Lancaster University
Department of Engineering
Communication Research Centre

COMPUTATIONALLY EFFICIENT TURBO CODE IMPLEMENTATION

STUDENT: **Cagri Cagatay Tanriover**

SUPERVISOR: **Prof. Bahram Honary**

SUBMISSION DATE: **September 1998**

submitted to Lancaster University for
partial fulfillment of the degree of Master of Science in
Digital Signal Processing Applications
in Communication Systems

ABSTRACT

Communications researchers have actively sought error correcting codes that will achieve approaching the levels of performance promised by Shannon nearly 50 years ago. It is known that error correcting codes provide maximum efficiency gain from the information channel with minimum degradation in the transmitted message, which has always been the focus of interest in all the implemented communication systems up to date.

Turbo codes are an important development, which have been demonstrated to perform close to the Shannon limit. This project is designed to investigate the application of turbo codes with a means to producing the most computationally efficient implementation with low complexity.

The existing turbo code implementation produced within the Communications Research Centre was used to provide a point of reference on which to base the study. Performance evaluation of this implementation was observed in terms of bit error rate and processing time to provide a basis of comparison.

Through careful analysis, the conventional structure was examined, programming techniques were simplified and whenever possible simple logic operations were implemented in the designed modified code to achieve an increase in operational speed of approximately 35%.

This project forms a useful study in how practical turbo codes can be efficiently implemented with the combination of low-level programming advantages and the powerful high-level programming algorithms.

LIST OF SYMBOLS

FEC: Forward error correction

PSK: Phase shift keying

BPSK: Binary phase shift keying

RSC code: Recursive systematic convolutional code

IIR: Infinite impulse response

AWGN: Additive White Gaussian Noise

SOVA: Soft output Viterbi algorithm

MAP: Maximum a posteriori probability

SNR: Signal-to-noise ratio

BER: Bit error rate

Codec: Coder-decoder

u_k: Branch information symbol (±1).

$L(u'_k)$: The soft output of the decoder. This value gives the improved probability of ones and zeros in the encoded stream.

$L(u_k)$: The initial a priori information of the probability of ones and zeros in the encoded stream.

$P(u_k=+1|y)$: The joint probability of the transmitted symbol being +1 given that y is received.

$P(u_k=-1|y)$: The joint probability of the transmitted symbol being -1 given that y is received.

p(s',s,y): The probability of state transition from s' to s, for a received symbol y.

L_c: The channel information. (This term is given by 4*(SNR) for a Gaussian channel.)

$x_{k,v}$: The v^{th} bit of the k^{th} codeword.

$y_{k,v}$: The v^{th} bit of the k^{th} received codeword.

II: Interleaved information.

UI: Uninterleaved information.

P: Parity.

FP: Flushed parity.

FI: Flushed information.

0: Zero input.

E_b: Bit energy.

N_o: Noise power spectral density.

σ: Standard deviation.

LSR: Logical shift right.

LSL: Logical shift left.

K: Constraint length

DSP: Digital Signal Processing

LIST OF FIGURES

TABLE OF CONTENTS

CHAPTER 1- INTRODUCTION

1.1)- Background

Channel coding refers to the class 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, fading, and jamming. Usually, the goal of channel coding is to reduce the probability of bit error (P_b), or to reduce the required E_b/N_o, at the cost of expending more bandwidth than would otherwise be necessary. The exceptions to this are the combined modulation and coding techniques for bandlimited channels [9].

Practical motivation for the use of coding is to reduce the required E_b/N_o for a fixed bit error rate. This reduction in E_b/N_o may, in turn, be exploited to reduce the required transmission power or to reduce the hardware costs by requiring a smaller antenna size in the case of radio communications [10].

However, there exists a limiting value of E_b/N_o below which there can be no error-free communication at any information rate. This value for E_b/N_o is –1.59 dB and is called the *Shannon limit*. In practice, it is not possible to reach the Shannon limit since the bandwidth requirement and the implementation complexity increase without bound. Shannon's work has provided a theoretical proof for the existence of codes that could improve the P_b performance, or reduce the E_b/N_o required, from the levels of the uncoded binary modulation schemes to levels approaching the limiting curve [9].

Error correcting codes are used to increase the efficiency and accuracy of information transfer from one place to another. In communications transmission, data is transferred from a transmitter to a receiver across a physical medium called *channel*, which is generally affected by noise, that introduces error in the data being transferred. Thus, error correcting codes are needed and used for correction of errors introduced in the channel. They are signal-processing techniques that are used

to improve the reliability of digital data transmission. This is achieved by encoding the transmitted data and introducing such redundancy that the decoder can later reconstruct the data transmitted. These codes have evolved from containing simple linear operations to more complex convolutional, iterative, feedback operations.

Turbo codes are a new series of error correcting codes, which were introduced in 1993, in the paper titled "*Near Shannon Limit Error-Correcting Coding and Decoding: Turbo* Codes" by C. Berrou, A. Glavieux and P. Thitimajshima. Performance this new class of codes, in terms of bit error probability, has been shown to be very close to the Shannon limit [1,2,3,4,5]. Therefore, this topic has been the focus of attention in the coding community due to the remarkable coding gains close to theoretical limits.

As turbo coding offers an improved BER performance compared to that of other error correcting codes, it appears to be an attractive field of research. This project was carried out to revise the theoretical background and to introduce complexity reduction in an available system. Additionally, simulations have been performed to test the effects of the interleaver length, number of iterations and SNRs on the bit error performance.

1.2)- Aims

The purpose of this project was to design a low complexity turbo codec by using a conventional source code, in which simplified programming techniques were introduced. The work also involved familiarisation with turbo coding techniques, through theory and available applications. The comparative performance analysis of the modified and the conventional system was made in terms of BER and the degree of complexity.

1.3)- Methodology

The project consisted of two areas:

1)- Implementation of turbo codec in standard C programming language.

2)- Complexity comparison of the conventional and the modified program codes.

The methods followed in these stages will be explained in this section.

The codec was implemented in C programming language keeping the source code as simple as possible. An existing turbo codec program [5] was used as a point of reference for the modified system, in which sub-functions were improved by using efficient programming methods. Simulations of the codec were carried out to provide feedback on complexity reduction.
The articles written on turbo codes were reviewed in order to get a better understanding of the necessary theoretical information. The gathered information was recorded carefully and was frequently revised. This formed the theoretical basis of the research in the project. In addition to that, previous work done including reports and source codes written by other researchers was studied with particular focus on the results obtained at the time. Additionally, web search was conducted in order to gather more information about the performance of turbo codes in various universities by various researchers. Revision of the background material also included literature search on coding, modulation and noise as it was necessary for the modelling of the turbo codec.
Due to the complexity of implementation of turbo codes, it was necessary to prepare flow charts and block diagrams at different stages of programming which helped in writing the C code in a simplistic way. The task was divided into sub-blocks, which could be implemented individually. In other words, the main program could be kept simple with sub-functions used throughout the program. In this way the program structure would be easily followed, making it easier for the researcher to trouble-shoot.

The execution of turbo codec on a personal computer is costly in terms of memory and processing power, with the use of long interleaver lengths (1000 and up). Therefore, a Unix environment was chosen for compiling and execution of the code. C code was written in a Windows environment, using Borland C++ 4.52 compiler and then executed in Genug C compiler in Unix.

Code design was initiated by the encoder structure, which was simple compared to the decoder. During this stage, the block diagrams and flowcharts were used and modified where necessary. The encoder structure comprised of puncturing, interleaving, preparation of the input and output information packets and state transitions for the given convolutional code structure (K=3).

The program was executed for the transmission of the binary data and the addition of Gaussian noise to the modulated signal. Noise was generated by using a look-up table of 1000 random noise values calculated for zero mean and a standard deviation of 1.

The received signal was fed to the decoder, which used maximum a posteriori probability (MAP) algorithm for decoding. Similar to the encoder, the decoder comprised of puncturing, interleaving, de-interleaving, iteration and hard decision blocks.

The code was designed so that during execution, simulation results were recorded. The code was completed by a data sink routine, which stored the simulation results in a file named ‘results.txt’.

After the code design was completed, simulations were carried out. Some errors resulting from the incompatibility of Unix compiler and the Borland C++ compiler were encountered and corrected. Also, problems with the number formats and the floating point errors were overcome by making changes in the code. This stage of the project was one of the most time consuming and tedious procedures in the overall work.

Having solved all the problems with the code, simulations were run to test the effects of parameter variations on the performance of turbo codec, involving signal-to-noise ratio, interleaver length, and the number of iterations. Some tests on the number of transmitted information bits and its effects on

the error performance were also made as they proved to be of great importance in the performance analysis.

After the completion of these stages, complexity comparison of the conventional and the modified source codes was performed.

In the conventional source code, certain variations had to be made in order to make the necessary complexity measurements i.e. processing time measurements for individual blocks. Time measurements were made by the use of stopwatch functions in the source code that have been included in Borland C++ compiler used. Initially the measurements were made for the encoder and the decoder sub-functions, and the overall simulation. Results showed that these blocks were the most influential parts on the speed performance, and hence were the focus of attention for the complexity reduction.

The important point in the complexity reduction was to make the source code run faster without any compromise in the error performance. In order to make a relevant comparison, the input information sequence, the number of iterations and the interleaver lengths were kept the same in the conventional and the modified code simulations. This way a direct processing time comparison between the two source codes could be made in seconds.

As mentioned earlier, the execution of the program was carried out in Unix environment, where access to a local server was necessary. Since the number of users accessing the server at a give time influenced the speed performance, average time figures of 50 simulations were recorded and analysed and hence the reliability of the results was increased. It should be noted that the results presented in Chapter 4 were recorded when the access to the server was close to zero.

As a final step, error and processing speed performance results were analysed and presented by means of plots and interpretations.

1.4)- Overview

The thesis has been organised in five chapters and three appendices.

Chapter 1 deals with the background and the aim of the project, together with the methodology followed throughout the project and the overview of the entire written work. Chapter 2 covers the necessary theoretical information for the project. Such as error control coding and phase shift keying. The topics in Chapter 2 have been ordered according to their significance in the work, the first one being the most significant. Chapter 3 comprises of the implementation of turbo codec in C programming language. In other words, the focus of this chapter is the actual work performed. It begins with the specifications of the codec structure and continues with a detailed explanation of the encoder, decoder and other major program blocks with the aid of relevant illustrations for each case. Results of the simulations have been included in Chapter 4 with the comparison of the conventional and the modified versions of the turbo codec. Chapter 5 includes the conclusion and the suggestions for further work.

Appendix A is included to provide the reader with the relevant background in constructing the least squares line, which has been used in Chapter 4. Listing of the modified program's source code is presented in Appendix B. In addition to the error performance plots presented in Chapter 4, various images have been included in Appendix C, to aid in visualizing the BER performance of the turbo codes.

CHAPTER 2 – THEORETICAL BACKGROUND

2.1)- Error Correcting Codes

For channel encoding, simple error detection with repetition is often insufficient; by proper encoding when an error is detected, it can be corrected. The codes currently used for error correction are very sophisticated and can correct many simultaneous errors in a block.

Error control for data integrity may be performed by means of forward error correction (FEC). For a digital communication system exploiting FEC, channel encoder in the transmitter accepts message bits and adds redundancy according to a prescribed rule, thereby producing encoded data at a higher bit rate. The channel decoder in the receiver exploits the redundancy to decide which message bits were actually transmitted. The combined goal of the encoder and the decoder is to minimize the effect of channel noise. That is, the number of errors between the channel encoder input (derived from the source) and the channel decoder output (delivered to the user) is minimized.

The addition of redundancy in the coded messages implies the need for increased transmission bandwidth. Moreover, the use of error-control coding adds complexity to the system, especially in the implementation of decoding operations in the receiver. Thus, the design trade-offs in the use of error-control coding to achieve acceptable error performance include considerations of bandwidth and system complexity.

There are many different error-correcting codes (with roots in diverse mathematical disciplines) that can be used. Historically, they have been classified into *block codes* and *convolutional codes*. The distinguishing feature for the classification is the presence or absence of memory in the encoders [10]**.** It may be asserted that, all encoding-decoding procedures invented so far are a combination of these two basic codes, namely block codes and convolutional codes [11].

2.1.1)- Block Codes

To generate an (n, k) block code, the channel encoder, which is a memoryless device, accepts information in successive k-bit blocks; for each block, it adds (n-k) redundant bits that are algebraically related to the k message bits, thereby producing an overall encoded block of n bits (n>k). The term 'memoryless' indicates that each n-symbol block depends only upon a specific k-symbol block and no others. The n-bit block is called a *code word*, and n is called the *block length* of the code. This code word of length n, which carries k information digits, is transmitted via the discrete channel. On this line there are $M = 2^k$ possible messages to be transmitted and, hence, 2^k possible code words out of n-tuples. The way which any code word is formed is characteristic of a particular block code (and hence, of a particular block encoder). Generally speaking, this way is described by giving a code matrix G that specifies how the (n-k) *redundancy digits* can be formed from the k info digits and added to them to constitute the message to be transmitted. Hence, the so-called dimensionless ratio *code rate* $r = k/n$, where $0 < r < 1$, will represent the fraction of information carried by any transmitted digit [11]. The channel encoder produces bits at the rate $R = (n/k)* R_s$, where R_s, is the bit rate of the information sequence. The bit rate R, coming out of the encoder, is called the *channel data rate*. Thus, the code rate is a dimensionless ratio, whereas the data rate produced by the source and the channel data rate are both measured in bits per second [10].

2.1.2)- Convolutional Codes

In 1955, Elias suggested *convolutional codes* as an extension of the linear block codes to allow memory to be extended for a given block. In fact, the parity check bits in a block of linear block codes depend only on the information digits in the same block. In other words, the *memory* of the information digits of a certain block is confined to the block itself.

However, it is possible to extend this concept by allowing the information digits in a block to influence not only the block itself but also a certain number of successive blocks. This is the fundamental character of the so-called *convolutional codes* [11].

In a convolutional code, the encoding operation may be viewed as the *discrete-time convolution* of the input sequence with the impulse response of the encoder. The duration of the impulse response equals the memory of the encoder. Accordingly, the encoder for a convolutional code operates on the incoming message sequence, using a 'sliding window' equal in duration to its own memory, which is defined as the constraint length (K) taken to be the length of the longest shift register in practice. This, in turn, means that in a convolutional code, unlike a block code, the channel encoder accepts message bits as a continuous sequence and thereby generates a continuous sequence of encoded bits at a higher rate [10]**.** In other words, the encoder, whose rate is given by the ratio of the simultaneous input digits to the number of output digits, converts the entire data stream, regardless of its length, into a single code word. If the input data stream is reproduced unaltered in the output code word, the code is said to be systematic, as in block codes.

2.2)- The Viterbi Algorithm

The Viterbi algorithm has become a standard tool in communication receivers, performing such functions as demodulation, decoding, equalization etc. [12]**.** The Viterbi algorithm was proposed in 1967 as a method of decoding convolutional codes. Since that time, it has been recognized as an attractive solution to a variety of digital estimation problems [13].

It is a powerful, fast and easily implemented technique for decoding convolutional codes. Its analysis and simulation are easily carried out. Also, the amount of the decoding computation needed is independent of the noise in the received sequence of channel symbols. The best match to the transmitted sequence is searched by the comparison of the received sequence to the relevant branch codes on the trellis.

All the possible paths in the trellis are considered as a transmitted word. The output words at each branch are compared to the received word for calculating the degree of difference (i.e. Hamming distance). The paths, which have the minimum Hamming distances up to a certain node in the trellis, are called the *survivor paths*. Having traveled through all the depths of the trellis, one survivor path, the maximum-likelihood path is retained, which at the same time is the path with the minimum metric.

2.3)- Soft versus Hard Decision Decoding

In hard decision decoding, the output of the modulator is quantized to two levels, namely 0 and 1, and fed into the decoder. Since the decoder operates on the hard decisions 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 the probability of the received wave, greater than two levels or with an unquantized or the analogue value of the waveform. When the number of possible output levels of the demodulator is greater than two, the decoding is called *soft-decision decoding.* Soft-decision decoding provides the decoder with a level of confidence in the received waveform. This additional piece of information helps to increase the error performance of the receiver. The price paid is the increased complexity of the receiver. Block decoding algorithms have been devised to operate with hard or soft decisions. However, soft decision decoding is generally not used with block codes because it is considerably more difficult than hard-decision decoding to implement. The most prevalent use of soft-decision decoding is with the Viterbi convolutional decoding algorithm, since with Viterbi decoding, soft decisions represent only a trivial increase in computation [9].

2.4)- Turbo Codes

Since the early days of information and coding theory the goal has always been to come close to the Shannon performance with a tolerable complexity. The results achieved so far show that it is relatively easy to operate at signal-to-noise ratios of E_b/N_o above the value determined by the channel cut-off rate.

Recently interest has focused on iterative decoding of product or concatenated codes using 'soft in/soft out' decoders with simple component codes in an interleaved scheme. The paradigm is to break up decoding of a complex and long code into steps while the transfer of probabilities or 'soft' information between the decoding steps guarantees almost no loss of information [1].

Turbo Codes were first introduced by Berrou, Glavieux, and Thitimajshima [3], [4]. For a bit error rate of 10^{-5} and code rate ½, the authors report a required E_b/N_o of 0.7 dB. The codes are constructed by applying two or more component codes to different interleaved versions of the same information sequence. For any single traditional code, the final step at the decoder yields hard-decision decoded bits (or more generally, decoded symbols). In order for a concatenated scheme such as a turbo code to work properly, the decoding algorithm should not limit itself to passing hard decisions among the decoders. To best exploit the information learned from each decoder, the decoding algorithm must affect an exchange of soft rather than hard decisions. For a system with two component codes, the concept behind turbo decoding is to pass soft decisions from the output of one decoder to the input of the other, and to iterate this process several times to produce better decisions [2].

In other words, a turbo code is a concatenated code decoded by an iterative decoding of the component codes using soft-input/soft-output decoders.

Turbo decoding can be performed by either MAP or SOVA algorithm. Within the limits of this project, MAP algorithm has been implemented. Therefore, detailed background information is

given only for this method. For information on SOVA algorithm, reader is referred to [1].

2.5)- The MAP Algorithm

In digital communication, all the techniques involved in the transmission of a given message boils down to a hard decision stage where a choice between one or a zero is made according to a prescribed rule. For example, for a BPSK antipodal modulation scheme, hard decision is made according to the *sign* of the received symbol. When soft decision is introduced, as mentioned earlier, a degree of confidentiality in the hard decision to be made comes into effect, which improves the reliability of decoding. The idea behind MAP algorithm is to determine this confidence level and to improve it through a feedback mechanism i.e iterative decoding. Due to the nature of the MAP algorithm, all the trellis states and branches are taken into account at each stage of trellis decoding as opposed to the Viterbi algorithm, where a maximum likelihood path is selected, and elimination of nonsurvivor paths takes place at each node.

The goal of the MAP algorithm [1] is to provide us with a logarithmic ratio, which compares the probability of a received symbol y being +1 with that of –1 (for the modulation used in this work), as shown in (2.1).

$$L(u_k') = \log\left[\frac{P(u_k = +1 \mid y)}{P(u_k = -1 \mid y)}\right] = \log\left[\frac{\sum_{(s',s).u_k=+1} p(s', s, y)}{\sum_{(s',s).u_k=-1} p(s', s, y)}\right] \qquad \textbf{(2.1)}$$

The index pair s’ and s determines the information bit u_k and the coded bits $x_{k,v}$, for v = 2,…….,n. The sum of the joint probabilities p(s’, s, y) in the numerator or in the denominator of (2.1) is taken

over all existing transitions from state s' to state s labeled with the information bit $u_k = +1$ or -1, respectively. This is the soft output of the decoder at the end of all the iterations, to which hard decision is applied for a final output.

Assuming a memoryless transmission channel, the joint probability p(s', s, y) can be written as the product of three independent probabilities:

$$p(s',s,y) = p(s',y_{j<k}) \cdot p(s,y_k|s') \cdot p(y_{j>k}|s) \tag{2.2}$$

which can also be expressed as

$$p(s',s,y) = \alpha_{k-1}(s') \cdot \gamma_k(s',s) \cdot \beta_k(s) \tag{2.3}$$

In (2.2) $y_{j<k}$ denotes the sequence of received symbols y_j from the beginning of the trellis up to time k-1 and $y_{j>k}$ is the corresponding sequence from time k+1 up to the end of the trellis. It should be noted that y symbols are received through the channel, and therefore they are expressed as floating point values.

The forward recursion of the MAP algorithm yields,

$$\alpha_k(s) = \sum_{s'} \gamma_k(s',s) \cdot \alpha_{k-1}(s') \tag{2.4}$$

The backward recursion yields,

$$\beta_{k-1}(s') = \sum_{s} \gamma_k(s', s) \cdot \beta_k(s)$$

(2.5)

In other words, the decoding trellis is scanned twice; once from the beginning to the end (forward recursion) and once vice versa (backward recursion). Alpha values are calculated for the first pass, namely from the beginning to the end whereas beta values are calculated for the second pass through the trellis. After all the beta and alpha calculations are completed, each trellis state ends up having one alpha and one beta value for a given code word. Alpha values *involve the transition to a given future state, from all the possible past states merging at that future state*. Beta values, on the other hand, are *related to the state transitions from a given past state to the possible future states emerging from that past state*. It can be seen from (2.4) and (2.5) that all alpha and beta calculations have the probability information on the previous alpha and beta values, respectively, as the calculation is recursive.

In order to perform the optimum 'symbol-by-symbol' MAP rule, the trellis has to be of finite duration. We assume that at the start and at the end of the observed sequence, all paths merge at the zero state. The forward and backward recursion are initialized with $\alpha_{start}(0)=1$ and $\beta_{end}(0)=1$.

As presented earlier, (2.4) and (2.5) make use of all the state transition probabilities, namely gamma values. Whenever a transition between s' and s exists, the branch transition probabilities are given by (2.6), by which the calculation of gamma values is initiated.

$$\gamma_k(s', s) = p(y_k | u_k) \cdot P(u_k)$$

(2.6)

(2.6) involves the computation of the probability of the information symbol u_k being transmitted, $P(u_k)$, and the joint probability of a given u_k being received and y_k corresponding to that u_k value, $p(y_k|u_k)$. Note that gamma calculations are done for every branch in the decoding trellis, and therefore the u_k and y_k of interest are those for *a certain branch code word*.

Using the log-likelihoods, the a priori probability $P(u_k)$ can be expressed as,

$$P(u_k) = \frac{e^{\pm L(u_k)}}{1+e^{\mp L(u_k)}} = \left[\frac{e^{-L(u_k)/2}}{1+e^{-L(u_k)/2}}\right] \cdot e^{L(u_k)\cdot u_k/2}$$
$$P(u_k) = A_k \cdot e^{L(u_k)\cdot u_k/2}$$

(2.7)

Similarly for $p(y_k|u_k)$,

$$p(y_k|u_k) = B_k \cdot e^{\left(\frac{1}{2}\cdot L_c \cdot y_{k,1}\cdot u_k + \frac{1}{2}\cdot \sum_{v=2}^{n} L_c \cdot y_{k,v}\cdot x_{k,v}\right)}$$

(2.8)

Here, Lc is referred to as the channel information and is given by $4*(E_B/N_o)$ for a Gaussian channel. A_k and B_k for all transitions from level k-1 to level k are equal and hence will cancel out in the ratio in (2.1). Therefore, the branch transition operation to be used in (2.4) and (2.5) reduces to,

$$\gamma_k(s',s) = e^{\cdot\left(\frac{1}{2}\cdot u_k\cdot\left[L_c\cdot y_{k,1}+L(u_k)\right]\right)} \cdot \gamma_k^{(e)}(s',s)$$

(2.9)

with,

$$\gamma^{(e)}(s',s) = e^{\cdot\left(\frac{1}{2}\cdot\sum_{v=2}^{n} L_c \cdot y_{k,v} \cdot x_{k,v}\right)}$$

(2.10)

Extrinsic gamma, as can be seen from (2.10), provides information on the parity check symbols in the trellis $x_{k,v}$, and the received parity symbols, $y_{k,v}$. Since the first exponential function in (2.9) is common in all terms in the sums of (2.1), we divide all terms by those and obtain,

$$L(u_k') = L_c \cdot y_{k,1} + L(u_k) + \log\left[\frac{\sum_{(s',s),u_k=+1} \gamma_k^{(e)}(s',s)\cdot\alpha_{k-1}(s')\cdot\beta_k(s)}{\sum_{(s',s),u_k=-1} \gamma_k^{(e)}(s',s)\cdot\alpha_{k-1}(s')\cdot\beta_k(s)}\right]$$

(2.11)

Note that in (2.11), the log term represents the extrinsic information, which represents the soft output obtained after each iteration. Therefore, at each iteration, this log ratio is replaced by the new value, $L(u_k')$ calculated in this formula. The iterations are carried out for a number of times until no significant improvement in the decoding is achieved. When gamma is used in stead of extrinsic gamma in the log ratio in (2.11), $L(u_k')$ can be directly equated to the log ratio without including the first two terms.

In general, MAP algorithm can be thought as the calculation of *state transition probabilities* in both directions through the entire depth, and *branch transition probabilities*, for a given trellis structure, and *improving the soft output* by exploiting the extrinsic information obtained after each iteration.

2.6)- White Noise

The analysis of communication systems is customarily based on an idealized form of noise called *white noise*, the power spectral density of which is independent of the operating frequency.

We express the power spectral density of white noise as (2.12), which is illustrated in Figure 2.6.1(a). The unit of N_o is in watts per Hertz. The parameter N_o is usually referenced to the input stage of the receiver of a communication system.

$$S(f) = \frac{N_o}{2} \tag{2.12}$$

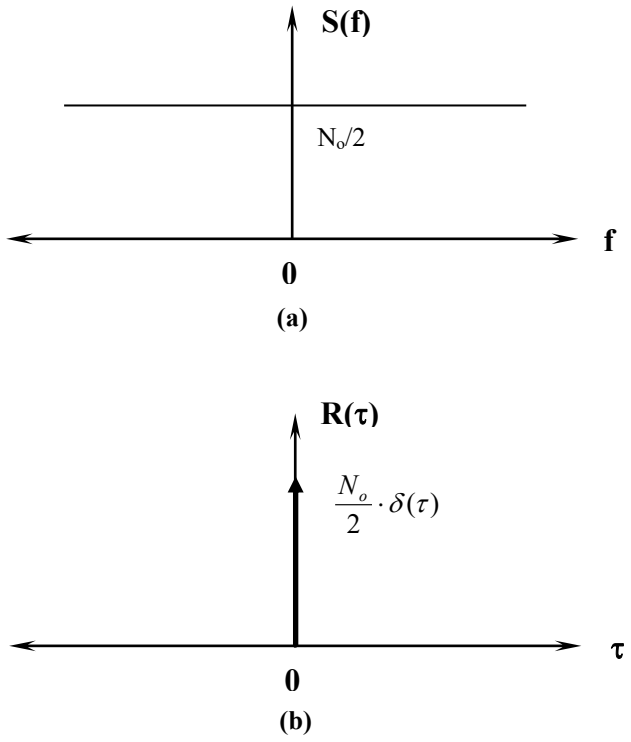


Figure 2.6.1- Characteristics of white noise. **(a)-** Power spectral density. **(b)-** Autocorrelation function.

Since the autocorrelation function is the inverse Fourier transform of the power spectral density, it follows that for white noise

$$R(\tau) = \frac{N_o}{2} \cdot \delta(\tau)$$

(2.13)

That is, the autocorrelation function of white noise consists of a delta function weighted by the factor $N_o/2$ and occurring at $\tau=0$, as in Figure 2.6.1(b). We note that $R(\tau)$ is zero for $\tau \neq 0$. Accordingly, any two different samples of white noise, no matter how closely in time they are taken, are uncorrelated. If the white noise is also Gaussian, then the two samples are statistically independent. In a sense, white Gaussian noise represents the ultimate in 'randomness'.
Strictly speaking, white noise has infinite average power and, as such, it is not physically realizable. Nevertheless, white noise has simple mathematical properties exemplified by the formulas above, which make it useful in statistical system analysis [10].

2.6.1)- Variance of White Noise

White noise is an idealized process with two-sided power spectral density equal to a constant, $N_o/2$, for all frequencies from $-\infty$ to $+\infty$. Hence the noise variance (average noise power, since the noise has zero mean) is given by (2.14).

$$\sigma^2 = \int_{-\infty}^{+\infty} \left(\frac{N_o}{2} \right) \cdot df = \infty$$

(2.14)

Although the variance for AWGN is infinite, the variance for filtered AWGN is finite [9]. For example, if AWGN is correlated with one set of a set of orthonormal functions, the variance of the correlator output is given by

$$\sigma^2 = \frac{N_o}{2}$$

(2.15)

2.7)- Phase Shift Keying

Phase shift keying was developed in the early days of the deep-space program; PSK is now widely used in both military and commercial communications systems [9].

PSK is a digital modulation technique where a single tone is used for transmission and the wave phase is shifted according to the bits transmitted. In other words the information of the wave is stored in the phase of the wave.

The general analytic expression for PSK is

$$s_i(t) = \sqrt{\frac{2E}{T}} \cdot \cos[\omega_o \cdot t + \phi_i(t)] \qquad 0 \le t \le T, \quad i = 1,2,\ldots,M$$

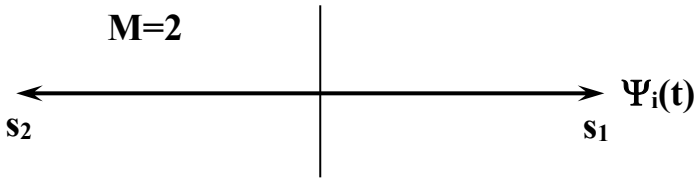
(2.16)

where the phase term, $\phi_i(t)$, will have M discrete values typically given by

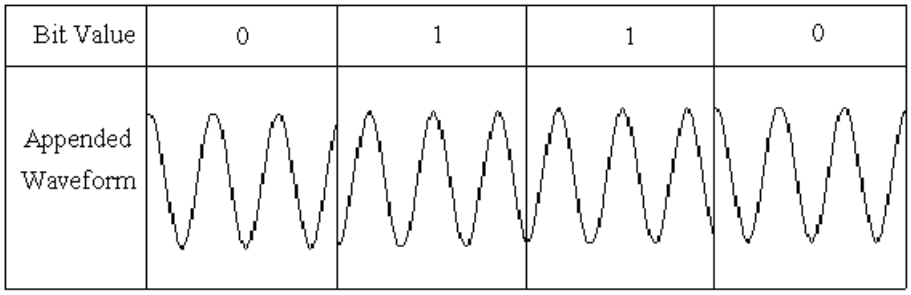
$$\phi_i(t) = \frac{2\pi i}{M} \qquad i = 1,2,\ldots.M$$

(2.17)

For the binary PSK (BPSK) example in Figure 2.7.1, M is 2. The parameter E is the symbol energy, T is symbol time duration, and $0 \leq t \leq T$. In BPSK modulation, the modulating data signal shifts the phase of the waveform, $s_i(t)$, to one of two states, either zero or π (180^o). The waveform sketch in Figure 2.7.1b shows a typical BPSK waveform with its abrupt phase changes at the symbol transitions. Figure 2.7.1a shows the vector representation of BPSK modulation. Signal sets with such opposing vectors are called *antipodal signal sets* [9].



(a)



(b)

FIGURE 2.7.1- Illustration of BPSK modulation. **(a)-** Vector representation. **(b)-** Waveform.

CHAPTER 3- SIMULATION IMPLEMENTATION AND OPERATION

3.1)- Specifications

The turbo codec was designed for constraint length, K=3. The implemented code was a recursive systematic convolutional code (RSC) of rate ½.

3.2)- Turbo Encoder

Block diagram of the turbo encoder has been presented in Figure 3.2.2. The two convolutional encoder blocks used in the simulation were concatenated in parallel, and were identical in terms of structure.

The code was designed to be systematic, and hence, the information bits were retained in the encoded stream. Parity bits were generated by:

$$p_{ik} = x_{k-2} \oplus \left[(x_{k-1} \oplus x_{k-2}) \oplus d_k \right] \tag{3.1}$$

where,

p_{ik}: Parity bit of the i^{th} encoder at time k.

x_{k-1}: The content of the k-1 register. (see Figure 3.2.2)

x_{k-2}: The content of the k-2 register (the oldest input). (see Figure 3.2.2)

d_k: The information bit at time k.

In the program, the encoder block also constructs the information packet as an array according to Figure 3.2.1.

array index interval	**0.....(i-1)**	**i.....(3i/2)-1**	**3i/2 .....(2i-1)**	**2i.....(2i+$2^{(K-1)}$)-1**	**(2i+2^{K-1}) .... (2i+2^{K})-1**
stored data	information bits	outputs from encoder 1	outputs from encoder 2	flushed outputs for encoder 1	Flushed outputs for encoder 2

FIGURE 3.2.1- Encoded information packet.

where ‘i’, and ‘K’ stand for the interleaver length and constraint length, respectively. This information packet is then fed to the AWGN channel.

3.2.1)- Interleaver

Interleaver is a device that rearranges (or permutes) the ordering of a sequence of symbols in a deterministic manner. In cases where burst errors are a problem, one potential solution involves utilizing a random error correcting encoders/decoders with a suitable interleaver/de-interleaver pair [14].

The interleaver block in the project is used in order to provide statistical independence between the two encoders’ input data. This independence provides better estimation of the feedback information, which is important in the iteration process involved in turbo coding. In other words, the parity bits generated by the encoders can be regarded to root from two statistically independent versions of the same input data. This way more information has been provided to the decoder for improved BER performance. After interleaving, the input data appears to be random to the channel but yet it includes enough information for decoding. The interleaver block used in the project is *pseudo random* [14]. Figure 3.2.3 illustrates the operation of the interleaver.

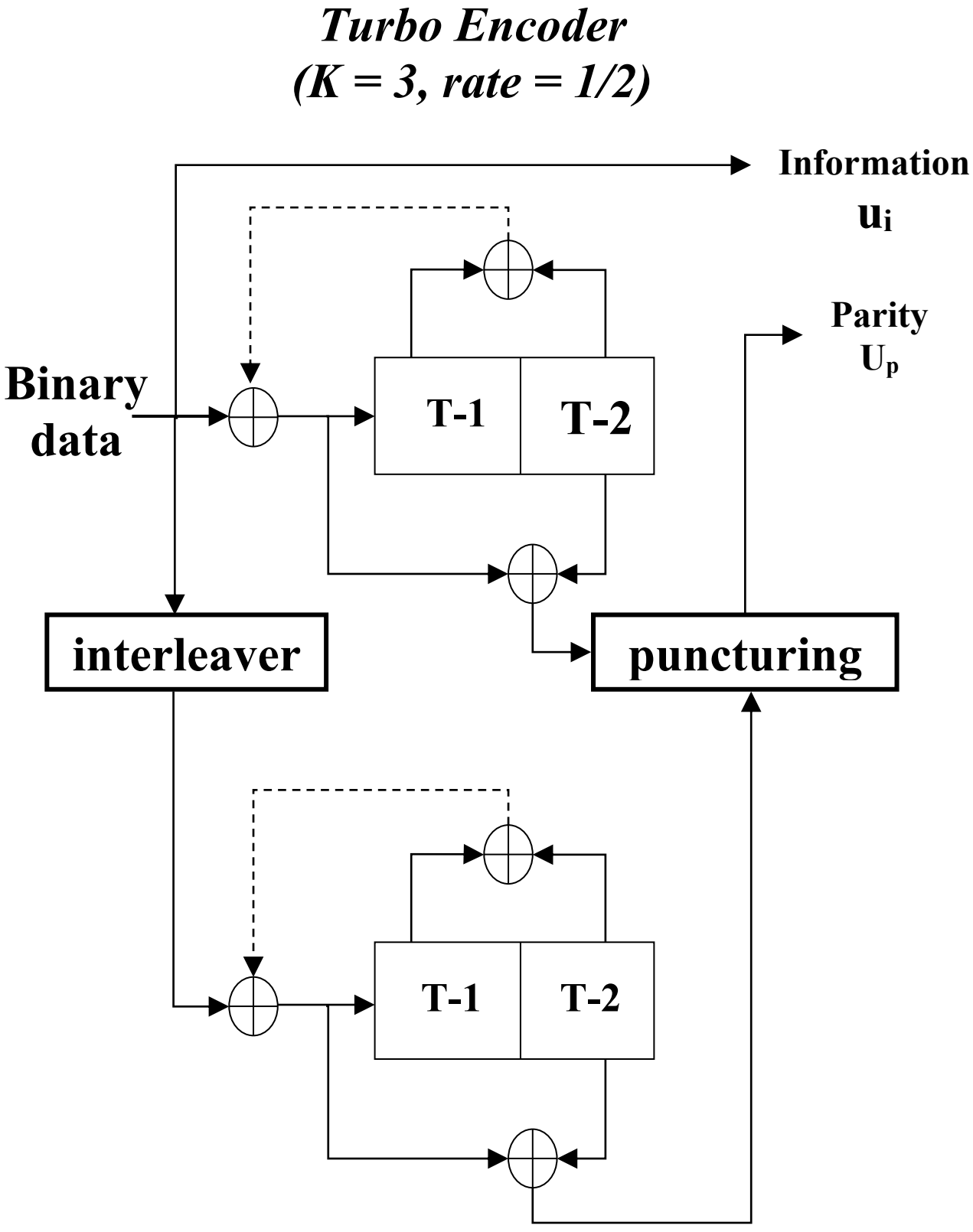


FIGURE 3.2.2- Turbo Encoder block diagram.

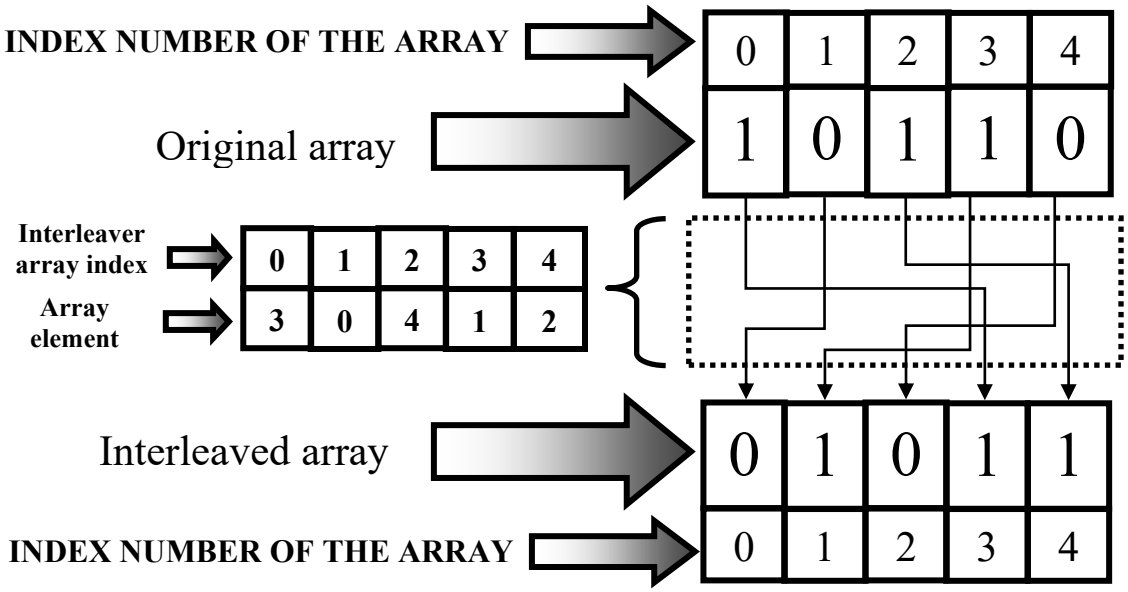


FIGURE 3.2.3- Interleaving operation.

Five different numbers are interleaved in this example, which indicates that the interleaver length is five. The original and the interleaved stream are of the same length and have the same elements. The only difference is the order of those elements,which is determined randomly by using a simple permutation routine in the program. Note that the numbers interleaved are ones and zeros.

The interleaver implemented in the source code, works as illustrated in Figure 3.2.2. It should be noted that the random numbers generated were between zero and (interleaver length – 1), which totals to the interleaver length. The permutation generated in the interleaver block is stored in the program, and corresponds to the index of the array that keeps the information bits. According to the index specified in the interleaver array, the information bit corresponding to that index is picked up and processed as required.

The interleaving process is used in the encoder section. Another operation that does the inverse of interleaving is referred to as ‘de-interleaving’ and used in the decoder as it shall be discussed later.

3.2.2)- Puncturing

Often in applying coding to a communication system there may be constraints on the code block length, n, or the number of information symbols, k, that require values other than those which the code designer would prefer. In such cases one can usually modify the desired code to fit the particular constraints on n and k [14]. Puncturing is the deletion of parity symbols to decrease the block length. Puncturing operation is illustrated in Figure 3.2.4.

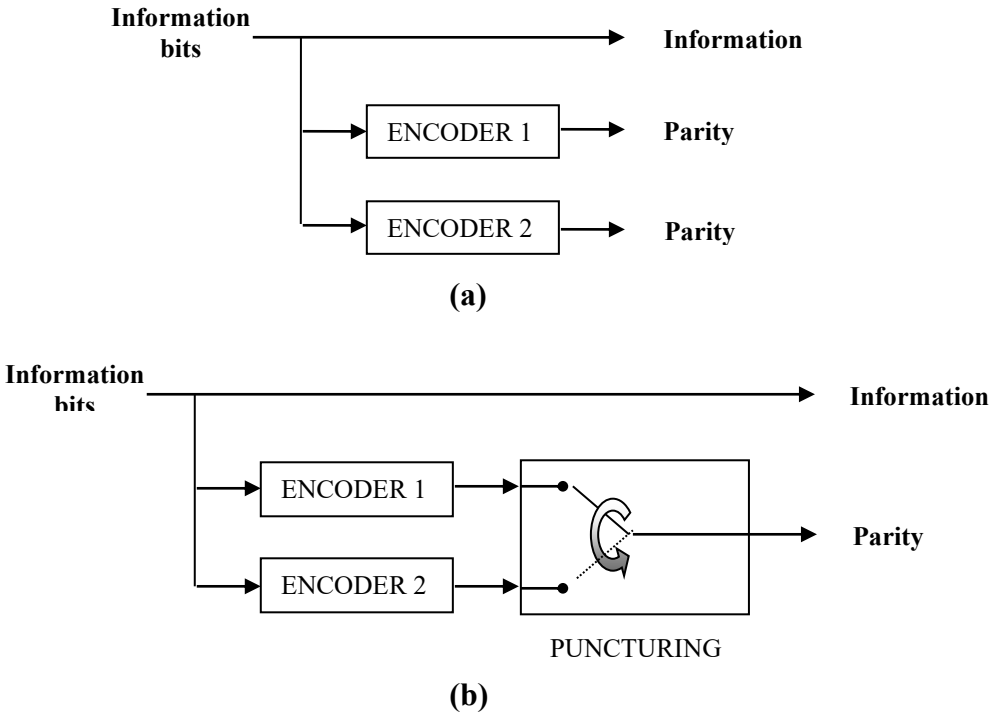


FIGURE 3.2.4- Puncturing operation. **(a)-** Before puncturing. (rate = 1/3)

(b)- After puncturing. (rate = ½)

As illustrated in Figure 3.2.4(a), for a given X information bits, 2X parity bits are generated (X parity bits from each encoder). Therefore the code rate in this case is 1/3. The insertion of a puncturing block after the encoders generates X parity bits for X information bits. In this case the code rate has been increased to ½ since parity bits are selected from each encoder interchangeably.

Due to the encoding structure implemented in the program code, puncturing was necessary for keeping the rate at ½. The even parity bits were picked from the first encoder whereas the odd ones were taken from the second encoder. This way an array was prepared for transmitting the information bits with the selected parity bits. Figure 3.2.5 illustrates the parity selection in the encoding process. As it can be seen, the code words are formed in pairs, each pair having an information bit and a parity bit from one of the encoders.

information bit index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
encoder 1	√	×	√	×	√	×	√	×	√	×
encoder 2	×	√	×	√	×	√	×	√	×	√

√: The encoded bit is picked. **×:** The encoded bit is discarded.

FIGURE 3.2.5- Parity check selection.

3.2.3)- Flushing

Flushing is the process of bringing the encoder states to zero after the encoding of a given number of information bits. For RSC codes this flushing can not be achieved simply by consecutive zero input to the encoder due to the IIR structure of the encoder. Consequently, each time an information bit block of interleaver length is encoded, it is necessary to *force* the encoder states to zero. Figure 3.2.6 illustrates how this mechanism works in the encoder structure implemented in the program code. The encoder structure is the same as the one presented in Figure 3.2.1. Here, one component code is considered for illustration of the flushing procedure. It should be noted that the two switches in the flushing selector are synchronized, and are at the same position at a given time ‘t’ i.e. either at position A or B. During the encoding of the information bits, the switches in the flushing selector block are at position B. It was mentioned earlier that the number of information bits that could be encoded at a time was equal to the interleaver length.

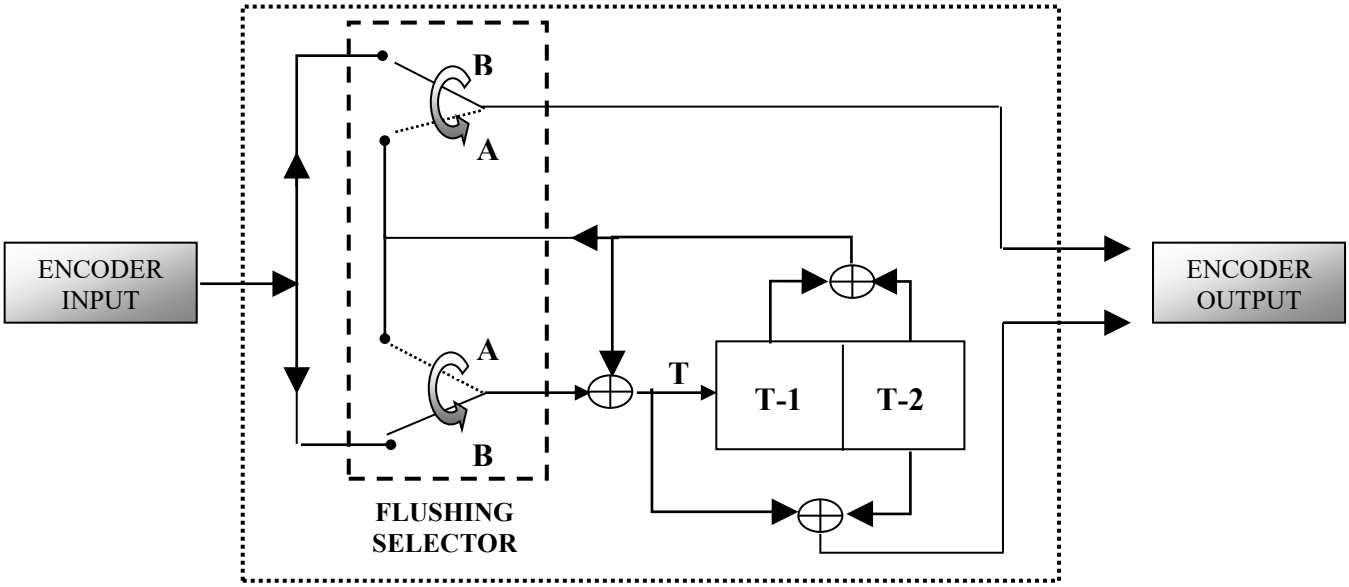


FIGURE 3.2.6- Flushing operation.

It can be followed from Figure 3.2.6 that the information bits and the corresponding encoded bits are transferred to the encoder output. When all the information bits have been encoded, the switches change position from B to A. At this point the encoder starts flushing the register state to zero by using the current shift register contents. It is known that when the switch is in position A, the input to the shift register is always zero since the same inputs to an EXOR gate give zero output. During flushing, the information bits at the encoder output is replaced by the EXOR output of the shift register content.

In order to force the states to zero, the minimum number of zero inputs to the shift register should be equal to (K-1), where K is the constraint length. Therefore, for the implemented system, inserting two zeros (i.e. clocking the shift register twice) was necessary to bring the state to zero.

3.3)- Turbo Decoder

Block diagram of the turbo decoder is given in Figure 3.3.3, which consists of two identical decoders that exploit the MAP algorithm.

The received signal from the channel is de-multiplexed and fed into the two decoders as input. The first decoder processes the interleaved information and the second does the same for the uninterleaved data. Therefore, the same interleaver structure used in the encoder was employed in the decoder as well as the 'de-interleaving' structure, which performs the inverse of interleaving. The de-interleaving process will be explained later in this chapter.

Soft output of the second decoder is fed back into the first decoder as the *a priori* information to be used in the consecutive iteration. Similarly, the soft output of the first decoder is interleaved and fed to the second decoder as input. After the last iteration, the output of the second decoder is de-interleaved and fed to a hard decision block for a final output. The final output of the second decoder is referred to as *a posteriori* information.

The structure of the received packet by the decoder is presented in Figure 3.2.7.

array index interval	**0.....(i-1)**	**i.....(3i/2)-1**	**3i/2 .....(2i-1)**	**2i.....(2i+$2^{(K-1)}$)-1**	**(2i+2^{K-1}) ..... (2i+2^{K})-1**
stored data (all float values)	information data	outputs from encoder 1	outputs from encoder 2	flushed outputs for encoder 1	flushed outputs for encoder 2

FIGURE 3.3.1- Received information packet.

Note that the received packet is identical to the encoded packet except that the stored values are all floating point due to modulation and the noise effect. The received packet is divided into two sections, one for the first decoder, the other for the second decoder. The structure of these packets is as shown in Figure 3.3.2.

array index for packet 1	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**...**	**2i-2**	**2i-1**	**2i**	**2i+1**	**2i+2**	**2i+3**
stored data (all float values)	**UI**	**P**	**UI**	**0**	**UI**	**P**	**UI**	**...**	**UI**	**0**	**FI**	**FP**	**FI**	**FP**

(a)

array index for packet 2	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**...**	**2i-2**	**2i-1**	**2i**	**2i+1**	**2i+2**	**2i+3**
stored data (all float values)	**II**	**0**	**II**	**P**	**II**	**0**	**II**	**...**	**II**	**P**	**FI**	**FP**	**FI**	**FP**

(b)

FIGURE 3.3.2- Decoder input packets. **(a)-** Packet for the first decoder.
(b)- Packet for the second decoder.

As mentioned earlier in this section, the two decoders used in the program code were identical. However, the input information to the first decoder is uninterleaved, whereas for the second one it is interleaved. Note that both packets have periodical zero insertion because of the puncturing operation performed in the encoder section. In order to keep the consistency in the length of the packets, puncturing has to be repeated in the decoder as well.

3.3.1)- De-Interleaver

De-interleaving is the inverse operation of interleaving. It is used in the decoder block for de-interleaving a priori information that is fed to the first decoder from the second one. The de-interleaving process is illustrated in Figure 3.3.4.

In the actual program, de-interleaver information is kept in an array. The elements in this array correspond to the uninterleaved index value. In other words both interleaver and de-interleaver structures are arrays that store the permuted and the original index values of a given information array, respectively.

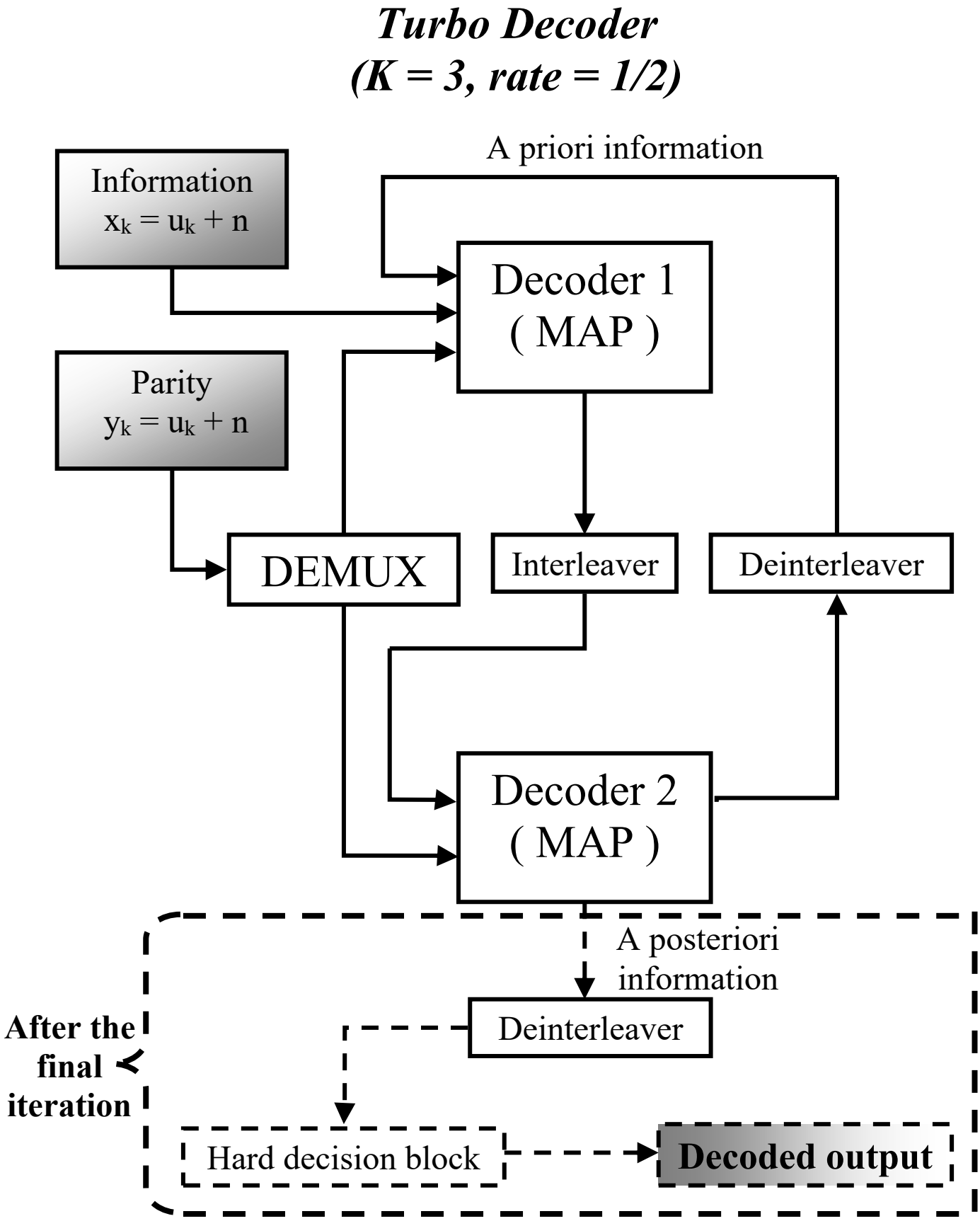


FIGURE 3.3.3- Turbo Decoder block diagram.

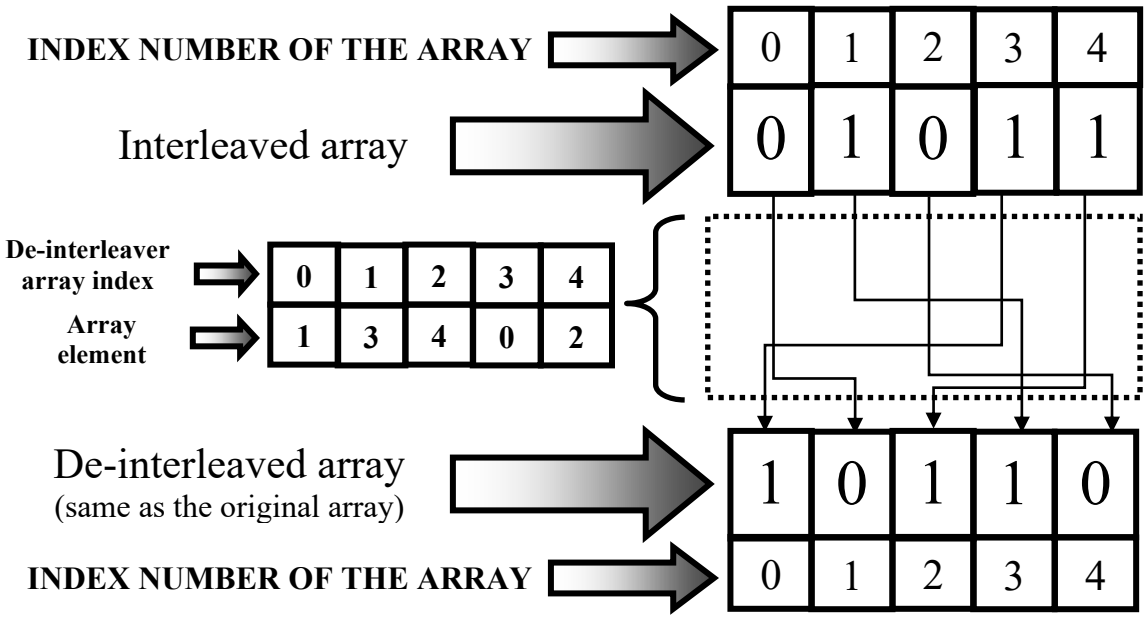


FIGURE 3.3.4- De-interleaving operation.

3.3.2)- Decoding Trellis Structure

The structure of the decoding trellis is given in Figure 3.3.5.

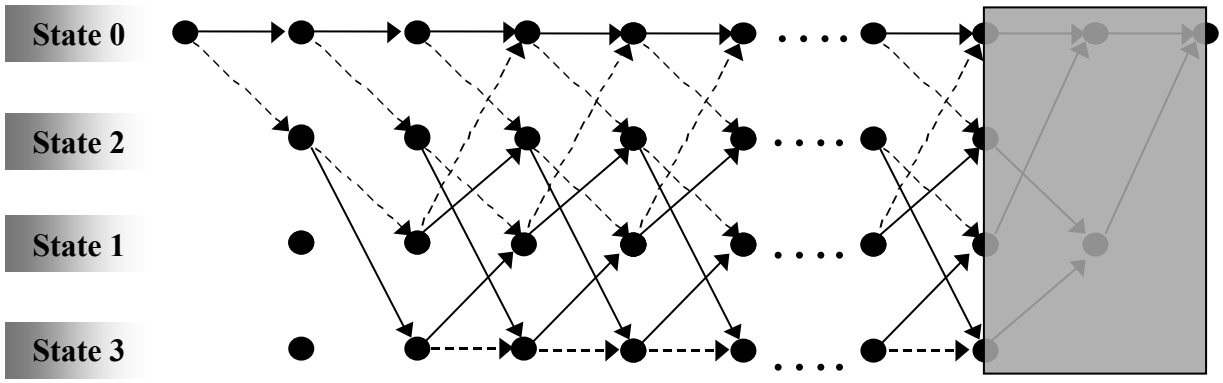


FIGURE 3.3.5- Decoding trellis structure.

Dashed and solid lines show input 1 and 0 to the encoder, respectively. The starting state is 0. The order of the states at each depth is 0-2-1-3, from top to bottom. The branch code words are presented in Figure 3.3.6, where '>>>' shows the direction of the transition.

Trellis State Transition	Branch Code Word (in binary format)
0 >>> 0	00
0 >>> 2	11
2 >>> 3	01
2 >>> 1	10
1 >>> 2	00
1 >>> 0	11
3 >>> 1	01
3 >>> 3	10

FIGURE 3.3.6- State transition table.

The depth of the trellis and the number of states change with the interleaver size, and the constraint length (K), respectively. Number of states in the decoding trellis (N) can be calculated from (3.2).

$$N = 2^{(K-1)} \qquad (3.2)$$

It is important that the encoder is brought to zero state after each coded information block. Since the structure of the encoder is IIR, the shift registers have to be 'forced' to zero state instead of applying a zero input to the encoder, as explained earlier in this chapter. The highlighted section of the trellis (at the end) corresponds to the flushing operation. Note that all the branches correspond to input zero since during flushing, encoded information bit is always zero. Flushing operation has been explained in detail in the encoder section.

3.3.3)- Log-Map Decoder Block

Turbo decoder block in the program code calls another function block, which does the actual MAP decoding. This block is the log-MAP decoder block. It is used for both the first and the second decoder, which are connected in series. The only application difference between the two operations is the received array input to the decoders.

Log-MAP decoder block is the most complicated function block in the entire program since the decoding algorithm is carried out including the initialization and the calculation of alpha, beta and gamma as well as the extrinsic gamma values. It should be kept in mind that all values processed in this sub-function are logarithmic values. Using logarithmic values enables the use of formula approximation as mentioned in [1] and reduced numerical complexity in execution of the code.

The end section of the log-MAP decoder calculates the soft output for the decoder of interest (either of the first or the second), and returns these values to the decoder function for feedback. The reader is referred to Figure 3.3.3 for a better understanding of the log-MAP decoder block. This block has been the focus of attention throughout the project both for error performance and complexity analysis. The results obtained are presented in Chapter 4.

3.4)- Other Blocks Involved

Besides the blocks explained up to this point, there are other sub-blocks that were used in the program code. These have been categorized in three main groups, which will be discussed in the next section.

3.4.1)- Initialization Blocks

Three main initialization blocks have been implemented in the source code.

3.4.1.a)- Reading Noise Values

A file named ‘ch_noise.dat’ was used for storing 1000 random noise values that have been computed for zero mean and a variance of 1. Different noise levels are calculated by using the pre-stored values in this file. A block for storing these noise values in an array was implemented in the program code.

3.4.1.b)- Interleaver Initialization

Recall that interleaving and de-interleaving operations are the inverse operations of each other. These two operations have been implemented by using two different arrays: one for interleaving and one for de-interleaving. In the program, random numbers are generated between zero and the interleaver length which correspond to the indices of an information array. These indices are stored in the first array, called ‘perm’, for interleaving. While ‘perm’ is being constructed, its elements are mapped to a second array, ‘d_perm’, as indices and the corresponding indices of ‘perm’ are mapped to ‘d_perm’ as elements.

This procedure is carried out in the interleaver initialization section and can be further studied from the modified source code presented in Appendix B.

3.4.1.c)- Decoder Initialization

The preparation of the received information packets for decoding that was mentioned in the decoder section is carried out in a sub-block named decoder initialization. In addition to this, the initialization of the $L(u_k)$ (a priori values) to zero is performed as well.

3.4.2)- Channel

Channel block is used for the addition of white Gaussian noise. The necessary Gaussian variables used for each transmitted symbol are stored in a look-up table that consists of 1000 numbers. This table is a file named ‘ *ch_noise.dat* ‘, which was constructed for Gaussian variables with zero mean and unit variance [8].

SNR value is calculated from (3.3)

$$\frac{E_b}{N_o} = 10 \cdot \log\left[\frac{1/R}{2 \cdot \sigma^2}\right] \qquad \textbf{(3.3)}$$

where R is the code rate and σ^2 is the variance. As mentioned earlier, the code rate used in the system was ½. After substituting for R in (3.3), (3.4) is obtained

$$\frac{E_b}{N_o} = -20 \cdot \log \sigma \qquad \textbf{(3.4)}$$

The last expression shows that the SNR calculation in the system depends only on the standard deviation (σ) of the Gaussian noise. Since the error performance of the system was evaluated under different SNR values, required standard deviation values were calculated from (3.5), which is the inverse of (3.4).

$$\sigma = 10^{-\left(\frac{E_b}{20N_o}\right)} \qquad \textbf{(3.5)}$$

The simulated SNR values and the calculated standard deviations are presented in Figure 3.4.1. These standard deviation values are kept in an array in the program source. For a given SNR value the relevant σ is picked up and used for the generation of noise. In order to make the operation of the channel block clearer Figure 3.4.2 has been constructed.

As it can be seen from the figure, noise level can be changed by using different σ values. Note that the multiplication operation can be applied directly with σ, since the look-up table has been constructed for zero mean, unit variance Gaussian variables.

E_b/N_o RATIO (dB)	STANDARD DEVIATION (σ)
1.0	0.890
1.2	0.870
1.4	0.851
1.6	0.832
1.8	0.813
2.0	0.790

FIGURE 3.4.1- E_b/N_o and corresponding σ table.

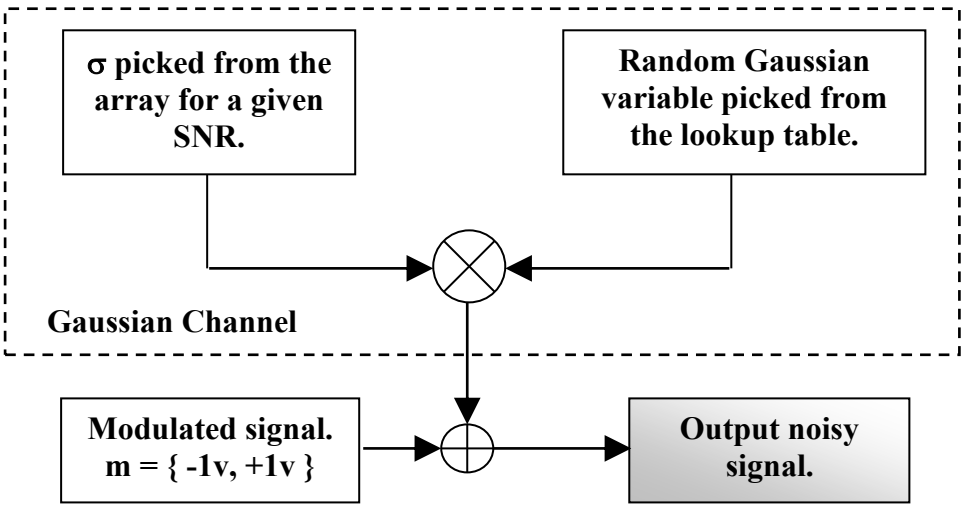


FIGURE 3.4.2- Noise addition in the modified program code.

3.4.3)- Data Sink

The program code is designed to print a report of each simulation result, both to a file and the screen, for evaluating the BER and speed performance of the source code.

This block has been updated frequently throughout the project, depending on the required results. Program can keep a track of statistical data such as the number of ones and zeros transmitted, and the number of errors before and after decoding for a given SNR, interleaver length, and the number of iterations in each simulation.

3.5)- Modification of the Technique

The focus of the project was to design a modified code of lower complexity compared to the conventional code.

The degree of complexity was measured in terms of processing time. These measurements were performed for the encoder, decoder and for the whole program. Decoder block was the focus of interest for complexity reduction, as most of the computations were carried out in this section. The Log-MAP decoder block in the decoder block was examined for possible improvements. Initial improvements in the processing time were done through the reductions in the number of parameters passed to sub-functions from the main function, and by passing pointers instead of long arrays to sub-functions in the program code. Additionally, all the loop operations in the conventional code were reviewed for programming simplifications, which would not influence the error performance of the codec at all. Trellis structure used in the modified program made it possible to use such short cuts. Many parallel operations were carried out within the same loop. The basic method followed was reducing the number of loops and to do *as many operations as possible in the minimum number of loops*.

Another technique used for decreasing the processing time was the use of variables instead of matrix elements. In other words, frequently accessed matrix elements were assigned to variables

before they were called by a function. This operation provided further reduction in computation time.

The programming techniques used allowed a considerable amount of complexity reduction due to the iterative decoding structure used in the decoder. For each iteration, log-MAP block is called twice in the program, which has most of the share in the overall computation time. As the number of iterations increase, the time consumption of this block becomes more significant.

Major complexity reductions were accomplished by the use of logic operations, which replaced matrix use, which will be explained in detail later in the text.

In the conventional program, four matrices of dimensions 4 by 2 are used in order to obtain the branch words and the state transitions for alpha and beta calculations in the decoder. The modified program uses two matrices of the same dimensions to do the same computations with the aid of logic operations, which are faster compared to accessing a matrix element stored in memory.

Assume that the beta value for a given state ‘s’’ at a given depth ‘j’ is to be calculated as shown in Figure 3.5.1. The calculation of interest has been indicated on the trellis with dark solid lines, and also the relevant gamma and branch words have been labeled.

Beta calculation at depth ‘j’, for the illustration in Figure 3.5.1 is done as expressed in (3.6), general form of which was covered in Chapter 1, MAP algorithm.

$$\beta_j(0) = \beta_{j+1}(0) \cdot \gamma_j(0,0) + \beta_{j+1}(2) \cdot \gamma_j(0,2) \qquad \textbf{(3.6)}$$

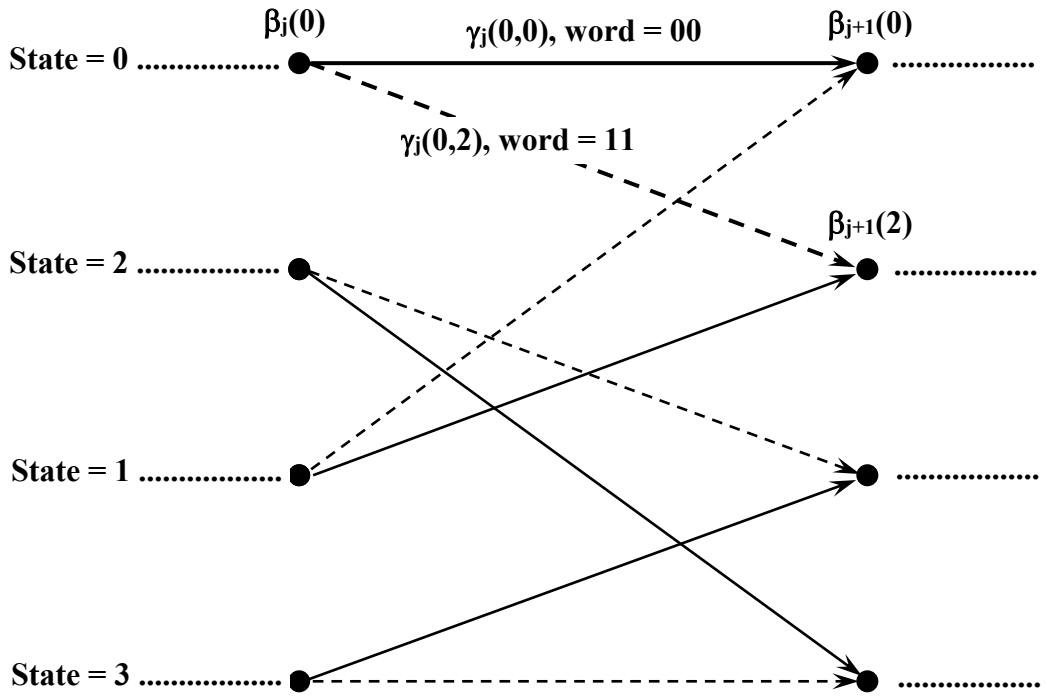


FIGURE 3.5.1- Illustration of beta calculation.

It can be seen that for one beta calculation at a given depth, four different terms are needed; beta values of the states at depth 'j+1' for the relevant transitions from depth 'j', and gamma values of these transitions. Consequently, two pieces of information are needed; from which *future states* to *that beta* state transitions are available, and what the *branch code words* for these transitions are. It is obvious that the future states depend on the branch information bit, either one or zero, at state s'. In the conventional program code, future states for a given previous one in beta calculation, are stored in a 4 by 2 matrix called 'forward', where the rows and columns correspond to the current state and the branch information bit, respectively.

The gamma values are calculated for four different code words; 00, 01, 10, and 11 and are stored in a matrix for all the depths in the decoding trellis. The relevant branch code word is picked from another 4 by 2 matrix defined in a sub-function, namely 'enc', for a given current state and branch

information bit. It should be noted that alpha values are calculated in in a similar fashion, but by using a matrix called 'back' in stead of 'forward'. Also the sub-function 'enc' returns different branch code word in the case of alpha calculation.

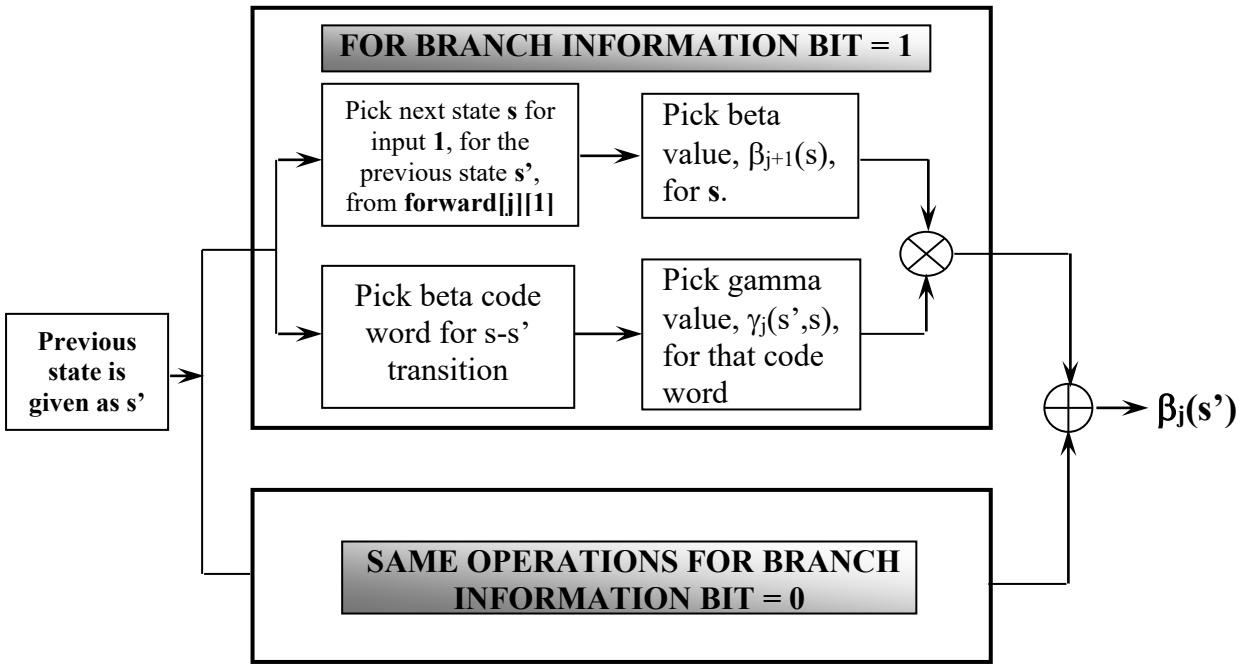


FIGURE 3.5.2- Beta calculation in the conventional program.

Figure 3.5.2 shows the beta calculation for a single state in the conventional program code.

As it was found out, the matrix access in the program code was time consuming, and a proper substitute for this method could provide significant time gain.

The modified program exploits logic operations for the same computation. It only uses two of 4 by 2 matrices to access the branch words for alpha and beta calculations. Future state is calculated by bit manipulation for a given previous state 's''. This has been illustrated in Figure 3.5.3.

In order to obtain the future state and the code word for a given s', two matrices were used in the modified program. An interesting property of these matrices enabled the aforementioned logic manipulation, as indicated in Figure 3.5.4.

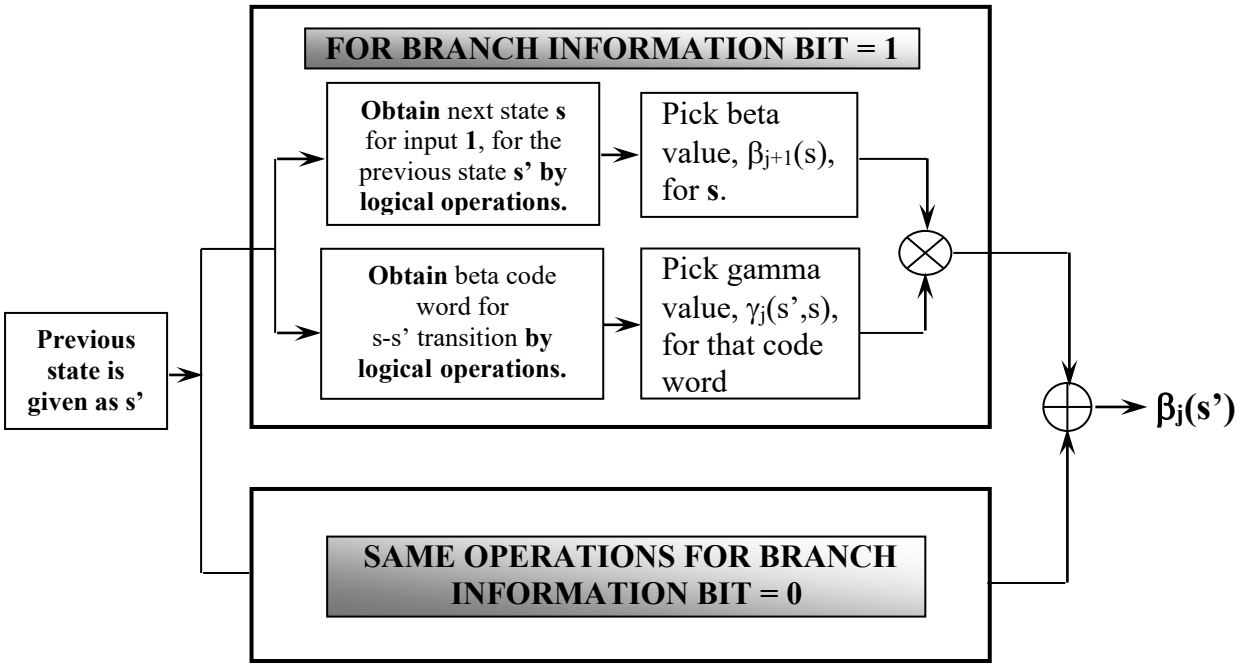


FIGURE 3.5.3- Beta calculation in the modified program.

alpha words =

	0	1
0	0	3
1	1	2
2	0	3
3	1	2

beta words =

	0	1
0	0	3
1	0	3
2	1	2
3	1	2

FIGURE 3.5.4- Matrices used in the modified program.

Recall that the row and column indices correspond to the previous states (s') and branch information bits, respectively. As indicated in Figure 3.5.4, for alpha matrix the code words for s'=0 and 2, are the same for the same inputs. This is the same for s'=1 and 2. A similar relationship exists for the beta matrix, but this time for s'= 0 and 1, and s'= 2 and 3. This is the key property

used in order to obtain the code words for a given s' and branch input value. The technique has been illustrated for obtaining the code words for accessing the relevant gamma value used in alpha calculation, in Figure 3.5.5.

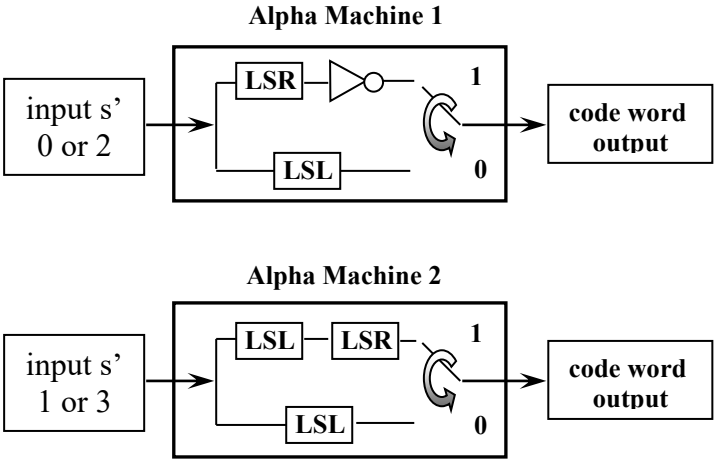


FIGURE 3.5.5- Alpha code word model in the modified program.

The system has been modeled in terms of two code word machines, namely Alpha Machine 1 and 2. Each machine takes the previous states, s', as input. Then relevant logical operations are applied to s' to obtain the code words. Note that the machines carry out different operations depending on the branch information bit. This is determined by the position of the switches, being either one or zero as shown in Figure 3.5.5.

Same machine model exists for the beta code words as well. Figure 3.5.6 shows the machine structures for beta code words. It should be noted that the code word outputs are used to pick the gamma values that are pre-calculated for all the depths of the decoding trellis.

Similar algorithm is used for obtaining the future states (s) given the previous state, s' for a given branch information bit. The logical operations will not be explained, as it is similar to the code word computation.

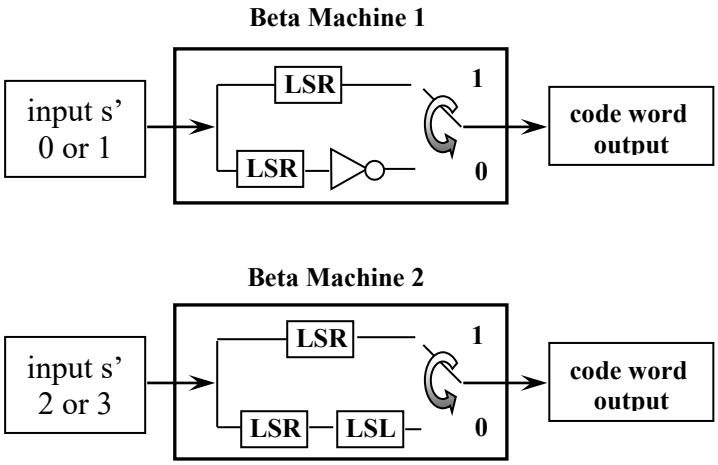


FIGURE 3.5.6- Beta code word model in the modified program.

While working on the modified code, special attention was paid to the modifications in order to stick to original model of the turbo codec in general. In other words, all the operations were performed in the relevant blocks. This way the simulation results have been kept closest to a real-life model. The results regarding the complexity reduction have been presented in the following chapter.

CHAPTER 4 - RESULTS

4.1)- Error Performance

Error performance of the conventional and the modified source code has been studied in terms of E_b/N_o vs. BER plots for variable interleaver length and number of iterations. Simulations for this section have been carried out between the E_b/N_o range of 1 dB and 2 dB incremented by 0.2 dB. BER for each simulation has been calculated by taking the ratio of the bits in error after decoding to the total number of transmitted information bits.

4.1.1)- Interleaver Length Effect

The simulation parameters have been presented in Figure 4.1.1.

Number of information bits transmitted.	10 000 000
Number of iterations.	4
SNR values in dB.	1, 1.2, 1.4, 1.6, 1.8, 2.0
Interleaver lengths simulated.	100, 200, 400, 500, 800, 1000, 2000, 2500, 4000, 5000, 10000

FIGURE 4.1.1- Simulation parameter table.

Figures 4.1.2, and 4.1.3 show the E_b/N_o v.s BER change for the modified and the conventional code, respectively. It can be seen that the BER performance at a given SNR improves with the increasing interleaver length in both cases. This is due to the fact that the statistical independence between the information bits fed into the two decoders increases with the increasing interleaver length. In this case the feedback information input to the decoders provides significant amount of information to increase the confidence level of soft decision with each iteration. Note the closeness of the results obtained for both codes, which shows the degree of the error performance similarity.

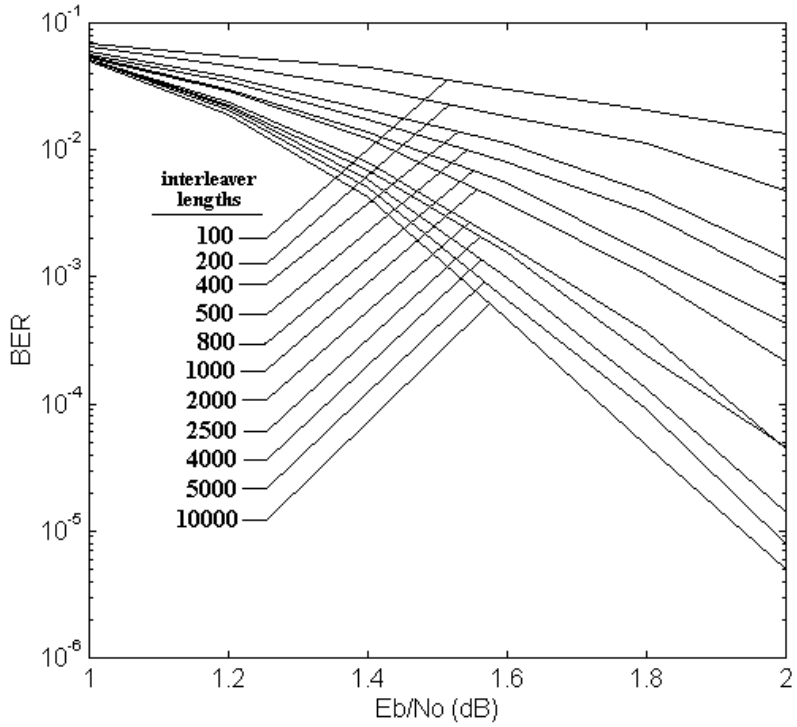


FIGURE 4.1.2- E_b/N_o vs. BER graph for the modified code.

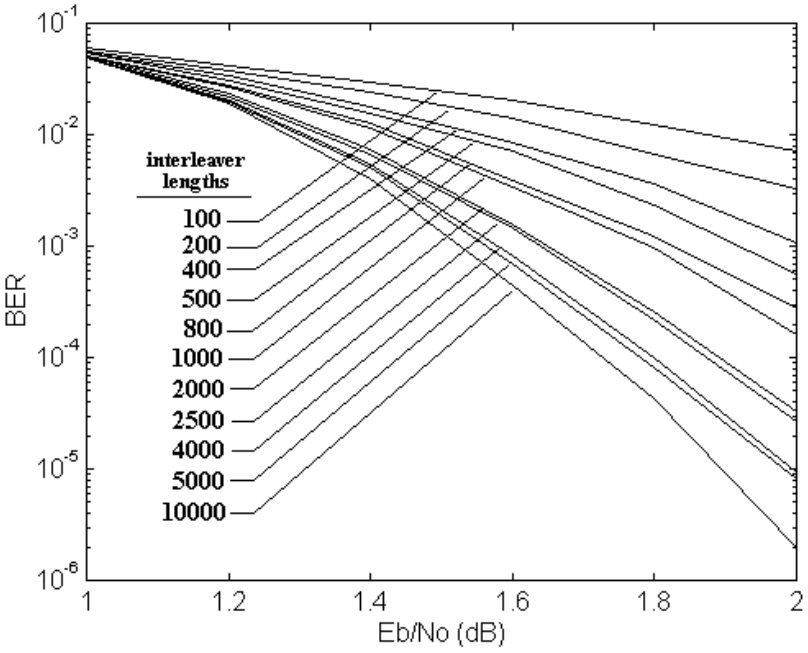


FIGURE 4.1.3- E_b/N_o vs. BER graph for the conventional code.

In order to see the effect of increasing interleaver length on the BER performance, Figures 4.1.4 and 4.1.5 were obtained. They both show this change for different E_b/N_o values, which were varied according to the table in Figure 4.1.1 presented earlier.

Considering the modified code, even though Figure 4.1.2 suggests that as the BER decreases with the interleaver length, one suspects that there should be a limit to this increase. Figure 4.1.4 indicates that for all SNR values the BER improvement with the increasing interleaver length becomes trivial after 5000. For low SNR values (about 1 dB), greater interleaver lengths provide less improvement in the error performance, whereas the same improvement for higher SNR levels is more significant. Even though greater interleaver lengths provide a better statistical independence in decoding algorithm, this is not enough to increase the error performance of the system on its own. When noise level is high, the data corruption level is high as well. Therefore, estimation of the actual transmitted data cannot approach the correct value, despite the high degree of statistical

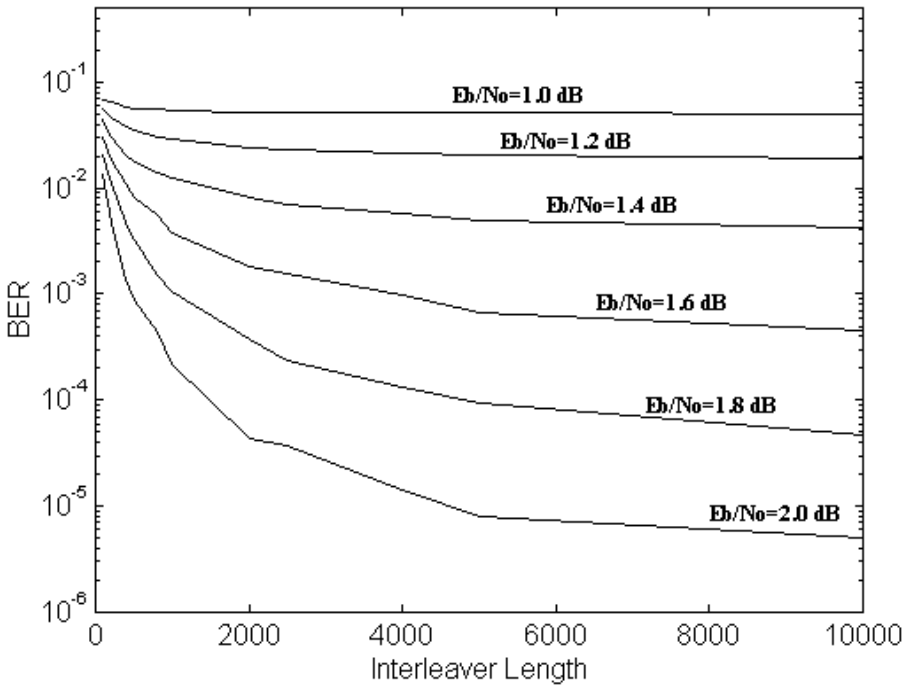


FIGURE 4.1.4- Interleaver length vs. BER graph for the modified code.

independence.

Although larger interleaver lengths seem to be an attractive solution for high SNRs (2 dB and higher), lower SNRs (as close to the Shannon limit as possible) are of more interest in communication systems where large interleaver lengths provide no significant gain.

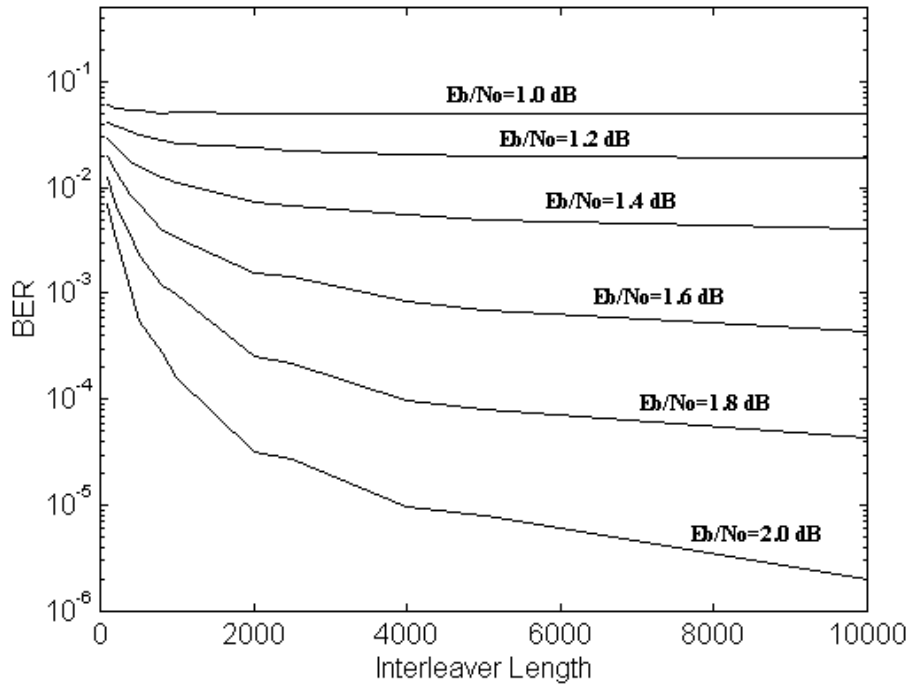


FIGURE 4.1.5- Interleaver length vs. BER graph for the conventional code.

The similarity between Figure 4.1.4 and 4.1.5, should be noted here as it provides a good means of comparison between the two program codes in terms of error performance.

4.1.2)- Iteration Effect

The simulation parameters used for both program codes have been presented in Figure 4.1.6. Figure 4.1.7 shows the effect of the number of iterations on the BER simulated for the specified SNR values for both program codes. Same axes have been used for both plots in order to provide easy comparison.

Number of information bits transmitted.	10 000 000
Number of iterations simulated.	2,4,6,8,10,12
SNR values in dB.	1, 1.2, 1.4, 1.6, 1.8, 2.0
Interleaver length.	1000

FIGURE 4.1.6- Simulation parameter table.

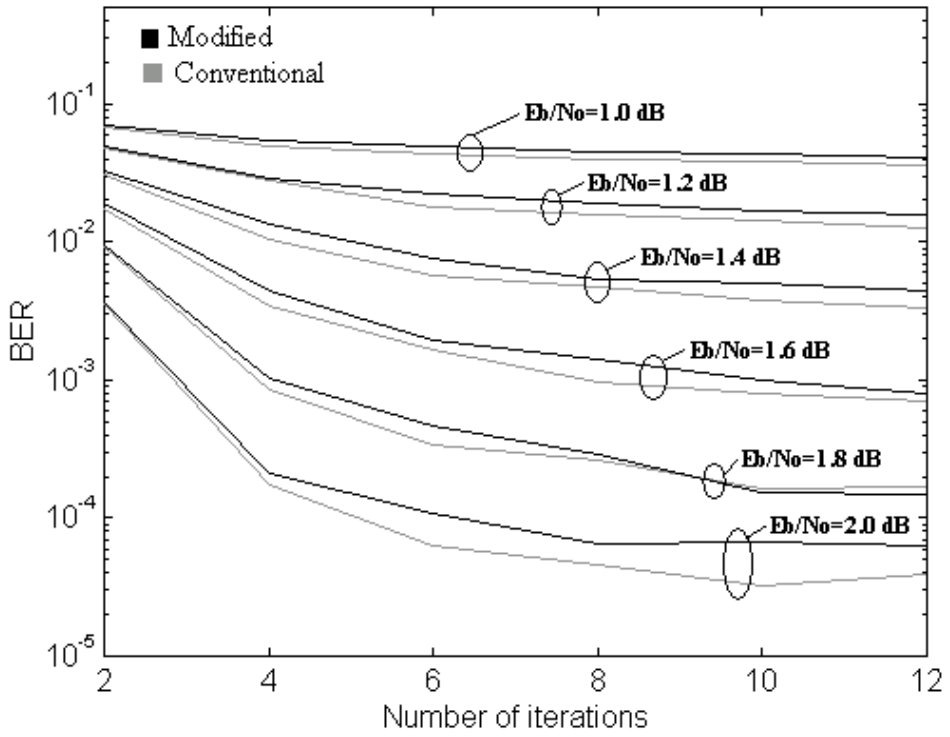


FIGURE 4.1.7- Number of iterations vs. BER graphs.

As it can be observed from Figure 4.1.7, the BER performances of both programs are close to each other. The difference in the two performance plots arises from the randomness of the noise. For each simulation the performance varies slightly which is expected due to the varying effect of noise on each simulation. In other words, the error performance of the two systems can be assumed to be the same.

In order to give an idea of the closeness of the BER performance for different number of iterations, Figure 4.1.8 has been provided. It shows the BER values obtained in the simulation that was carried out for 4 iterations at different E_b/N_o values.

E_b/N_o level (dB).	Conventional program's BER.	Modified program's BER.
1.0	0.049902301	0.054540601
1.2	0.027840201	0.029210901
1.4	0.010533900	0.013394000
1.6	0.003396700	0.004453400
1.8	0.000859200	0.001016400
2.0	0.000171400	0.000212000

FIGURE 4.1.8- BER comparison table for 4 iterations.

4.1.3)- Visualization of the Error Performance

In order to provide a better basis of comparison, various images were used as input to the turbo codec. SNR value was kept constant at 1.6 dB and the interleaver length was chosen as 1000. Numbers of iterations tested were 2, 4 and 12. The results of these simulations have been presented in Figure 4.1.9 including the BER for each iteration. Image displayed in Figure 4.1.9 is a 256 colors gray scale image of total of 2000000 bits.

More image results have been presented in 256 colors for 1,2,4,8, 10 and 16 iterations and 1.8 dB SNR values [Appendix C].

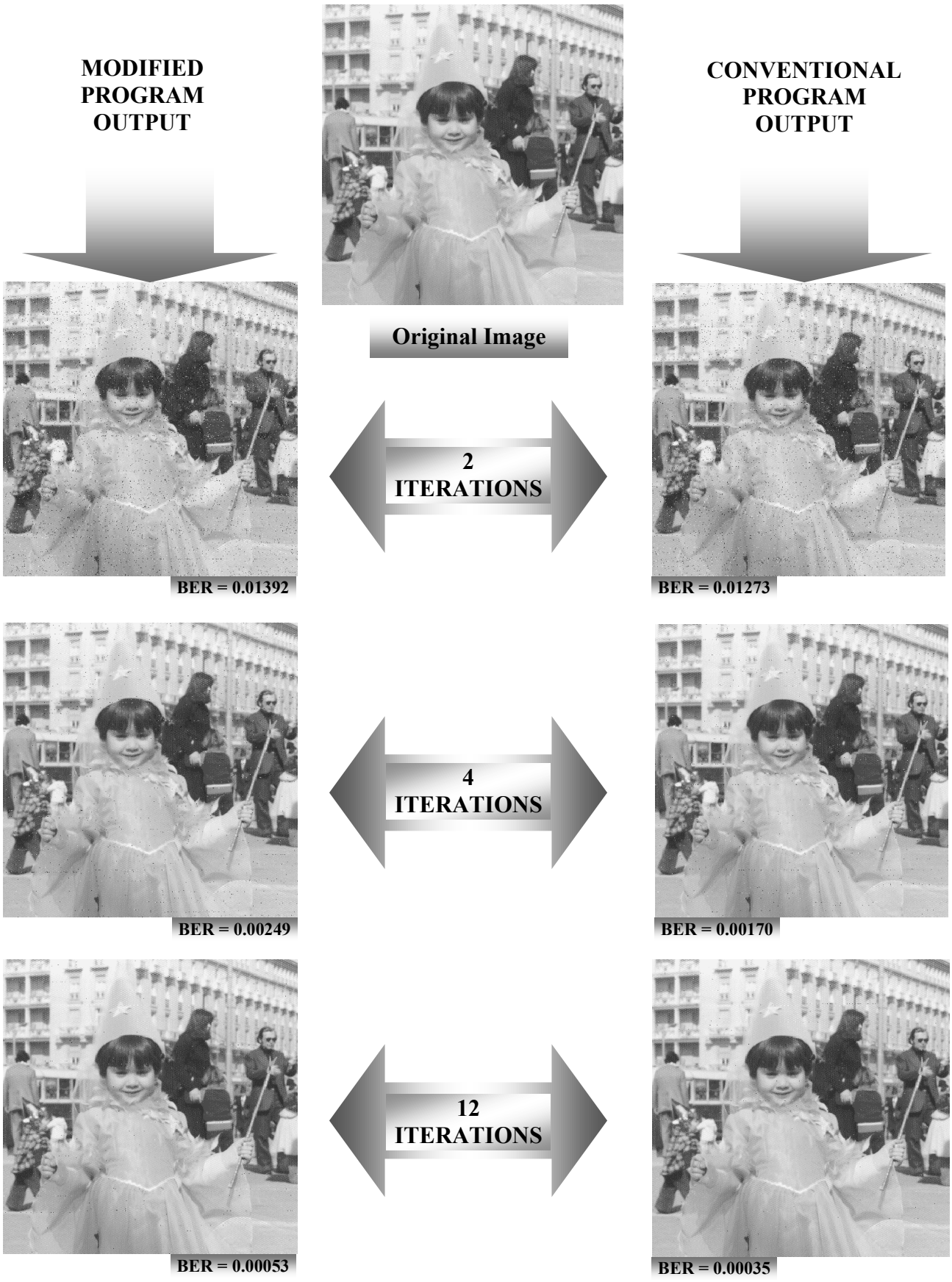


FIGURE 4.1.9- Results for image transmission at E_b/N_o =1.6 dB.

4.2)- Complexity Comparison

The simulation parameters have been presented in Figure 4.2.1.

Number of information bits transmitted.	2 000 000
Number of iterations simulated.	2,4,6,8,10,12
SNR values in dB.	2.0
Interleaver length.	1000

FIGURE 4.2.1- Simulation parameter table.

These parameters have been used for both the conventional and the modified program simulations unaltered, to provide a relevant comparison basis. The complexity comparison i.e. the speed performance of the conventional and the modified program has been presented together.

4.2.1)- Encoder Complexity

Figure 4.2.2 shows the number of iterations vs. encoding time of the two program codes.

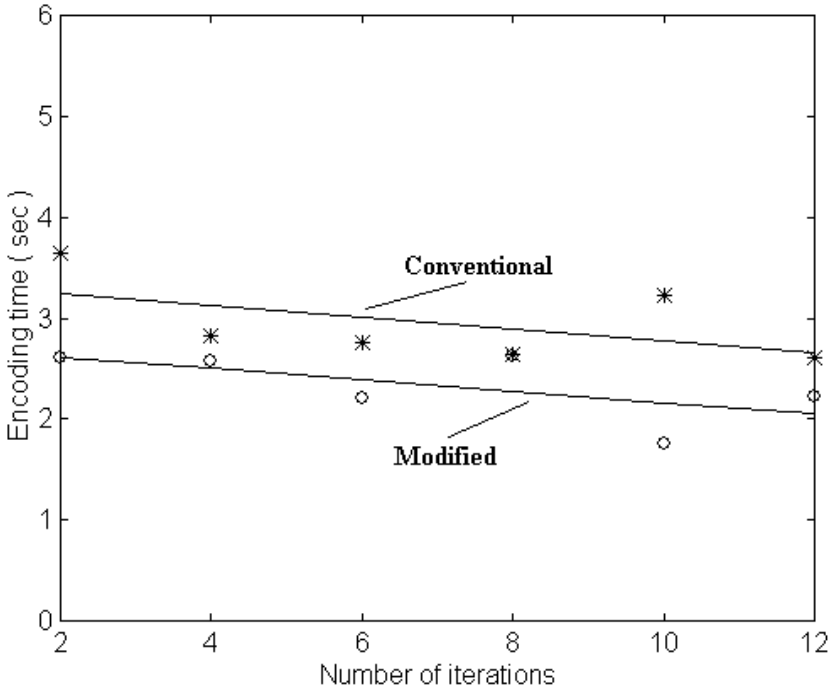


FIGURE 4.2.2- Number of iterations vs. encoding time in seconds.

The asterisks and circles indicate the actual data points for the conventional and the modified program codes, respectively. Two straight lines are the *least squares lines* [Appendix A] drawn for the two data sets. As indicated, the modified program's encoder block works faster than the conventional's. It is important to note that the negative slope of the two lines suggest that, as the number of iterations increase, the encoders work faster. However, encoder speed block is *independent of the number of iterations* as this parameter is related to the decoder block. Therefore, encoder speed was expected to be constant for variable number of iterations. Making more measurements by running more simulations can prove the validity of this argument. Yet, from Figure 4.2.2 the distribution of the actual data points suggests a constant encoder speed since lines are a close approximation of the real model.

4.2.2)- Decoder Complexity

Figure 4.2.3 shows the number of iterations vs. decoding time in seconds, where the conventional and the modified program plots have been indicated.

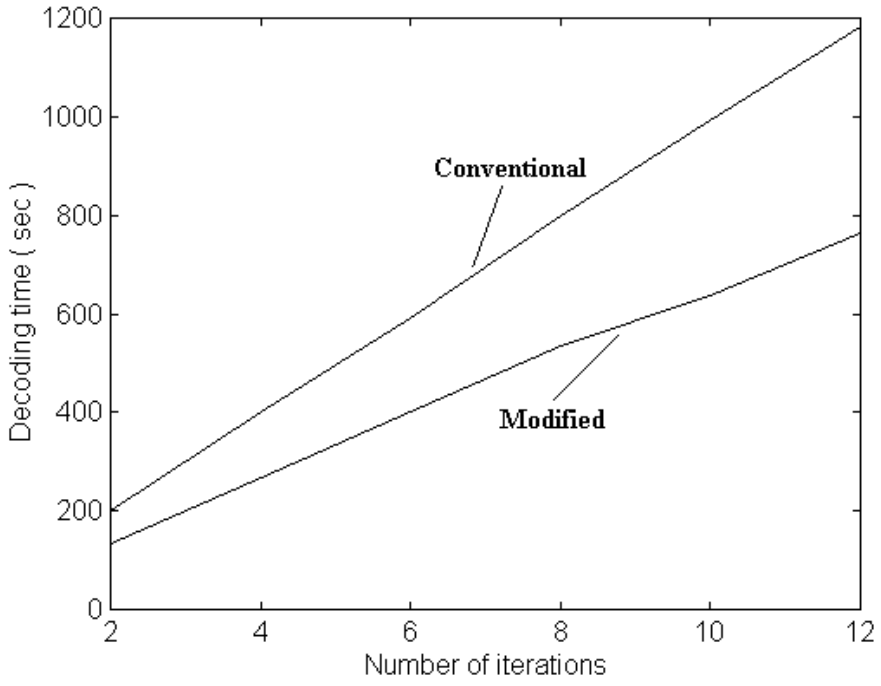


FIGURE 4.2.3- Number of iterations vs. decoding time in seconds.

Decoding speed difference between the two program codes can be seen clearly in Figure 4.2.3. Slopes of both lines are positive, showing the increase in decoding time with the increasing number of iterations.

Another significant result is that the speed difference between the two decoders increases as the number of iterations increase. To illustrate this point, the Figure 4.2.4 is constructed, which shows the bit rate comparison of the two decoders in terms of decoded bits per second, and their difference for different number of iterations.

NUMBER OF ITERATIONS	MODIFIED DECODER BIT RATE (bits/sec.)	CONVENTIONAL DECODER BIT RATE (bits/sec.)	GAIN OF THE MODIFIED DECODER
2	14813	9960	48.7%
4	7481	4993	49.8%
6	4980	3378	47.4%
8	3751	2508	49.6%
10	3146	2015	56.1%
12	2619	1692	54.8%

FIGURE 4.2.4- Bit rate comparison of the decoders.

In Figure 4.2.4, two results have been highlighted, in order to show the significance of the speed difference. According to this, the modified decoder can implement 12 iterations while the conventional decoder implements 8 iterations, and it can still work faster than the conventional one. In other words, the modified decoder can achieve better error performance at about 4% higher speed compared to the convolutional system, considering this example.

4.2.3)- Overall Complexity

Figure 4.2.5 shows the number of iterations vs. overall operation time for a single simulation of the modified and the conventional program codes. This timing information includes the encoder, decoder, and all the other sub-blocks involved in the encoding and decoding process. Overall time plot has the characteristic of the decoder time graph, as the decoder's share in the overall timing is the biggest compared to all the other blocks in the program, which is valid for both decoders. In order to illustrate this point, the overall time distribution for the modified decoder has been presented in Figure 4.2.6.

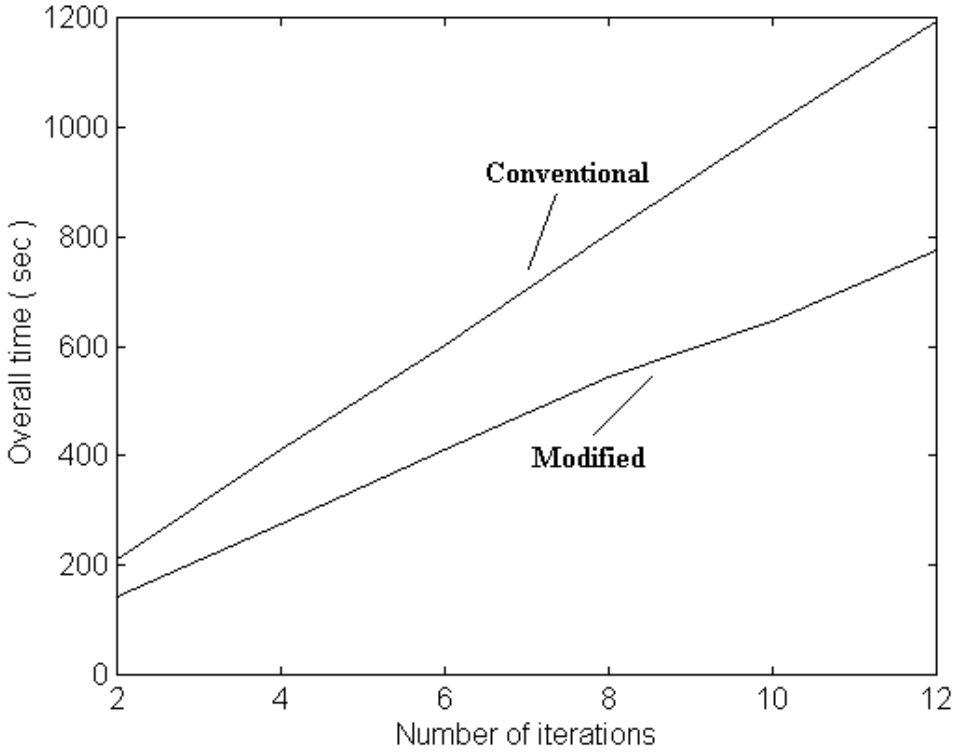


FIGURE 4.2.5- Number of iterations vs. overall time in seconds.

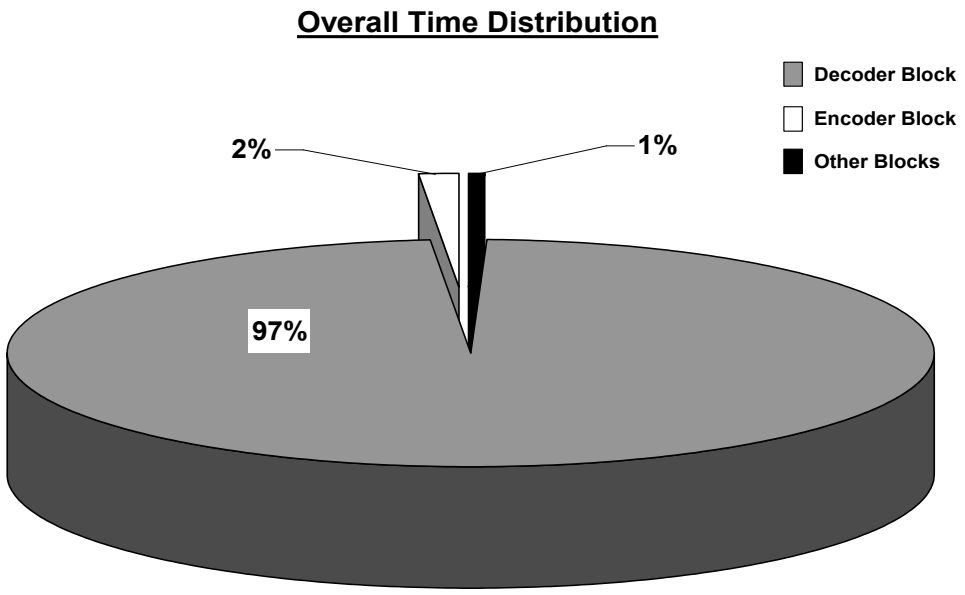


FIGURE 4.2.6- Overall time distribution for the modified program code.

Note that Figure 4.2.6 shows the average of all the simulations of the modified program. As indicated, decoder time takes up 97%, and the encoder and the other blocks occupy 3% of the overall processing time required for one simulation. Figure 4.2.6 also shows the significance of concentrating the research on the decoder block for further complexity reductions.

Figure 4.2.5, as mentioned earlier, was a comparative illustration of the modified and the conventional programs' performance. In order to provide a more comprehensible comparison between the two program codes, Figure 4.2.7 has been presented. The modified program's speed gain over the conventional one has been expressed in terms of percentage for each simulation on each bar.

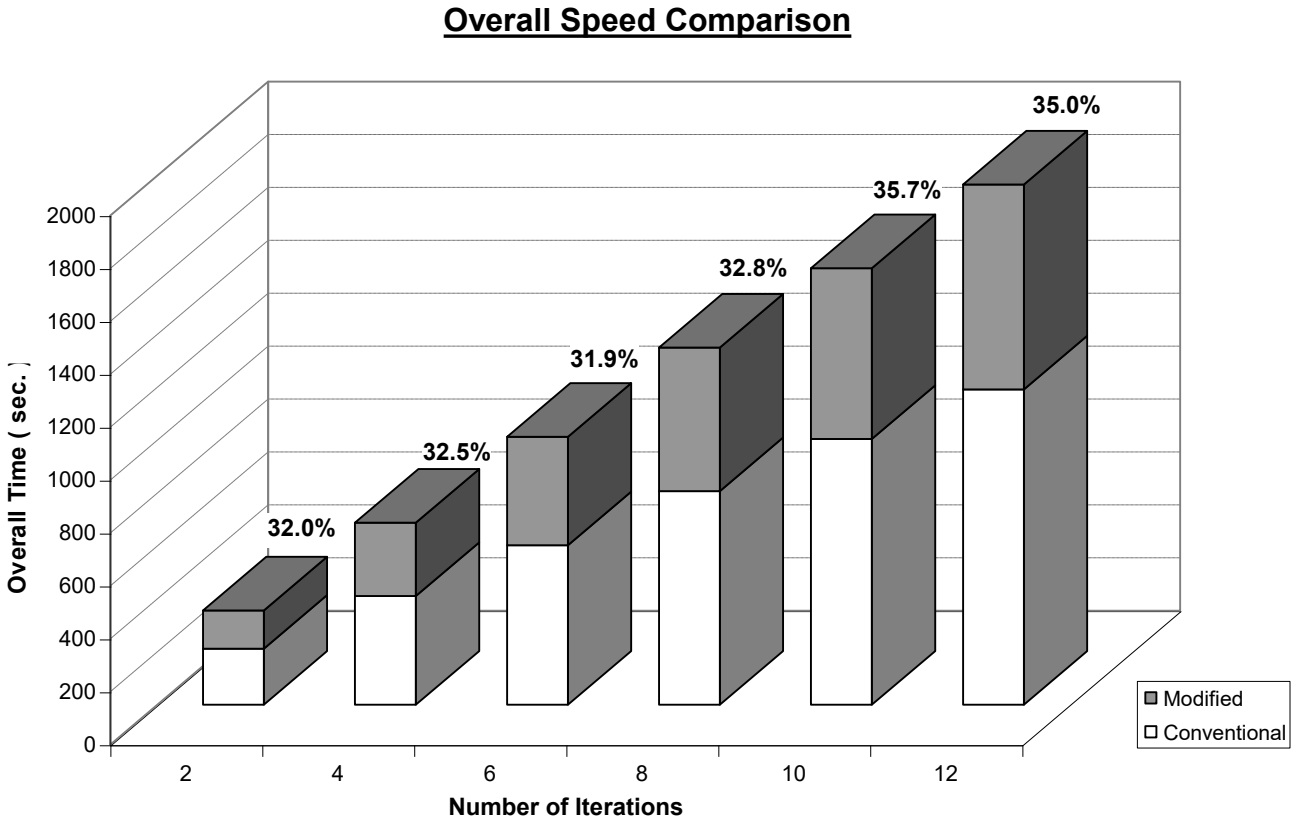


FIGURE 4.2.7- Number of iterations vs. overall time bar graph.

CHAPTER 5- CONCLUSION & FURTHER WORK

This project was aimed at establishing a detailed understanding of turbo codes and the MAP decoding algorithm through an application and theoretical study. Having optimized the source code, analysis of BER performance, effects of interleaver length and number of iterations were studied and observed through simulation. These results were compared to the previously designed code.

Although error performance is an important aspect for error correcting codes, reduced complexity for such codes become significant for real-time applications. This was the motivation for designing a computationally efficient modified program code with a low degree of complexity, which was the primary task. Time gains up to 35% could be achieved by the application of simplified programming techniques based on low-level programming algorithms with no degradation in the error performance.

This work can be extended by the implementation of the modified source code in a different software environment, namely COSSAP, which would provide a more detailed analysis of the current system. It will also provide the opportunity of implementing different combinations of error correcting code schemes and the turbo codes.

Even though performance of MAP algorithm is known to be better than SOVA, a comparative analysis of a SOVA system of lower complexity could provide useful analysis, in terms of implementation. Additionally, convolutional component code use in this project can be replaced by a Partial Unit Memory (PUM) code to provide a comparative performance analysis.

Trellis Coded Modulation (TCM), which combines a multilevel/phase modulation signaling set with a trellis scheme, can be combined with the powerful error correcting capability of turbo codes, and this would provide a further opportunity to extend the research.

Finally, in order to test the real-time performance of a turbo coding system, DSP implementation of the current work could be performed. As the implementation of code in assembly language platform is quite time consuming and tedious, it would be advantageous to use a DSP system with a C programming interface for such a task.

REFERENCES

[1] J. Hagenauer, Fellow, IEEE, E. Offer, and L. Papke, '*Iterative decoding of Binary Block and Convolutional Codes*', IEEE Transactions on Information Theory, Vol. 42, No. 2, March 1996.

[2] B. Sklar, Communications Engineering Services, '*A Primer on Turbo Code Concepts*', IEEE Communications Magazine, December 1997.

[3] C. Berrou, A. Glavieux, and P. Thaitimajshima, '*Near Shannon Limit Error Correcting Coding and Decoding: Turbo Codes*' IEEE Proc. ICC '93, Geneva, Switzerland, May 1993, pp. 1064-1070.

[4] C. Berrou, and A. Glavieux, '*Near Optimum Limit Error Correcting Coding and Decoding: Turbo Codes*' IEEE Trans. Commun., vol. 44, no 10, Oct. 1996, pp. 1261-71.

[5] B.Honary, B.Thomas, P. Coulton, M. Darnell(97), '*Novel Application of Turbo Decoding for Radio Channels*', Proc. of Sixth IMA International Conference on Cryptography and Coding, Cirencester Dec. 1997 (pp 119-130).

[6] H. Schildt, '*C++ from the Ground Up*', Second Edition, Mc Graw Hill, U.S.A, 1998.

[7] E. A. Blaisdell, '*Statistics in Practice*', Saunders College Publishing, U.S.A., 1992.

[8] A. D. Whalen, '*Detection of Signals in Noise*', Academic Press, 1971, New York.

[9] B. Sklar, '*Digital Communications*', Prentice Hall, USA, 1988.

[10] S. Haykin, '*Communication Systems*', third edition, John Wiley & Sons Inc., Canada, 1994.

[11] V. Cappellini, '*Data Compression and Error Control Techniques with Applications*', Academic Press, 1985.

[12] J. Hagenauer, P. Hoeher, '*A Viterbi Algorithm with Soft-Decision Outputs and its Applications*', IEEE Proc.

[13] G. D. Forney, JR, '*The Viterbi Algorithm*', IEEE Proc., Vol. 61, NO. 3, March 1973.

[14] G. C. Clark, Jr., J.B. Cain, '*Error-Correction Coding for Digital Communications*', Plenum Press, 1981.

[15] http://www331.jpl.nasa.gov, Communications Systems and Research Section, JPL, Information Processing Group Home Page.

[16] http://www.ee.virginia.edu, University of Virginia Turbo Codes Home Page.

[17] http://charli.levels.unisa.edu.au, University of South Australia Turbo Codes Home Page.

APPENDIX A- The Least Squares Line

After collecting a set of data for variables x and y, next step would be to build a scatter diagram to determine the relationship that might exist between x and y. If a linear relationship is observed, next challenge would be to find out which line would be used to approximate this relationship.

Statisticians usually define the 'best' choice to be the *least squares line* [7].

Before giving the equation of the least squares line, it would be appropriate to introduce the relevant terminology first.

Variance of a variable x is given by

$$SS(x) = \sum (x - \bar{x})^2 = \sum x^2 - \frac{(\sum x)^2}{n}$$

[A1]

where n is the number of x samples. Similarly SS(xy) is given by

$$SS(xy) = \sum (x - \bar{x}) \cdot (y - \bar{y}) = \sum x \cdot y - \frac{(\sum x) \cdot (\sum y)}{n}$$

[A2]

The equation of the least squares line is

$$y = b_0 + b_1 \cdot x$$

[A3]

where the slope b_1 of the line is given by

$$b_1 = \frac{SS(xy)}{SS(x)}$$

[A4]

and the intercept b_o of the line is given by

$$b_0 = \bar{y} - b_1 \cdot \bar{x}$$

[A5]

APPENDIX B – Modified Program Source Code

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

#define iterations 2
#define TIMES 500
#define inter_length 1000
#define no_sim 1 // number of simulations to be run.

void init_interleaver(int perm[], int d_perm[]);
void interleave(unsigned char *,int perm[], unsigned char i_info[]);
void turbo_encoder (unsigned char info_block[], int perm[],unsigned char code_packet[]);
void channel(unsigned char code_packet[],int noise_level,float *);
int error_count(unsigned char code_packet[],float *);
void read_noise();
void turbo_decoder(float *,int *,int *,int noise_level,float soft_out[]);
void decoder_init(float *,int *,float *,float *,float *);
void float_interleaved(float *,float *,int *);
void map_decoder(float *,int iter_number,float Lc,float *,float *);
void hard_decision(float *,unsigned char *);
int list_result(unsigned char *,unsigned char *,float signal_to_noise,int chan_errors);


float noise[1000];
float sigma1[11]={0.891,0.881,0.871,0.861,0.851,0.841,0.831,0.822,0.813,0.804,0.794};
/* These sigma values are for Eb/No ratios between 1 dB and 2 dB increasing at steps of 0.1 dB. */

float sigma[16]= {1.0,0.977,0.955,0.933,0.912,0.89,0.87,0.851,0.832,0.813,0.79,0.776,0.759,0.741,0.724,0.71};
/* sigma values are calculated from Eb/No=-20log[sigma]. Eb/No values are chosen as 0, 0.2, 0.4,.....,3.0 dB in the
array. */

void main()
{
        FILE *out_perf,*ifp,*ofp,*report,*out_noisy;

        int noise_level=10;  // Noise_level*0.2 dB = SNR.
        int l,i,j,k,r,err_after_dec=0,accum_errors,pixel,times;
        float acc_enc,acc_dec,acc_over;
        int perm[inter_length],d_perm[inter_length], errors=0, channel_errs;
        unsigned char byte,bit,randinfo[inter_length],code_packet[2*inter_length+8],info_block[inter_length];
        unsigned char decoded_stream[inter_length];
        float received[2*inter_length + 8],soft_info[inter_length];
        float Lc,Lu[inter_length],SNR, Pb, Pb_ch,decoder_time,encoder_time;
        time_t t_enc_b,t_enc_e,t_dec_b,t_dec_e,time_begin,time_end;  /* This is for using the computer clock for
        random seed generation, and timing the whole simulation. */


        read_noise();                   /* noise values are retrieved from ch_noise.dat file. */
        init_interleaver(perm,d_perm); // interleaver is initialized.

        acc_enc=0;
        acc_dec=0;
        acc_over=0;

        report=fopen("myreport.txt","w"); // for storing the time information.
```

```
for(r=1;r<no_sim+1;++r)
{// report loop.

        encoder_time=0;
        decoder_time=0;
        SNR=noise_level*0.2; // Eb/No in dB.

        ifp=fopen("fairybw.bmp","rb");
        ofp=fopen("outmine.bmp","wb");
        out_noisy=fopen("mynoisy.bmp","wb");

        for(l=0;l<1078;++l) // Header bits are transferred to the new file.
        {
                byte=fgetc(ifp);
                fputc(byte,ofp);
                fputc(byte,out_noisy); // to the noisy file as well.
        }

        time_begin=time(NULL); // beginning of simulation time is stored.

        for(times=0;times<TIMES*4;++times)
        {
                for(i=0;i<125;i++)
                {
                        byte=fgetc(ifp);
                        for(l=0;l<8;l++)
                        {
                                bit=(byte&1);
                                info_block[(i*8)+l]=bit;
                                byte=byte>>1;
                        }
                }

                t_enc_b=time(NULL);
                turbo_encoder(info_block,perm,code_packet); // encoding starts.
                t_enc_e=time(NULL);

                encoder_time+=difftime(t_enc_e,t_enc_b); // encoder time is stored.

                channel(code_packet,noise_level,received);

                hard_decision(received,decoded_stream); // decoded stream is constructed.

                t_dec_b=time(NULL);
                turbo_decoder(received,perm,d_perm,noise_level,soft_info); // decoding starts.
                t_dec_e=time(NULL);

                decoder_time+=difftime(t_dec_e,t_dec_b); // decoder time is stored.


                for(i=0;i<125;i++)
                {
                        pixel=0;
                        for(l=0;l<8;l++)
                        {
                                pixel=pixel>>1;
                                if(soft_info[(i*8)+l]>0) pixel=pixel+128;
                        }
                  fputc(pixel,ofp);
                }
```

```
                        // The following mapping is from the received information ( before decoding!! ).
                        for(i=0;i<125;i++)
                        { pixel=0;
                          for(l=0;l<8;l++)
                                {
                                        pixel=pixel>>1;
                                                if(decoded_stream[(i*8)+l]>0) pixel=pixel+128;
                                }

                          fputc(pixel,out_noisy);
                        }
                }// TIMES loop ends.

                time_end=time(NULL); // finishing time is stored.
                fclose(ifp);
                fclose(ofp);
                fclose(out_noisy);

                fprintf(report,"\n#sim.= %d enc. time= %f dec.time= %f, overall
                time=%f\n",r,encoder_time,decoder_time,difftime(time_end,time_begin));

                acc_enc+=encoder_time; // encoder time.
                acc_dec+=decoder_time; // decoder time.
                acc_over+=difftime(time_end,time_begin);  // overall time.

                printf("\n\nDURATION OF THE SIMULATION = %f seconds.",difftime(time_end,time_begin));
                printf("\nDURATION OF THE ENCODER = %f seconds.",encoder_time);
                printf("\nDURATION OF THE DECODER = %f seconds.",decoder_time);

        }// r ends. 'r' simulations are completed.

        fclose(out_perf);
        fprintf(report,"\n\nAVG.ENC.TIME= %f***AVG.DEC.TIME= %f***AVG.OVERALL TIME = %f\n"
        ,acc_enc/no_sim,acc_dec/no_sim,acc_over/no_sim);
        fclose(report);
} // end main.


// SUBROUTINES ARE DEFINED AS FOLLOWS:

void turbo_encoder(unsigned char info_block[], int perm[],unsigned char code_packet[])
{       /* encoding the information block for a given structure.  */

        int i;
        int state_enc1, state_enc2; /* encoder states. */
        unsigned char temp,temp2,code[3*inter_length+8],i_info[inter_length];

        state_enc1=0;
        temp=0;
        state_enc2=0;
        temp2=0;

        interleave(info_block,perm,i_info);

        for(i=0;i<inter_length;++i)
        {
                /* first sizeof(inter_length) bits are information to be coded; sytematic code. */
                code[i]= info_block[i];
```

```
                /* encoder 1 outputs are calculated. */
                temp=info_block[i]^((state_enc1&1)^((state_enc1&2)>>1));
                code[i+inter_length]=temp^(state_enc1&1); // storing output in code array.
                state_enc1=((state_enc1>>1)&3)^((temp<<1)&3); // new state of the encoder 1 is stored.

                /* encoder 2 outputs are calculated. */
                temp2=i_info[i]^((state_enc2&1)^((state_enc2&2)>>1));
                code[i+(inter_length*2)]=temp2^(state_enc2&1);/* storing output in packet array.  */
                state_enc2=((state_enc2>>1)&3)^((temp2<<1)&3); // new state of the encoder 2 is stored.
        }

        for(i=0;i<2;++i) // encoder 1 state is brought to 00;
        {// flushed outputs for encoder 1 are being calculated and stored.
                code[(2*i)+(inter_length*3)]=(state_enc1&1)^((state_enc1&2)>>1);
                code[(2*i+1)+(inter_length*3)]=((state_enc1&2)>>1);
                state_enc1=((state_enc1>>1)&3);

                /* flushed outputs for encoder 2 are calculated and stored. */
                code[(2*i)+4+(inter_length*3)]=(state_enc2&1)^((state_enc2&2)>>1);
                code[(2*i+1)+4+(inter_length*3)]=((state_enc2&2)>>1);
                state_enc2=((state_enc2>>1)&3);
        }


        /* Now the code_packet will be prepared according to the following format:

        code_packet[0.....(inter_length-1)] : Information bits.
        code_packet[inter_length....(3*inter_length/2)-1] : Encoder one 1,5,9,13,..... outputs.
        code_packet[(3*inter_length/2)....(2*inter_length)-1] : Encoder two 3,7,11,15,..... outputs.
        code_packet[(2*inter_length)....[(2*inter_length)+(2^(K-1))]-1] : Encoder one flushed outputs.
        code_packet[(2*inter_length)+(2^(K-1))....[(2*inter_length)+(2^(K))]-1] : Encoder two outputs. */


        for(i=0;i<inter_length;++i)
        {
                code_packet[i]=code[i];
                // info bits are stored as packet.

                if(i<inter_length/2)
                {
                        code_packet[inter_length+i]=code[inter_length+(2*i)];
                        // Encoded bits from encoder 1 are stored.
                        code_packet[(3*inter_length/2)+i]=code[(2*inter_length)+(2*i+1)];
                        // Encoded bits from encoder 2 are stored.
                        if(i<4)
                        {
                                code_packet[(2*inter_length)+i]=code[(3*inter_length)+i];
                                // Flushed outputs from encoder 1 are stored.
                                code_packet[(2*inter_length)+4+i]=code[(3*inter_length+4)+i];
                                // Flushed outputs from encoder 2 are stored.
                        }// if ends.
                }// if ends.
        }// for ends.
} // turbo_encoder ends.


void read_noise()
{
        FILE *inp;
        int i;
```

```
	inp = fopen("ch_noise.dat","r"); /* noise file is read and stored in noise[] array. */
	for(i=0;i<1000;++i)
	fscanf(inp,"%f",&noise[i]);
	fclose(inp);  /* noise file is closed. 1000 values are stored in noise[] array. */
} // read_noise ends.


void interleave(unsigned char info[],int perm[],unsigned char i_info[])
{ /* Interleaver section startts here. */
	int i;
	for(i=0;i<inter_length;++i)
		i_info[i]=info[perm[i]];
} // interleaver ends.


void init_interleaver(int perm[], int d_perm[])
{  /* interleaver and deinterleaver blocks are initialized here.  */

	int i,j,number,same;
	perm[0]=rand()%inter_length;
	d_perm[perm[0]]=0;
	for(i=0;i<inter_length;++i)
	{
		same = 1;
		while(same==1)
		{
			number = rand()%inter_length;
			same = 0;
			for(j=0;j<i;++j) /* checking if same numbers have been generated. */
			{/* if same numbers are generated, randomizing repeats itself. */
				if(number == perm[j])
				same = 1;
			} // 'j' for ends.
		}  // while ends.
			perm[i]= number;
			d_perm[perm[i]]=i;
	 }  // 'i' for ends.
} /* init_interleaver ends. */


void channel(unsigned char code_packet[],int noise_level,float received[])
{/* This is the channel where the noise is added to the transmitted data. */

	int i;
	time_t t;  /* This is for using the computer time for random seed generation. */

	srand((unsigned) time(&t)); /* This is for unix compiling. It fuctions as a randomizer. */
	/*randomize(); random number generator is initialized with a new number.
	this way the noise generated is different everytime the 'channel' function
	is called. If the randomize() function is not called before the use of
	rand() function, the noise[] array is always the same. */

	for(i=0;i<(2*inter_length)+8;++i)
		{
		received[i]=(float)(2*code_packet[i]-1) + (sigma[noise_level]*noise[rand()%1000]);
		}
}// channel is closed.

int error_count(unsigned char code_packet[],float received[])
{
```

```
	int i, errors;
	errors=0;
	for(i=0;i<(2*inter_length)+8;++i)
	{
		if(received[i]<0 && (int)code_packet[i]==1)
			errors+= 1;
		if(received[i]>0 && (int)code_packet[i]==0)
			errors+=1;
	} /* number of errors for a given SNR is calculated. */
	return(errors);
} // errors end.

void turbo_decoder(float received[],int perm[],int d_perm[],int noise_level,float soft_out[])
{
int i,iter_no;
float received_enc1[2*inter_length+4],received_enc2[2*inter_length+4];
float Lu[inter_length],Lc;

decoder_init(received,perm,received_enc1,received_enc2,Lu);

Lc = 4*pow(10,(float)noise_level/5); // Channel information is calculated. Eb/No (in dB) = noise_level*0.2.
// Lc = 4*(Eb/No) ( pure Eb/No ratio is used ).

	for(iter_no=0;iter_no<iterations;++iter_no)
	{

		/* FIRST DECODER IS FOLLOWING. */

		map_decoder(received_enc1,0,Lc,Lu,soft_out); // uninterleaved received signal will be decoded.
		float_interleaved(soft_out,Lu,perm);

		/* SECOND DECODER IS FOLLOWING. */

		map_decoder(received_enc2,(iter_no+1)/iterations,Lc,Lu,soft_out); // interleaved received signal will
		//be decoded.
		float_interleaved(soft_out,Lu,d_perm);
	} // iterations loop ends.
	for(i=0;i<inter_length;++i)
	  soft_out[i] = Lu[i];
}// turbo_decoder ends.

void decoder_init(float received[],int perm[],float received_enc1[],float received_enc2[],float Lu[])
{

	int i;
	float temp[inter_length];    // for storing the interleaved received info sequence.

	float_interleaved(received,temp,perm);

	for(i=0;i<inter_length;++i)
	{/*i =   01 23 45 67 89 ..... index i.
		IP I0 IP IO IP ..... received_enc1[] array structure.
		I: Information.
		0: Zero symbol.
		P: Parity. */

	 /*i =   01 23 45 67 89 ..... index i.
		I0 IP I0 IP IO ..... received_enc2[] array structure.
		I: Information.
		0: Zero symbol.
```

```
                P: Parity. */

                Lu[i]=0;  /*        initializing the a priori information to zero. Note this is the
                                                                log ratio of the probabilities. */

                received_enc1[2*i]=received[i]; // even bits are uninterleaved information.
                received_enc2[2*i]=temp[i];          // even bits are interleaved information.

                if(i%2 == 0)
                {
                        received_enc1[2*i+1]=received[inter_length+(i/2)];
                        received_enc2[2*i+1]=0;
                }
                else
                {
                        received_enc1[2*i+1]=0;
                        received_enc2[2*i+1]=received[(3*inter_length/2)+((i-1)/2)];
                }

                if(i<4)
                { /* i= 0  1  2  3   index i.
                         T1 T2 T3 T4  tail bits in received[] that are added to the end of
                                                                received_enc1[] array.
                         T5 T6 T7 T8      tail bits in received[] that are added to the end of
                                                                received_enc2[] array. */
                        received_enc1[2*inter_length+i]= received[2*inter_length+i];
                        received_enc2[2*inter_length+i]= received[(2*inter_length+4)+i];
                } // if ends.
        }// for ends.
}/*decoder_init ends.*/


void float_interleaved(float data[],float i_data[],int permutation[])
{

        int i;
        for(i=0;i<inter_length;++i)
                i_data[i]=data[permutation[i]];

}


void map_decoder(float received_stream[],int iter_number,float Lc,float Lu[],float soft[])
{
        int i,j,logic;
        int alpha_words[4][2]={{0,3},{1,2},{0,3},{1,2}};
        int beta_words[4][2]={{0,3},{0,3},{1,2},{1,2}};
        float ex_gamma[4][inter_length+2],gamma[4][inter_length+2];
        float alpha[4][inter_length+3],beta[4][inter_length+3],alpha_temp;
        float temp,temp1,num,den,same,same2,same_beta1,same_beta2;


        for(i=0;i<inter_length+2;++i)
        { /* This loop calculates the extrinsic gammas and gammas.*/

                if(i<inter_length)
                { /*      This section calculates the gamma values for the information bit stream.
                                the tail bits are not taken into account. There are two possible signs for the
                                information bits and also ex_gamma[] values vary with the branch codewords.
                                Therefore, four different combinations for the calculation of the gamma[]
```

```
                        values are needed. The gamma[] values are calculated for every single possible
                        codewords at each branch at each depth. So here a matrix is being constructed. */

                gamma[0][i]= -0.5*(Lc*(received_stream[2*i]+received_stream[2*i+1])+ Lu[i]);
                gamma[1][i]= -0.5*(Lc*(received_stream[2*i]-received_stream[2*i+1])+ Lu[i]);
                gamma[2][i]=  0.5*(Lc*(received_stream[2*i]-received_stream[2*i+1])+ Lu[i]);
                gamma[3][i]=  0.5*(Lc*(received_stream[2*i]+received_stream[2*i+1])+ Lu[i]);
        }
        else
        {/*  The same gamma[] calculation is performed for the tail bits which cause
                the decoding trellis to terminate. The only difference between the two
                gamma[] calculations is that for the tail bits there is no a priori
                information ( Lu[] ) to be considered since there is no information bits
                in the tail codewords. */

                gamma[0][i]= -0.5*(Lc*(received_stream[2*i]+received_stream[2*i+1]));
                gamma[1][i]= -0.5*(Lc*(received_stream[2*i]-received_stream[2*i+1]));
                gamma[2][i]=  0.5*(Lc*(received_stream[2*i]-received_stream[2*i+1]));
                gamma[3][i]=  0.5*(Lc*(received_stream[2*i]+received_stream[2*i+1]));

                for(j=0;j<4;++j)
                {
                        alpha[j][i]=-1e32;//          log(0)=-infinity;
                        beta[j][i]=-1e32;  /* All alpha and beta values are initialized to
                        very small negative numbers since log values are used throughout the decoding
                        process. */
                }
        }
}// end for.

i=inter_length+2;
for(j=0;j<4;++j)
{
        alpha[j][i]=-1e32;//          log(0)=-infinity;
        beta[j][i]=-1e32;  /* Last alpha and beta values are initialized to
                        very small negative numbers since log values are used throughout the decoding
                        process. */
        if(j==0)
        {
                alpha[j][0]=0;       // alpha and beta values for the end section of the trellis.
                beta[j][i]=0;        // are initialized to zero. ( log(1)=0 )

        }// if ends.
}// for ends.

for(i=1;i<inter_length+2;++i) // i refers to the trellis depth.
{

        for(j=0;j<4;++j) // j refers to the future state.
        { /*      In this loop alpha and beta values are being calculated and are stored
                in a matrix format. This calculation is done by using the the approximation,
                log(exp[x]+exp[y])= max(x,y)        ( Hagenauer ). */

                /* Alpha calculations proceed.*/

                if(alpha_words[j][0]==0)
                {
                        temp=alpha[(j>>1)&3][i-1]+gamma[(j<<1)&3][i-1];
                        temp1=alpha[(((~j)&3)>>1)&3][i-1]+gamma[(~((j<<1)&3))&3][i-1];
                        if(temp>temp1)
```

```
                                alpha[j][i]=temp;
                        else
                                alpha[j][i]=temp1;
                }// if ends.

                else
                {
                        temp=alpha[(~(j>>1))&3][i-1]+gamma[(((j<<1)&3)>>1)&3][i-1];
                        temp1=alpha[((j>>1)&3)+2][i-1]+gamma[(j<<1)&3][i-1];
                        if(temp>temp1)
                                alpha[j][i]=temp;
                        else
                                alpha[j][i]=temp1;
                }// else ends.

                /* Beta calculations proceed.*/

                if(beta_words[j][0]==0)
                {
                        temp=beta[(j<<1)&3][inter_length+2-i+1]+gamma[(j>>1)&3][inter_length+2-i];
                        temp1=beta[((j+1)<<1)&3][inter_length+2-i+1]
                                        +gamma[(~((j>>1)&3))&3][inter_length+2-i];
                        if(temp>temp1)
                                beta[j][inter_length+2-i]=temp;
                        else
                                beta[j][inter_length+2-i]=temp1;
                }// if ends.

                else
                {
                        temp=beta[(~(j<<1))&3][inter_length+2-i+1]+gamma[(j>>1)&3][inter_length+2-i];
                        temp1=beta[((j<<1)&3)+1][inter_length+2-i+1]
                                        +gamma[(((j>>1)&3)<<1)&3][inter_length+2-i];
                        if(temp>temp1)
                                beta[j][inter_length+2-i]=temp;
                        else
                                beta[j][inter_length+2-i]=temp1;
                }// else ends.
        }// for j ends.
}// for i ends.

/* NOW THE A PRIORI OUTPUTS ( SOFT OUTPUTS ) FOR EACH DEPTH WILL BE CALCULATED.
THE TAIL BITS AND THE CODED BITS ARE CALCULATED SEPARATELY BECAUSE OF THE
DIFFERENCE IN THE GAMMA AND EXTRINSIC GAMMA USE */
/*************************************************************************/


if(iter_number==0) /* This line is checking if the iteration is the final one or not. In the final iteration
ex_gamma is replaced by the gamma expression. The calculation is the same as usual. */
{

        /* extrinsic gammas are being calculated here. Note that ex_gamma[] values
        are calculated only for parity bits. Therefore for K=3, there are two
        possible ex_gamma[] values; one for x = -1 and one for x = +1. That`s why
        00 codeword = 10, and 01 = 11. Only last bits are different. Remember that
        0 = -1 and 1 = +1 when modulated. */


        ex_gamma[0][inter_length+1]= (-0.5*Lc*(received_stream[2*(inter_length+1)+1]));
        ex_gamma[1][inter_length+1]= (0.5*Lc*received_stream[2*(inter_length+1)+1]);
```

```
//      ex_gamma[2][inter_length+1]= ex_gamma[0][inter_length+1];
//      ex_gamma[3][inter_length+1]= ex_gamma[1][inter_length+1];

        for(i=0;i<inter_length;++i)
        {
                num=0;
                den=0;

                ex_gamma[0][i]= (-0.5*Lc*(received_stream[2*i+1]));
                ex_gamma[1][i]= (0.5*Lc*received_stream[2*i+1]);
        //      ex_gamma[2][i]= ex_gamma[0][i];  No need.
        //      ex_gamma[3][i]= ex_gamma[1][i];  No need.

                        for(j=0;j<4;++j)
                        {
                          /*    for same ;(((~j)&3)>>1)&3; if j=0 or 1 same=1, if j=2 or 3 same=0. */
                          /*    for same2; (j>>1)&3; if j=0 or 1 same2=0, if j=2 or 3 same2=1. */

                                alpha_temp=alpha[j][i];

                                same=ex_gamma[(((~j)&3)>>1)&3][i];
                                same2=ex_gamma[(j>>1)&3][i];

                                /* The following if else is for choosing the relevant beta in
                                        num, den calculations.*/

                                if(((j>>1)&3)==0) // if j=0 or 1.
                                {
                                        same_beta1=beta[(((~j)&3)<<1)&3][i+1]; // to be used in
                                        // numerator.
                                        same_beta2=beta[(j<<1)&3][i+1];  // to be used in denominator.

                                }// if ends.

                                else // if j=2 or 3.
                                {
                                        same_beta1=beta[((j<<1)&3)+1][i+1];  // to be used in numerator.
                                        same_beta2=beta[(((j<<1)&3)+3)&3][i+1]; // to be used in
                                        //denominator.

                                }// else ends.

                                /* The numerator and the denominator will be calculated next.*/

                                        temp1=alpha_temp+same_beta1+same;
                                        if(temp1>num)
                                                num=temp1;

                                        temp1=alpha_temp+same_beta2+same2;
                                        if(temp1>den)
                                        den=temp1;
                        }// for ends.
                        soft[i]=num-den;
        }// for ends.
}// if ends.

else // for the final iteration ex_gammas are replaced by gammas.
{
        for(i=0;i<inter_length;++i)
        {
```

```
                num=0;
                den=0;
                        for(j=0;j<4;++j)
                        {
                                alpha_temp=alpha[j][i];

                                logic=(j>>1)&3;
                                same=gamma[(~logic)&3][i];
                                same2=gamma[logic][i];

                                /* The following if else is for choosing the relevant beta in
                                num, den calculations.*/

                                if(((j>>1)&3)==0) // if j=0 or 1.
                                {
                                        same_beta1=beta[(((~j)&3)<<1)&3][i+1]; // to be used in
                                        //numerator.
                                        same_beta2=beta[(j<<1)&3][i+1];  // to be used in denominator.

                                }// if ends.

                                else // if j=2 or 3.
                                {
                                        same_beta1=beta[((j<<1)&3)+1][i+1];         // to be used in
                                        //numerator.
                                        same_beta2=beta[(((j<<1)&3)+3)&3][i+1]; // to be used in
                                        //denominator.

                                }// else ends.

                                        /* The numerator and the denominator will be calculated for the
                                        final iteration.*/

                                        temp1=alpha_temp+same_beta1+same;
                                        if(temp1>num)
                                                num=temp1;

                                        temp1=alpha_temp+same_beta2+same2;
                                        if(temp1>den)
                                                den=temp1;
                        }// for ends.
                        soft[i]=num-den;
                }// for ends.
        }// else ends.
} // end map_decoder.

void hard_decision(float soft[],unsigned char decoded_data[])
{       /* Hard decision is done according to the following criteria:
                soft[i] > 0 , then decoded_data[i]=1
                soft[i] < 0 , then decoded_data[i]=0  */

        int i;

        for(i=0;i<inter_length;++i)
        {

        if(soft[i]>=0)
                decoded_data[i]=1;
        else
                decoded_data[i]=0;
```

```
        }

} // hard_decision ends.


int list_result(unsigned char infobits[],unsigned char decoded_bits[],float signal_to_noise,int chan_errors)
{/* This subroutine is the data sink. It lists the results of the simulation. */

        int i,dec_errs,corrected=0,no_ones=0,no_zeros=0;

        printf("\n\nThe signal-to-noise ratio (SNR): %f dB\n\n",signal_to_noise );
        printf("The interleaver length: %d\n\n", inter_length);
        printf("The number of iterations: %d\n\n", iterations);

        dec_errs=0;

        for(i=0;i<inter_length;++i) /* number of errors after decoding are counted. */
        {
                if(infobits[i] != decoded_bits[i])
                        dec_errs+=1;
                if(infobits[i]==1)
                        no_ones += 1;
        }

        no_zeros = inter_length-no_ones;

        printf("The number of 1s transmitted: %d\n\n", no_ones);
        printf("The number of 0s transmitted: %d\n\n", no_zeros);
        printf("The number of bit errors during transmission: %d\n\n", chan_errors);
        printf("The number of bit errors after decoding: %d\n\n",dec_errs);

        corrected=chan_errors - dec_errs;

        printf("The number of bit errors corrected by the code: %d\n\n", corrected);

        return(dec_errs);
}
```

APPENDIX C – Error Performance by Color Images

Properties of the original image used are given below.

SIZE	2000000 bits.
DIMENSION	500 by 500 pixels.
FORMAT	*.bmp , Windows RGB encoded.
QUALITY	256 colors.

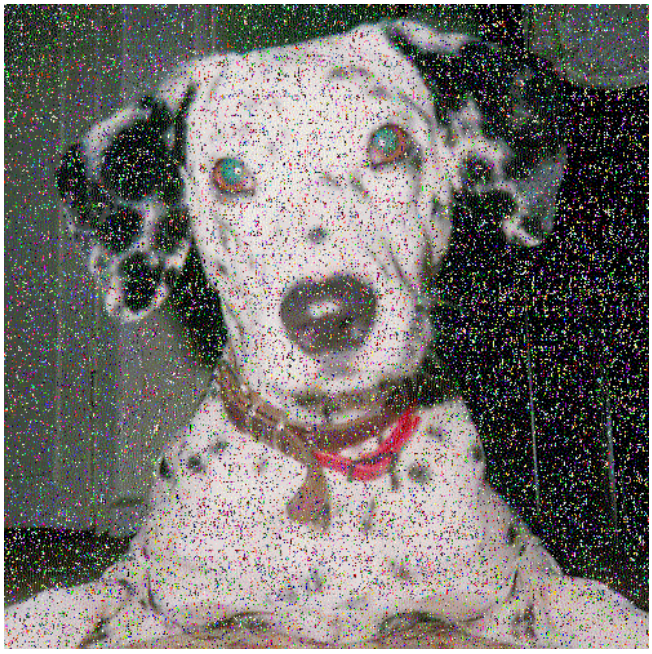
Interleaver length and SNR are kept constant at 1000 and 1.8 dB, respectively. Iterations simulated are 2, 4, 8 and 10 as indicated on each page. Simulations for the same iterations of the original and modified codes are placed on the same page to aid in convenient comparison. The original image is presented below.

ORIGINAL IMAGE

Number of Iterations: 1 ***Interleaver length:*** 1000 **E_b/N_o*:*** 1.8 dB

Simulation: **Modified Code** ***BER*** **= 0.03466**

Simulation: **Conventional Code** ***BER*** **= 0.03510**



Number of Iterations: 2 ***Interleaver length:*** 1000 **E_b/N_o*:*** 1.8 dB

Simulation: **Modified Code** ***BER*** **= 0.00828**

Simulation: **Conventional Code** ***BER*** **= 0.00826**

Number of Iterations: 4 ***Interleaver length:*** 1000 E_b/N_o***:*** 1.8 dB

Simulation: **Modified Code** ***BER*** **= 0.00092**

Simulation: **Conventional Code** ***BER*** **= 0.00075**

Number of Iterations: 8 ***Interleaver length:*** 1000 E_b/N_o***:*** 1.8 dB

Simulation: **Modified Code** ***BER*** **= 0.00027**

Simulation: **Conventional Code** ***BER*** **= 0.00023**

Number of Iterations: 10 ***Interleaver length:*** 1000 **E_b/N_o*:*** 1.8 dB

Simulation: **Modified Code** ***BER*** **= 0.00020**

Simulation: **Conventional Code** ***BER*** **= 0.00016**

Number of Iterations: 16 ***Interleaver length:*** 1000 **E_b/N_o*:*** 1.8 dB

Simulation: **Modified Code** ***BER*** **= 0.00016**

Simulation: **Conventional Code** ***BER*** **= 0.00013**