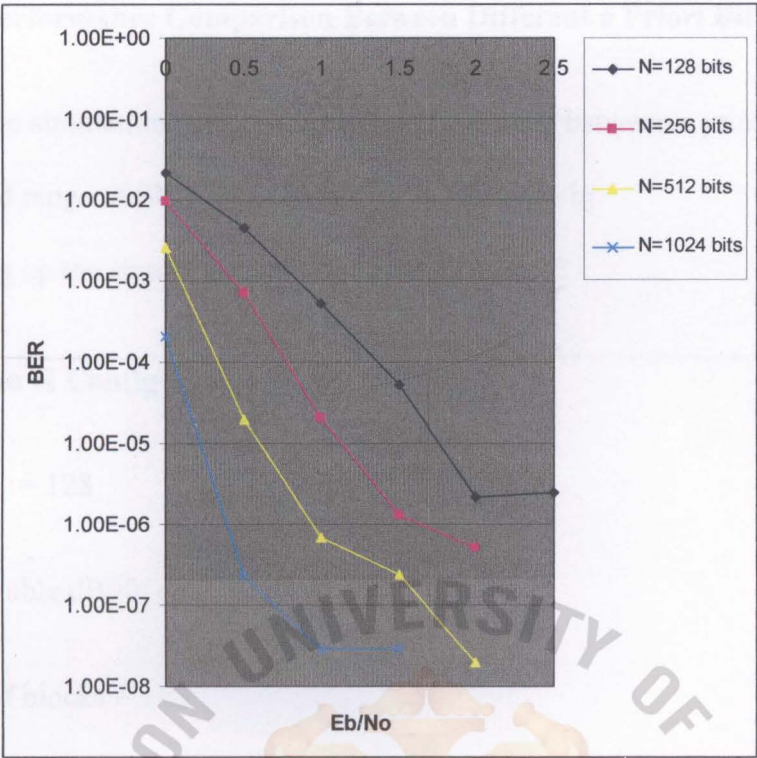


Figure 7.7: BER without a priori bits

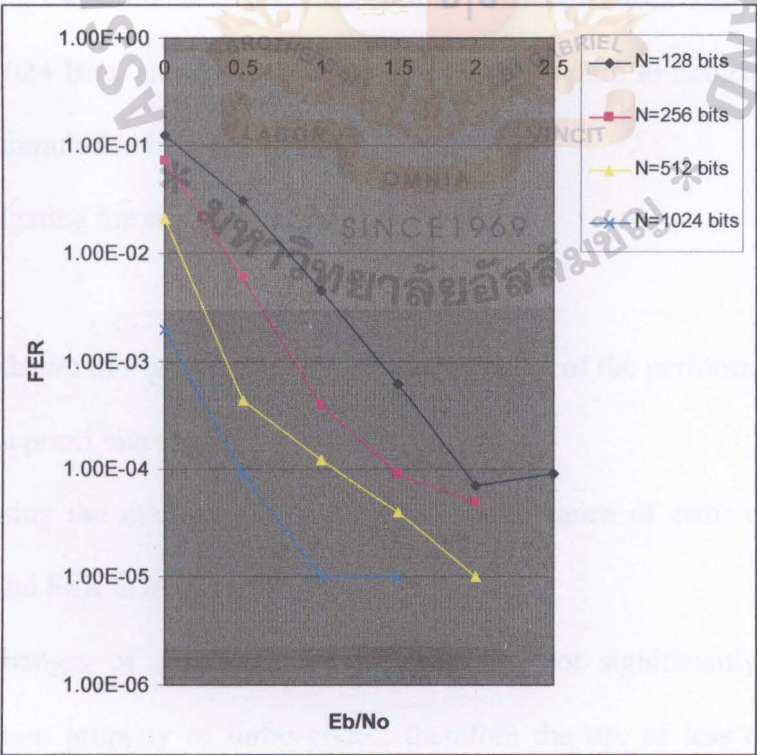


Figure 7.8: FER without a priori bits

Simulation II: Performance Comparison Between Different a Priori Bits Distributions

The purpose of the simulation is to evaluate the relationship between a priori bits distribution patterns at defined range of SNR. The evaluation is complete by

- 1. Setting up the simulation configuration as follow :

Simulation II Configuration:

block size = 128

SNR= (double dB=0; dB<=2.5; dB+=0.5)

Number of blocks = 1E5

Distribution =Equal Distance, Gaussian (Dispersion = 10), Uniform

- 2. Running the simulation software with the different block size such as 128, 256, 512, 1024 Bits, and different a priori bits distribution to compare the throughput of the simulation by BER and FER.
- 3. Investigating the simulation result

From the results shown in Figures 7.11-7.18, the conclusion of the performance evaluation of turbo codec with a priori bits can be summarized as follow

- 1. Increasing the number of a priori bits, performance of error correcting in both BER and FER is improved.
- 2. The changes of a priori bit distribution do not significantly affect the error correction property of turbo codec, therefore the use of less complicated equal distance distribution is recommended to reduce the turbo code complexity.

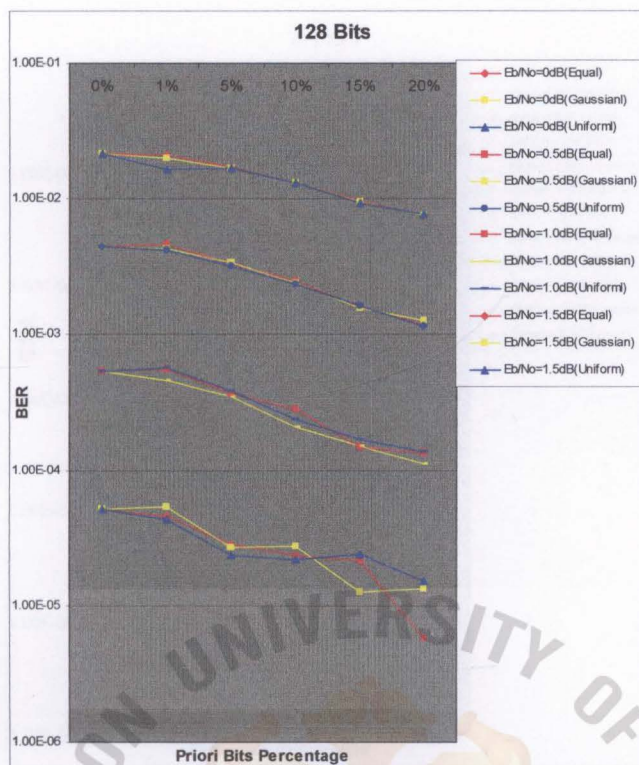


Figure 7.9: BER at Block Size=128 bits

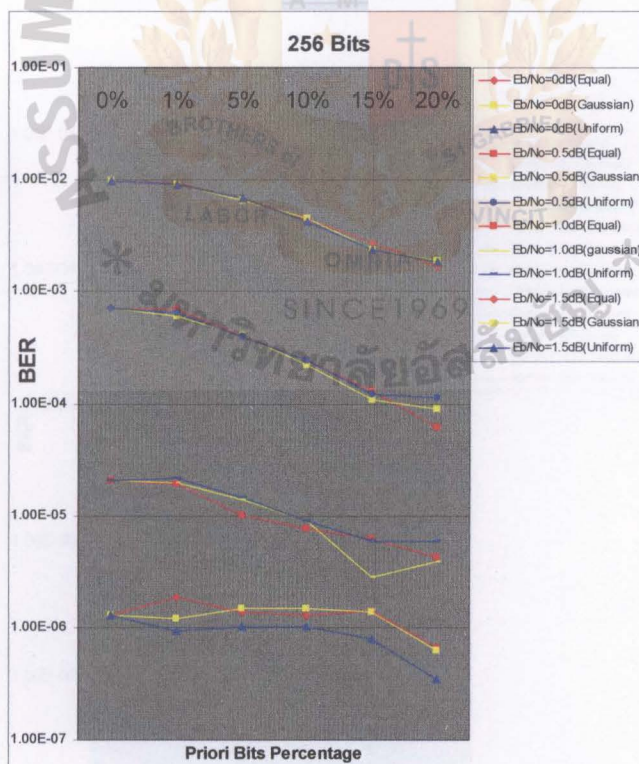


Figure 7.10: BER at Block Size=256 bits

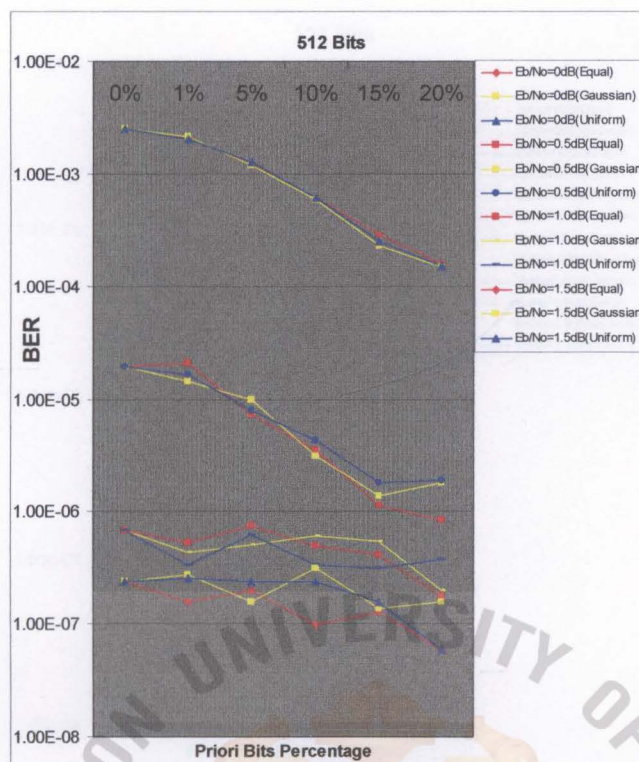


Figure 7.11: BER at Block Size=512 bits

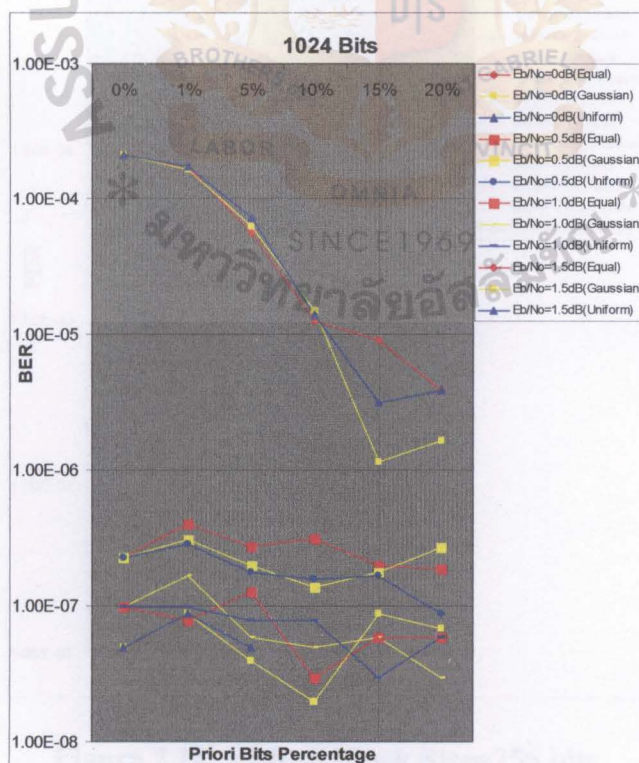


Figure 7.12: BER at Block Size=1024 bits

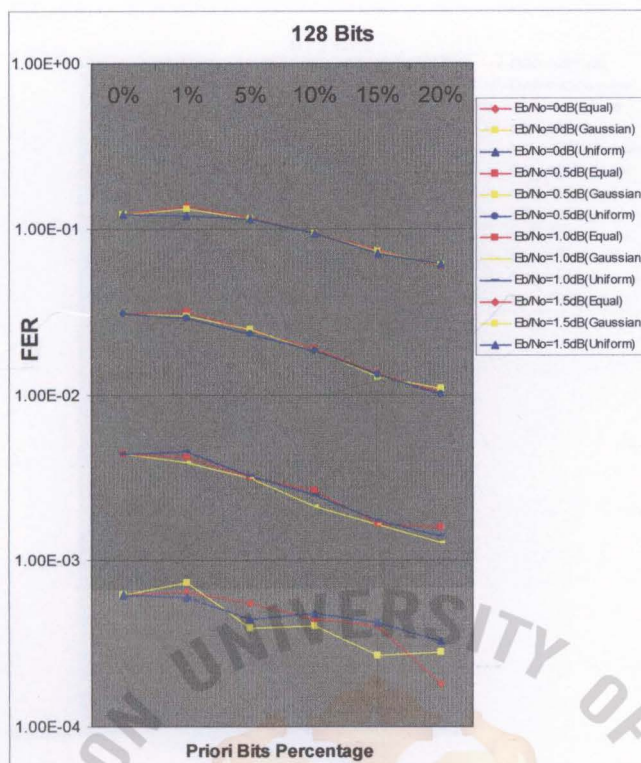


Figure 7.13: FER at Block Size=128 bits

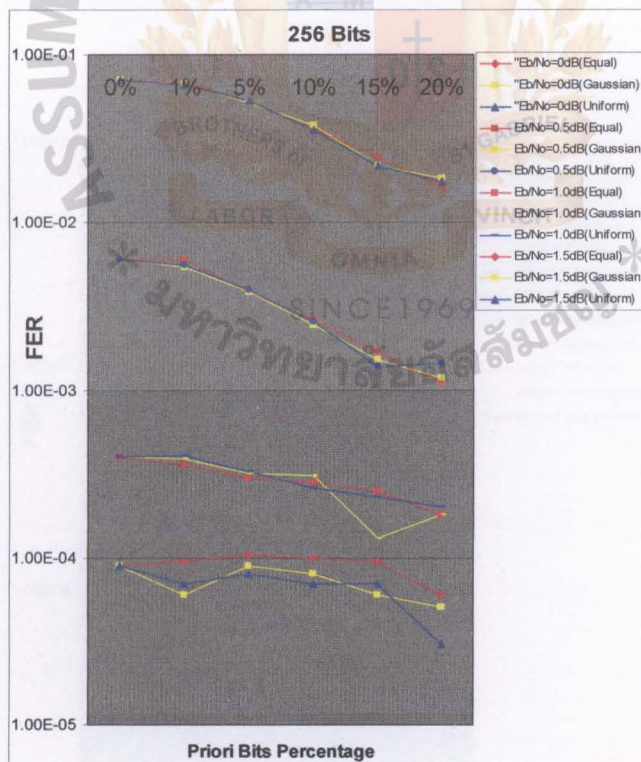


Figure 7.14: FER at Block Size=256 bits

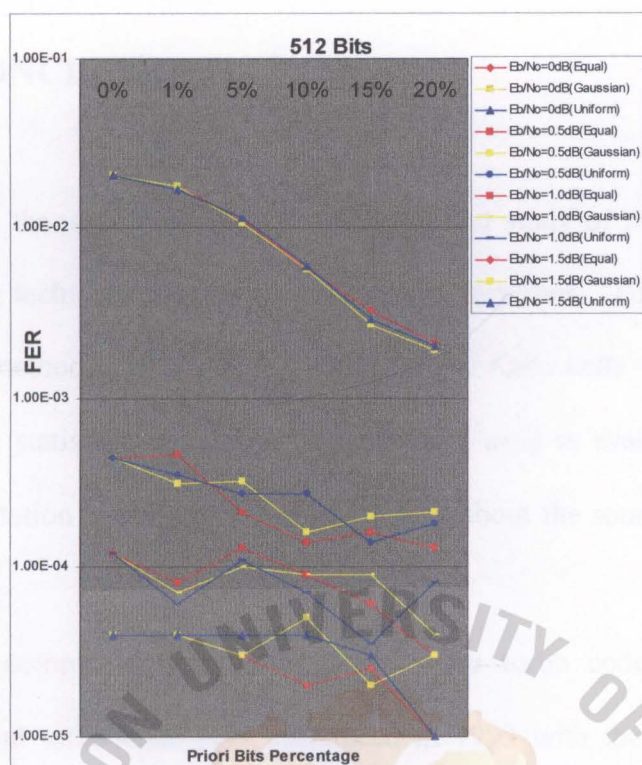


Figure 7.15: FER at Block Size=512 bits

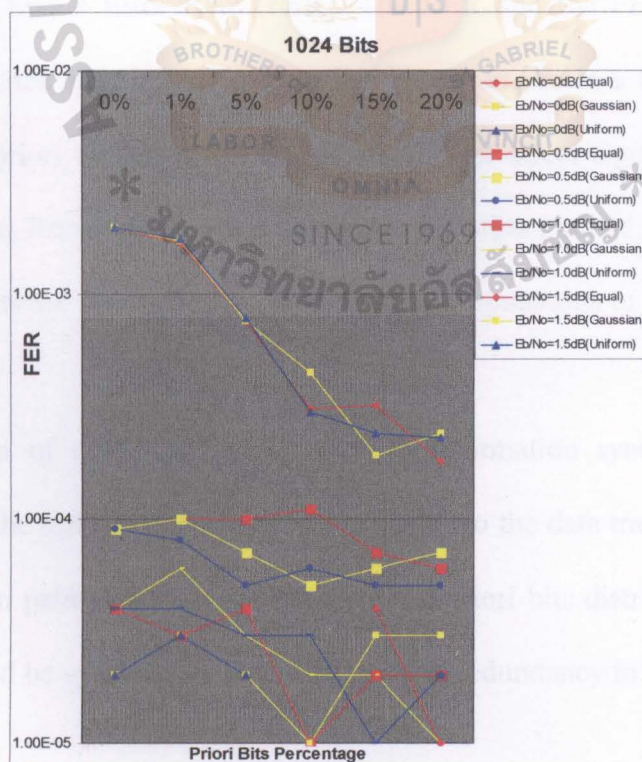


Figure 7.16: FER at Block Size=1024 bits

CHAPTER 8: CONCLUSION

This thesis discussed the approach of implementation and study of the performance of the existing turbo coding techniques for deep space communications. The main subject of the research focused on methods combining the advantages of turbo code with the inclusion of a priori information. A statistical simulation software was used to evaluate the effect of the turbo code implementation by using a priori knowledge about the source during the channel decoding.

Though only slight complexity gain was added to the turbo codec by the method of combining the original turbo code [11] introduced in 1993 with the inclusion of a priori information, the simulation results indicate the advantage which is the clear improvement that the source decoder gives better performance by observing an improvement of the Bit Error Rate (BER) and the Frame Error Rate (FER). The BER and the FER were improved by increasing the block size and the ratio of a priori bits inclusion in a block. The simulation results also show a priori bits distribution patterns do not affect significantly on the turbo decoding performance, therefore, by using this implementation method, the use of the equally distributed a priori bits are recommended to make the computations of decoding process less complicated.

In the real operation of communications, a priori information synchronization must be established between the turbo encoder and decoder prior to the data transmission for initial a priori bits distribution pattern though the changes of a priori bits distribution pattern during communications could be synchronized as using another redundancy in code block during the data transmission.

The future work in this field would include the development of a priori bits inclusion approach in dealing with the turbo codec. The suggested approaches of a priori information inclusion are in parity bits after encoding process, the different patterns of a priori information in each parity bit. Also at the decoding process, the probability of error occurrence from the previous decoding history could be used to estimate the error pattern to modify a priori bits distribution pattern during the decoding process. For this self a priori bits distribution pattern modification, a priori information synchronization would be done during the data communication. A proper inclusion of a priori information and error estimation algorithm are to be further developed.



REFERENCES

- [1] K. Andrews, V. Stanton, S. Dolinar, V. Chen, J. Berner, and F. Pollara, "Turbo-Decoder Implementation for the Deep Space Network (2002)", IPN PR 42-148, October-December 2001, pp. 1-20, February 15, 2002.
- [2] Thomas Hindelang, Joachim Hagenauer, Stefan Heinen, "Source-Controlled Channel Decoding: Estimation of Correlated Parameters (2000)", 3rd ITG Conference Source and Channel Coding, München, Januar 2000 S.251-258
- [3] Tim Fingscheidt, Thomas Hindelang, Richard V. Cox, Nambi Seshadri, "Combined Source/Channel Decoding: When Minimizing Bit Error Rate is Suboptimal (2000)", in Proc. Of 3rd ITG Conference Source and Channel Coding, Munich, Germany, January 2000.
- [4] T. Hindelang, T. Fingscheidt, N. Seshadri, R.V. Cox, "Combined Source/Channel (De-) Coding: Can A Priori Information Be Used Twice?" ICC (3) 2000: 1208-1212
- [5] M. Kaindl, T. Hindelang, "Estimation Of Bit Error Probabilities Using A Priori Information (1999)", in proc. of IEEE GLOBECOM'99, Rio de Janeiro, Brazil, Dec 1999, Vol.5, pp.2422-2426.
- [6] Oscar Y Takeshita, Oliver M. Collins, Peter C. Massey, and Daniel J. Costello, "On the frame Error Rate of Turbo-Codes (1998)", Proc. ITW 1998, Killarney, Ireland, pp.118-119, June 22-26, 1998.

[7] D. Divsalar and F. Pollara “Turbo Codes for Deep-Space Communications (1995)”, JPL TDA Progress Report 42-120, Feb. 15, 1995.

[8] Johan Hokfelt, “On The Design Of Turbo Code (2000)” ISSN1402-8662:17, ISBN 91-7874-061-4, Lund University, Sweden, August 2000.

[9] Charles Wang, Dean Sklar, and Diana Johnson “Forward Error-Correction Coding (2002)”, Crosslink Volume 3, Number 1 (Winter 2001/2002)
<http://www.aero.org/publications/crosslink/winter2002/04.html>

[10] Oscar Y. Takeshita, *Member, IEEE*, and Daniel J. Costello, Jr., *Fellow, IEEE*, “New Deterministic Interleaver Designs for Turbo Codes (2000)”, IEEE Transactions on Information Theory, VOL. 46, NO. 6, 1988-2006, SEPTEMBER 2000

[11] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit error correcting coding and decoding: Turbo-Codes,” ICC’93, Geneva, Switzerland, pp. 1064–1070, May 1993.

[12] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, “Optimal decoding of linear codes for minimizing symbol error rate,” IEEE Transactions on Information Theory, vol. IT-20, pp. 284–287, March 1974.

[13] J. Hagenauer and P. H^oher, “A Viterbi algorithm with soft-decision outputs and its applications,” in Globecom, pp. 1680–1686, IEEE, November 1989.

- [14] William E. Ryan, "A Turbo Code Tutorial," <http://telsat.nmsu.edu/~wryan/turbo2c.ps>
- [15] J. Hagenauer, P. Roberston and L. Papke, "Iterative (turbo) decoding of systematic convolutional codes with the MAP and SOVA algorithms," *ITG Conf.*, Frankfurt, Germany, Oct. 1994.
- [16] D. Divsalar and F. Pollara, "On the design of turbo codes," *JPL TDA Progress Report*, vol. 42-123, pp. 99-121, July-Sep. 1995.
- [17] C. E. Shannon, "A mathematical theory of communication," *Bell Sys. Tech. J.*, vol.27, pp. 379-423, July 1948 and pp. 623-656, Oct. 1948.
- [18] X. Wang and S. B. Wicker, "A soft output decoding algorithm for concatenated systems," submitted to *IEEE Transactions on Information Theory*, Dec. 1994.
- [19] P. Robertson, "Improving decoder and code structure of parallel concatenated recursive systematic (Turbo) codes," *3rd Int. Conference on Universal Personal Communication*, San Diego, California, pp. 183-187, Sep. 1994.
- [20] S. Benedetto and G. Montorsi, "Performance evaluation of turbo codes," *Electron. Lett.*, vol. 31, No. 3, pp. 163-165, Feb. 1995.
- [21] P. L. McAdam, L. R. Welch, and C. L. Weber, "M.A.P. Bit Decoding of Convolutional Codes" (Abstract), *1972 International Symposium on Information Theory*, Asilomar, California, p. 91, May 1972.

[22] B. Mielczarek, "Synchronization in Turbo Coded systems", PhD thesis, Chalmers University of Technology, Sweden, April 2000.

[23] Guido Masera, Gianluca Piccinini, Massimo Ruo Roch, and Maurizio Zamboni, "VLSI Architectures for Turbo Codes", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 7, No: 3, September 1999.



APPENDIX A: BCJR ALGORITHM

The following text has been taken with permission from the lecture file of Coding Theory at <http://www.s-t.au.ac.th/~Alib/students/TS6313/TS631317.doc> and explains theoretically the BCJR algorithm [14] used for Turbo code simulations.

For a discussion of turbo decoding to be complete, a mathematical exposition of the BCJR algorithm for MAP estimation is in order.

Let $x(t)$ be the input to a trellis encoder at time t . Let $y(t)$ be the corresponding output observed at the receiver. Note that $y(t)$ may include more than one observation; for example, a rate $1/n$ code produces n bits for each input bit, in which case we have an n -dimensional observation vector. Let the observation vector be denoted by

$$\mathbf{y}_{(1,t)} = [y(1), y(2), \dots, y(t)]$$

Let $\lambda_m(t)$ denote the probability that a state $\mathbf{s}(t)$ of the trellis encoder equals m , where $m = 1, 2, \dots, M$. We may then write

$$\lambda(t) = P(\mathbf{s}(t) | \mathbf{y}) \quad (10.77)$$

where $\mathbf{s}(t)$ and $\lambda(t)$ are both M -by-1 vectors. Then, for a rate $1/n$ linear convolutional code with feedback as in the RSC code, the probability that a symbol “1” was the message bit is given by

$$P(x(t) = 1 | \mathbf{y}) = \sum_{s \in F_A} \lambda_s(t) \quad (10.78)$$

where F_A is the set of transitions that correspond to a symbol “1” at the input, and $\lambda_s(t)$ is the s -component of $\lambda(t)$.

Define the **forward estimation** of state probabilities as the M -by-1 vector

$$\alpha(t) = P(\mathbf{s}(t) | \mathbf{y}_{(1,t)}) \quad (10.79)$$

where the observation vector $\mathbf{y}_{(1,t)}$ is defined above. Also define the **backward estimation** of state probabilities as the M -by-1 vector

$$\beta(t) = P(\mathbf{s}(t) | \mathbf{y}_{(t,k)}) \quad (10.80)$$

where

$$\mathbf{y}_{(t,k)} = [y(t), y(t+1), \dots, y(k)]$$

The vectors $\alpha(t)$ and $\beta(t)$ are estimates of the state probabilities at time t based on the past and future data, respectively. We may then formulate the **separability theorem** as follows:

The state probabilities at time t are related to the forward estimator $\alpha(t)$ and backward estimator $\beta(t)$ by the vector

$$\lambda(t) = \frac{\alpha(t) \cdot \beta(t)}{\|\alpha(t) \cdot \beta(t)\|_1} \quad (10.81)$$

where $\alpha(t) \cdot \beta(t)$ is the vector product of $\alpha(t)$ and $\beta(t)$, and $\|\alpha(t) \cdot \beta(t)\|_1$ is the L_1 norm of this vector product.

The **vector product** $\alpha(t) \cdot \beta(t)$ (not to be confused with the inner product) is defined in terms of the individual elements of $\alpha(t)$ and $\beta(t)$ by

$$\alpha(t) \cdot \beta(t) = \begin{bmatrix} \alpha_1(t)\beta_1(t) \\ \alpha_2(t)\beta_2(t) \\ \vdots \\ \alpha_M(t)\beta_M(t) \end{bmatrix} \quad (10.82)$$

and the L_1 **norm** of $\alpha(t) \cdot \beta(t)$ is defined by

$$\|\alpha(t) \cdot \beta(t)\|_1 = \sum_{m=1}^M \alpha_m(t)\beta_m(t) \quad (10.83)$$

The separability theorem says that the state distribution at time t given the past is independent of the state distribution at time t given the future, which is intuitively satisfying recalling the Markovian assumption for channel encoding, which is basic to the BCJR algorithm. Moreover, this theorem provides the basis of a simple way of combining the forward and backward estimates to obtain a complete description of the state probabilities.

To proceed further, let the state transition probability at time t be

$$\gamma_{m',m}(t) = P(s(t) = m', \mathbf{y}(t) | s(t-1) = m) \quad (10.84)$$

and denote the M -by- M matrix of transition probabilities as

$$\Gamma(t) = \{\gamma_{m',m}(t)\} \quad (10.85)$$

We may then formulate the **recursion theorem** as follows:

The forward estimate $\alpha(t)$ and backward estimate $\beta(t)$ are computed recursively as

$$\alpha^T(t) = \frac{\alpha^T(t-1)\Gamma(t)}{\|\alpha^T(t-1)\Gamma(t)\|_1} \quad (10.86)$$

and

$$\beta^T(t) = \frac{\Gamma(t+1)\beta(t+1)}{\|\Gamma(t+1)\beta(t+1)\|_1} \quad (10.87)$$

where the superscript T denotes matrix transposition.

The separability and recursion theorems together define the BCJR algorithm for the computation of a posteriori probabilities of the states and transitions of a code trellis, given the observation vector. Using these estimates, the likelihood ratios needed for turbo decoding may then be computed by performing summations over selected subsets of states as required.



APPENDIX B: DIFFERENTIATION ENTROPY OF UNIFORM DISTRIBUTION

The result of better matches with uniform distribution in chapter 6 can be proven by following mathematical solutions which has been taken with permission from the lecture file of Coding Theory at <http://www.s-t.au.ac.th/~Alib/students/TS6313/TS631307.doc>.

Problem 2.8.2 A continuous random variable X is constrained to a peak magnitude M ; that is, $-M < X < M$.

- (a) Show that the differential entropy of X is maximum when it is uniformly distributed, as shown by

$$f_X(x) = \begin{cases} \frac{1}{2M}, & -M < x < M \\ 0, & \text{otherwise} \end{cases}$$

- (b) Show that the maximum differential entropy of X is $\log_2 2M$.

Solution:

From the fundamental inequality in information theory of Eq. (10.12) to the situation at hand, we may write

$$\int_{-\infty}^{\infty} f_Y(x) \log_2 \frac{f_X(x)}{f_Y(x)} dx = \int_{-M}^M f_Y(x) \log_2 \frac{f_X(x)}{f_Y(x)} dx \leq 0$$

or equivalently

$$-\int_{-M}^M f_Y(x) \log_2 f_Y(x) dx \leq -\int_{-M}^M f_Y(x) \log_2 f_X(x) dx$$

The quantity on the left-hand side of Eq.(10.71) is the differential entropy of the random variable Y ; hence,

$$h(Y) \leq -\int_{-M}^M f_Y(x) \log_2 f_X(x) dx$$

Suppose now the random variable X is described as follows:

- The random variable X is uniformly distributed as shown by

$$f_X(x) = \begin{cases} \frac{1}{2M}, & -M < x < M \\ 0, & \text{otherwise} \end{cases}$$

Hence, substituting the uniform distribution into the inequality, we get

$$h(Y) \leq \int_{-M}^M f_Y(x) \log_2(2M) dx$$

We now recognize the following property of the random variable Y :

$$\int_{-M}^M f_Y(x) dx = 1$$

We may therefore simplify the inequality as

$$h(Y) \leq \log_2(2M)$$

The quantity on the right-hand side of the inequality is in fact the differential entropy of the uniformly distributed random variable X :

$$h(X) = \log_2(2M)$$

Finally, we may write

$$h(Y) \leq h(X) \quad \begin{cases} X : \text{uniformly distributed random variable} \\ Y : \text{another random variable} \end{cases}$$

where equality holds if, and only if, $Y = X$.

We may now summarize the result of this important example as follows:

1. ***For a finite interval $(-M < x < M)$, the uniformly distributed random variable X has the largest differential entropy $h(X)$ attainable by any random variable Y .***

APPENDIX C: CODE USED TO SIMULATE THE TURBO CODEC

All turbo codec simulations are performed by using the code based on following turbo code simulator which is originally written by Mathys Walma taken from <http://www.eccpage.com>. This codec is using BCJR algorithm[14], based on the pseudocode in W.E.Ryan's tutorial paper[X].

Turbo.cpp

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "random.h".

const int SEED=1000;
unsigned int N=128;
unsigned int ITERATIONS=20;
double No;

// zero mean RV with variance as given
double gaussian(double variance);
// encodes mesg into parity1, and if force it true it modifies mesg to
// force the encoder to the zero state by the last bit.
void encode(bool *mesg, bool *parity1, unsigned size, bool force);
void interleave(bool *mesg, unsigned size);
// binary addition with no carry
bool add(bool a, bool b);
void interleave(bool *mesg, unsigned size);
void deinterleave(bool *mesg, unsigned size);
void deinterleavedouble(double *mesg, unsigned size);
void interleavedouble(double *mesg, unsigned size);
void createinterleave(unsigned size);
void createencodetable();
void inttobool(unsigned state, bool *array, unsigned size);
void booltoint(bool *array, unsigned size, unsigned *state);
unsigned decode (double *channelmesg, double *parity1, double *parity2, unsigned size, bool *mesg);
unsigned *interleavearray;
unsigned *deinterleavearray;

// global information about the encoder used by the encoder and the decoder
// routine
// how many states are in the encoder (a power of 2)
unsigned numstates;
// log2(numstates)
unsigned memory;
```



```

// [2] = input, [16] = current state, tostate[2][16] = next state
unsigned *tostate[2];
// [2] = last input, [16] = current state, fromstate[2][16] = previous state
unsigned *fromstate[2];
// [2] = input, [16] = current state, output[2][16] = output of encoder
bool *output[2];
Random before;

main()
{
    bool *mesg;
    bool *parity1;
    bool *parity2;
    double *channelmesg;
    double *channelparity1;
    double *channelparity2;

    interleavearray = new unsigned[N];
    deinterleavearray = new unsigned[N];
    mesg = new bool[N];
    parity1 = new bool[N];
    parity2 = new bool[N];
    channelmesg = new double[N];
    channelparity1 = new double[N];
    channelparity2 = new double[N];

    // only needs to be done once
    createencodetable();

    bool firstloop=true;

    // change if you want to loop for other values of dB
    for (double dB=-4.0;dB<=-4.0;dB+=0.5)
    {
        unsigned totalN=0;
        unsigned totalerrors=0;
        unsigned numiter;
        unsigned totaliter=0;
        unsigned numloops=0;

        // load the previous state
        if (firstloop)
        {
            FILE *fp;
            char line[30];

            firstloop = false;

            if ((fp = fopen("laststate","r")) != NULL)
            {
                fgets(line,30,fp);
                dB = atof(line);
                fgets(line,30,fp);
                totalerrors = atoi(line);
                fgets(line,30,fp);
                totalN = atoi(line);
            }
        }
    }

```

```

        fgets(line,30,fp);
        totaliter = atoi(line);
        fgets(line,30,fp);
        numloops = atoi(line);

    fclose(fp);
}

// dB = 10*log(Eb/No) where Eb is 1
No = 1/pow(10.0,dB/10.0);

bool keepgoing=true;

do
{
    (int x=0;x<N;x++)
    mesg[x] = before.boolrandom();

// create a random interleaver for each decode trial
    createinterleave(N);

    encode(mesg,parity1,N,true);
    interleave(mesg,N);
    encode(mesg,parity2,N,false);

// deinterleave for the decoder
    deinterleave(mesg,N);

    for (int x=0;x<N;x++)
    {
        mesg[x] ? channelmesg[x] = 1.0 : channelmesg[x]=-1.0;
        parity1[x] ? channelparity1[x]=1.0 : channelparity1[x]=-1.0;
        parity2[x] ? channelparity2[x]=1.0 : channelparity2[x]=-1.0;
    }

// add gaussian noise, mean=0, variance=No/2
    for (int x=0;x<N;x++)
    {
        channelmesg[x] += gaussian(No/2);
        channelparity1[x] += gaussian(No/2);
        channelparity2[x] += gaussian(No/2);
    }

    numiter = decode(channelmesg, channelparity1, channelparity2,N,mesg);

    unsigned numerrors=0;

    for (int x=0;x<N;x++)
    {
        bool temp = channelmesg[x] == 1 ? true : false;

        if (mesg[x] != temp)
            numerrors++;
    }

    totalerrors += numerrors;
    totalN += N;

```

```

        totaliter += numiter;
        numloops++;

        char name[30];

// open a file for output
    sprintf (name,"%dN%lddB",N,dB);

    FILE *fp;

// if file exists, we will append to it
    fp = fopen(name,"a");

    ("N=%d, e=%d, totITER=%d, numITER=%d, No=%ld, tote=%d, totn=%d,
dB=%ld\n",N,numerrors,ITERATIONS,numiter,No,totalerrors,totalN,dB);
    fprintf (fp,"N=%d, e=%d, totITER=%d, numITER=%d, No=%ld, tote=%d, totn=%d,
dB=%ld\n",N,numerrors,ITERATIONS,numiter,No,totalerrors,totalN,dB);

    fclose(fp);

// save this state in case we need to start over again
    fp = fopen ("laststate", "w");
    fprintf(fp,"%ld\n",dB);
    fprintf(fp,"%u\n",totalerrors);
    fprintf(fp,"%u\n",totalN);
    fprintf(fp,"%u\n",totaliter);
    fprintf(fp,"%u\n",numloops);

    fclose(fp);

// a number of conditions for stopping the simulation for this dB value
    if (totalerrors> 1000)
        keepgoing = false;
    else if (totalN>500000 && totalerrors > 100)
        keepgoing = false;
    else if (totalN>1E6 && totalerrors > 10)
        keepgoing = false;
    else if (totalN>5E6 && totalerrors > 5)
        keepgoing = false;
    else if (totalN>7.5E6 && totalerrors > 3)
        keepgoing = false;
    else if (totalN>1E7)
        keepgoing=false;
    }

    while (keepgoing == true); // 10 million is as high as I'm willing to go
// 1000 errors should be enough for good stats.

// add a summary of the last run to the summary file
    FILE *fp;

    fp = fopen("summary","a");

    printf ("dB = %ld, N=%ld, error=%ld, avgiter = %ld, interleaver size=%d, original encoder\n",dB,
totalN,totalerrors,(double)totaliter/(double)numloops,N);
    fprintf(fp,"dB = %ld, N=%ld, error=%ld, avgiter = %ld, interleaver size=%d, original encoder\n",dB,
totalN,totalerrors,(double)totaliter/(double)numloops,N);

```



```

    fclose(fp);
}

delete interleavearray;
delete deinterleavearray;
delete mesg;
delete parity1;
delete parity2;
delete channelmesg;
delete channelparity1;
delete channelparity2;
}

void createencodetable()
{
    bool *boolstate;
    bool *newstate;

    numstates = 16;
    memory = 4;

    // create arrays used by encode and decode
    output[0] = new bool[numstates];
    output[1] = new bool[numstates];
    fromstate[0] = new unsigned[numstates];
    fromstate[1] = new unsigned[numstates];
    tostate[0] = new unsigned[numstates];
    tostate[1] = new unsigned[numstates];
    boolstate = new bool[memory];
    newstate = new bool[memory];

    for (unsigned input=0;input<2;input++)
    for (unsigned intstate=0;intstate<numstates;intstate++)
    {
        bool boolinput = (input == 0) ? false : true;

        inttobool(intstate,boolstate,memory);

        // calculate output due to the input
        output[input][intstate] = add(boolinput,boolstate[0]);
        output[input][intstate] = add(output[input][intstate],boolstate[3]);
        output[input][intstate] = add(output[input][intstate],boolstate[1]);
        output[input][intstate] = add(output[input][intstate],boolstate[2]);
        [input][intstate] = add(output[input][intstate],boolstate[3]);

        // calculate new states
        newstate[3] = boolstate[2];
        newstate[2] = boolstate[1];
        newstate[1] = boolstate[0];
        newstate[0] = add(add(boolinput,boolstate[0]),boolstate[3]);
        // from s' to s
        booltoint (newstate,memory,&tostate[input][intstate]);
        // from s to s'
        fromstate[input][tostate[input][intstate]] = intstate;
    }
}

```

```

delete boolstate;
delete newstate;
}

void inttobool(unsigned state, bool *array, unsigned size)
{
    for (unsigned x=0;x<size;x++)
    {
        unsigned next = state >> 1;

        if ((next << 1) == state)
            array[x] = false;
        else
            array[x] = true;
        state = next;
    }
}

void booltoint(bool *array, unsigned size, unsigned *state)
{
    *state = 0;

    for (int x=0;x<size;x++)
        if (array[x] == true)
            (*state) |= (1 << x);
}

unsigned decode (double *mesg, double *parity1, double *parity2, unsigned size, bool *boolmesg)
{
    static double **a[2];
    static double **b[2];
    static double *L[2];
    static double **gamma[2][2];
    static double **gammaE[2][2];
    static bool initialized=false;
    static unsigned lastsize=0;
    unsigned returnvalue=ITERATIONS;

    // minimize new's and delete's to only when needed
    if (size != lastsize && initialized == true)
    {
        // delete all the arrays and rebuild
        for (int y=0;y<2;y++)
            for (int x=0;x<2;x++)
            {
                for (int z=0;z<lastsize;z++)
                {
                    delete gamma[y][x][z];
                    delete gammaE[y][x][z];
                }

                delete gamma[y][x];
                delete gammaE[y][x];
            }
    }
}

```

```

// create L[encoder #]
for (int y=0;y<2;y++)
    delete L[y];

// create alpha[encoder #][k][state]
for (int x=0;x<2;x++)
{
    for (int y=0;y<lastsize;y++)
    {
        delete a[x][y];
        delete b[x][y];
    }

    delete a[x];
    delete b[x];
}

if (initialized == false || size != lastsize)
{
    initialized = true;
    lastsize = size;
}

// create the arrays dynamically at runtime, delete at end of routine
// create gamma[encoder #][uk][k][state]
for (int y=0;y<2;y++)
for (int x=0;x<2;x++)
{
    gamma[y][x] = new double*[size];
    gammaE[y][x] = new double*[size];
    for (int z=0;z<size;z++)
    {
        gamma[y][x][z] = new double[numstates];
        gammaE[y][x][z] = new double[numstates];
    }
}

// create L[encoder #]
for (int y=0;y<2;y++)
    L[y] = new double[size];

// create alpha[encoder #][k][state]
for (int x=0;x<2;x++)
{
    a[x] = new double*[size];
    b[x] = new double*[size];
}

// each Yk has 'numstates' values of gamma
for (int y=0;y<size;y++)
{
    a[x][y] = new double[numstates];
    b[x][y] = new double[numstates];
}

}

// initialization of iteration arrays

```



```

for (int x=0;x<numstates;x++)
{
    a[0][0][x] = b[0][size-1][x] = a[1][0][x] = (x==0) ? 1.0 : 0.0;
// extrinsic information from 2-1
}

// initialization of extrinsic information array from decoder 2, used in decoder 1
for (int x=0;x<size;x++)
    L[1][x] = 0.0;

// 4*Eb/No
double Lc = (4.0*1.0)/No;

for (int c=0;c<ITERATIONS;c++)
{
// k from 0 to N-1 instead of 1 to N
    for (int k=0;k<size;k++)
    {
// calculate the gamma(s',s);
        for (int input=0;input<2;input++)
        {
            double uk = (input == 0) ? -1.0 : 1.0;

            for (int s=0;s<numstates;s++)
            {
                double xk = (output[input][s] == 0) ? -1.0 : 1.0;

                gammaE[0][input][k][s]=exp(0.5*Lc*parity1[k]*xk);
                gamma[0][input][k][s]=exp(0.5*uk*(L[1][k]+Lc*mesg[k]))*gammaE[0][input][k][s];
            }
        }
    }

// calculate the alpha terms
// from 1 to N-1, 0 is precalculated, N is never used
    for (int k=1;k<size;k++)
    {
        double temp=0;

// calculate denominator
        for (int state=0;state<numstates;state++)
            temp += a[0][k-1][fromstate[0][state]]*gamma[0][0][k-1][fromstate[0][state]] +
                a[0][k-1][fromstate[1][state]]*gamma[0][1][k-1][fromstate[1][state]];

        for (int state=0;state<numstates;state++)
            a[0][k][state] = (a[0][k-1][fromstate[0][state]]*gamma[0][0][k-1][fromstate[0][state]] +
                a[0][k-1][fromstate[1][state]]*gamma[0][1][k-1][fromstate[1][state]])/temp;
    }

// from N-1 to
    for (int k=size-1;k>=1;k--)
    {
        double temp=0;

// calculate denominator
        for (int state=0;state<numstates;state++)

```

```

temp += a[0][k][fromstate[0][state]]*gamma[0][0][k][fromstate[0][state]] +
        a[0][k][fromstate[1][state]]*gamma[0][1][k][fromstate[1][state]];

for (int state=0;state<numstates;state++)
b[0][k-1][state] = (b[0][k][tostate[0][state]]*gamma[0][0][k][state] +
                    b[0][k][tostate[1][state]]*gamma[0][1][k][state])/temp;
}

for (int k=0;k<size;k++)
{
    double min=0;

// find the minimum product of alpha, gamma, beta
    for (int u=0;u<2;u++)
        for (int state=0;state<numstates;state++)
        {
            double temp=a[0][k][state]*gammaE[0][u][k][state]*b[0][k][tostate[u][state]];

            if ((temp < min && temp != 0) || min == 0)
                min = temp;
        }

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

    double topbottom[2];

    for (int u=0;u<2;u++)
    {
        topbottom[u]=0.0;

        for(int state=0;state<numstates;state++)
            topbottom[u] += (a[0][k][state]*gammaE[0][u][k][state]*b[0][k][tostate[u][state]]);
    }

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

    L[0][k] = (log(topbottom[1]/topbottom[0]));
}

interleavedouble(L[0],size);
// remember to deinterleave for next iteration
interleavedouble(mesg,size);

// start decoder 2
// code almost same as decoder 1, could combine code into one but too lazy
    for (int k=0;k<size;k++)
    {
// calculate the gamma(s',s);
        for (int input=0;input<2;input++)
        {
            double uk = (input == 0) ? -1.0 : 1.0;

```

```

    for (int s=0;s<numstates;s++)
    {
        double xk = (output[input][s] == 0) ? -1.0 : 1.0;

        gammaE[1][input][k][s]=exp(0.5*Lc*parity2[k]*xk);

        gamma[1][input][k][s]=exp(0.5*uk*(L[0][k]+Lc*mesg[k]))*gammaE[1][input][k][s];
    }
}

// calculate the alpha terms
for (int k=1;k<size;k++)
{
    double temp=0;

// calculate denominator
    for (int state=0;state<numstates;state++)
        temp += a[1][k-1][fromstate[0][state]]*gamma[1][0][k-1][fromstate[0][state]] +
                a[1][k-1][fromstate[1][state]]*gamma[1][1][k-1][fromstate[1][state]];

    for (int state=0;state<numstates;state++)
        a[1][k][state] = (a[1][k-1][fromstate[0][state]]*gamma[1][0][k-1][fromstate[0][state]] +
                a[1][k-1][fromstate[1][state]]*gamma[1][1][k-1][fromstate[1][state]])/temp;
}

// in the first iteration, set b[1][N-1] = a[1][N-1] for decoder 2.
// this decoder can't be terminated to state 0 because of the interleaver
// the performance loss is supposedly negligible.
if (c==0)
{
    double temp=0;

// calculate denominator
    for (int state=0;state<numstates;state++)
        temp += a[1][size-1][fromstate[0][state]]*gamma[1][0][size-1][fromstate[0][state]] +
                a[1][size-1][fromstate[1][state]]*gamma[1][1][size-1][fromstate[1][state]];

    for (int state=0;state<numstates;state++)
        b[1][size-1][state] = (a[1][size-1][fromstate[0][state]]*gamma[1][0][size-1][fromstate[0][state]] +
                a[1][size-1][fromstate[1][state]]*gamma[1][1][size-1][fromstate[1][state]])/temp;
}

for (int k=size-1;k>=1;k--)
{
    double temp=0;

// calculate denominator
    for (int state=0;state<numstates;state++)
        temp += a[1][k][fromstate[0][state]]*gamma[1][0][k][fromstate[0][state]] +
                a[1][k][fromstate[1][state]]*gamma[1][1][k][fromstate[1][state]];

    for (int state=0;state<numstates;state++)
        b[1][k-1][state] = (b[1][k][tostate[0][state]]*gamma[1][0][k][state] +

```



```

        b[1][k][tostate[1][state]]*gamma[1][1][k][state])/temp;
    }

    for (int k=0;k<size;k++)
    {
        double min = 0;

        // find the minimum product of alpha, gamma, beta
        for (int u=0;u<2;u++)
        for (int state=0;state<numstates;state++)
        {
            double temp=a[1][k][state]*gammaE[1][u][k][state]*b[1][k][tostate[u][state]];

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

        double topbottom[2];

        for (int u=0;u<2;u++)
        {
            topbottom[u]=0.0;

            for(int state=0;state<numstates;state++)
            topbottom[u] += (a[1][k][state]*gammaE[1][u][k][state]*b[1][k][tostate[u][state]]);
        }

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

        L[1][k] = (log(topbottom[1]/topbottom[0]));
    }

    deinterleavedouble(mesg,size);
    deinterleavedouble(L[1],size);
    // get L[0] back to normal after decoder 2
    deinterleavedouble(L[0],size);

    bool temp=true;
    for (int k=0;k<size;k++)
        if (boolmesg[k] != ((Lc*mesg[k] + L[0][k] + L[1][k] > 0.0) ? true : false))
            temp = false;

    // we can quit prematurely since it has been decoded
    if (temp==true)
    {
        returnvalue = c;
        c=ITERATIONS;
    }
}
// end decoder 2

```

```

// make decisions
for (int k=0;k<size;k++)
    if ((Lc*mesg[k] + L[0][k] + L[1][k]) > 0)
        mesg[k] = 1.0;
    else
        mesg[k] = -1.0;

    return returnvalue;
}

void deinterleavedouble(double *mesg, unsigned size)
{
    double *temp;

    temp = new double[size];

    for (int x=0;x<size;x++)
        temp[x] = mesg[x];

    for (int x=0;x<size;x++)
        mesg[deinterleavearray[x]] = temp[x];

    delete temp;
}

void interleavedouble(double *mesg, unsigned size)
{
    double *temp;

    temp = new double[size];

    for (int x=0;x<size;x++)
        temp[x] = mesg[x];

    for (int x=0;x<size;x++)
        mesg[interleavearray[x]] = temp[x];

    delete temp;
}

void interleave(bool *mesg,unsigned size)
{
    bool *temp;

    temp = new bool[size];

    for (int x=0;x<size;x++)
        temp[x] = mesg[x];

    for (int x=0;x<size;x++)
        mesg[interleavearray[x]] = temp[x];

    delete temp;
}

```

```

void deinterleave(bool *mesg,unsigned size)
{
    bool *temp;

    temp = new bool[size];

    for (int x=0;x<size;x++)
        temp[x] = mesg[x];

    for (int x=0;x<size;x++)
        mesg[deinterleavearray[x]] = temp[x];

    delete temp;
}

void createinterleave(unsigned size)
{
    bool *yesno;

    yesno = new bool[size];

    for (int x=0;x<N;x++)
        yesno[x]=false;

    // create an interleave array
    for (int x=0;x<N;x++)
    {
        unsigned val;

        do
        {
            val=before.longrandom(N);
        }
        while(yesno[val] == true);

        yesno[val] = true;
        interleavearray[x] = val;
        deinterleavearray[val] = x;
    }

    delete yesno;
}

void encode(bool *mesg,bool *parity,unsigned size, bool force)
{
    unsigned state=0;

    for (int x=0;x<size;x++)
    {
        // force the encoder to zero state at the end
        if (x>=size-memory && force)
        {
            if (tostate[0][state]&1)
                mesg[x] = true;
            else

```



```

    mesg[x] = false;
}

// can't assume the bool type has an intrinsic value of 0 or 1
// may differ from platform to platform
int uk = mesg[x] ? 1 : 0;

// calculate output due to new mesg bit
parity[x] = output[uk][state];
// calculate the new state
state = tostate[uk][state];
}
}

bool add(bool a, bool b)
{
    return a==b ? false : true;
}

double gaussian(double variance)
{
    // static because we don't want to have it initialized each time we go in
    double returnvalue=0;
    double k;

    k = sqrt(variance/2.0);

    // add 24 uniform RV to obtain a simulation of normality
    for (int x=0;x<24;x++)
        returnvalue += before.doublerandom();

    return k*(returnvalue-0.5*24);
}

```

Random.cpp

```

#include "random.h"
/*
Long period (221 - 1) random number generator of L'Ecuyer with Bays-Durham shuffle
and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (exclusive of
the endpoint values). Call with idum a negative integer to initialize; thereafter, do not alter
idum between successive deviates in a sequence. RNMIX should approximate the largest floating
value that is less than 1.
--

```

```

*/
double Random::ran2()
{
    int j;
    long k;
    double temp;

    k=(idum)/IQ1;
    idum=IA1*(idum-k*IQ1)-k*IR1; // Compute idum=(IA1*idum) % IM1 without overflows by Schrage's
method.
    if (idum < 0)
        idum += IM1;
    k=idum/IQ2;
    idum2=IA2*(idum-k*IQ2)-k*IR2;    // Compute idum2=(IA2*idum) % IM2 likewise.
    if (idum2 < 0)
        idum2 += IM2;
    j = iy/NDIV;
    iy=iv[j]-idum2;
    // iy=iv[j]-idum2; // Here idum is shuffled, idum and idum2 are combined to generate output.
    iv[j] = idum;
    if (iy < 1)
        iy += IMM1;
    if ((temp=AM*iy) > RNMX)
        return RNMX;    // Because users don't expect endpoint values.
    else
        return temp;
}

void Random::init(long seed)
{
    idum2=123456789;
    idum=0;
    iy=0;

    if (seed != 0)
        idum = seed;
    else
        idum = 1;

    for (int j=NTAB+7;j>=0;j--) // Load the shuffle table (after 8 warm-ups).
    {
        long k=(idum)/IQ1;

        idum=IA1*(idum-k*IQ1)-k*IR1;
        if (idum < 0)
            idum += IM1;
        if (j < NTAB)
            iv[j] = idum;
    }
    iy=iv[0];
}

Random::Random(long seed)
{

```

```

init(seed);
}

Random::Random()
{
    time_t t;

    time(&t);

    init((long)t);
}

double Random::doublerandom()
{
    double t = ran2();
    return t;
}

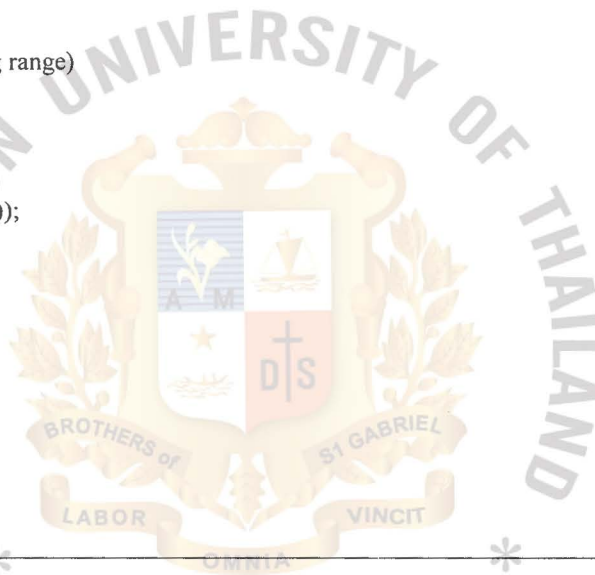
long Random::longrandom(long range)
{
    double t;

    t = doublerandom();
    return((long)(t*(double)range));
}

bool Random::boolrandom()
{
    double t=doublerandom();

    if (t>0.5)
        return true;
    else
        return false;
}

```



* มหาวิทยาลัยอัสสัมชัญ *
SINCE 1969

Random.h

```

#include <stdlib.h>
#include <stdio.h>
#include <values.h>
#include <math.h>
#include <time.h>
#include <limits.h>
#ifndef RANDOM_H
#define RANDOM_H
#define IM1 2147483563
#define IM2 2147483399

```



```

#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS MINDOUBLE
#define RNMX (1.0-EPS)

```

// a uniform random number generator between zero and 1.

```
class Random
```

```
{
```

```
    long idum2;
```

```
    long idum;
```

```
    long iy;
```

```
    long iv[NTAB];
```

```
    unsigned memory;
```

```
    void init(long seed);
```

```
    double ran2();
```

```
public:
```

```
    Random(long seed);
```

```
    Random();
```

```
    double doublerandom();
```

```
    long longrandom(long range);
```

```
    bool boolrandom();
```

```
};
```

```
#endif // RANDOM_H
```



