Technical Report

Investigation of Different Constituent Encoders in a Turbo-code Scheme for Reduced Decoder Complexity

Submitted to:

NASA Lewis Research Center
21000 Brookpark Road
Cleveland, Ohio 44135

Submitted by:

Dr. S.C. Kwatra, Principal Investigator
Peter Curry, Graduate Research Assistant

Department of Electrical Engineering
College of Engineering
The University of Toledo
Toledo, Ohio 43606

Report No. DTVI-55

May, 1998

This report contains part of the work performed under NASA grant NAG3-1718 during the period September 1995 to April 1996. The research was performed as part of the Masters thesis requirement of Mr. Peter Curry

S.C. Kwatra

Principal Investigator

An Abstract of

Investigation of Different Constituent Encoders in a Turbo-Code System for Reduced Complexity Decoding

Peter Curry

Submitted in partial fulfillment of
the requirements of the
Master of Science Degree

The University of Toledo
May 1998

A large number of papers have been published attempting to give some analytical basis for the performance of Turbo-codes. It has been shown that performance improves with increased interleaver length. Also procedures have been given to pick the best constituent recursive systematic convolutional codes (RSCC's). However testing by computer simulation is still required to verify these results. This thesis begins by describing the encoding and decoding schemes used. Next simulation results on several memory 4 RSCC's are shown. It is found that the best BER performance at low E_b/N_o is not given by the RSCC's that were found using the analytic techniques given so far. Next the results are given from simulations using a smaller memory RSCC for one of the constituent encoders. Significant reduction in decoding complexity is obtained with minimal loss in performance. Simulation results are then given for a rate 1/3 Turbo-code with the result that this code performed as well as a rate ½ Turbo-code as measured by the distance from their respective Shannon limits. Finally the results of simulations where an inaccurate noise variance measurement was

used are given. From this it is observed that Turbo-decoding is fairly stable with regard to noise variance measurement.

Acknowledgments

I would like to thank my advisor, Dr. S. C. Kwatra for his patience. I would also like to thank Dr. J. Kim and Mr. R.E Jones for serving on my committee. I would also like to thank NASA for funding me and making this research possible. Finally I would like to thank my friends and family.

Table of Contents

List of Figures

Chapter 1

Introduction

Low bit error rates (BER) in high noise environments have required the use of very complex channel coding and decoding schemes. According to Shannon's theorem very long random codes can approach Shannon's limit [1]. This limit is defined as zero probability of bit error (usually this is taken as BER of 10^{-5} or some other convenient figure of merit) when the E_b/N_o is larger than

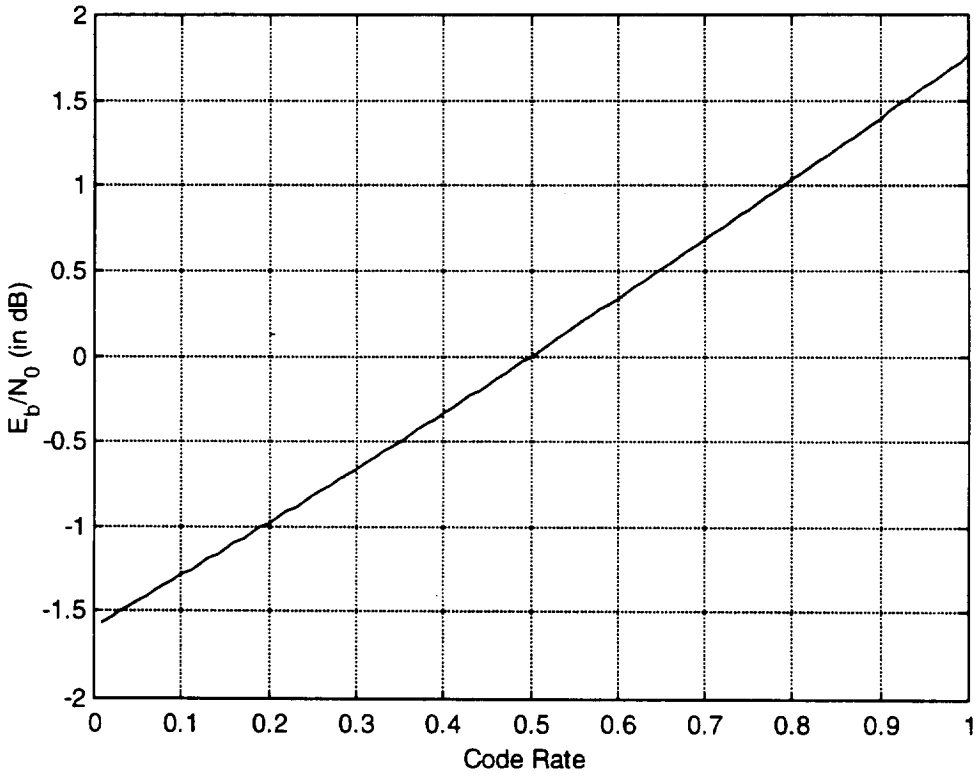


Figure 1.1 The Limits for Reliable Communication

a given value which depends on the rate of the code. E_b/N_o required for given rates is shown in Figure 1.1 assuming no intersymbol interference, and minimum Nyquist bandwidth [2]. However, long random codes are, in general, extremely difficult to decode. In order to decrease the complexity of the decoder several approaches have been tried. A typical practice, introduced by Forney [3], is the concatenation of more than one code. This method is composed of coding the information bits by an outer encoder and inputting the output of the outer encoder into a second inner encoder which is then output to the channel. The bits can be decoded by decoding the output of the channel by the inner decoder first and using that as an input to the outer decoder. A typical example of this would be a Reed Solomon code as an outer code with a convolutional code as the inner code as shown in Figure 1.2.

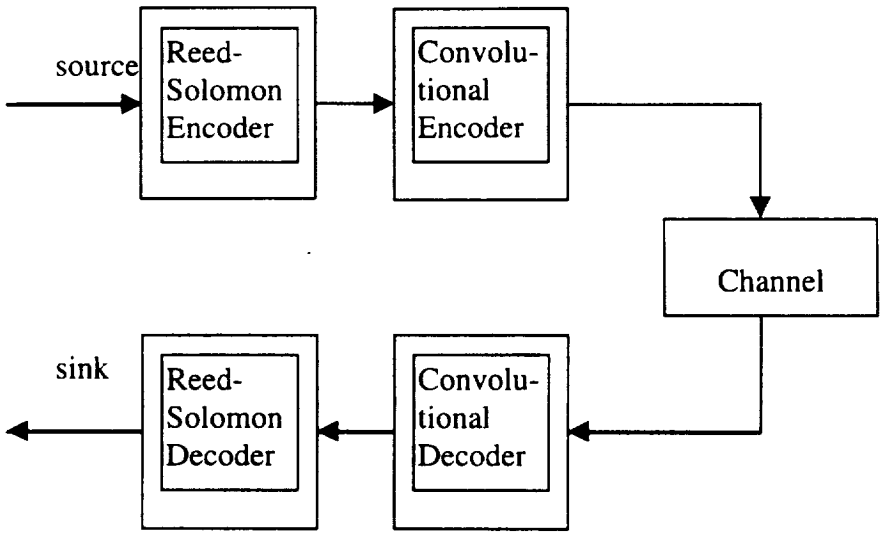


Figure 1.2 A Serial Concatenated Scheme

Recently a new concatenation scheme has been proposed. This scheme is called parallel concatenation. Parallel concatenation is done by encoding information streams that are linked through a pseudo-random interleaver as shown in Figure 1.3. Delays are not shown in the figure. The input to the interleaver is presented as blocks of bits. The process of using parallel concatenation in conjunction with recursive systematic convolution codes (RSCC's) has produced codes, nicknamed Turbo-codes [4], that have phenomenal error correcting capacity at very low bit energy to noise variance ratios (E_b/N_o). For example the rate ½ code (accomplished by puncturing every other bit from each RSCC output) in [4] was found to have a BER of 10^{-5} at E_b/N_o of only .7 dB. This is a savings of about 9 dB over uncoded BPSK which is shown in Figure 1.4, but more importantly it is within .7 dB of the Shannon limit for a rate ½ code (see Figure 1.1).

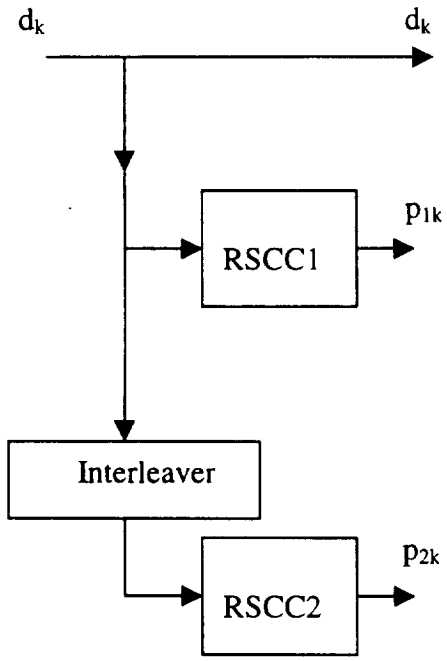


Figure 1.3 The General Encoding Scheme for Turbo-codes.

While these codes have very good BER performance there are some difficulties with these codes. One of the problems is the fact that the decoding of these codes requires soft outputs. The optimal decoding algorithm, the Maximum A posteriori Probability (MAP) algorithm is very complex due to the number of operations needed and the amount

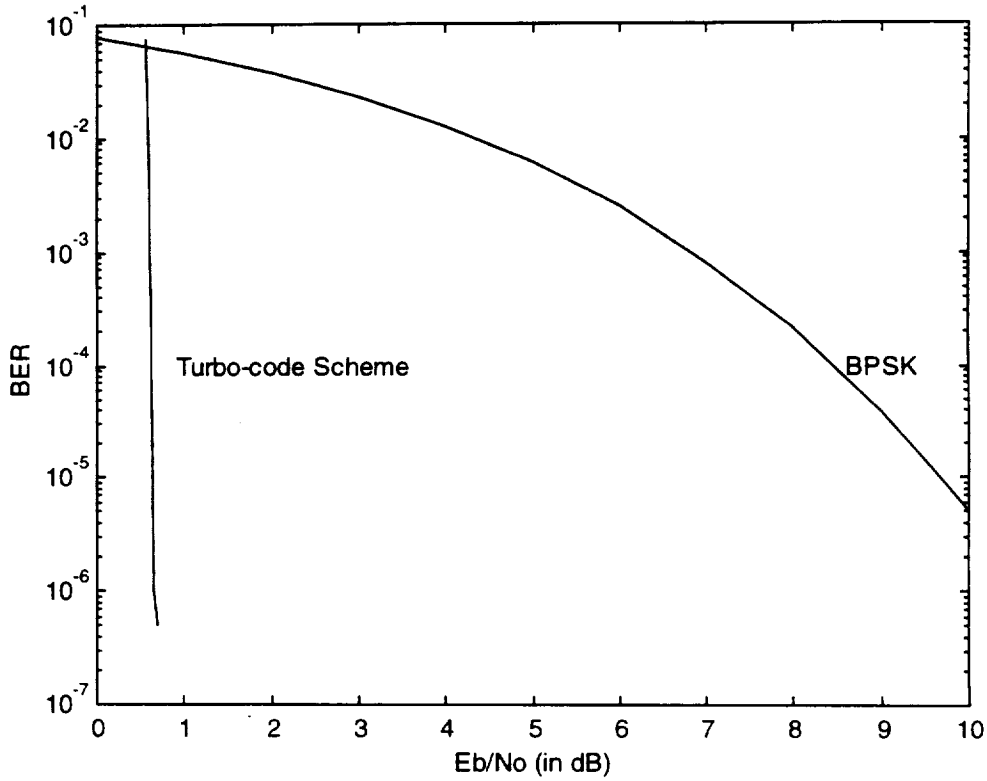


Figure 1.4 BER vs E_b/N_o for Uncoded BPSK

of memory required. There are simpler decoders, such as the Soft Output Viterbi Algorithm (SOVA) and the Max-Log MAP, but they are both sub-optimal algorithms.

One of the objectives of this research is to investigate the effects of using different generators for the RSCC's on the performance of the Turbo-codes. This

will be done using computer simulation. While several analytical methods have been proposed for choosing proper RSCC's in the Turbo-code system they have not all been tested by computer simulation. Computer simulation is necessary to confirm results that were given by analytical methods. Also it has been seen that concatenating a smaller memory convolutional encoder with a memory four convolutional code does not degrade performance levels very much, while decoding is less complicated [5]. The performance of these schemes will be evaluated. In cases where bandwidth is not a concern but power is limited, lower rate encoding schemes can be of use. Simulations will be run to determine if the performance of a lower rate (1/3) Turbo-code scheme generates good results. The results of the rate 1/3 code will be compared with the rate ½ Turbo-code scheme. Also the effects of inaccurate measurement of noise variance on Turbo-code performance will be investigated (the MAP decoder requires an estimate of noise variance). This is done to see how stable the Turbo-decoding process is in the case when noise variance is measured inaccurately.

This thesis will begin by describing the general encoding scheme. Then detailed descriptions of the encoding components of Turbo-codes including descriptions of the constuction of RSCC's and the interleavers, as well as motivations for their use, will be given. Next will be the description of the decoding process beginning with a description of the soft output decoders (specifically the MAP algorithm) and then describing the Turbo-decoding process. Finally the research findings will be presented.

Chapter 2

Overview of Encoding Components

2.1 General Overview

Most Turbo-codes are encoded by concatenating two RSCC's through an interleaver. A block of message bits is encoded with a RSCC. That same block of message bits is interleaved by a pseudo-random interleaver and encoded with another RSCC (see Fig. 1.3). The systematic information is sent only once, not separately with each RSCC.

The reasons that this channel coding scheme works so well are that it combines three different areas that help to produce good codes [6]. The three areas are:

- combining several codes by concatenation
- maximum use of channel information (i.e. soft decoding)
- random like distribution of codewords

The purpose of this chapter is to show how Turbo-codes use the RSCC's and the interleaver to mimic random codes in some ways. Soft decoding algorithms will be discussed in chapter 3.

It was shown by Shannon that large random codes can decode near the Shannon limit. This suggests that good codes should have a distance distribution that mimics that of random coding rather than simply having a large minimum

distance. The weight distribution histogram of a fixed length random block code would be very close to a binomial distribution. It would have very few low weight or high weight codewords, and the majority of the codewords would have a weight very close to the middle of the weight spectrum. Designing such codes with enough structure to decode with a reasonable amount of complexity and arbitrary parameters (i.e. length, rate) is not possible yet. However Turbo-codes are able to generate a weight distribution that has been shown to have a distribution with a shape similar to that of random codes. The following sections will detail how each component of the Turbo-encoder allows Turbo-codes to mimic random codes.

2.2 Recursive Systematic Convolution Codes

This section will begin with an example of a non-systematic convolutional code (NSCC). From there it is shown how to construct RSCC's and some of the properties of RSCC's are given.

The structural sequences of channel coding have been classified into two main categories, block and convolutional encoding. Block coding is performed by accepting a given number of bits (k) and using algebraic rules to form a number of parity bits (p). When the information is transmitted the parity bits are tacked onto the information bits. The total rate of the code, k/n, is given as the number of information bits (k) divided by the total number of bits sent (k+p).

Usually convolutional encoding is done by accepting bits serially, one bit at a time through m tapped delay lines (a more general procedure is shown in [7]). This means that the output bits will not only depend on the current input bit but will also depend on at least the previous m input bits. An (n, k = 1, m) convolutional code can be implemented that accepts 1 input bit at a time, has n output linear sequential circuits with input memory of order m. An example of a (2, 1, 2) nonsystematic encoder is shown in Figure 2.2.1. One way to think about the output of the convolutional encoder is to consider the output to an impulse when the encoder is in the zero state. The impulse

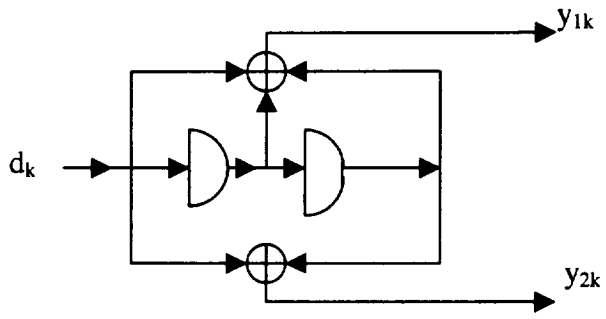


Figure 2.2.1 A Non-Systematic Convolutional Encoder

response of the system can be used to obtain a semi-infinite generator matrix due to the linearity of the response. The generator matrix, G, of the circuit shown is given in Figure 2.2.2. Notice that the output of the first row is the impulse response of the system (1 1 1 0 1 1). The generator bits are grouped in pairs of two. The first number is from y_{1k} and the second number is from y_{2k}. One way to generate the output for a given input sequence, $\{d_k\}$, is to multiply the row

vector by the generator matrix, remembering that addition is done modulo 2. Thus, if d = [1 0 1] then the output is given by

$$d \cdot G = \left|\begin{array}{cccccc} & 1\ 1 & 1\ 0 & 1\ 1 & 0\ 0 & 0\ 0 \\ [1\ 0\ 1] & 0\ 0 & 1\ 1 & 1\ 0 & 1\ 1 & 0\ 0 \\ & 0\ 0 & 0\ 0 & 1\ 1 & 1\ 0 & 1\ 1 \end{array}\right| = [1\ 1 \quad 1\ 0 \quad 0\ 0 \quad 1\ 0 \quad 1\ 1]$$

$$\left|\begin{array}{ccccc} 1\ 1 & 1\ 0 & 1\ 1 & 0\ 0 & 0\ 0 \\ 0\ 0 & 1\ 1 & 1\ 0 & 1\ 1 & 0\ 0 \\ 0\ 0 & 0\ 0 & 1\ 1 & 1\ 0 & 1\ 1 \\ . & . & . & . & \\ . & . & . & . & \end{array}\right| \quad G =$$

Figure 2.2.2 A Generator Matrix

That the output of a convolutional encoder is dependant not only on the current input but also the previous m inputs, suggests that we can gain insights into the properties of a convolutional encoder with a state diagram. A state diagram for the encoding circuit in Figure 2.2.1 is shown in Figure 2.2.3. This diagram can be important for determining some of the distance properties of convolutional codes. These distance properties can give information about how well a given code will perform. The state diagram shows the states (0, 1, 2, 3), the inputs and the outputs they cause. For example if the encoder was in state 2 and a 1 was received, the next state would be state three and the output at that time would be (0 1).

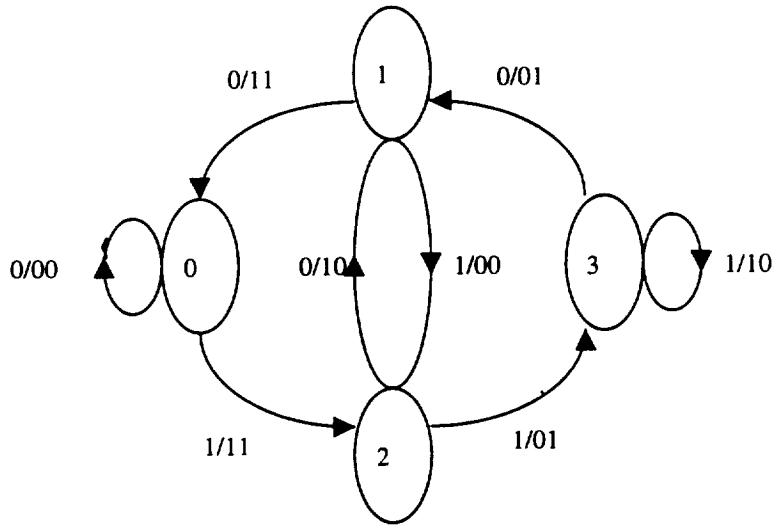


Figure 2.2.3 The Input-Output State Diagram of the NSCC

Usually the most important distance measure for convolutional codes is the minimum free distance. This is defined as [7]

$$d_{free} = \min\{d(v',v''):u'\neq u''\}$$

where v' and v'' are the codewords corresponding to the input vectors u' and u'' respectively (d_{free} is not related to $\{d_k\}$ which was defined as the input sequence). This means that d_{free} is the minimum distance between any two codewords in the code. Another way of saying this is that the free distance of a code is the number of bits that need to be changed in a given word for the output to be a different codeword. This is important for determining the error correcting ability of a code.

The example given is for a NSCC. However RSCC's have been discovered which perform better than the best NSCC's at any SNR for high code rate (rate > 2/3) [8]. These encoders are constructed from NSCC's by using a feedback loop and setting one of the outputs, y_k, equal to the input, d_k. Since the output of these codes is separated into the systematic portion of the output and the

other portion, the other portion will be called the parity sequence and the parity bit at time k will be denoted by p_k. An example of a RSCC is shown in Figure 2.2.4 with the state diagram of this encoder given in Figure 2.2.5.

The generator given in Figure 2.2.4 is called a 5_7 RSCC. The 5 and 7 represent octal numbers that are converted to binary to represent the connections in a generator circuit. The first number will be called the FB (feedback) connection, while the second will be called the FF (feedforward) connection.

It was claimed that these codes perform better than the NSCC's at high code rates. A high code rate is accomplished by puncturing the outputs of the convolutional encoder. This means systematically deleting some of the output bits. While puncturing can be done in different ways, it is usually done by eliminating every other bit out of the non-systematic portion (p_k in Figure 2.2.4) and will be done this way for the remainder of this thesis. For this punctured code the rate would then be 2/3 (1 information bit transmitted for every 1½ bits transmitted). For Turbo-codes the overall rate has generally been ½ by using two punctured

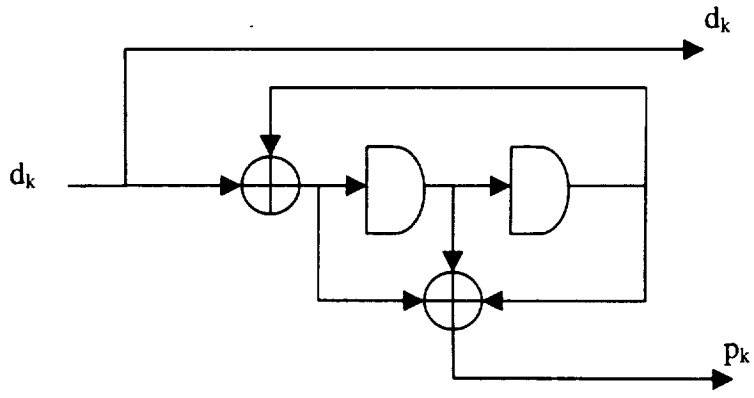


Figure 2.2.4 A Recursive Systematic Convolutional Encoder

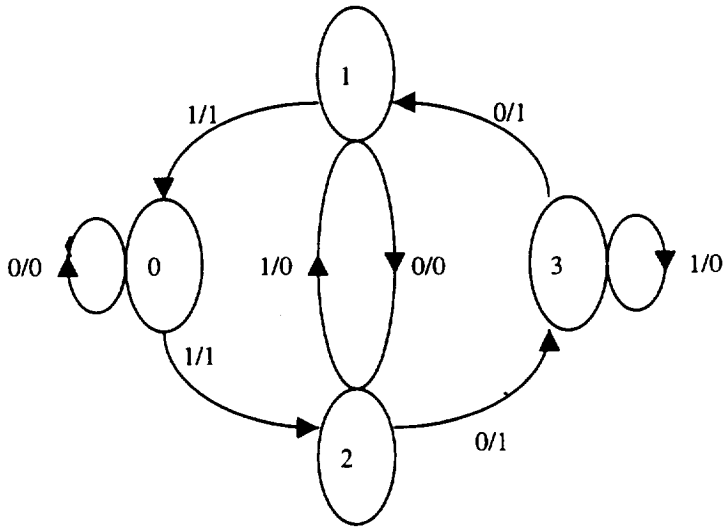


Figure 2.2.5 Input-Output State Diagram of the RSCC

RSCC's and transmitting the systematic portion only once.

The reason that RSCC's are important is that they have been found to give the greatest gain when used as the parallel concatenated codes [2] (it has been shown that NSCC's give almost no gain when constructed as Turbo-codes). One of the ways that they can be seen to be different from the NSCC's is that a finite weight input sequence can be mapped into an infinite weight output sequence. This is shown by the impulse response of the encoder of Figure 2.2.4 which is p_k = [1 1 1 0 1 1 0 1 1 0 1 1 0 . . .]. Notice that after the first parity bit the sequence repeats itself with a period of 3 bits. In general the impulse response of a well designed memory m RSCC will repeat itself after $2^m - 1$ bits. A nonrecursive NSCC maps a finite weight input sequence into a finite weight output sequence. Since one of the goals is to make the codewords have a random distribution and since the output weight of a nonrecursive NSCC is somewhat

correlated with its' input weight, using NSCC's would not be as good for designing random like codes.

[6] showed that for most input sequences the output weight of RSCC's has the same distribution as that of a random code sequence. While most input sequences will have an output weight that approximates that of random sequences there are input sequences that cause low output weights. For example, there are sequences with as few as two ones that will cause the encoder to go from the zero state to a nonzero state and back and generate low weight codewords. For the encoder of Figure 2.2.4 a sequence that would do this is d_k = [1 0 0 1 0 0 0 0 ...]. The parity output for this sequence is p_k = [1 1 1 1 0 0 0 0 ...]. This means that any sequence that is a shifted version of the one mentioned will have an output weight of 6. These codewords are examples of the codewords that cause the codes to perform poorly. The object of encoding of Turbo-codes through an interleaver is to "boost" the low output weight codewords that would be generated by a single RSCC. In other words what the interleaver is designed to do is to force most of those input words that produce low weight output codewords through RSCC1 (i.e. few ones in p_{1k}) to produce higher weight codewords through RSCC2 (p_{2k}).

When decoding convolutional codes it is desirable to force the encoder into a known final state to protect the final few information bits. RSCC's cannot be driven to the all zero state by adding a specific number of zeros (this can be seen in the state diagram, Figure 2.2.5) as can be done with NSCC's. Some simple, sub-optimal solutions to this are to fail to protect the final bits sent in a

block by appending no bits onto the end. This way neither the final state of the encoder or the final bits are known. Another choice that can be made is to force the encoder into the all zero state by a proper choice of m (where m is the encoder memory) end bits. This allows the decoder to know that it is in the all zero state while not knowing the final m bits.

Choosing the best RSCC generators for Turbo-codes has been done by several methods. One method that has been used to determine the best generators is using the encoder with the best distance properties [8]. Another method is given in [9]. This method involves using a primitive polynomial as the FB connection and determining the FF connections based on the resulting BER. That paper also lists several good generators.

2.3 Interleavers

The use of a good interleaver is the most important factor in achieving the best possible performance of Turbo-codes [10]. The interleaver permutes the information bits in such a way as to make the output of RSCC2 (from Figure 1.3) appear to be independent of the information sequence and therefore random-like, but to have a structure that permits decoding. While the mechanics of what exactly makes up the best psuedo-random interleavers is not completely understood, and the mathematics needed to analyze them is somewhat difficult, there have been some investigations that give heuristic ideas as to why random interleavers work [10]. Also it has been found that good interleavers for Turbo-codes are not hard to find [11]. This section will discuss a procedure for creating

a pseudo-random interleaver and also show why nonrandom block interleavers do not work well in Turbo-codes [10].

In this discussion nonrandom block interleavers will refer to a structure that reads bits in through the rows and out by the columns. Pseudo-random and random interleavers will be referred to when discussing block interleavers that read bits in through the rows but are read out using some other method.

Interleavers had been used prior to Turbo-codes in order to break up patterns of errors in bursty channels. To do this a nonrandom block interleaver would often be used. As mentioned, in this type of interleaver the bits would be read in by rows and read out by columns. In this way a sequence of the form

$$d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}$$

that was read into a four by four square matrix would be read out as

$$d_0, d_4, d_8, d_{12}, d_1, d_5, d_9, d_{13}, d_2, d_6, d_{10}, d_{14}, d_3, d_7, d_{11}, d_{15}$$

Although this sequence has been mixed up, it does not appear random to the channel. It can be seen that if a sequence is correlated then this interleaving procedure will change the correlation in a uniform way.

One procedure for creating a pseudo-random interleaver is given in [10]. The procedure is as follows: for an M*M memory (where M is a power of 2) the bits to be interleaved are read into a square matrix. If i and j are the addresses of the row and column for writing, respectively (with the first row and column being labeled row 0 and column 0 respectively) and i_r and j_r are the row and column for reading, respectively then the rule for reading is

$$i_r = (M/2 + 1)(i + j) \mod M$$

$$E = (i + j) \qquad \text{mod } 4$$

$$j_r = [P(E) * (j + 1)] - 1 \text{ mod } M$$

P(E) is a function of E that is relatively prime with M and is a function of the row address (i + j) mod 4. P(E) is given as follows:

$$P(0) = 17; \quad P(1) = 37; \quad P(2) = 19; \quad P(3) = 29;$$

$$P(4) = 41; \quad P(5) = 23; \quad P(6) = 13; \quad P(7) = 7;$$

(The only difference between this interleaver algorithm and the one used in our simulations is that the row address E is taken modulus 8 for a 256x256 interleaver). The sequence

$$d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}$$

will now be interleaved by this random interleaver. The output is given by

$$d_0, d_{13}, d_8, d_7, d_{12}, d_9, d_6, d_3, d_{10}, d_5, d_2, d_{15}, d_4, d_1, d_{14}, d_{11}$$

While the output from this interleaved pattern is not random per se, it does appear, at first glance, to be more "random" than the previous interleaver. However it is difficult to say how random an interleaver looks, especially for small blocks. Right now the only way to test whether an interleaver is random enough in a Turbo-code scheme is to run simulations with it. Deinterleaving is the inverse function of interleaving.

The reason that random interleavers work in Turbo-coding schemes is because they better "imitate" a random sequence to the channel. Since the goal of Turbo-codes is to create somewhat random codewords (as given by their output weight distribution) for a given input codeword, it can be seen that an output sequence that is only distantly related to its' input would be desirable. This means

that the output of RSCC2, p_{2k} from Figure 1.3 should be nearly independent from the sequence d_k.

Some analysis of the distance properties of nonrandom block interleaved sequences is given in Appendix 1. It is shown that nonrandom block interleavers can produce output sequences with high weights for input sequences with weights 2 or 3. But for input sequences of weight 4 this is not necessarily the case. This motivates the need for random interleavers.

Chapter 3

Soft Output Decoding

3.1 Overview of Soft Decoding

One of the factors that makes Turbo-codes work well was discussed in the previous chapter (approximating random codes). In this chapter it is shown how all the information from the channel is used. To do this soft output decoding is needed. This allows information to be passed from one decoder to another without loss of information. This requires a more complicated decoding system than is usually used with convolutional codes. Several algorithms have been proposed to generate the soft decisions. The Maximum-A-posteriori Probability (MAP) algorithm [12] is the optimal algorithm and will be discussed extensively in section 3.2. The Max-Log MAP [13], a simplification of the MAP algorithm, and the Soft Output Viterbi Algorithm (SOVA) [14] will also be discussed briefly. After an example of the MAP algorithm is given in section 3.3, the procedure for decoding Turbo-codes will be discussed in sections 3.4-3.6.

3.2 MAP Algorithm

The MAP algorithm is the optimal algorithm for the minimization of probability of bit error. The algorithm can also generate the probabilities of a bit

being 1 or 0. This is important because it is used to give a reliability value by using the log-likelihood value of a bit d_k, $\Lambda(d_k) = \log(Pr(d_k = 1)/Pr(d_k = 0))$. $Pr(d_k = i)$ is the probability that the decoded bit $d_k = i$ (i = 0, 1). This $\Lambda(d_k)$ is used to determine a soft output value. The sign of $\Lambda(d_k)$ determines whether the bit is a zero or one while the magnitude determines the reliability of the decoded bit. The log function is the natural logarithm (base e). The notation used in this derivation is as follows. R_{k1}^{k2} is the received sequence from states at time k1 to time k2. This is an encoded sequence that has been corrupted by noise. R_1^f is the entire received sequence from time 1 to time f. R_k is the received information at time unit k. S_k is the state of the encoder at time unit k. The value of the state at time k, S_k, is denoted by m, while the value of the state at time k-1, S_{k-1}, is denoted by m'. M is the total number of states. Hence m, m' = 0, 1, ..., M-1. It will be assumed that the encoder starts in the zero state.

As stated, the MAP algorithm gives the decision for every bit (i.e. 0 or 1) and a reliability value for the bit (higher reliability's being more reliable) given that all bits have been received. Mathematically this can be done by finding the probabilities of all state transitions. To do this we find

$$Pr\{S_{k-1} = m'; S_k = m \mid R_1^f\} \tag{3.2.1}$$

Since this form is more difficult to work with, it is converted to an equivalent form

$$Pr\{S_{k-1} = m'; S_k = m ; R_1^f\} / Pr\{R_1^f\} \tag{3.2.2}$$

The equivalence between (3.2.1) and (3.2.2) is given by Bayes rule. Since $Pr\{R_1^f\}$ is a constant for a given received sequence only the numerator of (3.2.2) needs to be found. The following notation is introduced to allow for ease of exposition.

$$\sigma_k(m,m') = Pr\{S_{k-1} = m'; S_k = m; R_1^f\} \tag{3.2.3}$$

The probability that a bit is zero or one can be determined from (3.2.3) as:

$$Pr\{d_k = i\} = \sum_{(m',m)\in A_k(i)} \sigma_k(m',m) \tag{3.2.4}$$

where $A_k(i)$ is the set of state transitions that cause the output i at time k.

The essential idea of decoding a bit is to split the probability that a state transition has occurred into three portions. The first part is developed from the received information prior to the time of the state transition. The second portion is formed from the received information after the state transition. The third portion is based on the received information at the time of the state transition. This can be expressed symbolically by introducing the following symbols.

$$\alpha_k(m) = Pr\{S_k = m, R_1^k\} \tag{3.2.5}$$

$$\beta_k(m) = Pr\{ R_{k+1}^f \mid S_k = m\} \tag{3.2.6}$$

$$\gamma_i(R_k,m,m') = Pr\{d_k = i, S_k = m, R_k \mid S_{k-1} = m'\} \tag{3.2.7}$$

Assuming that any state transition is described by a Markov process the value of $\sigma_k(m,m')$ is given by

$$\sigma_k(m,m') = \alpha_{k-1}(m) * \gamma_i(R_k, m, m') * \beta_k(m). \tag{3.2.8}$$

What (3.2.8) has shown is that the transition probability, $\sigma_k(m,m')$, can be broken up into those determined by the first k-1 transitions, the final (f - k) transitions and the transition determined at time k. This is important because the

transitions determined by $\alpha_k(m)$ and $\beta_k(m)$ can be calculated recursively with the following formulas [12]

$$\alpha_k(m) = \sum_{m'} \sum_{i=0}^{1} \gamma_i(R_k, m', m) * \alpha_{k-1}(m') \tag{3.2.9}$$

$$\beta_k(m) = \sum_{m'} \sum_{i=0}^{1} \gamma_i(R_{k+1}, m', m) * \beta_{k+1}(m') \tag{3.2.10}$$

Sometimes the $\gamma_i(R_k, m', m)$ values are not probability values but are distribution values (as will be seen in the example). The $\alpha_k(m)$ and $\beta_k(m)$ will then need to be normalized as follows.

$$\alpha_k(m) = \frac{\sum_{m'} \sum_{i=0}^{1} \gamma_i(R_k, m', m) * \alpha_{k-1}(m)}{\sum_{m} \sum_{m'} \sum_{i=0}^{1} \gamma_i(R_k, m', m) * \alpha_{k-1}(m')} \tag{3.2.11}$$

$$\beta_k(m) = \frac{\sum_{m'} \sum_{i=0}^{1} \gamma_i(R_{k+1}, m', m) * \beta_{k+1}(m')}{\sum_{m'} \sum_{m} \sum_{i=0}^{1} \gamma_i(R_{k+1}, m', m) * \alpha_k(m')} \tag{3.2.12}$$

Since the probabilities at the first state are known (the encoder begins in the zero state) the $\alpha_k(m)$ can be calculated recursively from 1 to f. As soon as all the $\alpha_k(m)$ are calculated, the $\beta_k(m)$ can be calculated from the final bit back to the first.

With this information $\sigma_k(m,m')$ can be determined. Knowing $\sigma_k(m,m')$ allows for the calculation of the log likelihood value, $\Lambda(d_k)$ which is

$$\Lambda(d_k) = \text{Log}\frac{\sum_{m}\sum_{m'} \gamma_1(R_k, m', m) * \beta_k(m) * \alpha_{k-1}(m')}{\sum_{m}\sum_{m'} \gamma_0(R_k, m', m) * \beta_k(m) * \alpha_{k-1}(m')} \qquad (3.2.13)$$

The essence of the algorithm is the use of probabilities to decode bits as opposed to the Viterbi algorithm, which uses metric values. The MAP algorithm, given the probabilities that the encoder is in a state at time zero, and the received channel values, calculates the probabilities of the encoder being in any state at any time recursively. All of these $\alpha_k(m)$ have to be stored for all values of k, and m (for the decoder that achieved BER 10^{-5} in [3] at E_b/N_o .7 dB, k and m are approximately 65000 and 16 respectively). A similar process is used to find the $\beta_k(m)$ after the entire sequence has been received. With these parameters the probabilities that the encoder was in any state can be derived and, along with the received channel value, is used to find the log likelihood probability.

3.3 Example of the MAP Algorithm

A simple example of the use of the MAP algorithm will now be given. The example will be done using a (5, 7) octal generator (Figure 2.2.4). For this generator, parity bit outputs and state transitions are given by the state transition diagram of Figure 2.2.5.

Ten random bits have been generated and the output of the encoder is

data bits $\{d_k\}$:	[0 0 1 0 0 0 1 0 0 0]
parity bits $\{p_k\}$:	[0 0 1 1 1 0 0 0 1 1]

Turbo-codes are usually punctured to increase the rate of the total code. When this is done certain bits are deleted according to a given rule. Every other parity bit is not sent in this example. The information is sent over an AWGN channel. To make decoding simpler to understand the received bits are transformed by a linear transformation by the modulator, therefore the inputs to the decoder are x_k = ((2*d_k - 1) + noise) and y_k = ((2*p_k −1) + noise). The bits which have been deleted by puncturing are inserted as zeros. This is what happened to our received data with noise variance of 1.6:

{x_k}:	[-1.04	-1.14	1.73	-1.48	-.02	-1.49	-.53	-1.71	-1.94	-2.37]
{y_k}:	[-.70	0	-.23	0	1.78	0	-.59	0	1.53	0]

Errors have occurred in the 7th column of the systematic bits and the third column of the parity bits.

The decoding procedure can now be implemented. The first step is to calculate $\alpha_k(m) = Pr\{S_k = m, R_1^k\}$ for all states and times. Knowing that the encoder began in the zero state allows us to know that $\alpha_0(0) = 1$ while $\alpha_0(m) = 0$ for m not equal to 0. From this and the received values the rest of the $\alpha_k(m)$ can be calculated. They are

k =	0	1	2	3	4	5	6	7	8	9	10
state 3	[0	0	.02	.03	.80	.12	.03	.33	.19	.63	.02]
state 2	[0	.02	.07	.83	.11	.03	.77	.18	.36	.02	.32]
state 1	[0	0	.00	.11	.06	.79	.12	.36	.32	.33	.63]
state 0	[1	.98	.91	.03	.03	.06	.08	.13	.14	.02	.02]

From column k = 0 to column k = 1 the probability that the encoder went to state 0 at the time after the first bit had arrived is the sum of the probabilities of transitioning to state 0 from any previous state that could possibly come to state 0. The only two states that can arrive at state 0 (from the state diagram, Figure 2.2.5) are states zero and one. Therefore the probability of being in state 0 at time 1 (after the first systematic bit and parity bit arrive) is

$[Pr\{S_0=0,R_1^1\} * Pr\{d_1=0,S_1=0,R_1 \mid S_0=0\}] + [Pr\{S_0=1,R_1^1\} * Pr\{d_1=1,S_1=0,R_1 \mid S_0=1\}] = \alpha_0(0) * \gamma_0(R_1,0,0) + \alpha_0(1) * \gamma_1(R_1,0,1).$

Since the probability of being in state 1 at time zero ($\alpha_0(1)$) is zero this leaves only the first portion ($\alpha_0(0) * \gamma_0(R_1,0,0)$) to be considered. Using the fact that the information was sent over a Gaussian channel $\gamma_0(R_1,0,0)$ is calculated by the following formula:

$\gamma_i(R_k,m,m') = \text{constant} * \exp[-(x_k - b^s(i,m',m))^2/N_o] * \exp[-(y_k - b^p(i,m',m))^2/N_o]$

(3.3.1) for each pair of states which allow a transition.. I chose to leave the constant as one and normalize the α and β values after α and β are calculated at any state (this is done by equation 3.2.11 automatically). $b^s(i,m',m)$ is the systematic bit output at the modulator when there is a transition from state m' to state m. Likewise $b^p(i,m',m)$ is the parity bit output from the modulator when there is a transition from state m' to state m. As an example, if it is assumed that the encoder has gone from state 3 to state 1 at time k, then d_k would be 0 and p_k would be 1. Since the modulator transforms these outputs by the linear transformation given above $b^s(i = 0,m' = 3, m = 1) = -1$ while $b^p(i = 0,m' = 3,m =$

1) = 1. N_o is the noise variance (in this case the noise variance is 1.6). x_1 is the systematic bit received at time k = 1 which is -1.04. y_1 is the parity bit that has been received at time 1. This is -.70. This means the transition probability ($\gamma_0(R_1,0,0)$) is

$\exp[-(-1.04 - (-1))^2 /N_0]^* \exp[-(-.7 - (-1))^2 / N_0] = .99 * .95 = .98$

Similarly the transition from state 0 to state 2 ($\gamma_1(R_1,2, 0)$) is

$\exp[-(-1.04 - (+1))^2 /N_0]^* \exp[-(-.7 - (+1))^2 / N_0] = .09 * .19 = .02$

The rest of the $\alpha_k(m)$ can be calculated in the same way.

$\beta_k(m)$ are calculated in a similar way. However after the final bit has arrived the final state of the encoder is not known. For this reason $\beta_{10}(m)$ can either be initialized as $\alpha_{10}(m)$ or given equal weighting as (1/M). I have chosen to use the former method. $\beta_k(m)$ is then

k =	0	1	2	3	4	5	6	7	8	9	10
state 3	[.23	0.17	.10	.07	.26	.44	.23	.01	.34	.63	.02]
state 2	[.16	.10	.58	.26	.44	.24	.26	.33	.66	.03	.32]
state 1	[.20	.51	.18	.43	.06	.26	.44	.64	.00	.33	.63]
state 0	[.41	.22	.19	.24	.23	.06	.06	.02	.00	.19	.02]

Using $\beta_9(0)$ for an example of how to calculate the $\beta_k(m)$ will now be done. $\beta_9(0)$ is the probability that the sequence after time 9 (i.e. the last bit) would arrive given that the state is known to be state zero at that time. For this case we know that the sequence could only go to state 0 or to state 2. $\beta_9(0)$ is

$(\beta_{10}(0) * \gamma_0(R_{10},0,0)) + (\beta_{10}(2) * \gamma_1(R_{10},2,0))$.

To calculate $\gamma_0(R_{10},0,0)$ and $\gamma_1(R_{10},2,0)$ we use the same method as before.

$$\gamma_0(R_{10},0,0) = \exp[-(-2.37 - (-1))^2/N_0] * \exp[-(0 - (-1))^2/N_0] = .33 * .55$$

$$\gamma_1(R_{10},0,2) = \exp[-(-2.37 - 1)^2/N_0] * \exp[-(0 - 1)^2/N_0] = .001 * .55$$

So that $\beta_9(0)$ is .02 * .18+ .32 * .005. At this point you may notice that the sum of these does not come to .19. This is because the γ have not been normalized. This is why after all β_9 have been calculated in the way that was just described the values are normalized (this is from 3.2.12). Continuing this way through for each of the received bits generates all the values of $\beta_k(m)$ for all k and m although β_k can be discarded after it has been used for generating the output value at time k if lack of memory is a problem.

This information (α_k and β_k) has been generated to obtain the probability values of each transition so that the probabilities that each bit was either a 1 or 0 can be calculated using (3.2.4). Because we only need the ratio of the probabilities to generate the log likelihood value we will not need to find the probability per se. As an example I will find logarithm of the ratio of the probability the first bit was a one to the probability the first bit was a zero.

The only transitions that can occur with the arrival of the first bit are the transition from state 0 to state 0 (which generates a 0) and the transition from state 0 to state 2 (which generates a 1). Therefore the probability that this output is a one is given by $\sigma_1(0,2) = \alpha_0(0) * \gamma_1(R_1,0,2) * \beta_1(2)$. The probability that the output is zero is $\sigma_0(0,0) = \alpha_0(0) * \gamma_0(R_1,0,0) * \beta_1(0)$. Taking the ratio of these values and then the logarithm gives a value of -4.67. Since the sign of the bit is

negative it has been decoded correctly. The reliability value of 4.67 can give information about the actual probability of a bit being 1 or 0 if that is desired. Here is the complete decoded sequence.

[-4.67 -2.2 5.13 -6.15 -3.77 -6.15 1.66 -6.0 -7.1 -7.8]

As can seen by comparing this with the original sequence the sequence has been decoded correctly and the certainty of each bit can be measured relative to the others.

The disadvantages of this system are now apparent. There is a very large amount of memory needed for decoding (storage of α). Also the complexity of the decoder is apparent from the equations needed to calculate the parameters (large numbers of multiplies and adds).

The Soft Output Viterbi Algorithm [14] and the Log-MAP algorithm [13] will now be discussed breifly.

The SOVA is generally similar to the standard Viterbi Algorithm in that it compares metric values at each node of the trellis to decide which path is the maximum likelihood path (hence the minimum metric). The SOVA at each node will also compare the path with the minimum valued metric with the path with the second best metric, and use that information to update a reliability value of all bits which are not the same in the two paths. This requires only comparisons of metrics and table lookups, which are less time consuming than the MAP algorithm. Also only one pass through the information is required as opposed to the MAP algorithm, which requires a forward and a backward pass.

The Max-Log MAP algorithm is a simplification of the MAP algorithm that results from taking the log of the probability distribution of the transitions (γ) and replacing them by approximations. This algorithm is a better approximation than the SOVA but not as good as the MAP algorithm.

3.4 Decoding of Turbo-codes

The general scheme for the decoding is shown in Figure 3.4.1. As soon as the sequence is received the parity bits are demultiplexed. A soft output decoder is used with the inputs being the systematic information and the output of the first RSCC (d_k and p_{1k} after modulation and having noise added, producing x_k and y_{1k} respectively). The output of this decoder is an estimate of the information sequence and will be called $\Lambda 1$. This estimate is then interleaved according to the pseudorandom interleaver that was used at the encoding stage. This allows the new estimate $\Lambda 1$ to be used along with the parity bits from the second recursive convolutional code in a second soft output decoder. This produces a new estimate of the (interleaved) information bits. However, because the first decoder did not use all the information available (specifically it did not use the second set of parity bits, y_{2k}) the performance can be improved by adding a feedback path from the output of the second soft output decoder to the first decoder as shown in Figure 3.4.2.

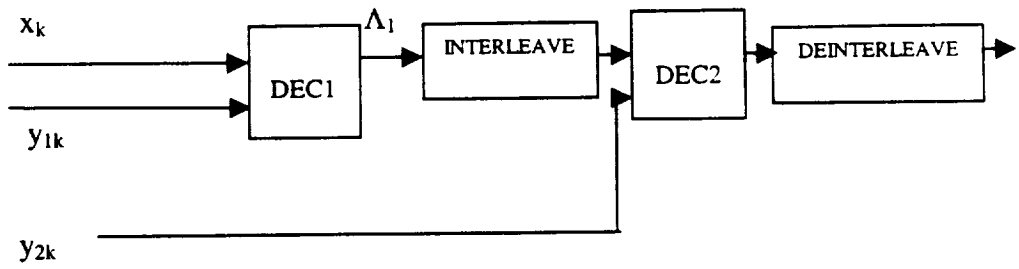


Figure 3.4.1 The General (Suboptimal) Turbo-decoding Structure

One important consideration when feeding information from the second decoder back to the first is that the information sent back to the DEC1 must be information that is independent of the information generated by DEC1 in the first place. It should be information that was generated by y_{2k}. If the information sent back to DEC1 was already generated by DEC1 there would be positive feedback and the decoding could become unstable. There are two methods for feeding back information. The first method is from [4] and the second from [5]. The first method uses slightly different decoding stuctures for DEC1 and DEC2. The second method has the same decoding structure for both decoding blocks.

3.5 Method 1 for the Decoding of Turbo-codes

The first method is achieved by considering the output of the first MAP decoder (DEC1) which is

$$\Lambda 1(d_k) = \log \frac{\sum_{m}\sum_{m'} \gamma_1(R_k, m', m) * \beta_k(m) * \alpha_{k-1}(m')}{\sum_{m}\sum_{m'} \gamma_0(R_k, m', m) * \beta_k(m) * \alpha_{k-1}(m')} \qquad (3.5.1)$$

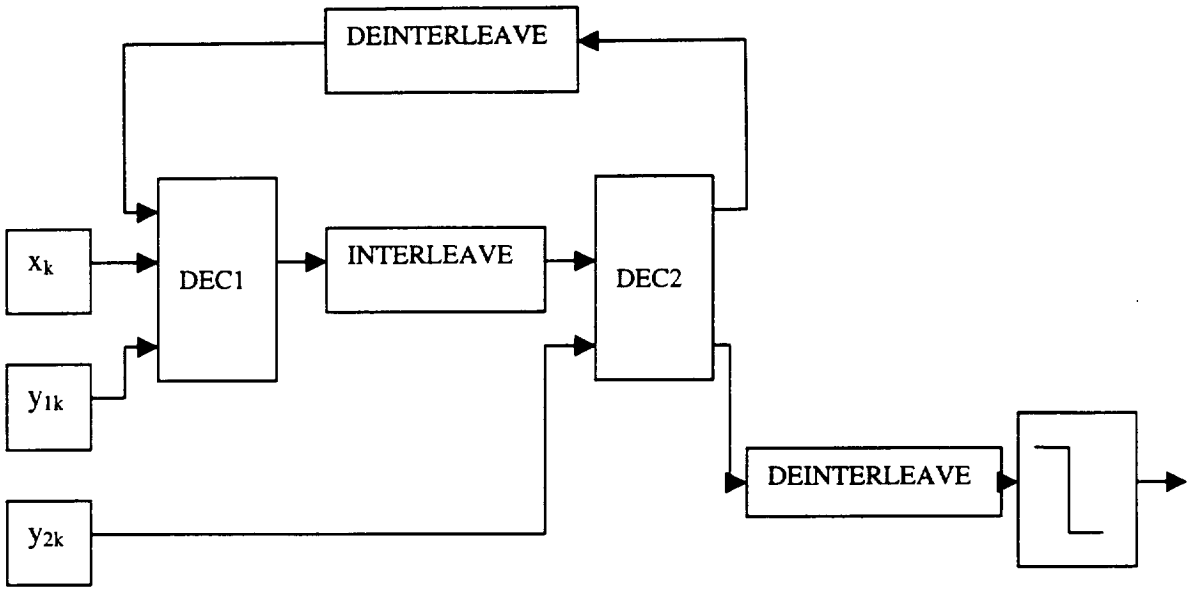


Figure 3.4.2 One Optimal Decoding Scheme

In the first decoder (in Fig 3.4.2), the sequence R_k consists of the channel values x_k and y_{1k}. Because the encoder is systematic the transition probability $p(x_k|d_k=i,S_k=m,S_{k-1}= m')$ in $\gamma_i(R_k, m', m)$ (from 3.3.1) is independent of the state value of the encoder. Being independent of states means that the summations over m and m' (the current and previous states) will have no effect on it. What this means is that this can be factored out in the numerator and denominator of (3.5.1). Now

$$\Lambda 1(d_k) = \log \frac{\sum_m \sum_{m'} \gamma_1(y_{1k}, m', m) * \beta_k(m) * \alpha_{k-1}(m')}{\sum_m \sum_{m'} \gamma_0(y_{1k}, m', m) * \beta_k(m) * \alpha_{k-1}(m')} + \log \frac{p(x_k \mid d_k = 1)}{p(x_k \mid d_k = 0)} \quad (3.5.2)$$

This can be expressed more concisely as

$$\Lambda 1(d_k) = W_{1k} + (2/\sigma^2) * x_k \quad (3.5.3)$$

where W_{1k} is the logarithm of the quotient of the summations in (3.5.2). Notice that the γ term in (3.5.2) depends on y_{1k}, not the systematic term x_k. So $W_{1k} = \{\Lambda 1(d_k) \mid x_k = 0\}$. The α and β terms are still built with systematic terms as well as the parity information. The x_k is multiplied by $(2/\sigma^2)$ in (3.5.2) because x_k is Gaussian with mean +/- 1 and variance σ^2. This shows that W_{1k} is the information produced using the structure of RSCC1 (it is the information output from DEC1 that depends on memory).

Now $\Lambda 1(d_k)$ will act as the systematic information in the input to second decoder. The output of the second decoder will be

$$\Lambda 2(d_k) = W_{2k} + f(\Lambda 1(d_k)) \quad (3.5.4)$$

with W_{2k} defined similarly to W_{1k} in (3.5.3). f(*) is some function of $\Lambda 1(d_k)$. W_{2k} is a function of the sequence y_{2k} and uses a priori information from the sequence $\{\Lambda 1(d_k)\}$. Because of interleaving between decoders W_{2k} is only weakly correlated with x_k and y_{1k} (the hope is that it is independent of $\{\Lambda 1(d_k)\}$). This means that a new decoding process can take place with x_k, y_{1k} and using W_{2k} as a

priori information in DEC1 after the first decoding iteration has occurred. [4] sets $z_k = W_{2k}$ and assumes that it can be approximated by a Gaussian random variable with a variance of σ_z^2 (the variance of σ_z^2 must be estimated at every iteration). After the first iteration the output of DEC1 will be determined by x_k, y_{1k}, and z_k and will be equal to

$$\Lambda 1(d_k) = W_{1k} + (2/\sigma^2) * x_k + (2/\sigma_z^2) * z_k \tag{3.5.5}$$

In (3.5.5) the W_{1k} term has used x_k, y_{1k}, and z_k to build α and β (as the a priori information). Now since z_k has been built by DEC2 it cannot be reused as input information for DEC2. This means that $(2/\sigma_z^2) * z_k$ must be subtracted off after decoding has been done. The decoder structure is shown in Figure 3.4.2.

3.6 Method 2 for the Decoding of Turbo-codes

The first method of decoding Turbo-codes involved passing the information to the second decoder that was obtained from both the systematic sequence and the first parity sequence. The second method involves sending the systematic sequence directly (after interleaving) and also using the a priori information directly. The output of either of the MAP decoders in this method is split into three parts in a manner similar to (3.5.5). The result is

$$\Lambda(d_k) = \log \frac{\sum_{m}\sum_{m'} \gamma_1(y_k, m', m) * \beta_k(m) * \alpha_{k-1}(m')}{\sum_{m}\sum_{m'} \gamma_0(y_k, m', m) * \beta_k(m) * \alpha_{k-1}(m')} + \log \frac{p(x_k \mid d_k = 1)}{p(x_k \mid d_k = 0)} + L(d_k)$$

(3.6.1)

$L(d_k)$ is set to zero for the first iteration of the first decoder. After that $L(d_k)$ is the a priori information generated by the previous decoder (i.e. the log of the summation of products). This means $L(d_k)$ is generated by the parity information from the previous decoder. The systematic information is interleaved (or deinterleaved) and passed to the next decoder separately. The use of $L(d_k)$ in decoding comes in considering the value of

$$\gamma_i(x_k, y_{1k}, L(d_k), m', m) = \Pr(x_k | d_k = i, S_k = m, S_{k-1} = m') * \Pr(y_k | d_k = i, S_k = m, S_{k-1} = m') * \Pr(d_k = i | S_k = m, S_{k-1} = m') * \Pr\{S_k = m | S_{k-1} = m', L(d_k)\} \quad (3.6.2)$$

$\Pr(d_k = i | S_k = m, S_{k-1} = m')$ is either zero or one depending on whether there is an output i associated with a state transition from m' to m. With $\Pr\{S_k = m | S_{k-1} = m', L(d_k)\}$ the use is made of the information from the previous decoder. $L(d_k)$ was generated as the log of the summation products from the previous decoder. This means that $L(d_k)$ is equal to $\log(\Pr(d_k=1)/\Pr(d_k=0))$ using information generated by the previous decoder. By exponentiating $L(d_k)$ and using the fact that $\Pr(d_k=1) + \Pr(d_k=0) = 1$ its' value can be given as follows

$$\Pr\{S_k = m | S_{k-1} = m', L(d_k)\} = \frac{e^{L(d_k)}}{1 + e^{L(d_k)}} \quad (3.6.3)$$

if the state transition from m' to m determines a 1; and

$$\Pr\{S_k = m | S_{k-1} = m', L(d_k)\} = 1 - \frac{e^{L(d_k)}}{1 + e^{L(d_k)}} \tag{3.6.4}$$

if the state transition from m' to m determines a 0. In plain language what this gives is the probability that a bit is one or zero depending on the information generated from the previous decoder. The decoding scheme used in this case is shown in Figure 3.6.1. The advantage of this method is that no variance estimate is required. For this reason I used this decoding method in my decoder.

With either method the number of iterations can be determined by knowing the number of iteration needed to achieve the BER required.

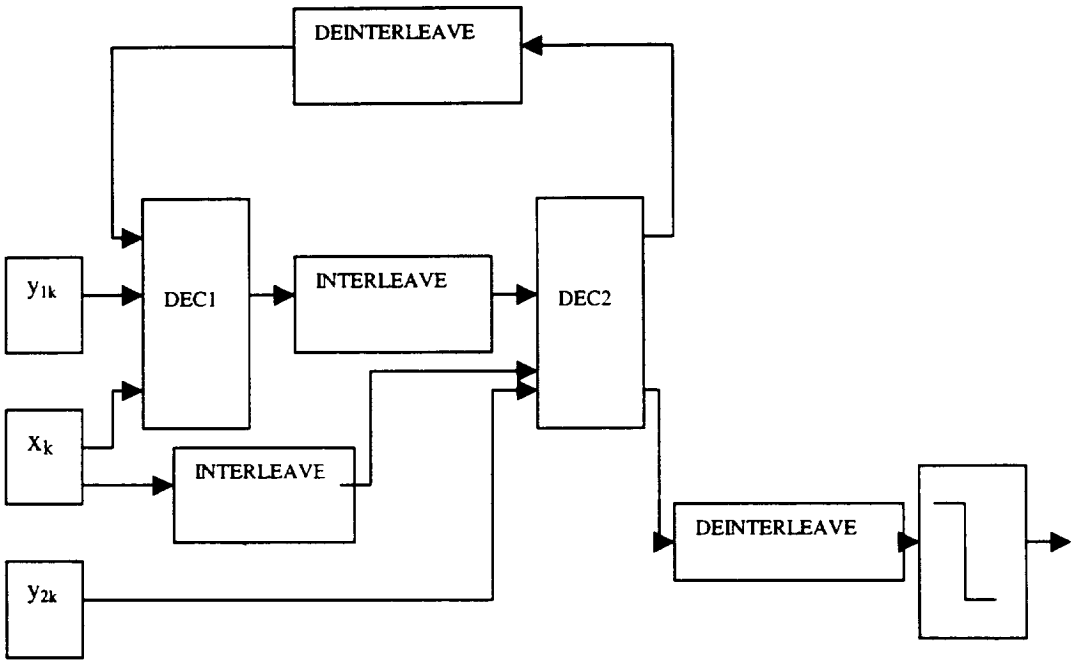


Figure 3.6.1 The Second Optimal Decoding Method

Chapter 4

Results

In Turbo-coding there are several components (i.e. random interleavers, RSCC's, and decoders), each with different parameters. Even separately these components can be difficult to analyze. Several papers have helped in the separate analysis of both the interleavers and the RSCC's [8],[11]. One of the important results claimed in [11] is that the interleaver size is the most important factor in determining the performance of Turbo-codes and that BER performance is inversely proportional to the size of the interleaver for large enough, random enough interleavers. This is important because it allows for testing of other components somewhat independently of the interleaver. For this reason only one interleaver was tested. The implementation of the interleaver is given in section 4.1.

The MAP decoding algorithm is used in the simulation. The reason for this is that this will give the best possible performance. Also simpler, more memory efficient versions of the MAP algorithm are becoming available [15]. The second decoding method, described in 3.6, was used because the variance at the output of the second decoder did not need to be estimated.

In the simulations there were no zero bits tacked on to the end of each block and the final state of the encoder was unknown. This did not seem to degrade performance. Most of the decoding was done for a maximum of 18 iterations. This is because it was done this way in [4] and is considered a benchmark for my research.

The first requirement was to test the memory 4 generators to determine which produce the best BER curves. Memory 4 codes are generally used because they can generate very good performance. Higher memory generators do not generally add much performance gain and the decoding process is much more complex (remember that decoding complexity and memory requirements increase by more than a factor of 2 for every memory element added). Section 4.2 will give the simulation results of memory 4 RSCC's concatenated in a Turbo-code scheme.

The next consideration is the reduction of decoder complexity while maintaining good performance levels by reducing the memory for RSCC1 and using the standard memory four RSCC2. Because the decoding complexity of each (MAP) decoder grows exponentially with encoder memory the complexity of a Turbo-code with memory 4 RSCC2 and memory 3 RSCC1 is approximately 75% of the complexity (ignoring the interleaving and deinterleaving operations, which in any case are just reading and writing operations). For memory 4 RSCC2 and memory 2 RSCC1 the complexity is about 5/8 of the standard. This analysis assumes the same number of iterations for both decoding structures being compared. If the performance is not degraded significantly then the savings in

decoding complexity can be a significant factor. Section 4.3 will give the simulation results of the concatenation of 2 different RSCC's, one with a smaller memory.

The next idea that was considered was observing the effect of reducing the rate of the Turbo-codes by sending all parity bits and rejecting puncturing. Using lower rate codes can result in power savings at the expense of extra bandwidth. In cases when power is limited it is important to know how well Turbo-codes can perform without puncturing. Section 4.4 will give the simulation results of a rate 1/3 Turbo-code.

Section 4.5 will give the simulation results of a Turbo-code where noise variance was measured inaccurately. This is done because the MAP algorithm requires an estimate of the noise variance. If Turbo-codes were to decode poorly because of a small error in the noise variance estimate then they would be of almost no practical use. These simulation results will show how much performance is degraded by some poor estimates.

Of course this research has not closed the book on Turbo-codes. Section 4.6 will give ideas for further research.

4.1 Interleaver Implementation

The interleaver algorithm used in this simulation is implemented as follows [10]: for an M*M memory (where M is 256, hence there are 65536 bits/block) the bits to be interleaved are read into a square matrix. If i and j are

the addresses of the line and column for writing, respectively (with the first line and column being labeled line 0 and column 0 respectively) and i_r and j_r are the line and column for reading respectively, then the rule for reading is

$$i_r = (M/2 + 1)(i + j) \quad \text{mod } M$$

$$E = (i + j) \quad \text{mod } 8$$

$$j_r = [P(E) * (j + 1)] - 1 \quad \text{mod } M$$

where P(E) is a function of E that is relatively prime with M and is a function of the line address (i + j) mod 8

P(E) is given as follows:

P(0) = 17; P(1) = 37; P(2) = 19; P(3) = 29;

P(4) = 41; P(5) = 23; P(6) = 13; P(7) = 7;

4.2 Memory 4 Generators

5 different generators have been considered. The first is a 27_31 encoder which is shown in Figure 4.2.1 with results shown in Figure 4.2.2. The 27_31 circuit had the best BER curves after 18 iterations. For this reason iterations were continued beyond 18 to determine how well it would perform. This code decoded below BER 10^{-5} at .65 dB after 28 iterations. Although the number of iterations is very large, it may be worth it if power is a constraint in a given application. The BER of the 27_31 code after 18 iterations was used as the reference against other Turbo-encoders tested. The dashed line in the BER curves is the result of the 27_31 after 18 iterations.

Next a 23_35 encoder was tested [8]. This encoder is shown in Figure 4.2.3 with results shown in Figure 4.2.4. This RSCC has the best distance properties. It can be seen that BER curves are not as good as the 27_31 code after 18 iterations. However the BER after 1 and 2 iterations is better than the 27_31 code. What this seems to show is that this encoder may perform better asymtotically at higher E_b/N_o.

The next two generators were given in [9]. This required the FB portion of the encoding circuit to be a primitive polynomial while the FF portion of the circuit should be chosen to minimize BER using certain criterion. Two generator polynomials given in that paper were 31_27 and 31_33 generators. Of these two, only results of the 31_27 encoder, which is shown in Figure 4.2.5 with results shown in Figure 4.2.6, are given. This is because the generators were obtained by the same method and the results are similar. The BER curves of these circuits are very similar to the BER curves obtained by the 23_35 circuit. Both are approximately .1 dB away from the 27_31 circuit after 18 iterations at BER 10^{-5} and both of them have steeper dropoffs at higher E_b/N_o.

Finally the original circuit used in [4] which was a 37_21 circuit, shown in Figure 4.2.7, was tested. Results are shown in Figure 4.2.8. This circuit performed better than any circuit at low E_b/N_o after many iterations with the exception of the 27_31 encoder.

The 37_21 RSCC and the 27_31 RSCC's were chosen arbitrarily while the other RSCC's were chosen based on analytical techniques. The 23_35 RSCC was determined based on distance properties and not on how it would perform in a

Turbo-code scheme. It was not necessarily expected to perform well as a Turbo-code. However the 31_33 RSCC and the 31_27 RSCC were designed to be optimal in a Turbo-code scheme. What this analysis has shown is that the RSCC's that are selected based on the analytical techniques may not perform the best at very low E_b/N_o. From the results of the simulations completed here it appears that the best memory 4 encoder obtained so far is the 27_31 RSCC but this does not mean that better RSCC's will not be found. Better analytical methods need to be found for generating good RSCC's to remove any doubt as to which RSCC will perform best.

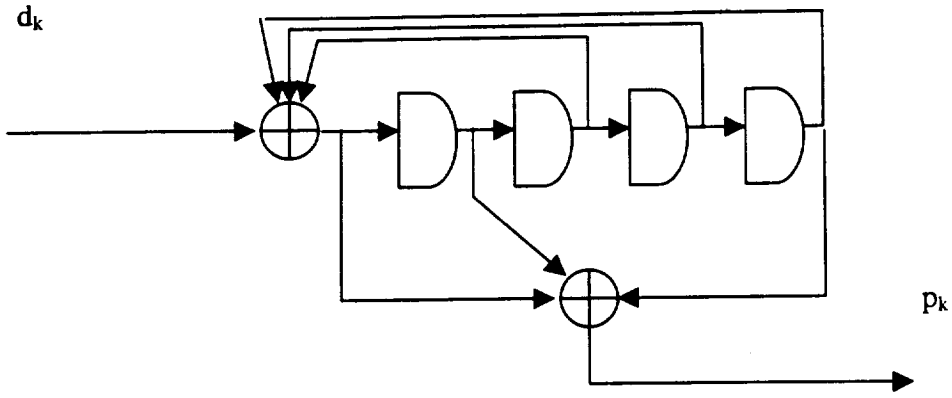


Figure 4.2.1 A 27_31 Generator Circuit

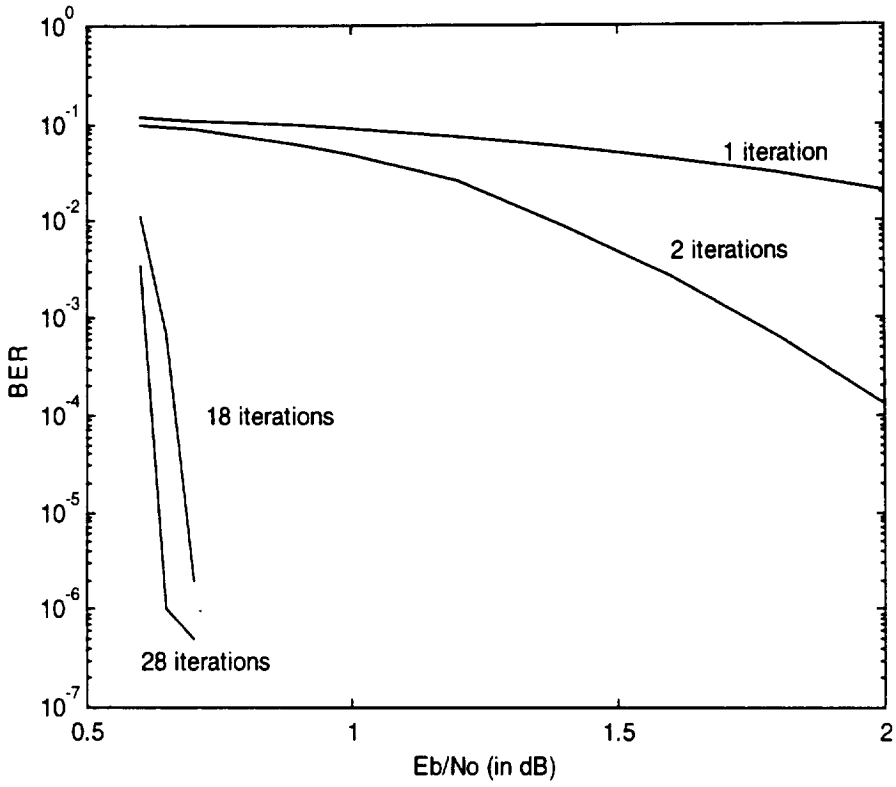


Figure 4.2.2 Performance for 27_31 Code Turbo-code Scheme

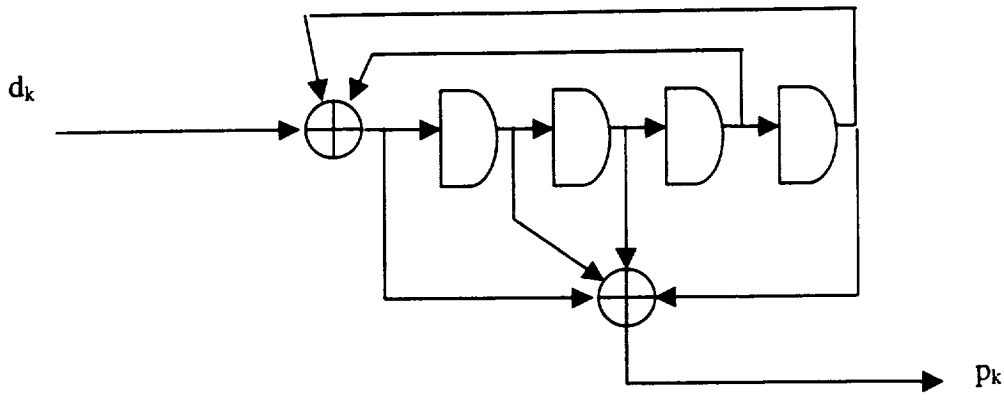


Figure 4.2.3 23_35 Generator Circuit

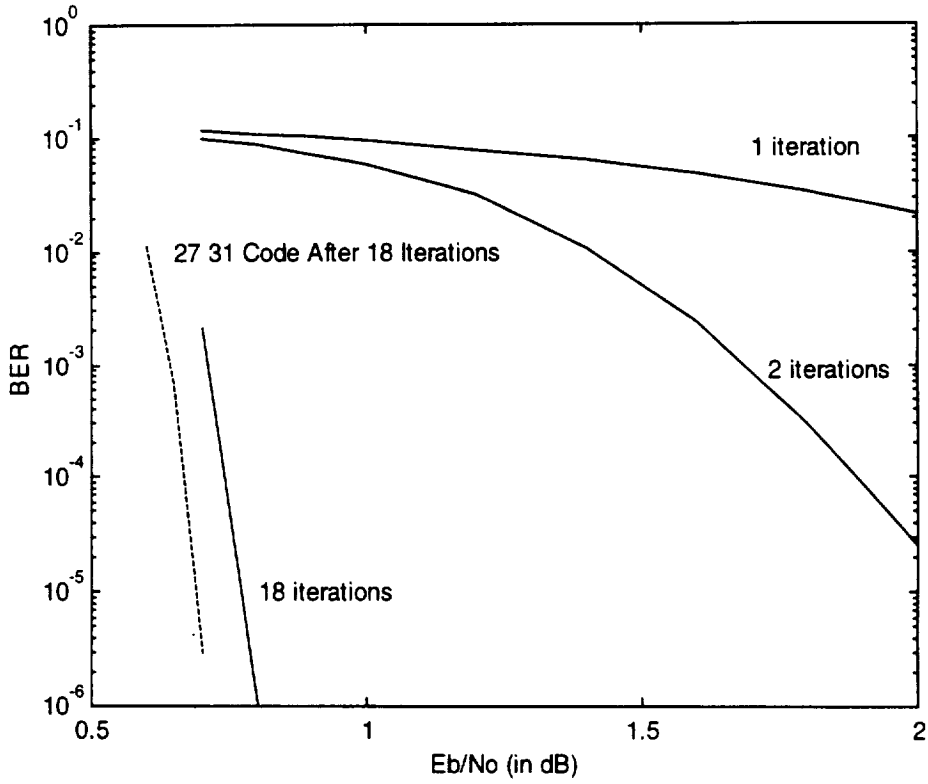


Figure 4.2.4 Performance for 23_35 Code Turbo-code Scheme

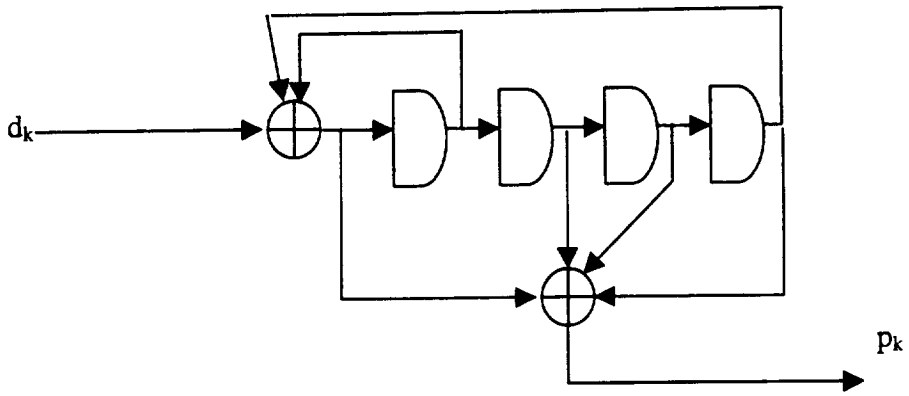


Figure 4.2.5 A 31_27 Generator Circuit

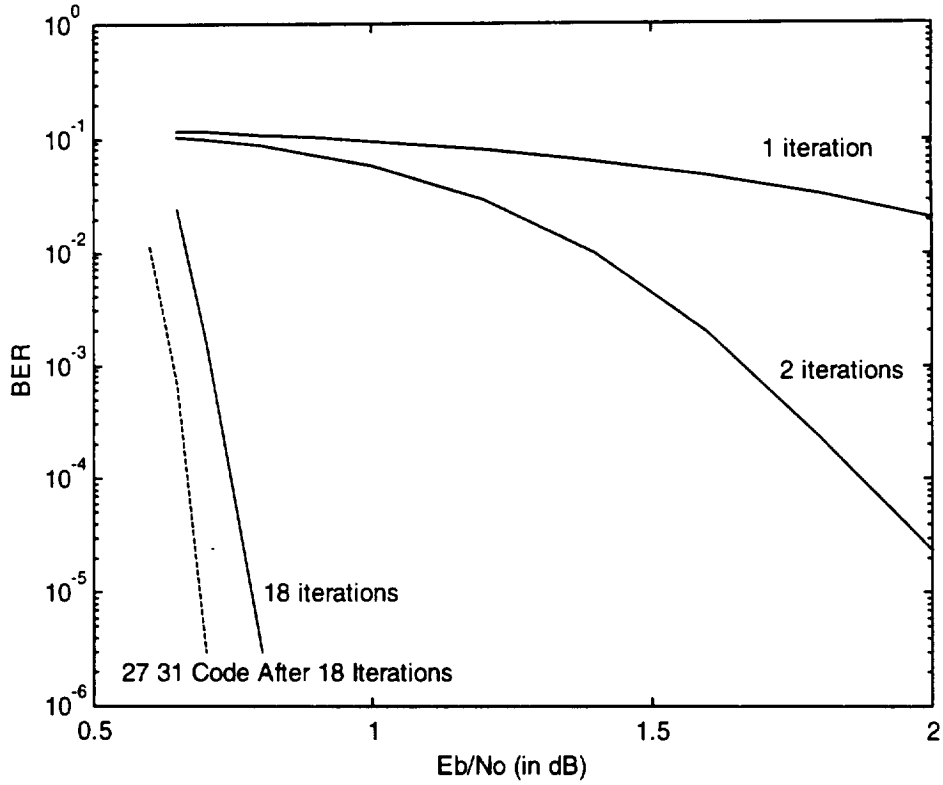


Figure 4.2.6 Performance for 31_27 Code Turbo-code Scheme

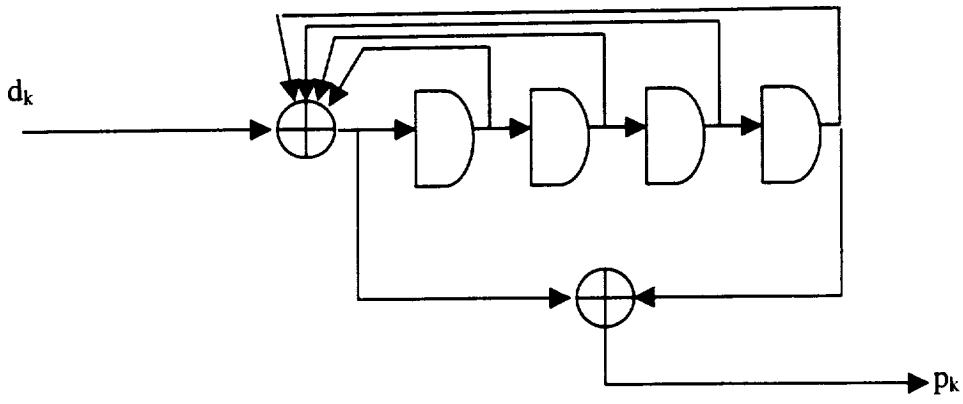


Figure 4.2.7 A 37_21 Generator Circuit

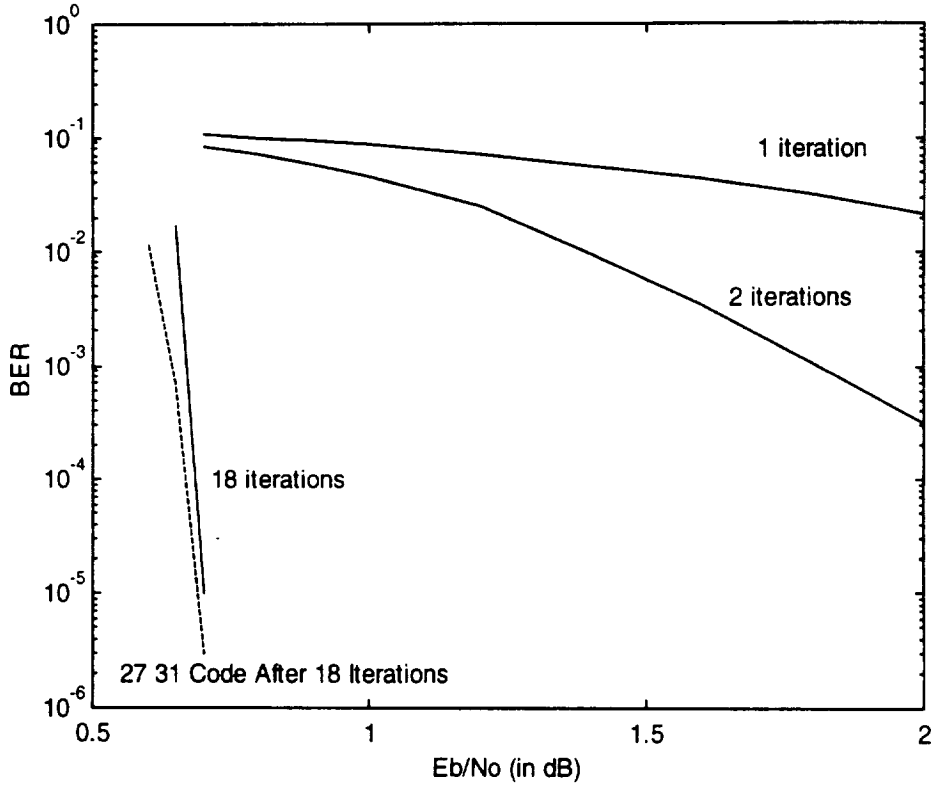


Figure 4.2.8 Performance for 37_21 Code Turbo-code Scheme

4.3 Lowering Decoder Complexity

The next consideration is the reduction of decoder complexity while maintaining good performance levels by reducing the memory for RSCC1 and using the standard memory four RSCC2. It was shown in [11] that RSCC1 should be the encoder with reduced memory.

The smaller memory generators that were used were obtained from [8]. The 7_5 circuit is shown in Figure 4.3.1. The results of the 7_5 RSCC1 concatenated with the 27_31 RSCC2 are shown in Figure 4.3.2. A closeup of these results is shown in Figure 4.3.3 to highlight the differences between the curves. The 15_17 circuit is shown in Figure 4.3.4. The results of the 15_17 RSCC1 concatenated with the 27_31 RSCC2 are shown in Figure 4.3.5. A closeup of these results is shown in Figure 4.3.6.

As can be seen in the Figures the loss in coding gain is not very much. For decoding at 10^{-5} the loss in power is only .12 dB and .10 dB for memory 2 and 3 RSCC1 respectively concatenated with the memory 4 RSCC2. At 10^{-4} the difference was even less pronounced, with losses of only .07 and .04 dB. In many cases it seems this would be a fair tradeoff given the reduced decoding complexity. If decoding complexity is a problem the smaller memory should be used since the difference in power savings is not significant between the two.

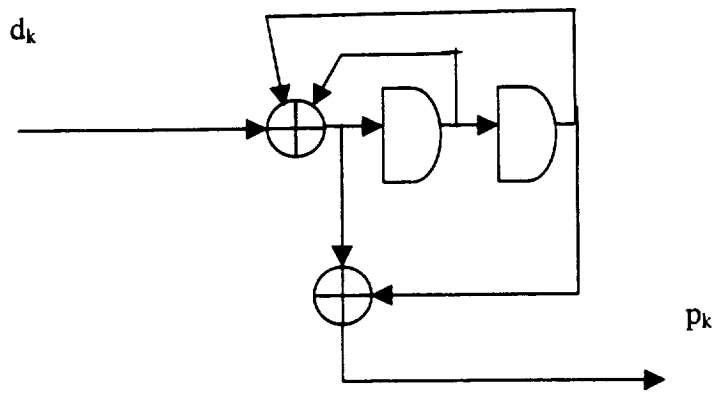


Figure 4.3.1 A 7_5 Generator Circuit

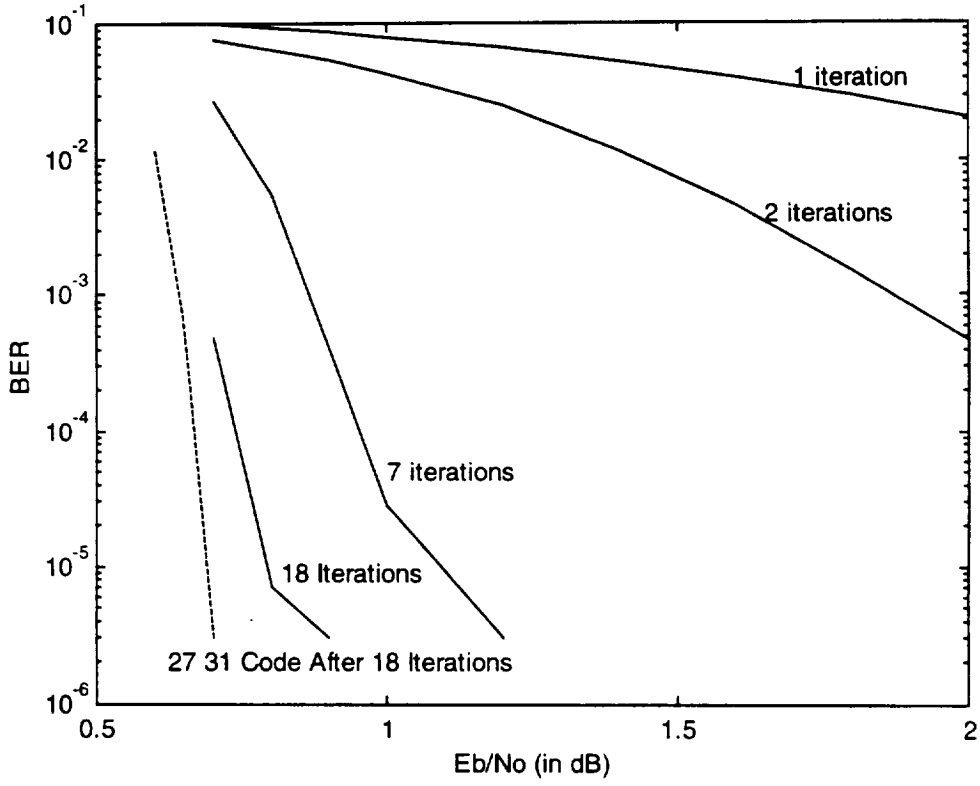


Figure 4.3.2 BER Curve for a Concatenated 7_5 Generating Circuit

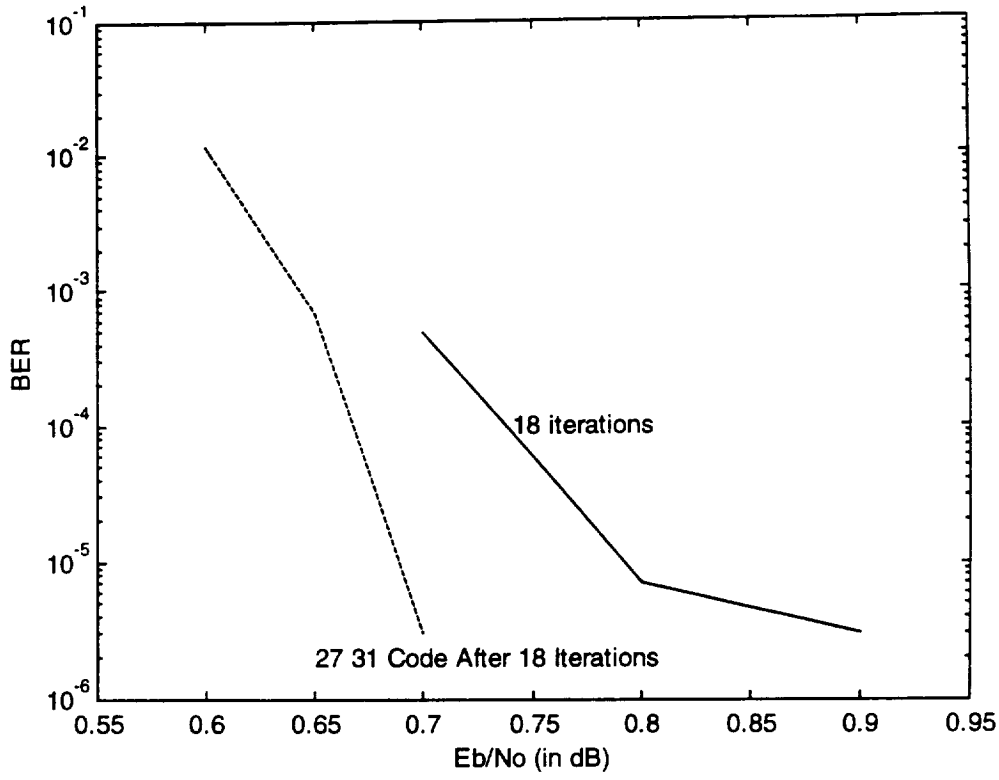


Figure 4.3.3 Closeup BER Curve for a Concatenated 7_5 Generating Circuit

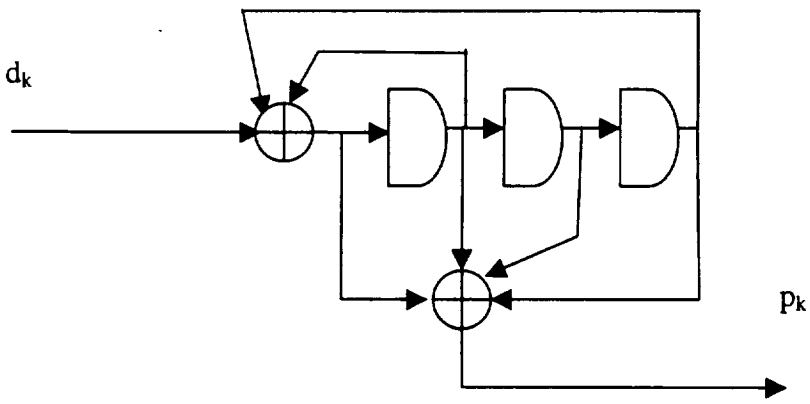


Figure 4.3.4 A 15_17 encoder

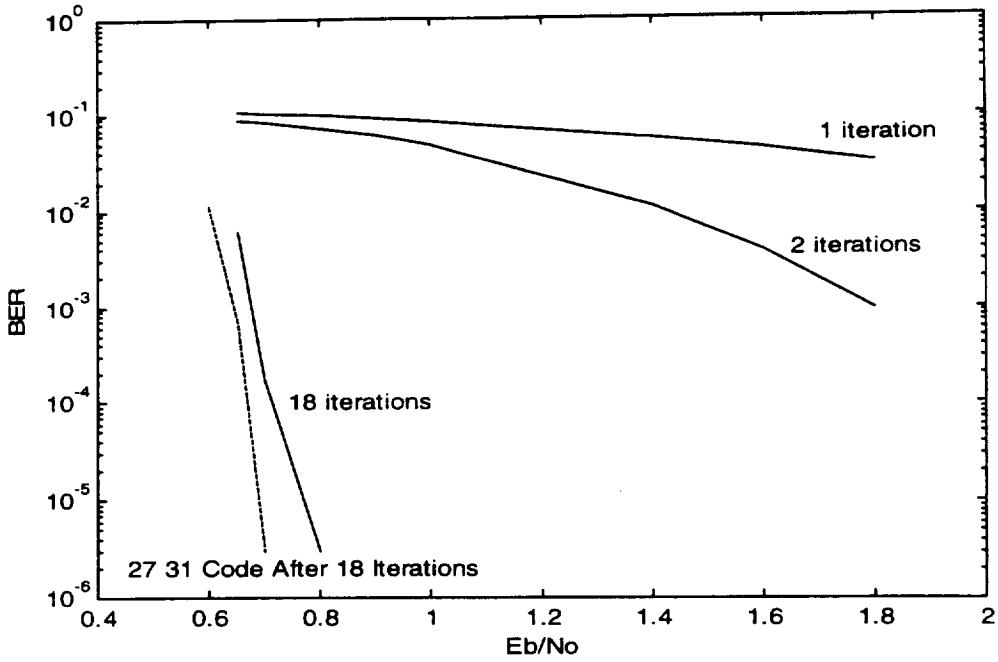


Figure 4.3.5 BER Curve for a Concatenated 15_17 Generating Circuit

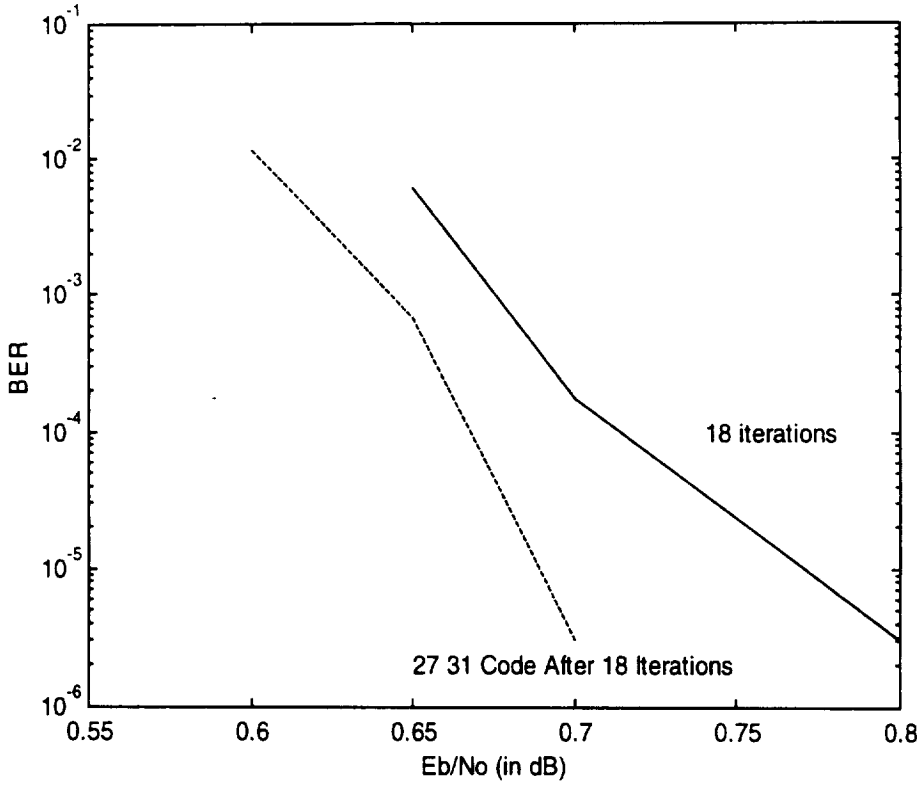


Figure 4.3.6 Closeup BER Curve for a Concatenated 15_17 Generating Circuit

4.4 Lower Rate Turbo-codes

It was suggested in [11] that unpunctured Turbo-codes might not perform as well as punctured Turbo-codes. To determine the validity of these claims simulations were done on an overall rate 1/3 turbo code with results shown in Figure 4.4. Since the Shannon limit at rate 1/3 is -.55 dB the results are very good. They decode at only .65 dB away from the Shannon limit in only 14 iterations. This is the same distance away from the Shannon limit as the punctured codes after 28 iterations. The tradeoff is increased bandwidth requirements which may not be a problem in some applications.

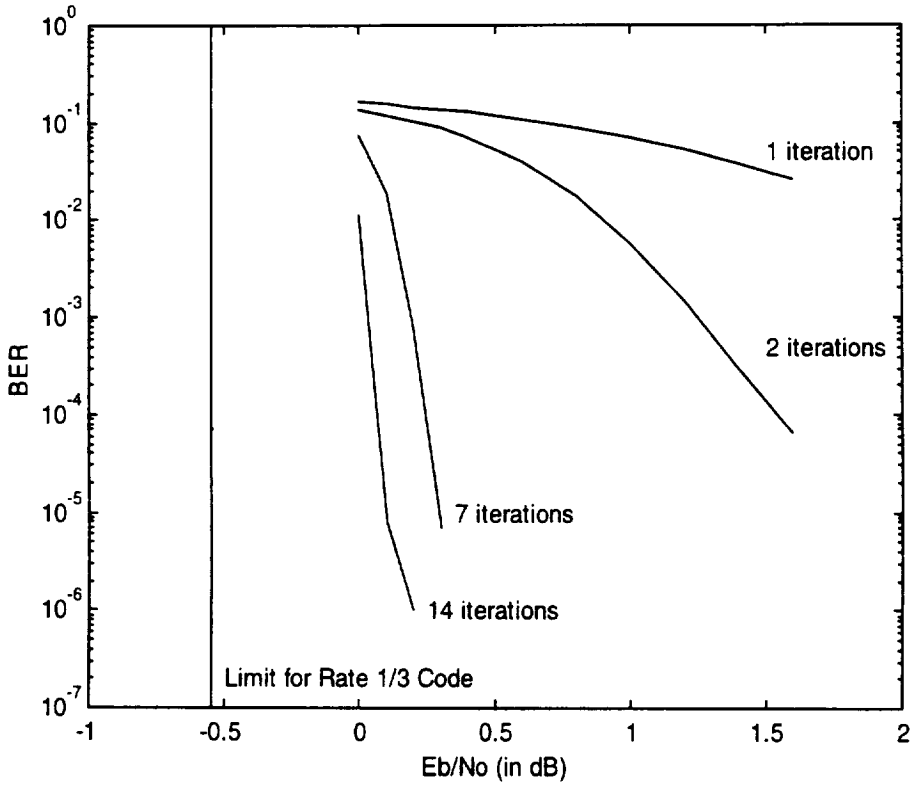


Figure 4.4 Performance of 27_31 RSCC's Without Puncturing.

4.5 Inaccurate Noise Variance Measurement

Finally the effect of inaccurate noise variance measurement on the decoder was observed. The effect of underestimating the variance is given in Figure 4.5.1 with the results of an overestimate of the variance given in Figure 4.5.2. From these Figures it can be seen that an error of 20% either way in the estimate of the variance will result in approximately a .1 dB loss. Of course the worse the estimate is, the worse the decoding performance will be. This seems to be a reasonable amount of loss. This shows that the MAP algorithm is not terribly unstable for inaccurate noise variance measurements.

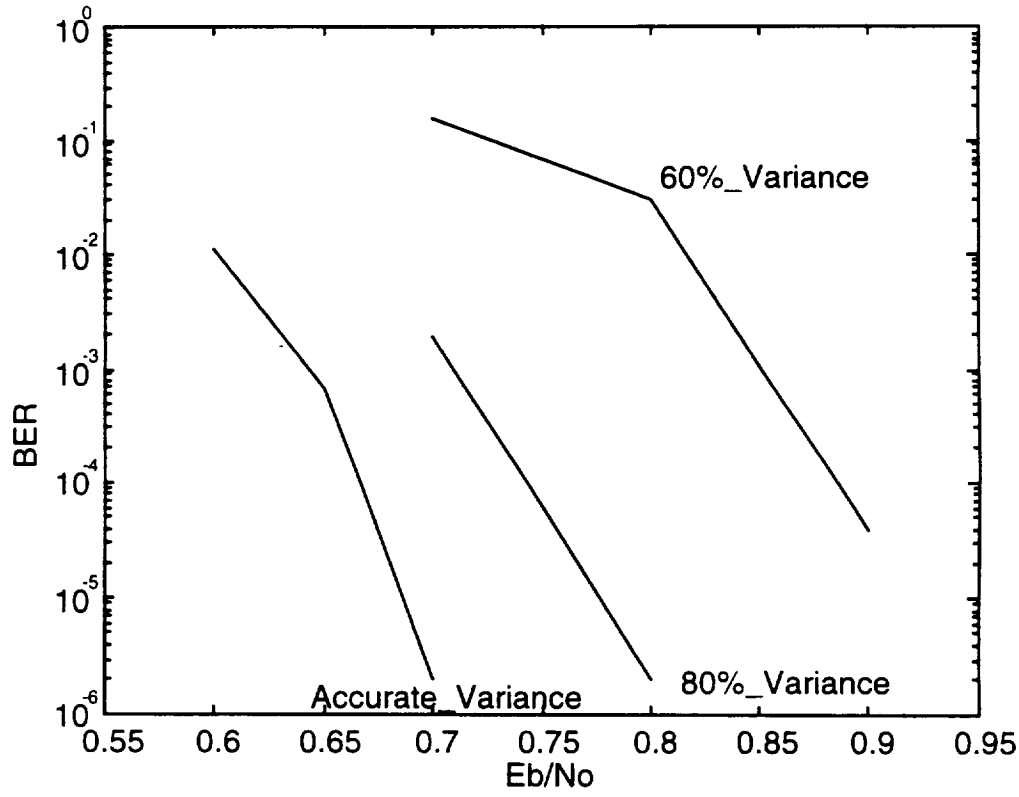


Figure 4.5.1 Underestimating Variance

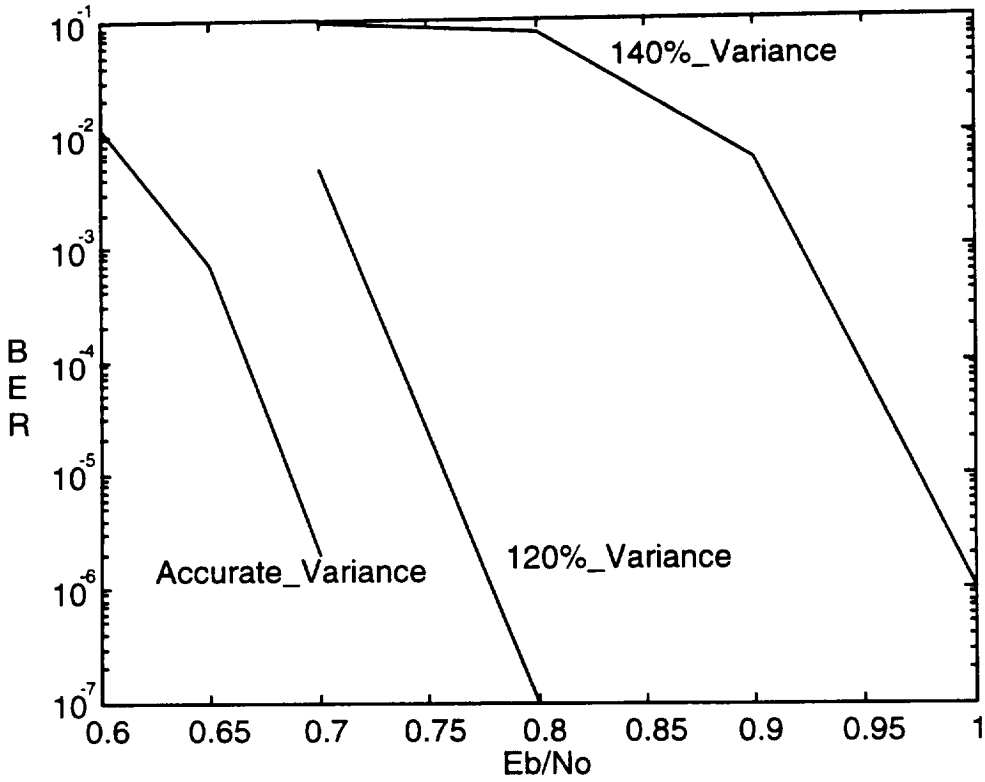


Figure 4.5.2 Overestimating Variance

4.6 Further Research

Some of the questions about Turbo-codes that are still unanswered at this time will now be presented, some of which were posed in [11].

It has been found that the MAP algorithm used with Turbo-codes approaches analytical bounds given in [11] after many iterations. One question is whether suboptimal decoding algorithms, such as the log-MAP algorithm and the Soft Output Viterbi Algorithm (SOVA), will also converge to same levels. Also the complexity of these algorithms versus the optimal MAP algorithm needs to be analysed. Perhaps two of these algorithms could be used for decoding, first

decoder being a less complicated one for the first few iterations and the MAP algorithm as a "clean up" type of decoder that eliminates the residual error.

While it has been shown that it is not hard to obtain a good large size interleaver it remains to be seen whether an analytical device can be found that will give an optimal interleaver for a given interleaver size. Also the analysis of the optimal interleaver for a small interleaver still has not been completely solved.

Multi-dimensional Turbo-codes have also been investigated. Multi-dimensional Turbo-codes are codes that are encoded by sending the systematic information and sending the information through multiple interleavers to be encoded through multiple RSCC's

The combined modulation and coding technique, Trellis Coded Modulation (TCM) provides good coding gain as well as bandwidth efficiency. Combining the ideas of Turbo-codes and TCM was begun in [16].

Reference:

1. Hamming, R , Coding and Information Theory, Prentice Hall,1986

2. D. Divsalar, S. Dolinar, R.J. McEliece, and F. Pollara, "Performance Analysis of Turbo-codes," in Proc., IEEE MILCOM, pp. 91-6, Nov. 1995.

3. Forney, G., Concatenated Codes, M.I.T., Cambridge, MA, 1966.

4. Berrou, C., Glavieux, A. and Thitimajshima, P., i ''Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes," Proc. of ICC '93, pp 1064-1070.

5. Robertson, P., ''Illuminating the Structure of Code and Decoder of Parallel Concatenated Recursive Systematic (Turbo) Codes," Proc. of GLOBECOM '94, San Francisco, CA, USA, Nov. 1994, pp 1298-1303.

6. Battail, G, "A Conceptual Framework for Understanding Turbo Codes", IEEE Journal on Selected Areas in Comm.Vol 16, No 2, Feb.1998

7. Lin,S. Costello,D, Error Control Coding: Fundamentals and Applications, Prentice Hall 1983

8 .Thitimajshima, P., ``Recursive Convolutional Codes and Application to Parallel concatenation," Proc. of GLOBECOM '95, Singapore, Nov. 1995, pp. 2267-2272.

9.Benedetto, S. and Montorsi, G., ``Performance Evaluation of Parallel Concatenated Codes," Proc. of ICC '95, Seattle, WA, USA, June 1995, pp. 663-667.

10 .Berrou, C., et. al., ``Near Optimum Error Correcting Coding and Decoding : Turbo-Codes," IEEE Trans. on Communications, pp. 1261-1271, Oct. 1996.
, pp. 737-740.

11.Benedetto, S. and Montorsi, G., ``Design of Parallel Concatenated Convolutional Codes," IEEE Trans on Info. Theory, Nov. 1995, pp. 2273-2277.

12.Bahl, L, Cocke, J. Jelinek, F, and Raviv, J, ''Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate," IEEE Trans. on Info. Theory, pp. 284-287, Mar. 1974.

13.P. Robertson, E. Villebrun and P. Hoeher, "A Comparison of Optimal and Sub-Optimal MAP Decoding Algorithms Operating in the Log Domain", Int. Conf. on Commun., pp. 1009-13, 1995.

14.Hagenauer, J. and Hoeher, P., ``A Viterbi Algorithm with Soft-Decision Outputs and Its Applications," Proc. of GLOBECOM '89, Dallas, TX, USA, Nov. 1989, pp. 711-717.

15.Viterbi, A, "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes", IEEE Journal on Selected Areas in Comm.Vol 16, No 2, Feb.1998

16 S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Parallel Concatenated Trellis Coded Modulation," in Proc., IEEE Int. Conf. on Commun., (May), pp. 974-8, 1996.

Appendix A
Properties of Nonrandom Block Interleavers

Some analysis of the distance properties of nonrandom block interleaved sequences will now be given. This will show that some low weight input sequences (i.e. input weight 2 or 3) will produce output words that have a high output weight and who's output weight increases for larger interleavers. This is a good result because the goal of encoding Turbo-codes through an interleaver is to boost the output weight for sequences that would produce a low weight codeword through a single RSCC. However the analysis will also show that nonrandom block interleavers produce too many low output weight codewords that are not affected by interleaver size for input weight 4. This will show that nonrandom block interleavers do not adequately "randomize" the output from RSCC2. This analysis will follow Berrou closely [10]

Consider the Turbo-encoder shown in Figure 1.3. To simplify analysis and to give some concrete numbers to observe, the RSCC generator will be a 23_35 (octal) punctured encode which is shown in Figure 4.2.3. Those sequences that produce finite weight outputs of both RSCC's and have a finite weight input sequence are called global finite codewords or FC patterns. Some FC patterns with low output weight will be shown.

Consider a large, M*M nonrandom block interleaving matrix (assuming M is a power of 2). Information bits are read in through the rows and read out

through the columns. By assuming the matrix is filled with only a small number of ones and the rest of it is filled with zeros the analysis is greatly simplified. Because of the recursive nature of the codes at least 2 information bits being one is necessary for a FC to be produced. The RSCC's repeat every $2^m - 1$ bits for an m memory code. With d representing a systematic sequence with weight w and p1 and p2 representing the parity information generated by RSCC1 and RSCC2 respectively, distance from the zero codeword of the FC is given as

$$\text{distance}(w) = w + \text{distance}_{p1}(w) + \text{distance}_{p2}(w) \qquad (A.1)$$

w is the weight given for the systematic portion of the output and $\text{distance}_{p1}(w)$ is the weight given for the punctured output of RSCC1 with the input being d_k. $\text{distance}_{p2}(w)$ is the weight of the punctured output of RSCC2 with the input being the interleaved version of d_k. Puncturing is done by transmitting p_{1k} only at odd times k (k = 1, 3, 5, ...) and p_{2k} at even times k.

For an input weight of 2, distance(2) can be given if $\text{distance}_{p1}(w)$ is assumed to be generated by the minimum distance between bits that will produce a FC (for a memory 4 encoder the distance between 2 one's that will cause a finite output weight is 15 because a RSCC repeats itself after $2^m - 1$ bits meaning that $\text{distance}_{p1}(w)$ is 4). Now

$$\text{distance}(2) = 2 + 4 + \text{INT}((15*M + 1)/4) \qquad (A.2)$$

The final term is generated by assuming that the (15*M +1)/2 symbols output from RSCC2 are 1, half of the time. For the size of interleaver used in the simulation (M = 256), distance(2)would be about 966. This shows that M is the main factor for the output weight for large interleavers with weight 2 input

sequences. In fact it has been shown [11] that increasing the size of the interleaver in a Turbo-code scheme by a factor of N will decrease the BER by a factor of 1/N. This means that if an interleaver size of 100 bits in a Turbo-code scheme generates BER 10^{-3} at a given E_b/N_o then an interleaver size of 1000 should generate BER of 10^{-4}.

For an input d_k with weight 3 some of the patterns that can cause a FC can be seen by tracing the output on the state diagram for three inputs that are one's, but they are not easy to catalogue. It might be assumed once again that the distances are similar to the case of 2 1's because the finite codeword output from RSCC2 will still be several times M long. This means that weight 3 input sequences will produce output weights that will increase with larger interleavers and therefore give better performance.

For higher input weight sequences the analysis comes down to viewing the input as the separate combination of several lower weight codewords. For example an input of weight 4 can be viewed as an input of 2 weight 2 codewords. The minimum output weight for input weight 4 is when global FC is interleaved with the input at the corners of a square with 1's on the corners (Fig. A.1). The minimum output weight for this is given by

$$d(4) = 4 + 2^* \min\{ \text{distance}_{p1}(w)\} + 2^*\min\{ \text{distance}_{p2}(w)\}$$

$$= 4 + 8 + 8 = 20 \qquad \text{(A.3)}$$

Also notice that any rectangular input pattern with weight 4, and with distances between ones that are a multiple of 15 will cause a FC. What this example shows is that with a block interleaver the output weight of both RSCC codes may be

small. The desire is to map most codewords into medium weight codewords. It is hoped that interleavers

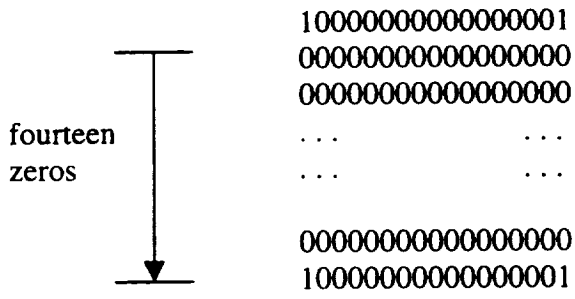


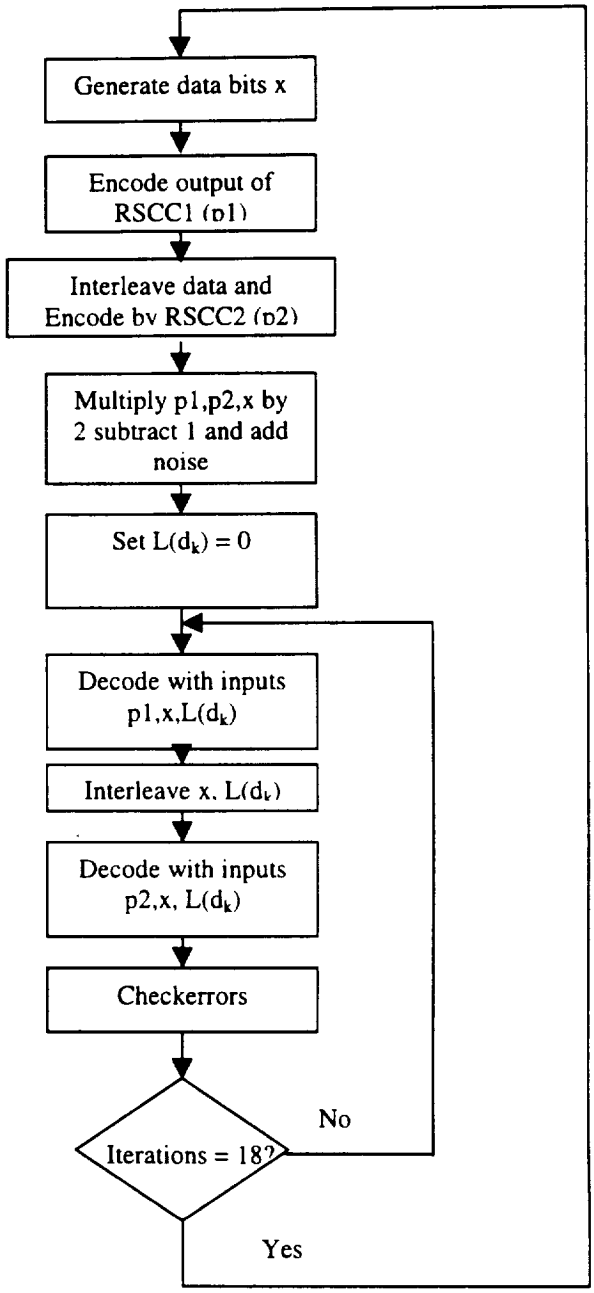
Figure A.1 An Input Pattern That Will Cause a Global FC

that are more random could stand a better chance of mapping those low weight output sequences from RSCC1 into higher weight output sequences of RSCC2. What is desired when data is interleaved is the maximum scattering of data and also the maximum amount of disorder in the interleaved data.

Some of the difficulties in determining good random interleavers are these: How can it be determined that an interleaver that does a good job breaking up, say w = 4 inputs like the one in Figure 2.3.2 will not create more code words with low weights for w = 2? Also the complexity must be limited due to the many times data must be interleaved and deinterleaved in a decoding operation.

For higher weight inputs analysis becomes more difficult due to the fact that the inputs can be viewed as combinations of other patterns of codewords. However it seems that as long as the interleaver used does not have too much structure (i.e. a block interleaver) it should work well enough in a Turbo-code scheme.

Appendix B
Flowcharts For Simulation Program



Appendix C
Program Listing

```
/*                  Simulation Program                */

/* This runs an entire simulation of a turbo coding scheme.  It calls
functionsdecout(out,trans,numstates,sysreal,sysfb,parity, N). N is noise.
numbits is the number of`bits decoded per block,which should be defined in
here.In mmt.h we have all the memory allocation tricks and gasdev() which
is a gaussian random number generator.   The interleavers
which are of the form interfloat( *data, M) where M is the root of the
sizeof the interleaver (square root of numbits)  */
/* To run change filename to dump output to, generator polynomial, EbNo
    memory and number of state  */

#include <stdio.h>
#include <math.h>
#include "mmt.h"
#include "header.h"

void main(void)
{
      FILE *in1;
      int **g1,**g2,i,j,k,prevstate,numbits=16384,numblocks =150;
      int *state,in=0,mem1=4,mem2=4,numstates1=16,numstates2=16;
/*numstates has to be size 2^mem */
      int **out1,**trans1,**out2,**trans2; /* These give output and
transition information about the encoders */
      int *d,numerr=0,stat,M=128,numits=18,file_num_errs[28]={0} /*must be
size numits + 10 */,**intoint,**outofint;
      float N,std,*dcorrupt,*p1,*p2,*intrinsic,EbNo=.8,rate=.5,max =2;
/*rate is .5 because of puncturing.  EbNo is in dB */
      float *sysfb,**alfa1, **beta1;
        float **intofloat,**outoffloat;
        double x;
      long idum[1] = {0};
       /* i,j, ir,jr are indexes that stand for inputs to the interleaving
      matrix and reading from interleaving matrix.   */

      in1 = fopen("3127.txt","a+t"); /* this is the name of the file it
      will be stored in */

      /* g1 and g2 are generator matrices that help create out1,out2,
      trans1,trans2 with prevstate, *state */
      /* mem1 and mem2 are the memory for g1,g2.  numstates1 = 2^mem1
      numstates2 = 2^mem2 */
```

```
        /* d is the information bits which create dcorrupt,p1 (parity bits
        from the first generator), p2 (likewise for the interleaved info) */
        /* EbNo is given for rate 1/2.  numbits is M*M */
        /* The other variables are used in generating the information */
        /* time to allocate memory for **all** the variables. from mmt.h */
        out1 = int_matrix_2d(numstates1,2);
        trans1 = int_matrix_2d(numstates1,2);
        out2 = int_matrix_2d(numstates2,2);
        trans2 = int_matrix_2d(numstates2,2);
        state = (int *) calloc(1,sizeof(int));
        d = (int *) calloc(numbits,sizeof(int));
        dcorrupt = (float *) calloc(numbits,sizeof(float));
        p1 = (float *) calloc(numbits,sizeof(float));
        p2 = (float *) calloc(numbits,sizeof(float));
        intrinsic = (float *) calloc(numbits,sizeof(float));
        sysfb = (float *) calloc(numbits,sizeof(float));
        alfa1 = float_matrix_2d(numstates2,numbits+1);
        beta1 = float_matrix_2d(numstates2,numbits+1);
        intofloat = float_matrix_2d(M,M);
        outoffloat = float_matrix_2d(M,M);
          intoint = int_matrix_2d(M,M);
        outofint = int_matrix_2d(M,M);      /* allocating memory */
        /* converts EbNo to a noise variance */
        N = (2)/((float)((2.0 * rate * (float)(pow(10,EbNo/10)))));
        std = sqrt(N/2);
        /*  printf("EbNo = %f    variance = %f \n",EbNo,N/2); */
        g1=int_matrix_2d(2,mem1+1);          /* allocating mem for g1 */
        g1[0][0]=1; g1[0][1]=1; g1[0][2]=0;  g1[0][3]=0; g1[0][4]=1;
        g1[1][0]=1; g1[1][1]=0; g1[1][2]=1;  g1[1][3]=1; g1[1][4]=1;
        g2=int_matrix_2d(2,mem2+1);          /* allocating mem for g2 */
        g2[0][0]=1; g2[0][1]=1; g2[0][2]=0;  g2[0][3]=0; g2[0][4]=1 ;
        g2[1][0]=1; g2[1][1]=0; g2[1][2]=1;  g2[1][3]=1; g2[1][4]=1 ;
          /* create output and transition matrices  */
        for(in =0;in<=1;in++){
              for(prevstate =0;prevstate<=numstates1-1;prevstate++){
                    state[0] = prevstate;
                    out1[prevstate][in]  = encode(g1,in,state,mem1);
                    trans1[prevstate][in]= state[0];
              }
        }
          for(in =0;in<=1;in++){
              for(prevstate =0;prevstate<numstates2 ;prevstate++){
                    *state = prevstate;
                    out2[prevstate][in]  = encode(g2,in,state,mem2);
                    trans2[prevstate][in]= *state;
              }
        }
/*********************  START SIMULATION  ******************************/
 for(k=0;k<numblocks;k++){

        for(i=0;i<numbits;i++){        /* making info bits */
              d[i] =  (int)(uniform()+.5) ;
              dcorrupt[i] =   2 * ((float)(d[i]))-1 + std*gasdev(idum) ;
        }
        for(i=0;i<numbits;i++){sysfb[i]=0;}
        stat = 0;
        for(i=0;i<numbits;i++){       /* making p1 bits */
```

```
            p1[i] =  ((float)(out1[stat][d[i]]))*2 -1 + std*gasdev(idum) ;
            stat = trans1[stat][d[i]];
      }
      for(i=0;i<=numbits-1;i++) if(i%2 != 0){p1[i] = 0.0;}   /* puncturing
p1 */
      interint(d,M,intoint,outofint);       /* interleave to make p2 bits
*/
stat = 0;
      for(i=0;i<=numbits-1;i++){
            p2[i] =  ((float)(out2[stat][d[i]]))*2-1 + std*gasdev(idum) ;
            stat = trans2[stat][d[i]];
      }
        for(i=0;i<=numbits-1;i++)  if(i%2!=1){  p2[i] = 0.0;}   /*
puncturing p2 */
      deinterint(d,M,intoint,outofint);
      for(i=0;i<=numbits-1;i++){  /* truncate to prevent overflow */
            if(dcorrupt[i]>max){dcorrupt[i]=max;}
            if(dcorrupt[i]<-max){dcorrupt[i]=-max;}
            if(p1[i]>max){p1[i]=max;}
            if(p1[i]<-max){p1[i]=-max;}
            if(p2[i]>max){p2[i]=max;}
            if(p2[i]<-max){p2[i]=-max;}
      }

      numerr= checkerr(d,dcorrupt,numbits); /* see how many errors there
are originally */
      file_num_errs[0] += numerr;
      printf(" number of errors for %d bits after 0 iterations is %d
\n",numbits,numerr);
      printf("   error percentage = %f
\n",((float)numerr)/((float)numbits));

for(i=1;i<=numits;i++){   /* turbo decoding process useing process from
                                          Robertsons paper */
decout1(numbits,out1,trans1,numstates1,dcorrupt,sysfb,p1,N,alfa1,beta1);
/*first decoder   uses  p1 to build info. Output of decoder is in sysfb */
      for(j=0;j<numbits;j++){ intrinsic[j] = sysfb[j];}    /* stores
output of first decoder for errorchecking purposes */
      interfloat(sysfb,M,intofloat,outoffloat); /* interleave inputs to
dec2 */
      interfloat(dcorrupt,M,intofloat,outoffloat);
      decout1(numbits,out2,trans2,numstates2,dcorrupt,sysfb,p2,N,alfa1,bet
a1);
            /* output of dec2 is built by p2. Again output of this decoder
is in sysfb*/
            deinterfloat(sysfb,M,intofloat,outoffloat);
            deinterfloat(dcorrupt,M,intofloat,outoffloat);

            for(j=0;j<numbits;j++){ intrinsic[j] = intrinsic[j] + sysfb[j]
+ (2/(N))*dcorrupt[j];}
            numerr=checkerr(d,intrinsic,numbits);/* checking number of
errors */
            if(numerr == 0){i=numits+1;}

file_num_errs[i] += numerr;
 printf("number of errors for %d bits after %d iterations is %d
\n",numbits,i,numerr);
```

```
            printf("   error percentage = %f
\n",((float)numerr)/((float)numbits));

      }
}
/* print results to a file */
fprintf(in1,"g1 is ");
fprintf(in1,"\n");
fprintf(in1,"%d %d %d %d %d \n%d %d %d %d
%d",g1[0][0],g1[0][1],g1[0][2],g1[0][3],g1[0][4],g1[1][0],g1[1][1],g1[1][2
],g1[1][3],g1[1][4]);
fprintf(in1,"\n\n");
fprintf(in1,"g2 is \n");
fprintf(in1,"%d %d %d %d %d \n%d %d %d %d
%d",g2[0][0],g2[0][1],g2[0][2],g2[0][3],g2[0][4],g2[1][0],g2[1][1],g2[1][2
],g2[1][3],g2[1][4]);
fprintf(in1,"\n \n ");
fprintf(in1,"Eb/No is %f \n",EbNo);
fprintf(in1,"number of blocks is %d  \n",numblocks);
fprintf(in1,"number of bits/block is %d \n",numbits);
for(i=0;i<=numits;i++){
      fprintf(in1,"number of errors for %d iterations is %d BER = %f
\n",i,file_num_errs[i],
((float)(file_num_errs[i]))/((float)(numblocks*numbits)));
      }
      fclose(in1);
}
/* MAP decoding function. */
/* function returns the estimate in sysfb */
void decout1(int numbits,int **out,int **trans,int numstates, float
*sysreal,float *sysfb,float *parity,float N,float **alfa,float **beta){
float bsysreal[2],bpar[2],bfb[2],temp1,temp2,mz=0,probzero,probone;
int i,j,k,l,m;          /* indexs */
/* alfa and beta follow bahl et.al. '73 */
/* bpar, bfb, and bsysreal are components of gamma. it is done this way to
      save processing time */
/* probone and probzero are temporary variables to get log likelihood
value*/
for(i=0;i<numstates;i++){ /* initialize alfa, beta */
      for(j=0;j<numbits;j++){
            alfa[i][j] = 0;
            beta[i][j] = 0;
      }
}
alfa[0][0] = 1.0;                    /* initialising alfa */
/* computes all alfa's */
for(i=0;i<numbits;i++){
      bsysreal[0] = exp(-((sysreal[i] + 1)*(sysreal[i] + 1))/N);     /*
components for gamma */
      bsysreal[1] = exp(-((sysreal[i] - 1)*(sysreal[i] - 1))/N);
      bpar[0] = exp(-((parity[i] + 1)*(parity[i] + 1))/N);
      bpar[1] = exp(-((parity[i] - 1)*(parity[i] - 1))/N);
      bfb[1] = (exp(sysfb[i]))/(1+exp(sysfb[i]));
        bfb[0] = 1-bfb[1];
        temp1 = 0;
        temp2 = 0;
      for(m=0;m<numstates;m++){
```

```
            for(l=0;l<=1;l++){
                  temp1=alfa[m][i]*bsysreal[l]*bfb[l]*bpar[out[m][l]];
                  alfa[trans[m][l]][i+1] += temp1;
                  temp2 += temp1;
                  }
            }           /* calculates alfa for the next i */

            for(m=0;m<numstates;m++){
                  alfa[m][i+1] = alfa[m][i+1]/temp2; /* normalize */
                  }
      }                 /* alfa is done */

/* initialize beta at the last time */
for(i=0;i<numstates;i++){
      beta[i][numbits] = 1.0/((float)(numstates))  ;
      }
for(i=numbits;i>0;i--){   /* recursively calculate beta */
      bsysreal[0] = exp(-((sysreal[i-1] + 1)*(sysreal[i-1] + 1))/N); /*
components for gamma */
      bsysreal[1] = exp(-((sysreal[i-1] - 1)*(sysreal[i-1] - 1))/N);
      bpar[0] = exp(-((parity[i-1] + 1)*(parity[i-1] + 1))/N);
      bpar[1] = exp(-((parity[i-1] - 1)*(parity[i-1] - 1))/N);
      bfb[1] = (exp(sysfb[i-1]))/(1+exp(sysfb[i-1]));
      bfb[0] = 1-bfb[1];
      temp1 = 0;
      temp2 = 0;
        for(m=0;m<numstates;m++){
            for(l=0;l<=1;l++){
            temp1 =
beta[trans[m][l]][i]*bsysreal[l]*bfb[l]*bpar[out[m][l]];
            beta[m][i-1]+=temp1;
            temp2 += temp1;
            }
            }           /* calculates beta for the next i */

            for(m=0;m<numstates;m++){
                  beta[m][i-1] = beta[m][i-1]/temp2; /* normalize */
            }
      }           /* beta is done */
/* now to put it together to get approximation of output */
/* and put it in sysfb                                   */
for(i=0;i< numbits;i++){
      bpar[0] = exp(-((parity[i] + 1)*(parity[i] + 1))/N);        /*
components for gamma */
      bpar[1] = exp(-((parity[i] - 1)*(parity[i] - 1))/N);
      probzero = 0;
        probone = 0;
      for(m=0;m<numstates;m++){           /* go through all the states */
            for(j=0;j<=1;j++){
                  if(j==0){
                  probzero +=
alfa[m][i]*beta[trans[m][0]][i+1]*bpar[out[m][0]];
                  }
                  else{
                  probone  +=
alfa[m][i]*beta[trans[m][1]][i+1]*bpar[out[m][1]];
                  }
```

```
            }
      }
      sysfb[i] = log(probone/probzero);
}
for(i=0;i<=numbits-1;i++){    /* truncate to prevent overflow */
            if(sysfb[i]>17){sysfb[i]=17;}
            if(sysfb[i]<-17){sysfb[i]=-17;}

      }
}
/* program to check errors */
int checkerr(int *d,float *sys,int numbits){
int sum=0,i;
for(i=0;i<numbits;i++){
      if(d[i] == 0){
            if(sys[i]>=0){
                  sum++;
            }
      }
      else{
            if(sys[i]<=0){
                  sum++;
            }
      }
}
return sum;
}
void interfloat(float *data,int M,float **into,float **outof){
int i,j;
int p[8]={17,37,19,29,41,23,13,7},inc,ir,jr,eps;
/* this is from berrou '95   */

inc = 0;                                    /* load into matrix */
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            into[i][j] =data[inc++];
      }
}
for(i=0;i<M;i++){    /* read out of the matrix */
      for(j=0;j<M;j++){
            ir = ((M/2 +1)*(i+j))%M;
            eps = (i+j)%8;
            jr = ((p[eps]*(j+1))-1)%M;
            outof[i][j] = into[ir][jr];
      }
}
inc=0;     /* read it back into the data stream  */
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            data[inc]=outof[i][j];
        inc++;
      }
}
}
void deinterfloat(float *data,int M,float **into,float **outof){
int i,j;
int p[8]={17,37,19,29,41,23,13,7},inc,ir,jr,eps;
```

```
/* this is from berrou '95   */
inc = 0;                                        /* load into matrix */
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            outof[i][j] = data[inc++];
      }
}
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            ir = ((M/2 +1)*(i+j))%M;
            eps = (i+j)%8;
            jr = ((p[eps]*(j+1))-1)%M;
            into[ir][jr] = outof[i][j];
      }
}
inc=0;
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            data[inc]=into[i][j];
        inc++;
      }
}
}
void interint(int *data,int M,int **into,int **outof){
int i,j;
int p[8]={17,37,19,29,41,23,13,7},inc,ir,jr,eps;
/* this is from berrou '95   */
inc = 0;                                        /* load into matrix */
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            into[i][j] =data[inc++];
      }
}
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            ir = ((M/2 +1)*(i+j))%M;
            eps = (i+j)%8;
            jr = ((p[eps]*(j+1))-1)%M;
            outof[i][j] = into[ir][jr];
      }
}
inc=0;
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            data[inc]=outof[i][j];
            inc++;
      }
}
}
void deinterint(int *data,int M,int **into,int **outof){
int i,j;
int p[8]={17,37,19,29,41,23,13,7},inc,ir,jr,eps;
/* this is from berrou '95   */
inc = 0;                                        /* load into matrix */
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            outof[i][j] = data[inc++];
```

```
      }
}
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            ir = ((M/2 +1)*(i+j))%M;
            eps = (i+j)%8;
            jr = ((p[eps]*(j+1))-1)%M;
            into[ir][jr] = outof[i][j];
      }
}
inc=0;
for(i=0;i<M;i++){
      for(j=0;j<M;j++){
            data[inc]=into[i][j];
        inc++;
      }
}
}
int encode(int **g,int in,int *state,int mem)
/* program to help generate output and state transition matrices. it takes
      the generator matrix, the input, and the state (in integer form) and
      returns the output value and the transition state (in *state).
memory
      is size of number of delay units.  To see how this is done look at
      berrou et.al. */
{
      int i, k, a[4]={0},b[4]={0}, c = 0,fb;
      k = state[0];
      binstat( k, mem,a);
        c += in;
      for(i=1;i<=mem;i++){
            c += a[i-1]*  g[0][i];      /* determines feedback bit c */
      }
      fb = c%2;
        c = fb;
      for(i=1;i<=mem;i++){
            c += a[i-1]*g[1][i];      /* c is the outputed bit now. */
      }
      c = c%2;                        /* now to get the next state */
      for(i=0;i<mem-1;i++){                 /* shifting previous state */
            b[i+1]=a[i];
      }
      b[0] = fb;                            /* putting feedback bit into
first space */
      state[0] = intstat(mem,b);
      return c;
}
void binstat(int k,int m, int *mvect)
      {  /* converts k into m bit row vector */
            int i;
            for (i=m;i>0;i--)
                  {
                        mvect[m-i]=(int)(k/((int)pow(2,i-1)));
                        k -= mvect[m-i]*pow(2,i-1);
                  }
      }
int intstat(int m,  int *mvect)
```

```
{ /* converts a m bit row vector, *mvect, into an integer k */
        int i,k=0;
      for(i=0;i<m;i++){
            k += mvect[i]*(int)(pow(2,m-i-1));
      }
      return k;
}
```