SOME STUDIES ON TRELLIS AND TURBO CODES (IMPLEMENTATION IN C)

A Dissertation
Submitted in partial fulfillment
of the requirement for the award of the Degree of

MASTER OF ENGINEERING

In

ELECTRONICS AND COMMUNICATION

Submitted by

PARMOD KUMAR
College Roll No. (19/E&C/03)
Delhi University Roll No. 3105

Under the guidance of
Dr. ASOK DE

Department of Electronics and Communication Engineering

Delhi College of Engineering,

University of Delhi

2003-2005

DEPARTMENT OF ELECTRONICS & COMMUNICATION

DELHI COLLEGE OF ENGINEERING, DELHI-42

CERTIFICATE

This is to certify that the Thesis entitled

“Some Studies on Trellis and Turbo Codes (Implementation In C)”

is being submitted by **Parmod Kumar**, Class Roll no. 19/E&C/03, University Roll no. 3105 in partial fulfillment for the award of “Master of Engineering Degree in Electronics & Communication” in Delhi College of Engineering, Delhi University, Delhi is the original work carried out by him under my guidance and supervision. The matter contained in this thesis has not been submitted elsewhere for award of any other degree.

Dr. A. Bhattacharyya
Prof. & Head
Deptt. of E&C Engg.
Delhi College of Engg.
Delhi-110042

Dr. Asok De
Prof. & Head
Deptt. of IT Engg
Delhi College of Engg.
Delhi-110042

ACKNOWLEDGEMENT

Firstly, I would like to express my heartily gratitude and thanks to my project guide Prof. Dr. Asok De, Head Deptt of IT Engineering, DCE Delhi for continuous inspiration, encouragement and guidance in every stage of preparation of this Thesis work.

I would also like to thank Prof. Asok Bhattacharyya, Head Deptt. of Electronics and Communication Engineering, DCE Delhi, and Mrs. Rajesvari Pandey Lecturer Deptt of Electronics and Communication Engineering, DCE Delhi for the support provided by them during the entire duration of degree and especially in this Thesis.

Lastly, I am thankful to all non-teaching staff, especially Mr. M.L Chandna, who have helped me directly or indirectly in the completion of this Thesis report.

Parmod Kumar
ME (E&C)
Univ. Roll no. 3105
Class Roll no. 19/E&C/03

ABSTRACT—The main objective of this thesis is to study and implement two of the advanced and latest channel decoding algorithms and compare their performance. Although for implementing these decoding algorithm, the algorithms for channel encoder and AWGN channel (for adding noise) are developed as well. But our main objective is to study, analyze and compare the performance of decoding algorithms. The two decoding algorithms developed and implemented are (i) A Simplified Trellis-Based Decoder and (ii) Log-MAP-Based Iterative Turbo Decoder with reduced storage requirements.

In “A Simplified Trellis-Based Decoder” a simplified branch metric and add-compare-select (ACS) unit is presented for use in trellis-based decoding architecture. This simplification is based on a complementary property of some feed forward encoders. As a result, one adder is saved in every other ACS unit. Hence only half the branch metrics have to be calculated.

In “Log-MAP-Based Iterative Turbo Decoder” efforts are made to reduce the memory requirements for implementing the algorithm, although that is achieved at the cost of degraded speed performance. Also the odd-even symmetric interleaver structure used here for implementing turbo code is implemented with reduced storage memory requirements.

For implementing the above algorithms an intense study of various decoding algorithms is done, and then algorithms are developed. Afterwards these algorithms are implemented in C language. Results are produced in form of text files, after executing the programs for encoder, AWGN channel and decoder in the sequence.

After getting the results in form of text files, the decoded output is compared with the original file and bit error probability (P_B) is calculated for different values of E_b/N_0. Finally P_B versus E_b/N_0 is plotted for above algorithms and compared. At E_b/N_0 of 2 dB P_B for A simplified trellis based decoder is 2×10^{-2} and for Log-MAP-Based turbo decoder P_B of 6×10^{-4} is obtained for a text file containing 3000 bit values. The error performance of Log-MAP-Based Turbo decoder is found to be better than the A simplified trellis based decoder but the speed of later is better than the former.

CONTENTS

CHAPTER 1
INTRODUCTION

1.1 INTRODUCTION

The objective of this thesis is to study and implement two of the advanced and latest channel decoding algorithms and compare their performance. Although for implementing these decoding algorithm, the algorithms for channel encoder and AWGN channel (for adding noise) are developed as well. But our main objective is to study, analyze and compare the performance of decoding algorithms. The first algorithm implemented here is “A Simplified Trellis-Based Decoder” which is modified version of standard Viterbi Decoder. The Viterbi decoding algorithm was discovered and analyzed by Viterbi in 1967. The Viterbi algorithm essentially performs maximum likelihood decoding [chapter 3(3.4)]. The second algorithm implemented is “Log-MAP-Based Iterative Turbo Decoder” with reduced memory requirements which is based on standard MAP (maximum a posteriori) algorithm. The process of turbo code decoding starts with the formation of a posteriori probabilities (APP) for each data bit, which is followed by choosing the data bit value that corresponds to the maximum a posteriori (MAP) probability for that data bit. Upon reception of a corrupted code-bit sequence, the process of decision making with APPs, allows the MAP algorithm to determine the most likely information bit to have been transmitted at each bit time [chapter 5]. This is unlike the Viterbi algorithm (VA), where the APP for each data bit is not available.

1.2 GOAL OF THESIS

The goal of thesis is to implement some channel decoding algorithms that are saving some sort of resources or efforts. As will be clear from the text below, that we are saving some hardware/computations in case of first algorithm (Trellis-Based) and we are saving memory in case of second algorithm (Turbo-Based).

In “A Simplified Trellis-Based Decoder” a simplified branch metric and add-compare-select (ACS) unit is presented for use in trellis-based decoding architecture. This simplification is based on a complementary property of some feed forward encoders [chapter 4]. As a result, one adder is saved in every other ACS unit. Hence only half the branch metrics have to be calculated. It is also shown that this simplification becomes especially beneficial for rate ½ convolutional codes. Consequently, area and power consumption will be reduced in a hardware implementation.

In “Log-MAP-Based Iterative Turbo Decoder” with reduced storage requirements efforts are made to reduce the memory requirements for implementing the algorithm [chapter 6]. But that is achieved at the cost of degraded speed performance. Also the odd-even symmetric interleaver is implemented with reduced storage requirements. The idea used here is that the text file, which we want to send, can be broken down into smaller files (typically 3000 –5000 bits), which are then sent serially one by one. The time taken is increased but by doing so we are able to save huge amount of memory.

1.3 PROCEDURE FOLLOWED

For developing and implementing the above algorithms an intense study of various decoding algorithms and related topics is done from the various books and IEEE Transactions and IEEE communication letters. The three papers, which are used a lot in this thesis work [1], [2], [3], are given below:

(i) VHDL Implementation of a Turbo Decoder With Log-MAP-Based Iterative Decoding; By Yanhui tong, Tet-Hin Yeap, Member, IEEE, and Jean-Yves Chouinard, Senior Member, IEEE; IEEE Transaction on Instrumentation and Measurement, Vol. 53, No. 4, August 2004.

(ii) Interleaver Structure for Turbo Codes with Reduced Storage memory Requirement; By Johan Hokfelt, Ove Edfors and Torleiv Maseng; Department of Applied Electronics, Lund university, Lund, Sweeden.

(iii) A Simplified computational Kernel for Trellis-Based Decoding; By Matthias Kamuf, Student Member, IEEE, John B. Anderson, Fellow, IEEE, and Viktor Owall, Member, IEEE; IEEE Communication Letters, Vol. 8,No. 3, March 2004.

After studying the material the algorithms are written, and then code is written in language C. The C code is written for the channel encoder, AWGN channel and the channel decoder for both of the algorithms. The efforts are being made to generate standard AWGN noise but due to limitation of C language we are able to generate noise, which is very much similar, as shown in Graph 1 (Chapter 7). The idea used here for plotting P_B versus E_b/N_0 is that we introduce more error in the channel if we want to plot for lower value of $E_b/N_{0,}$ and vice versa because as we know, if E_b/N_0 decreases the noise increases and vice versa.

These C programs are then executed in a particular sequence and results are obtained in form of text files. Again a C program is executed for comparing these text files and results are obtained which are put in form of a table manually. The tabular results are then used for drawing the graph. The graphs thus obtained are found to be in close approximation to the graphs shown in books and published papers [4].

1.4 ORGANIZATION OF THESIS

The whole thesis work is divided into 7 chapters. The first chapter is about the overview of the thesis work. It includes the thesis subject, thesis goals, procedure followed and organization of thesis. The second chapter is about introduction to digital communication system, channel coding that is what is channel coding? Why we use channel coding? Advantages of channel coding and types of channel coding.

The third chapter is about convolutional coding, convolutional decoding terminology and Viterbi decoding algorithm. This chapter forms the basis for next chapter. The fourth chapter is concerned with the first algorithm implemented here. It describes how we are able to save hardware if simulated on hardware kit and computation if implemented in C. it describes the complementary property of some of the feed forward encoders and its affect on BM and ACS unit.

The fifth chapter is concerned is about the turbo codes, its terminology and standard turbo decoder that is the MAP decoder. This chapter forms the basis for the next chapter. The chapter sixth is concerned with the second algorithm implemented in this thesis. This chapter briefly describes the various SISO algorithms and compares their performance and chooses the optimum one i.e. Log-MAP Decoder, which is approximately as efficient as MAP but requires simpler and lesser no of computations. It also describes the interleaver structure with reduced storage requirements used for Log-MAP-Based Iterative Turbo Decoder.

The chapter seven is about the results. The results are shown in three different formats. The first format of results is in the form of text files. It shows the original file, which is sent through the channel, the decoded file and the file, which would have been received if sent through the AWGN, channel without encoding. This form of result gives the user a visual look, how decoding algorithms are able to reduce the error. The second format of result is in form of tables. This form of results shows the no of errors present originally and after applying decoding algorithm. The third format of result is in form of graphs. This type of format is necessary for analysis of results. These graphical results are used for comparison purposes. The first graph shows the comparison of standard AWGN noise and the noise generated using C language. The second graph shows the performance of “Log-MAP-Based Iterative Turbo decoder” for two iterations, after iteration second no performance improvement is observed. The third graph shows the performance of “ A Simplified Trellis-Based Decoder”. The fourth graph compares the performance of two decoders described above.

CHAPTER 2

CHANNEL CODING

2. 1 DIGITAL COMMUNICATION SYSTEM

To analyze a Digital Communication System let us observe the functional elements of the system, as shown in Fig (2.1). The overall purpose of the system is to transmit the message (or sequences of symbols) coming out of a source to a destination point at a high rate and accuracy.

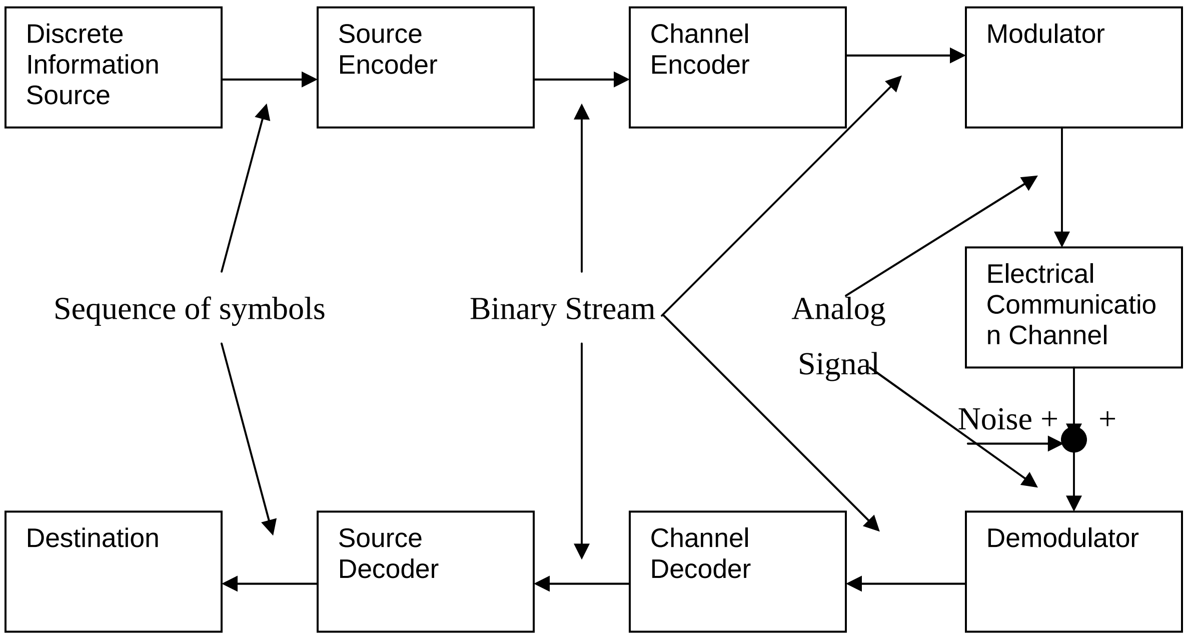


Fig (2.1): Functional Block of a Digital Communication System

The source and the destination point are physically separated in space and a communication channel of some sort connects the source to destination point. Then channel accepts electrical (electromagnetic) signals and the output of the channel is usually a smeared or distorted version of the input due to the non-ideal nature of the communication channel. The smearing and noise introduce errors in the information being transmitted and limits the rate at which information can be communicated from the source to destination. The probability of incorrecting decoding a message symbol at the receiver is often used as a measure of performance of digital communication system. The main function of the coder, the modulator, the demodulator and the decoder is to combat the degrading effects of the channel on the signal and maximize the information rate and accuracy in communication process. *This thesis work is based on the function of three blocks shown above which are: channel encoder, communication channel, and channel decoder.*

2.2 WHAT IS CHANNEL CODING?

Channel coding refers to the part of signal transformations designed to improve communications performance by enabling the transmitted signals to better withstand the effects of various channel impairments, such as noise, interference, and fading. These signal-processing techniques can be thought of as vehicles for accomplishing desirable system trade-offs (e.g., error-performance versus bandwidth, power versus bandwidth). The channel coding has become a very popular way to bring these beneficial effects. The use of large-scale integrated circuits (LSI) and high-speed digital signal processing (DSP) techniques have made it possible to provide as much as 10 DB performance improvement through these methods, at much less cost than through the use of most other methods such as high power transmitters or large antennas.

Channel coding can be partitioned into two study areas, waveform coding and structured sequences. *Waveform coding* deals with transforming waveforms into “better waveforms” to make the detection process less subject to errors. *Structured sequences* deals with transforming data sequences into “better sequences” having structured redundancy (redundant bits). The redundant bits can then be used for the detection and correction of errors. The encoding procedure provides the coded signal (whether waveforms or structured sequences) with better distance properties than those of their un-coded counterparts. This thesis work is based on structured sequences.

2.3 ADVANTAGES OF CHANNEL CODING

- **Error Performance Versus Bandwidth**

  Using channel coding one can obtain the better error performance for the same value of E_b/N_0 but the price paid is the increased bandwidth along with the new components (encodes and decoders).

- **Power Versus Bandwidth**

  This is a trade-off in which the same quality of data is achieved, but the coding allows for a reduction in power or $E_b/N_{0.}$ The price paid is additional circuitry (encoders and decoders).

- **Data Rate Versus Bandwidth**

  In uncoded system increasing the data rate leads to degraded quality of data because power requirement is inversely proportional to the rate. But the use of error-correction coding brings back the same quality at the same power level. The price paid is same as in first trade-off.

- **Capacity Versus Bandwidth**

  In CDMA, where users simultaneously share the same cell or nearby cells, the capacity (maximum number of users) per cell is inversely proportional to E_b/N_0. By using channel coding one can lower E_b/N_0 for the same error performance hence more capacity; the code achieves a reduction in each user’s power, which in turn allows for an increase in the number of users. Price paid is same as in first trade-off.

2.4 WAVEFORM CODING

Waveform coding procedures transform a waveform set (representing a message set) into an improved waveform set. The improved waveform set can then be used to provide improved P_B (probability of bit error) compared to the original set. The most popular of such *waveform codes* are referred to as *orthogonal* and *bi-orthogonal* codes. The encoding procedure endeavors to make each of the waveforms in the coded signal set as unlike as possible; the goal is to render the cross-correlation coefficient Zij among all pairs of signals as small as possible. The smallest possible value of cross correlation coefficient occurs when the signals are anti-correlated (Zij = -1); however this can be achieved only when number of symbols in the set is two and the symbols are antipodal. In general, it is possible to make all the cross-correlation coefficients equal to zero. The set is then said to be orthogonal.

The cross-correlation between two signals is a measure of the distance between the signal vectors. The smaller the cross-correlation, the more distant are the vectors from each other.

2.5 STRUCTURED SEQUENCES

In case of orthogonal M-ary signaling, we can decrease P_B by increasing M. The major disadvantage with such orthogonal coding techniques is the associated inefficient use of bandwidth. For an orthogonally coded set of $M=2^k$ waveforms, the required transmission bandwidth is M/k times that needed for the uncoded case. The structured sequence can be thought of as a process of inserting structured redundancy into the source data so that the presence of errors can be detected or the errors corrected. The structured sequences can be partitioned into various sub-categories; these are *block*, *Cyclic*, *convolutional*, and *turbo.* The block and cyclic codes were studied and implemented in minor project, now in thesis the objective is to study and implement latest and modified version of existing Trellis (Convolutional) and Turbo codes.

CHAPTER 3
CONVOLUTIONAL CODING

3.1 INTRODUCTION

The linear block codes are described by two integers, n and k, and a generator matrix or polynomial. The integer k is the number of data bits that form an input to a block encoder. The integer n is the total number of bits in the associated codeword out of encoder. A characteristic of linear block code is that each codeword n-tuple is uniquely determined by the input message k-tuple. The ratio k/n is called the rate of the code - a measure of the amount of added redundancy. A *convolutional code* is described by three integers, n, k, K, where the ratio k/n has the same code rate significance (information per coded bit) that it has for block codes; however, n does not define a block or codeword length as it does for block codes. The integer K is a parameter known as the constraint length; it represents the number of k-tuple stages in the encoding shift register. An important characteristic of convolutional codes, different from block codes, is that the encoder has memory, the n – tuple emitted by the convolutional encoding procedure is not only a function of an input k – tuple but is also a function of the previous K – 1 input k-tuples. In practice, n and k are small integers and K is varied to control the capability and complexity of code.

3.2 CONVOLUTIONAL ENCODING

The input message source is denoted by the sequence $m = m_1, m_2, \ldots.., m_i, \ldots\ldots,$ where each m_i represents a binary digit (bit), and i is the time index. We shall assume that each m_i is equally likely to be a one or zero, and independent from digit to digit. Being independent, the bit sequence lacks any redundancy; that is, knowledge about m_i gives no information about m_j $(i \neq j)$. The encoder transforms each sequence m into a unique codeword sequence U = G(m). Even though the sequence m uniquely defines the sequence U, a key feature of convolutional code is that a given k-tuple with in m does not uniquely define its associated n-tuple with in U since the encoding of each k-tuple is not only a function of that k-tuple but is also a function of the K-1 input k-tuples that precede it. The sequence U can be partitioned into a sequence of branch words: U = U_1, U_2, …., U_i,….. Each branch word U_i is made up of binary code symbols, often called channel symbols, channel bits or code bits; unlike the input message bit the code symbols are not independent.

In typical communication application, the codeword sequence U modulates a waveform s(t). During transmission, the waveform s(t) is corrupted by noise, resulting in a received waveform s'(t) and a demodulated sequence Z = Z_1, Z_2, …., Z_i, ….. The task of decoder is to produce an estimate m' = m'_1, m'_2, …, m'_i,….. of the original message sequence using the received sequence Z together with a priori knowledge of the encoding procedure.

A general convolutional encoder is shown in Fig (3.1) is mechanized with a kK- stage shift register and n modulo-2 adders, where K is the constraint length. The constraint length represents the number of k-bit shifts over which a single information bit can influence the encoder output. At each unit of time, k bits are shifted into the first k stages of the register; all bits in the register are shifted k stages to the right, and the outputs of the n adders are sequentially sampled to yield the binary code symbols or code bits. These code symbols are then used by the modulator to specify the waveforms to be transmitted over the channel. Since there are n code bits or each input group of k message bits, the code rate is k/n message per code bit, where k < n.

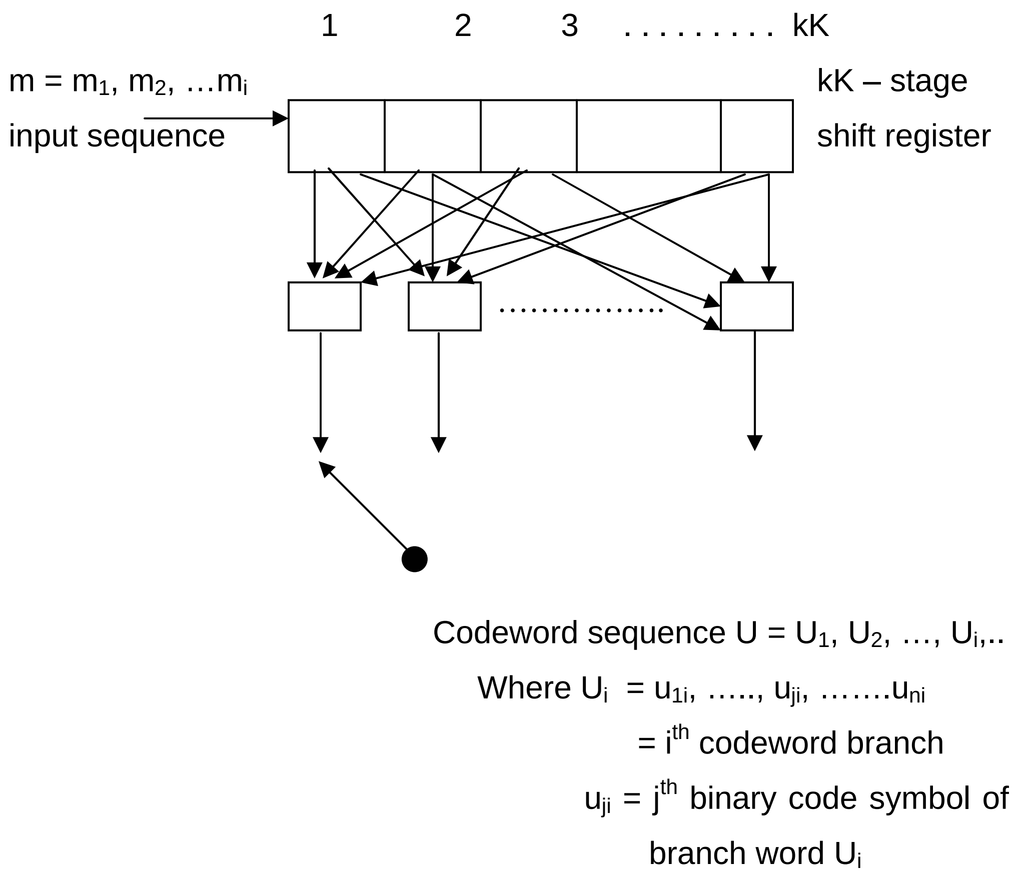

1 2 3 . . . . kK
m = m1, m2, …mi
input sequence
kK – stage
shift register
……………
Codeword sequence U = U1, U2, …, Ui,..
Where Ui = u1i, ….., uji, …….uni
= ith codeword branch
uji = jth binary code symbol of
branch word Ui

Fig (3.1): Convolutional encoder with constraint length K and rate k/n

We shall consider the most commonly used binary convolutional encoders with K =1 that is, those encoders in which the message bits are shifted into encoder one bit at a time, although generalization to higher order alphabets is straightforward. For the K = 1 encoder, at the ith unit of time, the message bit m_i is shifted into the first shift register stage; all the previous bits in the register are shifted one stage to the right, and as in the more general case, the outputs of the n adders are sequentially sampled and transmitted. Since there are n code bits for each message bit, the code rate is 1/n. the n code symbols occurring at time t_i comprise the ith branch word, Ui = u_{1i}, u_{2i}, ….., u_{ni}, where u_{ji} (j = 1,2 ,…n) is the jth code symbol belonging to the ith branch word. Note that for the rate 1/n encoder, the kK stage shift register can be referred to simply as a K- stage register, and the constraint length K, which was expressed in units of k-tuple stages can be referred to as constraint length in units of bits.

3.3 CONVOLUTIONAL ENCODER REPRESENTATION

To describe a convolutional code, one needs to characterize the encoding function G(m), so that given an input sequence m, one can readily compute the output sequence U. several methods are used for representing a convolutional encoder [5], the most popular being the connectional pictorial as shown in Fig (3.1), connection vectors or polynomials, the state diagram, the tree diagram and the trellis diagram. We will discuss in detail the trellis diagram and others in brief.

3.3.1 Connection Representation

We shall use the convolutional encoder, shown below in Fig (3.2) as a model for discussing convolutional encoders. This figure illustrates a (2, 1) convolutional encoder with constraint length K = 3. There are n = 2 modulo – 2 adders; thus the code rate k/n is ½. At each input bit time, a bit is shifted into the leftmost stage and the bits in the register are shifted one position to the right. Next, the output switch samples the output of each modulo – 2 adders (i.e., first the upper adder, then the lower adder), thus forming the code symbol pair making up the branch word associated with the bit just inputted. The sampling is repeated for each inputted bit. The choice of connections between the adders and the stages of the register gives rise to the characteristics of the code. Any change in the choice of connections results in different code. The connections are of course, not chosen or changed arbitrarily.

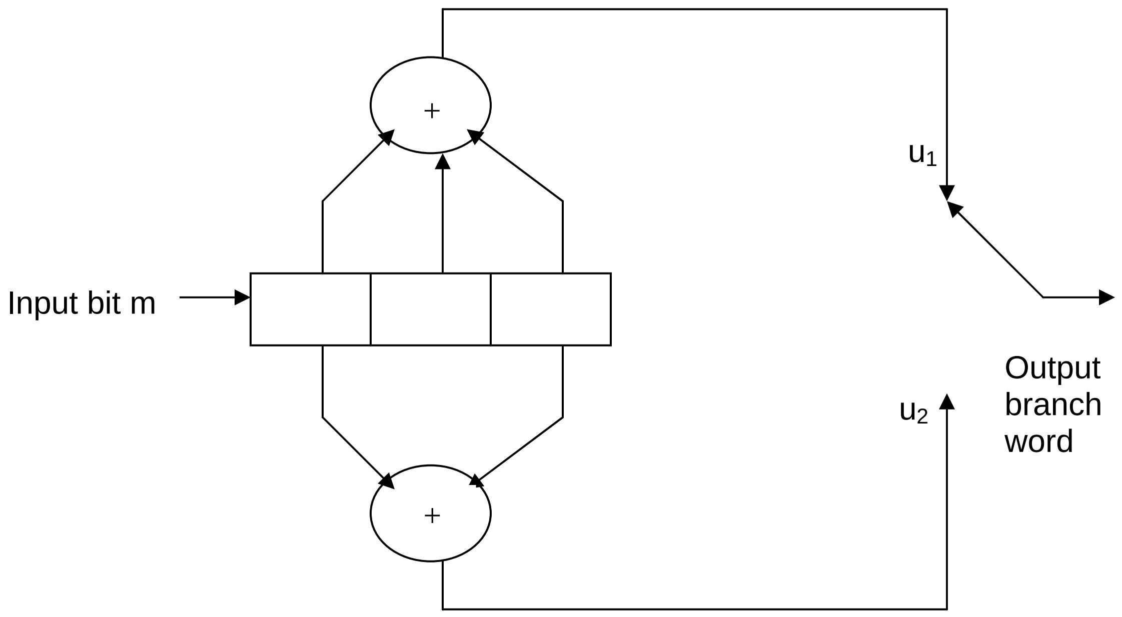


Fig (3.2) Convolutional encoder (rate ½, K = 3).

Unlike a block code that has a fixed word length n, a convolutional code has no particular block size, however, convolutional codes are often forced into a block structure by periodic truncation [21]. This requires a number of zero bits to be appended to the end of the input data sequence, for the purpose of clearing or flushing the encoding shift register of the data bits.

3.3.2 Polynomial Representation

Sometimes, the encoder connections are characterized by generator polynomial. We can represent a convolutional encoder with a set of n generator polynomial, one for each of the n modulo-2 adders. Each polynomial is of degree K-1 or less and describes the connection of the encoding shift register to that modulo-2 adder, much the same way that a connection vector does. The coefficients of each term in the (K-1) degree polynomial are either 1 or 0, depending on whether a connection exists or does not exist between the shift register and modulo-2 adder in question. For the encoder example in Fig (3.2), we can write the generator polynomial g_1 (X) for the upper connection and g_2 (X) for the lower connection as follows:

$$g_1(X) = 1+X+X^2$$
$$g_2(X) = 1+X^2$$

3.3.3 State Representation and State Diagram

A convolutional encoder belongs to a class of devices known as finite state machines, which is the general name given to machines that have a memory of past signals. In the most general sense, the state consists of the smallest amount of information that, together with a current input to the machine, can predict the output of the machine. A future state is restricted by the past state. For a rate 1/n convolutional encoder, the state is represented by the contents of the rightmost K-1 stages. Knowledge of the state together with knowledge of the next input is necessary and sufficient to determine the next output.

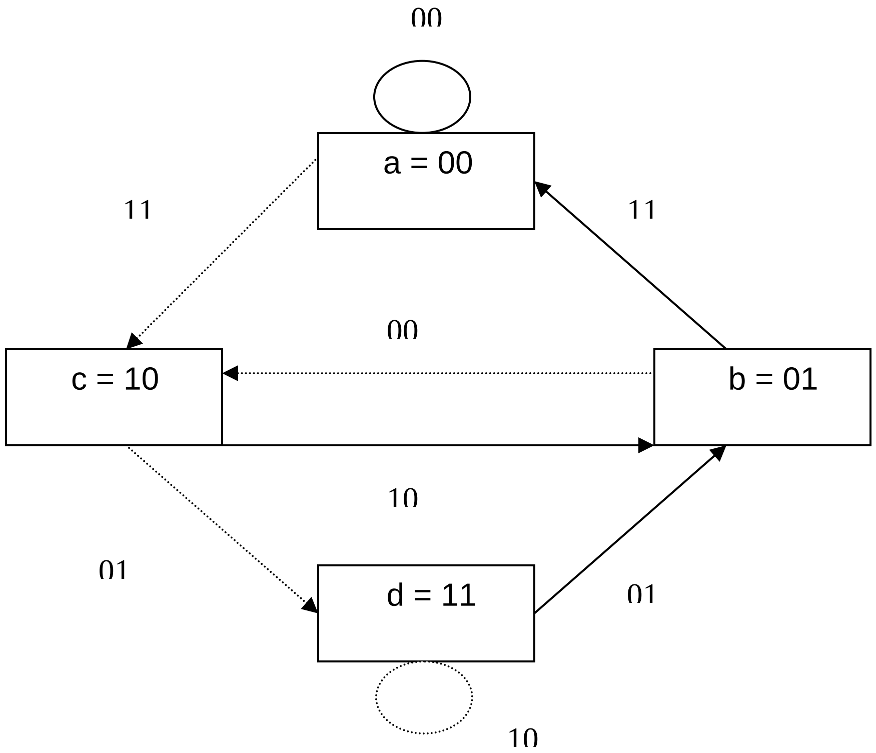


Fig (3.3): Encoder state diagram (rate ½, K=3)

The states of the register are designated a = 00, b = 01, c = 10, d = 11; the diagram shown in the Fig (3.3) illustrates all the state transitions that are possible for the encoder in the Fig (3.2). There are only two transitions emanating from each state, corresponding to the two possible input bits. Next to each path between states is written the output branch word associated with the state transition. In drawing the path, we use the convention that a solid line denotes a path associated with an input bit, zero, and a dashed line denotes a path associated with an input bit, one.

3.3.4 The Tree Diagram

The *tree diagram* for the convolutional encoder shown in Fig (3.2) is shown in Fig (3.4) shown below. At each successive input bit time the encoding procedure can be described by traversing the diagram from left to right, each tree branch describing an output branch word. The branching rule for finding a codeword sequence is as follows: if the input bit is zero, its associated branch word is found by moving to the next rightmost branch in the upward direction. If the input bit is a one, its branch word is found by moving to the next rightmost branch in the downward direction. Assuming that the initials contents of the encoder is all zeros, the diagram shows that if the first input bit is zero, the output branch word is 00 and, if the input bit is a one, the output branch word is 11. Similarly, if the first input bit is one and the second input bit is zero, the second output branch word is 10. Or, if the first input bit is a one and the second input bit is also a one, the second output branch is 01. Following this procedure we see that the input sequence 1 1 0 1 1 traces the heavy line drawn on the tree diagram in Fig (3.4). This path corresponds to the output codeword sequence 1 1 0 1 0 1 0 0 0 1.

The added dimension of time in the tree diagram allows one to dynamically describe the encoder as a function of a particular input sequence. However we see a major problem in trying to use a tree diagram for describing a sequence of any length. The number of branches increases as a function of 2^L, where L is the number of branch words in the sequence. We would quickly run out of paper and patience.

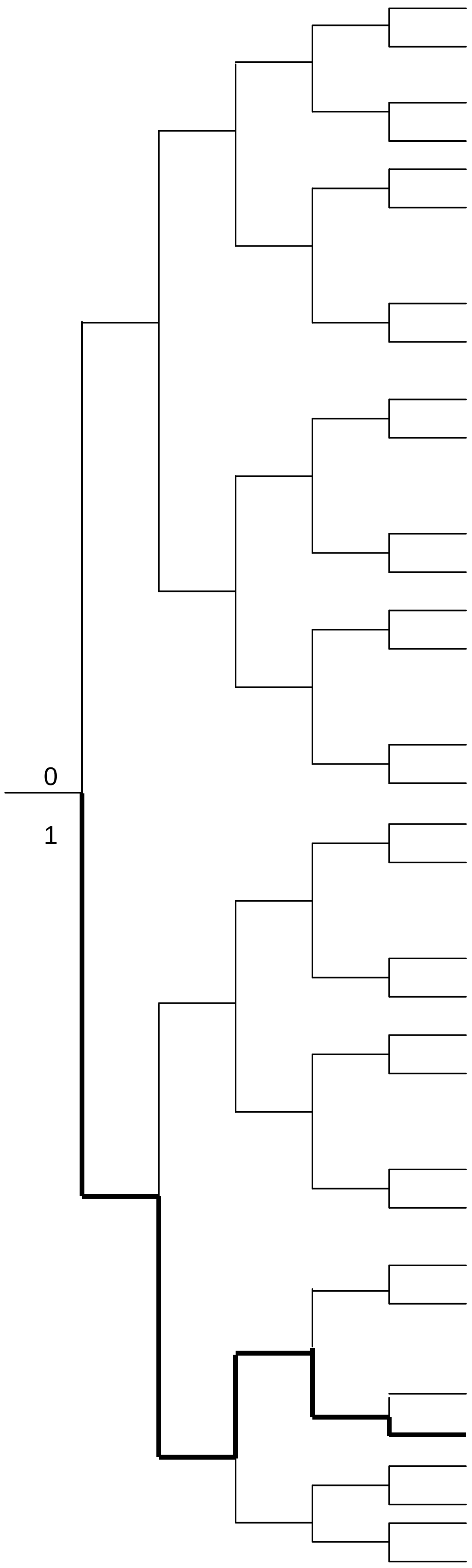
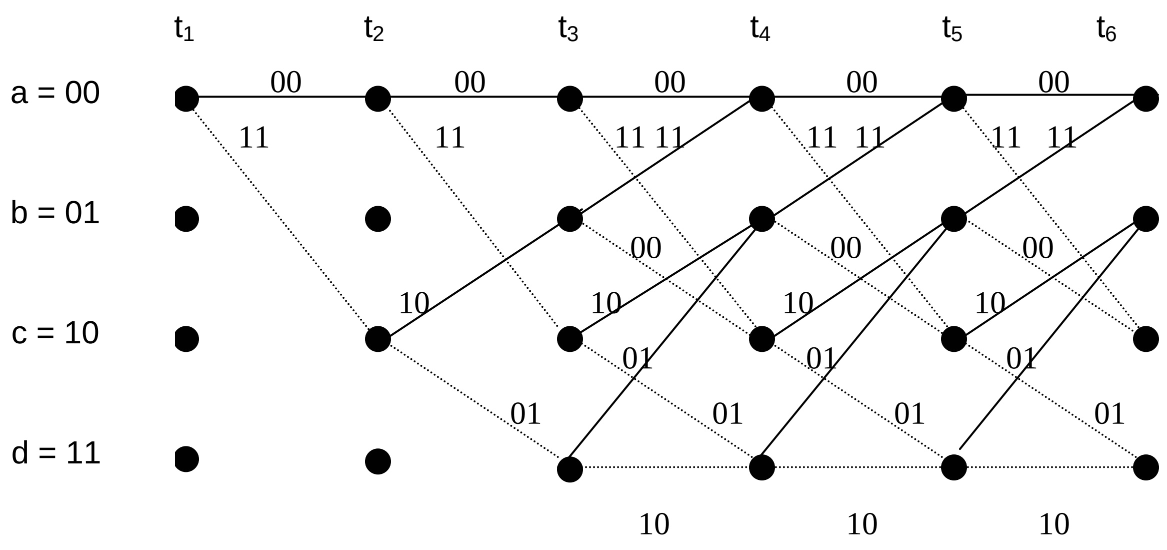


Fig (3.4) Tree representation of encoder (Rate ½, K = 3).

3.3.5 The Trellis Diagram

Observation of Fig (3.4), tree diagram shows that for this example, the structure repeats itself at time t_4, after the third branching (in general, the tree structure repeats itself after K branching, where K is the constraint length). We label each node in the tree diagram of Fig (3.4) to correspond to the four possible states in the shift register, as follows: a = 00, b = 01, c = 10, d = 11. The first branching of the tree structure, at time t_1, produces a pair of nodes. At each successive branching the number of nodes gets doubled. The second branching, at time t_2, results in four nodes. After the third branching, there are total of eight nodes. We can see that all branches emanating from two nodes of the same state generate identical branch word sequences. From this point on, the upper and the lower halves of the tree are identical. The reason for this should be obvious from examination of the encoder in Fig (3.2). As the fourth input enters the encoder on the left, the first input bit is rejected on the right and no longer influences the output branch words. Consequently, the input sequences 1 0 0 x y ….. and 0 0 0 x y ….., where the left most bit is the earliest bit, generate the same branch words after the (K = 3)rd branching. This means that any two nodes having the same state label at the same time t_i can be merged, since all succeeding path will be indistinguishable. If we do this to the tree structure of Fig (3.4), we obtain another diagram, called the trellis diagram. The trellis diagram, by exploiting the repetitive structure, provides a more manageable encoder description than does the tree diagram. Te trellis diagram for the convolutional encoder of Fig (3.2) is shown in Fig (3.5).

In drawing the trellis diagram, we use the same convention that we introduced with the state diagram—a solid line denotes the output generated by the input bit zero, and a dashed line denotes the output generated by an input bit one. The nodes of trellis characterize the encoder states; the first row nodes correspond to the state a = 00, the second and subsequent rows correspond to the states b = 01, c = 10, and d = 11. At each unit of time, the trellis requires 2^{K-1} nodes to represent the 2^{K-1} possible encoder states. The trellis in our example assumes a fixed periodic structure after trellis depth 3 is reached (at time t_4).



Legend

———	Input bit 0
········	Input bit 1

Fig (3.5): Encoder trellis diagram (rate = ½, K = 3)

In the general case the fixed structure prevails after dept K is reached. At each point and thereafter, each of the state can be entered from either of two preceding states. Also, each of the state can transition to one of two states. Of the two outgoing branches, one corresponds to an input bit zero and other corresponds to an input bit one. On Fig (3.5) the output branch words corresponding to the state transitions appear as labels on trellis branches.

3.4 FORMULATION OF THE CONVOLUTIONAL DECODING PROBLEM

3.4.1 Maximum Likelihood Decoding

If all input messages sequences are equally likely, a decoder that achieves the minimum probability of error is one that compares the conditional probabilities, also called the likelihood functions P (Z/U $^{(m)}$), where Z is the received sequence and U $^{(m)}$ is one of the possible transmitted sequences and chooses the maximum. The decoder chooses U $^{(m')}$ if

$$P(Z/U^{(m')}) = \max_{\text{Over all } U^{(m)}} P(Z/U^{(m)}) \tag{3.1}$$

The maximum likelihood concept, as stated above is a fundamental development of decision theory; it is the formalization of a “common-sense” way to make decisions when there is statistical knowledge of the possibilities. In the binary demodulation treatment there are only two equally likely possible signals $s_1(t)$ or $s_2(t)$ that might have been transmitted. Therefore, to make the binary maximum likelihood decision, given a received signal meant only to decide that $s_1(t)$ was transmitted if

$$P(Z/s_1) > P(Z/s_2) \tag{3.2}$$

Otherwise, to decode that $s_2(t)$ was transmitted. However, when applying maximum likelihood to the convolutional decoding problem, we observe that the convolutional code has memory (the received sequence represents the superposition of current bits and prior bits). Thus, applying maximum likelihood to the decoding of convolutional encoded bits is performed in the context of choosing the most likely sequence as shown in (3.1). There are typically a multitude of possible codeword sequences that might have been transmitted. To be specific, for a binary code, a sequence of L branch word is a member of a set of 2^L possible sequences. Therefore, in maximum likelihood context, we can say that the decoder chooses a particular $U^{(m')}$ as the transmitted sequence if the likelihood $P(Z/U^{(m')})$ is greater than the likelihood of all the other possible transmitted sequences. Such an optimal decoder [12], which minimizes the error probability (for the case where all transmitted sequences are equally likely), is known as a *maximum likelihood decoder*. The likelihood functions are given or computed from the specifications of the channel.

3.4.2 Channel Models: Hard Versus Soft Decisions

Before specifying an algorithm that will determine the maximum likelihood decision, let us describe the channel. The channel over which the waveform is transmitted is assumed to corrupt the signal with Gaussian noise. When the corrupted signal is received, it is first processed by the demodulator and then by the decoder.

The demodulator output can be configured in a variety of ways. It can be implemented to make a firm or hard decision at to whether a received signal represents a zero or one, and fed in to the decoder. Since the decoder operates on the hard decision made by the demodulator, the decoding is called *hard-decision decoding*. The demodulator can also be configured to feed the decoder with a quantized value of received signal greater than two levels. When the quantization level of the demodulator output is greater than two, the decoding is called soft-decision decoding. Eight level (3-bits) of quantization are illustrated on the abscissa of Fig (3.6). When the demodulator sends a hard decision to the decoder, it sends a single binary symbol. When the demodulator sends a soft binary decision, quantized to eight levels, it sends the decoder a 3-bit word describing an interval shown in Fig (3.6). In effect, sending such a 3-bit word in place of a single binary symbol is equivalent to sending the decoder a measure of confidence along with code-symbol decision. It should be clear that ultimately, every message decision out of the decoder must be a hard decision.

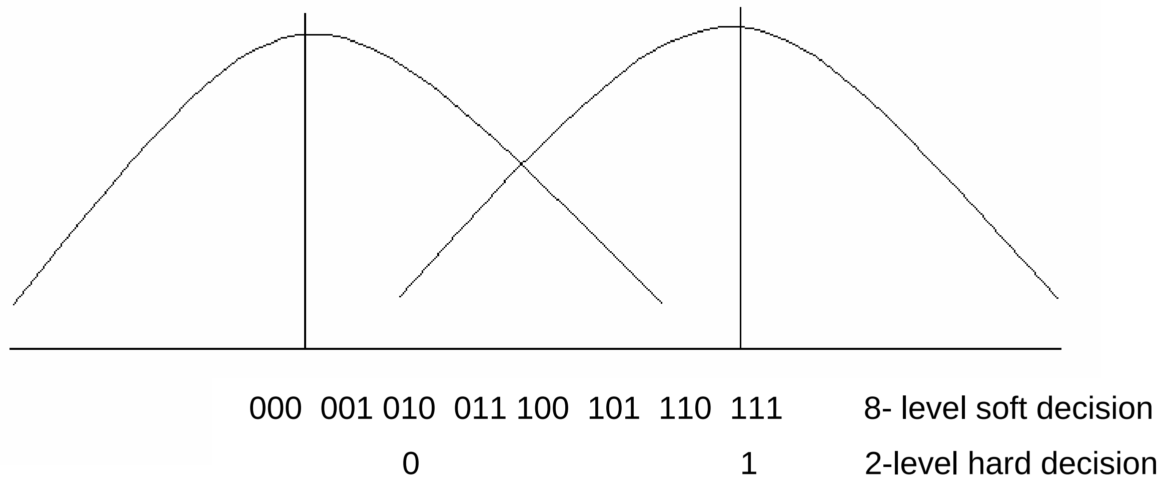


Fig (3.6): Hard and soft decoding decision

For a Gaussian channel, eight level quantization results in a performance improvement of approximately 2 dB in the required signal-to-noise ratio compared to two-level quantization. This means that eight-level-soft decision decoding can provide the same probability of error as that of hard decision decoding, but requires 2 dB less E_b/N_0 for the same performance.

3.4.3 Binary Symmetric Channel

A binary symmetric channel (BSC) is a discrete memory less channel that has binary input and output alphabets and symmetric transition probabilities. It can be described by the conditional probabilities

$$P(0|1) = P(1|0) = p$$
$$P(1|1) = P(0|0) = 1\text{-}p$$

as illustrated in Fig(3.7) below.

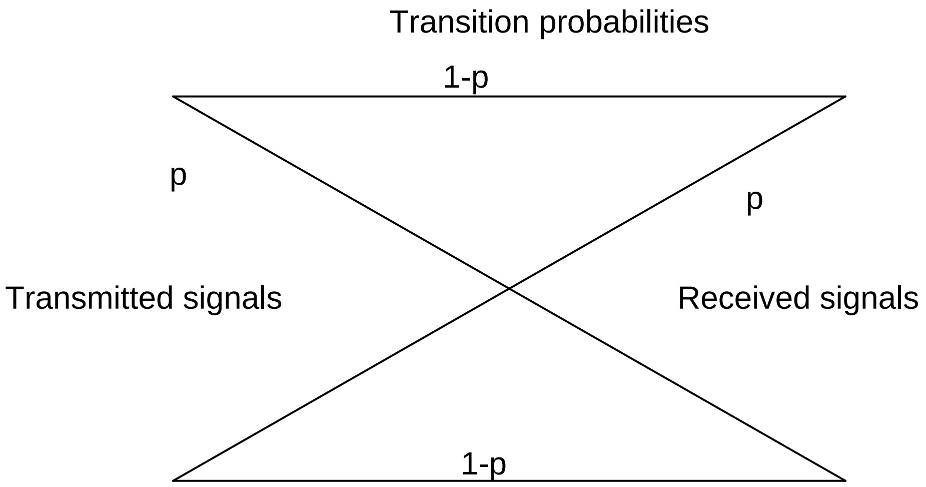


Fig (3.7): Binary symmetric channel (hard-decision channel)

The probability that an output symbol will differ from the input symbol is p, and the probability that the output symbol will be identical to the input symbol is (1-p).

The BSC is an example of hard-decision channel, which means that, even though the demodulator may receive continuous-valued signals, a BSC allows only firm decision such that each demodulator output symbol consists of one of two binary values.

3.5 THE VITERBI CONVOLUTIONAL DECODING ALGORITHM

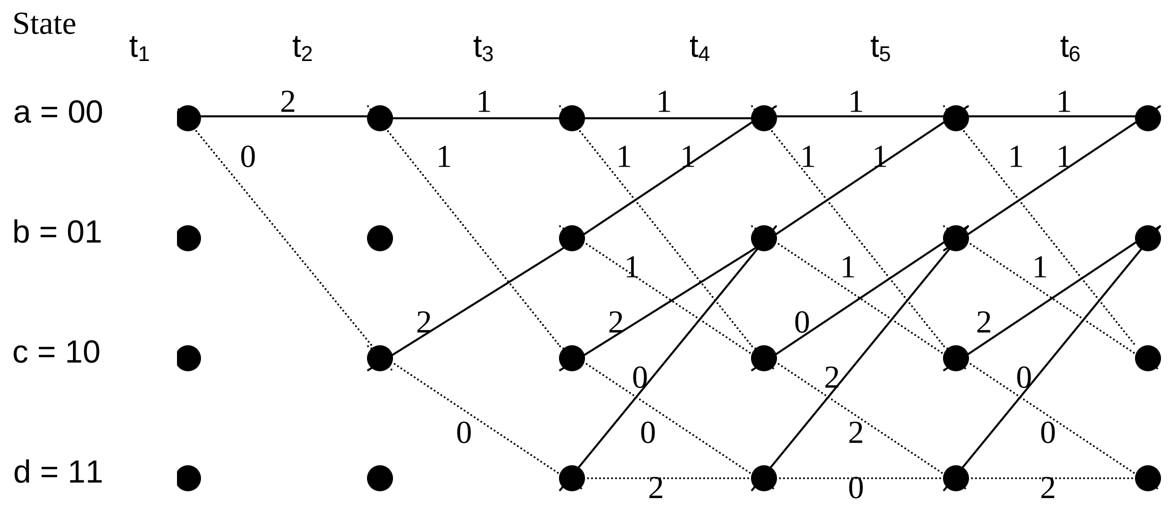
The Viterbi decoding algorithm was discovered and analyzed by Viterbi in 1967. The viterbi algorithm [13], [14], [22] essentially performs maximum likelihood decoding; however it reduces the computational load by taking the advantage of the special structure in the code trellis. The advantage of Viterbi decoding is that the complexity of a Viterbi decoder is not a function of the number of symbols in the codeword sequence. The algorithm involves calculating a measure of similarity, or distance, between the received signal at time t_i and the entire trellis path entering each state at time t_i. The Viterbi algorithm removes from consideration those trellis paths that could not possibly are the candidates for the maximum likelihood choice. When two paths enter the same state, the one having the best metric is chosen; this path is called the surviving path. This selection of surviving paths is performed for all the states. The decoder continues in this way to advance deeper into the trellis, making decisions by eliminating the least likely paths. The early rejection of the unlikely paths reduces the decoding complexity. Note that the goal of selecting the optimum path can be expressed, equivalently, as choosing the codeword with the maximum likelihood metric, or as choosing the codeword with the minimum distance metric.

3.5.1 An Example of Viterbi Convolutional Decoding

For simplicity, a BSC is assumed; thus Hamming distance is a proper distance measure. The encoder for this example is shown in Fig (3.2) and the encoder trellis diagram is shown in Fig (3.5). A similar trellis can be used to represent the decoder as shown in Fig (3.8). We start at time t_1 in the 00 state. Since in this example, there are only two possible transitions leaving any state, not all branches need be shown initially. The full trellis structure evolves after time t_3.

The basic idea behind the decoding procedure can best be understood by examining the Fig (3.5) encoder trellis in concert with Fig (3.8) decoder trellis. For the decoder trellis it is convenient at each time interval, to label each branch with the Hamming distance between the received code symbols and the branch word corresponding to the same branch from the encoder trellis. The example in Fig (3.8) shows a message sequence m, the corresponding codeword sequence U, and a noise corrupted sequence Z = 11 01 01 10 01 …… The branch words seen on the encoder trellis branches characterize the encoder in Fig (3.2) and are known a priori to both the encoder and decoder.

Input data sequence	m: 1	1	0	1	1
Transmitted codeword	U: 11	01	01	00	01
Received sequence	Z: 11	01	01	10	01



Legend

———————— Input bit 0

·················· Input bit 1

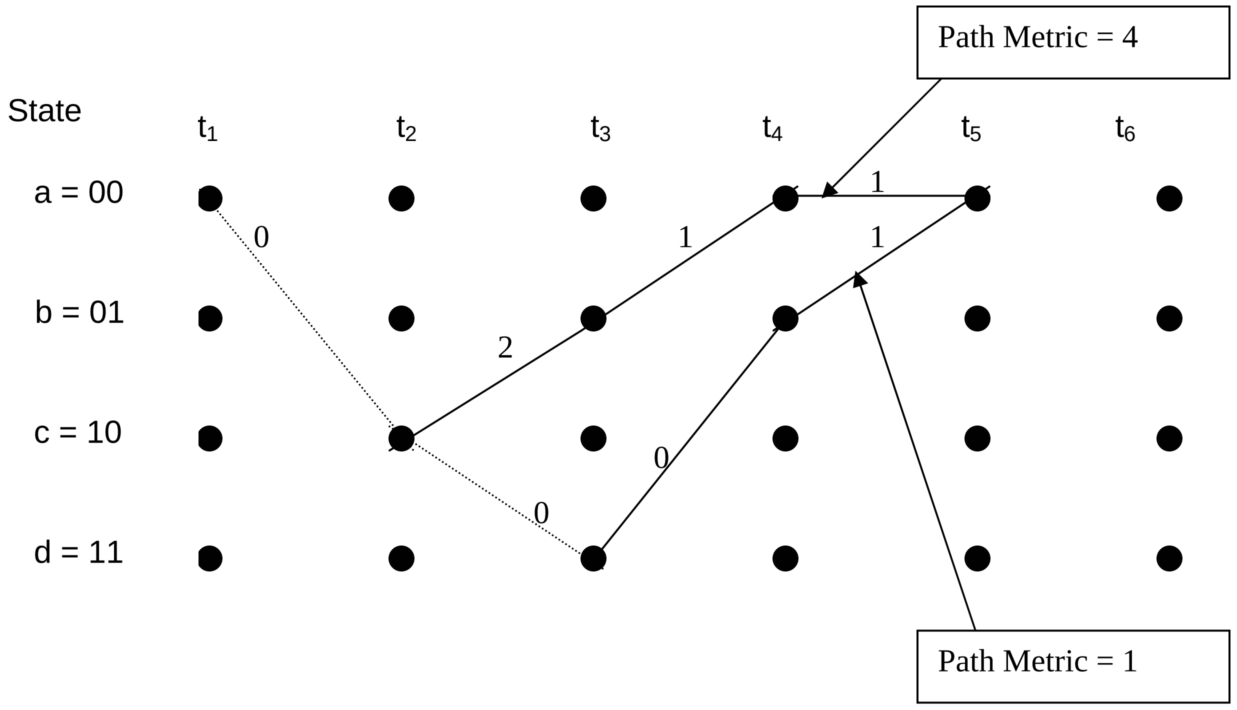
Fig (3.8): Decoder trellis diagram (rate = ½, K = 3)

From the received sequence Z, shown in Fig (3.8), we see that the code symbols received at time t_1 are 11. In order to label the decoder branches at (departing) time t_1 with the appropriate Hamming distance metric, we look at the Fig (3.5) encoder trellis. Here we see that a state 00 -> 00 transition yields an output branch word of 00. But we receive 11. Therefore, on the decoder trellis we label the state 00 -> 00 transition with Hamming distance between them, namely 2. Looking at the encoder trellis again, we see that a state 00 -> 10 transition yields an output branch word of 11, which corresponds exactly with the code symbols we received at time t_1. Therefore, on the decoder trellis, we label the state 00 -> 10 with a Hamming distance of 0. In summary, the metric entered on a decoder trellis branch represents the difference (distance) between what was received and what "should have been" received had the branch word associated with that branch been transmitted. In effect, these metrics describe a correlation like measure between a received branch word and each of the candidate branch words. We continue labeling the decoder trellis branches in this way as the symbols are received at each time t_i. The decoding algorithm uses these Hamming distance metrics to find the most likely (minimum distance) path through the trellis.

The basis of Viterbi decoding is the following observation: If any two paths in the trellis merge to a single state, one of them can always be eliminated in the search for an optimum path. For example, Fig (3.9) shows two paths merging at time t_5 to state 00. Let us define the cumulative Hamming path metric of a given path at t_i as the sum of the branch Hamming distance metrics along that path up to time t_i. In Fig (3.9) the upper path has metric 4; the lower has metric 1. The upper path cannot be a portion of the optimum path because the lower path, which enters the same state, has a lower metric.

At each time t_i there are 2^{K-1} states in the trellis, where K is the constraint length, and each state can be entered by means of two paths. Viterbi decoding consists of computing the metrics for the two paths entering each state and eliminating one of them. This computation is done for each of the 2^{K-1} states or nodes at time t_i; then the decoder moves to time to t_{i+1} and repeats the process.

At a given time, the winning path metric for each state is designated as the *state metric* for that state at that time.



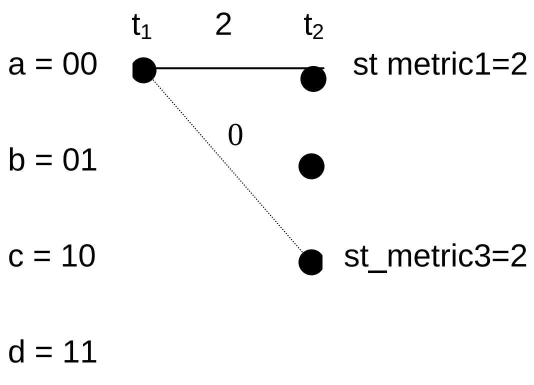
Legend

———— Input bit 0

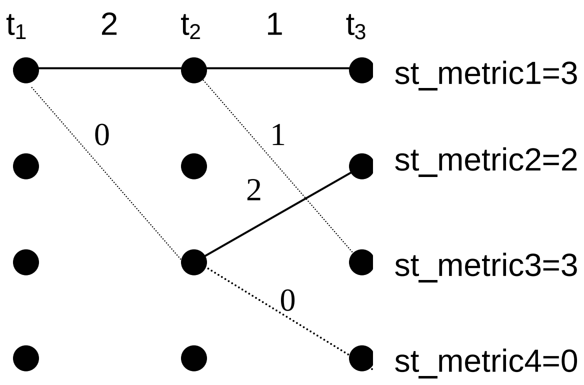
········ Input bit 1

Fig (3.9): Path metrics for two merging paths.

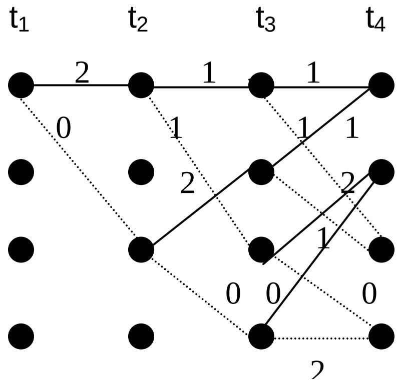
The first few steps in our decoding example are as follows (see Fig (3.10)). Assume that the input data sequence m, codeword U, and received sequence Z are as shown in Fig (3.8). Assume that the decoder knows the correct initial state of trellis. At time t_1 the received code symbols are 11. From state 00 the only possible transitions are to state 00 or state 10, as shown in Fig (3.10a). State 00 -> 10 transition has branch metric 0. At time t_2 there are two possible branches leaving each state, as shown in Fig (3.10b). The cumulative metrics of these branches are labeled state metrics st_metric1, st_metric2, st_metric3 and st_metric4, corresponding to terminating state. At time t_3 in Fig (3.10c) there are again two branches diverging from each state. As a result, there are two paths entering each state at time t_4. One path entering each state can be eliminated, namely, the one having the larger cumulative path metric. Should metrics of the two entering paths be of equal value, one path is chosen for elimination by using an arbitrary rule. The surviving path into each state is shown in Fig (3.10d). At this point in decoding process, there is only a single surviving path, termed the common stem, between times t_1 and t_2. Therefore, the decoder can now decide that the state transition which occurred between t_1 and t_2 was 00 -> 10. Since this transition is produced by an input bit one, the decoder outputs a one as the first decoded bit.



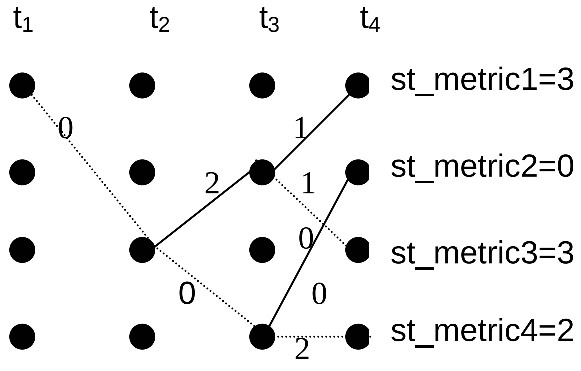
(a)



(b)



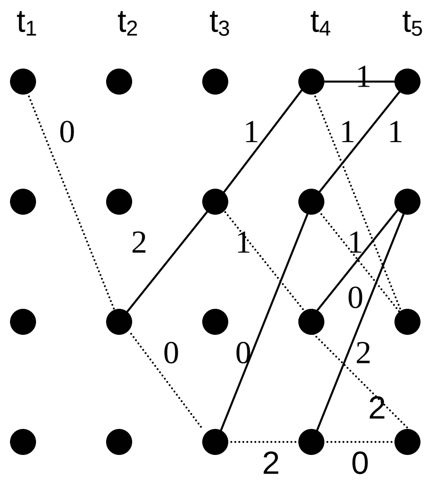
(c)



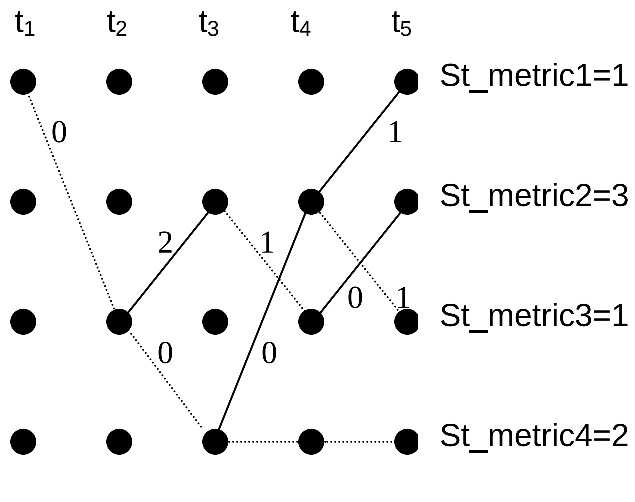
(d)

Fig (3.10) Selection of survivor paths (a) survivors at t_2. (b) Survivors at t_3. (c) Metric comparison at t_4. (d) Survivors at t_4. (e) Metric comparisons at t_5. (f) Survivors at t_5. (g) Metric comparisons at t_6. (h) Survivors at t_6.

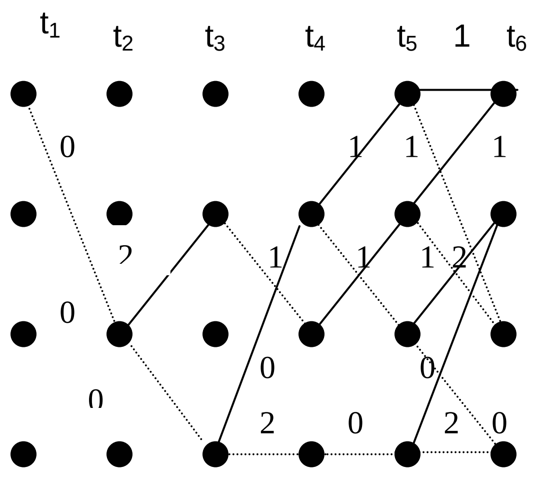
Fig (3.10e) shows the next step in the decoding process. Again, at time t_5 there are two paths entering each state, and one of each pair can be eliminated. Fig (3.10f) shows the survivors at time t_5. Notice that in our example we cannot yet make a decision on the second input data bit because there still are two paths leaving the state 10 node at time t_2. At time t_6 in Fig (3.10g) we again see the pattern of remerging paths, and in Fig (3.10h) we see the survivors at time t_6. Also in Fig (3.10h) the decoder outputs one as the second decoded bit, corresponding to the single surviving path between t_2 and t_3. The decoder continues in this way to advance deeper into the trellis and to make decisions on the input data bits by eliminating all paths but one. Pruning the trellis (as paths remerge) guarantees that there are never more paths than there are states. For this example, verify that after each pruning in Fig (3.10b, d, f, h), there are only 4 paths.



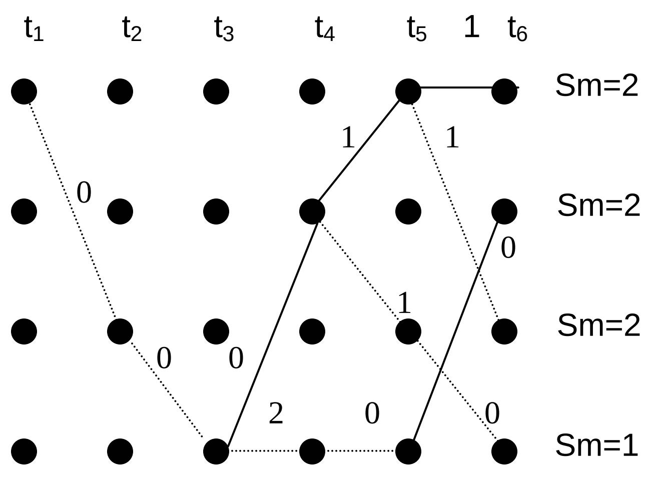
(e)



(f)



(g)



(h)

3.5.2 Decoder Implementation

In the context of the trellis diagram of Fig (3.8), transitions during any one time interval can be grouped into 2^{v-1} disjoint cells [5], each cell depicting four possible transitions, where v = K-1 is called the encoder memory. For the K = 3 example, v = 2 and 2^{v-1} = 2 cells. These cells are shown in Fig (3.11), where a, b, c and d refer to the states at time t_i, and a', b', c' and d' refer to the state at time t_{i+1}. Shown on each transition is the branch metric bmet.

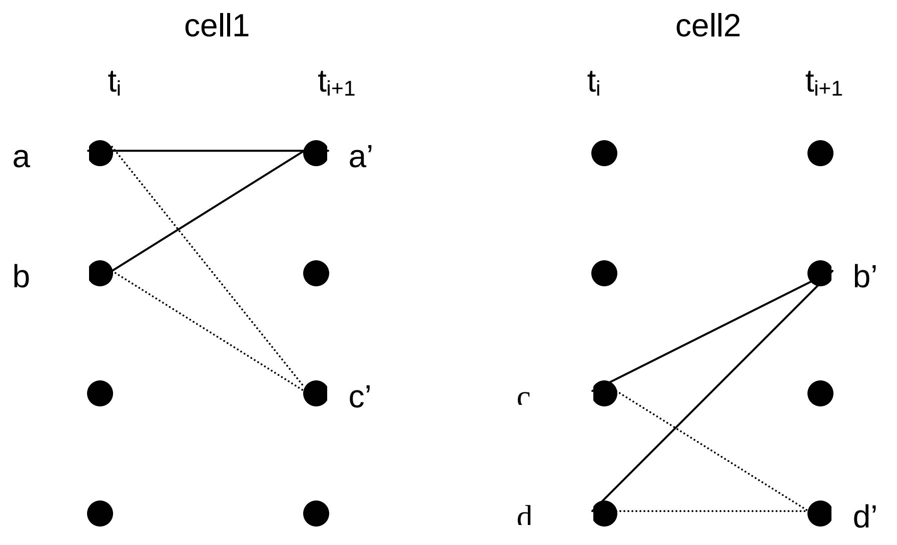


Fig (3.11): Example of decoder cells

3.5.3 Add-Compare-Select Computation

Continuing with K = 3, 2-cell example, Fig (3.12) illustrates the logic unit that corresponds to cell1. The logic executes the special purpose computation called add-compare-select (ACS). The state metric of state a, st_metric(a') is calculated by adding the previous time state metric of state a, st_metric(a), to the branch metric bmet and the previous time state metric of state c, st_metric(c), to the branch metric bmet. This results in two possible path metrics as candidates for the new state metric (a'). The two candidates are compared in logic unit of Fig (3.12). The largest likelihood (smallest distance) of the two path metrics is stored as the new state metric st_metric(a') for state a. Also stored is the new path history m'(a') for state a.

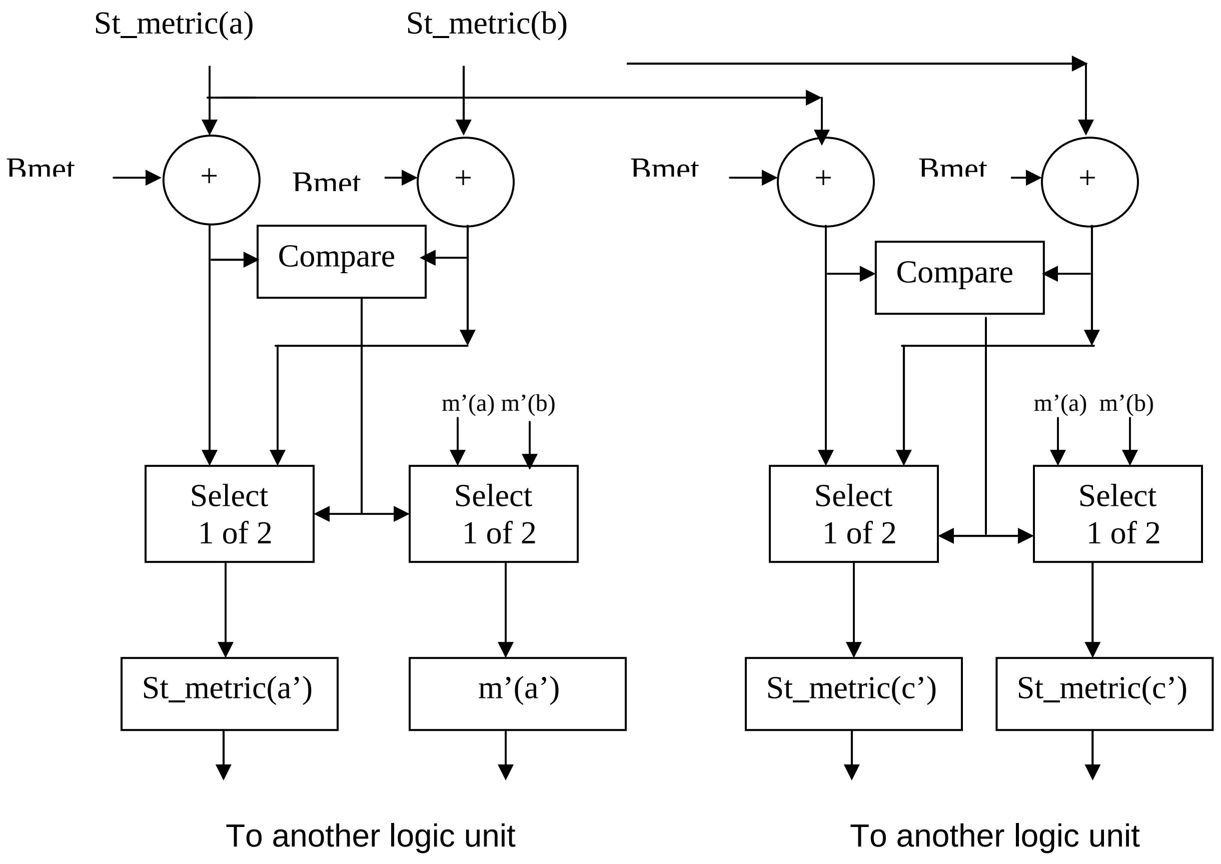


Fig (3.12) Logic unit that implements the add-compare-select functions corresponding to cell1

Also shown in Fig (3.12) is the cell1 ACS logic that yields the new state metric st_metric(c') and new path history m'(c'). This ACS operation is similarly performed for the paths in other cells. The oldest bit on the path with the smallest state metric forms the decoder output.

3.5.4 Add-Compare-Select as seen on the Trellis

Consider the same example that was used for describing Viterbi decoding earlier. The message sequence was m = 1 1 0 1 1, the corresponding sequence was U = 11 01 01 00 01, and the received sequence was Z = 11 01 01 10 01. Fig (3.13) depicts a decoding trellis diagram similar to Fig (3.8), as shown below.

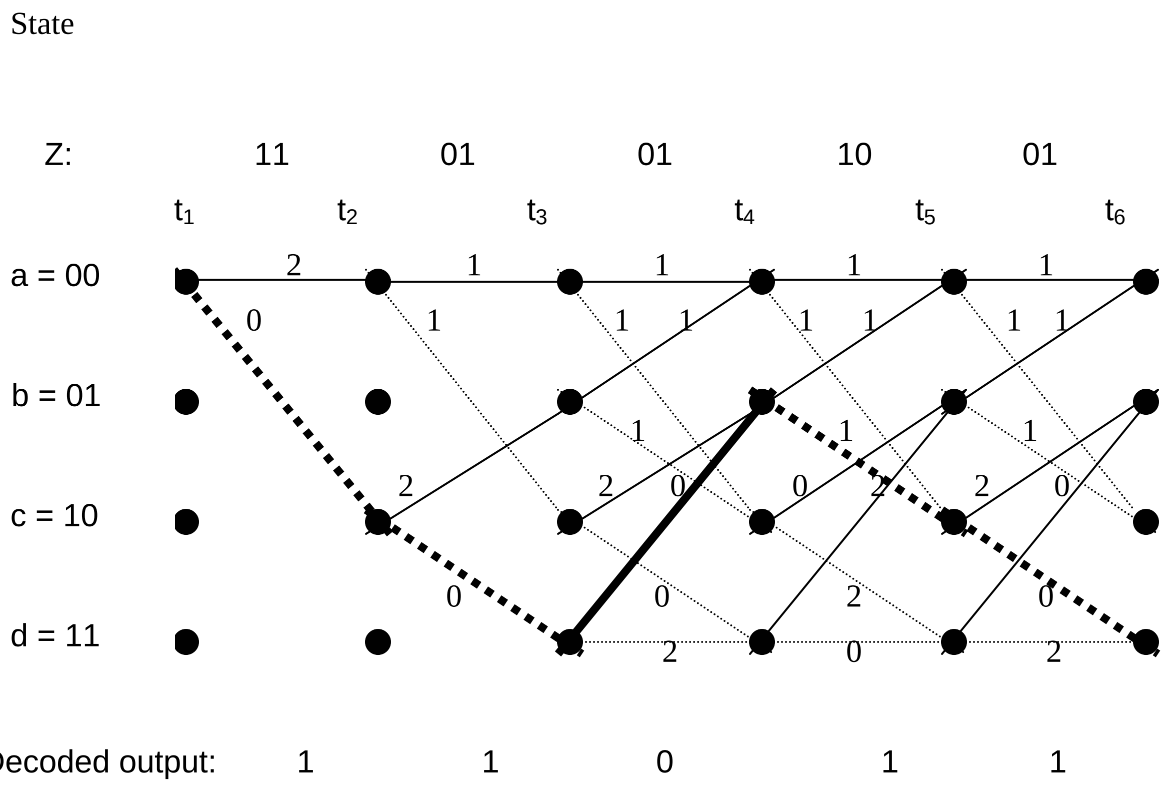


Fig (3.13): Add-compare-select computations in Viterbi decoding

A branch metric that labels each branch is the Hamming distance between the received code symbols and the corresponding branch word from the encoder trellis. We perform the add-compare-select (ACS) operation when there are two transitions entering a state, as there are for times t_4 and later. For example at time t_4, the value of state metric for state a is obtained by incrementing the state metric st_metric1 = 3 at time t_3 with the branch metric bmet1 = 1 yielding a candidate value of 4. Simultaneously, the state metric st_metric2 = 2 at time t_3 is incremented with the branch metric bmet3 = 1 yielding a candidate value of 3. The select operation of ACS process selects the largest-likelihood (minimum distance) path metric as the new state metric; hence, for state a at time t_4, the new state metric is st_metric(a') = 3. The winning path is shown with a heavy line. On the trellis of Fig (3.13), observe the state metrics from left to right. Verify that at each time, the value of each state metric is obtained by incrementing the connected state metric from the previous time along the winning path with the branch metric between them. At some point in trellis; the oldest bit can be decoded. As an example, looking at time t_6 in Fig (3.13), we see that the minimum distance state metric has a value of 1. From this state d, the winning path can be traced back to time t_1, and one can verify that the decoded message is the same as the original message, by the convention that dashed and solid line represent binary ones and zeros respectively.

CHAPTER 4
A SIMPLIFIED TRELLIS-BASED DECODER

4.1 INTRODUCTION

Trellis-Based decoding is a popular method to recover convolutionally encoded information corrupted during transmission over a noisy channel. For example, the Viterbi algorithm [22] and Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm [24] are two schemes that work on an underlying trellis description of encoded sequence.

Basic computations in either algorithm involve branch metric (BM) calculations and add-compare-select (ACS) operations. In case of the VA, an ACS operation successively discards branches that cannot be part of the survivor path. In case of the BCJR in the logarithmic domain (the Log-MAP algorithm), this operation corresponds to an add-max* operation which is basically an ACS operation with an added offset (ACSO) to correct for the Jacobian logarithm. Hence, the presented considerations for the ACS hold for the ACSO as well.

All most all-good rate 1/n convolutional codes, n an integer, have the property that the code symbol labels on the two branches into each trellis node are complementary. This results into simplifications of the BM and ACS units that present a simplified architecture with reduced complexity, thus saving hardware [3].

4.2 NOTATION

The ACS operation is best described by equation (4.1) written below. Let Γ(s, k+1) be the updated metric of states at time k+1, based on the preceding state metric at time k and respected branch metric $\lambda()$:

$$\Gamma\,(s, k+1) = \min\{[\Gamma(s', k)+\lambda(s', s), \Gamma(s'', k)+\lambda(s'', s)]\} \tag{4.1}$$

A channel symbol received from a soft output demodulator is quantized with q bits and denoted y_i. Clearly, there are 2^q quantization levels and $y_i \in [0, 2^q-1]$.

This symbol is the output of a discrete memory less channel with binary input x_j and transition probabilities $P(y_i|x_j)$. The expected code symbol $c_i(s', s)$ along the branch from state s' to state s is derived by the mapping x_0->0 and x_1->2^q-1.

In additive white Gaussian noise channel the optimal distance measure is the squared Euclidean distance

$$\sum_{i=0}^{n-1} |y_i-c_i(s', s)|^2. \qquad (4.2)$$

However, given the preceding symbol constraints, this measures simplifies to

$$\begin{aligned}\lambda_i(s', s) &= y_i, && \text{for } c_i(s', s) = 0\\ &= (2^q-1) - y_i, && \text{for } c_i(s', s)=2^q - 1 \qquad (4.3)\end{aligned}$$

and the complete branch metric is then written as

$$\lambda(s', s)= \sum_{i=0}^{n-1} \lambda_i(s', s). \qquad (4.4)$$

4.3 COMPLEMENTARY PROPERTY

This discussion is restricted to rate ½ codes, that is n = 2, although the considerations can be generalized to 1/n codes. Rate ½ codes play by far the most important role in today's communication systems since they are a good compromise between achievable coding gain, bandwidth efficiency, and implementation complexity. In practice, high-rate codes are usually obtained by puncturing a basic rate ½ code. However, we begin with a general notation that shows that the most beneficial simplification results for n = 2.

We consider both feed-forward encoders and some systematic feedback encoders. These encoders have one thing in common: The code symbols of merging branches are always complementary as shown in Fig (4.1).

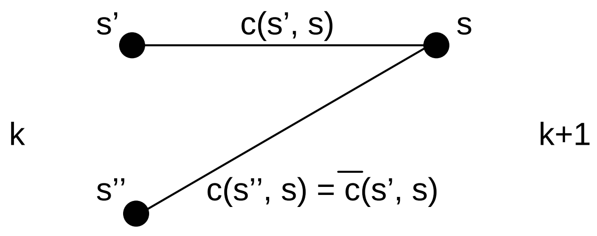


Fig (4.1): Complementary property of merging branches.

The complementary operation on c is defined as the complementation of its elements, that is $\overline{c} = (\overline{c}_0\ \overline{c}_1, \ldots\ldots \overline{c}_{n-1})$, where $c_i + \overline{c}_i = 2^q - 1$.

From the considerations in section 4.2 it is clear that the branch metrics share this property since they linearly depend on the code symbols. Hence, one branch metric can be expressed by means of the other and we write

$$\begin{aligned}\lambda(s'', s) &= n(2^q-1)-\lambda(s', s)\\ &= (n-1)\lambda(s', s)+n[(2^q-1)-\lambda(s', s)]. \end{aligned} \tag{4.5}$$

We define the modified branch metric

$$\lambda^*(s', s) \equiv n[(2^q-1)-\lambda(s', s)] \tag{4.6}$$

Which is a signed number, and (4.5) becomes

$$\lambda(s'', s) = (n-1)\lambda(s', s) + \lambda^*(s', s). \tag{4.7}$$

Substituting (4.7) into (4.1) becomes

$$\Gamma(s, k+1) = \min\{\Gamma(s', k)+\lambda(s', s), \Gamma(s'', k)+ (n-1)\lambda(s', s) + \lambda^*(s', s)\} \tag{4.8}$$

Finally, the factor λ(s', s) in the first argument of (4.8) can be taken out of the comparison and we get

$$\Gamma(s, k+1) = \lambda(s', s) + \min\{\Gamma(s', k), \Gamma(s'', k) + (n-2)\lambda(s', s) + \lambda^*(s', s)\}. \qquad (4.9)$$

Or, equivalently

$$\Gamma(s, k+1) = \lambda(s', s) + \overline{\Gamma}(s, k+1) \qquad (4.10)$$

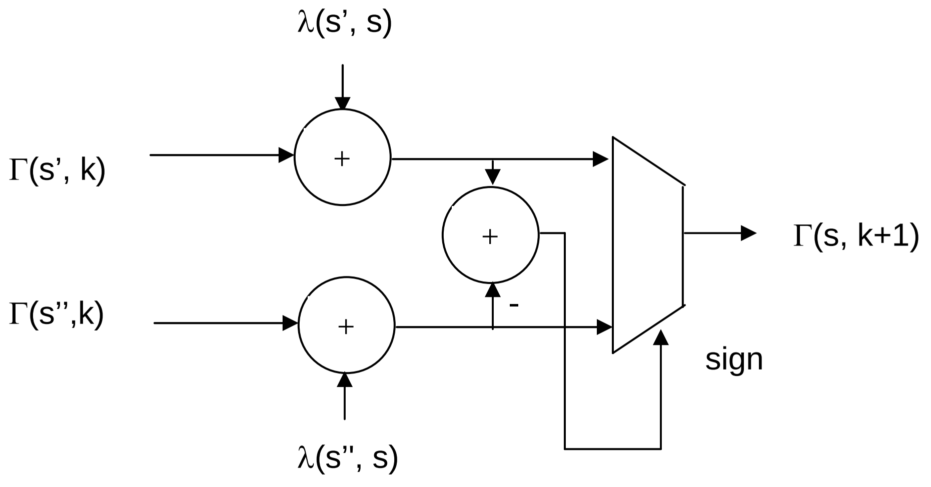
Where $\overline{\Gamma}$(s, k+1) is the new outcome of the min operation.

There are several things to be observed in (4.9) and (4.10). First, considering that the branch metrics are pre-calculated, there is one addition less needed to carry out the comparison since the first argument in the comparison remains unchanged. Second, for n = 2 the factor λ(s', s) disappears in the second argument and the comparison solely depend on one (modified) branch metric. Third, in order to retain the numerical relation between interconnected state metrics with different λ() we have to add this factor after having determines $\overline{\Gamma}$(s, k+1). However, one can subtract this factor from all state metrics and it will be shown that in that case half the ACS units do not need this correction, that is Γ(s, k+1) = $\overline{\Gamma}$(s, k+1). Note that if the butterflies in a trellis were disjoint, this correction could be neglected in all ACS units.

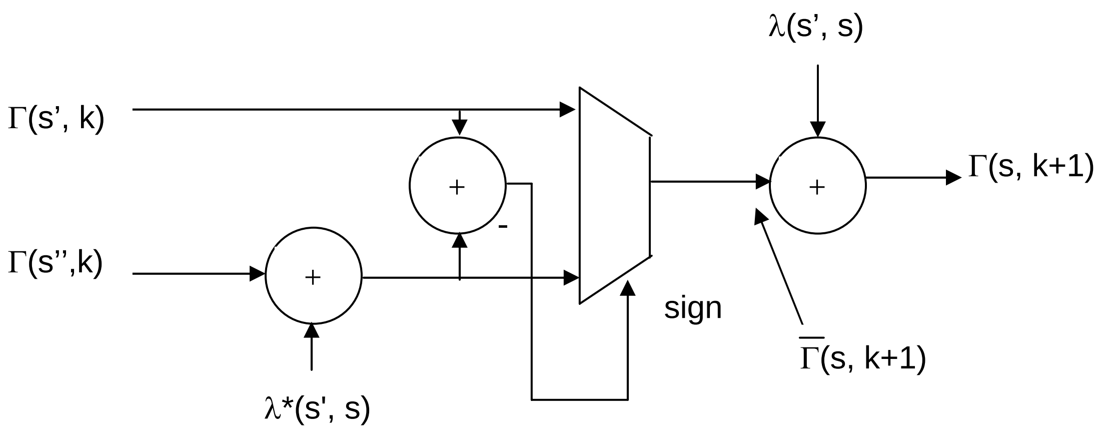
4.4 MODIFIED BM AND ACS UNITS FOR RATE ½

We start by noting that the branch metric λ(s', s) can take four different values, namely $\lambda(x_0x_1)$ for every possible combination of symbols $x_j \in \{0, 1\}$.

Fig (4.2) shows both the conventional and transformed ACS unit. Both units have the same complexity but the later needs one adder less to determine $\overline{\Gamma}$(s, k+1).



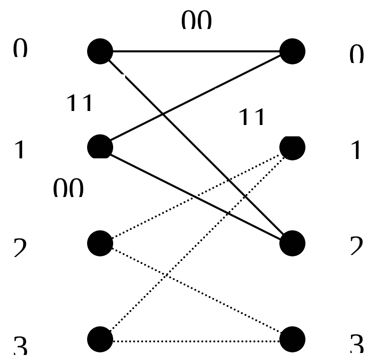
(a)

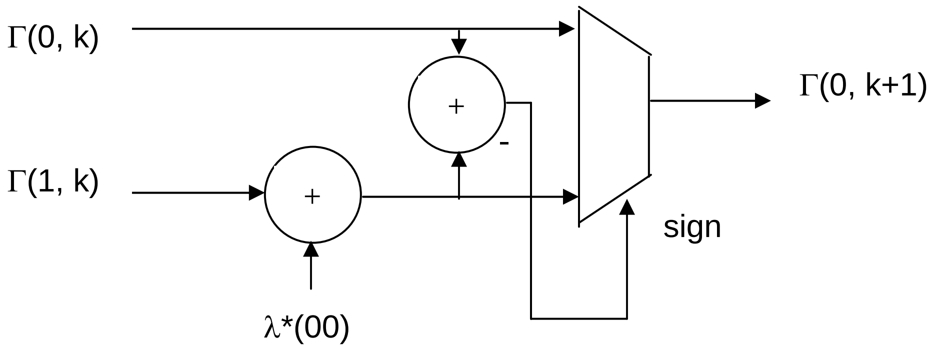


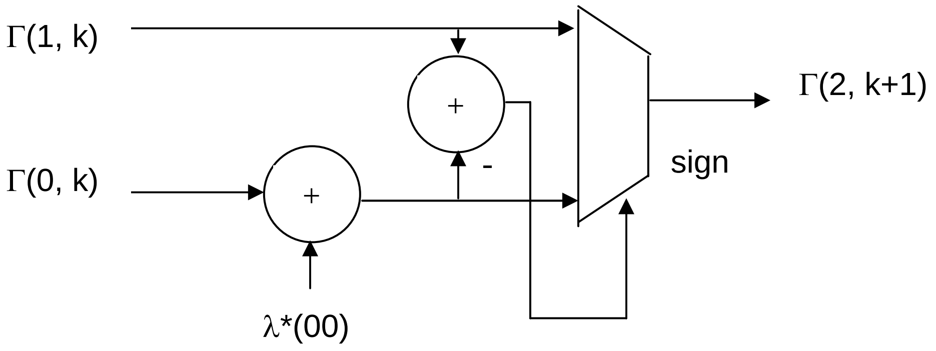
(b)

Fig (4.2) a: Conventional and (b) transformed ACS unit for a rate ½ code. Both units have the same complexity but the later needs one adder less to determine the outcome of comparison.

The hardware savings now become apparent by looking at an example, an ACS unit setup for decoding a (7,5) code in Fig (4.3).


00
0
0
11
11
1
1
00
2
2
3
3


Γ(0, k)
Γ(1, k)
+
+
-
Γ(0, k+1)
sign
λ*(00)


Γ(1, k)
Γ(0, k)
+
+
-
Γ(2, k+1)
sign
λ*(00)

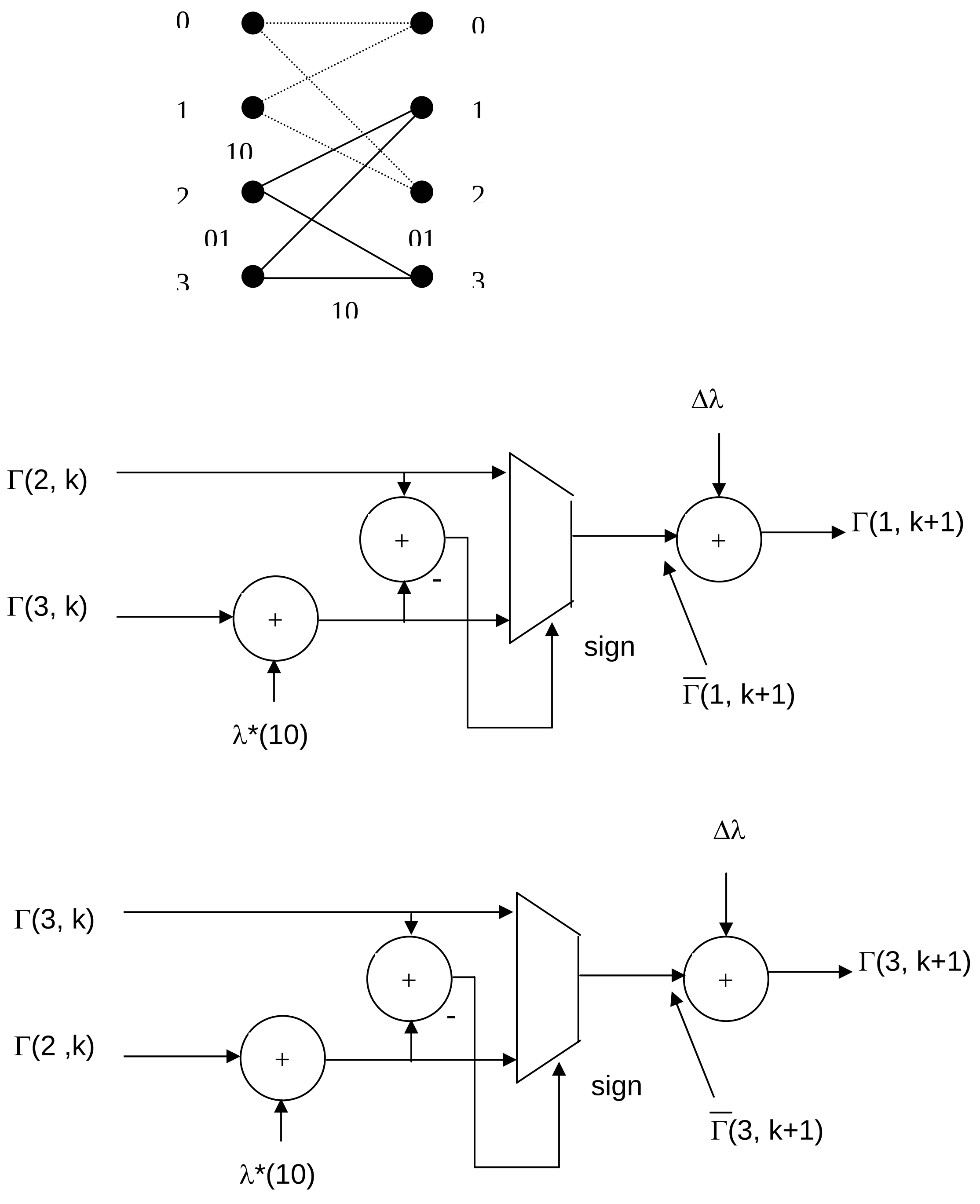


Fig (4.3): Proposed ACS unit setup for decoding a (7, 5) code.

In this picture, the factor λ(s', s) of Fig 4.2(b) to be added in an ACS unit is either λ(00) or λ(10). However, we can subtract, for example λ(00) from all state metrics. This factor belongs to the two ACS units on the left and, therefore, the state metric corrections in these units become unnecessary while

$$\Delta\lambda = \lambda(10) - \lambda(00)$$

has to be added to the other units. Hence, for rate ½ codes that has the complementary property half the ACS units save one adder compared to a conventional setup. If speed is an issue, $\Delta\lambda$ could be stored in the BM unit and added in the next computation cycle instead, thus maintaining the original critical path of conventional ACS unit. However, the BM unit becomes slightly more complex in this case.

The calculation of the modified branch metric $\lambda^*(s', s)$ based on (4.6) for n = 2 is shown in Fig (4.4). Normally, the expression in square brackets in (4.6) would be the bit-complement of $\lambda(s', s)$. However, since $\lambda(s', s) \in [0, 2(2^q - 1)]$ one has to exclude the most significant bit (MSB), which indicates the sign of the modified branch metric, from the negation. Since n = 2 the multiplication in (4.6) reduces to a left shift by one bit. Note that if n is not a power of two, this multiplication cannot reduce to bit-shift operations.

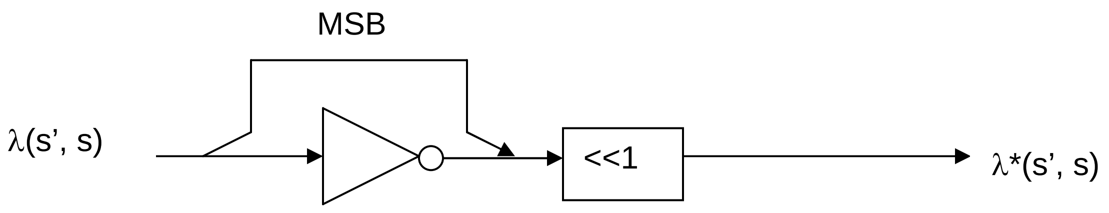
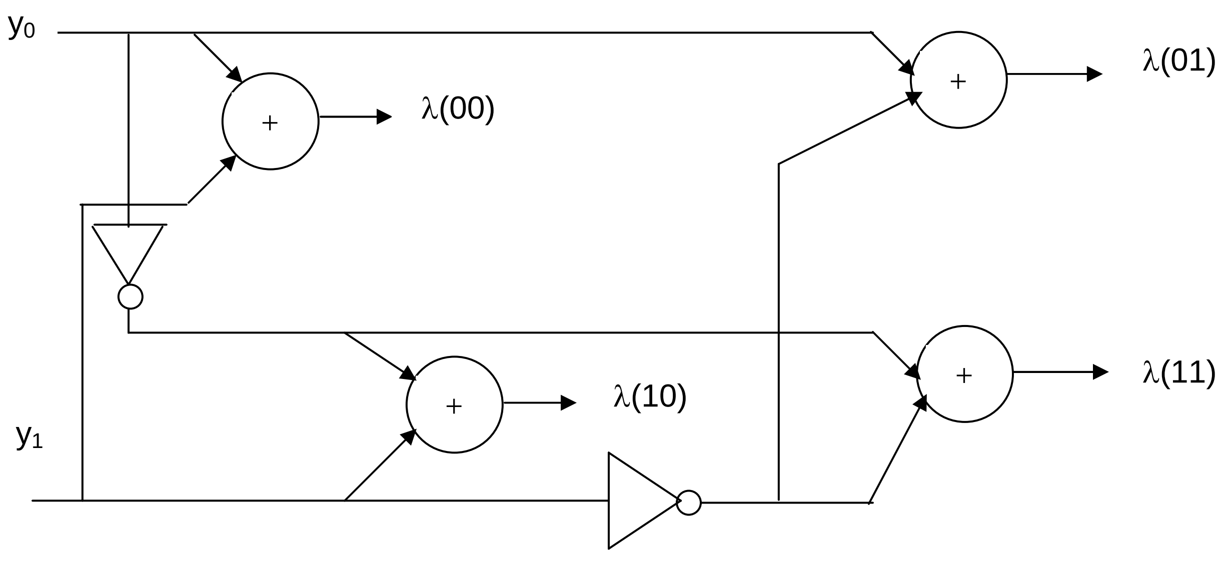


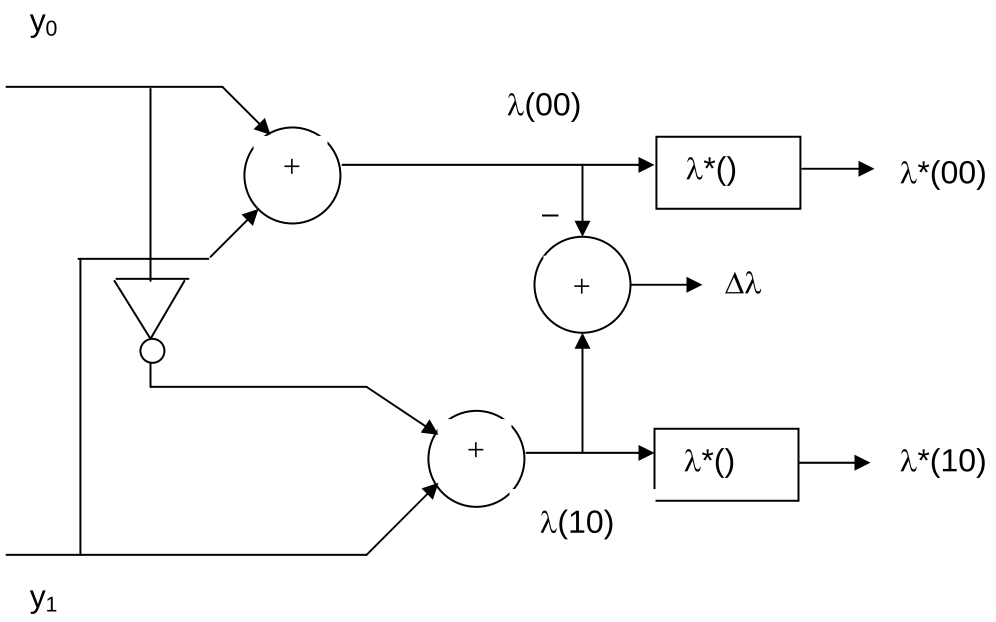
Fig (4.4): Generation of $\lambda^*()$, <<1 denotes a left shift by one bit.

4.5 COMPARISON

If there are 2^n distinct code sequences, a conventional BM unit requires $2^n(n-1)$ additions and n negations to calculate 2^n branch metrics. Hence for a rate ½ code we need four adders and two negations to calculate four branch metrics, see Fig 4.5(a). The proposed BM unit shown in Fig 4.5(b) requires only three additions, one negation of a channel symbol, and two negations of intermediate branch metrics to calculate two branch metrics.



(a)



(b)

Fig 4.5(a): Conventional and (b): proposed BM unit for a rate ½ code.

Notice that a bit-shift operation comes at negligible cost in a hardware implementation. Furthermore, the difference between the two branch metrics, $\Delta\lambda$, needed to normalize half the state metrics becomes in this case simply $(2^q-1)-2y_0$. This operation can be further simplified on the bit level into a bit-shift followed by a negation (MSB excluded) of y_0 and is hence not considered an adder in Table I

TABLE I

Number of Additions for BM/ACS Unit Setup of a Rate ½ Code

Unit	ACS	BM
Conventional	3.2^m	4
Proposed	5.2^{m-1}	2

This table shows that the number of additions for a BM/ACS unit setup for code rate ½ and memory m. The proposed scheme halves the additions in the BM unit and reduces the number of additions for the ACS unit by 17%. By software simulation of the hardware circuits, we have verified that decoder error performance stays the same.

4.6 CONCLUSION

We have shown that the implementation of BM and ACS units in trellis-based decoding architectures can be simplified for a certain class of convolutional codes. For a rate ½ code, half the ACS units save one adder compared to conventional implementation. Furthermore, only two branch metrics have to be calculated instead of four. These potential hardware savings will also lead to savings in power consumption.

4.7 ALGORITHM: A SIMPLIFIED TRELLIS-BASED DECODER

1. Start
2. Take a 2-D array of integers of size 4×2 named output with contents [(0,3), (3,0), (1,2), (2,1)] {for two code generators with coefficients 111 and 101}.
3. Take a file pointer (ifp) and associate it with a text file containing bit values after encoding and passing through AWGN channel.
4. Take a file pointer (ofp) and associate it with an empty output text file.
5. Count the number of bits in input file and store it in variable n.
6. Take an array (numoct) of integers of size n/2 for storing octal equivalent of bits taking two each time in input file.
7. Take an array (decod) of integers of size n/2+1 for storing decoding bit values.
8. Take a double array (st_metric) of integers of size (n/2+1×4) for storing state metric values for each value stored in numoct.
9. Convert the bit values (a pair) from input file to equivalent octal values
   - (i) Let i = 0 and count =0
   - (ii) Let c = bit from input file at position ‘count’
   - (iii) c = c-‘0’
   - (iv) Let c1 = bit from input file at position ‘count+1’
   - (v) c1 = c1-‘0’
   - (vi) numoct [count/2] = 2*c+c1
   - (vii) Is count >=n, if True go to step 10.
   - (viii) count = count+2
   - (ix) Repeat step (ii) to (vii)
10. Equate state metric at time t = 1 to all zeros
    - (i) for i = 0 to 3
    - (ii) st_metric[0][i] = 0]
    - (iii) end for
11. Calculate state metric for time t = 2 using branch metric values bm1 and bm2 and function hammingdist described after main algorithm.
    - (i) bm1 = hammingdist (numoct[0],output[0][0])

```
        (ii)    st_metric[1][0] = st_metric[0][0]+bm1
        (iii)   bm2 = hammingdist (numoct[0], output[0][1])
        (iv)    st_metric[1][2] = st_metric[0][0]+bm2
12. Calculate state metric for time t =3
        (i)     bm1 = hammingdist (numoct[1], output[0][0])
        (ii)    bm2 = hammingdist (numoct[1], output[0][1])
        (iii)   st_metric[2][0] = st_metric[1][0]+bm1
        (iv)    st_metric[2][2] = st_metric[1][0]+bm2
        (v)     bm1 = hammingdist (numoct[1],output[2][0])
        (vi)    bm2 = hammingdist (numoct[1],output[2][1])
        (vii)   st_metric[2][1] = st_metric[1][2]+bm1
        (viii)  st_metric[2][3] = st_metric[1][2]+bm2
13. Calculate state metric values for time t>=4
        (i)     Let i = 2
        (ii)    Let sym = numoct[i]
        (iii)   Let bm1 = hammingdist (sym, output[0][0])
        (iv)    Let 'modified branch metric value' modbm1 = 2*(1-bm1)
        (v)     Let bm2 = hammingdist (sym, output[2][0])
        (vi)    Let modbm2 = 2*(1-bm2)
        (vii)   If st_metric[i][0] < = st_metric[i][1] + modbm1 then
                st_metric[i+1][0] = st_metric[i][0]
                else
                st_metric[i+1][0]= st_metric[i][1]+modbm1
                end if
        (viii)  If st_metric[i][1] < = st_metric[i][0] + modbm1 then
                st_metric[i+1][2] = st_metric[i][1]
                else
                st_metric[i+1][2]= st_metric[i][0]+modbm1
                end if
        (ix)    If st_metric[i][2] < = st_metric[i][3] + modbm2 then
                st_metric[i+1][1] = st_metric[i][2]
```

```
            else
            st_metric[i+1][1]= st_metric[i][3]+modbm2
            end if
    (x)     st_metric[i+1][1] = st_metric[i+1][1]+bm2-bm1
    (xi)    If st_metric[i][3] < = st_metric[i][2] + modbm2 then
            st_metric[i+1][3] = st_metric[i][3]
            else
            st_metric[i+1][3]= st_metric[i][2]+modbm2
            end if
    (xii)   st_metric[i+1][3] = st_metric[i+1][3]+bm2-bm1
    (xiii)  Is i >= n/2, if true then go to step 14
    (xiv)   i =i + 2
    (xv)    Repeat step (ii) to (xiii)
14. Calculating minimum value of state metric at time t = n/2
    (i)     Let min = st_metric[n/2][0]
    (ii)    Let i = 1
    (iii)   If st_metric[n/2][i] < min then
            min = st_metric[n/2][i]
            index =j
            end if
    (iv)    Is j = 3 true then go to step 15
    (v)     i = i+1
    (vi)    Repeat step (iii) and (iv)
15. Traversing back the path from t=n/2 to t =1 and get the decoded bits
    (i)     Let i = n/2
    (ii)    If index = 0 then
            If st_metric[i-1][0] < = st_metric[i-1][1] then
            index =0
            else
            index = 1
            end if
```

```
         decod[i] = 0
         i= i-1
         end if
(iii)    If index = 1 then
         If st_metric[i-1][2] < = st_metric[i-1][3] then
         index =2
         else
         index = 3
         end if
         decod[i] = 0
         i= i-1
         end if
(iv)     If index = 2 then
         If st_metric[i-1][0] < = st_metric[i-1][1] then
         index =0
         else
         index = 1
         end if
         decod[i] = 1
         i= i-1
         end if
(v)      If index = 3 then
         If st_metric[i-1][2] < = st_metric[i-1][3] then
         index = 2
         else
         index = 3
         end if
         decod[i] = 1
         i= i-1
         end if
(vi)     Is i < = 0 if true then go to step 16
```

(vii) Repeat (ii) to (vi)

16. Write the decoded bit values (decod array) to the output file.
17. Convert the byte string in output file to alphanumeric characters and write into another text file.
18. The above file containing characters is the final result. Compare it with the original file, which was encoded and sent and with the file, which would have been received if sent uncoded.
19. Stop

Define below a function Hammingdist, which is used in main algorithm written above. It takes two integer values as inputs and calculates the hamming distance between them and then returns this value back to the main program.

```
1.  Take two integer input values x1 and x2
2.  Let x3=x1-x2
3.  if x3<0 then
    x3=-x3
4.  if x3=0 then
    return 0
5.  if x3=1 then
    if(x1=1 and x2=2)or(x1=2 and x2=3) then
    return 2
    else
    return 1
    end if
    end if
6.  if x3=2 then
    return 1
7.  if x3=3 then
    return 2
8.  end hammingdist
```

CHAPTER 5
TURBO CODES

5.1 INTRODUCTION

Concatenated coding schemes were first proposed by Forney as a method for achieving large coding gains by combining two or more relatively simple building-block or component codes (sometimes called constituent codes). The resulting codes [18] had the error-correction capability of much longer codes, and they were endowed with a structure that permitted relatively easy to moderately complex decoding. *A turbo code can be thought of as a refinement of the concatenated encoding structure plus an iterative algorithm [26] for decoding the associated code sequence*. Because of its unique form, we choose to list turbo as a separate category under structured sequences.

Turbo codes were first introduced in 1993 by Berrou, Glavieux, and Thitimajshima [10], where a scheme is described that achieves a bit-error-probability of 10^{-5}, using a rate ½ code over an additive white Gaussian noise (AWGN) channel and BPSK modulation at an E_b/N_0 0f 0.7 dB. The codes are constructed by using two or more component codes on different interleaved versions of the same information sequence. For a system with two components codes, the concept behind turbo decoding is to pass soft decisions from the output of one decoder to the input of the other decoder, and to iterate this process several times so as to produce more reliable decisions.

5.2 TURBO CODE CONCEPTS

5.2.1 Likelihood functions

The mathematical foundation of hypothesis testing rests on Bayes' theorem. For communication engineering, where application involving an AWGN channel are of great interest, the most useful form of Bayes' theorem expresses the a posteriori probability (APP) of a decision in terms of a continuous-valued random variable x as

$$P(d = i|x) = \frac{p(x|d = i)\,P(d = i)}{p(x)} \qquad i = 1,\ldots\ldots, M \tag{5.1}$$

and

$$p(x) = \sum_{i=1}^{M} p(x|d = i)P(d = i) \tag{5.2}$$

Where P (d = i|x) is the APP, and d = i represents data d belonging to the ith signal class from a set of M classes. Further, p(x|d = i) represents the probability density function (pdf) of a received continuous-valued data-plus noise signal x, conditioned on the signal class d = i. Also, P(d = i), called the a priori probability, is the probability of occurrence of the ith signal class. Typically x is an "observable" random variable or a test statistic that is obtained at the output of a demodulator or some other signal processor. Therefore, p(x) is the pdf of the received signal x, yielding the test statistic over the entire space of signal classes. In (5.1), for a particular observation, p(x) is a scaling factor since it is obtained by averaging over all the classes in the space. Lower case p is used to designate the pdf of a continuous-valued random variable, and upper case P is used to designate probability (a priori and APP). Determining the APP of a received signal from equation (5.1) can be thought of as the result of an experiment. Before the experiment, there generally exists (or one can estimate) an a priori probability P(d = i). The experiment consists of using equation (5.1) for computing the APP, P (d = i|x), which can be thought of as a "refinement" of the prior knowledge about the data, brought about by examining the received signal x.

5.2.2 The Two-Signal Class Case

Let the binary logical elements 1 and 0 be represented electronically by voltages +1 and –1, respectively. The variable d is used to represent the transmitted data bit, whether it appears as a voltage, or as a logical element. For signal transmission over an AWGN channel, Fig (5.1) shows the conditional pdfs, referred to as likelihood functions. The rightmost function p(x|d = +1) shows the pdf of random variable x conditioned on d = +1 being transmitted. The leftmost function p(x|d = -1) illustrates a similar pdf conditioned on d = -1 being transmitted. The abscissa represents the full range of possible values of the test statistic x generated at receiver.

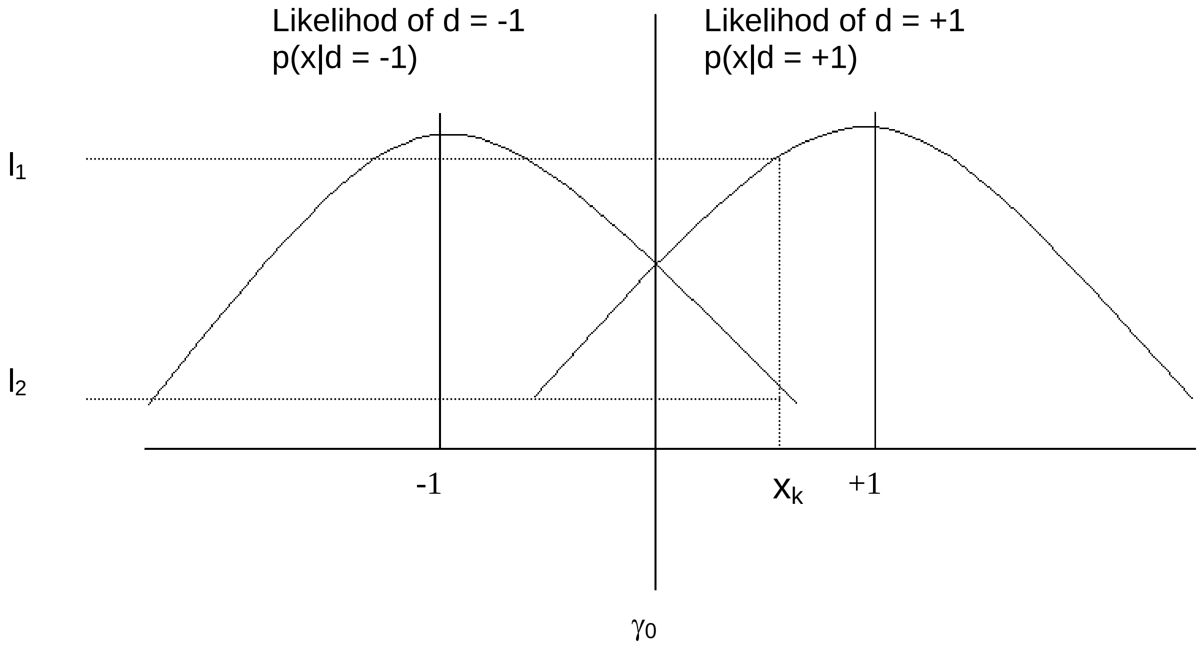


Fig (5.1): Likelihood function

In Fig (5.1), one such arbitrary value x_k is shown, where the index denotes an observation in the kth time interval. A line subtended from x_k intercepts the two likelihood functions yielding two likelihood values $l_1 = p(x_k|d_k = +1)$ and $l_2 = p(x_k|d_k = -1)$. A well-known hard decision rule, known as maximum likelihood, is to choose the data $d_k = +1$ or $d_k = -1$ associated with the larger of two intercept values l_1 and l_2, respectively. For each data bit at time k, this is tantamount to deciding that $d_k = +1$ if x_k falls on the right hand side of the decision line labeled γ_0, otherwise deciding that $d_k = -1$.

A similar decision rule, known as maximum a posteriori (MAP) [17], which can be shown to be a minimum-probability-of-error rule, takes into account the a priori probabilities of the data. The general expression for the MAP rule in terms of APPs is

$$P(d = +1|x) \underset{H_2}{\overset{H_1}{\gtrless}} P(d = -1|x) \tag{5.3}$$

Equation (5.3) states that one should choose the hypothesis H_1, (d = +1) if the APP, P(d = +1|x) is greater than the APP, P(d = -1|x). Otherwise, one should choose hypothesis H_2, (d = -1). Using the Bayes' theorem of equation (5.1), the APPs in equation (5.3) can be replaced by their equivalent expressions, yielding

$$p(x \mid d = +1)P(d = +1) \underset{H_2}{\overset{H_1}{\gtrless}} p(x \mid d = -1)P(d = -1) \tag{5.4}$$

Where the pdfs p(x) appearing on both sides of the inequality in equation (5.1) has been cancelled. Equation (5.4) is generally expressed in terms of a ratio, yielding the so-called likelihood ratio test, as follows:

$$\frac{p(x|d = +1)}{p(x|d = -1)} \underset{H_2}{\overset{H_1}{\gtrless}} \frac{P(d = -1)}{P(d = +1)}$$

or

$$\frac{p(x|d = +1)\, P(d = +1)}{p(x|d = -1)\, P(d = -1)} \underset{H_2}{\overset{H_1}{\gtrless}} 1 \tag{5.5}$$

5.2.3 Log-Likelihood Ratio

By taking the logarithm of the likelihood ration developed in equation (5.3) through equation (5.5), we obtain a useful metric called the log-likelihood ratio (LLR). It is a real number representing a soft decision out of a detector, designated by

$$L(d \mid x) = \log \left[\frac{P(d = +1|x)}{P(d = -1|x)} \right] = \log \left[\frac{p(x|d=+1)P(d =+1)}{p(x|d=-1)P(d = -1)} \right] \quad (5.6)$$

So that

$$L(d \mid x) = \log \left[\frac{p(x \mid d = +1)}{p(x \mid d = -1)} \right] + \log \left[\frac{P(d =+1)}{P(d= -1)} \right] \quad (5.7)$$

or

$$L(d \mid x) = L(x \mid d) + L(d) \quad (5.8)$$

Where L(x|d) is the LLR of the test statistic x obtained by measurement of the channel output x under the alternate conditions that d = +1 or d = -1 may have been transmitted, and L(d) is the a priori LLR of the data bit d. To simplify the notation, equation (5.7) can be rewritten as

$$L'(d') = L_c(x) + L(d) \quad (5.9)$$

Where the notation $L_c(x)$ emphasizes that this LLR term is the result of a channel measurement made at the receiver. For a systematic code, it can be shown that the LLR (soft output) out of the decoder is equal to

$$L(d') = L'(d') + L_e(d') \quad (5.10)$$

Where L'(d') is the LLR of a data bit out of demodulator (input to the decoder), and $L_e(d')$, called the extrinsic LLR, represents the extra knowledge that is gleaned from the decoding process. The output sequence of a systematic decoder is made up of values representing data bits and parity bits. From equation (5.9) and (5.10), the output LLR of the decoder is now written as

$$L(d') = L_c(x) + L(d) + L_e(d') \qquad (5.11)$$

Equation (5.11) shows that the output LLR of a systematic decoder can be represented as having three LLR elements-a channel measurement, a priori knowledge of the data, and an extrinsic LLR stemming solely from the decoder. To yield the final L(d'), each of the individual LLRs can be added as shown in equation (5.11), because the three terms are statistically independent. The soft decoder output L(d') is a real number that provides a hard decision as well as the reliability of that decision. The sign of L(d') denotes the hard decision-that is, for positive values of L(d') decide that d = +1, and for negative values that d = -1. The magnitude of L(d') denotes the reliability of that decision. Often the values of $L_e(d')$ due to the decoder has the same sign a $L_c(x) + L(d)$ and therefore acts to improve the reliability of L(d').

5.2.4 Principle of Iterative (Turbo) Decoding

With turbo codes, where two or more component codes are used, and decoding involves feeding outputs from one decoder to the inputs of other decoders in an iterative fashion, hard-output decoder would not be suitable [7]. That is because hard decisions into a decoder degrade system performance (compared with soft decision). Hence what is needed for the decoding of turbo codes is a soft-input/soft-output decoder. For the first decoding iteration of such a soft-input/soft-output decoder illustrated in Fig (5.2), one generally assumes the binary data to be equally likely, yielding an initial a priori LLR value of L(d) = 0 for the third term in Equation (5.11). The channel LLR value $L_c(x)$ is measured by forming the logarithm of the ratio of the values of l_1 and l_2 for a particular observation of x (see Fig (5.1)), which appears as the second term in equation (5.11). The output L(d') of the decoder in Fig (5.2) is made up of the LLR from the detector L'(d') and the extrinsic LLR output $L_e(d')$, representing knowledge gleaned from the decoding process. As illustrated in Fig (5.2), for iterative decoding, the extrinsic likelihood is fed back to the input (of another component decoder) to serve as a refinement of the a–priori probability of the data for the next iteration.

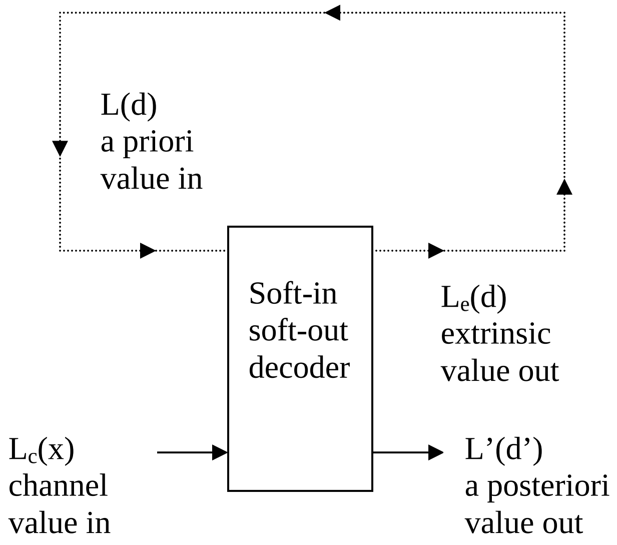


Fig (5.2): Soft input/soft output decoder (for a systematic code)

5.3 LOG-LIKELIHOOD ALGEBRA

To best explain the iterative feedback of soft decoder outputs, the concept of log-Likelihood algebra is introduced. For statistically independent data d, the sum of two log likelihood ratios (LLRs) is defined as

$$L(d_1) \otimes L(d_2) \equiv L(d_1 \oplus d_2) = \log_e [(e^{L(d_1)} + e^{L(d_2)})/(1 + e^{L(d_1)} e^{L(d_2)})] \quad (5.12)$$

$$\approx (-1) \times \text{sgn}[L(d_1)] \times \text{sgn}[L(d_2)] \times \min(|L(d_1)|, |L(d_2)|) \quad (5.13)$$

Where the natural logarithm is used, and the function sgn(.) represents the "polarity of". There are three addition operations in equation (5.12). The + sign is used for ordinary addition. The $\oplus$ sign is used to denote the modulo-2 sum of data expressed as binary digits. The $\otimes$ sign denotes log-likelihood addition, or equivalently, the mathematical operation described by equation (5.12). The sum of two LLRs denoted by the operator $\otimes$ is defined as the LLR of the modulo-2 sum of the underlying statistically independent data bits. Equation (5.13) is an approximation of equation (5.12) that will prove useful later in a numerical example. The sum of LLRs, as described by equations (5.12) or (5.13), yields the following interesting results when one of the LLRs is very large and very small:

$$L(d) \otimes \propto = -L(d)$$

And

$$L(d) \otimes 0 = 0$$

5.4 ENCODING WITH RECURSIVE SYSTEMATIC CODES

The basic concepts of concatenation, iteration, and soft decision decoding are applied to the implementation of turbo codes that are formed by the parallel concatenation of component convolutional codes.

A short review of simple binary rate ½ convolutional encoders with constraint length K and memory K-1 is in order. The input to the encoder at time k is a bit d_k, and the corresponding codeword is the bit pair (u_k, v_k), where

$$u_k = \sum_{i=0}^{K-1} g_{1i} d_{k-i} \quad \text{modulo-2,} \qquad g_{1i} = 0,1$$

and

$$v_k = \sum_{i=0}^{K-1} g_{2i} d_{k-i} \quad \text{modulo-2,} \qquad g_{2i} = 0,1$$

Where $G_1 = \{g_{1i}\}$ and $G_2 = \{g_{2i}\}$ are the code generators, and d_k is represented as a binary digit. This encoder can be visualized as a discrete-time finite impulse response (FIR) linear system, giving rise to the familiar nonsystematic convolutional (NSC) code, and example of which is shown in Fig (5.3). In this example, the constraint length is K = 3, and the two code generators are described by $G_1 = \{1\ 1\ 1\}$and $G_2 = \{1\ 0\ 1\}$.

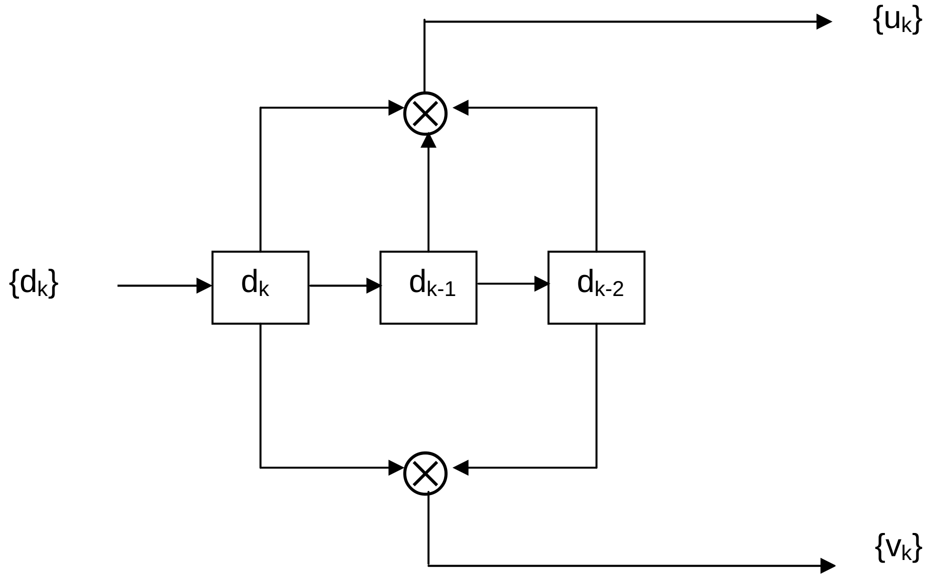


Fig (5.3): Nonsystematic convolutional (NSC) code

It is well known that at large E_b/N_0 values, it is generally the other way around. A class of infinite impulse response (IIR) convolutional codes has been proposed as building blocks for a turbo codes because previously encoded information bits are continually fed back to the encoder's input. For high code rates, RSC codes result in better error performance than the best NSC code by using a feedback loop, and setting one of two outputs (u_k or v_k) equal to d_k. Fig (5.4) illustrates an example of such an RSC code, with K=3, where a_k is recursively calculated as

$$a_k = d_k + \sum_{i=1}^{K=1} g'_i a_{k-i} \qquad \text{modulo-2}$$

and g'_i is equal to g_{1i} if $u_k=d_k$, and to g_{2i} if $v_k=d_k$.

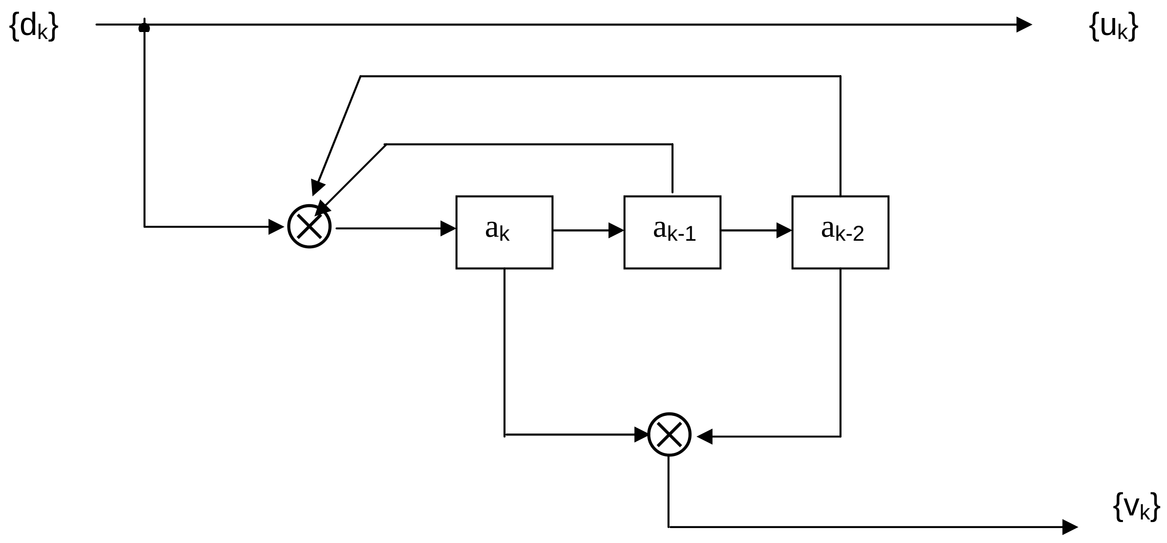


Fig (5.4): Recursive Systematic Convolutional (RSC) Code.

5.4.1 Concatenation of RSC Codes

Consider the parallel concatenation of two RSC encoders [18] of the type shown in Fig (5.4). Good turbo codes have been constructed from the component codes having short constraint length (K = 3 to 5). An example of such a turbo encoder is shown in Fig (5.5), where the switch yielding v_k provides puncturing, making the overall code rate ½. Without the switch, the code rate would be 1/3 [23]. The goal in designing turbo codes is to choose the best component codes by maximizing the effective free distance of the code [19]. At large values of E_b/N_0, this is tantamount to maximizing the minimum weight codeword. However, at low values of E_b/N_0, optimizing the weight distribution of the codewords is more important than maximizing the minimum weight codeword.

The turbo encoder in Fig (5.5) produces codewords from each of two component encoders. The weight distribution for the codewords out of this parallel concatenation depends on how the codewords from one of the component encoders are combined with codewords from other encoder.

Intuitively, we should avoid pairing low-weight codewords from one encoder with low-weight codewords from the other encoder. Any such pairings can be avoided by proper design of the interleaver.

If the component encoders are not recursive, the unit weight input sequence (0 0 … 0 0 1 0 0 …. 0 0) will always generate a low weight codew $\{u_k\}$ at the input of a second encoder for any interleaver design. In other words, the interleaver would not influence the output codeword weight distribution if the components codes were not recursive. However if the component codes are recursive, a weight-1 input sequence generates an infinite impulse response (infinite-weight output).

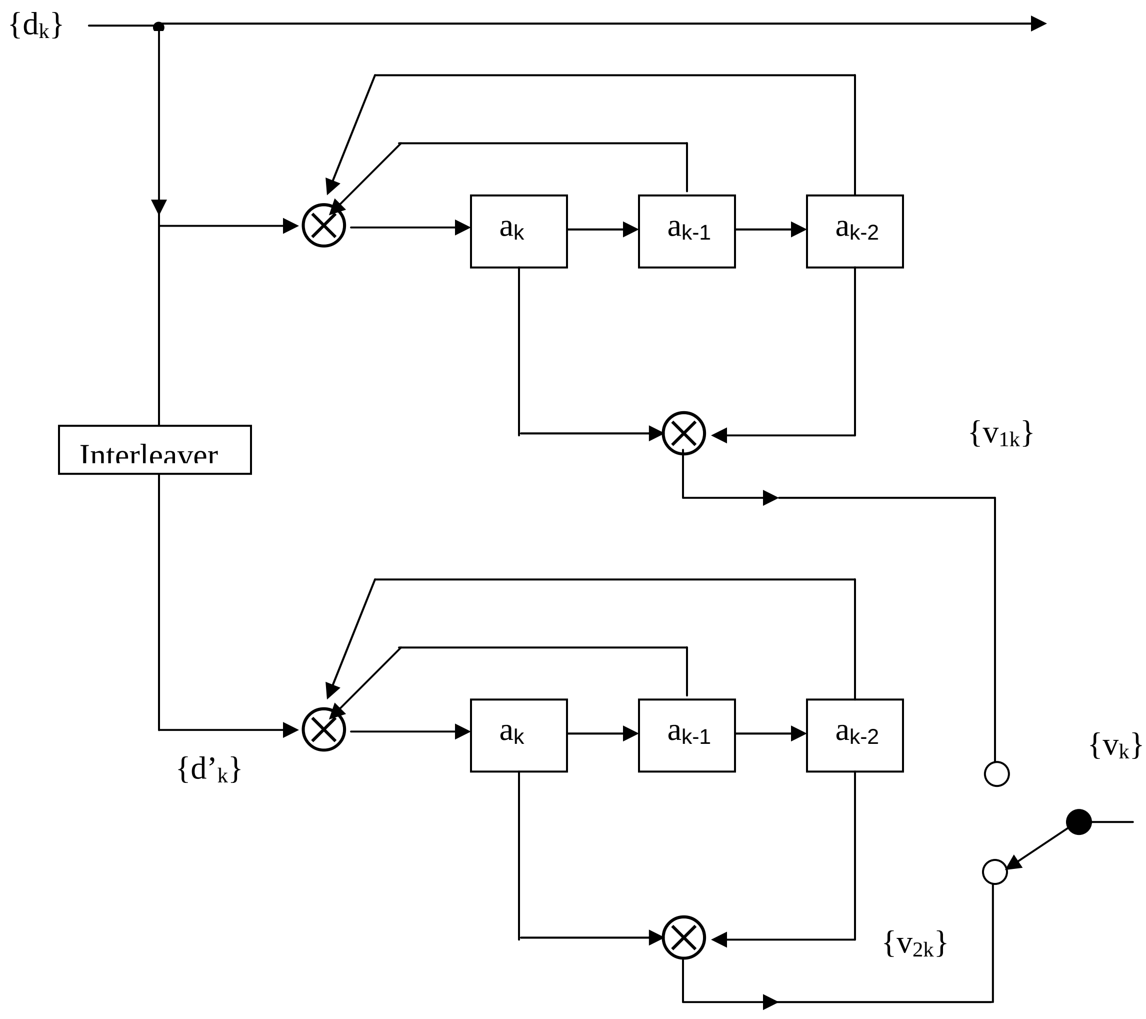


Fig (5.5): Parallel concatenation of two RSC encoders

The important aspect of the building blocks used in turbo codes is that they are recursive (the systematic aspect is merely incidental). It is the RSC code's IIR property that protects against the generation of low-weight codewords that cannot be remedied by an interleaver. One can argue that turbo code performance is largely influenced by minimum weight codewords that result from the weight-2 input sequence. The argument is that weight-1 inputs can be ignored since they yield large codeword weights due to the IIR encoder structure. For input sequences having weight-3 and larger, a property-designed interleaver makes the occurrence of low weight output codewords relatively rare.

5.5 A FEEDBACK DECODER

The Viterbi algorithm (VA) is an optimal decoding method for minimizing the probability of sequence error. Unfortunately, the (hard decision output) VA is not suited to generate the a posteriori probability (APP) or soft-decision output for each decoded bit. A relevant algorithm for doing this has been proposed by Bahl et. al. The Bahl algorithm was modified by Berrou, et. al. for use in decoding RSC codes [10]. The APP that a decoded data bit $d_k = i$ can be derived from the joint probability $\lambda_k^{i,m}$ defined by

$$\lambda_k^{i,m} = P\{d_k = i, s_k = m|R_1^N\} \tag{5.14}$$

where $s_k = m$ is the encoder state at time k, and R_1^N is a received binary sequence from time k = 1 through some time N.

Thus, the APP that a decoded data bit $d_k = i$, represented as a binary digit, is obtained by summing the joint probability over all states, as follows

$$P\{d_k = i|R_1^N\} = \sum_m \lambda_k^{i,m} \qquad i = 0,1 \tag{5.15}$$

Next, the log-likelihood ratio (LLR) is written as the logarithm of the ratio of APPS, as

$$L(d'_k) = \log \left[\frac{\sum_m \lambda_k^{1,m}}{\sum_m \lambda_k^{0,m}} \right] \qquad (5.16)$$

The decoder makes a decision, known as the maximum a posteriori (MAP) decision rule, by comparing $L(d'_k)$ to zero threshold,. That is

$$\begin{aligned} d'_k &= 1 \quad \text{if} \quad L(d'_k) > 0 \\ d'_k &= 0 \quad \text{if} \quad L(d'_k) < 0 \end{aligned} \qquad (5.17)$$

For a systematic code, the LLR $L(d'_k)$ associated with each bit d'_k can be described as the sum of the LLR of d'_k, out of the demodulator and of other LLRs generated by the decoder (extrinsic information), as was expressed in (5.12) and (5.13). Consider the detection of a noisy data sequence that stems from the encoder from the encoder of Fig (5.5), with the use of a decoder shown in Fig (5.6).

Assume binary modulation and a discrete memory-less Gaussian channel. The decoder input is made up of a set R_k of two random variables x_k and y_k. For the bits d_k and v_k at time k, expressed as binary numbers (1, 0), the conversion to received bipolar (+1, -1) pulses can be expressed as

$$x_k = (2d_k - 1) + i_k \qquad (5.18)$$

and

$$y_k = (2v_k - 1) + q_k \qquad (5.19)$$

Where i_k and q_k are two statistically independent random variables with the same variance σ^2, accounting for noise distribution. The redundant information y_k is demultiplexed and send to decoder DEC1 as y_{1k}, when $v_k = v_{1k}$, and to decoder

DEC2 as y_{2k}, when $v_k = v_{2k}$. When the redundant information of a given encoder (C1 or C2) is not emitted, the corresponding decoder input is set to zero.

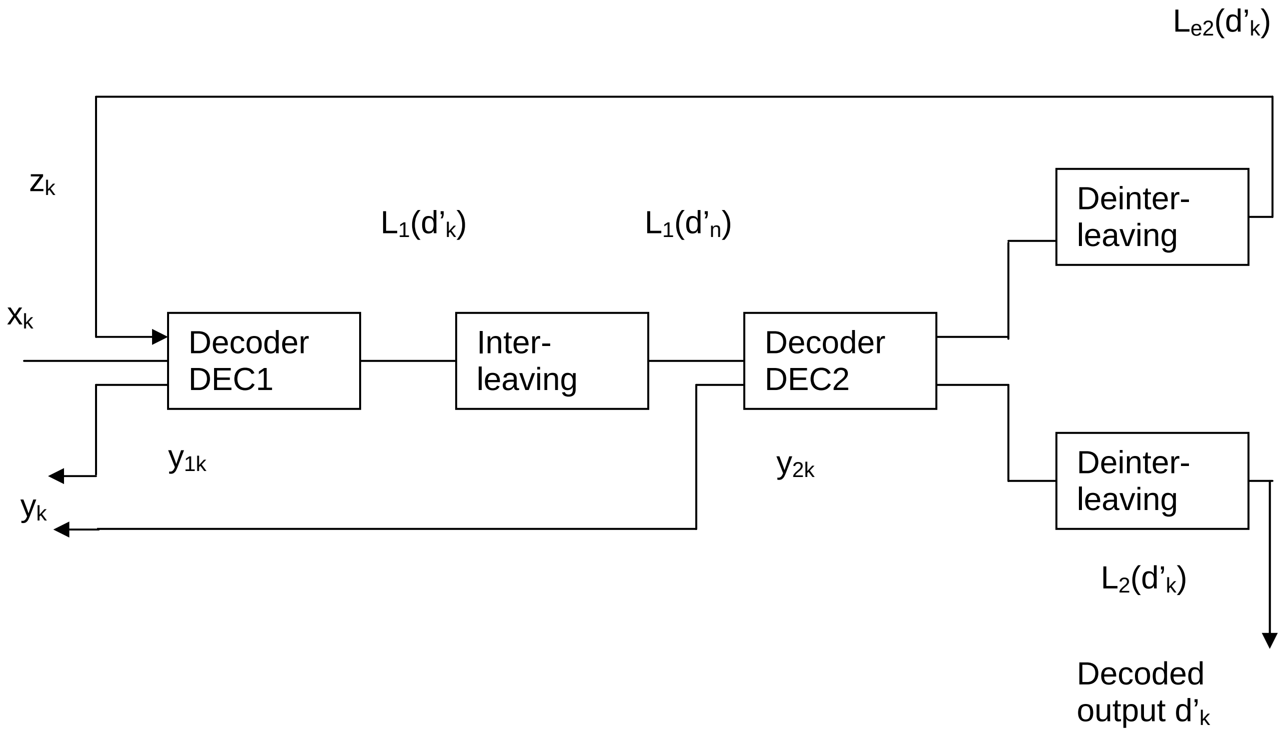


Fig (5.6): Feedback decoder

Notice that the output of DEC1 has an interleaver structure identical to the one used at the transmitter between the two component encoders. This is because the information processed by DEC1 is the no interleaved output of C1 (corrupted by channel noise). Conversely, the information processed by DEC2 is the noisy output of C2, whose input is the same data going into C1, however permuted by the interleaver. DEC2 makes use of the DEC1 output, provided this output is time ordered in the same way as the input of C2 (i.e., the two sequences into DEC2 must appear “in step” with respect to the positional arrangements of the signals in each sequence).

5.5.1 Decoding with a Feedback Loop

We rewrite equation (5.11) for the soft-decision output at time k, with the a priori LLR $L(d'_k)$ initially set to zero. This follows from the assumption that the data bits are equally likely. Therefore

$$L(d'_k) = L_c(x_k) + L_e(d'_k) \tag{5.20}$$

$$= \log\left[\frac{p(x_k|d_k = 1)}{p(x_k|d_k = 0)}\right] + L_e(d'_k) \tag{5.21}$$

where $L(d'_k)$ is the soft-decision output at the decoder, and $L_c(x_k)$ is the LLR channel measurement, stemming from the ratio of likelihood functions $p(x_k|d_k = i)$ associated with the discrete memory-less channel model. $L_e(d'_k) = L(d'_k)|_{xk=0}$ is a function of the redundant information. It is the extrinsic information supplied by the decoder and does not depend on the decoder input x_k. Ideally $L_c(x_k)$ and $L_e(d'_k)$ are corrupted by uncorrelated noise, and thus $L_e(d'_k)$ may be used as a new observation of d_k by another decoder to form an iterative process. The fundamental principal for feeding back information to another decoder is that a decoder should never be supplied with information that stems from its own input (because the input and output corruption will be highly correlated).

For the Gaussian channel, the natural logarithm in equation (5.11) is used to describe the channel LLR $L_c(x_k)$ which can be further written as

$$L_c(x_k) = \text{Loge}\left[\frac{\frac{1}{\sigma\sqrt{2\Pi}}\exp\left[\frac{-(x_k-1)^2)}{2\sigma^2}\right]}{\frac{1}{\sigma\sqrt{2\Pi}}\exp\left[\frac{-(x_k+1)^2)}{2\sigma^2}\right]}\right]$$

$$= \frac{-1}{2}\left[\frac{(x_k-1)^2}{\sigma^2}\right] + \frac{1}{2}\left[\frac{(x_k+1)^2}{\sigma^2}\right]$$

$$L_c(x_k) = \frac{2}{\sigma^2} x_k$$

Both decoders, DEC1 and DEC2 use the modified Bahl algorithm. If the inputs $L_1(d'_k)$ and y_{2k} to decoder DEC2 are statistically independent, then the LLR $L_2(d'_k)$ at the output of DEC2 can be written as

$$L_2(d'_k) = f[L1(d'_k)] + L_{e2}(d'_k)$$

with

$$L_1(d'_k) = \frac{2}{\sigma^2} x_k + L_{e1}(d'_k)$$

Where f[.] indicates a functional relationship. The extrinsic information $L_{e2}(d'_k)$ out of DEC2 is a function of the sequence $\{L_1(d'_k)\}_{n\neq k}$. Since $L_1(d'_n)$ depends on the observation R_1^N, then the extrinsic information $L_{e2}(d'_k)$ is correlated with the observations x_k and y_{1k}. Nevertheless, the greater $|n-k|$ is, the less correlated are $L_1(d'_n)$ and the observations x_k and y_k. Thus due to the interleaving between DEC1 and DEC2, the extrinsic information $L_{e2}(d'_k)$ and the observations x_k and y_{1k} are weakly correlated. Therefore, they can be jointly used for the decoding of bit d_k. In Fig (5.6), the parameter $z_k = L_{e2}(d'_k)$ feeding into DEC1 acts as a diversity effect in an iterative process. In general, $L_{e2}(d'_k)$ will have the same sign as d_k. Therefore $L_{e2}(d'_k)$ may increase the associated LLR and thereby improve the reliability of each decoded data bit.

CHAPTER 6
LOG-MAP-BASED ITERATIVE TURBO DECODER

6.1 INTRODUCTION

In 1948, Claude E. Shannon proved that it is possible to transmit information with arbitrary high reliability provided that the rate of transmission R does not exceed a certain Value C known as Shannon capacity or Shannon limit. However, before the introduction of Turbo code, designing channel coding for practical communication system aimed at cut-off rate instead of the ultimate Shannon capacity: this is because previous attempts to exceed the cut-off rate usually result in inefficient channel coding schemes, where very large additional complexity is required to obtain little transmission improvement.

In 1993, a parallel-concatenated channel coding scheme, named Turbo code, was proposed by Berrou et al, Which achieves transmission performance a few tenths of a dB from Shannon limit when applied to a BPSK transmission over AWGN channel [10]. More importantly, by employing a sub-optimal iterative decoding structure and soft-in/soft-out (SISO) maximum a posteriori (MAP) decoding algorithm, the near-capacity performance is achieved with a feasible decoding complexity. Because of its excellence performance, turbo code has been employed in several transmission systems such as CDMA2000, WCDMA, and the next generation ADSL systems [8], [31], [32].

With the application of turbo coding to more communication systems, low complexity implementation of turbo decoder becomes a more popular and challenging topic [1], [6], [9]. Although employing iterative decoding significantly reduces the decoding complexity, compared to a maximum-likelihood (ML) decoder, the MAP decoding algorithm is still very computation-intensive in comparison with the traditional Viterbi algorithm (VA). Besides, turbo codes with good performance normally introduce a long encoding/decoding delays because of the long interleaver length in both the encoder and the decoder. For delay sensitive applications, i.e., real-time applications, this delay must be kept very low [27], [29]. However, the delay is usually reduced at the cost of performance degradation.

The considerations in implementing an efficient turbo decoder is to choose a proper SISO algorithm and interleaver design [11], [2], [20]. The original turbo decoder consists of SISO decoders based on MAP algorithm, which involves a large amount of multiplications, exponentials, and logarithm computations. Implementations of these mathematical operations are usually quite complex especially in VLSI design [6], [16]. Suboptimal, but much simpler, varieties of MAP algorithms, Max-log-MAP and Log-MAP, were proposed in order to reduce the computational complexity. Another Suboptimal SISO algorithm is the soft output Viterbi algorithm (SOVA) [7], which is derived from the traditional VA. Each of these Suboptimal SISO algorithms brings certain level of complexity reductions with some performance degradations. In a turbo decoder design, the SISO algorithm should be selected as a compromise between the decoding performance and implementation complexity.

6.2 TURBO CODES

A typical turbo code consists of two systematic convolutional codes separated by an interleaver [15]. A generic encoding structure of a binary turbo code with two identical rate-1/2 constituent codes (CCs) is shown in Fig (6.1).

The turbo code encoder processes a block of K information bits each time. The first CC takes the block as input and produces K parity bits. The interleaved version of the same block is input to the second CC, which produces another block of K parity bits. Typically, the output of a turbo encoder is the multiplex of the information bit sequence and two parity bit sequences. Hence for every K information bits, there are 3K output bits, resulting in a code rate of 1/3. The parity bits from the two CCs can be punctured alternatively to obtain a code rate of ½ with some performance lost. Higher density puncturing schemes can be applied to further increase the code rate, at more performance degradation.

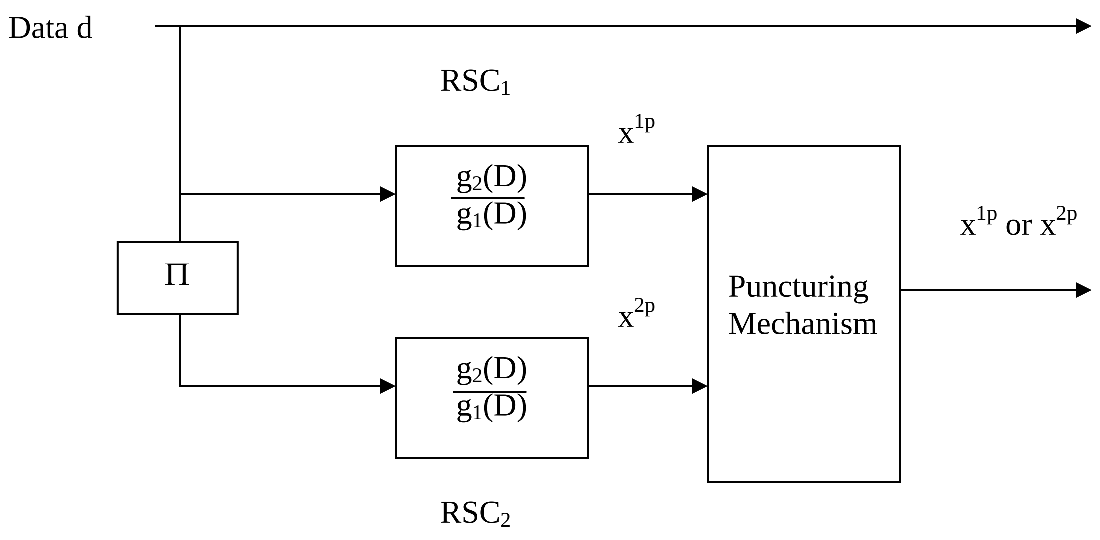


Fig (6.1): Typical structure of turbo encoder

Theoretical performance analysis of turbo code always assumes using a maximum-likelihood decoder at the receiver. However, ML decoder is often too complex to be implemented for turbo decoding because of the very complex trellis structure caused by the interleaver between the two CCs. Iterative decoding is proposed in as a Suboptimal but feasible alternative for turbo decoding [11]. The basic iterative decoding structure corresponding to the turbo encoder shown in Fig (6.1) is depicted in Fig (6.2). The two constituent decoders are used to perform SISO decoding over the coded sequences generated by the two CCs respectively, where the reliability information is exchanged between them during the decoding iterations.

As shown in Fig (6.2), the received noisy sequence is demultiplexed into three sequences: the systematic sequence y^s and two parity sequences y^{1p} and y^{2p}. One SISO decoder takes y^s and y^{1p} (or y^{2p}) as inputs and computes the log-likelihood ratio (LLR) of each information bit based on the trellis structure of the CC, which is defined for the kth information bit d_k, as

$$L(dk) = \log\left[\frac{\Pr\{d_k = 1|Y\}}{\Pr\{d_k = 0|Y\}}\right] \qquad (6.1)$$

Where Y is the received symbol block. The decision $d_k = 1$ is made for a positive LLR and $d_k = 0$ for a negative LLR. The absolute value of the LLR represents the reliability of this decision. The larger is the absolute value, the more reliable is the decision.

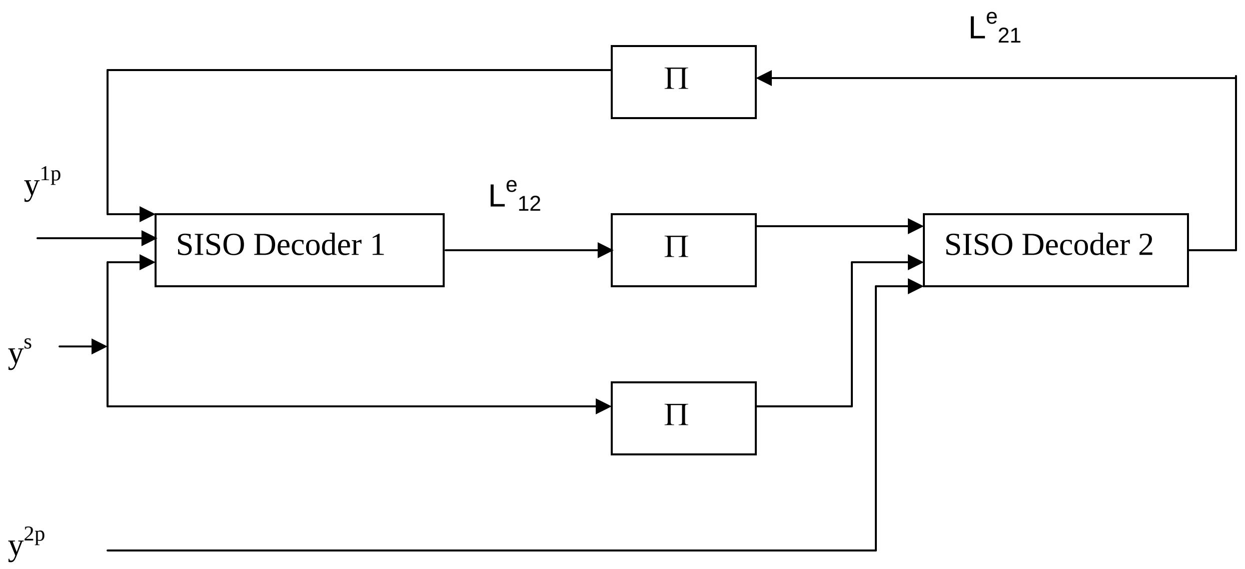


Fig (6.2): Block diagram of iterative (turbo) decoder (Π-interleaver)

Several SISO decoding algorithms are proposed in the literature. MAP algorithm is an optimal SISO algorithm in the sense of minimizing the symbol error probability [12], but is computationally intensive. A simplified version of MAP, Max-Log-MAP algorithm, achieves a significant complexity reduction with small performance degradation [1]. A modified Max-Log-MAP algorithm, the log-MAP algorithm, provides nearly optimum performance while still keeping the low complexity [1]. Another SISO decoding algorithm, the SOVA, is obtained by making some modifications to the traditional VA to generate the soft reliability information. Using any of the above SISO decoding algorithms, with a proper interleaver, it has been shown that the output LLR of the SISO decoder can be divided into three approximately independent terms

$$L(d_k) = L_c(d_k) + L_{apri}(d_k) + L_e(d_k) \qquad (6.2)$$

Where $L_c(d_k)$ is the channel information, $L_{apri}(d_k)$ is the a priori information of d_k, and $L_e(d_k)$ is the extrinsic information, which is represented by L^e_{12} and L^e_{21} in Fig (6.2). The channel information depends only on the noise corrupted systematic symbol, while the extrinsic information is calculated based on the trellis structure together with the other systematic symbols and parity symbols. It is important that the extrinsic information $L_e(d_k)$, be uncorrelated (or weakly correlated) of $L_c(d_k)$ and $L_{apri}(d_k)$, since it will be used as a priori information by the other SISO decoder. However, with the number of iterations increasing, the correlation increases and the performance gained form additional iteration becomes less. Until certain stage, further iteration brings little (if any) performance improvement: this is when the iterative decoding should stop.

Although iterative decoding is suboptimum in the sense of achieving the maximum likelihood decoding results, it has been shown by simulations to approach ML decoding performance provided the number of iterations is large enough.

6.3 TURBO DECODING ALGORITHMS

The complexity related to turbo coding mainly comes from the iterative turbo decoding process [6]. The turbo encoder basically consists of only two shift registers and an interleaver, whose complexity is negligible compared to any SISO decoding process. As aforementioned, there are several SISO algorithms that can be selected in turbo decoder implementation: they require different complexities and offer different decoding performances

6.3.1 SISO Decoding Algorithms

The first SISO algorithm used in turbo decoding is the MAP algorithm. The MAP algorithm is designed to produce the LLR of each information bit, as defined in (6.1). In MAP, the LLR is calculated as

$$L(d_k) = \ln \left[\frac{\sum_{S_k}\sum_{S_{k-1}} \gamma_1(y_k, S_{k-1}, S_k).\alpha_{k-1}(S_{k-1}).\beta_k(S_k)}{\sum_{S_k}\sum_{S_{k-1}} \gamma_0(y_k, S_{k-1}, S_k).\alpha_{k-1}(S_{k-1}).\beta_k(S_k)} \right] \tag{6.3}$$

Where α is the forward recursion path metrics, β is the backward recursion path metrics and γ is the branch metrics. The forward path metrics can be calculated recursively as

$$\alpha_k(S_k) = \left[\frac{\sum_{S_{k-1}}\sum_{i=0}^{1} \gamma_i(y_k, S_{k-1}, S_k).\alpha_{k-1}(S_{k-1})}{\sum_{S_{k-1}}\sum_{i=0}^{1} \gamma_i(y_k, S_{k-1}, S_k).\alpha_{k-1}(S_{k-1})} \right] \tag{6.4}$$

Where $\alpha_0(S_0) = 1$ and $\alpha_0(S_i) = 0$ for $i \neq 0$, when both CCs in the turbo encoder are terminated. The backward path metric, β, is calculated in a similar manner, except in the reverse direction. The branch transition probabilities are calculated as

$$\begin{aligned} \gamma_i[(y_k^s,y_k^p), S_{k-1}, S_k)] = {} & q(d_k = i|S_k, S_{k-1}) \\ & .p(y_k^s|d_k = i) \\ & .p(y_k^p|d_k = i,S_k,S_{k-1}) \\ & .\Pr\{S_k|S_{k-1}\}. \end{aligned} \tag{6.5}$$

The value of $q(d_k = i|S_k, S_{k-1})$ is either one or zero depending on whether there is a transition from state S_{k-1} to S_k with input d_k. The a priori information is used to calculate $\Pr\{S_k|S_{k-1}\}$.

Calculation of the LLR requires both α that is calculated recursively from the beginning to the end of block, and β that is calculated from end to the beginning. The decoding process is, therefore, performed as follows:

- The decoder starts calculating α in the sequential order of the input block.
- When the decoder reaches the end of the block, i.e., all αs are calculated, it starts computing the value of β.
- Whenever the β values for an information bit is computed, the LLR is calculated

It is observed that the large amounts of complicated mathematical operations are required for MAP decoding, including multiplications, exponentials, and logarithm computations. To avoid these operations, MAP decoding can be performed in the logarithm domain, where the multiplication becomes addition. The logarithm and exponential computations can be avoided by using following approximation

$$\ln(e^{\delta 1}+e^{\delta 2}+\ldots\ldots\ldots\ldots ++e^{\delta n}) \approx \max_{i\in(1,\ldots,n)} \delta \qquad (6.6)$$

Equation (6.3)-(6.5) then becomes

$$L(dk) \approx \max_{(s_k,s_{k-1})} \{\gamma'_1(y_k, S_{k-1}, S_k) + \alpha'_{k-1}(S_{k-1}) + \beta'_k(S_k)\} - \max_{(s_k,s_{k-1})} \{ \gamma'_0(y_k, S_{k-1}, S_k) + \alpha'_{k-1}(S_{k-1}) + \beta'_k(S_k)\} \qquad (6.7)$$

$$\alpha'_k(S_k) = \max_{(s_{k-1},i)} \{\gamma'_i(y_k, S_{k-1}, S_k) + \alpha_{k-1}(S_{k-1})\} \qquad (6.8)$$

and

$$\gamma' = \frac{2y_k^s x_k^s(i)}{N_0} + \frac{2y_k^p x_k^p(i, S_k, S_{k-1})}{N_0} + \ln\Pr[S_k \mid S_{k-1}] + K \qquad (6.9)$$

Where α', β', γ' are the logarithms of α, β, γ. The MAP algorithm performed in the logarithm domain with the approximation shown in (6.6) becomes Max-Log-MAP algorithm. In General 0.5 dB performance degradation is observed because of this simplification. The inferior performance of the Max-Log-MAP algorithm comes from the simplification made in (6.6). The logarithm of the sum of two exponents can be accurately calculated as

$$\begin{aligned} \ln(e^{\delta 1}+e^{\delta 2}) &= \max(\delta_1,\delta_2) + \ln(1+e^{|\delta 1-\delta 2|}) \\ &= \max(\delta_1,\delta_2) + f(|\delta_1-\delta_2|). \end{aligned} \tag{6.10}$$

The only difference between this equation and (6.6) is a modification factor $f_c = f(|\delta_1-\delta_2|)$. The Max-Log-MAP algorithm with this modification is called Log-MAP. A different SISO decoding algorithm, SOVA, was derived from the conventional VA. In SOVA, the difference between the path metrics entering the same trellis state node is used to generate the reliability of the information bits, i.e., the software decision. The complexity of SOVA is smaller than that of Max-Log-MAP; while the performance of turbo decoding using SOVA is inferior to that obtained using Max-Log-MAP.

6.3.2 Comparison of SISO Algorithms

It is observed that turbo decoding with MAP algorithm provides the best performance out of all the decoding algorithms. Performance of Log-MAP algorithm at BER of 10^{-4} is very close from that obtained using the MAP algorithm and is approximately 0.6 dB better than that obtained using SOVA. The Max-Log-MAP is only about 0.06 dB better than the SOVA at BER of 10^{-4}. it is noticed that at a BER of 10^{-4} , the performance of turbo decoding with Max-Log-MAP is about 0.4 dB better than with the SOVA algorithm.

It has been observed that in Max-Log-MAP and Log-MAP, the most frequently used mathematical operations include additions, inversion, max operation and table lookups. As per these operations Log-MAP is more computational intensive compared to Max-Log-MAP algorithm but it has good performance than Max-Log-MAP, so Log-MAP is a best compromise with reasonable complexity.

6.4 LOG-MAP-BASED TURBO DECODER IMPLEMENTATION

6.4.1 Introduction

During the turbo decoder, design there is a trade-off between the memory requirement and processing delay. The memory requirement can be reduced by reusing same memory modules as much as possible. However, since, at any time, a memory element can perform the function for one specific module only, different modules have to use it one after another, which results in longer decoding delay. Providing a separate copy of the same hardware component to any module can significantly reduce the delay but using same memory elements for reducing memory requirement slow down the speed. Hence saving memory requirement occurs at the expense of slower speed.

In this thesis, the turbo decoder is designed to minimize the memory requirements for algorithm implementation as well interleaver design, and the decoding speed being a second consideration. Therefore memory modules are shared as much as possible during execution.

6.4.2 Interleaver Structure for Turbo Codes with Reduced Storage Memory Requirements

6.4.2.1 Introduction

A turbo code typically consists of two recursive encoders in parallel, separated by an interleaver as shown in Fig (6.1). The design parameters of a turbo code are primarily the generator polynomials of the constituent encoders, normally chosen to be identical, and the particular choice of interleaver mapping [30]. The interleaver structure used here is referred to as an odd-even symmetric structure, which reduces the memory requirement with much more than 50% compared to storing the entire interleaver vector [2].

6.4.2.2 Design

Let the odd-even symmetric interleaver rule be represented by a vector of N integers, $\Pi = \{\Pi(1)\ \ \Pi(2)\ \ldots\ldots\Pi(N)\}$, where $\Pi(i) = j$ indicates that input position i is interleaved to position j and N is the size of the interleaver. The interleaver structures presented here impose certain restrictions on the permissible choices of the mappings $\Pi(i)$. Consider an interleaver rule that swaps pairs of positions i.e. a symmetric interleaver. If all the pairs are known, the interleaver rule is known. Since there are only N/2 such pairs, if organized properly, the storage of these pairs requires less memory than storing an entire interleaver vector with N addresses. One possible organization strategy is to require every position in the first half of the input sequence to be swapped with a position in the second half. However, this restriction severely reduces the design freedom of the interleaver, notably deteriorating the error correcting performance of the code. There is however other sequence partitions that yield a simple organization of the swapping pairs, without degrading the interleaver performance. One such partition is to swap every odd position with even position, and vice versa. The interleaver structure, denoted odd-even symmetric, is thus achieved with the following two restrictions:

1. $i \bmod 2 \neq \Pi(i) \bmod 2, \forall i$ (odd to even)
2. $\Pi(i) = j \Rightarrow \Pi(j) = I$ (symmetry)

With these restrictions, it is sufficient to store the interleaver rules for all the odd positions, since by performing swaps; the even positioned bits are automatically interleaved.

Assume that only the odd positions in the interleaver vector are stored. All the stored addresses are then even integers, implying that the least significant bit (LSB) in the binary representation of each address is always zero. Thus the LSB need not be stored, which offers additional memory savings if the interleaver rule is stored with custom made memory cells. This shift of the binary representation corresponds to dividing each number by 2, so that the stored vector consists of N/2 integers ranging from 1 to N/2. This vector will in the following be denoted Π', and is given by $\Pi' = (2i-1)/2$, $i \in (1,2\ldots..,N/2\}$.

As an example, the swapping pairs of an 8-bit odd-even symmetric interleaver are illustrated in Fig (6.3). The shown vector is $\Pi = \{6\ 3\ 2\ 7\ 8\ 1\ 4\ 5\}$. And reduced memory requirement vector is $\Pi' = \{3\ 1\ 4\ 2\}$. The implementation of the interleaving rule of an odd-even symmetric interleaver is straightforward: elements at even positions are interleaved by storing them sequentially and reading them in order specified by Π'; elements at odd positions are interleaved by storing them in the order specified by Π' and reading them sequentially.

Input positions	**1**	2	**3**	4	**5**	6	**7**	8
Interleaved position	**6**	3	**2**	7	**8**	1	**4**	5

Fig (6.3): Example of an 8-bit odd-even symmetric interleaver. Each odd position in the input sequence is mapped to an even position, and vice versa. Further, if input i is mapped to position j, then input j is mapped to position i (symmetry)

As an example, we study the interleaving of the extrinsic outputs produced by the first constituent decoder. For illustrative purposes, it is suitable to partition the memory used to store the extrinsic information between the decoders into two logically separated memory areas. A and B. With these, odd extrinsic outputs of the form 2n-1, $n \in \{1,2,\ldots\ldots,N/2\}$ are stored at address Π'(n) in memory A, while even outputs, an, $n \in \{1,2,\ldots\ldots,N/2\}$ are stored at address n in memory B. the second constituent decoder performs a similar action when reading its extrinsic inputs: odd inputs are read from memory B at address Π'(n), and even inputs are read from memory A at address n. Such an interleaver implementation is illustrated in Fig (6.4). The deinterleaving implementation is identical, due to symmetric property.

Note: The interleaved structure described above saves approximately 50% memory. If total no of bits are N then memory requirement is about N/2 bits. But in C implementation, the interleaver uses only 8 bytes (memory for saving 4 integer values) memory; and these integers are used each time for implementing interleaving structure (iterations). This results into huge reduction of memory requirement, as these four integer values are used iteratively for interleaving any number of bits in input sequence.

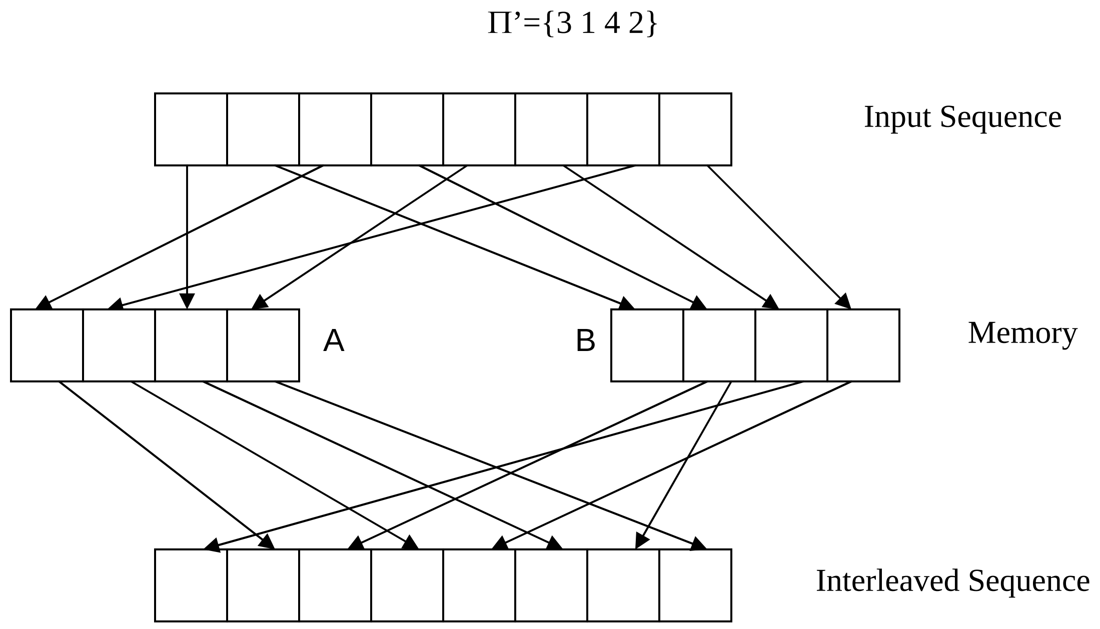


Fig (6.4): implementation example of an 8-bit odd-even symmetric interleaver. The interleaver rule is stored by the 4-element vector Π' = {3 1 4 2}.

6.4.3 Log-MAP Turbo Decoder

The block diagram of the turbo decoder is shown in Fig (5.6) and is reproduced in Fig (6.5) below. The major components are two decoders (Log-MAP) and interleaver and deinterleaver blocks. As we have used odd-even symmetric interleaver, so the blocks named interleaver and deinterleaver could be replaced with same block named Interleaver/Deinterleaver [2].

It is assumed that the received symbol sequence is first demultiplexed into three sequences: systematic sequence, and two parity sequences. After the Log-MAP decoder finished decoding over one block of data, it writes the result to LLR memory (Array in C), which will be used as the a priori information during the next SISO decoding process. The interleaving /deinterleaving processes are implemented implicitly by reading from the pattern stored in memory array.

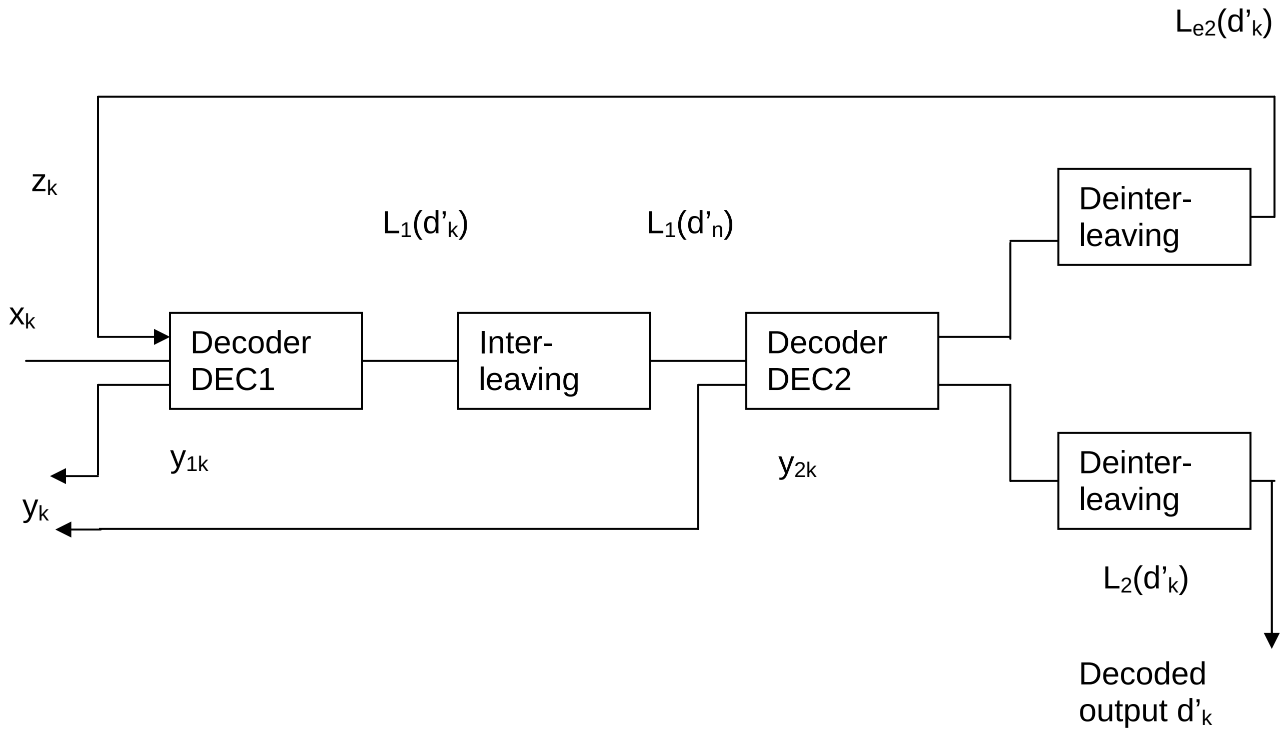


Fig (6.5): Log-MAP Turbo Decoder

The algorithm is as following. First of all, the received data for each constituent codes are divided into several contiguous non-overlapping sub-blocks; so called windows [25], [4], [28]. Then, each window is decoded serially using the Log-MAP algorithm from the last window (here each window consists of 8 bits). However the values of alpha (first iteration) for each window is calculated each time starting from the first window. This is very time consuming process, but leads to huge reduction in memory requirements, because there is no need of storing the terminating alpha values of each window. However initial values of beta variables come from previous window. In the next iteration the branch metric is recalculated using a-priori information from the last iteration and then the alpha and beta variables are recalculated using new branch metric values for that window. The block diagram of Log-MAP decoder used for this algorithm is shown below.

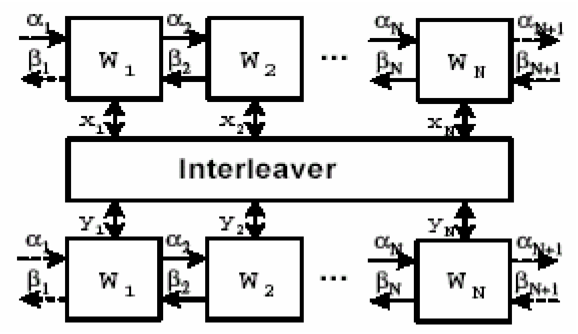


Fig (6.6): Log-MAP decoder structure

The decoding of each window includes: a branch metric calculation module, forward and backward path metric calculation modules, a LLR calculation module, and some control logics. The Log-MAP decoding is performed as follows

1. Starting from the beginning of each block, the SISO decoder calculates the forward path metrics and stores the values for the required window.
2. After calculating the forward path metric, the decoder calculates the backward path metric in the backward direction, and stores the values for the required window
3. After calculation of forward and backward path metric, the decoder calculates the LLR.
4. It repeats the step from 1 to 3 some number of times (usually 4 to 5 times) and uses LLR value calculated in previous iteration o enhance the result. It writes the finally calculated LLR value to the output file.

The γ-calculation module calculates the path metric using the input systematic and parity symbols and the a priori information. The input systematic and parity symbols are already scaled by the SNR. Therefore this calculation only contains additions and inversions. Note that there are only four possible γ values corresponding to the four possible ($x'^{s}_{k}(i)$ and $x'^{p}_{k}(i, S_k, S_{k-1})$) combinations. Therefore the γ-calculation module is designed to calculate all the four values at the same time and saves them as four float values. This can greatly simplify the α/β/LLR calculations since they need to find the corresponding γ values instead of calculating in each step.

Calculation of the forward path metric α is performed according to (6.8), where the max is replaced by max* that includes the modification factor shown in (6.10). For Log-MAP algorithm and rate ½ CCs calculation of $\alpha(s_k)$ at time k is performed as

$$\alpha(s_k) = \max\{\alpha(S^0_{k-1})+\gamma(S_{k-1}, S_k, d_k = 0), \alpha(S^1_{k-1})+\gamma(S_{k-1}, S_k, d_k = 1)\}+ \\ f\,|\{(\alpha(S^0_{k-1})+\gamma(S_{k-1}, S_k, d_k = 0)) - (\alpha(S^1_{k-1})+\gamma(S_{k-1}, S_k, d_k = 1))\}| \qquad (6.11)$$

Where s^i_{k-1} is the trellis state at time (k-1) that has a transition to state s_k at time k, caused by an input of i, $i \in \{0, 1\}$. It is observed that each $\alpha(S_k)$ is calculated by an add-compare-select-offset (ACSO) operation over two previous α values and two branch transition metrics. The ACSO unit performs the following calculation

$$r = \max\{a+b, c+d\} + f(|(a+b)-(c+d)|) \qquad (6.12)$$

Where {a, b, c, d} are four inputs, r is the single output, and f(|x|) is the modification factor defined in (6.10). Given that the α metrics at time (k-1) are stored in 2^M memory locations at the end of last α calculation operation and the four γ values have been saved in four memory locations, the implementation objective becomes to efficiently find the four inputs of the ACSO used to calculate each α from these saved values. This is implemented using 2^M entry

LUTs, one for the transition caused by an input of “0” (LUT0) and for an input of “1” (LUT1), where the jth entry of the LUTi, i $\in$ {0, 1}, contains the value of s^{i}_{k-1} to calculate $\alpha(s_k = j)$. These two LUTs actually contains all the information about the trellis structure of this CC. When the trellis structure is changed, the α-calulation operation can be updated by simply updating the content of the LUTs.

The backward path metric calculation module has exactly the same data structure as the α calculation module. However, the two LUTs are different from the LUTs used in α calculation module.

Calculation of LLR/Ext. values requires 2^M α values, 2^M β values, and all possible γ values. Combining (6.7) and (6.10), the LLR/Ext is calculated as

$$L(d_k) = \max^*_{(s_k,\, s_{k-1})}(\gamma'_1(y_k, S_{k-1}, S_k) + \alpha'_{k-1}(S_{k-1}) + \beta'_k(S_k)) - \max^*_{(s_k,\, s_{k-1})}(\gamma'_0(y_k, S_{k-1}, S_k) + \alpha'_{k-1}(S_{k-1}) + \beta'_k(S_k)) \qquad (6.13)$$

Where max* stands for multiple input ACSO operations. Each operand in the first max* corresponds to one trellis transition from a S_{k-1} to a S_k caused by an input information bit 1, while each operand in the second max* corresponds to one trellis transition from a S_{k-1} to a S_k caused by an input information bit ‘0’. Since there are only one transition coming out of a state caused by an input 1 or 0, both max* operations contain only 2^M operands.

6.5 ALGORITM: LOG-MAP-BASED ITERATIVE TURBO DECODER

1. Start
2. Define a variable named infin with a very small negative value.
3. Define a 2-D array of integers of size 2×8 (LUTFS) that work as Look-up table for implementing trellis structure for forward state metrics, with contents {(0,2,5,7,1,3,4,6),(1,3,4,6,0,2,5,7)}.
4. Define a 2-D array of integers of size 2×8 (LUTBS) that work as Look-up table for implementing trellis structure for reverse state metrics, with contents {(0,4,1,5,6,2,7,3),(4,0,5,1,2,6,3,7)}.
5. Take a file pointer (ifp) and associate it with a text file-containing float values after encoding and passing through AWGN channel.
6. Take a file pointer (ofp) and associate it with an empty output text file.
7. Take an array of integer (inleav) containing 4 integer values {3,1,4,2} for implementing odd-even symmetric interleaver structure.
8. Count the number of float values in the input text file and store in variable n.
9. Declare a reverse state metric rsmet of size 8×9 of type float and initialize rsmet[k][8] to infin with k varying from 1 to 7 and rsmet[0][8] to 0.0.
10. Let l=n/24
11. Initialize a priori information to 0.0

```
    (i)     for k =0 to 7
    (ii)    apri[k]=0.0
    (iii)   end for
```

12. Let big=0
13. Point file pointer to the first character in input file
14. Initialize the forward state metric

```
    (i)     for k =0 to 7
    (ii)    fsmet[k][0]=infin
    (iii)   end for
    (iv)    fsmet[0][0]=0.0
```

15. Let i =0

```
16. Let j = 0
17. Read three consecutive values from the input file and store them in three
    variables ch, ch1, ch2
18. Calculate branch metric values
        (i)     if i = l-1 is true then
                bmet[0][j/3]= -ch-ch1-apri[j/3]
                bmet[1][j/3]= -ch+ch1-apri[j/3]
                bmet[2][j/3]= ch-ch1+apri[j/3]
                bmet[3][j/3]= ch+ch1+apri[j/3]
                else
                bmet[0][j/3]= -ch-ch1
                bmet[1][j/3]= -ch+ch1
                bmet[2][j/3]= ch-ch1
                bmet[3][j/3]= ch+ch1
                end if
19. If j>=24 is true then go to step 23.
20. j = j+3
21. Repeat step 17 to 20.
22. Calculate forward state metric values
        (i)     for k = 1 to 8
        (ii)    for j = 0 to 8
        (iii)   if j is divisible by 2 is true then
                fsmet[j][k]=max(bmet[0][k-1]+fsmet[LUTFS[0][j]][k-1],bmet[3][k-
                1]+fsmet[LUTFS[1][j]][k-1])
                else
                fsmet[j][k]=max(bmet[1][k-1]+fsmet[LUTFS[0][j]][k-1],bmet[2][k-
                1]+fsmet[LUTFS[1][j]][k-1]
                end if
        (iv)    end for (j)
            (v)     end for (k)
23. If i = l -1 is true then go to step no 30.
```

```
24. Find minimum value among 8 forward state metric values using
    min=minimum (fsmet, k-1)
25. Normalize and assigning the forward state metric values to the next block
      (i)     for k =0 to 7
      (ii)    fsmet[k][0]=fsmet[k][8]-min
      (iii)   end for
26. Is i > = l, if true then go to step 30.
27. i = i + 1
28. Repeat step 16 to 27.
29. Assign the last calculated rs metric values to the first row of next block . (if
    not calculated till now than this will assign the initial values)
30. Calculate the reverse state metric values
      (i)     for k= 7 down to 0
      (ii)    for j 0 to 7
      (iii)   if (j=0) or (j=1) or (j=4) or (j=7) is true then
              rsmet[j][k]=max(bmet[0][k]+rsmet[LUTBS[0][j]][k+1],bmet[3][k]+
              rsmet[LUTBS[1][j]][k+1])
              else
              rsmet[j][k]=max(bmet[1][k]+rsmet[LUTBS[0][j]][k+1],bmet[2][k]+
              rsmet[LUTBS[1][j]][k+1]
          end if
      (iv) end for (j)
      (v)  end for (k)
31. Calculate Likelihood ratio
      (i)     for k = 0 to 7
      (ii)    Let num1=fsmet[0][k]+rsmet[4][k+1]+bmet[3][k]
      (iii)   Let den1= fsmet[0][k]+rsmet[0][k+1]+bmet[0][k]
      (iv)    For j = 1 to 7
      (v)     If j=2 or j = 3 or j = 6 or j = 7
                  num = fsmet[j][k]+rsmet[LUTBS[1][j]][k+1]+bmet[2][k]
                  den = fsmet[j][k]+rsmet[LUTBS[0][j]][k+1]+bmet[1][k]
```

```
            else
            num = fsmet[j][k]+rsmet[LUTBS[1][j]][k+1]+bmet[3][k]
            den = fsmet[j][k]+rsmet[LUTBS[0][j]][k+1]+bmet[0][k]
            end if
        (vi)    maxnum = max(num, num1)
        (vii)   maxden = max(den, den1)
        (viii)  end for (j)
        (ix)    app[k]=maxnum-maxden
        (x)     end for (k)
32. Calculate external LR ratio
        (i)     for k = 0 to 7
        (ii)    exlr[k]=app[k] - apri[k]
        (iii)   end for
33. Apply interleaving structure to the exlr array and to get apri information.
34. Normalize and assign the reverse state metric values for the next iteration
    for decoder 1.
35. Initialize the reverse state metric for decoder 2
36. Repeat step 14 to 17
37. Apply interleaving pattern to the input sequence read (24 bit at a time).
38. for j = 0 to 7
39. if i = l-1 is true then
    bmet[0][j] = -in[j]-apri[j]-yp[j]
    bmet[1][j] = -in[j]-apri[j]+yp[j]
    bmet[2][j] = in[j]+apri[j]-yp[j]
    bmet[3][j] = in[j]+apri[j]+yp[j]
    else
    bmet[0][j] = -in[j]-yp[j]
    bmet[1][j] = -in[j]+yp[j]
    bmet[2][j] = in[j]-yp[j]
    bmet[3][j]= in[j]+yp[j]
    end if (where in is interleaved sequence and yp is same as ch2)
```

40. end for(j)
41. Repeat step 21 to 32.
42. Apply interleaving structure to exlr to get apri information and to app to get lr ratio
43. Normalize and assign the reverse state metric values for the next iteration of decoder 2.
44. is big>= limit , if true then go to step 45
45. big = big+1
46. Repeat step 13 to 45
47. Calculate the decoded bit values

```
    (i)     for k = 0 to 7
    (ii)    if lr[k] > 0.0 then
            decod[k] = 1
            else
            decod[k] = 0
                end if
    (iii)   end for
```

48. Write the decoded byte string to the output file.
49. Is l<=0 , if true then go to step 53.
50. l = l - 1
51. Repeat step 11 to 50.
52. Convert the byte file into alphanumeric file
53. Compare this file with the original and with the file, which would have been received if sent uncoded to find the probability of error.
54. Stop.

Two function used in the above algorithms are given below

- The first function is max() which accepts two float values and returns back the log(1+exp(|a-b|)) which is a float value.
- The second function is minimum(), which accepts starting address of continuous 8 values and returns back the minimum out of these values.

CHAPTER 7

RESULTS

7.1 A SIMPLIFIED TRELLIS-BASED DECODING

7.1.1 File, Which is Encoded and Sent

read.txt

Every Sunday, Jessica went to see her father in the city and came home on the 6:00 o'clock train. One day she told her driver, Jack, that she would be back an hour earlier and to pick her up at the station. Jack forgot and went to get her at the usual time. When Jessica arrived and did not find Jack there, she started walking home. Jack met her on the road and took her.

7.1.2 File After Adding Noise

WRITE8.TXT

Everù Sunday, Jewsica went"to!sgm hmò fathev mn the°kity ant caíe hïmå on the 6:±±!o'slock tvain.!One$da} óhå vold(yer driveò, Jáck, ôhat she ÷ uld jm rack cn houv earlier end ~o péck her wð qththe statkon. Jack forgo|$qnd went uo get jer at$the uwual timg.Whån Jessmca arrived ánd did îot¨find Jack uøere.(shm started waì{ing"home. Jack met her0on`the"rïad and took her.@

7.1.3 File Received After Applying Algorithm

WRITE4.TXT

Every Sunday, Jessica went to see her father in the city and came home on the 6:00 o'clock train. One day she told her driver, Jack, that she would be back an hour earlier and to pick her up at the station. Jack forgot and went to get her at the usual time.When Jessica arrived and did not find Jack there, she started walking home. Jack met her on the road and took her.

7.1.4 Trellis Based Code Efficiency

Uncoded

No of errors: 83/3000

Trellis Based Code

No of errors: 0/3000

7.2 TURBO DECODER WITH LOG-MAP BASED ITERATIVE DECODING

7.2.1 File, Which is Encoded and Sent:

read.txt

Every Sunday, Jessica went to see her father in the city and came home on the 6:00 o'clock train. One day she told her driver, Jack, that she would be back an hour earlier and to pick her up at the station. Jack forgot and went to get her at the usual time. When Jessica arrived and did not find Jack there, she started walking home. Jack met her on the road and took her.

7.2.2 File After Adding Noise

WRITE8.TXT

Every Sujdã]. Bessica`÷enR vm$see`hdr¢g`tler a^ uhe`citù anL came$hkme°   o~ tje   :00 o'cl ck trál~.   n% dA9°rhe uold her driver, Jaãk, txat`sh-0 'M5lf `u bacc€!l lour!eazl)er and to!pick her up a4(thu stati k& JasK f/FgoT aÊa went$to og< je0"at thq esucì téem. hEn ÚessIca arrköEf an` oy`"nod &ind Nack tîere,(Shm starded wAlking h   me: Back met hev mo the roá$(and took èeb.

7.2.3 File Received After Applying Algorithm (iteration:1)

WRITE6.TXT

Every Sunday, Jessica went to see her father in the city and came home on the 6:00 o'clock train. One day she told her driver, Jack, that she would be back an hour earlier and to pick her up at the station. Jack forgot and went to get her at the usual time.When Jessica arrived and did not find Jack there, she started walking home. Jack met her on the road and took her.

7.2.4 File Received After Applying Algorithm (Iteration:2)

WRITE6.TXT

Every Sunday, Jessica went to see her father in the city and came home on the 6:00 o'clock train. One day she told her driver, Jack, that she would be back an hour earlier and to pick her up at the station. Jack forgot and went to get her at the usual time.When Jessica arrived and did not find Jack there, she started walking home. Jack met her on the road and took her.

7.2.5 Log-MAP-Based Code Efficiency

Uncoded

No of errors: 161/3000

Log-MAP Decoding

No of errors:

Iteration 1: 0/3000

Iteration 2: 0/3000

7.3 TABULAR RESULTS

7.3.1 Simplified Trellis Based Decoder

Results for 3000 bits:

E_b/N_0	Coefficients for Generating AWGN	Error (Uncoded)	Error(Coded)
-2	14, 13	308	302
0	13,12	226	92
2	12, 12	187	83
4	12,11	137	17
6	11, 11	83	0

7.3.2 Log-MAP Turbo Decoder

(a) Results for 3000 bits

E_b/N_0	Coefficients for Generating AWGN	Error (Uncoded)	Error(Coded) Iteration 1	Error(Coded) Iteration 2
-6	16, 15, 15	487	53	38
-4	15, 14, 14	383	31	15
-2	15, 14, 11	307	25	9
0	14, 13, 10	221	5	5
2	13,12, 9	161	0	0
4	12, 12, 9	112.5	0	0
6	12, 10, 9	79	0	0
8	11, 10, 9	31	0	0

(b) Results for 4000 bits

E_b/N_0	Coefficients for Generating AWGN	Error (Uncoded)	Error(Coded) Iteration 1	Error(Coded) Iteration 2
-6	16, 15, 15	641	66	46
-4	15, 14, 14	502	37	17
-2	15, 14, 11	405	28	9
0	14, 13, 10	303	4	4
2	13,12, 9	216	0	0
4	12, 12, 9	151	0	0
6	12, 10, 9	89	0	0
8	11, 10, 9	43	0	0

(c) Results for 5000 bits

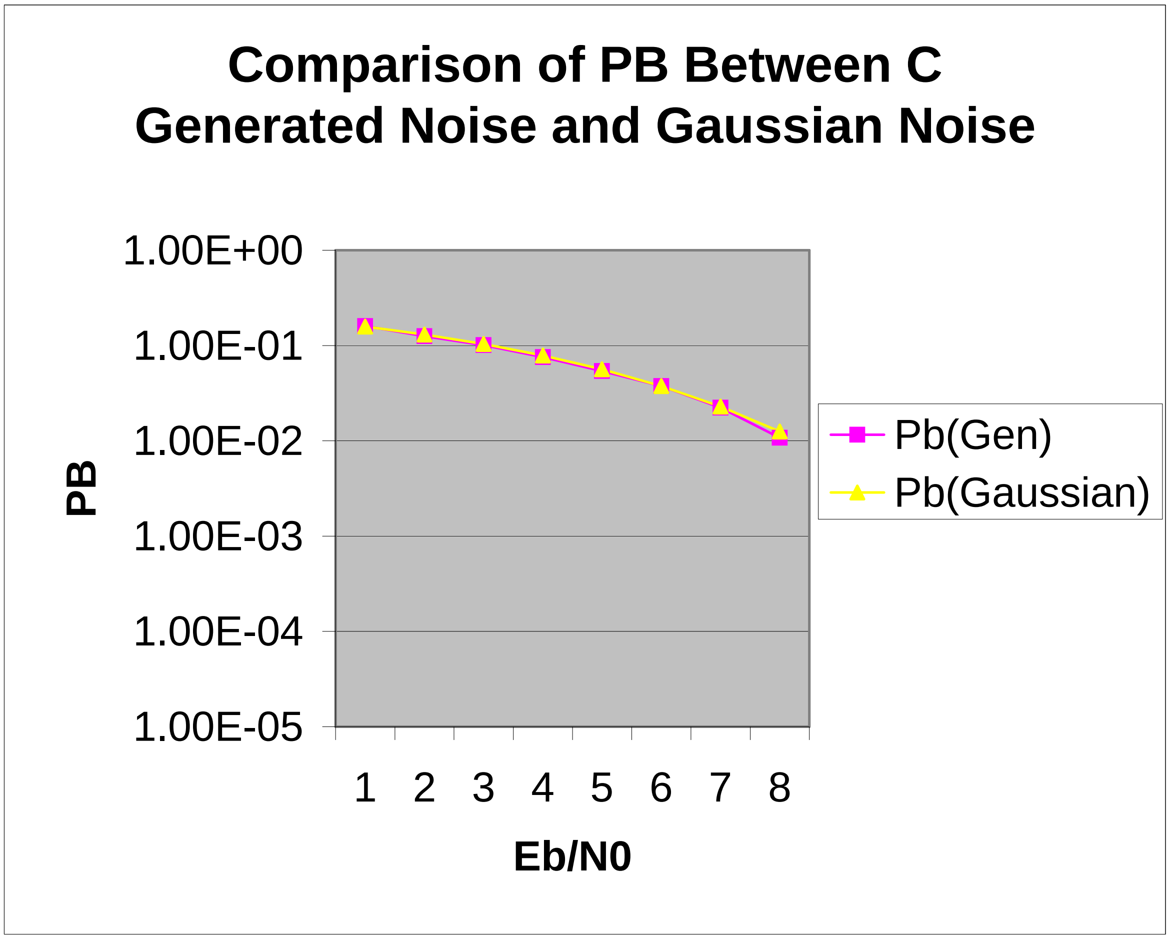
E_b/N_0	Coefficients for Generating AWGN	Error (Uncoded)	Error(Coded) Iteration 1	Error(Coded) Iteration 2
-6	16, 15, 15	786	91	72
-4	15, 14, 14	651	33	31
-2	15, 14, 11	494	29	22
0	14, 13, 10	346	1	1
2	13,12, 9	257	0	0
4	12, 12, 9	207	0	0
6	12, 10, 9	107	0	0
8	11, 10, 9	36	0	0

7.4 GRAPHICAL RESULTS

7.4.1 Comparison of Bit Error Rate Between Gaussian Noise and Noise Generated Using C Language (FOR 4000 BITS)

Eb/N0	Pb(C Generated)	Pb(Gaussian)
-6	0.16025	0.1584
-4	0.1255	0.1306
-2	0.10125	0.1038
0	0.07575	0.0786
2	0.054	0.0563
4	0.0375	0.0375
6	0.02225	0.0229
8	0.01075	0.0125

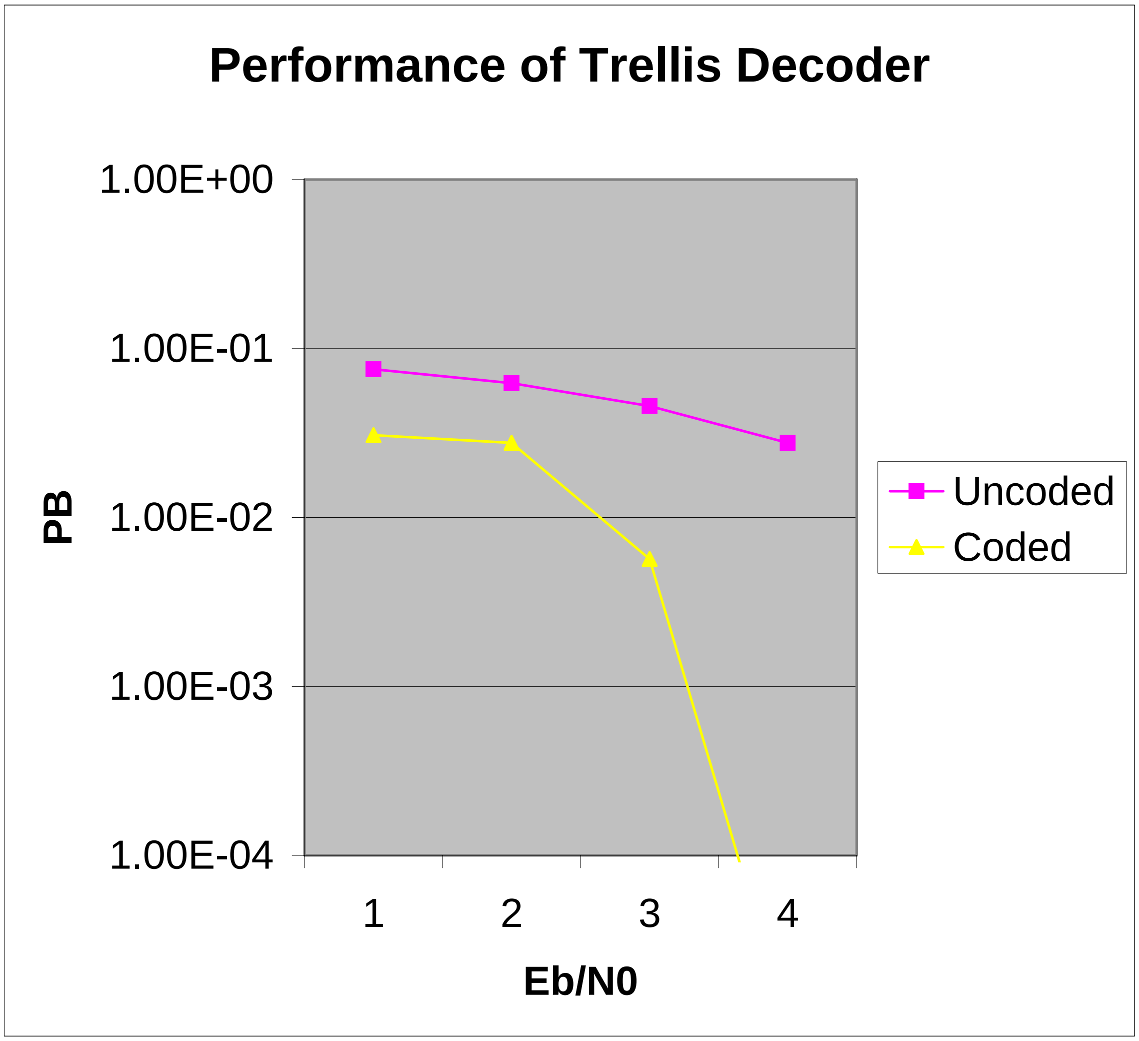
GRAPH 1



7.4.2 Performance of Simplified Trellis Based Decoder

Eb/N0	Uncoded	Coded
0	0.075333	0.030667
2	0.062333	0.027667
4	0.045667	0.005667
6	0.027667	0.00001

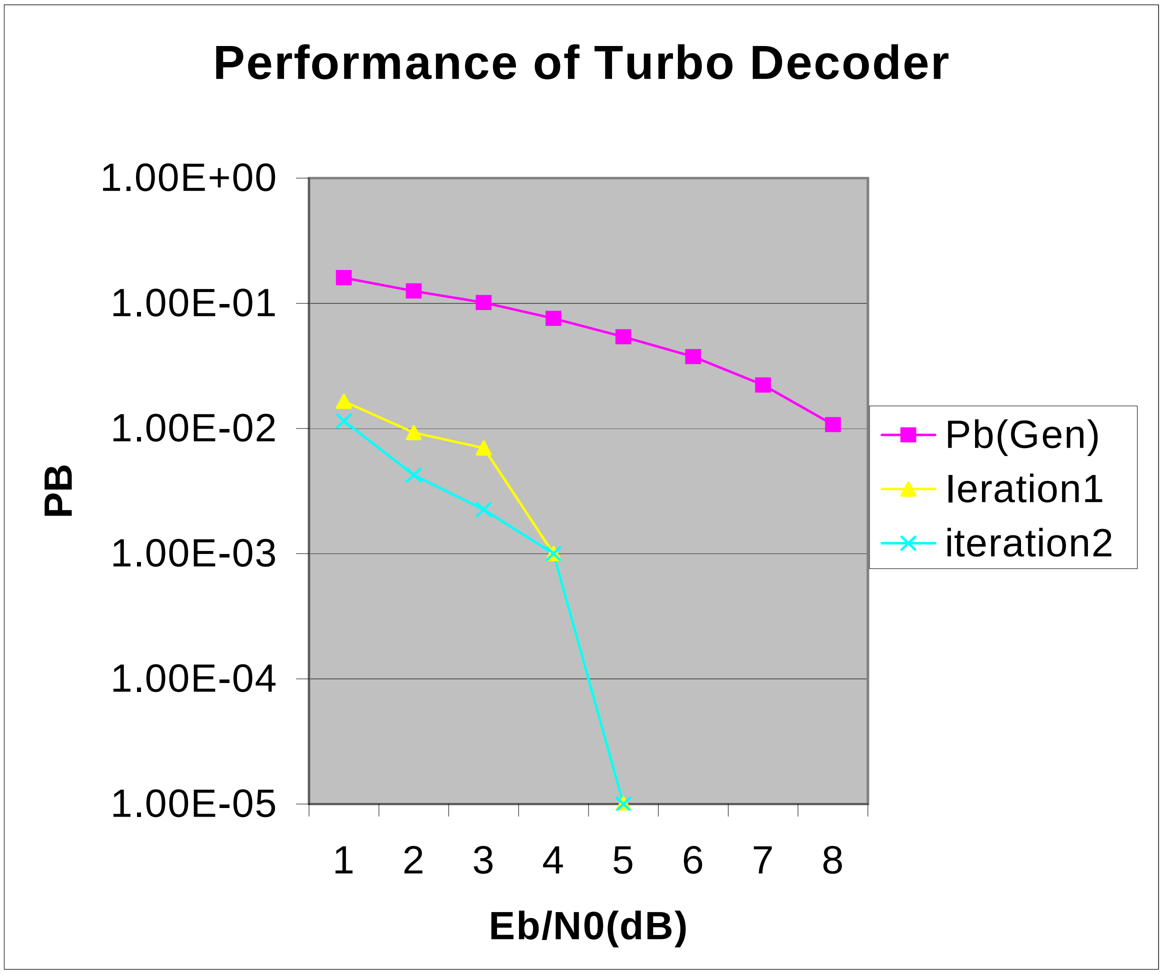
GRAPH 2



7.4.3 Performance of Log-MAP Turbo Decoder

Eb/N0	Pb(Gen)	Ieration1	iteration2
-6	0.16025	0.0165	0.0115
-4	0.1255	0.00925	0.00425
-2	0.10125	0.007	0.00225
0	0.07575	0.001	0.001
2	0.054	0.00001	0.00001
4	0.0375	0.00001	0.00001
6	0.02225	0.00001	0.00001
8	0.01075	0.00001	0.00001

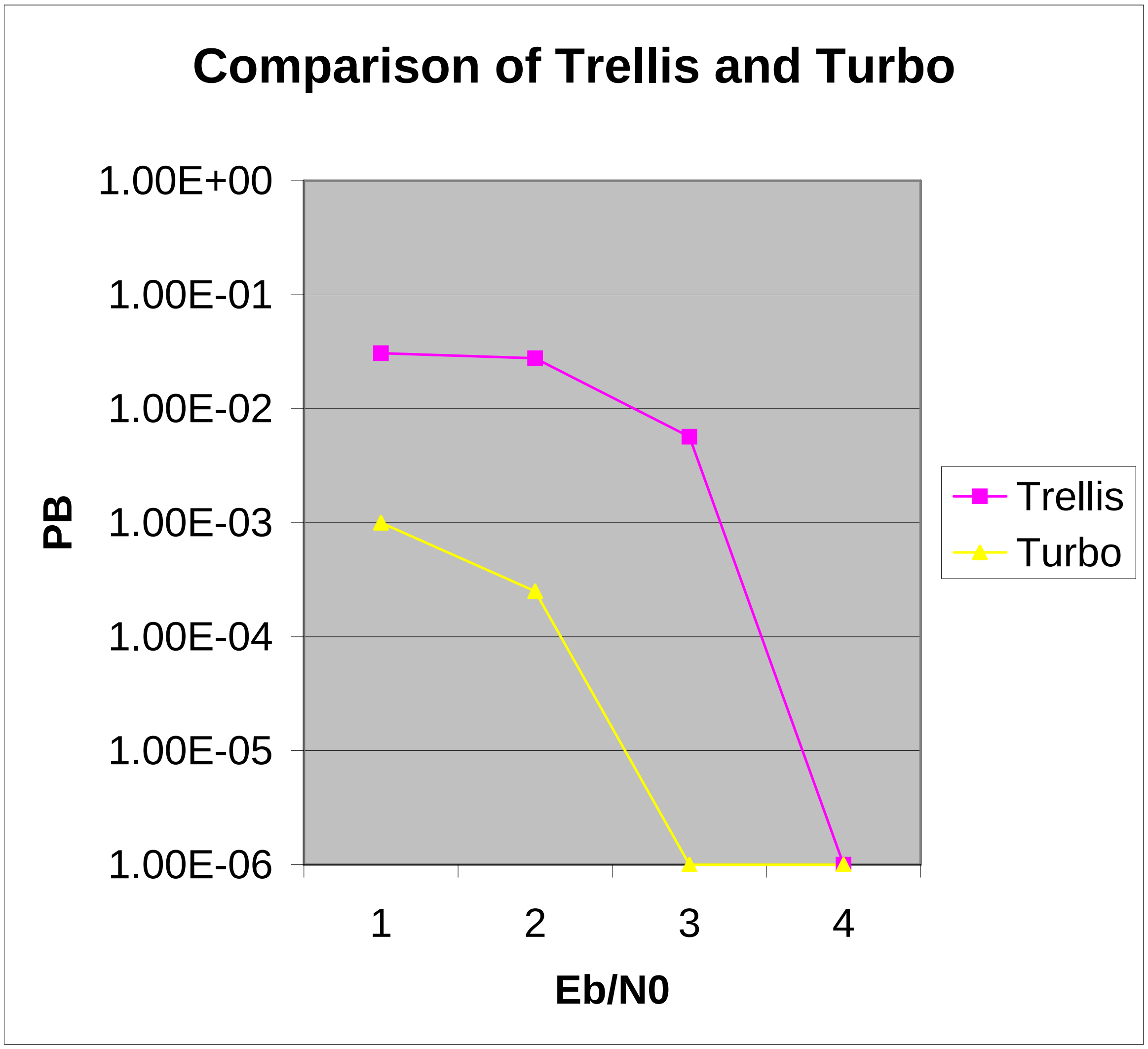
GRAPH 3



7.4.4 Comparison of Trellis and Turbo Code Performance

Eb/N0	Trellis	Turbo
0	0.030667	0.001
2	0.027667	0.00025
4	0.005667	0.000001
6	0.000001	0.000001

GRAPH 4



CONCLUSIONS

In this thesis two advanced and latest channel decoding algorithms are first developed and then implemented in language C. These two algorithms are modified version of some standard algorithms. The advantage of implementing these algorithms is that we are able to save some sort of resources/efforts as compared to standard ones and the efficiency is still approximately the same as that of the standard ones.

For implementing these decoding algorithms the C code is written for the channel encoder, AWGN channel and the channel decoder for both of the algorithms. The efforts are being made to generate standard AWGN noise but due to limitation of C language we are able to generate noise, which is very much similar, as shown in Graph 1. These C programs are then executed for both algorithms in a particular sequence (Encoder, Channel, Decoder) and results are obtained in form of text files. Again a C program is executed for comparing these text files and now numerical results are obtained which are put in form of a table manually. The tabular results are then used for drawing the graph.

The graph for the first algorithm “A Simplified Trellis-Based Decoder” is found to be in close approximation to the graph shown in [5]. But this algorithm is having a disadvantage that it cannot handle very large file. It can work with text file of maximum size 3500 bits only as it requires huge amount of memory, although speed of this algorithm is better than the second algorithm.

The graph for the second algorithm “Log-MAP-Based Iterative Turbo Decoder” is also found to be in close approximation to the standard Log-MAP algorithm [4], but with the limitation that there is no performance improvement after iteration 2, but in case of standard algorithm error P_B reduces up to 4-5 iterations, although with little performance improvement. This might be due to the effect of non-standard AWGN noise generated using C.

The error performance of Log-MAP-Based Turbo Decoder is found to be better than the A simplified trellis based decoder but the speed of later is better than the former.

FUTURE WORK AND RECOMMENDATIONS

The two channel decoding algorithm “A Simplified Trellis-Based Decoder” and “Log-MAP-Based Iterative Turbo Decoder” has been developed and implemented in this thesis. The results for both are in close approximation to the corresponding standard algorithm, but with certain limitations. The limitation in first algorithm is that it requires huge amount of memory but time taken is less, so can’t handle large files. The limitation for second algorithm is that the time taken in decoding rises exponentially with the size of the file, although memory requirement is less.

The possible future work, which may be carried out, related to this thesis work could be as follows:

1. In “A Simplified Trellis-Based Decoder” the memory requirement is very large. But the time required for executing the program is small. If possible some algorithm could be developed as a trade-off between memory and time requirement, such that memory requirement and time taken both are optimum.
2. In “Log-MAP-Based Iterative Turbo Decoder”, the memory required is small enough that it may handle large file also, but time taken rises exponentially with the size of file. It is due to this time requirement that we are not able to process a large file. The future work in this regard could be to reduce the time taken in executing the algorithm.
3. Out of these two algorithms the second one is having very good error performance as compared to first one. Given the outstanding performance of Turbo Code, the challenge could be to implement it into various communication systems at affordable decoding complexity, using current Very Large Scale Integration (VLSI) technologies.

REFERENCES

1. *VHDL Implementation of a Turbo Decoder With Log-MAP-Based Iterative Decoding*; Yanhui Tong, Tet-Hin Yeap, Member, IEEE, and Jean-Yves Chouinard, Senior Member, IEEE; IEEE Transactions on Instrumentation and Measurement, VOL 53, No. 4, AUGUST 2004
2. *Interleaver Structures for Turbo Codes with Reduced Storage Memory Requirement;* Johan Hokfelt, Ove Edfors and Torleiv Maseng, Department of Applied Electronics, Lund University, Lund, Sweden
3. *A Simplified Computational Kernel for Trellis-Based Decoding;* Matthias Kamuf, Student Member, IEEE, John B. Anderson, Fellow, IEEE, and Viktor Owall, Member, IEEE; IEEE Communication Letters, VOL. 8, No. 3, March 2004.
4. *Implementation of High Rate Turbo Decoders for Third Generation Mobile Communication*; Dr. Jason P Woodard.
5. *Digital Communications Fundamentals and Applications (Second Edition)* by Bernard Sklar; Communication Engineering Services, Tarzana, California and University of California, Los Angeles.
6. *VLSI Architecture for the MAP Algorithm;* Emmanuel Boutillon, Warren J. Gross, Student Member, IEEE, and P. Glenn Gulak, Senior Member, IEEE; IEEE Transactions on Communications VOL. 51, No. 2, February 2003.
7. *High Performance, High Throughput Turbo/SOVA Decoder Design*; Zhongfeng Wang, Member, IEEE, and Keshab K. Parhi, Fellow, IEEE; IEEE Transactions on Communications, VOL. 51, No. 4, April 2003
8. *An Efficient and Practical Architecture for High Speed Turbo Decoders*; Aliazam Abbasfar and Kung Yao Dept. of Electrical Engineering University of California, Los Angeles, USA

9. *FPGA Implementation of Parallel Turbo decoders;* Michael J. Thul, Norbert When, University of Kaiserslautern Erwin-Schrodinger-StraBe, 67663 Kaiserslautern, Germany {Thul, Wehn} @ eit.uni-kl.de

10. *Near Shannon limit error correcting coding and decoding: Turbo-Codes;* C. Berrou, A. Glavieux, and P. Thitimajshima, in Proc. ICC’93, pp. 1064-1070.

11. *A Comparison of optimal and sub-optimal MAP decoding algorithm operating in log domain;* P. Robertson, E. Villebrun, and P. Hoeher, in Proc. ICC’95, pp. 1009-1013.

12. *Optimal decoding of linear codes for minimizing symbol error rate*; R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, IEEE Trans. Inform. Theory, VOL. IT-13, pp. 284-287, Nar. 1974.

13. *A Viterbi algorithm with soft-decision outputs and its applications;* J. Hagenauer and P. Hoeher, in Proc. IEEE Globecom Conf., Dallas, TX, Nov. 1989, pp. 1680-1686

14. *A low complexity soft-output viterbi decoder architecture*; C. Berrou, P. Adde, E. Angui, and S. Faudeil, in Proc. IEEE ICC 1993, Geneva, Switzerland, May 1993, pp. 737-740.

15. *Design of fixed-point iterative decoders for concatenated codes with interleavers*; G. Montorsi and S. Benedetto, “,” IEEE J. Select Areas Communication, VOL. 49, Nov. 2001.

16. *VLSI architectures for the forward backward algorithm*; B. Emmanuel, J. G. Warren, and G. Glenn.

17. *An intuitive justification and simplified implementation of MAP decoder for convolutional codes;* A.J Viterbi, IEEE J.Select. Areas Communication, VOL. 16, pp. 260-264, Feb 1998

18. *Design of parallel concatenated convolutional codes*; S.Benedetto and G. Montorsi, IEEE Transactions on Communications, Vol. 44, pp. 591-600, May 1996.

19. *Effective free distance of Turbo codes*; D. Divsalar and R.J. McEliece, “,” Electronic Letters, Vol. 32, Feb 1996.

20. *On the design of prunable interleavers for turbo codes*; M. Eroz and A.R. Hammons, in Vehicular Technology Conference, Houston, USA, May 1999.
21. *Terminating the treillis of turbo-codes in the same state*; A. S. Barbulescu and S.S Pietrobon, Electronics Letters, Vol. 31, pp. 22-23, January 1995.
22. *Implementing the Viterbi algorithm*; H-L. Lou, IEEE Signal Processing Mag., Vol. 12, no. 5, pp. 42-52, sept 1995.
23. *Best Short Rate ½ Tailbiting Codes for the Bit-Error rate Criterion*; John B. Anderson, Fellow, IEEE; IEEE Transaction on Communications Vol. 48, No. 4, April 2000.
24. *Concatenated Decoding With a Reduced-Search BCJR Algorithm*; Volker Franz and John B. Anderson, Fellow, IEEE; IEEE Journal on Selected Areas in Communications Vol. 16, No. 2, February 1998.
25. *A Parallel Decoding Scheme For Turbo Codes*; Jah-Ming HSU and chin-Liang Wang, Department of Electrical Enginering, National Tsing Hua university, Hsinchu, Taiwan 30043, Republic of China.
26. *Iterative Decoding of Binary Block and Convolutional Codes*; Joachim Hagenauer, Fellow, IEEE, Elke Offer, and Lutz Papke; IEEE Transaction on Information Theory, Vol. 42, No. 2, March 1996.
27. *Real-Time Algorithms and VLSI Architectures for Soft Output MAP Convolutional Decoding;* Herbert Dawid and Heinrich Meyr, Aachen Univ. of Technology (RWTH), Integrated Systems for Signal Processing ISS-611810, Templergraben 55, D-52056 Aachen, Germany.
28. *A parallel MAP Algorithm for Low Latency Turbo Decoding*; Seokhyun Yoon, Student Member, IEEE and Yeheskel Bar-Ness, Fellow IEEE; IEEE Communications Letters, Vol. 6, No. 7, July 2002.
29. *A Real-Time Embedded Software Implementation of a Turbo Encoder and Soft Output Viterbi Algorithm Based Turbo Decoder; M. Farooq Sabir, Rashmi Tripathi, Brian L. Evans and Alan C. Bovik* Dept. *of* Electrical and Comp. Eng., The University of Texas at Austin, Austin, TX, 78712-1084 USA {mfsabir, rashmi, bevans, bovik} @ece.utexas.edu.

30. *Combined Turbo Codes and Interleaver Design*; Jinhong Yuan, Branka Vucetic, and Wen Feng; IEEE Transactions on Communications, Vol. 47, No. 4, April 1999.
31. *A Memory Efficient Scheme for Hardware Implementation of Log-Map Turbo Decoder*; Chunlong Bai Jun Jiang Ping Zhang, Wireless Tech Innovation Lab, Beijing University of Posts and Telecommunications, P.O. Box 92, BUPT, Beijing, 100876, P.R. China.
32. *VLSI Design of Dual-Mode VITERBI/TURBO decoder for 3GPP*; Kai Huang, Fan-Min Li, Pei-Ling Shen and An-Yeu Wu, Graduate Institute of Electronics Engineering, and Department of Electrical Engineering, National Taiwan University, Taipei, 106, Taiwan, R.O.C.

HOW TO EXECUTE C PROGRAMS?

Simplified Trellis Based Decoder

1. First execute the C program ‘Trellis1.c’. This program first converts the text (c:\read.txt) into equivalent ACSII Values (c:\write.txt) and then encodes this bit values using ½ convolutional encoder (write1.txt).
2. Execute the C program ‘Channel2.c’. This program adds random noise (Noise coefficients are entered by user to add varying amount of noise) to the encoded file and then apply threshold to get bit values again (c:\write2.txt).
3. Execute the C program ‘Trmod1.c’. This program takes noise-corrupted file (c:\write2.txt) as input, and produces the output byte file (c:\write3.txt) and output text file (c:\write4.txt).
4. Execute the C program ‘Effi.c’. This program compares the original file (c:\write.txt) and decoded file (c:\write3.txt) and displays the no of bit errors and total bits present.

5. Execute the C program ‘Uncoded.c’. This program adds random noise to the original file (c:\write.txt) to get the noise-corrupted file (c:\write7.txt).
6. Again execute the C program ‘Effi.c’, and compare the original file (c:\write.txt) and the noise-corrupted file (c:\write7.txt). It displays no of bit errors, while total no of bits remains the same as in step 4.
7. Now compare the efficiencies calculated in steps (4) and (6).

LOG-MAP Based Turbo Decoder

1. First execute the C program ‘Turboen1.c’. This program first converts the text (c:\read.txt) into equivalent ACSII Values (c:\write.txt) and then encodes this bit values using 1/3 convolutional encoder (write1.txt), and then rearranges them in proper order (c:\write2.txt).

2. Execute the C program ‘Channel1.c’. This program adds random noise (Noise coefficients are entered by user to add varying amount of noise) to the encoded file to get a file containing floating values (c:\write3.txt).
3. Execute the C program ‘Turbode4.c’ with iteration =1 (big loop). This program takes noise-corrupted file (c:\write3.txt) as input, and produces the output byte file (c:\write4.txt) and rearranged byte file (c:\write5.txt) and output text file (c:\write6.txt).
4. Execute the C program ‘Effi.c’. This program compares the original file (c:\write.txt) and decoded file (c:\write5.txt) and displays the no of bit errors and total bits present.
5. Execute the C program ‘Uncodest.c’. This program adds random noise to the original file (c:\write.txt) to get the noise-corrupted file (c:\write7.txt).
6. Again execute the C program ‘Effi.c’, and compare the original file (c:\write.txt) and the noise-corrupted file (c:\write7.txt). It displays no of bit errors, while total no of bits remains the same as in step 4.

7. Now compare the efficiencies calculated in steps (4) and (6).
8. Now change the no of iteration from 1 to 2 (big loop) in C program ‘Turbode4.c’ and repeat step 4 to 7.

Coding

File-1

```
// Program to add AWGN Noise to the bit stream received from the Turbo
encoder
# include<stdio.h>
# include<conio.h>
void main()
{
int i,j,k,c,c1,c2,rndi,nt=11,nt1=11,nt2=11;
double rndf,ch,ch1,ch2;
unsigned long n=0;
FILE *ifp,*ofp;
//nt,nt1,nt2: variables used for generating AWGN Noise
//*ifp: pointer to input file
//*ofp: pointer to output file
//i,j,k,c,c1,c2,n,ch,ch1,ch2: variables to store temporary values
//rndi: random generated integer value
//rndf: random generated float value
clrscr();
printf("\n Enter noise coffs for Generating Noise");
scanf("%d%d%d",&nt,&nt1,&nt2);
if((ifp=fopen("C:\\write2.txt","r"))==NULL)
printf("\n ERROR:Input file could not be opened");
if((ofp=fopen("C:\\write3.txt","w"))==NULL)
```

```
printf("\n ERROR:Output file could not be opened");
//Counting no of inputs bits
while((c=getc(ifp))!=EOF)
n++;
printf("\n count=%ld",n);
//Logic for adding noise
for(i=0;i<n;i+=3)
{
fseek(ifp,i,0);
if((c=getc(ifp))!=EOF)
{
c=c-'0';
printf("%d",c);
if(c==0)
c=-1;
rndi=(rand()%nt);
rndf=(float)rndi/10;
if(rndi%2==0)
ch=c+rndf;
else
ch=c-rndf;
}
if((c1=getc(ifp))!=EOF)
{
c1=c1-'0';
printf("%d",c1);
if(c1==0)
c1=-1;
rndi=(rand()%nt1);
rndf=(float)rndi/10;
if(rndi%2==0)
ch1=c1+rndf;
else
ch1=c1-rndf;
}
if((c2=getc(ifp))!=EOF)
{
c2=c2-'0';
printf("%d",c2);
if(c2==0)
c2=-1;
rndi=(rand()%nt2);
rndf=(float)rndi/10;
if(rndi%2==0)
ch2=c2+rndf;
else
ch2=c2-rndf;
}
fprintf(ofp,"%10.6f%10.6f%10.6f",ch,ch1,ch2);
}
fclose(ifp);
fclose(ofp);
}
```

File-2

```
// Program to add AWGN Noise to the bit stream received from the trellis
encoder
# include<stdio.h>
# include<conio.h>
void main()
{
int i,j,k,c,c1,rndi,nt=11,nt1=11;
double rndf,ch,ch1;
unsigned long n=0;
FILE *ifp,*ofp;
//nt,nt1: variables used for generating AWGN Noise
//*ifp: pointer to input file
//*ofp: pointer to output file
//i,j,k,c,c1,n,ch,ch1: variables to store temporary values
//rndi: random generated integer value
//rndf: random generated float value
clrscr();
if((ifp=fopen("C:\\write1.txt","r"))==NULL)
printf("\n ERROR:Input file could not be opened");
if((ofp=fopen("C:\\write2.txt","w"))==NULL)
printf("\n ERROR:Output file could not be opened");
printf("\n Enter noise coffs for Generating Noise");
scanf("%d%d",&nt,&nt1);
//Counting no of inputs bits
while((c=getc(ifp))!=EOF)
n++;
printf("\n count=%ld",n);
//Logic for adding noise
for(i=0;i<n;i+=2)
```

```
{
fseek(ifp,i,0);
if((c=getc(ifp))!=EOF)
{
c=c-'0';
printf("%d",c);
if(c==0)
c=-1;
rndi=(rand()%nt);
rndf=(float)rndi/10;
if(rndi%2==0)
ch=c+rndf;
else
ch=c-rndf;
if(ch>0.0)
putc(1+'0',ofp);
else
putc(0+'0',ofp);
}
if((c1=getc(ifp))!=EOF)
{
c1=c1-'0';
printf("%d",c1);
if(c1==0)
c1=-1;
rndi=(rand()%nt1);
rndf=(float)rndi/10;
if(rndi%2==0)
ch1=c1+rndf;
else
ch1=c1-rndf;
if(ch1>0.0)
putc(1+'0',ofp);
else
putc(0+'0',ofp);
}
}
fclose(ifp);
fclose(ofp);
}
```

File-3

```
/*Program used for calculating efficiency by comparing bits in original file
and decoded bit stream*/
# include<stdio.h>
# include<conio.h>
void main()
{
int i,j,c,c1,disagree=0,n=0;
FILE *fp1,*fp2;
//i,j,c,c1,n:variables used for storing temporary values
// disagree: variables used for storing no of bits in disagreement
//*fp1: pointer to original file
//*fp2: pointer to decoded file(after passing through channel)
if((fp1=fopen("C:\\write.txt","r"))==NULL)
printf("\n ERROR: Ist Input file could not be opened");
if((fp2=fopen("C:\\write5.txt","r"))==NULL)
printf("\nERROR:2nd Input file could not be opened");
while((c=getc(fp1))!=EOF)
n++;
printf("\n count=%d",n);
fseek(fp1,0,0);
fseek(fp2,0,0);
printf("\n");
//Logic for comapring bit stream in two file bit by bit
for(i=0;i<n;i++)
{
if((c=getc(fp1))!=EOF)
printf(" %d",c-'0');
if((c1=getc(fp2))!=EOF)
printf(" %d",c1-'0');
printf("\n");
if(c1!=c)
disagree++;
}
//displaying no of bits corrupted which could not be cottected by algorithm
printf("\n disagree=%d",disagree);
//displaying total no of bits  in decoded/original file
printf("\n n=%d",n);
fclose(fp2);
fclose(fp1);
}//end of main
```

File-4

```
/*Program to encode the bits using trellis diagram for constraint length K=3
  and two code generators with cofficients 111 and 101 */
# include<stdio.h>
# include<conio.h>
void main()
{
int i,j,k,c,ch,a,sym,state,n=0;
```

```
FILE *ifp,*ofp;
//state:for storing state at any time t
//output[]:for storing the output branch word for given state and input bits 0 and
1
//*ifp: pointer to input file
//*ofp: Pointer to output file
//i,j,c,ch,a,sym:variables to store temporary values
int nextstate[4][2]={0,2,
                     0,2,
                     1,3,
                     1,3};
int output[4][2]={0,3,
                  3,0,
                  1,2,
                  2,1};
if((ifp=fopen("C:\\read.txt","r"))==NULL)
printf("\nERROR:1st Input file could not be opened");
if((ofp=fopen("C:\\write.txt","w"))==NULL)
printf("\nERROR:1st Output file could not be opened");
//Converting alphabetical file into byte file
while((c=getc(ifp))!=EOF)
{
ch=c;
for(i=0;i<8;i++)
{
a=ch%2;
ch/=2;
putc(a+'0',ofp);
}
}
//Add 8 trailing bits
```

```
for(i=0;i<8;i++)
putc(0+'0',ofp);
fclose(ifp);
fclose(ofp);
if((ifp=fopen("C:\\write.txt","r"))==NULL)
printf("\n ERROR:2nd Input file could not be opened");
if((ofp=fopen("C:\\write1.txt","w"))==NULL)
printf("\n ERROR: 2nd Output file could not be opened");
//counting no of bits in byte file
n=0;
while((c=getc(ifp))!=EOF)
n++;
printf("\n count=%d",n);
//intial state is 00
state=0;
//simulation of trellis diagram for calculating encoded bits
fseek(ifp,0,0);
for(i=0;i<n;i++)
{
if((sym=getc(ifp))!=EOF)
{
sym=sym-'0';
switch(state)
{
case 0:if(sym==0)
    {
    putc(0+'0',ofp);
    putc(0+'0',ofp);
    state=nextstate[0][0];
    }
    else
```

```
       {
       putc(1+'0',ofp);
       putc(1+'0',ofp);
       state=nextstate[0][1];
       }
       break;
case 1:if(sym==0)
      {
      putc(1+'0',ofp);
      putc(1+'0',ofp);
      state=nextstate[1][0];
      }
      else
      {
      putc(0+'0',ofp);
      putc(0+'0',ofp);
      state=nextstate[1][1];
      }
      break;
case 2: if(sym==0)
      {
      putc(0+'0',ofp);
      putc(1+'0',ofp);
      state=nextstate[2][0];
      }
      else
      {
      putc(1+'0',ofp);
      putc(0+'0',ofp);
      state=nextstate[2][1];
      }
```

```
     break;
case 3: if(sym==0)
     {
     putc(1+'0',ofp);
     putc(0+'0',ofp);
     state=nextstate[3][0];
     }
     else
     {
     putc(0+'0',ofp);
     putc(1+'0',ofp);
     state=nextstate[3][1];
     }
     break;
     }
    }
}
}
```

File-5

```
/*Program to decode encoded bits using A Simplified Computational Kernel for
  Trellis Based Decoding(modified viterbi algorithm)for constraint length K=3
  and code rate=1/2 with two code generators with coefficients 111 101*/
# include<stdio.h>
# include<conio.h>
int output[4][2]={0,3,
                  3,0,
                  1,2,
                  2,1};
```

```
/*output matrix contains the output branch word corresponding to four
  states (row wise) and input bits 1 and 0 (column wise)*/
void main()
{
int hammingdist(int,int);
//This function is used for calculating hamming distance between two arguments
passed to it
int
n=0,c,c1,c2,bm1,bm2,modbm1,modbm2,i,j,k,sym,min,count,index=0,**st_metric
,*numoct,*decod;
FILE *ifp,*ofp,*fp;
//c,c1,c2: variables used for storing received bit values
//bm1,bm2:variables for storing branch metric values
//modbm1,modbm2:variables for storing modified branch metric values
//min:for storing minimum value of state metric
//index:for storing the index value of minimum state metric
//**st_metric:pointer used for storing the state metric at each instant of time
//*numoct:pointer used for storing octal value of binary bits received(pair wise)
//*decod:pointer used for storing decoded bits
//n:to store no of received symbols(bits)
//Sym:variable to store octal value
//i,j,k:local variables used for executing for loops
//*ifp: pointer to input file (contains bits received form channel)
//*ofp: Pointer to output file(contains decoded bits)
clrscr();
fp=fopen("C:\\decoder.txt","w");
if((ifp=fopen("C:\\write2.txt","r"))==NULL)
printf("\nERROR:1st Input file could not be opened");
if((ofp=fopen("C:\\write3.txt","w"))==NULL)
printf("\nERROR:1st Output file could not be opened");
while((c=getc(ifp))!=EOF)
```

```
n++;
//Dynamic memory allocation for storing octal equivalent values of received symbols (bits)
numoct=(int*)calloc(n/2,sizeof(int));
//Dynamic memory allocation for storing decoded bit values
decod=(int*)calloc(n/2+1,sizeof(int));
//Dynamic memory allocation for storing state metric values
st_metric=(int**)calloc(n/2+1,sizeof(int));
for(i=0;i<n/2+1;i++)
st_metric[i]=(int*)calloc(4,sizeof(int));
//Logic for calculating octal equivalent of received bits(taking two at a time
fseek(ifp,0,0);
for(i=0;i<n;i+=2)
{
if((c=getc(ifp))!=EOF)
c=c-'0';
if((c1=getc(ifp))!=EOF)
c1=c1-'0';
numoct[i/2]=2*c+c1;
}
//Equating state metric at time t=0 and t=1 to zero
for(j=0;j<2;j++)
for(i=0;i<4;i++)
st_metric[j][i]=0;
//Calculating state metric at time t=2
bm1=hammingdist(numoct[0],output[0][0]);
st_metric[1][0]=st_metric[0][0]+bm1;
bm2=hammingdist(numoct[0],output[0][1]);
st_metric[1][2]=st_metric[0][0]+bm2;
//Calculating state metric at time t=3
bm1=hammingdist(numoct[1],output[0][0]);
```

```
bm2=hammingdist(numoct[1],output[0][1]);
st_metric[2][0]=st_metric[1][0]+bm1;
st_metric[2][2]=st_metric[1][0]+bm2;
bm1=hammingdist(numoct[1],output[2][0]);
bm2=hammingdist(numoct[1],output[2][1]);
st_metric[2][1]=st_metric[1][2]+bm1;
st_metric[2][3]=st_metric[1][2]+bm2;
/*Calculating state metric for time t>=4 using previous state metrics and branch
  metric values i.e Implementation of MODIFIED ADD COMPARE SELECT
COMPUTATION Algorithm*/
for(i=2;i<n/2;i++)
{
sym=numoct[i];
bm1=hammingdist(sym,output[0][0]);
modbm1=2*(1-bm1);
bm2=hammingdist(sym,output[2][0]);
modbm2=2*(1-bm2);
if(st_metric[i][0]<=st_metric[i][1]+modbm1)
st_metric[i+1][0]=st_metric[i][0];
else
st_metric[i+1][0]=st_metric[i][1]+modbm1;

if(st_metric[i][1]<=st_metric[i][0]+modbm1)
st_metric[i+1][2]=st_metric[i][1];
else
st_metric[i+1][2]=st_metric[i][0]+modbm1;



if(st_metric[i][2]<=st_metric[i][3]+modbm2)
st_metric[i+1][1]=st_metric[i][2];
else
st_metric[i+1][1]=st_metric[i][3]+modbm2;
st_metric[i+1][1]=st_metric[i+1][1]+bm2-bm1;

if(st_metric[i][3]<=st_metric[i][2]+modbm2)
st_metric[i+1][3]=st_metric[i][3];
else
st_metric[i+1][3]=st_metric[i][2]+modbm2;
st_metric[i+1][3]=st_metric[i+1][3]+bm2-bm1;
}
//Displaying the State Metric Values
for(i=n/2;i>=0;i--)
{
for(j=0;j<4;j++)
{
printf("  %d", st_metric[i][j]);
fprintf(fp," %d",st_metric[i][j]);
}
fprintf(fp,"\n");
printf("\n");
}
//Calculating minimum value of state metric at time t=maximum time
i=n/2;
min=st_metric[i][0];
for(j=1;j<4;j++)
{
if(st_metric[i][j]<min)
{
min=st_metric[i][j];
index=j;
}
}
```

```
//logic for traversing back the path on trellis diagram from t=max to t=1 and
getting decoded bits
do
{
switch(index)
{
case 0:if(st_metric[i-1][0]<=st_metric[i-1][1])
    index=0;
    else
    index=1;
    decod[i]=0;
    i--;
    break;
case 1:if(st_metric[i-1][2]<=st_metric[i-1][3])
    index=2;
    else
    index=3;
    decod[i]=0;
    i--;
    break;
case 2:if(st_metric[i-1][0]<=st_metric[i-1][1])
    index=0;
    else
    index=1;
    decod[i]=1;
    i--;
    break;
case 3:if(st_metric[i-1][2]<=st_metric[i-1][3])
    index=2;
    else
    index=3;
```

```
        decod[i]=1;
        i--;
       break;
  }
}while(i>0);
//Displaying decoded bits and writing to output file
printf("\n The decoded bits are");
for(i=1;i<=n/2;i++)
{
printf("%d",decod[i]);
putc(decod[i]+'0',ofp);
}
fclose(ifp);
fclose(ofp);
//opening new output file to write alphabetical vales from binary values
if((ifp=fopen("C:\\write3.txt","r"))==NULL)
printf("\n ERROR:Input file could not be opened");
if((ofp=fopen("C:\\write4.txt","w"))==NULL)
printf("\n ERROR:Output file could not be opened");
n=0;
while((c=getc(ifp))!=EOF)
n++;
//Logic for Converting Binary values to alphabetical values
for(i=0;i<n/8;i++)
{
c1=0;
for(j=0;j<8;j++)
{
fseek(ifp,8*i+j,0);
c=getc(ifp);
c=c-'0';
```

```
c2=c;
for(k=0;k<j;k++)
c2*=c*2;
c1+=c2;
}
printf("%c",c1);
putc(c1,ofp);
}
fclose(ifp);
fclose(ofp);
}//end of main
int hammingdist(int x1,int x2)//function for calculating hamming distance
{
int x3=x1-x2;
if(x3<0)
x3=-x3;
if(x3==0)
return 0;
if(x3==1)
{
if(((x1==1)&&(x2==2))||((x1==2)&&(x2==1)))
return 2;
else
return 1;
}
if(x3==2)
return 1;
if(x3==3)
return 2;
}
```

File-6

```
//program to decode systematic convolutionally encoded stream with one code
// generator whose coefficients are entered by user using MAP Decoding
Algorithm
# include<stdio.h>
# include<conio.h>
#  include<math.h>
# define infin -1.0e100
double max(double,double);
//Program to find minimum value
double minimum(double f[][9], int i)
{
int j,flag=0;
double min=f[0][i];
for(j=0;j<8;j++)
if(f[j][i]<0.0)
break;
if(j==8)
flag=1;
else
flag=0;
if(flag==1)
{
for(j=1;j<8;j++)
if(f[j][i]<min)
min=f[j][i];
}
else
min=0.0;
```

```
return min;
}
void main()
{
  FILE *fp,*ifp,*ofp;
  int i,j,k,l,c,c1,c2,x,x1,big,rndi,decod[8],inleav[5];
//n: variable to store no of bits received
//rndi= variable to store integer random number
//decod:pointer used for storing decoded bit stream
//inleav[5]:array for storing interleaver structure
//output[][]: for storing the output values corresponding to different coefficients
of code generator g1
//i,j,k,c,c1,c2,x,x1,big: local variables used for calculation of other parameters
//*fp:pointer to a file used for storing intermediate values(optional)
//*ifp: pointer to input file
//*ofp: pointer to output file
 int LUTFS[2][8]={{0,2,5,7,1,3,4,6},{1,3,4,6,0,2,5,7}};
 int LUTBS[2][8]={{0,4,1,5,6,2,7,3},{4,0,5,1,2,6,3,7}};
//LUTFS[][]:look up table for simulating trellis structure for forward metrics
//LUTBS[][]:look up table for simulat9ing trellis structure for reverse metrics
 double
maxnum,maxden,num,den,num1,den1,rndf,fsmet[8][9],rsmet[8][9],bmet[4][8],lr
[8],in[8],exlr[8],apri[8],app[8];
 double
ch,ch1,ch2,yp[8],rstemp1[8],rstemp2[8],rstemp3[8],rstemp4[8],rstemp5[8],rstem
p6[8],rstemp7[8],rstemp8[8],min;
  unsigned long n=0;
 //maxnum,maxden,num,den,num1,den1: variables used for calculating other
parameters
 //rndf:v0ariable to store floating type random number
//fsmet[][]:array used for storing forward state metric values
```

```
//rsmet[][]:array used for storing reverse state metric values
//rstemp1[]-rstemp8[]:arrays used for storing and exchanging reverse state
metric values
//ch,ch1,ch2:variables used for calulating other parameters
//yp[8]:array used for storing interleaved parity sequence
//bmet[][]:array used for storing branch metric values
//num:variable to store numerator for likelihood ratio
//den:variable to store denominator for likelihood ratio
//lr[8]: array for storing LR values
//in[8]:array for storing interleaved data bit values
//exlr[8]:array for storing external values
//apri[8]:array for storing a priori values
//app[8]:
clrscr();
if((fp=fopen("C:\\decoder.txt","w"))==NULL)
printf("\n ERROR: Could not open the file");
if((ifp=fopen("C:\\write3.txt","r"))==NULL)
printf("\nERROR:1st Input file could not be opened");
if((ofp=fopen("C:\\write4.txt","w"))==NULL)
printf("\nERROR:1st Output file could not be opened");
//assiging values for simulating interleaver structure
inleav[0]=0;
inleav[1]=3;
inleav[2]=1;
inleav[3]=4;
inleav[4]=2;
//calculating number of float values (transmitted sequence+noise)
while((i=fscanf(ifp,"%lf %lf %lf",&ch,&ch1,&ch2))!=EOF)
n+=3;
printf("\n count=%d",n);
printf("\n");
```

```
fseek(ifp,0,0);
//intializing reverse state metric values for decoder1
for(k=1;k<8;k++)
rstemp2[k]=infin;
rstemp2[0]=0.0;
//intializing reverse state metric values for decoder2
for(k=1;k<8;k++)
rsmet[k][8]=infin;
rsmet[0][8]=0.0;
//l loop: used for scanning whole file
for(l=n/24;l>0;l--)
{
//intializing a priori values to zeros
for(k=0;k<8;k++)
apri[k]=0.0;
//big loop: used for number of iterations
for(big=0;big<2;big++)
{
fseek(ifp,0,0);
//intializing forward state metrics
for(k=1;k<8;k++)
fsmet[k][0]=infin;
fsmet[0][0]=0.0;
//i loop:used for calculating forward state metric values for each block
for(i=0;i<l;i++)
{
//j loop: used for accessing 24 bits at a time
for(j=0;j<24;j+=3)
{
if((k=fscanf(ifp,"%lf %lf %lf",&ch,&ch1,&ch2))!=EOF)
{
```

```
//fprintf(fp,"\n %lf %lf %lf",ch,ch1,ch2);
//printf("\n %f %f %f",ch,ch1,ch2);
//adding a priori information to branch metric values if reached to the block
which is being decoded
if(i==l-1)
{
bmet[0][j/3]=-ch-ch1-apri[j/3];
bmet[1][j/3]=-ch+ch1-apri[j/3];
bmet[2][j/3]=ch-ch1+apri[j/3];
bmet[3][j/3]=ch+ch1+apri[j/3];
}
//calulating branch metric values
else
{
bmet[0][j/3]=-ch-ch1;
bmet[1][j/3]=-ch+ch1;
bmet[2][j/3]=ch-ch1;
bmet[3][j/3]=ch+ch1;
}
}
}// end of j loop
/*fprintf(fp,"\n bmet");
//printf("\n bmet");
for(j=0;j<8;j++)
{
for(k=0;k<4;k++)
{
fprintf(fp,"  %f",bmet[k][j]);
//printf(" %f",bmet[k][j]);
}
//printf("\n");
```

```
fprintf(fp,"  \n");
}*/
//Logic for calculating forward state metric values
for(k=1;k<=8;k++)
{
for(j=0;j<8;j++)
{
if(j%2==0)
fsmet[j][k]=max(bmet[0][k-1]+fsmet[LUTFS[0][j]][k-1],bmet[3][k-
1]+fsmet[LUTFS[1][j]][k-1]);
else
fsmet[j][k]=max(bmet[1][k-1]+fsmet[LUTFS[0][j]][k-1],bmet[2][k-
1]+fsmet[LUTFS[1][j]][k-1]);
}
}
/*fprintf(fp,"\n forward st metric is");
for(k=0;k<=8;k++)
{
for(j=0;j<8;j++)
fprintf(fp," %f",fsmet[j][k]);
fprintf(fp,"  \n");
}*/
if(i==l-1)
break;
min=minimum(fsmet,k-1);
fprintf(fp,"min=%f",min);
//normalizing the forward state metric values
for(k=0;k<8;k++)
fsmet[k][0]=fsmet[k][8]-min;
}//end of i loop
//fprintf(fp,"For Comparing");
```

```
//for(k=0;k<8;k++)
//fprintf(fp," %f",fsmet[k][8]);
//saving reverse state metric values for different no of iterations
if((big==0)&&(l<n/24))
for(k=0;k<8;k++)
{
rsmet[k][8]=rstemp3[k];
rstemp2[k]=rstemp4[k];
}
if((big==1)&&(l<n/24))
for(k=0;k<8;k++)
{
rsmet[k][8]=rstemp5[k];
rstemp2[k]=rstemp6[k];
}
if((big==2)&&(l<n/24))
for(k=0;k<8;k++)
{
rsmet[k][8]=rstemp7[k];
rstemp2[k]=rstemp8[k];
}
//Logic for calculating reverse state metric values
for(k=7;k>=0;k--)
{
for(j=0;j<8;j++)
{
if((j==0)||(j==1)||(j==4)||(j==5))
rsmet[j][k]=max(bmet[0][k]+rsmet[LUTBS[0][j]][k+1],bmet[3][k]+rsmet[LUTB
S[1][j]][k+1]);
else
```

```
rsmet[j][k]=max(bmet[1][k]+rsmet[LUTBS[0][j]][k+1],bmet[2][k]+rsmet[LUTB
S[1][j]][k+1]);
}
}
/*fprintf(fp,"\n reverse st metric is");
for(k=8;k>=0;k--)
{
for(j=0;j<8;j++)
fprintf(fp,"  %f",rsmet[j][k]);
fprintf(fp,"  \n");
}

fprintf(fp,"For Comparing");
for(k=0;k<8;k++)
fprintf(fp," %f",rsmet[k][0]);
 */
//Calculating likelihood ratio for each decoded bit
for(k=0;k<8;k++)
{
num1=fsmet[0][k]+rsmet[4][k+1]+bmet[3][k];
den1=fsmet[0][k]+rsmet[0][k+1]+bmet[0][k];
for(j=1;j<8;j++)
{
if((j==2)||(j==3)||(j==6)||(j==7))
{
num=fsmet[j][k]+rsmet[LUTBS[1][j]][k+1]+bmet[2][k];
den=fsmet[j][k]+rsmet[LUTBS[0][j]][k+1]+bmet[1][k];
}
else
{
num=fsmet[j][k]+rsmet[LUTBS[1][j]][k+1]+bmet[3][k];
```

```
den=fsmet[j][k]+rsmet[LUTBS[0][j]][k+1]+bmet[0][k];
}
maxnum=max(num,num1);
maxden=max(den,den1);
//fprintf(fp,"\n
num1=%f,den1=%f,num=%f,den=%f,naxnum=%f,maxden=%f",num1,den1,nu
m,den,maxnum,maxden);
num1=maxnum;
den1=maxden;
}
app[k]=maxnum-maxden;
}
//calculating external lr ratio
for(k=0;k<8;k++)
exlr[k]=app[k]-apri[k];
//deinterleaving/interleaving
for(j=0;j<24;j+=3)
{
printf("\n x1=");
if(j%6==0)
{
x1=2*inleav[j/6+1];
printf("%d",x1);
}
else
{
for(k=1;k<=4;k++)
{
if((j/3+1)==2*inleav[k])
{
x1=2*k-1;
break;
}
}
printf("%d",x1);
}//else
apri[x1-1]=exlr[j/3];
}//end of j
for(k=0;k<8;k++)
rstemp1[k]=rsmet[k][8];
//Normalizing reverse state metric values for different number of iterations
if(big==0)
{
k=0;
min=minimum(rsmet,k);
fprintf(fp,"rsmin=%f",min);
for(k=0;k<8;k++)
rstemp3[k]=rsmet[k][0]-min;
}
if(big==1)
{
k=0;
min=minimum(rsmet,k);
fprintf(fp,"rsmin=%f",min);
for(k=0;k<8;k++)
rstemp5[k]=rsmet[k][0]-min;
}
if(big==2)
{
k=0;
min=minimum(rsmet,k);
fprintf(fp,"rsmin=%f",min);
```

```
for(k=0;k<8;k++)
rstemp7[k]=rsmet[k][0]-min;
}
/*decoder2:All logics are same as that of decoder 1 except the the input sequence
is
the interleaved version of sequence which is sent to decoder 1 and the output is
also
deinterleaved before sent to decoder1*/
fseek(ifp,0,0);
for(k=0;k<8;k++)
rsmet[k][8]=rstemp2[k];
for(k=1;k<8;k++)
fsmet[k][0]=infin;
fsmet[0][0]=0.0;
for(i=0;i<l;i++)
{
for(j=0;j<24;j+=3)
{
if((k=fscanf(ifp,"%lf %lf %lf",&ch,&ch1,&ch2))!=EOF)
{
//fprintf(fp,"\n %lf %lf %lf",ch,ch1,ch2);
//printf("\n %f %f %f",ch,ch1,ch2);
yp[j/3]=ch2;
if(j%6==0)
{
x1=2*inleav[j/6+1];
printf("%d",x1);
}
else
{
for(k=1;k<=4;k++)
{
if((j/3+1)==2*inleav[k])
{
x1=2*k-1;
break;
}
}
printf("%d",x1);
}//else
in[x1-1]=ch;
}
}// end of j loop

for(j=0;j<8;j++)
{
if(i!=l-1)
{
bmet[0][j]=-in[j]-yp[j];
bmet[1][j]=-in[j]+yp[j];
bmet[2][j]=in[j]-yp[j];
bmet[3][j]=in[j]+yp[j];
}
else
{
bmet[0][j]=-in[j]-apri[j]-yp[j];
bmet[1][j]=-in[j]-apri[j]+yp[j];
bmet[2][j]=in[j]+apri[j]-yp[j];
bmet[3][j]=in[j]+apri[j]+yp[j];
}
}
/*fprintf(fp,"\n bmet");
```

```
//printf("\n bmet");
for(j=0;j<8;j++)
{
for(k=0;k<4;k++)
{
fprintf(fp,"  %f",bmet[k][j]);
//printf(" %f",bmet[k][j]);
}
//printf("\n");
fprintf(fp,"  \n");
}*/
for(k=1;k<=8;k++)
{
for(j=0;j<8;j++)
{
if(j%2==0)
fsmet[j][k]=max(bmet[0][k-1]+fsmet[LUTFS[0][j]][k-1],bmet[3][k-
1]+fsmet[LUTFS[1][j]][k-1]);
else
fsmet[j][k]=max(bmet[1][k-1]+fsmet[LUTFS[0][j]][k-1],bmet[2][k-
1]+fsmet[LUTFS[1][j]][k-1]);
}
}
/*fprintf(fp,"\n forward st metric is");
for(k=0;k<=8;k++)
{
for(j=0;j<8;j++)
fprintf(fp," %f",fsmet[j][k]);
fprintf(fp,"  \n");
}*/
if(i==l-1)
```

```
break;
min=minimum(fsmet,k-1);
fprintf(fp,"min=%f",min);
for(k=0;k<8;k++)
fsmet[k][0]=fsmet[k][8]-min;
}//end of i loop
for(k=7;k>=0;k--)
{
for(j=0;j<8;j++)
{
if((j==0)||(j==1)||(j==4)||(j==5))
rsmet[j][k]=max(bmet[0][k]+rsmet[LUTBS[0][j]][k+1],bmet[3][k]+rsmet[LUTB
S[1][j]][k+1]);
else
rsmet[j][k]=max(bmet[1][k]+rsmet[LUTBS[0][j]][k+1],bmet[2][k]+rsmet[LUTB
S[1][j]][k+1]);
}
}
/*fprintf(fp,"\n reverse st metric is");
for(k=8;k>=0;k--)
{
for(j=0;j<8;j++)
fprintf(fp,"  %f",rsmet[j][k]);
fprintf(fp,"  \n");
}*/
//Calculating likelihood ratio for each decoded bit
for(k=0;k<8;k++)
{
num1=fsmet[0][k]+rsmet[4][k+1]+bmet[3][k];
den1=fsmet[0][k]+rsmet[0][k+1]+bmet[0][k];
for(j=1;j<8;j++)
```

```
{
if((j==2)||(j==3)||(j==6)||(j==7))
{
num=fsmet[j][k]+rsmet[LUTBS[1][j]][k+1]+bmet[3][k];
den=fsmet[j][k]+rsmet[LUTBS[0][j]][k+1]+bmet[0][k];
}
else
{
num=fsmet[j][k]+rsmet[LUTBS[1][j]][k+1]+bmet[2][k];
den=fsmet[j][k]+rsmet[LUTBS[0][j]][k+1]+bmet[1][k];
}
maxnum=max(num,num1);
maxden=max(den,den1);
//fprintf(fp,"\n
num1=%f,den1=%f,num=%f,den=%f,naxnum=%f,maxden=%f",num1,den1,nu
m,den,maxnum,maxden);
num1=maxnum;
den1=maxden;
}
app[k]=maxnum-maxden;
//fprintf(fp,"\nlr=%f",app[k]);
}
for(k=0;k<8;k++)
exlr[k]=app[k]-apri[k];
//Logic for hard decision
//deinterleaving+interleaving
for(j=0;j<24;j+=3)
{
printf("\n x1=");
if(j%6==0)
{
```

```
x1=2*inleav[j/6+1];
printf("%d",x1);
}
else
{
for(k=1;k<=4;k++)
{
if((j/3+1)==2*inleav[k])
{
x1=2*k-1;
break;
}
}
printf("%d",x1);
}//else
apri[x1-1]=exlr[j/3];
lr[x1-1]=app[j/3];
}//end of j
for(k=0;k<8;k++)
rsmet[k][8]=rstemp1[k];
if(big==0)
{
k=0;
min=minimum(rsmet,k);
fprintf(fp,"rsmin=%f",min);
for(k=0;k<8;k++)
rstemp4[k]=rsmet[k][0]-min;
}
if(big==1)
{
k=0;
```

```
min=minimum(rsmet,k);
fprintf(fp,"rsmin=%f",min);

for(k=0;k<8;k++)
rstemp6[k]=rsmet[k][0]-min;
}
if(big==2)
{
k=0;
min=minimum(rsmet,k);
fprintf(fp,"rsmin=%f",min);
for(k=0;k<8;k++)
rstemp8[k]=rsmet[k][0]-min;
}
}//end of big
for(k=0;k<8;k++)
{
if(lr[k]>0.0)
decod[k]=1;
else
decod[k]=0;
}
//Displaying the decoded string with respective LLR values
printf("decoded bits are:\n");
fprintf(fp,"decoded bits are:\n");
printf("Bit   LLR Value\n");
fprintf(fp,"Bit   LLR Value\n");
for(k=7;k>=0;k--)
{
printf("%d   %f",decod[k],lr[k]);
fprintf(fp,"%d  %f",decod[k],lr[k]);
```

```
putc(decod[k]+'0',ofp);
printf("\n ");
fprintf(fp,"\n");
}
}//end of l loop
// writing byte file in order into another file
fclose(ifp);
fclose(ofp);
if((ifp=fopen("C:\\write4.txt","r"))==NULL)
printf("\n ERROR:Input file could not be opened");
if((ofp=fopen("C:\\write5.txt","w"))==NULL)
printf("\n ERROR:Output file could not be opened");
n=0;
while((c=getc(ifp))!=EOF)
n++;
printf("\n count=%ld",n);
for(i=n-1;i>=0;i--)
{
fseek(ifp,i,0);
if((c=getc(ifp))!=EOF)
putc(c,ofp);
}
fclose(ifp);
fclose(ofp);
//converting binary file into alphanumeric file
if((ifp=fopen("C:\\write5.txt","r"))==NULL)
printf("\n ERROR:Input file could not be opened");
if((ofp=fopen("C:\\write6.txt","w"))==NULL)
printf("\n ERROR:Output file could not be opened");
n=0;
while((c=getc(ifp))!=EOF)
```

```
n++;
printf("\n count=%d",n);
for(i=0;i<n/8-1;i++)
{
c1=0;
for(j=0;j<8;j++)
{
fseek(ifp,8*i+j,0);
c=getc(ifp);
c=c-'0';
c2=c;
for(k=0;k<j;k++)
c2*=c*2;
c1+=c2;
}
printf("%c",c1);
fseek(ofp,i,0);
putc(c1,ofp);
}
fclose(ifp);
fclose(ofp);
}
//function for calculating maximum value
double max(double a,double b)
{
double temp,sub;
temp=(a>=b)?a:b;
sub=(a>=b)?a-b:b-a;
temp+=log(1+exp(-sub));
return temp;
}
```

File-7

```
//program to encode  input bit stream using Systematic Convolutional Encoding
// and odd-even symmetrical interleaver with rate 1/3
# include<stdio.h>
# include<conio.h>
void main()
{
int a,c,ch,ch1,r1[3],r2[3],arr[8],inleav[5];
unsigned long i,j,k,n=0;
FILE *ifp,*ofp;
//r1[3]:shift register of0 CC1 to store three bits
//r2[3]=shift register ogf CC2 to store three bits
//a,c,ch,ch1,i,j.n: local variabls used for calculating other parameters
//arr[]: aray to d
//inleav[5]: array to conatin interleverpattern
//*ifp: pointer to input file
//*ofp: pointer to output file
clrscr();
if((ifp=fopen("C:\\read.txt","r"))==NULL)
printf("\nERROR:1st Input file could not be opened");
if((ofp=fopen("C:\\write.txt","w"))==NULL)
printf("\nERROR:1st Output file could not be opened");
//converting alphabetick file into byte file
while((c=getc(ifp))!=EOF)
{
ch=c;
for(i=0;i<8;i++)
{
a=ch%2;
ch/=2;
putc(a+'0',ofp);
}
}
fclose(ifp);
fclose(ofp);
if((ifp=fopen("C:\\write.txt","r"))==NULL)
printf("\n ERROR:2nd Input file could not be opened");
if((ofp=fopen("C:\\write1.txt","w"))==NULL)
printf("\n ERROR: 2nd Output file could not be opened");
//counting no of bits in input file
while((c=getc(ifp))!=EOF)
n++;
printf("\n count=%d",n);
fseek(ifp,0,0);
//assiging values to inetrleaver to simualte odd-even symmetric interleaver
structure
inleav[0]=0;
inleav[1]=3;
inleav[2]=1;
inleav[3]=4;
inleav[4]=2;
for(i=0;i<3;i++)
{
r1[i]=0;
r2[i]=0;
}
//one bit of the 3 bit codeword is corresponding tranmitting bit
```

```
for(i=0;i<n;i++)
{
if((c=getc(ifp))!=EOF)
{
c=c-'0';
putc(c+'0',ofp);
}
}
fseek(ifp,0L,SEEK_SET);
//Logic for calculation of 2nd and 3rd bit of 3-bit codeword (parity bit)
for(i=0;i<n;i++)
{
if((c=getc(ifp))!=EOF)
{
int temp;
c=c-'0';
temp=r1[2]^r1[0]^c;
ch=temp^r1[0]^r1[1]^r1[2];
putc(ch+'0',ofp);
r1[2]=r1[1];
r1[1]=r1[0];
r1[0]=temp;
}
}
fseek(ifp, 0L, SEEK_SET);
for(i=0;i<n/8;i++)
{
for(j=0;j<8;j++)
{
if((c=getc(ifp))!=EOF)
{
```

```
int temp,x1;
c=c-'0';
if(j%2==0)
{
x1=2*inleav[j/2+1];
}
else
{
for(k=1;k<=4;k++)
{
if((j+1)==2*inleav[k])
{
x1=2*k-1;
break;
}
}
}
arr[x1-1]=c;
}
}
for(j=0;j<8;j++)
{
int temp=r2[2]^r2[0]^arr[j];
ch=temp^r2[0]^r2[1]^r2[2];
putc(ch+'0',ofp);
r2[2]=r2[1];
r2[1]=r2[0];
r2[0]=temp;
}
}
fclose(ifp);
fclose(ofp);
if((ifp=fopen("C:\\write1.txt","r"))==NULL)
printf("\n ERROR:3nd Input file could not be opened");
if((ofp=fopen("C:\\write2.txt","w"))==NULL)
printf("\n ERROR: 3nd Output file could not be opened");
i=0;
j=n;
k=2*n;
fseek(ifp,0L,0);
//rearrranging bits in sequence as one data bits,one first parity bit and
// one second parity bit and so on
while(i<n)
{
fseek(ifp,i,0);
c=getc(ifp);
putc(c,ofp);
fseek(ifp,j,0);
c=getc(ifp);
putc(c,ofp);
fseek(ifp,k,0);
c=getc(ifp);
putc(c,ofp);
i+=1;
j+=1;
k+=1;
}
printf("i=%ld,j=%ld,k=%ld",i,j,k);
for(i=0;i<24;i++)
putc(0+'0',ofp);
}
```

File-8

```
//program to decode uncoded-noise corrupted bit stream(Turbo)
# include<stdio.h>
# include<conio.h>
void main()
{
int i,j,k,c,c1,c2,nt=11,nt1,nt2,nt3,rndi;
unsigned long n=0;
double ch,rndf;
FILE *ifp,*ofp;
//nt,nt1,nt2,nt3: variables used for generating AWGN Noise
//*ifp: pointer to input file
//*ofp: pointer to output file
//i,j,k,c,c1,c2,n,ch: variables to store temporary values
//rndi: random generated integer value
//rndf: random generated float value
clrscr();
printf("\n Enter the Random Noise Generator Coeff");
scanf("%d%d%d",&nt1,&nt2,&nt3);
if((ifp=fopen("C:\\write.txt","r"))==NULL)
printf("\n ERROR:2nd Input file could not be opened");
if((ofp=fopen("C:\\write7.txt","w"))==NULL)
printf("\n ERROR: 2nd Output file could not be opened");
//Counting no of inputs bits
while((c=getc(ifp))!=EOF)
n++;
//printf("\n count=%d",n);
//Logic for adding noise
for(i=0;i<n;i++)
{
```

```
fseek(ifp,i,0);
if((c=getc(ifp))!=EOF)
{
c=c-'0';
//printf("%d",c);
//if(i%72==0)

j=i%3;
switch(j)
{
case 0:nt=nt1;
break;
case 1:nt=nt2;
break;
case 2:nt=nt3;
}
if(c==0)
c=-1;
rndi=(rand()%nt);
rndf=(float)rndi/10;
if(rndi%2==0)
ch=c+rndf;
else
ch=c-rndf;
if(ch>0.0)
putc(1+'0',ofp);
if(ch<0.0)
putc(0+'0',ofp);
if(ch==0.0)
putc(rand()%2+'0',ofp);
}
```

```
}
fclose(ifp);
fclose(ofp);
n=0;
if((ifp=fopen("C:\\write7.txt","r"))==NULL)
printf("\n ERROR:Input file could not be opened");
if((ofp=fopen("C:\\write8.txt","w"))==NULL)
printf("\n ERROR:Output file could not be opened");
while((c=getc(ifp))!=EOF)
n++;
//printf("\n count=%d",n);
//decoding noise corrupted bit stream
for(i=0;i<n/8;i++)
{
c1=0;
for(j=0;j<8;j++)
{
fseek(ifp,8*i+j,0);
c=getc(ifp);
c=c-'0';
c2=c;
for(k=0;k<j;k++)
c2*=c*2;
c1+=c2;
}
putc(c1,ofp);
}
fclose(ifp);
fclose(ofp);
}
```

File-9

```
//program to decode uncoded-noise corrupted bit stream(Trellis)
# include<stdio.h>
# include<conio.h>
void main()
{
int i,j,k,c,c1,c2,nt=11,nt1,nt2,rndi;
unsigned long n=0;
double ch,rndf;
FILE *ifp,*ofp;
clrscr();
//nt,nt1,nt2: variables used for generating AWGN Noise
//*ifp: pointer to input file
//*ofp: pointer to output file
//i,j,k,c,c1,c2,n,ch: variables to store temporary values
//rndi: random generated integer value
//rndf: random generated float value
printf("\n Enter the Random Noise Generator Coeff");
scanf("%d%d",&nt1,&nt2);
if((ifp=fopen("C:\\write.txt","r"))==NULL)
printf("\n ERROR:2nd Input file could not be opened");
if((ofp=fopen("C:\\write7.txt","w"))==NULL)
printf("\n ERROR: 2nd Output file could not be opened");
//Counting no of inputs bits
while((c=getc(ifp))!=EOF)
n++;
//printf("\n count=%d",n);
//Logic for adding noise
for(i=0;i<n;i++)
{
```

```
fseek(ifp,i,0);
if((c=getc(ifp))!=EOF)
{
c=c-'0';
//printf("%d",c);
//if(i%72==0)

j=i%2;
switch(j)
{
case 0:nt=nt1;
break;
case 1:nt=nt2;
}
if(c==0)
c=-1;
rndi=(rand()%nt);
rndf=(float)rndi/10;
if(rndi%2==0)
ch=c+rndf;
else
ch=c-rndf;
if(ch>0.0)
putc(1+'0',ofp);
if(ch<0.0)
putc(0+'0',ofp);
if(ch==0.0)
putc(rand()%2+'0',ofp);
}
}
fclose(ifp);
fclose(ofp);
n=0;
if((ifp=fopen("C:\\write7.txt","r"))==NULL)
printf("\n ERROR:Input file could not be opened");
if((ofp=fopen("C:\\write8.txt","w"))==NULL)
printf("\n ERROR:Output file could not be opened");
while((c=getc(ifp))!=EOF)
n++;
//printf("\n count=%d",n);
//decoding noise corrupted bit stream
for(i=0;i<n/8;i++)
{
c1=0;
for(j=0;j<8;j++)
{
fseek(ifp,8*i+j,0);
c=getc(ifp);
c=c-'0';
c2=c;
for(k=0;k<j;k++)
c2*=c*2;
c1+=c2;
}
putc(c1,ofp);
}
fclose(ifp);
fclose(ofp);
}
```