Order Number 9130287

Performance of nonbinary Projection Codes

Abdelatif, Nasser N., Ph.D.

City University of New York, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

PERFORMANCE OF NON-BINARY PROJECTION CODES

BY
NASSER N. ABDELATIF

A dissertation submitted to the Graduate faculty in Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy,
The City University of New York
1991

This manuscript has been read and accepted for the Graduate Faculty in Engineering in satsfaction of the dissertation requirement for the degree of Doctor of Philosophy.

February 21, 1991
Date

Chair of Examining Committee

2/21/91
Date

Executive Officer

Prof. T. Saadawi

Prof. D. Vaman

Prof. J. Barba

Prof. D. Manela

Supervisory Committee

The City University of New York

ACKNOWLEDGEMENT

I wish to express my deep gratitude to Professor Donald L. Schilling for his interest, encouragement and valuable guidance during the progress of this research.

I would like to thank Emmanual Kanterakis for his efforts and time in explaining the new decoding algorithm and his valuable advice.

I would like to thank the members of my doctoral committee: Professor T. Sadaawi, Professor D. Vaman, Professor J. Barba, and Professor D. Manela for the time and effort each has taken to read and constructively criticize this dissertation.

Special thanks to my family for their support and encouragement, and for putting up with me through out this research.

Table of Contents

Table of Figures

1 INTRODUCTION

Projection codes were introduced in 1987 in [1] by D. Schilling and D. Manela . Projection codes are a class of codes which can be implemented as FEC or ARQ codes. D. Manela in [4] showed three different classes of Projection Codes , the SM codes, The PSM codes and the TASM codes. The three classes of codes are similar and differed from one another in the degree of protection offered by each to the parity symbols used. It was shown then that the SM is the most basic and the simplest to implement but offered the least amount of protection for the parity symbols and therefore, had a lower coding gain than the other two. Some simulation results for the binary codes were presented.

G. Lomp in [5] presented some theoretical approximations for the codes, particularly for the TASM code. He also showed that the theoretical analysis of the code is very complicated and hence, presented some simulation results for the performance

of the code.

To understand the behavior of these codes more work was needed particularly for the non-binary case. Understanding the difficulty of theoretical analysis of these codes we decided to study the behavior of the codes ,with special emphasis on the non-bianry codes, by using computer simulations. We simulated the encoding and decoding algorithms using Fortran for many codes from each class , the SM, the PSM and the TASM as block codes. Using these simulation results we showed how each code reacts to variations in its parameters. We also showed how the codes compare with each other and showed that the code performs within bounds that were developed by colleague Y. Gang [21] on the performance of the codes.

E. Kanterakis [6] developed a new decoding algorithm for the convolutional TASM binary code. We extended the algorithm for the non-binary case and simulated many TASM codes using this algorithm. Again we showed the effect of variations in all parameters on the performance of the code. We also compared the non-binary TASM block codes and convolutional codes. And

also showed that the code performs within bounds that were developed for the convolutional codes by colleague D. LI [20]. Furthermore, we developed a hardware implementation design for the decoder of the 8-ary TASM rate 1/2 convolutional code with 3- data lines which can easily be altered to get a decoder for a different code.

2 BACKGROUND AND FUNDAMENTALS.

In recent years, there has been an increasing demand for efficient and reliable digital data transmission and storage systems. This demand is caused by the emergence of high-speed, large-scale data networks. A major concern is the control of errors so that reliable reproduction of data can be obtained.

In 1948, Shannon demonstrated that , if the signaling rate of the system is less than the channel capacity, reliable communication can be achieved if one chooses proper encoding and decoding techniques. Since that time a great deal of effort has been spent on the design of good codes and efficient decoding techniques. Work on coding in the 1950s and 1960s was devoted primarily to developing the theory of efficient encoders and decoders. During the 1970s the emphasis in coding research shifted form theory to practical applications. Since then many coding and decoding techniques have been developed. The need for faster and more reliable techniques will increase with the introduction of new communication systems.

The Coding Problem

For codes to be very effective they must be long, so as to average the effect of the noise over a large number of symbols. Such a code may have 10^{100} possible code words and many times this number of possible received words. While encoding and decoding can be conceptually described by a table , it becomes impossible to construct such table, or even to list all of the code words.

There are three main aspects of the coding problem:

(1) To find codes that have the required error correcting ability.

(2) To find a practical method of encoding .

(3) To find a practical method of detection and correction of errors.

The typical attack on the problem has been to find codes that could be proven mathematically to satisfy the required error-correcting condition. This mathematical structure is then exploited to meet the other two requirements, the abilities to encode and decode.

All codes can be classified into one of three types of

codes, Block codes, Convolutional codes and Automatic Repeat Request (ARQ) codes. Some of the most known codes of each type are presented in this chapter.

2.1 BLOCK CODES

A block code consists of a set of fixed-length vectors called code words. The length of a code word is the number of elements in the vector and is denoted by n . The elements of a code word are selected from an alphabet of q elements. Usually q is chosen to be a power of two, i.e, $q=2^b$ where b is an integer, so that each element has an equivalent binary representation consisting of b bits.

There are q^n possible code words in a block code of length n. From these q^n code words we may select $M=q^k$ code words ($k<n$) to form a code. Thus a block of k information symbols is mapped into a code word of length n selected from the set of $m=q^k$ code words. We refer to the resulting block code as an (n,k) code.

2.1.1 MATRIX DESCRIPTION OF LINEAR BLOCK CODES

The set of all n-tuples form a vector space S. If we select a set of $k<n$ linearly independent vectors from S and from these construct the set of all linear combinations

of these vectors, the resulting set forms a subspace of S, say S_C, of dimension k. In other words we choose q^k n-tuples out of q^n n-tuples to work with , which is exactly the way we described linear block codes before. Any set of basis vectors for the subspace S_C can be used as rows to form a k by n matrix G called the generator matrix of the code.

Any one-to-one pairing of k-tuples and codewords can be used as an encoding procedure, but the most natural approach is to use the following :

$$C=iG$$

where **i**, the information word, is a k-tuple of information symbols to be encoded and **C** is the codeword n-tuple.

Because S_C is a subspace ,it has an orthogonal complement $\overline{S_c}$, which is the set of all vectors orthogonal to S_C. The orthogonal complement has dimension n-k and n-k vectors in its basis. Let H be a matrix with these basis vectors as its rows. An n-tuple C is a codeword if and only if it is orthogonal to every row vector of H, that is,

$$CH^T=0$$

this gives a way of testing whether a word is a codeword. The matrix H is called a parity check matrix of the code [13].

Code Rate R :

The code rate is the ratio of the number of information symbols k, to the total number of symbols n, where n is the sum of the number of information symbols k and the number of parity check symbols, r,

$$R = \frac{k}{k+r} = \frac{k}{n} \tag{2.1}$$

2.1.2 THE HAMMING DISTANCE

The Hamming distance determines the random-error-detecting and random-error-correcting capabilities of a

code. Let $\mathbf{v}=(v_0,v_1,\ldots,v_{n-1})$ be an n-tuple. The Hamming weight of **v**, denoted by w(**v**), is defined as the number of nonzero components of **v**. The Hamming distance between any two n-tuples ,say **u** and **v**, is the number of elements in each n-tuple which differ, and is denoted by d(**u**,**v**). In the case of binary codes the Hamming distance between two n-tuples is equal to the weight of the modulo-2 sum of **u** and **v**. That is

$$d(u,v)=w(u+v) \tag{2.2}$$

For a block code C, the minimum Hamming distance is defined by:

$$d_{\min}=\min\{d(u,v):u,v\in C,u\neq v\} \tag{2.3}$$

In a linear code, the sum of every two codewords in C is another codeword in C. Then, because the all-0 n-tuple is a codeword in C [13],

$$\begin{aligned} d_{\min} &= \min\{w(x):x\in C,x\neq 0\} \\ &= w_{\min} \end{aligned} \tag{2.4}$$

The error correcting capabilities of a code is given by,

$$t = \frac{d-1}{2} \qquad (d \quad odd) \qquad\qquad t = \frac{d}{2} - 1 \qquad (d \quad even)$$

2.1.3 SOME SPECIFIC BLOCK CODES.

2.1.3.1 HADAMARAD CODE

The codewords in a Hadamard code are the rows of a Hadamard matrix. The Hadamard matrix is an (n,n) matrix in which $n=2^k$ where, as usual, k is the number of bits in the uncoded word. One codeword consists of all zeros and all other code words have n/2, 0's and n/2, 1's. Further, each code word differs from every other codeword in n/2 places and for this reason the code words are orthogonal to one another. It is well known [14] that as the length, n ,of an orthogonal code increases, the codes performance approaches Shannon's limit.

For n=2 the Hadamard matrix is

$$M_2 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{2.5}$$

Furthermore, from M_n, we can generate the Hadamard matrix M_{2m} according to the relation

$$M_{2n} = \begin{pmatrix} M_n & M_n \\ M_n & \overline{M}_n \end{pmatrix} \tag{2.6}$$

Where $\overline{M}_n$ is the matrix M_n with each element replaced by its complement.

Since by definition each codeword in a Hadamard code differs from every other codeword in n/2 places the minimum Hamming distance is given in [14] by:

$$d_{\min} = \frac{n}{2} = \frac{2^k}{2} = 2^{k-1} \tag{2.7}$$

Note that the distance d approaches infinity exponentially. Hence the number of errors in a codeword increases exponentially. Also $\frac{t}{n} = \frac{1}{4}$ for binary codes; therefore it cannot correct more than n/4 errors. Whereas the Reed-Solomon code corrects

$t = \frac{r}{2}$ therefore

$$\frac{t}{n}=\frac{r}{2n}=\frac{n-k}{2n}$$

$$\frac{1}{2}-\frac{k}{2n}$$

Hence $\frac{t}{n}<\frac{1}{2}$ for non-binary codes. While for the Golay code $\frac{t}{n}=\frac{1}{8}$ where t is the number of errors that can be corrected in a block.

Since $n=2^k$ the rate of the code is

$$R_c=\frac{k}{n}=\frac{k}{2^k}=k2^{-k} \qquad (2.8)$$

2.1.3.2 HAMMING CODES

There are both binary and nonbinary Hamming codes. This is a class of codes with the property that

$$n=q^r-1$$

$$k=q^r-1-r$$

Where r is any positive integer. Concentrating on the binary case, for example, when r=3, $n=2^3-1=7$ and k=7-3=4, giving the (7,4) Hamming code. The parity check matrix H of a Hamming code has a special property that allows us to describe the code rather easily. For the binary (n,k) Hamming code, the $n=2^r-1$ columns consist of all possible binary vectors with n-k=r elements, except the all-zero vector. For example, the (7,4) hamming code will have the following H matrix [14]

$$H=\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \qquad (2.9)$$

We make the observation that no two columns of H are linearly dependent for otherwise the two columns will be identical. However, for r>1 , it is possible to find 3 columns of H which add to zero. Consequently $d_{min}=3$ for an (n,k) Hamming code [14].

Extended Codes:

All codes can be extended ,that is, starting with a

parity check matrix H, a new, extended, matrix H_e can be formed as follows [14]

$$H_e = \begin{pmatrix} & & & H & & & \cdot\ 0 \\ & & & & & & \cdot\ 0 \\ & & & & & & \cdot\ 0 \\ & & & & & & \cdot\ 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\ 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{2.10}$$

the extended matrix H_e consists of the H matrix with an added row consisting of all 1's and an added right-hand column consisting of all 0's except For the bottom most element which remains a 1.

The matrix H_e defines a (n+1,k) code. This extension of a code increases the minimum distance by 1, so that

$$d_{e,min} = d_{min} + 1$$

Using an extended code increases error detection and therefore reduces the bit error rate. For example, a code having a d=9 can correct 4 errors and the probability of error, P_b is

$$P_b \sim \frac{1}{2} p^5$$

since, if the block is in error, 50% of the bits will be in error. However , using an extended code,

$$P_b \sim \frac{t+1}{n}\binom{n}{t+1}P^{t+1}$$

where $\frac{t+1}{n} < \frac{1}{2}$.

2.1.3.3 CYCLIC CODES.

Cyclic codes are a subset of the class of linear codes which satisfy the following cyclic shift property: If $C=[C_{n-1},C_{n-2},\ldots,C_1,C_0]$ is a code word of a cyclic code, then $[C_{n-2},C_{n-3},\ldots,C_1,C_0,C_{n-1}]$, obtained by a cyclic shift of the elements of C, is also a code word. As a consequence of the cyclic property, the code possess a considerable amount of structure which can be exploited in the encoding and decoding operations.

In dealing with cyclic codes it is convenient to associate with a code word $C=[C_{n-1},C_{n-2},\ldots,C_1,C_0]$ a polynomial C(p) of degree $\leq n-1$, defined as

$$C(p) = C_{n-1}P^{n-1} + C_{n-2}P^{n-2} + \ldots + C_1P + C_0 \qquad (2.11)$$

Suppose we form the following polynomial

$$PC(P)=C_{n-1}P^{n}+C_{n-2}P^{n-1}+\ldots+C_{1}P^{2}+C_{0}P \qquad (2.12)$$

this polynomial cannot represent a code word, since its degree may be equal to n. However, if we divide PC(p) by $P^{n}+1$, we obtain

$$\frac{PC(P)}{P^{n}+1}=C_{n-1}+\frac{C_{1}(P)}{P^{n}+1} \qquad (2.13)$$

Where $C_{1}(p)=C_{n-2}P^{n-1}+C_{n-3}P^{n-2}+\ldots+C_{0}P+C_{n-1}$ which is a code word since C is equivalent to shifting C(P) by one position. Since C_1(P) is the remainder obtained by dividing PC(P) by $P^{n}+1$, we say that

$$C_{1}(P)=PC(P)\bmod(P^{n}+1) \qquad (2.14)$$

or $$PC(P)=Q(P)(P^{n}+1)+C_{1}(P) \qquad (2.15)$$

where Q(P) is the quotient. In a similar manner, if C(P) is a code word in a cyclic code, then $P^{i}C(p)\bmod(P^{n}+1)$ is also a code word of a cyclic code.

Let us now consider a method of generating a cyclic code. Suppose g(P) is a polynomial of degree n-k which

is a divisor of P^n+1. Furthermore, we define a polynomial X(P) of degree $\leq k-1$ as follows:

$$X(P) = X_{k-1}P^{k-1} + X_{k-2}P^{k-2} + \ldots + X_1 P + X_0 \qquad (2.16)$$

where $[X_{k-1}, X_{k-2}, \ldots, X_1, X_0]$ represents the k information bits. Clearly, the product X(P)g(P) is a polynomial of degree $\leq n-1$ which represents a code word [12]. That is

$$C_m(P) = X_m(P)g(P) \qquad m = 1,2,3,\ldots,2^k \qquad (2.17)$$

2.1.3.4 THE GOLAY CODE.

Another example of a cyclic code is the Golay code. The Golay code is a (23,12) cyclic code whose generating function is

$$g(x) = x^{11} + x^9 + x^7 + x^6 + x^5 + x + 1 \qquad (2.18)$$

The minimum distance for the Golay code $d_{min}=7$. The extended form of the code is a (24,12) code whose $d_{min}=8$.

The distinctive feature of this code is that it is the only known code of code word length of 23 which is able to correct 3 errors [14].

2.1.3.5 BOSE-CHAUDHURI-HOCUEENGHEM (BCH) CODES.

The BCH codes are a large class of cyclic codes [14] The BCH code employes k information bits, r parity check bits and therefore the number of bits in a codeword is $n=2^m+1$. Furthermore , the number of errors t which can be corrected in an n-bit codeword is

$$t = r/m$$

where m is an integer related to the number of bits n in the codeword by the formula

$$n = 2^m - 1$$

The minimum distance between BCH codes is related to t by the inequality

$$2t + 1 \leq d_{\min}$$

The generator polynomials for these codes can be constructed from factors of $2^{2m+1}+1$ and have been tabulated.

2.1.3.6 REED-SOLOMON (RS) CODE

The Reed-Solomon block code is organized on the basis of symbols. Let each symbol consist of m-bits. Since we deal only with symbols we must consider that if an error occurs in even a single bit in of symbol, the entire symbol is in error. The RS code contains k information symbols , r parity symbols and a total codeword length of n(=k+r) symbols. It has the further characteristic that the number of symbols in the codeword is arranged to be [14]:

$$n=2^m-1$$

The RS code is able to correct errors in t symbols where t=r/2, therefore it can correct b=mt bit errors if the errors occur in a burst.

The RS code is used mainly for burst error correcting.

2.1.4 ERROR-DETECTING AND ERROR-CORRECTING CAPABILITIES OF A BLOCK CODE [12]

When a code vector **v** is transmitted over a noisy channel, an error pattern of l errors will result in a received vector **r** which differs from the transmitted vector **v** in l places. If the minimum distance of a block code is d_{min}, no error patterns of $d_{min}-1$ or fewer errors can change one code vector into another. Therefore, any error pattern of $d_{min}-1$ or fewer errors will result in a received vector **r** that is not a code word in C. When a receiver detects that the received vector is not a code word of C, we say that errors are detected. Hence, a block code with minimum distance d_{min} is capable of detecting all the error patterns of $d_{min}-1$ or fewer errors. However, it cannot detect an error pattern of d_{min} errors because there exists at least one pair of code vectors that differ in d_{min} places and there is an error pattern of d_{min} errors

that will carry one code vector into the other. Error patterns of more than d_{min} errors will cause the received vector **r** to be closer to a code vector **w** than to the transmitted code vector **v**. Also it should be noted that in some cases an error pattern of more than d_{min} errors will carry one code vector into the other and therefore, it cannot be guaranteed that every error pattern of more than d_{min} errors will be detected. Error patterns of d_{min} errors that are detected will be approximated by the "wrong" code vector if error-correcting is attempted. For this reason, we say that the random-error-detecting capability of a block code with minimum distance d_{min} is $d_{min}-1$.

Even though a block code with minimum distance d_{min} guarantees detecting all the error patterns of $d_{min}-1$ or fewer errors, it is also capable of detecting a large fraction of error patterns with d_{min} or more errors. In fact, an (n,k) linear code is capable of detecting 2^n-2^k error patterns of length n. This can be shown as follows: Among the 2^n-1 possible nonzero error patterns there are 2^k-1 error patterns that are identical to the 2^k-1 code words. If any of these 2^k-1 error patterns occurs, it

alters the transmitted word **v** into another codeword **w**. Thus, **w** will be considered as a correct word and results in an incorrect decoding. Therefore, there are 2^k-1 undetectable error patterns. Hence, there are exactly 2^n-2^k error patterns that are not identical to the code words of an (n,k) linear code. These 2^n-2^k error patterns are detectable. For orthogonal codes $d_{\min}=\frac{n}{2}$ and $2^n-2^k \gg d_{\min}=\frac{n}{2}$.

Let C be an (n,k) linear code. Let A_i be the number of code vectors of weight i in C. The numbers $A_0, A_1, \ldots, A_n$ are called the weight distribution of C. The probability of undetected error can be computed from the weight distribution of C. Let $P_u(E)$ denote the probability of an undetected error. Since an undetected error occurs only when the pattern is identical to a nonzero vector of C [12],

$$P_u(E)=\sum_{i=1}^{n} A_i P^i (1-P)^{n-i} \tag{2.19}$$

where P is the transition probability. If the minimum distance of C is d_{min} then A_1 to A_{dmin-1} are zeros. Hence

$$P_u(E) = \sum_{i=d_{min}}^{n} A_i P^i (1-P)^{n-i}$$

If a block code C with minimum distance d_{min} is used for random-error-correcting, it is important to determine how many errors the code is able to correct. The minimum distance d_{min} is either odd or even. Let t be a positive integer such that

$$t = \frac{d-1}{2} \quad (d \ \ odd) \quad or \quad t = \frac{d}{2} - 1 \quad (d \ \ even) \qquad (2.20)$$

We now show that the code C is capable of correcting all the error patterns of t or fewer errors. Let **v** and **r** be the transmitted code vector and the received code vector respectively. Let **w** be any other code vector in v. The Hamming distances among **v** ,**r**, and **w** satisfy the triangle inequality:

$$d(v,r) + d(w,r) \geq d(v,w) \qquad (2.21)$$

Suppose that an error pattern of t' errors occurs during transmission of **v**. Then the received vector **r** differs form **v** in t' places and therefore d(**v**,**r**)=t'. Since **v** and **w** are code vectors in C, we have

$$d(v,w) \geq d_{\min} \geq 2t+1 \tag{2.22}$$

and therefore, $d(w,r) \geq 2t+1-t'$

If $t' \leq t$ then $d(w,r) > t$

This inequality says that if an error pattern of t or fewer errors occur, the received vector **r** is closer to the transmitted vector **v** than any other code vector **w** in C. Based on the maximum likelihood decoding scheme ,**r is decoded into v**, which is the actual transmitted code vector, and thus the error is corrected.

On the other hand, the code is not capable of correcting all the error patterns of L errors with L>t, for there is at least one case where an error pattern of L errors results in a received vector which is closer to an incorrect code vector than to the actual transmitted vector. Let **v** and **w** be code vectors in C such that

$$d(v,w) = d_{\min} = 2t+1 \tag{2.23}$$

then

$$d(w,r) \geq 2t+1-L \tag{2.24}$$

but L>t

$$d(w,r) \leq t \tag{2.25}$$

and **r** will be decoded into **w** which is an incorrect decoding.

A block code with random-error-correcting capability t is usually capable of correcting many error patterns of t+1 or more errors. If a t-error-correcting block code is used strictly for error correcting the probability that the decoder commits an erroneous decoding is upper bounded by [12]

$$P(E) \leq \sum_{i=t+1}^{n} \binom{n}{i} P^i (1-P)^{n-i} \tag{2.26}$$

Here P(E) is the probability of a codeword being in error.

2.2 CONVOLUTIONAL CODES

Convolutional codes differ from block codes in that the encoder contains memory and the encoder output at any given time unit depend not only on the k-inputs at that time unit but also on m previous input blocks. An (n,k,m) convolutional can be implemented with a k-input, n-output, linear sequential circuit with input memory m. Typically, n and k are small integers with k<n, but the memory order m must be large to achieve low error probabilities. In the important special case when k=1, the information sequence is not divided into blocks and can be processed continuously.

A convolutional code is generated by passing the information sequence to be transmitted through a linear finite-state shift register. An information sequence is shifted in, beginning at time zero, and continuing until the sequence is completed. The stream of incoming information symbols is broken into segments, each of k symbols, called an information frame. An information frame may be as short

as one symbol, and in practice it often is this short. The encoder can store m frames. During each frame time a new information frame is shifted into the shift register, and the oldest information frame is shifted out and discarded. At the end of any frame time the encoder has stored the most recent m frames. At the beginning of a frame , from the incoming information frame and the m stored frames, the encoder computes a single codeword frame of length n symbols. This codeword frame is shifted out of the encoder as the next information frame is shifted in. Hence the channel must transmit n codeword symbols for each k information symbols.

Code Rate R:

A convolutional encoder generates n symbols for each h information symbols, as described before, and R=k/n is called the code rate. Note, however, that for an information sequence of finite length, k.L, where L is the number of the information frames in the sequence. The corresponding code word has length n(L+m), where the final n.m outputs are generated after the last nonzero information symbol has entered the

encoder. In other words, the information sequence is terminated with an all-zero block to allow the encoder memory to clear. Therefore the code rate for a convolutional code is given in [11] by

$$R = \frac{kl}{n(l+m)} \tag{2.27}$$

when L >> m, $\frac{l}{l+m} \approx 1$, thus $R \approx \frac{k}{n}$, which is the same as the rate for block codes.

Memory order m:

A tree encoder can be implemented by using shift registers. The encoder may contain K shift registers, not all of which must have the same length. If k_i is the length of the i^{Th} shift register, then the encoder memory order m is defined as [11]

$$m = \max_{1 \le i \le k} k_i \tag{2.28}$$

2.2.1 DISTANCE PROPERTIES OF CONVOLUTIONAL CODES

The performance of a convolutional code depend on the decoding algorithm employed and the distance properties of the code. The most important distance measure for convolutional codes is the minimum free distance d_{free} defined as

$$d_{free} = \min\{d(v,v'):u \neq u'\} \tag{2.29}$$

where v and v' are code words corresponding to the information sequence u and u' respectively. Hence d_{free} is the minimum distance between any two words in the code. Since a convolutional code is a linear code

$$\begin{aligned} d_{free} &= \min\{d(v,v'):u \neq u'\} \\ &= \min(w(v):u \neq 0\} \end{aligned} \tag{2.30}$$

Hence d_{free} is the minimum weight of the code words produced by the nonzero information sequence.

2.2.2 DECODING A CONVOLUTIONAL CODE

The Code Tree

With a view toward exploring procedures for decoding a convolutional code, we consider the code tree of figure 2.1. This code tree applies to the convolutional encoder with k=4,v=3, and which is constructed for the case $l=5$ corresponding to a 5-bit message sequence.

The starting point on the code tree is at the left and corresponds to the situation before the occurrence of the first message bit. The first message bit may be either a 0 or a 1. We adopt the convention that, when an input bit is a 0, we shall move upward from a node of the tree, and when the input bit is a 1 we shall move downward. Suppose the that the first bit is 1. Then entering the tree at node A, we move downward to the lower branch and to node B. Now using the following expressions for the output sequence [14]

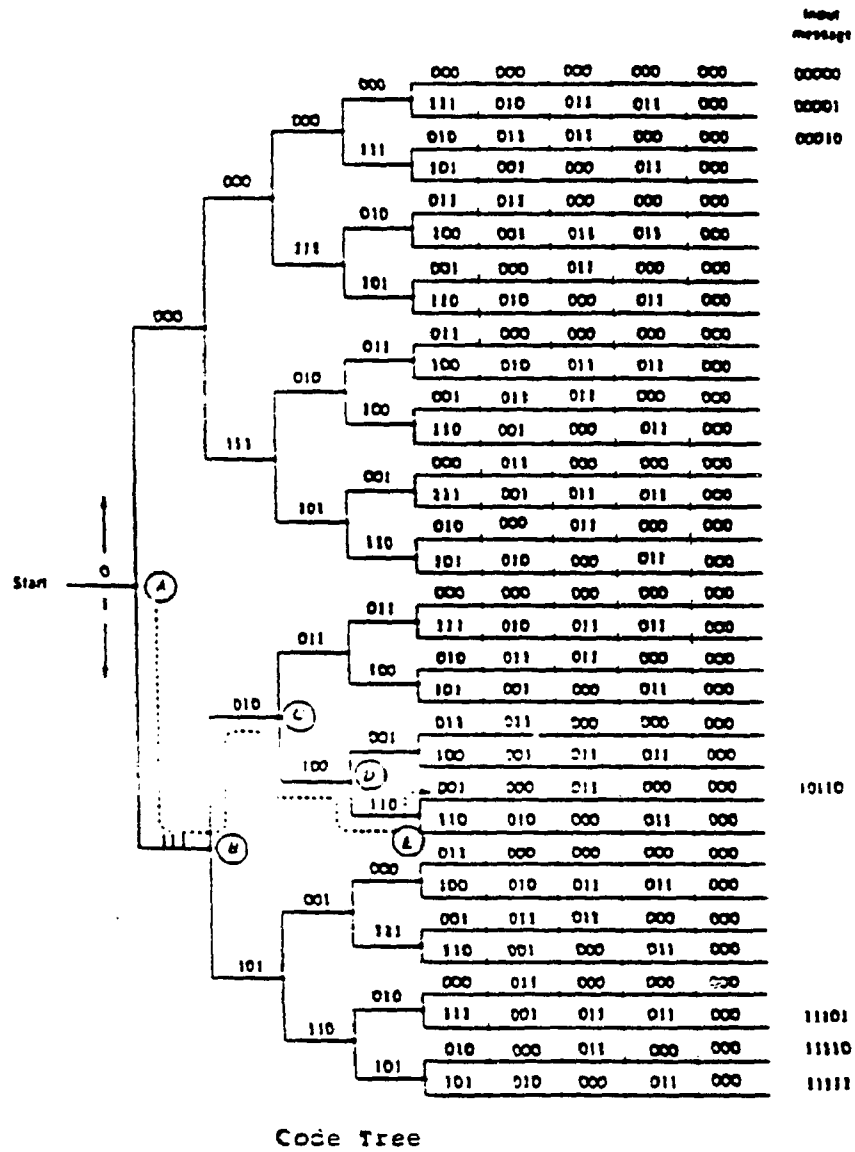
$$v_1 = s_1$$

$$v_2 = s_1 \oplus s_2 \oplus s_3 \oplus s_4$$

$$v_3 = s_1 \oplus s_3 \oplus s_4 \qquad (2.31)$$

we not that when the first input bit is one , the output of the coder is 111. Hence the lower branch associated with node A in the figure has correspondingly been marked 111. Suppose to continue, that the second input is 0. Then we now move upward form node B. We find again, that, when the first two input bits to the encoder are 10, the encoder output during this second input message bit interval is 010. Hence the upper branch moving from node B is correspondingly marked 010. Continuing in this fashion we find that the message m=101100 indicates a downward movement from node C to node D, a downward movement form D to E, an upward movement from node E, and form there out to the end of one of the branches of the tree. The path through the tree is shown by the dashed line in Fig. 2.1.

Figure 2.1 Code Tree [14]



Code Tree

2.2.3 RANDOM ERROR CORRECTING CAPABILITY OF CONVOLUTIONAL CODES

A maximum-likelihood decoder is the optimal decoding technique available for convolutional codes. To be able to evaluate convolutional random error correcting capabilities we will analyze the performance of the Viterbi algorithm for a BSC channel with transition probability P.

Since the code is linear the code performance is not affected by the data sequence transmitted. Hence, for simplicity we assume the all-0 sequence is transmitted. We say that the first event error is made at an arbitrary time unit j if the all-zero path (the correct path in the decoding tree) is eliminated for the first time at time unit j in favor of a competitor path. If the incorrect path has weight d compared to the all-0 word, a first event error is made with probability

[11]

$$P_d = \begin{cases} \sum_{e=(d+1)/2}^{d} \binom{d}{e} P^e (1-P)^{d-e} & d \quad odd \\ \frac{1}{2}\binom{d}{d/2} p^{d/2} + \sum_{e=(d/2)+1}^{d} \binom{d}{e} P^e (1-P)^{d-e} & d \quad even \end{cases} \tag{2.32}$$

Since all incorrect paths of length j or less, can cause an error at time unit j, P(E) can be over bounded, using the union bound, by the sum of error probabilities of each of these paths,

$$P(E) < \sum_{d=d_{free}}^{\infty} A_d P_d \tag{2.33}$$

where P(E) is the event error probability at any time unit, and A_d is the number of code words of weight d. The above bound can be simplified by noting that in equation 1.32 for d odd,

$$P_d = \sum_{e=(d+1)/2}^{d} \binom{d}{e} P^e (1-P)^{d-e}$$

$$< \sum_{e=(d+1)/2}^{d} \binom{d}{e} P^{d/2} (1-P)^{d/2}$$

$$= P^{d/2}(1-P)^{d/2} \sum_{e=\frac{d+1}{2}}^{d} \binom{d}{e}$$

$$< P^{d/2}(1-P)^{d/2} \sum_{e=0}^{d} \binom{d}{e}$$

$$= 2^d P^{d/2}(1-P)^{d/2} \qquad (2.34)$$

For small P the bound is dominated by its first term and the event error probability can be approximated as [11]

$$P(E) \approx A_{d_{free}} (2\sqrt{P(1-P)})^{d_{free}}$$

$$\approx A_{d_{free}} 2^{d_{free}} P^{d_{free}/2} \qquad (2.35)$$

The event error probability bound above can be modified to compute a bound on the bit error probability, $P_b(E)$. Each event error causes a number of information bit errors, equal to the number of nonzero information bits on the incorrect path. Hence, if each event error probability term P_d is weighted by the number of nonzero information bits on the weight d path, or, if there is more than one

weight d path, by the total number of nonzero information bit on all weight d paths, a bound on the expected number of information bit decoding error made at any time unit results. This can then be divided by k, the number of information bits per unit time, to obtain a bound on $P_b(E)$ given by

$$P_b(E) < \frac{1}{k} \sum_{d=d_{free}}^{\infty} B_d P_d \tag{2.36}$$

where B_d is the total number of nonzero information bits on all weight d paths. For small p the term is dominated by its first term, so that

$$\begin{aligned} P_b(E) &\approx \frac{1}{k} B_{d_{free}} (2\sqrt{P(1-P)})^{d_{free}} \\ &\approx \frac{1}{k} B_{d_{free}} 2^{d_{free}} P^{d_{free}/2} \end{aligned} \tag{2.37}$$

2.3 AUTOMATIC-REPEAT-REQUEST

There are two categories for error control in data transmission systems: the forward-error control (FEC) scheme and the automatic-repeat-request (ARQ) scheme. In an FEC system, an error-correcting code is used. When the receiver detects the presence of errors in a received vector, it attempts to determine , the error locations and then corrects the errors. If the exact location of errors are determined, the received vector will be correctly decoded; if the receiver fails to determine the exact locations of errors, the received vector will be decoded incorrectly and erroneous data will be delivered to the user. In an ARQ system, a code with good error detecting capability is used. The receiver attempts to detect the presence of errors if none is detected the receiver notifies the transmitter, via a return channel, that the transmitted code vector has been successfully received. If errors are detected in the received vector. Then the transmitter is instructed, through the return

channel, to retransmit the same code vector. Retransmission continues until the code vector is successfully received. With this system erroneous data are delivered to the user only if the receiver fails to detect the presence of error.

There are three basic ARQ systems. They are: (a) stop-and-wait ARQ, (b) go-back N ARQ, and (c) selective repeat ARQ.

The stop-and wait ARQ system is the simplest to implement and is represented if figure 2.2a. The transmitter sends a codeword to the receiver during the time T_W. The receiver receives and processes the received word and if the receiver detects no errors, it then sends back to the transmitter a positive acknowledgment (ACK) signal. Upon receipt of the ACK signal, the transmitter sends the next word. If the receiver does detect an error, it returns to the transmitter a negative-acknowledgement (NAK) signal. In this case the transmitter transmits the same message and then waits again for an ACK or NAK response before undertaking further transmission. The elapsed time between the end of transmission of one word and the start of transmission of the next word is T_I. Clearly the limitation of such a system is that it must stand by idle without transmission while

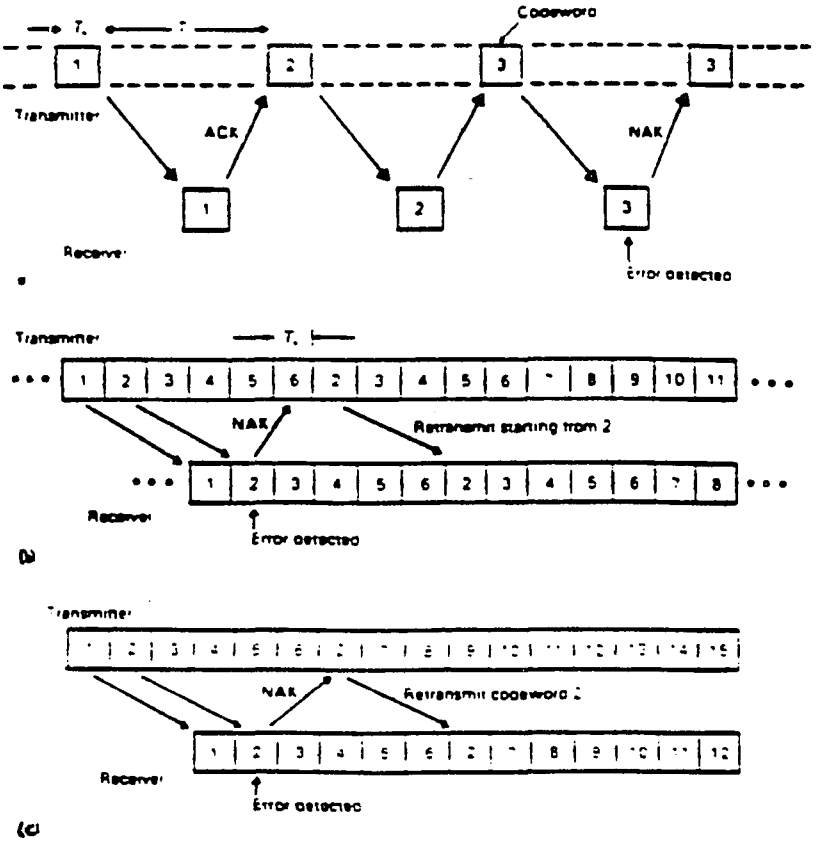
waiting for an ACK or NAK [14].

The go-back N ARQ scheme is represented in figure 2.2b. The transmitter sends messages, one after another, without delay and does not wait for an ACK signal. When, however, the receiver detects an error in a message, say message i, a NAk signal is returned to the transmitter. In response to the NAK the transmitter returns to codeword i and start all over again at that word. In figure 2.2b we have assumed that the propagation delay and the processing at the receiver occupies an interval of such length that when an error is detected in word i the number N=5. The go-back-N system is readily implemented and is an improvement over the stop-and-wait system [14].

The selective repeat ARQ system is represented in figure 2.2c. Here the transmitter sends messages in succession, again without waiting for an ACK after each message. If the receiver detects that there is an error in codeword i, the transmitter is notified. The transmitter retransmits codeword i and thereafter returns immediately to its sequential transmission. The selective ARQ, as might well

be anticipated, has the highest transmission efficiency of the three systems but, on the other hand, it is the most costly to implement [14].

Figure 2.2 ARQ Schemes [14]



(a) Stop-and-wait ARQ (b) Go-back-N ARQ (c) Selective-repeat ARQ

2.3.1 PERFORMANCE OF ARQ SYSTEMS

The performance of ARQ systems measured in two ways, by the probability of error and the transmission efficiency.

Probability of Error

In an ARQ system, block codes are used for error detection. As discussed earlier, whenever an error is detected a NAK is returned to the transmitter and the message is repeated. Thus, the only time that a received message is in error is when a received message has a sufficient number of error and looks like a different codeword. In such a case an ACK is returned and therefore and therefore an error is made.

For an (n,k) block code there are now 2^n possible received words and of these there are 2^k codeword. Thus, if a codeword is transmitted, an error will occur if one

of the 2^k-1 codewords is received. To upper bound p_e we again assume that all 2^n possible received words are equally likely. Then p_e is

$$P_e \leq \frac{2^k - 1}{2^n} \approx 2^{-(n-k)} \tag{2.38}$$

Throughput

The throughput efficiency is defined as the ratio of the average number of information bits accepted at the receiver per unit time to the number of information bits that would be accepted per unit time if ARQ were not used. While all the ARQ systems yield the same error rate, the throughput efficiencies are different.

Throughput of the Stop-and-Wait ARQ [14]

Let P_A be the probability that the receiver accepts the message on any particular transmission. Then the probability that only a single transmission is all that is needed for acceptance is P_A. The probability that two

transmissions will be required is $(1-P_A)P_A$ that is, the product of the probability $(1-P_A)$ that the first transmission was rejected and P_A the probability that it was accepted on the second try. The average number of transmissions required for acceptance of a single word is the sum of the products of the number of transmissions j and the probability of requiring j transmissions, $P_A(1-P_A)^{j-1}$. Thus

$$\begin{aligned}\overline{N}_{sw} &= P_A + 2P_A(1-P_A) + 3P_A(1-P_A)^2 + \ldots \\ &= \frac{1}{P_A}\end{aligned} \tag{2.39}$$

The total time devoted to a single attempt to get the receiver to accept a word is T_W+T_I. Hence, on average, the time required to transmit one word is

$$\overline{T}_{sw} = \frac{T_w + T_I}{P_A} \tag{2.40}$$

If ARQ is not used and no coding bits were added to the k information bits, the time needed to transmit the k bits would be

$$T_k = \frac{k}{n} T_w \tag{2.41}$$

The throughput efficiency of the stop- and- wait ARQ system is

$$\eta_{s\&w} = \frac{T_k}{T_{sw}} = \frac{k}{n} \frac{P_A}{1 + \frac{T_I}{T_w}} \tag{2.42}$$

Throughput of Go-Back-N ARQ

In this system, when the transmitter is informed that an error has been detected in a particular word, retransmission is required of that word and the N-1 words that followed. Hence the retransmission involves N words. Thus, if a word is received in error, N words are retransmitted. Thus, the total number of words transmitted is N+1. Following the analysis used in the stop-and-wait ARQ we find that the average number of word transmissions required for the acceptance of a single word is

$$\overline{N}_{GBN} = P_A + (N+1)P_A(1-P_A) + (2N+1)P_A(1-P_A)^2 + \ldots$$

$$= 1 + \frac{N(1-P_A)}{P_A} \tag{2.43}$$

Correspondingly the average time to transmit one word is [14]

$$\overline{T}_{GBN} = T_w\left(1 + \frac{N(1-P_A)}{P_A}\right) \tag{2.44}$$

and

$$\eta_{GBN} = \frac{T_k}{\overline{T}_{GBN}} = \frac{k}{n}\frac{1}{1 + \frac{N(1-P_A)}{P_A}} \tag{2.45}$$

Selective Repeat ARQ

The mean time for transmission of a word $\overline{T}_{SR}$ in this selective-repeat case is calculated exactly as in stop-and-wait case except that T_I is set to zero. Hence we find

$$\overline{T}_{SR} = T_w / P_A \tag{2.46}$$

$$\eta_{SR} = \frac{T_k}{\overline{T}_{SR}} = \frac{k}{n}P_A \tag{2.47}$$

Comparing ARQ codes to FEC codes.

ARQ codes are simple and provide high system reliability. However, ARQ systems have a severe drawback: Their throughput falls rapidly with increasing the channel error rate. Systems using FEC maintain the same throughput regardless of the channel error rate. FEC systems Have two major drawbacks: First, when a received vector is detected to be in error , it must be decoded delivered to the user regardless of whether it was corrected or not. Since the probability of a decoding error is much larger than the probability of an undetected error, FEC systems can not provide the same reliability offered by ARQ systems. Second, to obtain high reliability, a long powerful code must be used so that a large collection of error patterns can be corrected. This makes decoding hard to implement and expensive.

Projection Codes presented in the next chapter are a group of codes that can be used both as FEC and ARQ codes which is very simple to implement and hence solve some of

the classical problems of FEC and ARQ codes.

3 PROJECTION CODES

Figure 3.1 shows a typical code block. The data bits are arranged in k rows of w bits. The r rows of parity bits are placed below the data block. The code efficiency is k/k+r. There are three schemes for determining the parity bits, each providing greater error correcting capability than the preceding one. For simplicity we will describe the binary case and later will extend it for the nonbinary case.

The first coding scheme which is called the SM code, is as follows:

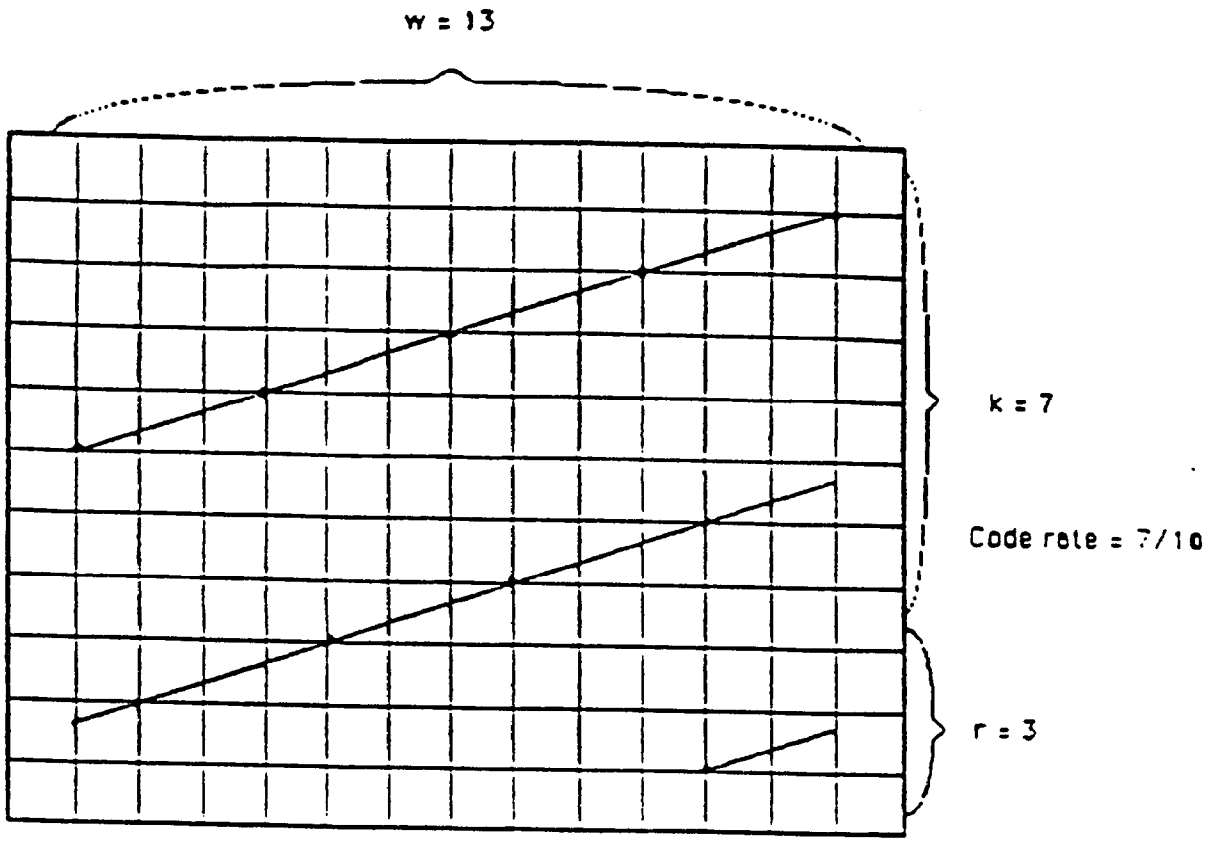
The location of each code bit is identified with a vertex of a uniform lattice in the plane. Slopes of the form $1/m_1$, $1/m_2$, $1/m_3$, ..., $1/m_r$ are chosen. To determine the j^{th} parity row a line having slope $1/m_j$ is drawn through each data bit. The lines wrap cyclically upon reaching the left end of a row as though the block formed a cylinder. These lines are

extended into the parity block and each terminates at a certain parity bit in the j^{th} parity row. This bit is determined so that the sum (mod 2) of the data bits on the line plus the single parity bit is zero. Thus, the j^{th} parity row is not affected by the other parity rows.

It follows from the construction that each data bit has r orthogonal estimators , corresponding to the r lines passing through it. Thus, if r is even, the code can correct [r/2] errors by majority logic, while if r is odd [r-1/2] errors may be corrected [4].

The code described above provides no "protection" to the parity bits. The second scheme, which is called PSM code, partially solves this problem as follows: The first row of parity is determined as above. The second parity row is determined in a similar fashion, but the encoded first row of parity is included in the parity equation. The third parity row checks both the first and second parity rows, already determined. Continuing in this manner, each parity row checks the data rows as well as the parity rows that lie above it.

Figure 3.1 Example of Code Block Showing Construction of a Parity Line [5].



The second scheme also provides r orthogonal estimates for each data bit, and thus has a minimum distance at of least r+1. This can significantly improved. By correctly choosing the slopes $1/m_1$, $1/m_2$, ..., $1/m_r$, the minimum distance of a PSM code may be shown to be 2^r-1. This is accomplished by exhibiting 2^r-2 orthogonal equations on each data bit. The procedure is illustrated in figure 3.2. The r equations are given by considering the original (primary) parity equations. To obtain a new orthogonal equation, a substitution for each estimating bit (such as d* in fig. 3.2) is made in one of the primary equations, using an orthogonal estimator having a different slope from the primary line's slope [4].

This new equation will be orthogonal to the primary equations if the substituted line does not intersect any primary line or any previously determined secondary line. For r=2 this condition is clearly illustrated in Fig. 3.2. For r=3 the construction is more complicated. We impose a constraint on the slopes: no triangle with vertices at the bit positions and sides with slopes equal to $1/m_1$, $1/m_2$, $1/m_3$ should be constructible [5].

The PSM code has greater minimum distance than the SM code, and therefore has greater inherent error correcting capability. The PSM provides r primary orthogonal equations for each data bit and each parity bit in row 1. It provides r-1 primary equations for parity row 2, r-2 for row 3, and only one for row r. The next code provides r primary orthogonal equations for each bit. [4]

The third coding scheme, which is called TASM, requires that the parity bits be determined so that all bits along a line with slope $1/m_j$ have zero sum. This condition cannot be produced directly as in the previous two codes, but the encoding problem may be approached algebraically as follows: Let us associate with each information row a polynomial $d_p(x)$, (p=1 to k), and with each parity row a polynomial $c_q(x)$, (q=1 to k).

It can then be shown [5] that the parity condition stated above leads to the equation:

$$\begin{pmatrix} 1 & x^{m_1} & x^{2m_1} & \cdots & x^{(k-1)m_1} \\ 1 & x^{m_2} & x^{2m_2} & \cdots & x^{(k-1)m_2} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & x^{m_r} & x^{2m_r} & \cdots & x^{(k-1)m_r} \end{pmatrix} \begin{pmatrix} d_1(x) \\ d_2(x) \\ \cdot \\ \cdot \\ \cdot \\ d_k(x) \end{pmatrix} = \begin{pmatrix} x^{-m_1} & x^{-2m_1} & \cdots & x^{-rm_1} \\ x^{-m_2} & x^{-2m_2} & \cdots & x^{-rm_2} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ x^{-m_r} & x^{-2m_r} & \cdots & x^{-rm_r} \end{pmatrix} \begin{pmatrix} c_1(x) \\ c_2(x) \\ \cdot \\ \cdot \\ \cdot \\ c_r(x) \end{pmatrix} \tag{3.1}$$

The above equations are in the ring of polynomials modulo x^W+1. Because this is not a field, matrix inversion is not possible; however, the structure of the matrix does permit a solution, which is expressible as

$$c_j(x) = \sum_{i=1}^{k} d_i(x) q_{ij}(x) \qquad \mathrm{mod}\, x^w + 1 \tag{3.2}$$

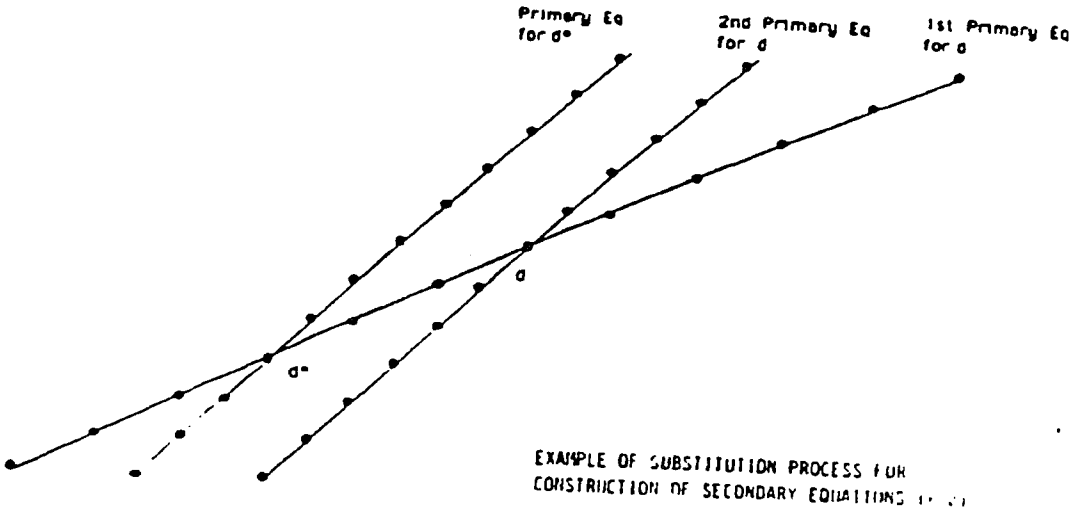
where the $\{ q_{ij}(x) \}$ are polynomial that depend upon the slopes. The above determination of the parity polynomial shows that the j^{th} parity bit vector is the sum (mod 2) of the outputs of k tapped cyclic shift registers Implementation is therefore straightforward [4].

Each of the codes SM, PSM, TASM described above as block codes have a convolutional code equivalent. Since in the case of convolutional codes there is no block length to consider, the

slopes will never reach the end of the data and therefore, wrap-around need not be used. Since, wrapping is not used, some lines near the ends of the data would require data bits that are nonexistent. Whenever such a non-existent data bit is required, it is assumed to be a zero bit. These 0-bits are not transmitted and therefore do not affect the code rate. This augmentation by zeros cause the parity lines to be longer than the data lines ,and therefore, decrease the code rate. But for the cases when the number of columns w is long, so that w>>k and w>>r, where k is the number of data rows and r is the number of parity rows, the decrease in the code rate is negligible and the code rate can be approximated by [4]

$$R = \frac{k}{r + k} \tag{3.3}$$

Figure 3.2 Example of Substitution Process For Construction of Secondary Equations (r=2) [5]



Nonbinary Codes [5]

A non-binary code having symbols of modulo-M can be described in the same manner as the binary case except that the sums are now mod-M rather than mod-2. Figure 3.3 shows a rate R=1/2 non-binary PSM code where M=8. Here r=k=2 and, for simplicity, a convolutional type code is illustrated. Note that symbols on the first parity row check vertical data symbols while symbols on the second row check the symbols on the first parity row and the data symbols along a 45° slope.

Figure 3.3 Non-Binary PSM Encoder [4]

1	2	3	4	5	6	7	8	9	10	11	12	
1	4	5	3	7	5	4	6	1	0	5	3	d_2
2	4	1	7	2	0	0	4	5	7	1	3	d_1
5	0	2	6	7	3	4	6	2	1	2	2	r_1
4	0	3	5	7	7	5	2	3	2	0	2	r_2

3.1 DECODING [5]

The following sub-optimal decoding scheme is capable of correcting many error patterns having weight greater than its minimum distance d_{min}. We describe it for the most powerful code, TASM. Decoding of the SM and PSM codes follow immediately:

Each received bit has r syndrome bits associated with it, corresponding to the r primary orthogonal parity equations. The decoding algorithm begins by determining the number of syndrome bits that are set for each code bit. Thus, an integer in the range [0,r] is associated with each code bit position. The maximum of all such scores is an integer, n. If n>1 all entries with score n are inverted. The process continues until $n \leq 1$. If the process terminates with n=1, a decoding fault is indicated.

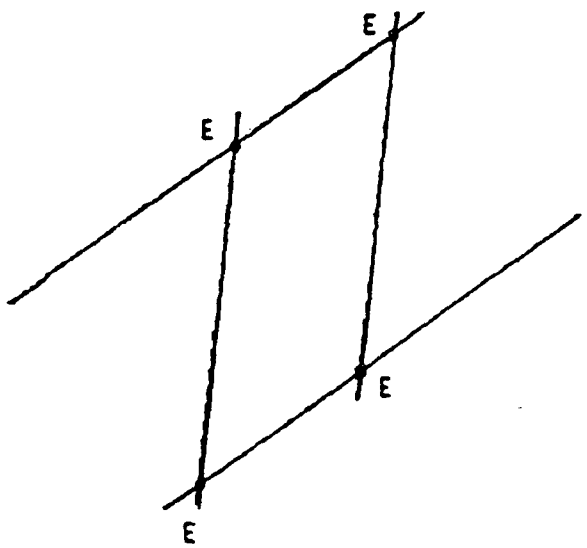
The above decoding algorithm is readily implemented using

shift registers. The received code block is loaded into k+r, w-bit, cyclic shift register (CSR). The r syndrome vectors are computed by tapping these registers. Thus, the syndrome for slope of 1/m is obtained by tapping the j^{th} register at tap number (j-1)m. These outputs are modulo-2 summed to form a syndrome bit. Successive syndrome bits are computed by cyclic shifting the k+r registers w times. Once the syndrome bits are determined and stored (also in a CSR), the error inversion step may be executed using simple logic. This also requires w cyclic shifts. Thus, each iteration requires 2w shift operations. Because of the parallel processing of each row in the code block, large blocks may be processed at high speeds [4].

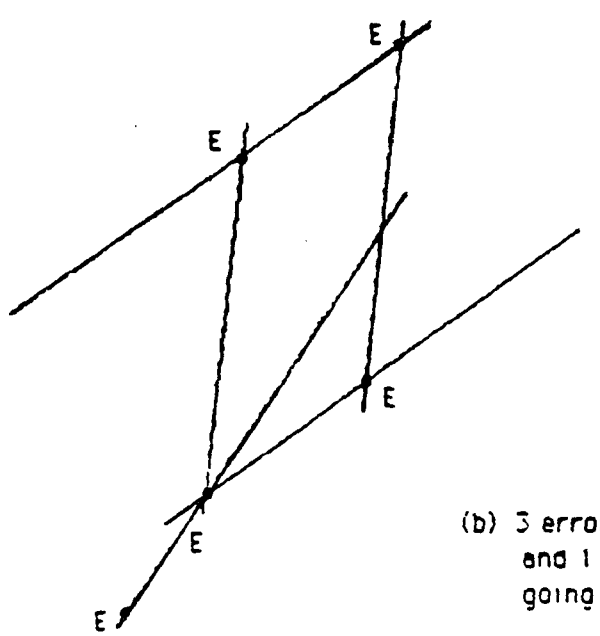
In a binary decoder, two symbols being in error along a parity slope result in a parity slope not being capable of indicating an error. For example, Fig. 3.4a shows that if there are 3 parity rows and 4 errors occur along the parallelogram shown, the errors cannot be corrected. Figure 3.4b shows that all 4 errors need not be on the parallelogram.

Here, 3 errors on the parallelogram; the forth error is along the third slope and will, after the first iteration, yield the error pattern of Fig. 3.4a [5].

Figure 3.4 Showing That 4 Properly Placed Errors cannot Be Corrected When r=3



(a) 4 errors on a parallelogram



(b) 3 errors on a parallelogram and 1 error along third slope going through fourth node

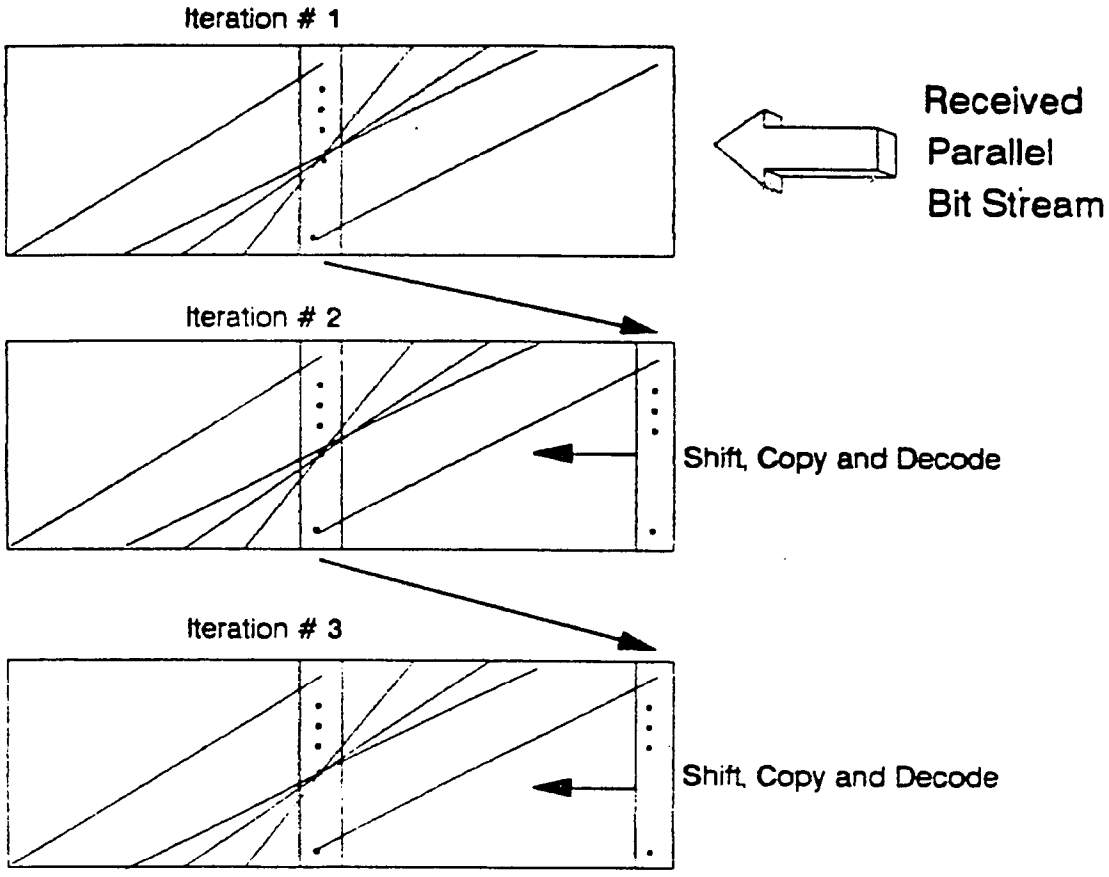
3.1.1 A NEW DECODING ALGORITHM

This decoding algorithm was first developed by Emmanual Kanterakis at SCS Telecomm Inc. for binary Projection Codes. Then it was expanded by me to include non-binary codes. Simulation results for decoding using this algorithm will be shown in Chapter 4.

For convolutional codes a new sub-optimal decoding algorithm was used. This procedure is shown in figure 3.5. In this procedure the syndromes are computed for a single column in a window of the received parallel bit stream. Bits with the highest number of nonzero checks are inverted, the new column formed is inserted in a subsequent window and shifted through. It should be noted that the bits in the first window are not changed. Decoding is again applied to this new stream by inverting bits which have a predetermined number of nonzero checks. The same procedure is repeated for a number of stages depending on the value of r.

We have found that this algorithm out-performed the decoding capabilities of the algorithm described in the previous section. This was because in the first algorithm, the decoding process allowed the more noisy input bits to influence the decoding decisions on bits within the decoding window. However, this was not the case with the new algorithm. During the second iteration, the decoder works on a bit stream which is completely disjoint from the more noisy incoming data bits. The same can be said for any subsequent iterations.

Figure 3.5 Iterative Decoding Procedure [6]



3.1.2 DECODING THE NON-BINARY CODE

Referring to Figure 3.4a it is seen that the errors on the parallelogram cancel since the errors are binary. If the errors are M-ary, the errors will cancel only if the sum, mod-M, is zero. This occurs with lower probability. As seen from Figure 3.4, for the case of 3 parity rows, errors E and E^1 are not correctable if $E+E^1=0$, mod-8. Thus, the probability of error of the nonbinary code is reduced by 1/M from the error rate of the binary code.

3.2 PERFORMANCE [5]

The code performance is given by the probability of block error P_{blk} or by the probability of bit error P_b

$$P_{blk} = \sum_{i=t+1}^{N} n_i p^i (1-p)^{N-i} \qquad (3.4)$$

$$P_b = \sum_{i=t+1}^{N} \frac{in_i}{N} p^i (1-p)^{N-i} \qquad (3.5)$$

where

N = (k+r)w =code block size

t = guaranteed correction capability

p = channel bit error probability

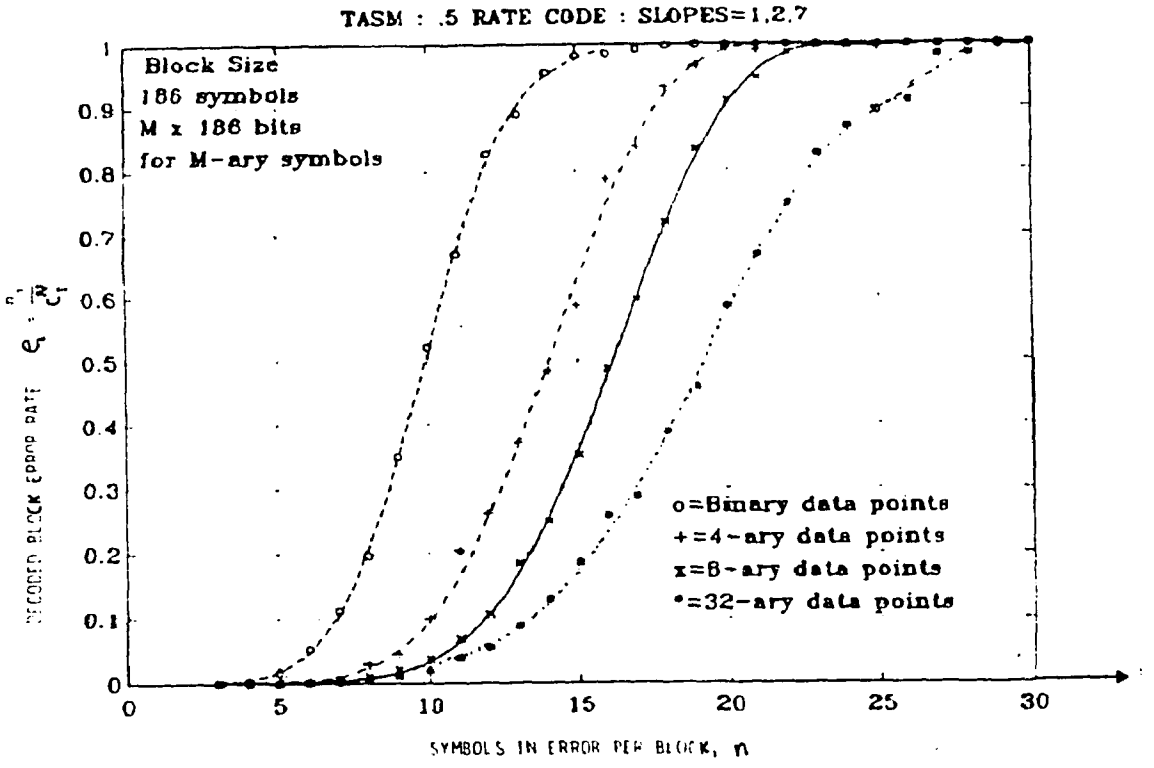
n_i= number of pattern of weight i not correctable.

The above formula includes the numbers n_i which are difficult to evaluate analytically. Simulation studies have been performed to determine performance.

Figure 3.6 typifies the results obtained. Here the TASM projection code was employed. The code rate was R=0.5 and three parity rows were used. The parity slopes employed were 1,2, and 7. Thus ,6 rows consisting of 3 data rows and 3 parity rows, and 31 columns formed the coded block contained 186 symbols. Results were obtained for binary, 4-ary, 8-ary and 32- ary symbols. Note that for the codes shown in the figure, the number of symbol errors that can be corrected is t=3. However, large numbers of errors can be corrected in excess of this number. Note also that, as expected. The

M-ary code results in a value n_i which is approximately 1/M the value of n_i for the binary code for values of i less than approximately 13. Simulation results will be presented in the next chapter.

Figure 3.6 Decoded Block Error Rate Versus Number of Symbols In Error Per Block. [5]



As an illustration, consider a t=3 error correcting code r=3. There exist patterns of weight 4 that are not correctable; in particular, 4 errors located on the vertices of a parallelogram whose sides have slopes $(1/m_1, 1/m_2)$, $(1/m_1, 1/m_3)$, or $(1/m_2, 1/m_3)$ represents an uncorrectable pattern. It was shown in Fig 3.4 that these patterns and certain perturbations of these patterns are the only patterns of weight 4 that are not correctable. The perturbed patterns are those that are transformed into parallelogram patterns by the decoding algorithm. Thus, the number of uncorrectable weight 4 patterns equal the number of possible parallelogram patterns multiplied by the number of such perturbations.

Letting q=(k+r-1), it can be shown that [3] there are

$$T = 1/2(q-1)q(q+1)w \tag{3.6}$$

possible parallelogram patterns of all shapes, sizes, and positions in the code block. There are $2q^2$ perturbations of each parallelogram that are uncorrectable. Thus,

$$n_4 = q^3(q-1)(q+1)w \tag{3.7}$$

or

$$n_4 = (q^4 - q^3)N \tag{3.8}$$

Retaining the first term in the expression for bit error probability in Eq. (3.4) gives

$$p_b = 4(q^4 - q^3)p^4 \tag{3.9}$$

which is **independent of the number of columns.**

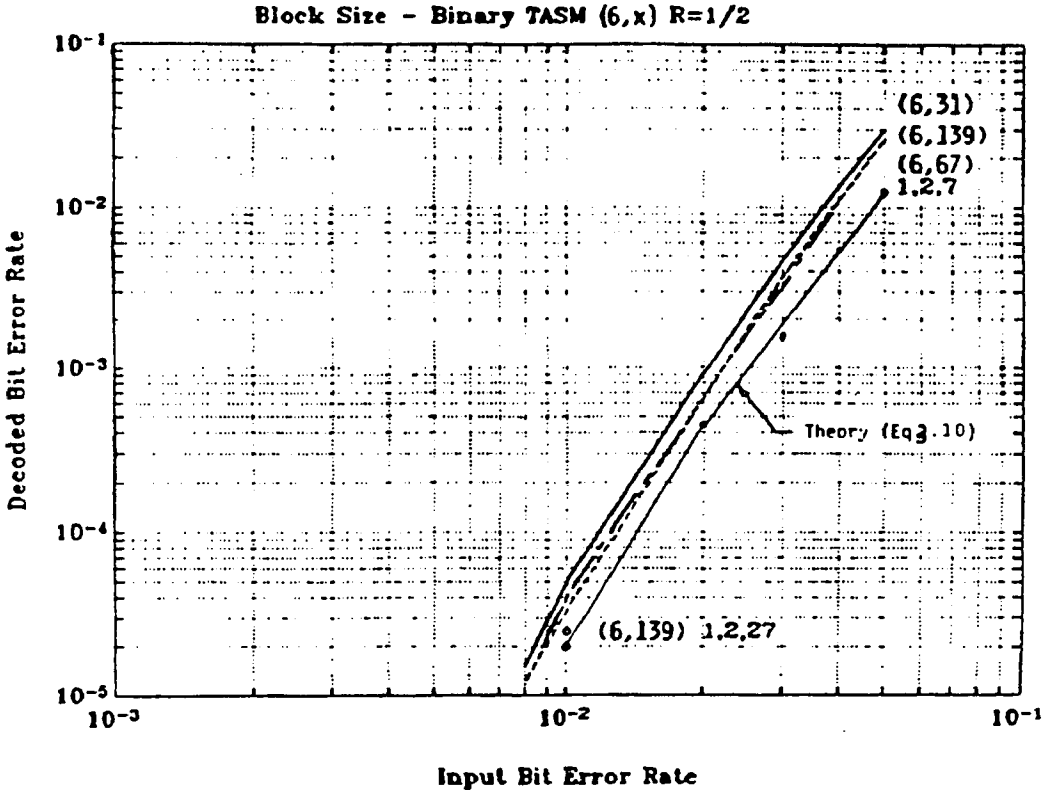
As an example, consider a TASM code with k=3, r=3, w=31 (N=186, rate=1/2)
we have from Eq. (3.9) with small p

$$p_b = (500)(4)p^4 = 2000p^4 \tag{3.10}$$

The TASM Projection code, rate-1/2, was simulated using the three slopes 1, 2, and 7. Three different column sizes, 31, 67 and 139 were used. Figure 3.7 shows the decoded output bit error rate as a function of the input bit error rate.

Conceptually we would expect that using the larger number of columns would serve to reduce the number of parallelograms that are generated as a result of the finite column size of the coded block. Indeed, as seen from the figure some improvement occurs for 67 columns, but at 139 columns there is no additional improvements. Also plotted on figure 3.7 is the asymptotic, theoretical results, given by Eq. (2.10), which is independent of the number of columns, w.[3]

Figure 3.7 Input Bit Error Rate versus Output Bit Error Rate. [5]



Probability of Error for the Non-binary TASM Code

When a nonbinary TASM code is used with m bits/symbol, an error will occur only if the errors cancel at each node of the parallelogram shown in Fig. 3.4. If cancelation does not occur, the error locations would be known and the symbols can be corrected. The probability of error for the nonbinary TASM code is found by modifying Eq. (3.5),

$$P_s = \left[\frac{1}{2^m - 1}\right]^3 4(q^4 - q^3)p^4 \qquad (3.11)$$

Equation (3.11) is valid for small values of the input symbol error rate, p. The term $[\frac{1}{2^m-1}]^3$ arises since if 1 of the 4 errors has a specific symbol error, each of the other remaining 3 error-symbols must have an error which can be only 1 of 2^m-1 possible values for an uncorrectable error pattern to result.

For example, if m=3, k=3 and r=3 [3]

$$P_s \approx 6p^4 \qquad (3.12)$$

Note that as m approaches infinity the probability of error P_S approaches zero if we keep p , the input symbol error rate, fixed. There are two major reasons why we do not use large values of m. Firstly, p, the input error rate increases as m, the number of bits in a symbol, increases; for even if one bit in the symbol is in error the symbol is considered to be in error. However, for burst errors m should be a large as possible Secondly, the complexity of the system increases as N, the number of possible symbols, increases which in turn increases exponentially with m.

Equation 3.11 is an approximation of the output symbol error rate P_S, that is true only if p, the input symbol error rate, is small, so that Eq. 3.5 is dominated by its first term. The true expression for the output error rate is easily obtained by modifying Eq. 3.5 to be,

$$P_s = \sum_{i=t+1}^{N} \frac{in_i}{N} P^i (1-p)^{N-i} \qquad (3.13)$$

where

N= number of symbols in a block.

p= symbol input error rate.

n_i= number of patterns weight i that are not correctable.

The output error rate for the Reed-Solomon code is given in [4] to be,

$$P_s = \sum_{i=t+1}^{N} \frac{i}{N}\binom{N}{i} p^i (1-p)^{N-i} \qquad (3.14)$$

Figures 3.8 and 3.9 [5] show the output error rate as a function of the input error rate. In each case the rate R=1/2, TASM projection code was used with 6 rows and 31 columns. The slopes 1,2, and 7 were employed.

Figure 3.8 shows output block error rate as a function of the input symbol error rate. Note that the improvement between the binary code and the 4-ary code is an error rate reduction of a factor of 10 and the improvement obtained using 8-ary is an additional factor of 4. The output symbol error rate for each of these codes is shown in Fig. 3.9.

Burst Error Correction.

In addition to their major use as a random error correcting codes Projection Codes are capable of correcting burst errors. To explore the burst error correction capabilities of the codes lets consider the following example.
Here, an SM code with 2 data lines and 2 parity lines are used. The coding equations for the code are given as follows:

$$c_1 = d_1 + d_3$$

$$c_2 = d_2 + d_4$$

$$c_3 = d_1 + d_4$$

$$c_4 = d_2 + d_3$$

This code is a one error correcting code and it almost corrects bursts of two errors. Actually there are only Two cases when the code cannot successfully correct bursts of two errors. In those two cases the error in the two symbols must be identical. Hence, the probability of any of those two events occurring is much lower than any other burst of two errors. Note that the probability of the occurrence of any of the two uncorrectable events is greatly reduced by

increasing m, the number of bit per symbol. For large values of m this code performance is very close to that of the (8,4) RS code. which is a two error correcting code.

Figure 3.8 Input Symbol Error Rate Versus Output Block Error Rate [5]

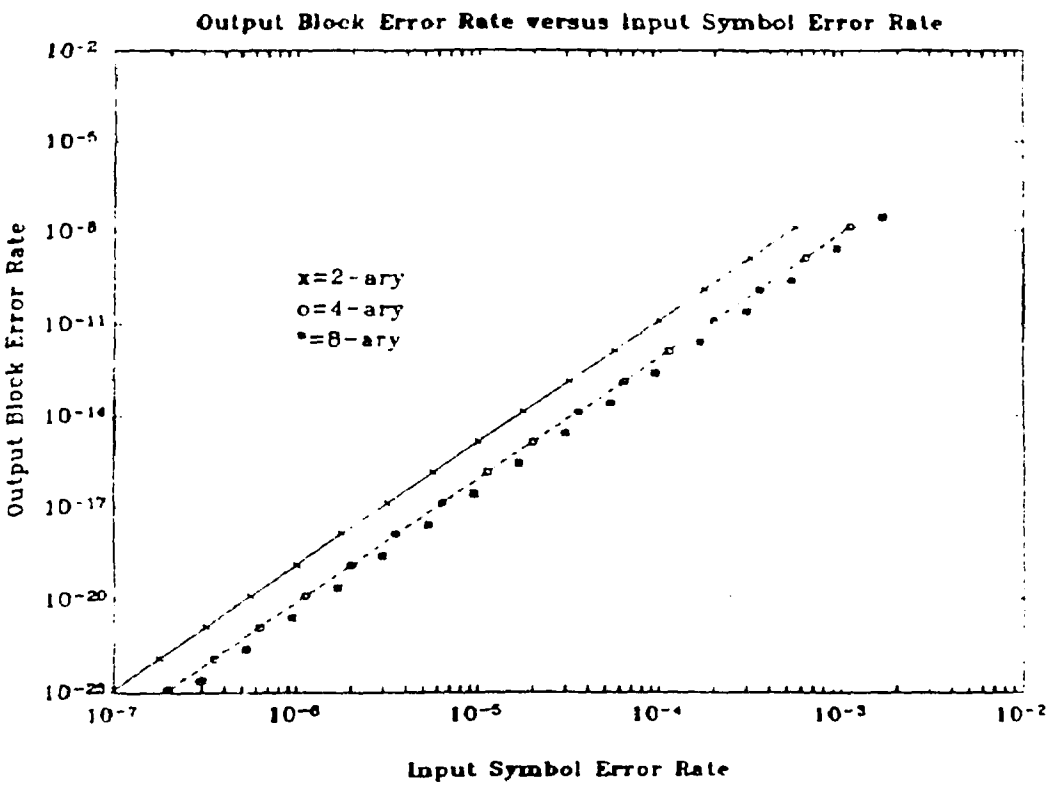
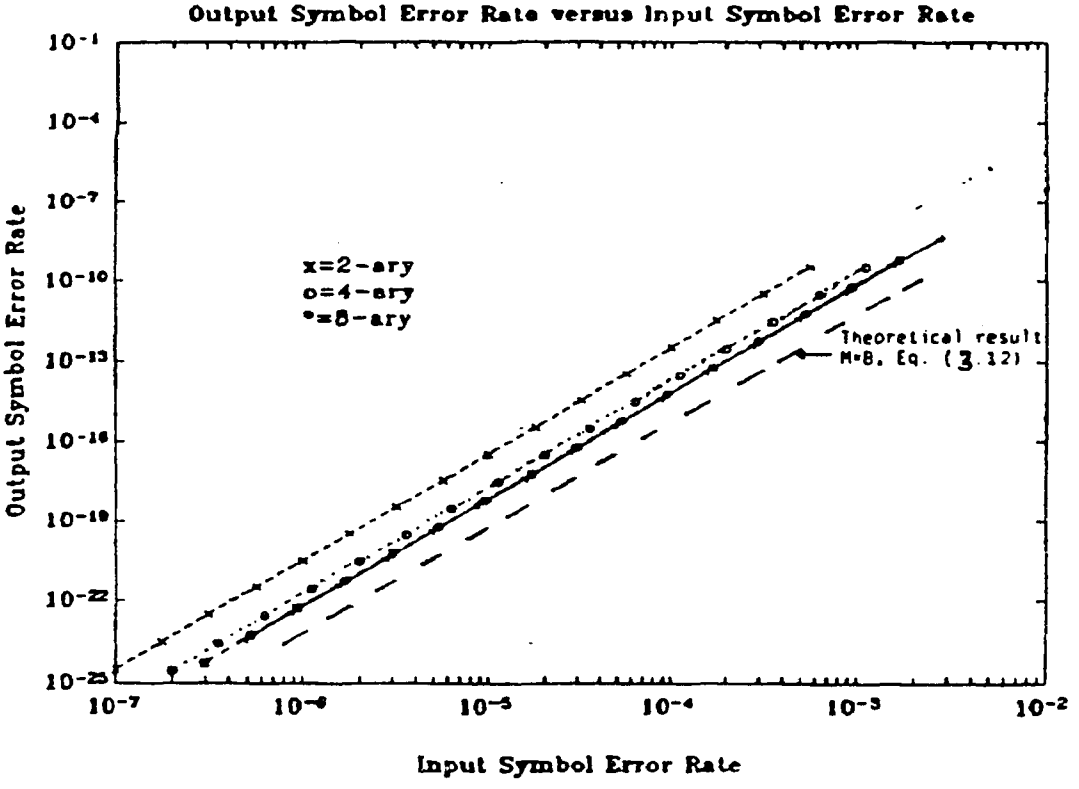


Figure 3.9 Input Symbol Error Rate Versus Output Symbol Error Rate [5]



4 SIMULATIONS.

One of the main contributions made by this author was the detailed simulation of the Projection code. Some simulations were previously presented by D. Manela [4] for the binary codes. Hence we concentrated on simulating non-binary codes. We started by simulating the SM ,PSM, and TASM codes as block codes then we simulated the TASM projection code as a convolution code.

A generic simulation was written for each group of codes as one unit and then obtaining different codes by changing parameters like k, the number of data lines, r, the number of parity lines ,and m, the number of bits per symbol. While attempting to simulate the first group of codes , the SM code, we encountered two major problems: First, the size of the program got too large and we faced memory management problems. In particular, the size of memory needed for compiling such a program became excessive. Second, processing time became too large as well, specially the compiling time. To solve the memory management problem and to reduce the compiling time,

the code was partitioned into smaller groups. In each group k, the number of data lines and r, the number of parity lines were fixed; but m, the number of bits per symbol was allowed to vary. To solve the run time problem r, the number of parity lines was restricted to be no larger than 4. The reason for choosing 4 as our limit is that for values of 5 or more the size of the block used would have to be very large compared to the block sizes of codes with less than 4 parity lines. The size of the block depends on the slopes chosen. The slopes have to be chosen in a certain way so as to avoid the forming of triangles which cannot be detected. The slopes that would avoid the forming of triangle were found to be defined as follows:

$$m_1 = 1$$

$$m_2 = 2$$

$$m_3 = k + r + 1$$

$$m_i = m_{i-1}(k+r-1) - m_{i-2}(k+r-2) + 1 \qquad for \quad i > 3 \qquad (4.1)$$

For example, we will compare the slopes used for rate 1/2 codes with 3,4, and 5 parity lines . In the first case k=3 and r=3, the slopes will be as follows

$m_1 = 1 \quad , \quad m_2 = 2 \quad , \quad m_3 = 7$

whereas for k=4 and r=4 the slopes would be :

$m_1 = 1 \quad , \quad m_2 = 2 \quad , \quad m_3 = 9 \quad , \quad m_4 = 52$

and for k=5 and r=5 the slopes would have to be

$$m_1 = 1 \quad , \quad m_2 = 2 \quad , \quad m_3 = 11 \quad , \quad m_4 = 89 \quad and \quad m_5 = 699 \qquad (4.2)$$

Note that length of the all-zero blocks at the beginning and at the end of the data block, which is given by the following

$$\lambda = m_r(k+r) \qquad (4.3)$$

would vary greatly depending on the value of r. For the above example the lengths of the all-zero blocks for the three codes were chosen to be:

$$\lambda_1 = 7 \times (3+3) = 35$$

$$\lambda_2 = 52 \times (4+4) = 416$$

$$\lambda_3 = 699 \times (5+5) = 6690 \qquad (4.4)$$

where λ_1, λ_2, and λ_3 are the lengths of the all-zero blocks for the first, second and third codes mentioned above respectively.

To keep the code rate at near 1/2, h, the length of the data block would have to be much larger than the respective

value of λ. If we let $h = 20\lambda$, then the lengths of data blocks would be as follows the block length of the code with 3-parity lines would be 700 , where the block length of the 4-parity lines code would be 8320, and the block length of the 5-parity lines would be 139,800. It can easily seen that the block length of the 5-parity lines code is much larger than the block lengths of the 3 and 4-parity lines, and would require processing time beyond the capacity of the computer employed.

For each class of codes, the SM ,PSM and TASM , many codes such as the 3-parity lines rate 1/2 , the 4-parity line rate 1/2, 3-parity line rate 3/4 and the 4-parity lines rate 3/4 were simulated. Each was run as a binary, 8-ary, and 256-ary code.

To study the performance of these codes a communications system, which consisted of three major components, the encoder, the channel, and the decoder was simulated. In the coming sections we will describe the simulation procedures for the three components for the SM, PSM, and TASM codes. For simplicity we will describe the simulations for the 8-ary rate 1/2 code with 3-data lines and 3-parity lines. Throughout this discussion

we will refer to the first, second, and third data lines as the vectors d_1, d_2, and d_3, respectively. We will also refer to the first, second and third parity lines as p_1, p_2, and p_3, respectively. The Error Pattern matrix, which is a matrix of size (k+r,h) used during error detection as described in chapter 3, will be referred to as the matrix EP.

4.1 THE SM CODE

We start with the most basic of the three codes the SM code. Thereafter we discuss the other codes as they relate to the SM code.

The Encoder

In the SM code, parity symbols are functions of data symbols only. That is, the entries of a parity line do not affect the entries of another. Hence, the order of calculating parity symbols is not important. We decided to calculate the three parity lines simultaneously.

To encode a block of data we read the data block from a data file. Then we insert an all-zero block before the data block and another all-zero block after it, as shown in Fig. 4.1. The encoding of the new block is done as shown in Fig. 4.2. Here we start with the first symbol in the first data

line, $d_1(\lambda+1)$, and calculate all parity symbols affecting it. Then we calculate the parity symbols affecting the second symbol in the first parity line, and so on until we encode all the data symbols as shown in Fig. 4.2. For example, for the i^{th} symbol in the first data line the equations to calculate the parity symbols affecting it are;

$$p_1(i-3) = -[d_1(i) + d_2(i-1) + d_3(i-2)]$$

$$p_2(i-8) = -[d_1(i) + d_2(i-2) + d_3(i-4)]$$

$$p_3(i-35) = -[d_1(i) + d_2(i-7) + d_3(i-14)] \qquad (4.5)$$

Note that as shown in figure 4.1 we need $h+\lambda$ lines of each slope. Where h is the number of columns in the data block, and λ is the length of the all-zero block. For the particular code discused in this section $\lambda = 21$. Therefore, the parity lines are longer than the data lines. This difference in length affects the rate of the code, to minimize the this effect on the code rate we choose $h \gg \lambda$.

Figure 4.1 Zero Augmentation of a Data Block

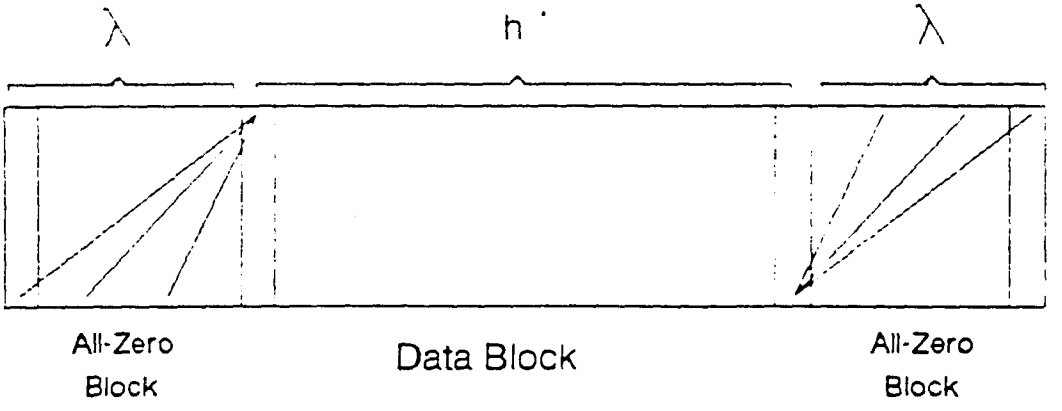
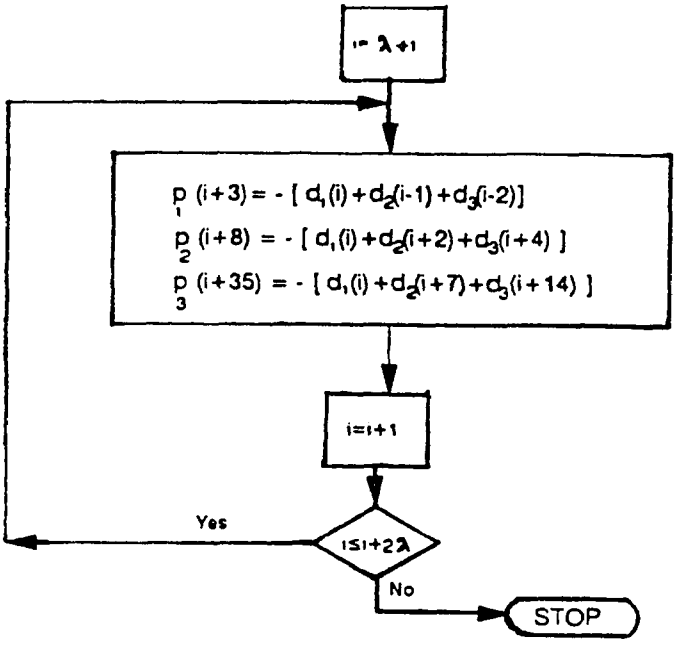


Figure 4.2 Flow Diagram for The Encoder of a Rate 1/2 SM Code with 3-data lines



The Channel

To simulate the channel we simply generated a very long PN sequence. Then to obtain a certain input error rate we altered a particular symbol with some probability. By changing the probability with which we alter the symbol we obtain the different error rates needed.

When altering the value of a symbol, we allow the new value for the symbol to take any possible value in the alphabet with the same probability. In practice the new value for the symbol takes some values with higher probability than the others. For it is more probable that one bit be in error than any other number of bits.

The Decoder

The decoder consists of two major parts, one for error detection and the other for error correction. For error detection we start by initializing the Error Pattern Matrix to all-zeros. The Error Pattern Matrix (EP) is a matrix of size (k+r,h) , whose entries represent the number of times each symbol in the transmitted block is thought to be in error.

As in the encoding process we start with the first symbol of the first data line , $d_1(\lambda+1)$, and check whether errors have occurred on the line of slope 1, starting at $d_1(\lambda+1)$. Then we check if errors have occurred on the line of slope 1 starting at the second symbol in the first data line ,$d_1(\lambda+2)$, and so on. In other words, we detect errors using the first parity line first. To determine whether an error did occur on a line of slope 1 passing through the i^{th} symbol in the first data line , we check if the sum modulo N of the data symbols along that line and the corresponding parity symbol is equal to zero or not. Here N is the size of the alphabet used , N= 2^m,where m is the number of bits per symbol. As shown in Fig. 4.3 if the sum is equal to zero , we assume that no errors have occurred. While if the sum is not zero we assume that at least one error did

occur along the line in question. When errors are detected the appropriate positions in the error pattern matrix are incremented. This procedure was repeated for all other slopes used as shown in Fig. 4.3.

Since every symbol was encoded by r parity lines in this particular example r=3, An error in a particular symbol will be detected by r slopes. Hence, its position in the EP matrix would have a value of r. This will be true only if no other errors that would cancel the effect of this error did occur along any of the lines associated with the symbol in question.

To correct errors we start by inspecting the entries of the EP matrix. When an entry that has a value of r is found, the corresponding data symbol is assumed to be in error. The error is then found by finding a value that will satisfy the most of the decoding equations shown in Fig. 4.3. For example, if the EP entry corresponding to the i^{th} symbol in the first data line has a value of r, then the error value in this example would have to satisfy the following equations:

$$d_1(i)+d_2(i-1)+d_3(i-2)+p_1(i-3)+E=0$$
$$d_1(i)+d_2(i-2)+d_3(i-4)+p_2(i-8)+E=0$$
$$d_1(i)+d_2(i-7)+d_3(i-14)+p_3(i-35)+E=0 \qquad (4.6)$$

Since a solution may not exist, we search for a value of E that would satisfy the most possible number of equations. Therefore, we assumed that every equation might be satisfied by a different value for E, say E_1, E_2, and E_3, which leads to the following equations:

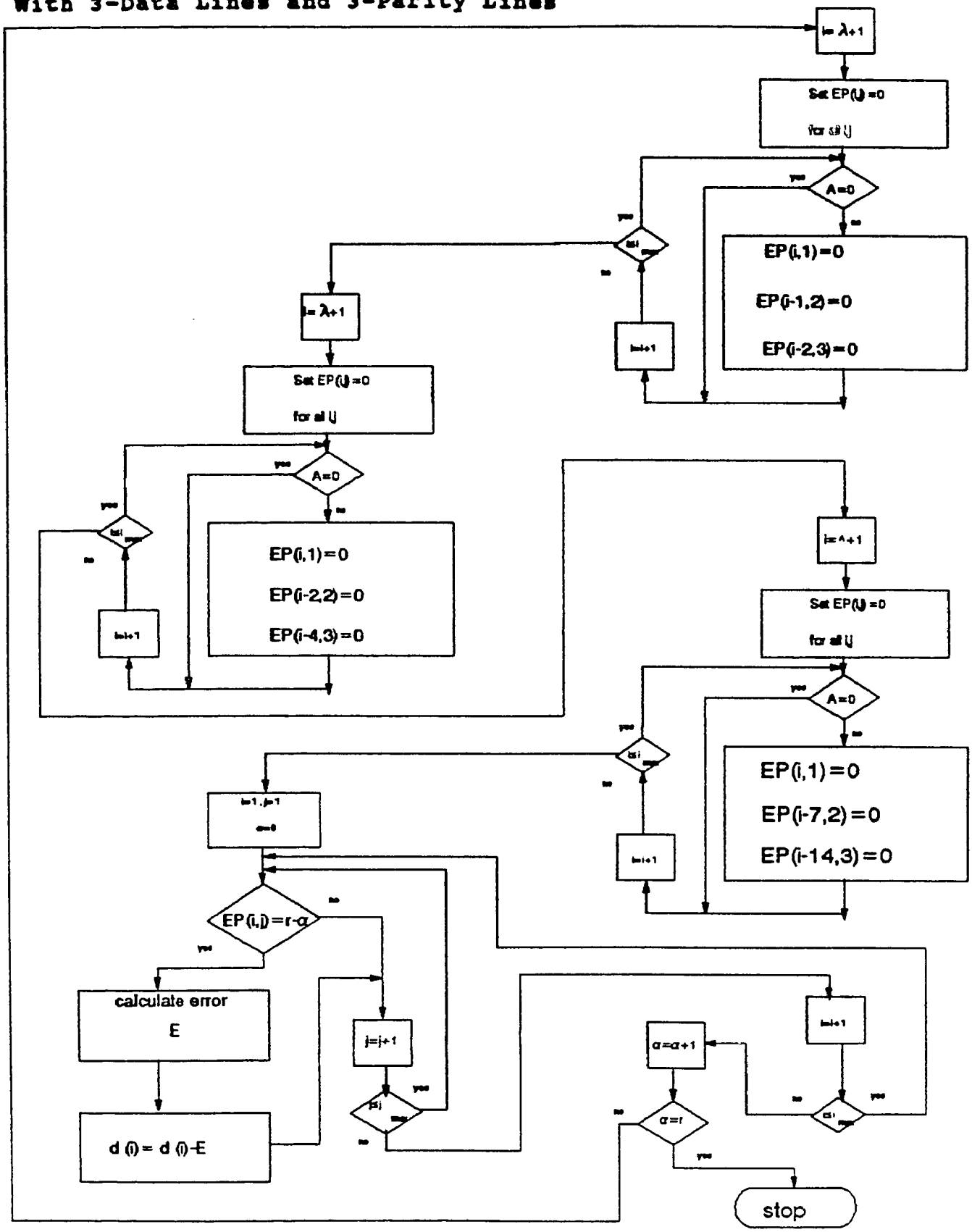
$$E_1=-[d_1(i)+d_2(i-1)+d_3(i-2)+p_1(i-3)]$$
$$E_2=-[d_1(i)+d_2(i-2)+d_3(i-4)+p_2(i-8)]$$
$$E_3=-[d_1(i)+d_2(i-7)+d_3(i-14)+p_3(i-35)] \qquad (4.7)$$

Then we compare E_1 with E_2 and E_3. If they all are equal then the error is assumed to equal to them as well. If they are not all equal then we compare them two at a time. If any two are equal then the error is assumed to equal to them. Note that in the general case we would compare the outcomes of the r equations if they are not all equal. We will first compare each r-1 elements at a time, then r-2 and so on.

After all the entries of the EP matrix are checked, and an attempt is made to correct all symbols of EP value of r, the whole decoding procedure is repeated on the modified data block. That is , we detect errors again by forming a new EP matrix and then we search the EP matrix to locate symbols with errors. However, this time we attempt to correct symbols with an EP value greater than r-1 . This procedure is repeated and every time we decrease the threshold on the value of the EP entries. The process is stopped on one of two conditions, first, if no errors were detected at any iteration, or when a certain preset number of iterations is reached.

When the decoding process is stopped, the output of the decoder is compared to the transmitted data block to determine whether all the errors have been corrected or not. The whole process, encoding ,transmitting and decoding, was then repeated until 10 blocks having errors are found. Then the Block output error rate was then found by dividing the number of blocks with errors, by the total number of blocks transmitted.

Figure 4.3 Flow Diagram for The Decoder of a Rate 1/2 SM Code With 3-Data Lines and 3-Parity Lines



Simulation Results of the SM Code.

We simulated many codes using this procedure. In this section we will present some of the simulation results obtained.

Figure 4.4 shows the results for the binary, 8-ary, and 256-ary SM rate 1/2 with 3-data lines and 3-parity lines. The figure shows that the 8-ary code represents a significant improvement over the binary code. The 256-ary code introduces even more improvement. As discussed in chapter 3 additional improvement would be introduced by increasing the number of bits per symbol. For by increasing the number of bits per symbol the chance of two symbols cancelling will be less. Therefore the probability of symbol error out would approach zero as m, the number of bits per symbols approaches infinity. Large values of m are undesirable as was discussed in chapter 3. We did not simulate any codes of m>8 to limit the run time of the simulation programs.

Figure 4.5 make the same comparison for the rate 3/4 codes. Hence showing that the error rate improves with increasing m, no matter the rate of the code used.

Figure 4.6 compares the results obtained for 8-ary rate 1/2 code, one using 3-parity lines and the other using 4-parity lines. The code using 4-parity lines shows some improvement over the code using 3-parity lines. This improvement is the result of using more estimators per symbol and therefore an error will be detected by more slopes , hence it will be easier to locate. Note that codes with more than 4-parity lines were not simulated for the reasons explained in the previous section.

Figures 4.7 and 4.8 compare the rate 1/2 to the rate 3/4 SM code. Figure 4.7 compares the 8-ary codes while Figure 4.8 compares the 256-ary codes. In both cases the rate 1/2 codes show a considerable "improvement" over the rate 3/4 codes which is expected if we fix r. The number of parity lines, k the number of data lines, will be less for the rate 1/2 code. Therefore, the code will have fewer errors.

Figure 4.4 Output Block Error Rate Versus Input Symbol Error Rate

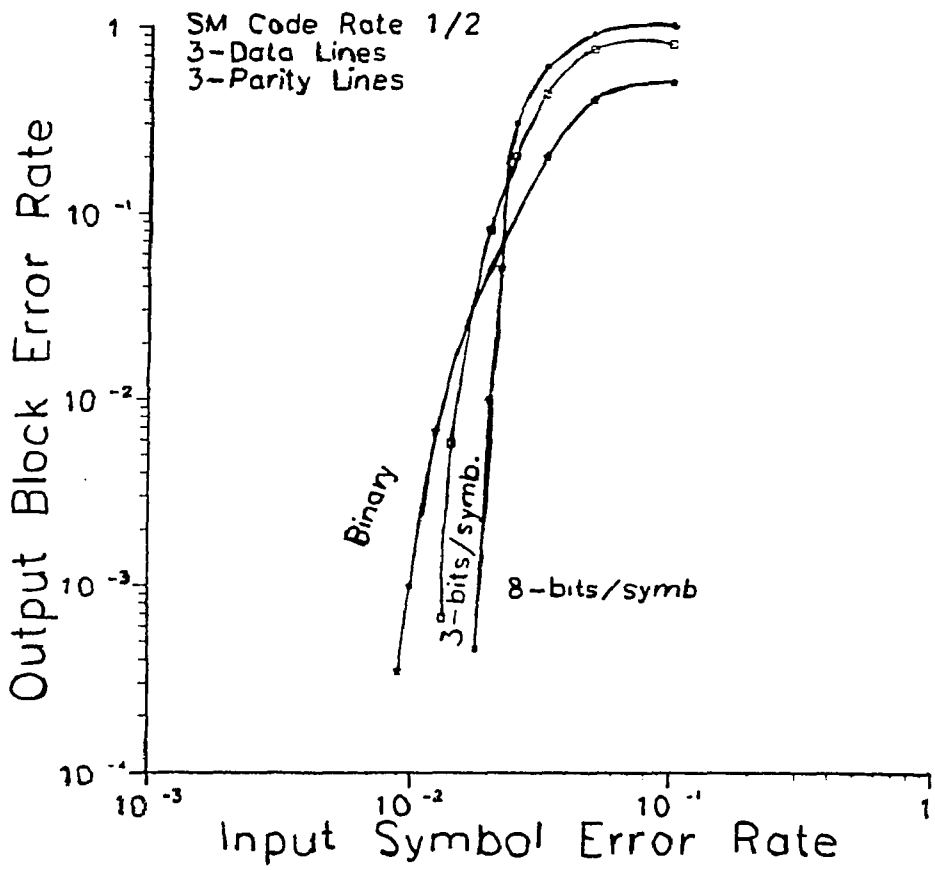


Figure 4.5 Output Block Error Rate Versus Input Symbol Error Rate

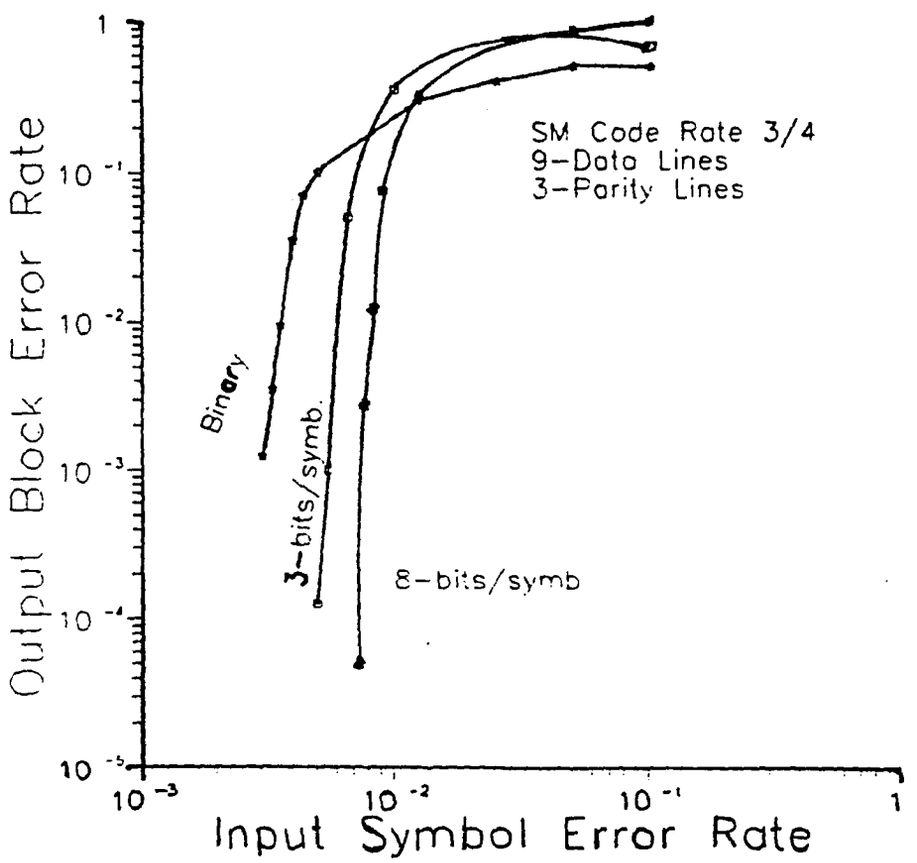


Figure 4.6 Output Block Error Rate Versus Input Symbol Error Rate.

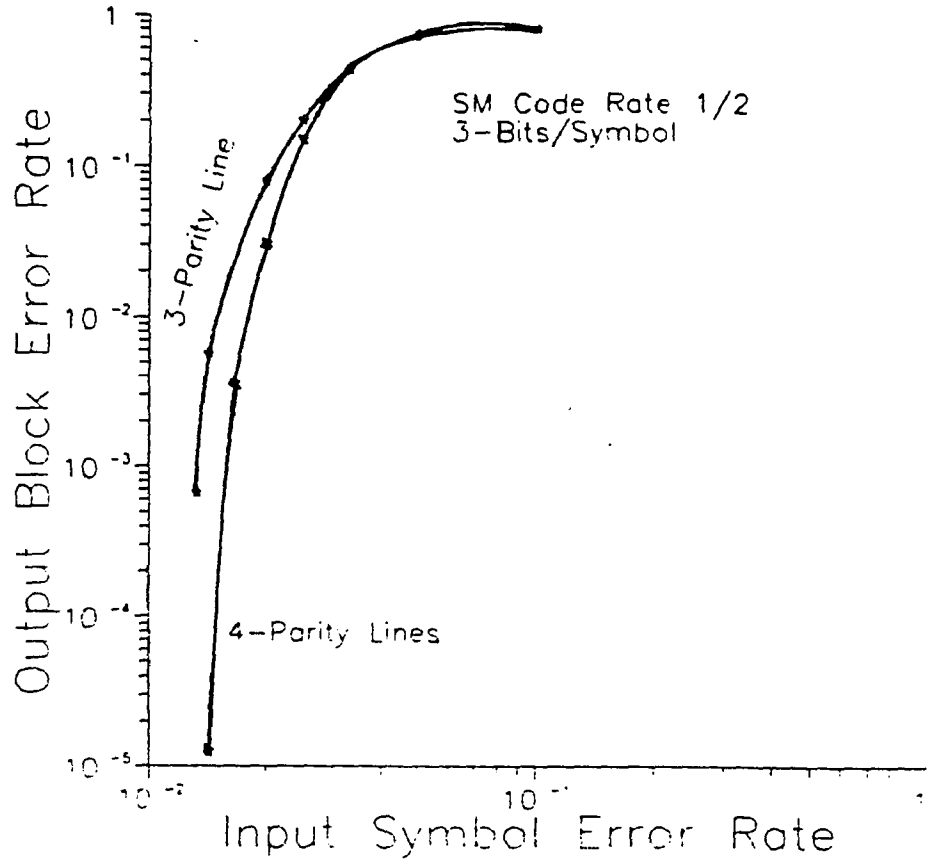


Figure 4.7 Output Block Error Rate Versus Input Symbol Error Rate

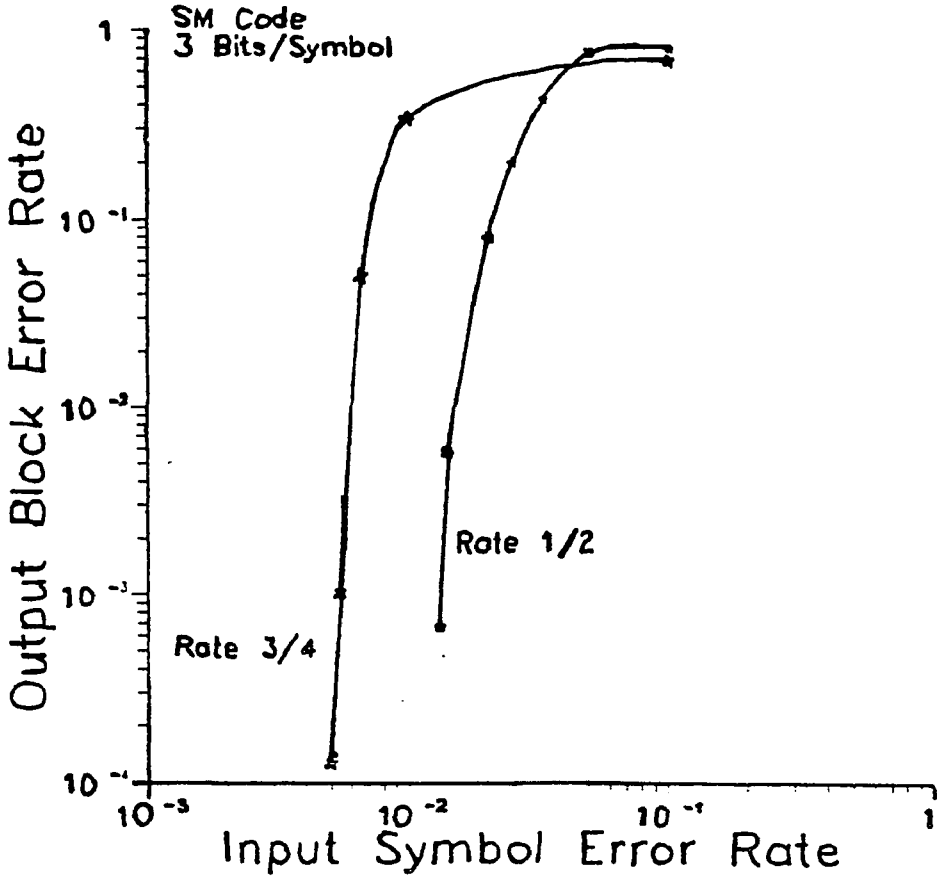
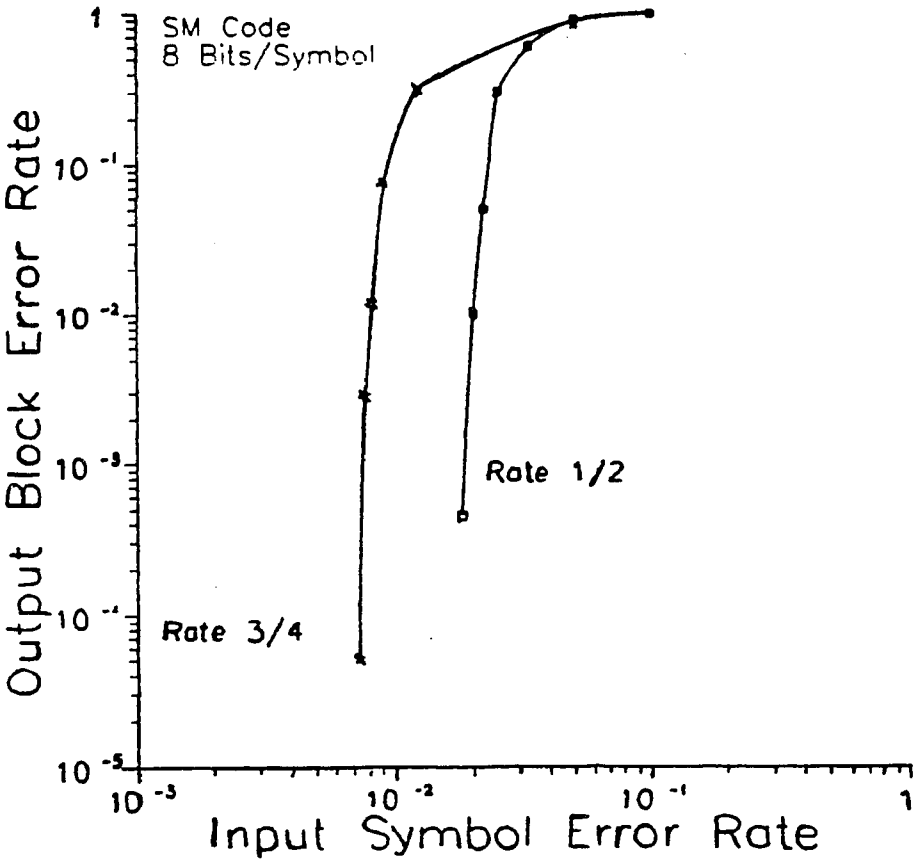


Figure 4.8 Output Block Error Rate Versus Input Symbol Error Rate.



4.2 THE PSM CODE

The simulation of the PSM code is another contribution introduced by this author. The simulation of the PSM code followed that for the SM code. At this point we were familiar of the limitations on the computer space and time available. To avoid facing the problems we encountered while simulating the SM code we used the same restrictions as in the SM code.

As discussed in Chapter 3 the PSM code is very similar to the SM code above. The main difference is that the PSM code offers some protection to the parity symbols while the SM code bid not.

In a PSM code every parity line encodes not only the data lines but also the parity lines before it. That is the first parity line encodes only the data lines , but the second parity line encodes the data lines and the first parity line, and so on.

As in the case of the SM code we will describe the encoding and decoding procedures for a specific example , the rate 1/2 PSM code with 3-data lines and 3-parity lines.

The Encoder

In the PSM code every parity line depends on the parity line ahead of it as mentioned above. This dependency dictates that we calculate the parity lines in order. As shown in Fig. 4.9 we first calculate the first parity line p_1 in the exact way we did for the SM code. The encoding equation for the i^{th} symbol in the first data line, $d_1(i)$ is as follows:

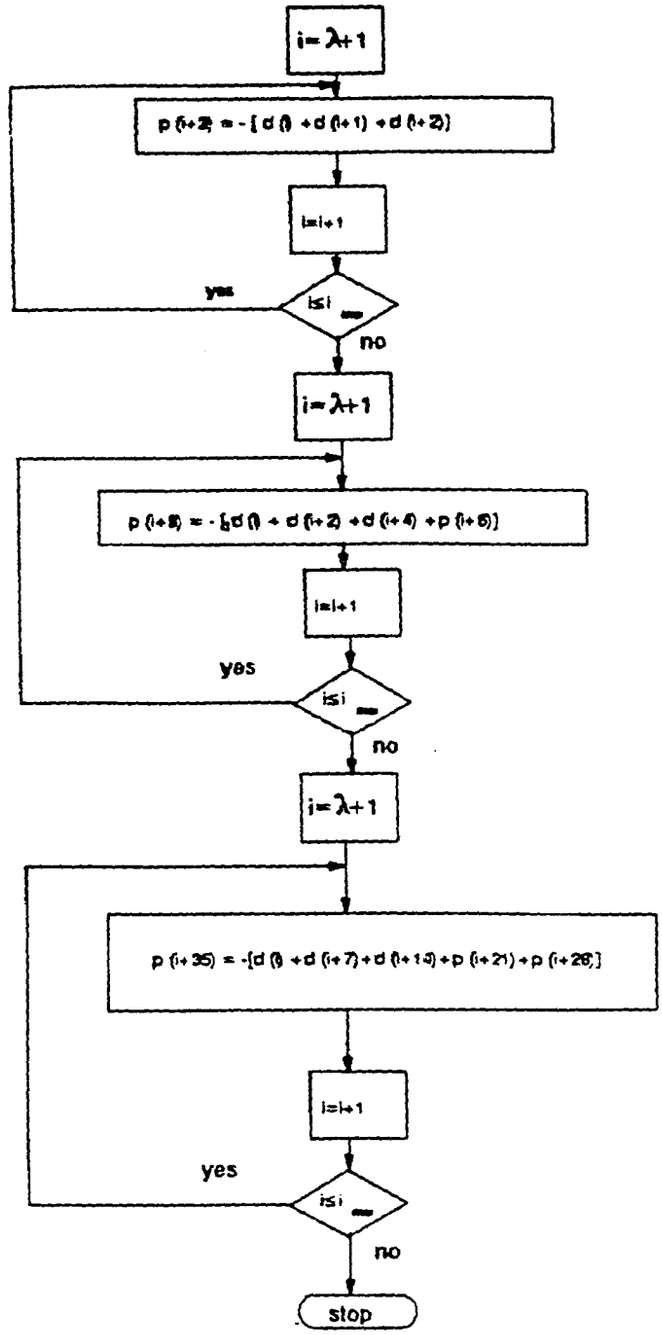
$$p_1(i+3) = -[d_1(i) + d_2(i-1) + d_3(i-2)] \qquad (4.8)$$

Once we calculate p_1, we can start calculating the second parity line, p_2. The second parity line is a function of the data lines and the first parity line. The encoding equation for the i^{th} symbol in the first parity line $d_1(i)$ is as follows

$$p_2(i-8) = -[d_1(i) + d_2(i-2) + d_3(i-4) + p_1(i-6)] \qquad (4.9)$$

Once p_2 is calculated, p_3 is then calculated in a similar manner except that it would be a function of the data lines and the first and second parity lines. In general we continue in the same manner until we calculate all r parity lines.

Figure 4.9 Flow Diagram for The Encoder of the Rate 1/2 PSM Code with 3-Data Lines and 3-Parity Lines



The Decoder.

The decoding procedure of the PSM code is very similar to that of the SM code, specially the detection part. Except that in the PSM code errors occurring on the parity lines will be detected by more than one slope as in the SM code. In the PSM code as mentioned in chapter 3, errors in the first parity line will be detected by r slopes , while errors in the second parity line will be detected by only r-1 slopes and so on, errors in the r^{th} parity line will be detected by one slope only.

The error correction procedure for The PSM is almost identical to that of the SM code , with two major differences. First, we attempt to correct errors in the parity lines. This is accomplished by letting j, the row index in figure 4.3 to attain values up to k+r instead of k for the SM code. Second, the decoding equations would have to be altered to correspond to the encoding equations used in the encoding

process. For the PSM code the equations used to calculate the error for the i^{th} symbol in the first data line are given by:

$$E_1 = -[d_1(i) + d_2(i-1) + d_3(i-2) + p_1(i-3)]$$
$$E_2 = -[d_1(i) + d_2(i-2) + d_3(i-4) + p_1(i-6) + p_2(i-8)]$$
$$E_3 = -[d_1(i) + d_2(i-7) + d_3(i-14) + p_1(i-21) + p_2(i-28) + p_3(i-35)] \quad (4.10)$$

The remainder of the process in identical to that of the SM code.

Simulation Results of the PSM Code

Using the simulation procedure described above, we simulated many PSM codes . Here are some of the results obtained.

Figures 4.10 and 4.11 compare the binary, 8- ary and 256-ary codes . Figure 4.10 compares the rate 1/2 codes , whereas, figure 4.11 compares the rate 3/4 codes. As in

the case of the SM code it is clear that the code performance improves by increasing m, the number of bit per symbol, hence, increasing the number of symbols in the code. As was mentioned in chapter 3 if m was allowed to approach infinity the probability of output error will approach zero. Reasons for limiting the study to a certain value of m were presented in chapter 3.

Figure 4.12 compares the rate 1/2 to the rate 3/4 PSM codes. It shows that when using the same number of parity lines, r, in this case r=3, the rate 1/2 has a lower output error rate. The reason for the improvement is that the rate 1/2 uses the same number of estimators to encode a smaller number of data lines, hence, a fewer number of uncorrectable errors patterns will form.

Figure 4.10 Output Block Error Rate Versus Input Symbol Error Rate

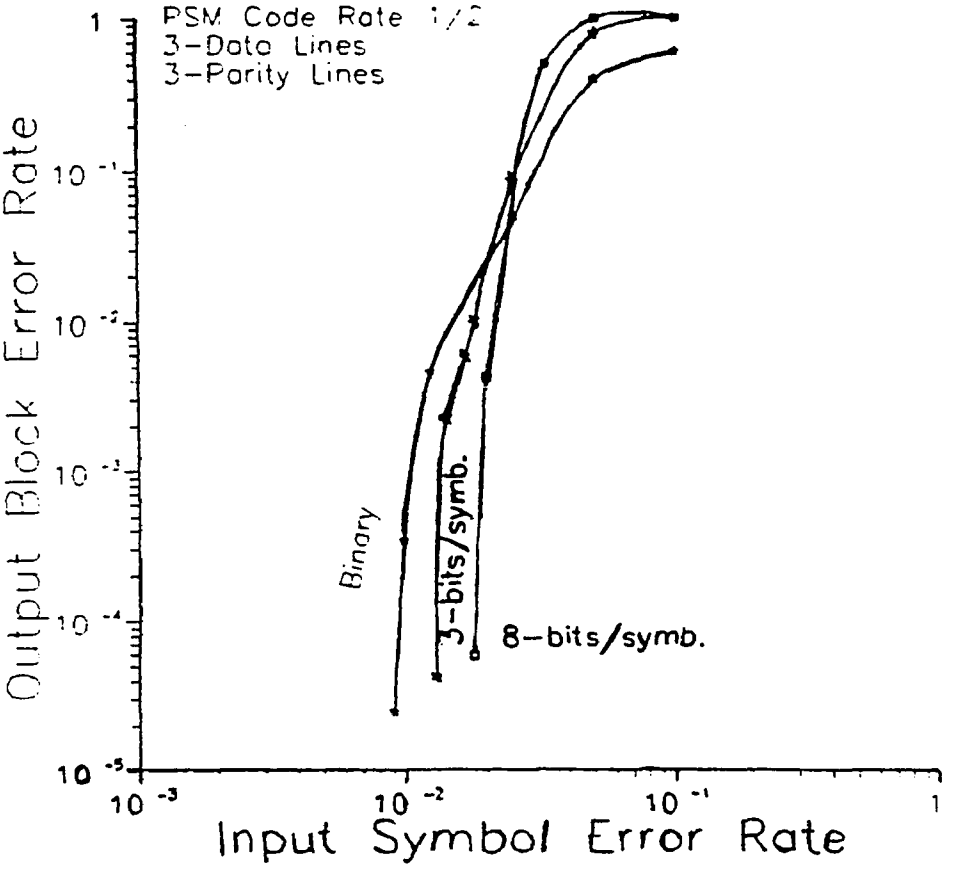


Figure 4.11 Output Block Error Rate Versus Input Symbol Error Rate

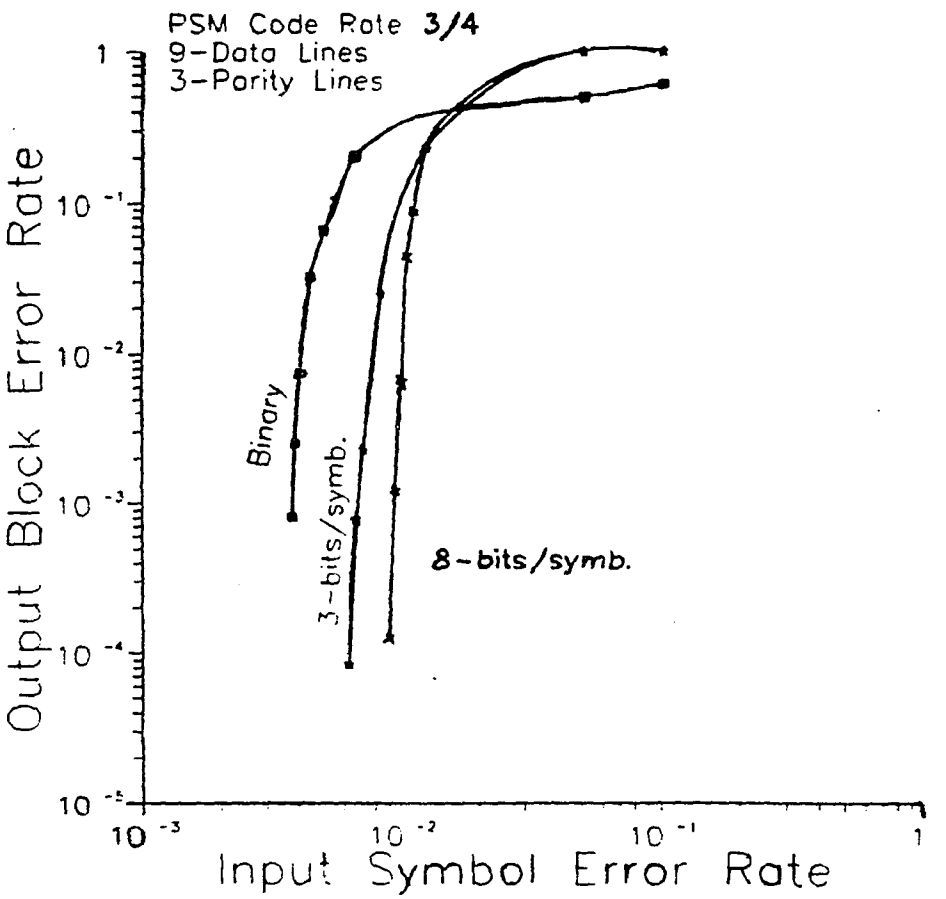
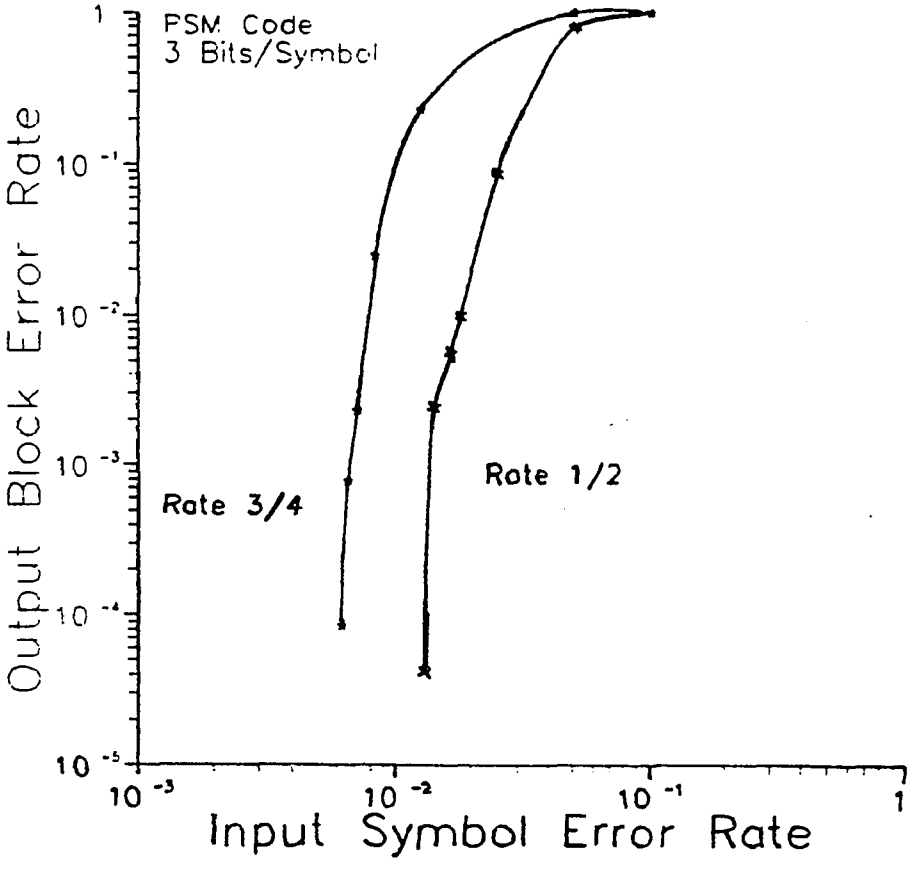


Figure 4.12 Output Block Error Rate Versus Input Symbol Error Rate



4.3 THE TASM CODE.

The TASM code is the most powerful of the three codes. The TASM code offers equal protection to the data lines and the parity lines, hence, it allows the correction of errors occurring in parity lines.

As for the previous two codes, the TASM code was simulated as a block code. We also did simulate the code as a convolutional code using the decoding algorithm discussed in section 3.1.1. The simulation procedures of the two types , the block and convolutional codes, are described below. The performance of the two procedures is discussed and then compared in the coming sections.

4.3.1 TASM BLOCK CODE.

The simulation of the TASM block code is similar to the SM and PSM codes described before. When a data block is received it is augmented by zeros as shown in figure

4.1. Except that for the TASM code the length of the all-zero blocks, λ, is longer than the length of the all-zero blocks needed for the SM and PSM codes. For the TASM code the length of the all-zero blocks should be, $\lambda = (k + r - 1)m_r$. When this augmentation with zeros is done no wrap-around is needed. When wrap-around is used the performance of the code will decrease since errors accruing around the edges may form uncorrectable error patterns which the augmentation by zero avoids. Of course, the augmentation by zeros decreases the code rate, but when h, the number of columns in the block is too large the change in the code rate can be neglected.

The Encoder

Since every parity line in the TASM code is dependent on all other parity line we would have to encode the data block in an exact manner. One way of accomplishing that is by the use of equation 3.1. Another way to encode the

TASM block code as described in figure 4.13. Here we start with the first symbol in the first data line and calculate the corresponding parity symbol in the first parity line. The encoding equation for our particular example will be as follows

$$p_1(15) = -[d_1(36) + d_2(29) + d_3(22) + p_2(8) + p_3(1)]$$

where $p_2(8)$ and $p_3(1)$ are assumed to be zeros. To avoid finding which parity symbols need to be set to zero we assume that every parity symbol of index $i < \lambda$ to be zero at the start. Usually the number of symbols that need to be set to zero is small. Some of the values that we set to zero at the beginning need not be set to zero, but their values to be calculated later are not affected by this initialization to zero.

The second parity symbol calculated will be the symbol in the second parity line along the line of slope 1/2 that passes through the symbol in the first parity line that was calculated just before it. The encoding equation to find this parity symbol will be

$$p_2(17) = -[d_1(9) + d_2(11) + d_3(13) + p_1(15) + p_3(19) \qquad (4.11)$$

Then we calculate the parity symbol in the third parity line along the line of slope 1 which passes through the last symbol calculated. The encoding equation for this symbol will be

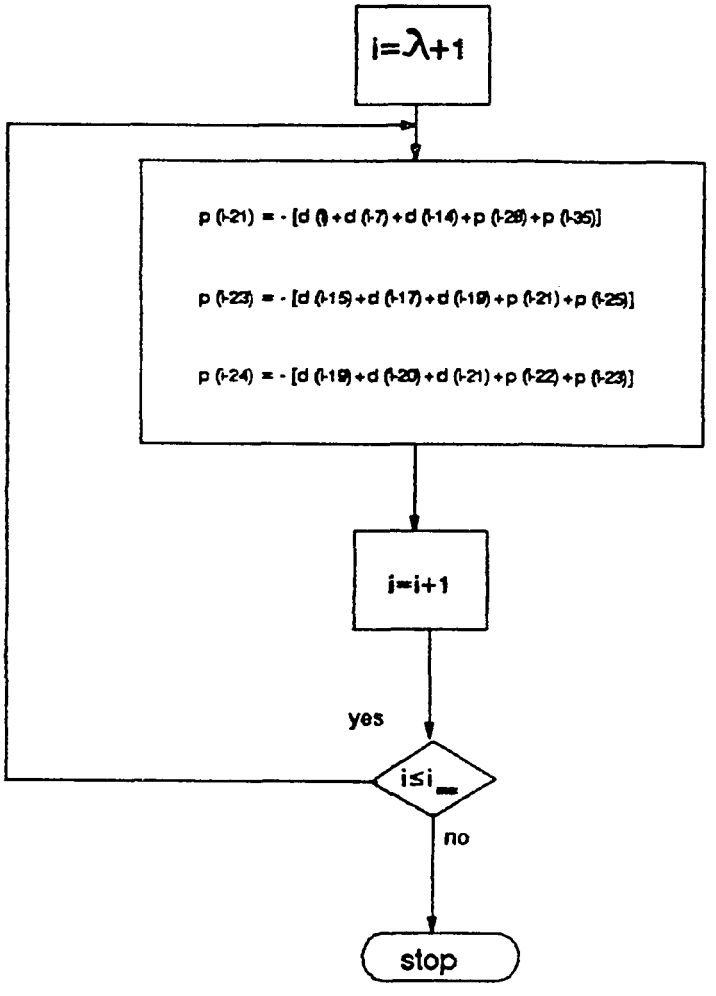
$$p_3(18) = -[d_1(13) + d_2(14) + d_3(15) + p_1(16) + p_2(17)] \qquad (4.12)$$

Note that we started this procedure with a line of slope 1/7 starting at the first data symbol in the first data line ,$d_1(\lambda + 1)$. The same procedure is then repeated until all data symbols are encoded. To encode the i^{th} symbol in the first data line the equations as shown in figure 4.13 would be

$$\begin{aligned}
p_1(i-21) &= -[d_1(i) + d_2(i-7) + d_3(i-14) + p_2(i-28) + p_3(i-35)] \\
p_2(i-23) &= -[d_1(i-15) + d_2(i-17) + d_3(i-19) + p_1(i-21) + p_3(i-25)] \\
p_3(i-24) &= -[d_1(i-19) + d_2(i-20) + d_3(i-21) + p_1(i-22) + p_2(i-23)]
\end{aligned} \qquad (4.13)$$

This encoding procedure is used whether the TASM code is used as a block code or as a convolutional code.

Figure 4.13 Flow Diagram for The Rate 1/2 TASM Block Code with 3-Data Lines and 3-Parity Lines.



The Decoder

The decoder for the TASM block code is almost identical to those of the SM and PSM codes. The only difference is in the decoding equations. Of course, the TASM code offers more protection for parity symbols actually it offers the same protection for all symbols in the block data and parity. Hence, errors in parity symbols will be detected by more equations and therefore can be corrected.

The decoding equations for the i^{th} symbol in the first data line in a TASM code are

$$E_1 = -[d_1(i) + d_2(i-7) + d_3(i-14) + p_1(i-21) + p_2(i-28) + p_3(i-35)]$$
$$E_2 = -[d_1(i-15) + d_2(i-17) + d_3(i-19) + p_1(i-21) + p_2(i-23) + p_3(i-25)]$$
$$E_3 = -[d_1(i-19) + d_2(i-20) + d_3(i-21) + p_1(i-22) + p_2(i-23) + p_3(i-24) \quad (4.14)$$

Simulation Results for TASM block Code

As for the cases of the two previous codes we simulated many TASM codes. The behavior of these codes with respect to the changing parameters r, the number of parity lines, and m, the number of bits per symbol is shown in figures 4.14, 4.15 , and 4.16 to be similar to the behavior of the SM and the PSM codes.

Figure 4.17 shows the 8-ary TASM codes with different rates all using 3-parity lines. Note that as expected the lower rate codes outperform the higher rate codes. Also note that the rate 9/10 code starts to show a coding gain around an input probability of error of 10^{-3} and that its curve is descending at a slope comparable to these of the rate 1/2 and 3/4 codes.

Bounds on the performance of TASM code were developed by a colleague, Yang Gang [21] for the rate 1/2 TASM binary code with 3-data lines and 3-parity lines. Figure 4.18

compares our simulation results to these bounds. It is clear that the code performs within the expected region. Note that the bounds were developed for an optimal receiver where our receiver is only sub-optimal. Also note that for low value of the input rate the simulation results move closer to the lower bound.

Figure 4.14 Output Block Error Rate Versus Input Symbol Error Rate

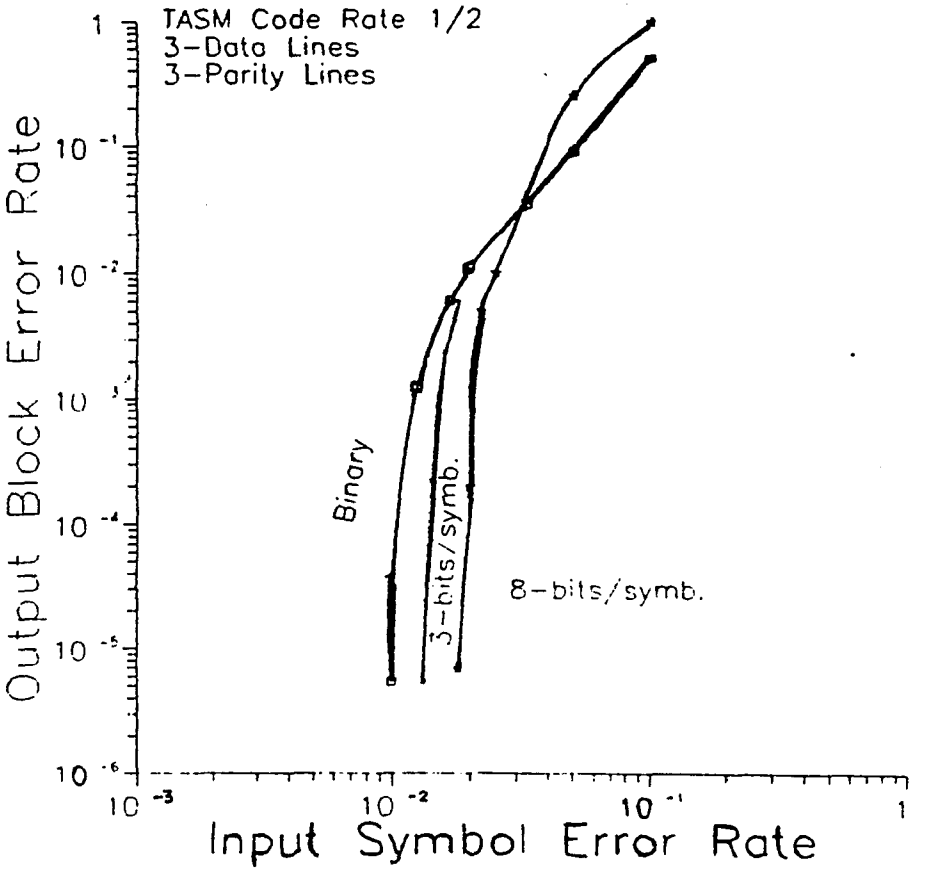


Figure 4.15 Output Block Error Rate Versus Input Symbol Error Rate

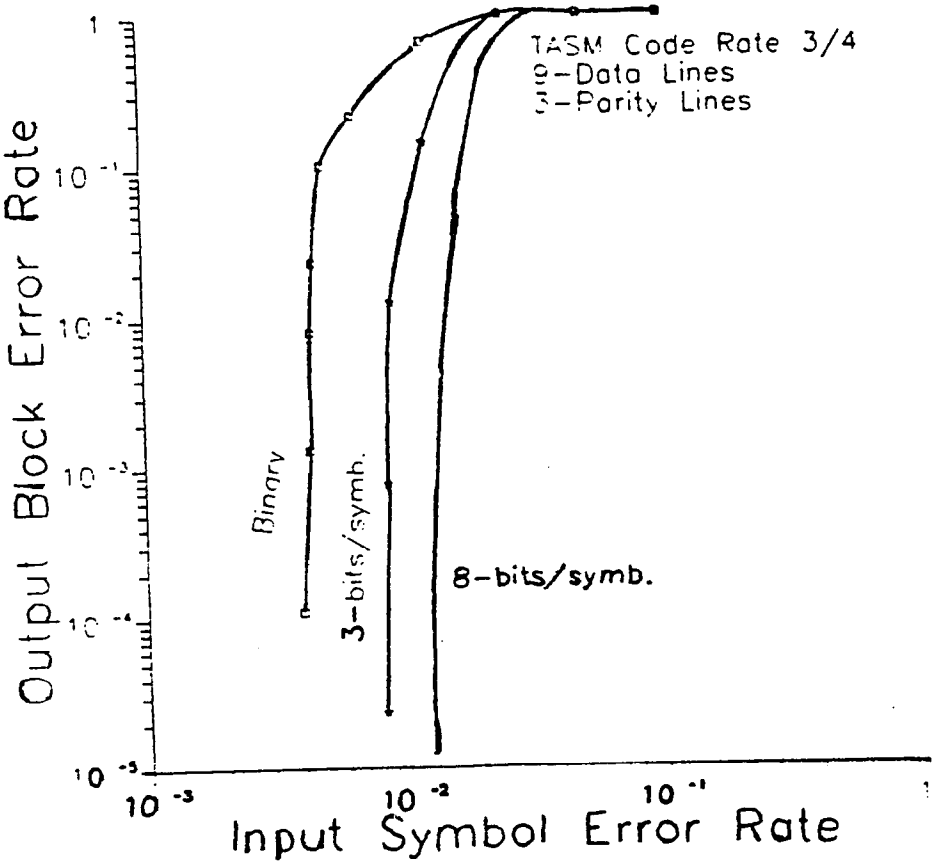


Figure 4.16 Output Block Error Rate Versus Input Symbol Error Rate

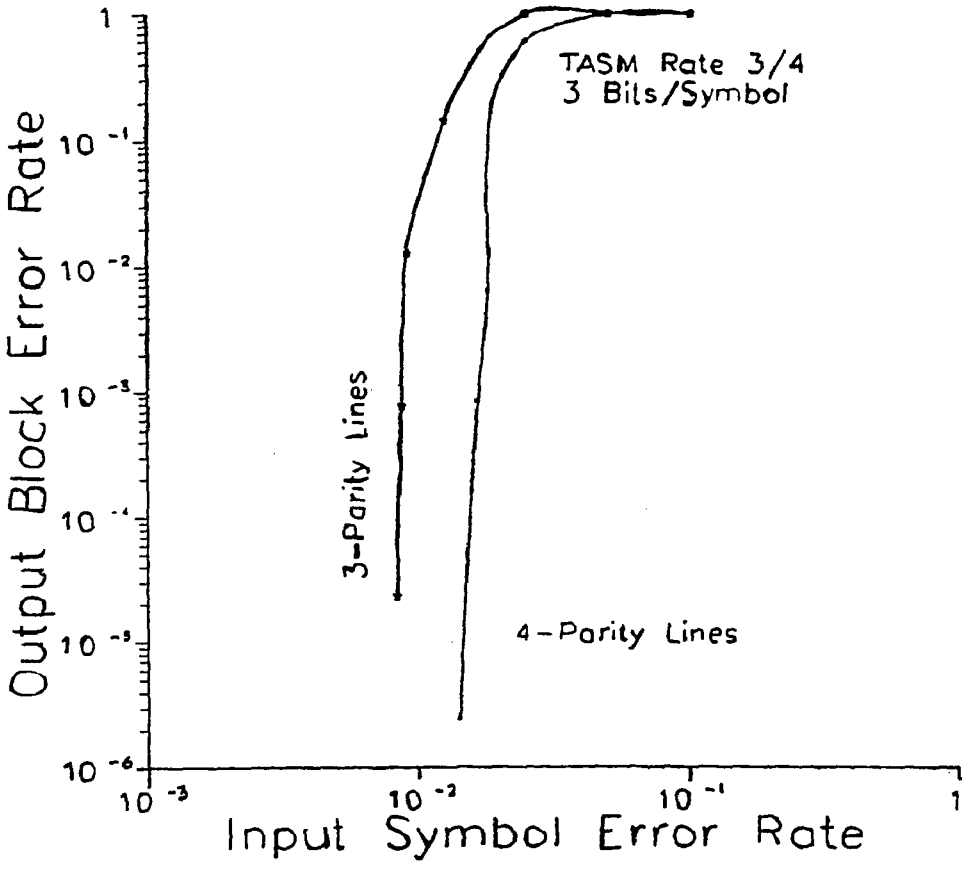


Figure 4.17 Output Block Error Rate Versus Input Symbol Error rate

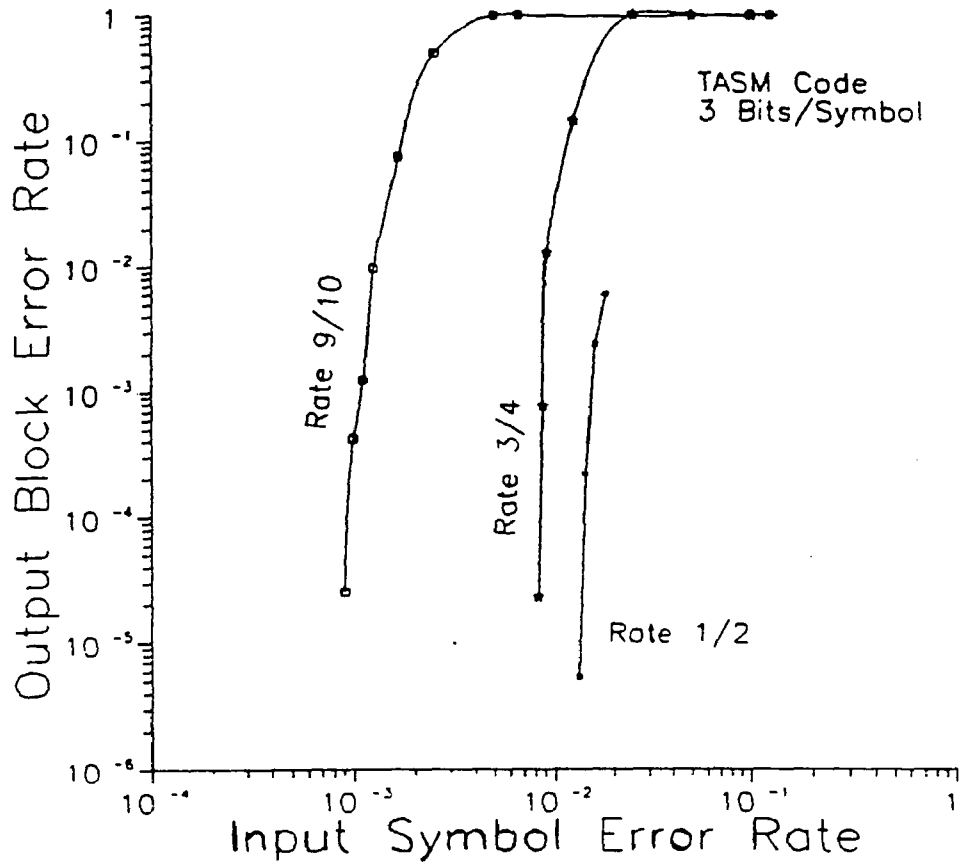
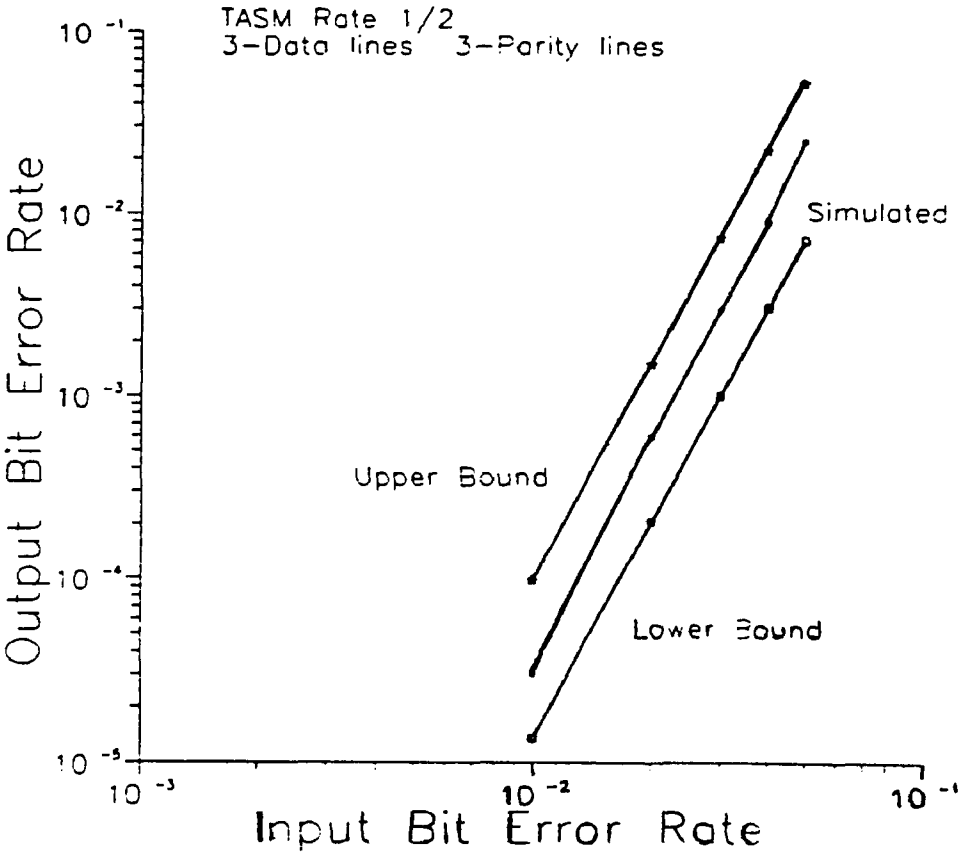


Figure 4.18 Output Bit Error Rate Versus Input Bit Error Rate



In Appendix A we present the simulation program for the communication system using a rate 3/4 TASM code with 9-data lines and 3-parity lines. By changing the value of m, the number of bits per symbol different codes can be obtained.

Comparing The Three Codes. SM, PSM and TASM.

In chapter 3 we argued that the TASM is the most powerful of the three codes , because it offers the most protection to parity symbols . Whereas the PSM code offers partial protection and the SM code offers no protection at all. Of course all three codes give the same protection to the data symbols. The following simulation results support our argument.

Figures 4.19 and 4.20 compare the performance of the three coding schemes. Figure 4.19 compares the rate 1/2 binary SM, PSM, and TASM codes . It shows that the TASM code out performs the PSM code which in turn out preforms the SM code. Figure 4.20 compares the rate 3/4 codes , it shows the same results as in figure 4.19. Figure 4.20a compares the three coding schemes to the Reed-Solomon

code. In this case a block of 155 bits was used for the RS code where block sizes of 168,192,192 were used for the SM, PSM, and TASM respectively. The figure shows that the Projection Codes specially the TASM and the PSM start to outperform the RS code at an input Error rate of about 10^{-2}. Also the Projection codes have steeper slopes and hence, would have larger coding gains as the input error rate decreases beyond 10^{-2}.

Figure 4.19 Output Block Error Rate Versus Input Symbol Error Rate

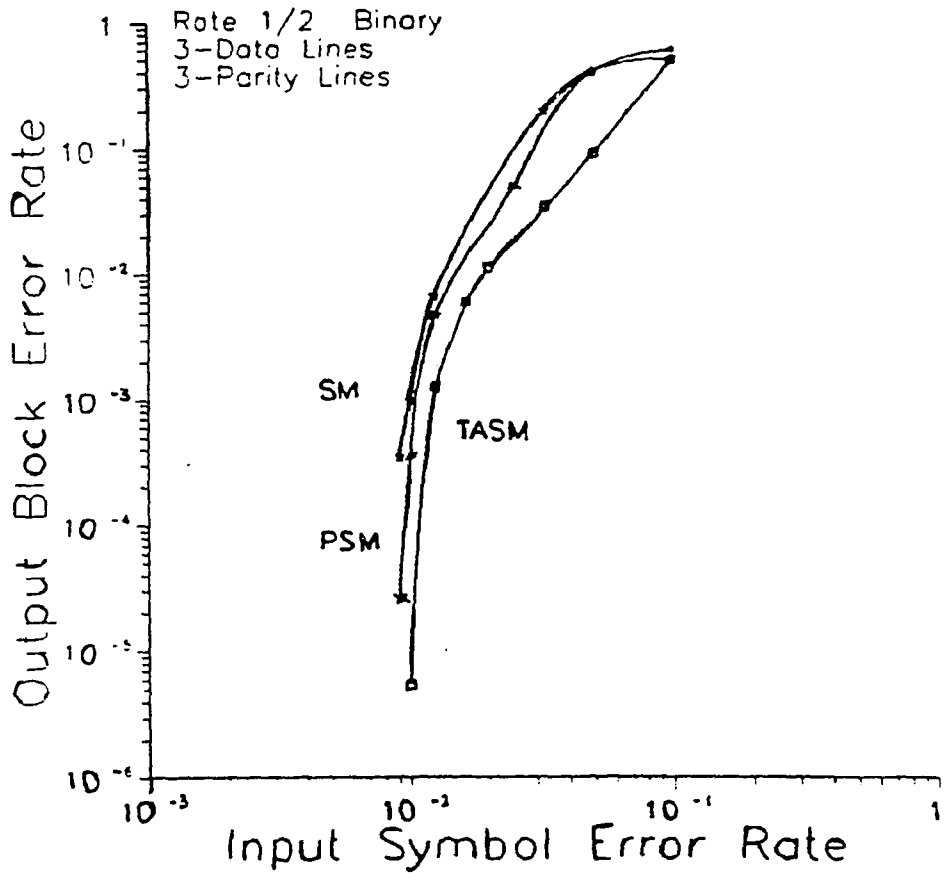


Figure 4.20 Output Block Error Rate Versus Input Symbol Error Rate

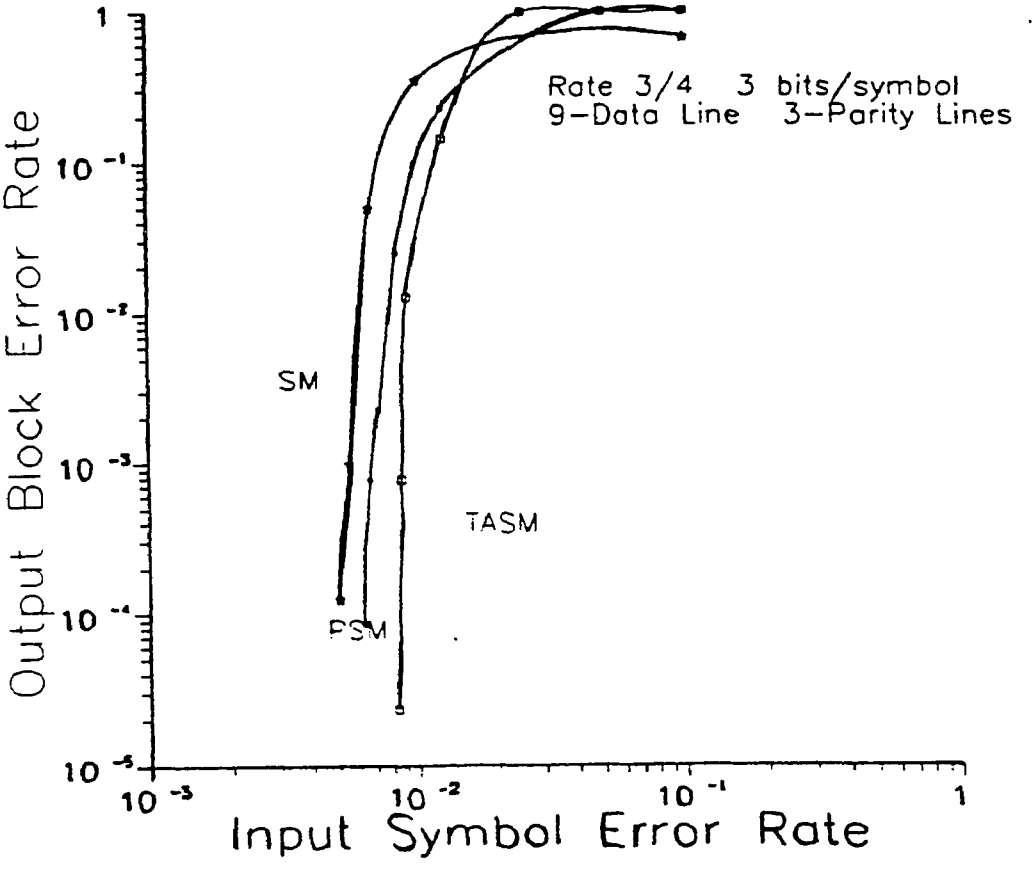
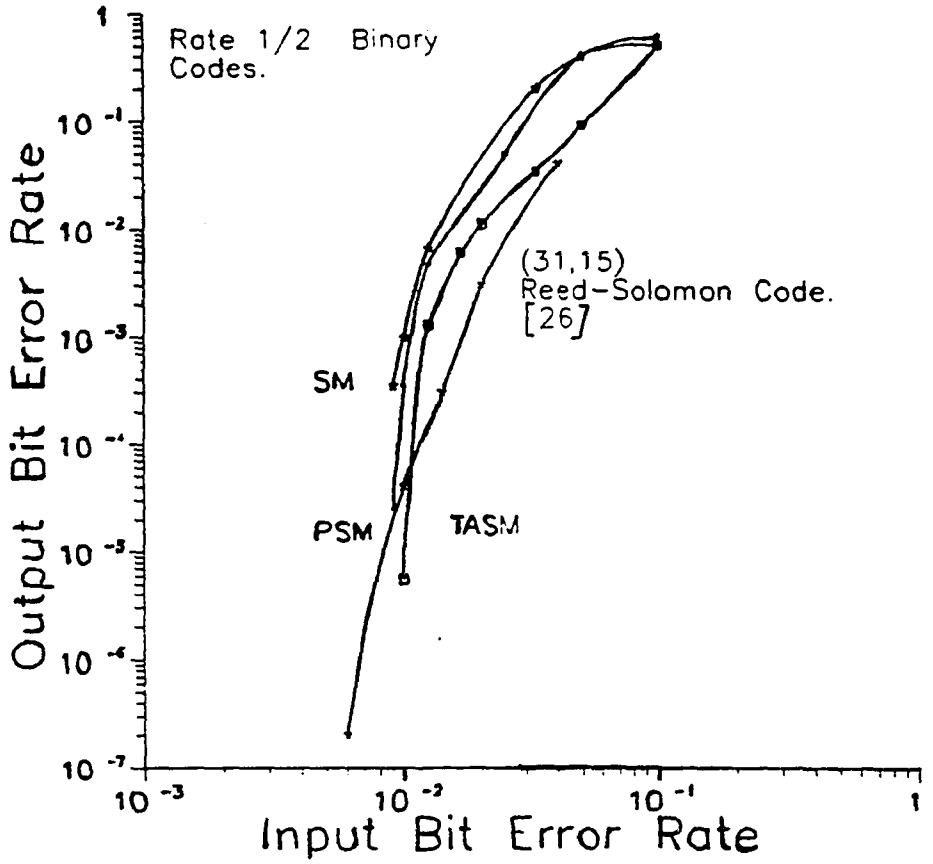


Figure 4.20a Output Bit Error Rate Versus Input Bit Error Rate



4.3.2 THE TASM CONVOLUTIONAL CODE

In the previous discussion it was assumed that the entire message was received and stored in a buffer before we started the decoding process. Under this assumption the decoder wastes a considerable amount of time waiting for the reception of the entire message . In this section we examine the convolutional decoder in section (3.1.1) in which we do not wait for the entire message to be received. In this case the decoder waits for a certain amount of data to be received first, then it proceeds to decode sequentially. The reason for the initial amount of data is the dependency of data symbols on some parity that will be received at a later time. The amount of data that should be received before decoding starts depends on the rate of the code and the number of estimators which govern the slops used in the encoding process.

In the previous section we established that the TASM class of codes is superior to the PSM and the SM classes,

figures 4.18 and 4.19 demonstrate that. Therefore we will concentrate our discussion on TASM codes.

In simulating the convolutional TASM code we started by encoding a small block of data mixed it with noise .This initial block is assumed to be received by the and stored into the shift register bank . The length of this initial block should be enough to include all symbols affecting the first column. Then we attempt to decode the first column transmitted. Once this is accomplished a new column is then encoded mixed with noise and then, inputted into the shift register bank so that the second column can be decoded and so on. The process is repeated until 10 decoding errors are detected, and then the probability of error is computed.

The Encoder

Another contribution by this author is the simulation of the TASM covolutional code. As mentioned in chapter 3

this algorithm was first developed by Emmanual Kanterakis in [6] for binary codes. We then extended it to include non-binary codes.

The encoder for the TASM convolutional code is very similar to that of the block code. As a matter of fact, the only difference is that in this case the encoding is done one column at a time as they are needed. The encoding equations for the convolutional code are exactly the same as for the TASM block code.

The Decoder.

The major difference between the block and convolutional TASM codes is in the decoding procedure.

Again we will describe the decoding procedure for the rate 1/2 TASM convolutional code with 3-data lines and 3-parity lines.

As discussed in chapter three the TASM decoder consists of three stages . The three stages are almost identical,

actually, the only difference between them is that each uses a different threshold on the EP value when correcting errors. Hence, we will discuss the first stage only.

The decoding process is again split into two major components, one for error detecting and the other for error correcting. We will start by introducing the error detecting procedure.

Error detection is accomplished as shown in figure 4.21. Again we start with the first data symbol in the first data line and try to detect errors on the column containing it. That is we try to detect errors in the following symbols

$,(d_1(\lambda+1),d_2(\lambda+1),d_3(\lambda+1),p_1(\lambda+1),p_2(\lambda+1),p_3(\lambda+1))$.

We chose to first check along all lines of slope one, then lines of slope 1/2 , and finally along lines of slope 1/7. This was done as follows: We start with the first data symbol in the first parity line and check if the sum of the data and parity symbols along a line of slope 1 starting from this symbol is equal to zero or not. If it is then we assume that $d_1(\lambda+1)$ is not in error. If the sum is not

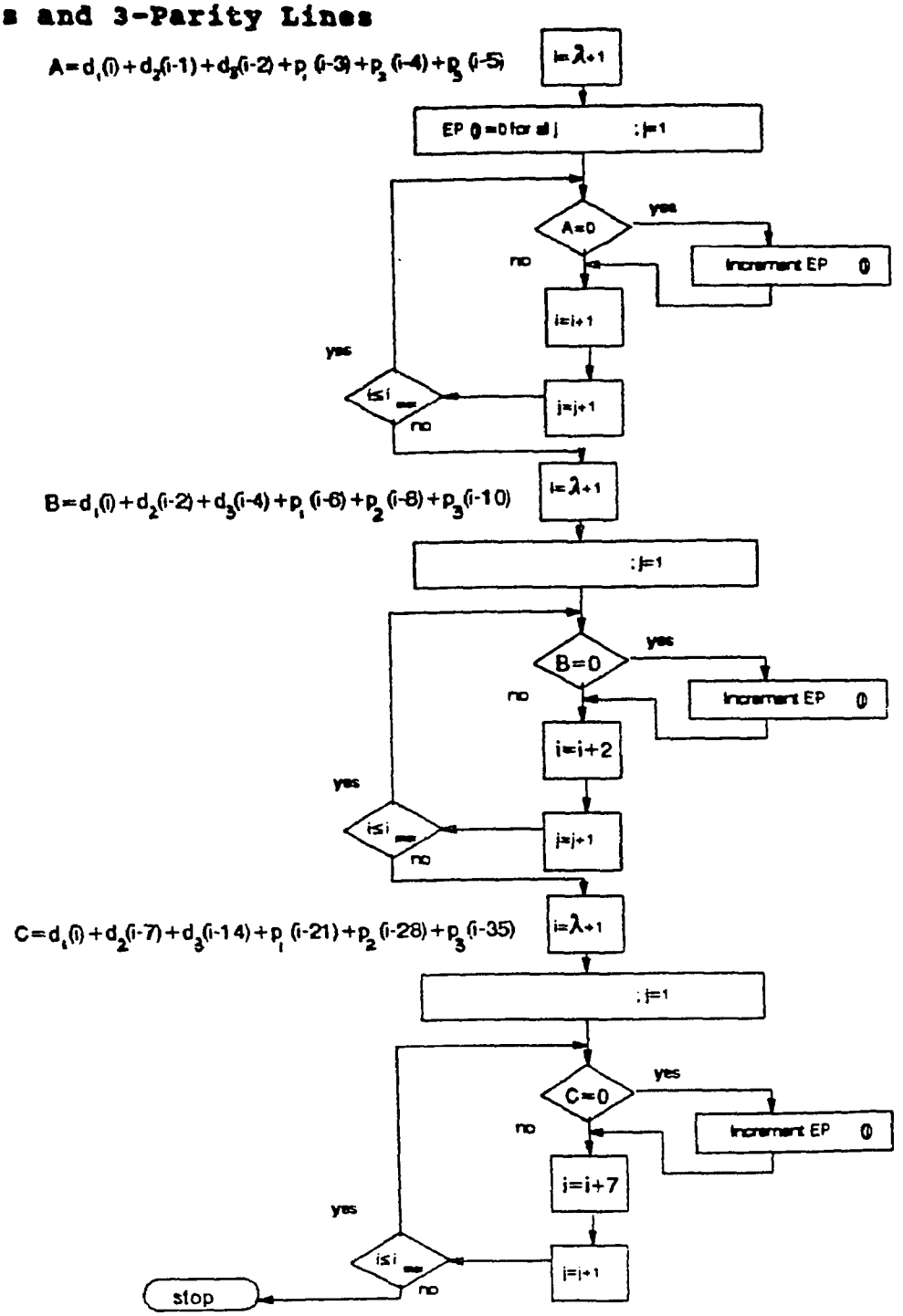
zero then we assume that the symbol may be in error and the EP value corresponding to that symbol is incremented. Note that the line of slope one that passes through the first data symbol in the second data line, $d_2(\lambda+1)$ starts at the second data symbol in the first data line, $d_1(\lambda+2)$, the line of slope one for $d_3(\lambda+1)$ starts at $d_1(\lambda+3)$ and so on. Therefore, we check lines of slope one starting at $d_1(\lambda+1), d_1(\lambda+2), \ldots, d_1(\lambda+6)$.

To detect errors along lines of slope 1/2 passing through symbols in the column being processed we notice that these lines start at $d_1(\lambda+1), d_1(\lambda+3), d_1(\lambda+5), \ldots, d_1(\lambda+11)$. We check if errors have occurred along these lines every time we make a decision only on the symbol in question, i.e. $d_1(\lambda+1), d_2(\lambda+1), \ldots, p_3(\lambda+1)$.

To detect errors along lines of slope 1/7 passing through symbols in the column in question, we notice that these line start at $d_1(\lambda+1), d_1(\lambda+8), d_1(\lambda+15), \ldots, d_1(\lambda+36)$, hence the increment of the index of d_1 by 7 in figure 4.20.

After the detection process is completed we check the entries of the EP vector when any entry of value of three is found the symbol is determined to be in error and a correction attempt is made. Calculating the error value for a specific symbol is done in exactly the same way as done in the TASM block code. The decoding equations for the i^{th} symbol in the first data line are given in equation 4.14. Once all entries of EP are checked and attempts are made to correct any symbol with corresponding EP value of three the corrected version of the column is then inputted into the second stage. Note that the original column is left in the first stage without changes. Then the process is repeated for the next column and so on. The second stage is identical to the first stage except that we attempt to correct all symbol with an EP value of 2 or more. The output of the second stage is inputted in the third stage. The third stage being the last stage the error correcting process is repeated until the maximum number of iterations is reached or of course, when no errors are detected.

Figure 4.21 Flow Diagram for The Error Detecting Portion of the Decoder of The TASM Convolutional Code. With 3-Data Lines and 3-Parity Lines



Simulation Results for The TASM Convolutional Code.

We simulated many TASM convolutional codes and here are some of the results obtained. Figure 4.22 compares the rate 1/2 codes with 3-data lines and 3- parity lines. It shows considerable improvement by using non-binary codes. It also shows that by increasing m, the number of bits per symbol, the output error rate decreases. These results are similar to those shown earlier in figure 4.14 for the TASM block code. Figure 4.23 compares the rate 1/2 to the rate 3/4 codes with 3-parity lines. As in the block code case it shows that the rate 1/2 outperforms the rate 3/4 when the number of parity lines is fixed. Figure 4.24 compares the 8-ary rate 3/4 convolutional codes, in one case 3-parity lines are used and in the second case 4-parity lines are used. Note that the code with 4-parity lines shows an improvement over the code with 3-parity lines even though the rate is the same for both cases. These results are similar to those for the TASM block code shown in figure 4.16.

Figure 4.25 compares the simulation results for the binary TASM convolutional code to The upper and lower bounds on the performance of the code introduced by a colleague Dong Li in [20] . It is easily seen that for reasonable error rates the The simulated results are approaching the lower bound. Note that the bounds were derived for an optimal decoder while ours is only sub-optimal.

Figure 4.22 Output Symbol Error Rate Versus Input Symbol Error Rate

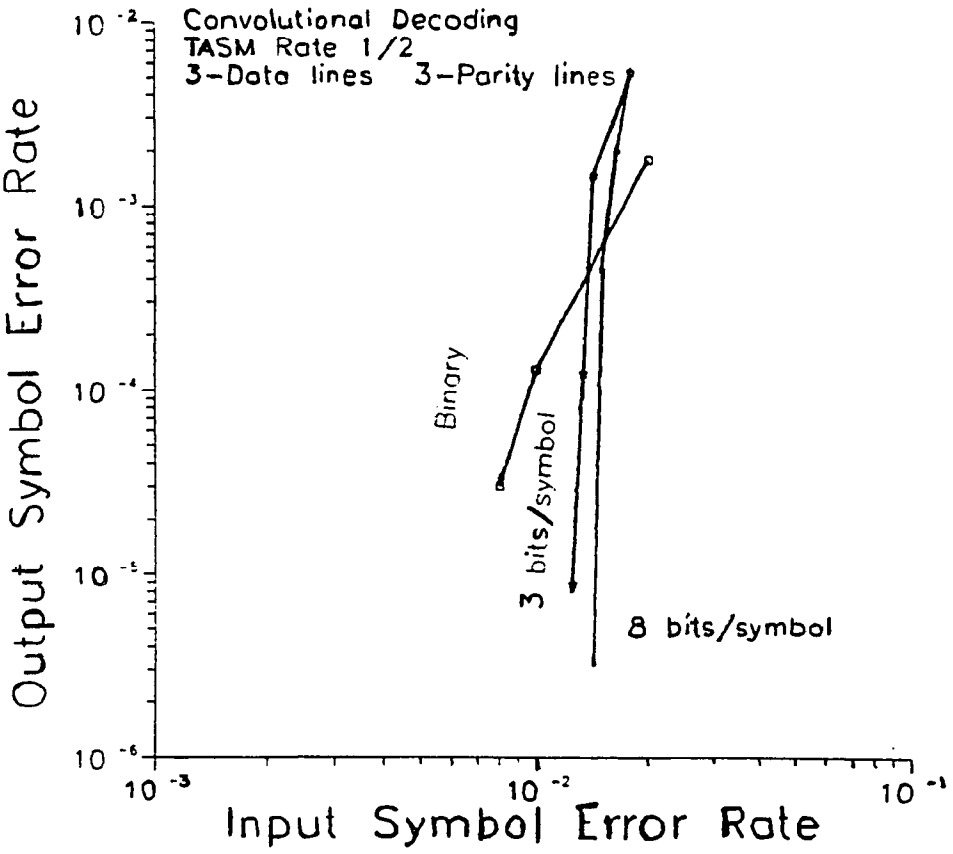


Figure 4.23 Output Symbol Error Rate Versus Input Symbol Error Rate

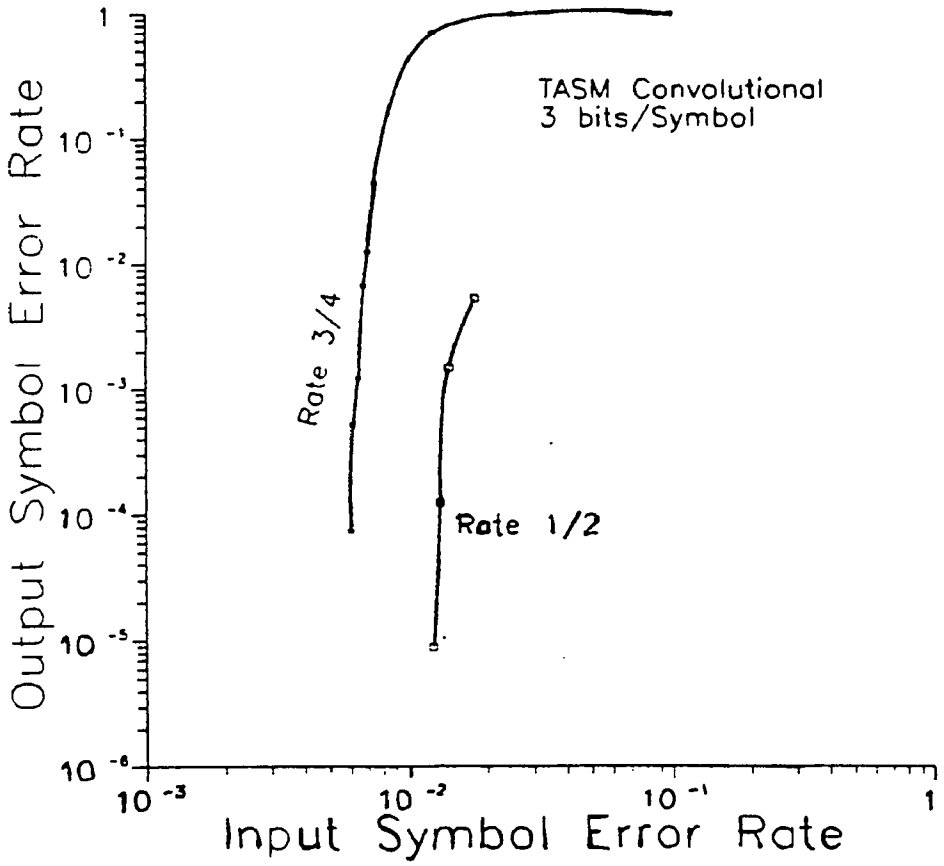


Figure 4.24 Output Symbol Error Rate Versus Input Symbol Error Rate

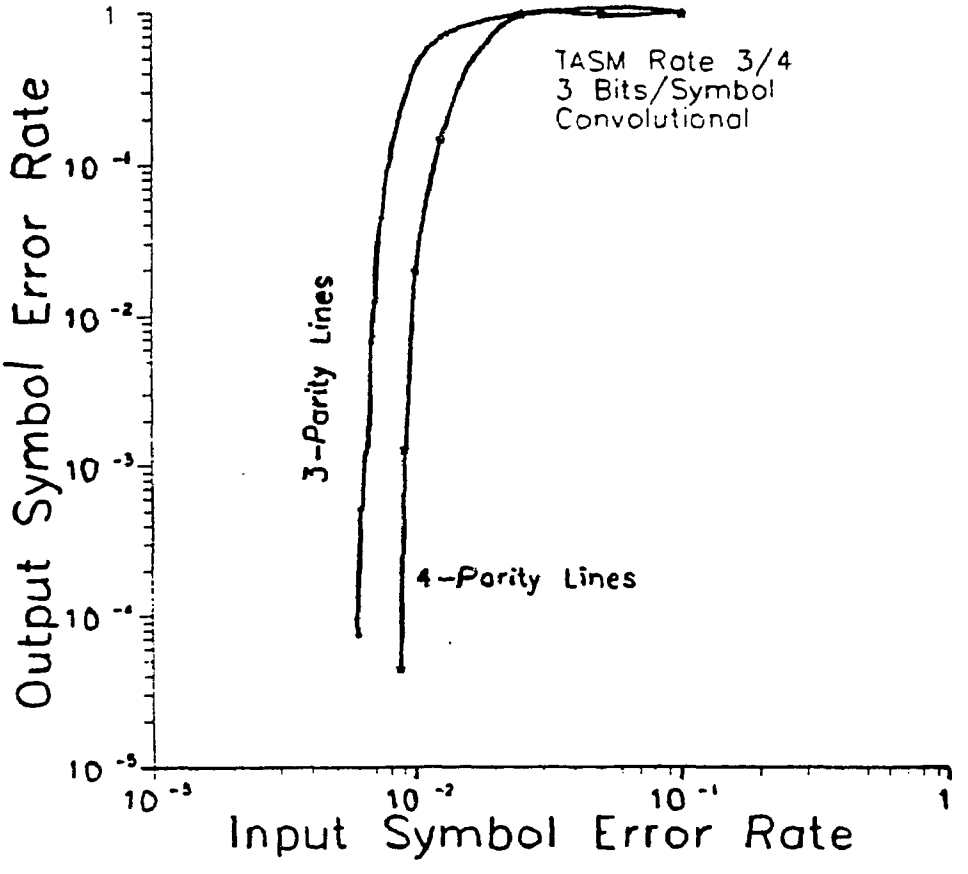
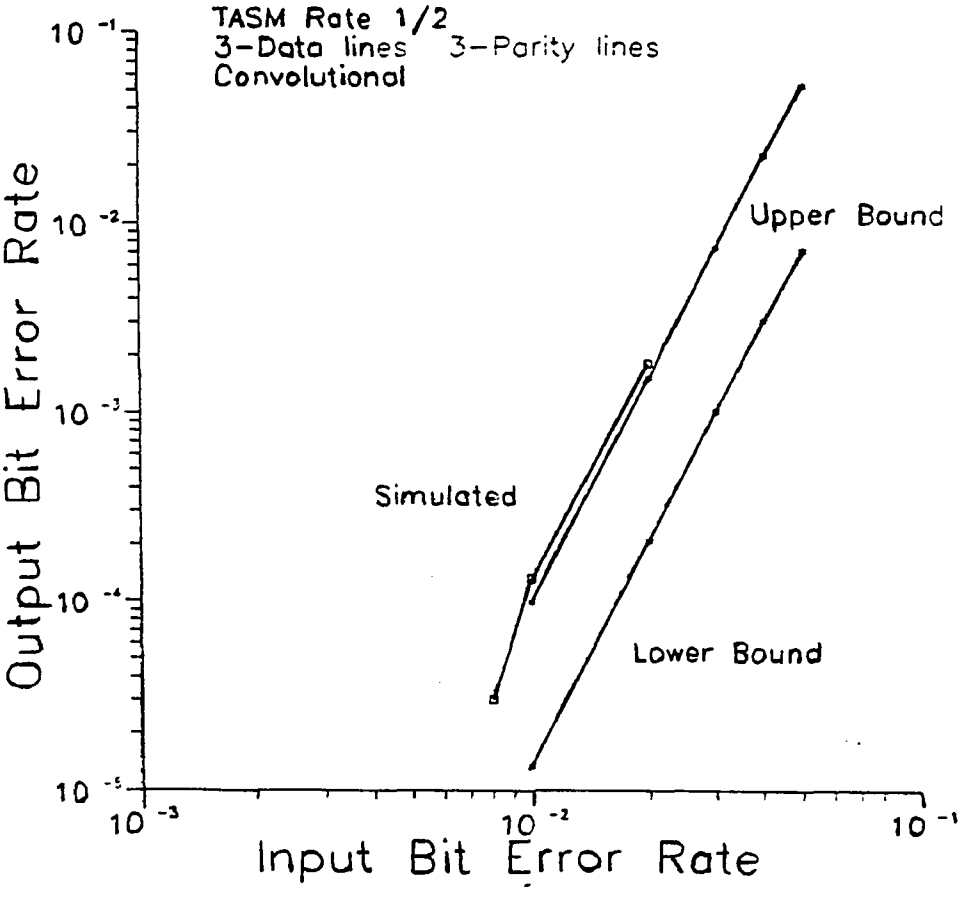


Figure 4.25 Output Bit Error Rate Versus Input Bit Error Rate



4.3.2.1 Comparing TASM Block and Convolutional Codes

In this section we compare the TASM block codes and the TASM convolutional codes. It is expected that the block code out performs the convolutional code, for in the first case the entire message is received before the start of the decoding process, whereas only a small part of the message is received in the convolutional code. The additional information used in the decoding of the block code is responsible for the lower error rate. Figures 4.26,4.27, and 4.28 show that the block code outperforms the convolutional codes. Figure 4.26 compares the 8-ary rate 1/2 codes with 3-data lines. Figures 4.27 and 4.28 compare the 8-ary rate 3/4 codes. Figure 4.27 uses 3-parity lines whereas figure 4.28 uses 4-parity lines.

It most be noted that the convolutional codes though outperformed in error rate sense by the block codes, are

recommended for tow major reasons: First, the convolutional codes do not waste time waiting for the entire message to be received, therefore the waiting time is reduced. This is very important with regard to voice communications. Second the buffer size for the convolutional codes is much smaller , because they do not need the entire message to be stored for the decoding process to start.

Figure 4.26 Output Symbol Error Rate Versus Input Symbol Error Rate

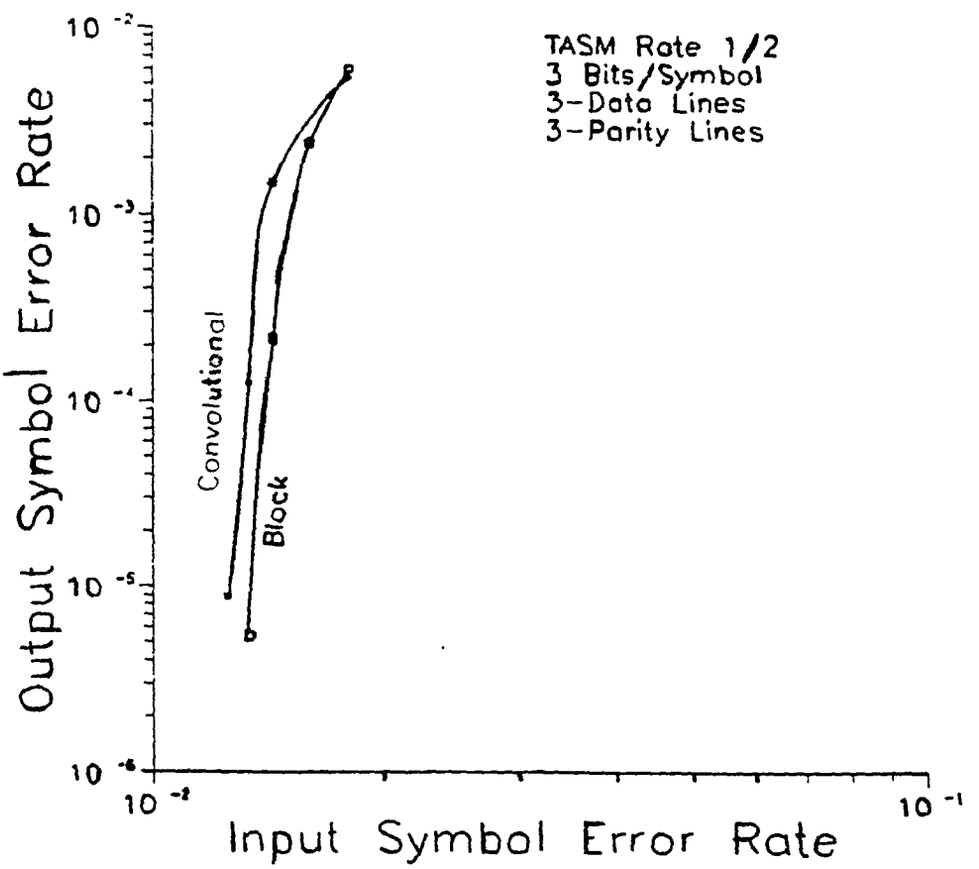


Figure 4.27 Output Symbol Error Rate Versus Input Symbol Error Rate

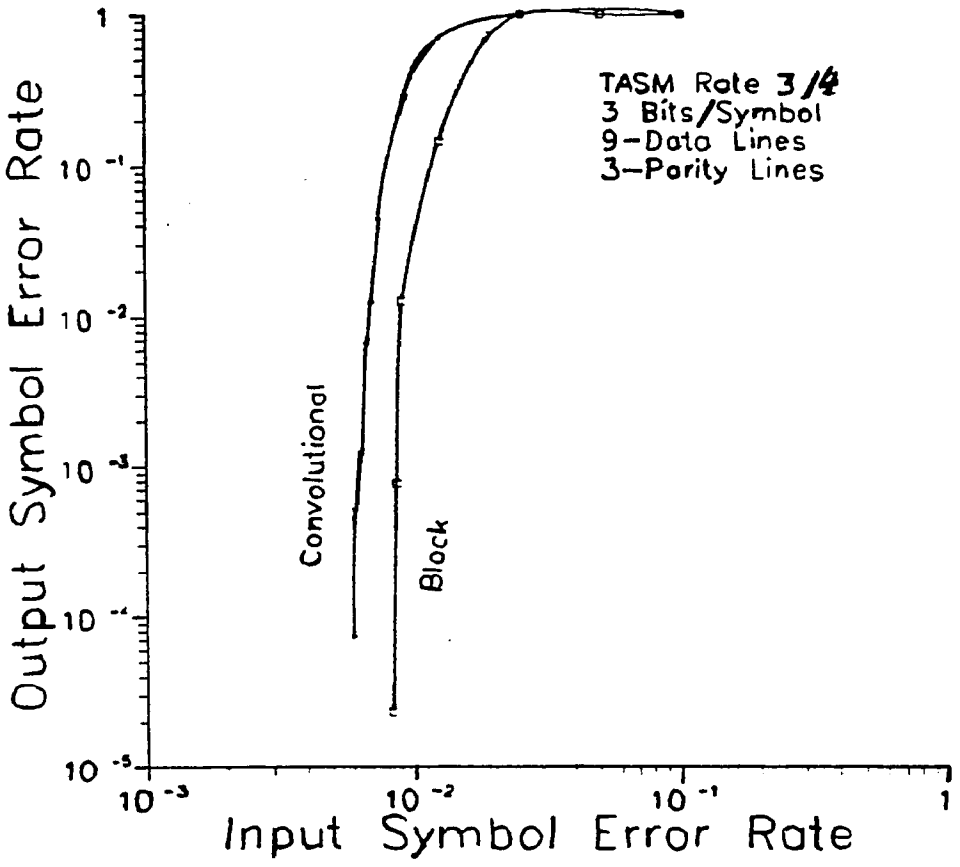
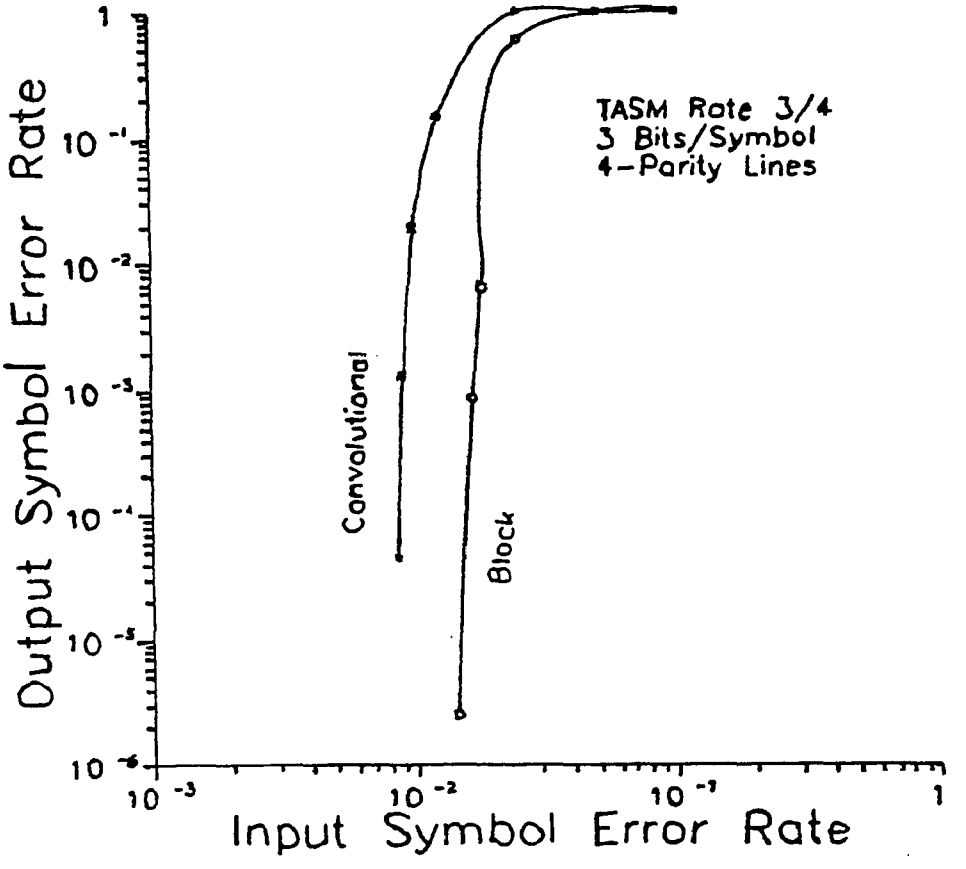


Figure 4.28 Output Symbol Error Rate Versus Input Symbol Error rate



5 HARDWARE IMPLEMENTATION

Another contribution by this author is the hardware implementation for the 8-ary rate 1/2 TASM convolutional code with 3-parity lines and 3-data lines. This implementation can be easily modified for other TASM convolutional codes. Figure 5.1 presents the first stage of this system. Other stages are almost identical to the first stage , therefore, we restrict our discussion to the first stage only.

This implementation employs the decoding algorithm discussed in section 3.1.1 . As discussed earlier as the data is received it is stored in a shift register bank (SRB) , The shift register bank is assumed to have all-zero entries before the start of the reception. The size of the shift register bank is dependent on r, the number of parity lines, k, the number of data lines and the slopes used to encode the signal. For our example r=3, k=3 , and the slopes used are m_1=1, m_2=2, and m_3=8. The width of the shift register bank is clearly equals k+r . To find the length of the bank in figure 5.1 note that

for a particular vector to be decoded we need a line of slope m_3 starting at the first symbol in the vector and a line of slope m_3 ending with the last symbol in the vector. Hence, the length of the shift register bank , $w=2(k+r)m_3$. In the case in question $w=2(6)8=96$.

Once the initial amount of data is received, we can start the decoding process. As discussed in chapter 3 the decoding algorithm decodes one vector at a time. Hence, we will describe the decoding procedure for one vector. To decode the whole message the procedure is repeated for every vector in the transmitted. Furthermore, we chose to decode symbols in the vector in parallel. Therefore the procedure is the same for all symbols and we will discuss the decoding of only one symbol.

To decode a particular symbol in the vector being decoded, lines of all slopes used, in the example 1, 1/2 , and 1/8 , passing through the symbol in question are drawn across the shift register bank. Registers along each line are tapped and the contents of the registers are inputted into a modulo 8

adder. Note that for each symbol in the vector 3 slopes, corresponding to the 3-parity line, are used, hence, 3 mod. 8 adders are needed for each symbol.

If the outputs of all 3 modulo 8 adders are non-zero, then the symbol is assumed to be in error. While if only one output is zero, then the symbol is stored unaltered into the temporary register to be sent to stage 2. When a symbol is assumed to be in error i.e. if the outputs of all modulo 8 adders are non-zero, the error correcting circuit is activated and an attempt to correct the symbol is made. The error correcting circuit is a majority logic network which works as follows. If the outputs of 2 or more outputs are equal then the error is taken to be equal to them as well. Hence, two or more decoding equations are satisfied. Once the error value is found it is subtracted from the symbol and the result is inputted into a temporary register to be inputted into stage 2.

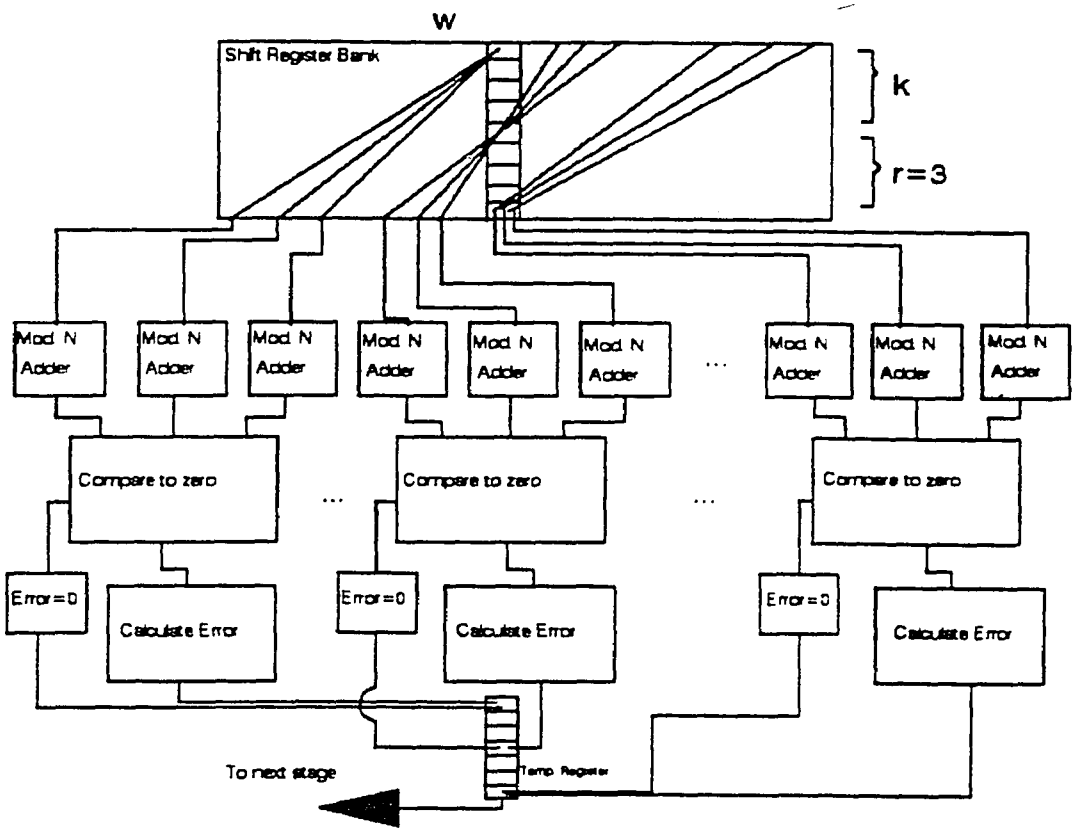
Once the decoding of all symbols in the vector is completed the contents of the temporary register are inputted into stage 2. Stage 2 is almost identical to stage one. The only difference

is the threshold on the number of modulo 8 adders outputs that need to be non-zero for the correction process to take place. In stage one all 3 outputs needed to be non-zero , while in stage two if two or more outputs are non-zero the symbol is assumed to be in error and an attempt to correct the error is made in the exact way as in stage one.

The output of stage two is then inputted into stage 3 whose output is then accepted as the decoded message.

Simulation results for this decoder are shown in section 4.3.2.

Figure 5.1 Hardware Implementation of 8-ary TASM Convolutional Code.



6 CONCLUSION

This dissertation has been directed towards the study of the performance of Projection Codes. Projection Codes are a new class of codes which can be implemented either as block codes or as convolutional codes.

In the second chapter we reviewed the most known coding systems and their performance. Most codes can be classified in one of three types of codes: Block codes in which the message is segmented into vectors which are encoded independently by adding a certain number of parity symbols. Convolutional codes in which the message is left as a whole ; parity symbols are inserted within the information symbols according to some rule . ARQ codes which are mainly used for computer communications and are in general error detecting codes. If a message was determined to have been received in error it will be repeated according to some rule until it is assumed to be received correctly.

Codes presented in this chapter perform best when the input error rate is low. Projection Codes presented in third chapter

outperform most of these codes for high input error rates (error rates between 10^{-2} -10^{-3}).

In the third chapter we reviewed the operation of Projection Codes. It was shown that there are three different classes of Projection Codes: SM ,PSM , and TASM . The three classes differ from each other by amount of protection they offer for the parity symbols. The three classes can be implemented as either block or convolutional codes. The coding and decoding procedures for all classes of codes were presented and some theoretical description of the most powerful of the three classes namely TASM was shown.

In the fourth chapter we presented some of the major aspects of our contribution which were the simulation results for all three classes of Projection Codes. We determined how block codes and convolutional codes perform and compared their error rates. We also showed the effects of all parameters on the error rates.

Finally we developed a hardware implementation for the 8-arry TASM, rate 1/2 convolutional code.

It was shown that though Projection Codes are not optimal they outperform many codes, particularly at high error rates. It was also shown that Projection Codes offer simple coding and decoding implementations. Because of their flexibility a Projection Code can be tailored for the use in many communication application.

7 APPENDIX A

In this appendix we present the simulation program for rate 3/4 block TASM code which employs 3- parity lines and 9-data lines . In this program we simulated the encoder, the channel and the decoder. This simulation program will also calculate the output error rate.

```
****************$JOB
C* SLOPES 1,2,13
C*
C* 9 DATA LINES , 3 PARITY
C*
C*****************************************************************
          INTEGER X1(10000),X2(10000),X3(10000),X4(10000),X5(10000)
          INTEGER X6(10000),X7(10000),X8(10000),X9(10000)
          INTEGER S1(10000),S2(10000),S3(10000),S4(10000),S5(10000)
          INTEGER S6(10000),S7(10000),S8(10000),S9(10000)
        INTEGER XW1(10000),XW2(10000),XW3(10000),XW4(10000),XW5(10000)
        INTEGER XW6(10000),XW7(10000),XW8(10000),XW9(10000),XP1(10000)
          INTEGER PL1(12000),PL2(12000),PL3(12000),XP3(10000),SL3(12000)
          INTEGER Z11(12000),Z12(12000),Z13(12000),SL1(12000),SL2(12000)
          INTEGER XP2(10000),ZX(6000,12),V(6000),SERR,PC,ERRO
         OPEN(9,FILE='NPSM')
          ZX(1,1)=1
          ZX(1,2)=1
          ZX(1,3)=1
          ZX(1,4)=1
          ZX(1,5)=1
          ZX(1,6)=1
          ZX(1,7)=1
          ZX(1,8)=0
          ZX(1,9)=1
          ZX(1,10)=1
          ZX(1,11)=1
          ZX(1,12)=1
          READ(9,42)NBIT,NS,NDLIN,NP
 42       FORMAT(' ',I4,2X,I4,2X,I4,2X,I4)
          WRITE(*,58)NBIT,NS,NDLIN,NP
 58     FORMAT(' ','NBIT=',2X,I3,'NS=',2X,I3,'NDLIN',2X,I2,'NP=',2X,I3)
C         READ,NBIT
          K=2**NBIT
C         READ,NS
          NN=NS+143
          N1=NN+1
C         READ,NDLIN
          NL=NDLIN
C         READ,NP
          SERR=0
          PC=0
          NDDD=0
          READ(9,*)(X1(I),I=144,NN)
          READ(9,*)(X2(I),I=144,NN)
          READ(9,*)(X3(I),I=144,NN)
          READ(9,*)(X4(I),I=144,NN)
          READ(9,*)(X5(I),I=144,NN)
          READ(9,*)(X6(I),I=144,NN)
          READ(9,*)(X7(I),I=144,NN)
          READ(9,*)(X8(I),I=144,NN)
          READ(9,*)(X9(I),I=144,NN)
           DO 01 I=1,143
           X1(I)=0
           X2(I)=0
           X3(I)=0
           X4(I)=0
           X5(I)=0
           X6(I)=0
           X7(I)=0
```

```
            X8(I)=0
            X9(I)=0
            PL1(I)=0
            PL2(I)=0
            PL3(I)=0
 01         CONTINUE
             N=NN+143
            DO 02 J=N1,N
            X1(J)=0
            X2(J)=0
            X3(J)=0
            X4(J)=0
            X5(J)=0
            X6(J)=0
            X7(J)=0
            X8(J)=0
            X9(J)=0
            PL1(J)=0
            PL2(J)=0
            PL3(J)=0
 02         CONTINUE
            DO 499 I=1,23
            Z11(I)=0
            Z12(I)=0
            Z13(I)=0
 499        CONTINUE
            DO 021 J=N1,N
            Z11(J)=0
            Z12(J)=0
            Z13(J)=0
 021        CONTINUE
          NSS=N
          DO 10 I=144,N
       Z11(I-117)=MOD(K-MOD((X1(I)+X2(I-13)+X3(I-26)+X4(I-39)+
     @X5(I-52)+X6(I-65)+X7(I-78)+X8(I-91)+X9(I-104)+
     @Z12(I-130)+Z13(I-143)),K),K)
       Z12(I-119)=MOD(K-MOD((X1(I-99)+X2(I-101)+X3(I-103)+X4(I-105)
     @+X5(I-107)+X6(I-109)+X7(I-111)+X8(I-113)+X9(I-115)
     @+Z11(I-117)+Z13(I-121)),K),K)
       Z13(I-120)=MOD(K-MOD((X1(I-109)+X2(I-110)+X3(I-111)+X4(I-112)
     @+X5(I-113)+X6(I-114)+X7(I-115)+X8(I-116)+X9(I-117)
     @+Z11(I-118)+Z12(I-119)),K),K)
 10        CONTINUE
           CALL PSTO(Z11,Z12,Z13,SL1,SL2,SL3,N)
            CALL STORE(X1,X2,X3,X4,X5,X6,X7,X8,X9,S1,S2,S3,
     @S4,S5,S6,S7,S8,S9,N,NL)
C          DO 316 I=1,N
C           WRITE(9,317)X1(I),S1(I),X2(I),S2(I),X3(I),S3(I),X4(I)
C     @,S4(I),X5(I),S5(I),X6(I),S6(I),X7(I),S7(I),X8(I),S8(I)
C317        FORMAT(' ',16(I3))
C316        CONTINUE
 211       JJ=0
C          WRITE(*,141)JJ
C141       FORMAT('211 REASHED BEF. RAND',2X,I3)
           NDDD=0
            NI=299
            CALL RAND(ZX,V,NS)
            DO 11 I=144,NN
               NII=NI+MOD(I,4)
             IF(MOD(V(I-143),NII).EQ.0)THEN
```

```
C          WRITE(*,131)NII,V(I-341)
C131        FORMAT(' ','RAND',2X,I3,2X,I6)
           X1(I)=ZX(I-143,1)*4+ZX(I-143,3)*2+ZX(I-143,4)
           ENDIF
  11       CONTINUE
        CALL RAND(ZX,V,NS)
         DO 12 I=144,NN
            NII=NI+MOD(I,4)
          IF(MOD(V(I-143),NII).EQ.0)THEN
C         WRITE(*,132)NII,V(I-341)
C132        FORMAT(' ','RAND',2X,I3,2X,I6)
           X2(I)=ZX(I-142,4)*4+ZX(I-142,11)*2+ZX(I-142,7)
           ENDIF
  12       CONTINUE
        CALL RAND(ZX,V,NS)
         DO 13 I=144,NN
            NII=NI+MOD(I,4)
          IF(MOD(V(I-143),NII).EQ.0)THEN
C         WRITE(*,133)NII,V(I-341)
C133        FORMAT(' ','RAND',2X,I3,2X,I6)
           X3(I)=ZX(I-140,9)*4+ZX(I-140,8)*2+ZX(I-140,6)
           ENDIF
  13       CONTINUE .
        CALL RAND(ZX,V,NS)
         DO 14 I=144,NN
            NII=NI+MOD(I,4)
          IF(MOD(V(I-143),NII).EQ.0)THEN
C         WRITE(*,134)NII,V(I-341)
C134        FORMAT(' ','RAND',2X,I3,2X,I6)
           X4(I)=ZX(I-142,9)*4+ZX(I-140,8)*2+ZX(I-140,6)
           ENDIF
  14       CONTINUE
        CALL RAND(ZX,V,NS)
         DO 15 I=144,NN
            NII=NI+MOD(I,4)
          IF(MOD(V(I-143),NII).EQ.0)THEN
C         WRITE(*,135)NII,V(I-341)
C135        FORMAT(' ','RAND',2X,I3,2X,I6)
           X5(I)=ZX(I-142,9)*4+ZX(I-142,8)*2+ZX(I-142,6)
           ENDIF
  15       CONTINUE
        CALL RAND(ZX,V,NS)
         DO 16 I=144,NN
            NII=NI+MOD(I,4)
          IF(MOD(V(I-143),NII).EQ.0)THEN
C         WRITE(*,136)NII,V(I-341)
C136        FORMAT(' ','RAND',2X,I3,2X,I6)
           X6(I)=ZX(I-142,9)*4+ZX(I-142,8)*2+ZX(I-142,6)
           ENDIF
  16       CONTINUE
        CALL RAND(ZX,V,NS)
         DO 17 I=144,NN
            NII=NI+MOD(I,4)
          IF(MOD(V(I-143),NII).EQ.0)THEN
C         WRITE(*,137)NII,V(I-341)
C137        FORMAT(' ','RAND',2X,I3,2X,I6)
           X7(I)=ZX(I-142,9)*4+ZX(I-142,8)*2+ZX(I-142,6)
           ENDIF
  17       CONTINUE
        CALL RAND(ZX,V,NS)
```

```
          DO 18 I=144,NN
             NII=NI+MOD(I,4)
           IF(MOD(V(I-143),NII).EQ.0)THEN
C          WRITE(*,138)NII,V(I-341)
C138        FORMAT(' ','RAND',2X,I3,2X,I6)
            X8(I)=ZX(I-142,9)*4+ZX(I-142,8)*2+ZX(I-142,6)
            ENDIF
  18        CONTINUE
         CALL RAND(ZX,V,NS)
          DO 19 I=144,NN
             NII=NI+MOD(I,4)
           IF(MOD(V(I-143),NII).EQ.0)THEN
C          WRITE(*,139)NII,V(I-341)
C139        FORMAT(' ','RAND',2X,I3,2X,I6)
            X9(I)=ZX(I-142,9)*4+ZX(I-142,8)*2+ZX(I-142,6)
            ENDIF
  19        CONTINUE
         CALL RAND(ZX,V,N)
         DO 292 I=24,NN
             NII=NI+MOD(I,4)
         IF(MOD(V(I-23),NII).EQ.0)THEN
         Z11(I)=ZX(I-22,5)*4+ZX(I-22,8)*2+ZX(I-22,2)
         ENDIF
 292     CONTINUE
         CALL RAND(ZX,V,N)
         DO 293 I=24,NN
             NII=NI+MOD(I,4)
         IF(MOD(V(I-23),NII).EQ.0)THEN
         Z12(I)=ZX(I-23,1)*4+ZX(I-23,12)*2+ZX(I-23,8)
          ENDIF
 293      CONTINUE
         CALL RAND(ZX,V,N)
         DO 294 I=24,NN
             NII=NI+MOD(I,4)
         IF(MOD(V(I-23),NII).EQ.0)THEN
         Z13(I)=ZX(I-23,3)*4+ZX(I-23,6)*2+ZX(I-23,9)
         ENDIF
 294     CONTINUE
 198      CALL CODE(X1,X2,X3,X4,X5,X6,X7,X8,X9,PL1,PL2,PL3,
     @Z11,Z12,Z13,NN,N,K)
         DO 333 I=1,N
         XW1(I)=0
         XW2(I)=0
         XW3(I)=0
         XW4(I)=0
         XW5(I)=0
         XW6(I)=0
         XW7(I)=0
         XW8(I)=0
         XW9(I)=0
         XP1(I)=0
         XP2(I)=0
         XP3(I)=0
 333     CONTINUE
          CALL DET(XW1,XW2,XW3,XW4,XW5,XW6,XW7,XW8,XW9,XP1,XP2,XP3,
     @N,PL1,PL2,PL3,Z11,Z12,Z13,NC)
         IF(NDDD.GE.17)THEN
C        PRINT,'NNDDDDD=================== 7     '
C        WRITE(*,986)NDDD
C986     FORMAT(' ','NDDD=    ',I4)
```

```
        GO TO 197
        ENDIF
        IF(NC.NE.0)THEN
C       WRITE(*,985)NC
C985    FORMAT(' ','NC =  ',I7)
        CALL CORR(XW1,XW2,XW3,XW4,XW5,XW6,XW7,XW8,XW9,XP1,
     @XP2,XP3,X1,X2,X3,X4,X5,X6,X7,X8,X9,PL1,PL2,
     @PL3,Z11,Z12,Z13,N,NL,K,JJ,ND,NP)
         NDDD=NDDD+1
         IF(ND.EQ.0)THEN
         JJ=JJ+1
         IF(NP-JJ.EQ.1)THEN
          ERRO=0
          GOTO 937
          ENDIF
          ENDIF
         GO TO 198
         ELSE
         GO TO 197
         ENDIF
 197     NDDD=0
         ERRO=0
         IF(NC.NE.0)THEN
         ERRO=2
C        WRITE(*,987)ERRO
C987     FORMAT(' ','NASSER ABDEL  ',2X,I5)
         GO TO 937
         ENDIF
         DO 111 I=144,NN
         IF(X1(I).NE.S1(I))THEN
           ERRO=ERRO+1
          ENDIF
 111      CONTINUE
          DO 422 I=144,NN
          IF(X2(I).NE.S2(I))THEN
           ERRO=ERRO+1
          ENDIF
 422      CONTINUE
          DO 33 I=144,NN
          IF(X3(I).NE.S3(I))THEN
           ERRO=ERRO+1
          ENDIF
 33       CONTINUE
          DO 44 I=144,NN
          IF(X4(I).NE.S4(I))THEN
           ERRO=ERRO+1
          ENDIF
 44       CONTINUE
          DO 55 I=144,NN
          IF(X5(I).NE.S5(I))THEN
           ERRO=ERRO+1
          ENDIF
 55       CONTINUE
          DO 66 I=144,NN
          IF(X6(I).NE.S6(I))THEN
           ERRO=ERRO+1
          ENDIF
 66       CONTINUE
          DO 77 I=144,NN
          IF(X7(I).NE.S7(I))THEN
```

```
            ERRO=ERRO+1
           ENDIF
 77        CONTINUE
           DO 88 I=144,NN
           IF(X8(I).NE.S8(I))THEN
            ERRO=ERRO+1
           ENDIF
 88        CONTINUE
           DO 99 I=144,NN
           IF(X9(I).NE.S9(I))THEN
            ERRO=ERRO+1
           ENDIF
 99        CONTINUE
 937      N=NSS
          CALL STORE(S1,S2,S3,S4,S5,S6,S7,S8,S9,X1,X2,X3,
     @X4,X5,X6,X7,X8,X9,N,NL)
         CALL PSTO(SL1,SL2,SL3,Z11,Z12,Z13,N)
C        DO 314 I=1,N
C         WRITE(9,315)X1(I),S1(I),X2(I),S2(I),X3(I),S3(I),X4(I)
C    @,S4(I),X5(I),S5(I),X6(I),S6(I),X7(I),S7(I),X8(I),S8(I)
C315      FORMAT(' ',16(I3))
C314      CONTINUE
C         PRINT,'ERROR===',ERRO
          IF(ERRO.EQ.0)THEN
          PC=PC+1
           IF(MOD(PC,100).EQ.0)THEN
           WRITE(*,323)PC
 323       FORMAT(' ','PCCC    =',I4)
           ENDIF
           IF(PC.GE.8000)THEN
           GO TO 301
          ENDIF
           GO TO 211
          ENDIF
           PC=PC+1
C         PRINT,'PC=  ',PC
          SERR=SERR+1
C          PRINT,'SERR======',SERR
           WRITE(*,322)PC,SERR,ERRO
 322       FORMAT(' ','PCCCCCCCCC=',I4,'SERRRRRRR=',I4,'ERROR===',I4)
           IF(PC.GE.8000)THEN
           GOTO 301
           ENDIF
          IF(SERR.LT.10)THEN
          GO TO 211
          ENDIF
 301       WRITE(*,321)PC,SERR,ERRO
 321       FORMAT(' ','PCCCCCCCCC=',I4,'SERRRRRRR=',I4,'ERROR===',I4)
          CLOSE(9)
          STOP
          END
C         END
        SUBROUTINE CODE(X1,X2,X3,X4,X5,X6,X7,X8,X9,
     @PL1,PL2,PL3,Z11,Z12,Z13,NN,N,K)
       INTEGER X1(10000),X2(10000),X3(10000),X4(12000),X5(12000)
       INTEGER X6(10000),X7(10000),X8(10000),X9(12000),PL1(12000)
       INTEGER PL2(12000),PL3(12000),Z11(12000),Z12(12000),Z13(12000)
         DO 10 I=144,N
       PL1(I-117)=MOD(K-MOD((X1(I)+X2(I-13)+X3(I-26)+X4(I-39)+
     @X5(I-52)+X6(I-65)+X7(I-78)+X8(I-91)+X9(I-104)+
```

```
     @Z12(I-130)+Z13(I-143)),K),K)
      PL2(I-119)=MOD(K-MOD((X1(I-99)+X2(I-101)+X3(I-103)+X4(I-105
     @+X5(I-107)+X6(I-109)+X7(I-111)+X8(I-113)+X9(I-115)
     @+Z11(I-117)+Z13(I-121)),K),K)
      PL3(I-120)=MOD(K-MOD((X1(I-109)+X2(I-110)+X3(I-111)+X4(I-112)
     @+X5(I-113)+X6(I-114)+X7(I-115)+X8(I-116)+X9(I-117)
     @+Z11(I-118)+Z12(I-119)),K),K)
 10     CONTINUE
       RETURN
       END
         SUBROUTINE DET(XW1,XW2,XW3,XW4,XW5,XW6,XW7,XW8,XW9,
     @XP1,XP2,XP3,N,PL1,PL2,PL3,Z11,Z12,Z13,NC)
         INTEGER XW1(10000),XW2(10000),XW3(10000),XW4(10000),XW5(10000)
         INTEGER XW6(10000),XW7(10000),XW8(10000),XW9(10000),XP1(10000)
         INTEGER XP3(10000),PL1(12000),PL2(12000),PL3(12000),Z11(12000)
         INTEGER Z12(12000),Z13(12000),XP2(10000)
         NM=N-24
C        WRITE(9,20)N
C20      FORMAT(' ','NNNN=======',2X,I3)
           NC=0
         DO 08 J=24,N
         IF(PL1(J).NE.Z11(J))THEN
           NC=NC+1
           XW1(J+117)=XW1(J+117)+1
           XW2(J+104)=XW1(J+104)+1
           XW3(J+91)=XW3(J+91)+1
           XW4(J+78)=XW4(J+78)+1
           XW5(J+65)=XW5(J+65)+1
           XW6(J+52)=XW4(J+52)+1
           XW7(J+39)=XW7(J+39)+1
           XW8(J+26)=XW8(J+26)+1
           XW9(J+13)=XW9(J+13)+1
           XP1(J)=XP1(J)+1
           XP2(J-13)=XP2(J-13)+1
           XP3(J-26)=XP3(J-26)+1
         ENDIF
 08      CONTINUE
         DO 07 I=24,NM
         IF(PL2(I).NE.Z12(I))THEN
           NC=NC+1
            XW1(I+20)=XW1(I+20)+1
            XW2(I+18)=XW2(I+18)+1
            XW3(I+16)=XW3(I+16)+1
            XW4(I+14)=XW4(I+14)+1
            XW5(I+12)=XW5(I+12)+1
            XW6(I+10)=XW6(I+10)+1
            XW7(I+8)=XW7(I+8)+1
            XW8(I+6)=XW8(I+6)+1
            XW9(I+4)=XW9(I+4)+1
            XP1(I+2)=XP1(I+2)+1
            XP2(I)=XP2(I)+1
            XP3(I-2)=XP3(I-2)+1
         ENDIF
 07      CONTINUE
         DO 09 I=1,NM
         IF (PL3(I).NE.Z13(I))THEN
           NC=NC+1
            XW1(I+11)=XW1(I+11)+1
            XW2(I+10)=XW2(I+10)+1
            XW3(I+9)=XW3(I+9)+1
```

```
            XW4(I+8)=XW4(I+8)+1
            XW5(I+7)=XW5(I+7)+1
            XW6(I+6)=XW6(I+6)+1
            XW7(I+5)=XW7(I+5)+1
            XW8(I+4)=XW8(I+4)+1
            XW9(I+3)=XW9(I+3)+1
            XP1(I+2)=XP1(I+2)+1
            XP2(I+1)=XP2(I+1)+1
            XP3(I)=XP3(I)+1
          ENDIF
 09     CONTINUE
        RETURN
        END
        SUBROUTINE CORR(XW1,XW2,XW3,XW4,XW5,XW6,XW7,XW8,XW9,XP1,XP2,
     @XP3,X1,X2,X3,X4,X5,X6,X7,X8,X9,PL1,PL2,PL3,
     @Z11,Z12,Z13,N,NL,K,JJ,ND,NP)
        INTEGER X1(10000),X2(10000),X3(10000),X4(10000),X5(10000)
        INTEGER X6(10000),X7(10000),X8(10000),X9(10000),XW1(10000)
        INTEGER XW2(10000),XW3(10000),XW4(10000),XW5(10000),XW6(10000)
        INTEGER XW7(10000),XW8(10000),XW9(10000),Z11(12000),Z12(12000)
        INTEGER XP1(10000),XP2(10000),XP3(10000),Z13(12000)
        INTEGER PL1(12000),PL2(12000),PL3(12000),E1,E2,E3
         ND=0
C        WRITE(9,99)N,K,NP,JJ
C99      FORMAT(' ','N=',2X,I3,'K=',2X,I3,'NP=',2X,I3,'JJ=',2X,I3)
         NM=N-24
         DO 23 I=1,N
          IF(XW1(I).EQ.NP-JJ)THEN
           ND=ND+1
           E1=MOD(K+MOD((PL1(I-117)-Z11(I-117)),K),K)
           E2=MOD(K+MOD((PL2(I-20)-Z12(I-20)),K),K)
           E3=MOD(K+MOD((PL3(I-11)-Z13(I-11)),K),K)
C          WRITE(9,27)X1(I),I,E1,E2,E3
C27        FORMAT(' ','OLD X1 ===',5(2X,I3))
           IF (E1.EQ.E2)THEN
            X1(I)=MOD(X1(I)+E1,K)
           ELSE
            IF(E1.EQ.E3)THEN
             X1(I)=MOD(X1(I)+E1,K)
            ELSE
             IF(E2.EQ.E3)THEN
              X1(I)=MOD(X1(I)+E2,K)
             ELSE
              X1(I)=MOD(X1(I)+E3,K)
             ENDIF
            ENDIF
           ENDIF
C          WRITE(9,26)X1(I),I
C26        FORMAT(' ','NEW X1 ==',2X,I3,2X,I3)
          ENDIF
          IF (XW2(I).EQ.NP-JJ)THEN
          ND=ND+1
          E1=MOD(K+MOD(PL1(I-104)-Z11(I-104),K),K)
          E2=MOD(K+MOD(PL2(I-18)-Z12(I-18),K),K)
          E3=MOD(K+MOD(PL3(I-10)-Z13(I-10),K),K)
C          WRITE(9,97)X2(I),I,E1,E2,E3
C97        FORMAT(' ','OLD X2 ===',5(2X,I3))
          IF(E1.EQ.E2)THEN
          X2(I)=MOD(X2(I)+E1,K)
          ELSE
```

```
            IF(E1.EQ.E3)THEN
            X2(I)=MOD(X2(I)+E1,K)
            ELSE
            IF(E2.EQ.E3)THEN
            X2(I)=MOD(X2(I)+E2,K)
            ELSE
            X2(I)=MOD(X2(I)+E3,K)
            ENDIF
            ENDIF
            ENDIF
C           WRITE(9,96)X2(I),I
C96         FORMAT(' ','NEW X2 ==',2X,I3,2X,I3)
            ENDIF
            IF (XW3(I).EQ.NP-JJ)THEN
            ND=ND+1
            E1=MOD(K+MOD(PL1(I-91)-Z11(I-91),K),K)
            E2=MOD(K+MOD(PL2(I-16)-Z12(I-16),K),K)
            E3=MOD(K+MOD(PL3(I-9)-Z13(I-9),K),K)
C            WRITE(9,37)X3(I),I,E1,E2,E3
C37          FORMAT(' ','OLD X3 ===',5(2X,I3))
            IF(E1.EQ.E2)THEN
            X3(I)=MOD(X3(I)+E1,K)
            ELSE
            IF(E1.EQ.E3)THEN
            X3(I)=MOD(X3(I)+E1,K)
            ELSE
            IF(E2.EQ.E3)THEN
            X3(I)=MOD(X3(I)+E2,K)
            ELSE
            X3(I)=MOD(X3(I)+E3,K)
            ENDIF
            ENDIF
            ENDIF
C           WRITE(9,36)X3(I),I
C36         FORMAT(' ','NEW X3 ==',2X,I3,2X,I3)
            ENDIF
            IF (XW4(I).EQ.NP-JJ)THEN
            ND=ND+1
            E1=MOD(K+MOD(PL1(I-78)-Z11(I-78),K),K)
            E2=MOD(K+MOD(PL2(I-14)-Z12(I-14),K),K)
            E3=MOD(K+MOD(PL3(I-8)-Z13(I-8),K),K)
C            WRITE(9,37)X4(I),I,E1,E2,E3
C37          FORMAT(' ','OLD X4 ===',5(2X,I3))
            IF(E1.EQ.E2)THEN
            X4(I)=MOD(X4(I)+E1,K)
            ELSE
            IF(E1.EQ.E3)THEN
            X4(I)=MOD(X4(I)+E1,K)
            ELSE
            IF(E2.EQ.E3)THEN
            X4(I)=MOD(X4(I)+E2,K)
            ELSE
            X4(I)=MOD(X4(I)+E3,K)
            ENDIF
            ENDIF
            ENDIF
C           WRITE(9,36)X4(I),I
C36         FORMAT(' ','NEW X4 ==',2X,I3,2X,I3)
            ENDIF
            IF (XW5(I).EQ.NP-JJ)THEN
```

```
                ND=ND+1
                E1=MOD(K+MOD(PL1(I-65)-Z11(I-65),K),K)
                E2=MOD(K+MOD(PL2(I-12)-Z12(I-12),K),K)
                E3=MOD(K+MOD(PL3(I-7)-Z13(I-7),K),K)
C                WRITE(9,37)X5(I),I,E1,E2,E3
C37              FORMAT(' ','OLD X5 ===',5(2X,I3))
                IF(E1.EQ.E2)THEN
                X5(I)=MOD(X5(I)+E1,K)
                ELSE
                IF(E1.EQ.E3)THEN
                X5(I)=MOD(X5(I)+E1,K)
                ELSE
                IF(E2.EQ.E3)THEN
                X5(I)=MOD(X5(I)+E2,K)
                ELSE
                X5(I)=MOD(X5(I)+E3,K)
                ENDIF
                ENDIF
                ENDIF
C               WRITE(9,36)X5(I),I
C36             FORMAT(' ','NEW X5 ==',2X,I3,2X,I3)
                ENDIF
                IF (XW6(I).EQ.NP-JJ)THEN
                ND=ND+1
                E1=MOD(K+MOD(PL1(I-52)-Z11(I-52),K),K)
                E2=MOD(K+MOD(PL2(I-10)-Z12(I-10),K),K)
                E3=MOD(K+MOD(PL3(I-6)-Z13(I-6),K),K)
C                WRITE(9,37)X6(I),I,E1,E2,E3
C37              FORMAT(' ','OLD X6 ===',5(2X,I3))
                IF(E1.EQ.E2)THEN
                X6(I)=MOD(X6(I)+E1,K)
                ELSE
                IF(E1.EQ.E3)THEN
                X6(I)=MOD(X6(I)+E1,K)
                ELSE
                IF(E2.EQ.E3)THEN
                X6(I)=MOD(X6(I)+E2,K)
                ELSE
                X6(I)=MOD(X6(I)+E3,K)
                ENDIF
                ENDIF
                ENDIF
C               WRITE(9,36)X6(I),I
C36             FORMAT(' ','NEW X6 ==',2X,I3,2X,I3)
                ENDIF
                IF (XW7(I).EQ.NP-JJ)THEN
                ND=ND+1
                E1=MOD(K+MOD(PL1(I-39)-Z11(I-39),K),K)
                E2=MOD(K+MOD(PL2(I-8)-Z12(I-8),K),K)
                E3=MOD(K+MOD(PL3(I-5)-Z13(I-5),K),K)
C                WRITE(9,37)X7(I),I,E1,E2,E3
C37              FORMAT(' ','OLD X7 ===',5(2X,I3))
                IF(E1.EQ.E2)THEN
                X7(I)=MOD(X7(I)+E1,K)
                ELSE
                IF(E1.EQ.E3)THEN
                X7(I)=MOD(X7(I)+E1,K)
                ELSE
                IF(E2.EQ.E3)THEN
                X7(I)=MOD(X7(I)+E2,K)
```

```
                ELSE
                X7(I)=MOD(X7(I)+E3,K)
                ENDIF
                ENDIF
                ENDIF
C               WRITE(9,36)X7(I),I
C36             FORMAT(' ','NEW X7 ==',2X,I3,2X,I3)
                ENDIF
                IF (XW8(I).EQ.NP-JJ)THEN
                ND=ND+1
                E1=MOD(K+MOD(PL1(I-26)-Z11(I-26),K),K)
                E2=MOD(K+MOD(PL2(I-6)-Z12(I-6),K),K)
                E3=MOD(K+MOD(PL3(I-4)-Z13(I-4),K),K)
C                WRITE(9,37)X8(I),I,E1,E2,E3
C37              FORMAT(' ','OLD X8 ===',5(2X,I3))
                IF(E1.EQ.E2)THEN
                X8(I)=MOD(X8(I)+E1,K)
                ELSE
                IF(E1.EQ.E3)THEN
                X8(I)=MOD(X8(I)+E1,K)
                ELSE
                IF(E2.EQ.E3)THEN
                X8(I)=MOD(X8(I)+E2,K)
                ELSE
                X8(I)=MOD(X8(I)+E3,K)
                ENDIF
                ENDIF
                ENDIF
C               WRITE(9,36)X8(I),I
C36             FORMAT(' ','NEW X8 ==',2X,I3,2X,I3)
                ENDIF
                IF (XW9(I).EQ.NP-JJ)THEN
                ND=ND+1
                E1=MOD(K+MOD(PL1(I-13)-Z11(I-13),K),K)
                E2=MOD(K+MOD(PL2(I-4)-Z12(I-4),K),K)
                E3=MOD(K+MOD(PL3(I-3)-Z13(I-3),K),K)
C                WRITE(9,37)X9(I),I,E1,E2,E3
C37              FORMAT(' ','OLD X9 ===',5(2X,I3))
                IF(E1.EQ.E2)THEN
                X9(I)=MOD(X9(I)+E1,K)
                ELSE
                IF(E1.EQ.E3)THEN
                X9(I)=MOD(X9(I)+E1,K)
                ELSE
                IF(E2.EQ.E3)THEN
                X9(I)=MOD(X9(I)+E2,K)
                ELSE
                X9(I)=MOD(X9(I)+E3,K)
                ENDIF
                ENDIF
                ENDIF
C               WRITE(9,36)X9(I),I
C36             FORMAT(' ','NEW X9 ==',2X,I3,2X,I3)
                ENDIF
                IF (XP1(I).EQ.NP-JJ)THEN
                ND=ND+1
                E1=MOD(K+MOD(PL1(I)-Z11(I),K),K)
                E2=MOD(K+MOD(PL2(I-2)-Z12(I-2),K),K)
                E3=MOD(K+MOD(PL3(I-2)-Z13(I-2),K),K)
C              WRITE(9,25)E1,E2,E3
```

```
C25          FORMAT(' ',9(I3,8X))
C           WRITE(9,77)Z11(I),I
C77         FORMAT(' ','OLD Z11=====',2X,I3,3X,I3)
            IF(E1.EQ.E2)THEN
            Z11(I)=MOD(Z11(I)+E1,K)
            ELSE
            IF(E1.EQ.E3)THEN
            Z11(I)=MOD(Z11(I)+E1,K)
            ELSE
            IF(E2.EQ.E3)THEN
            Z11(I)=MOD(Z11(I)+E2,K)
            ELSE
            Z11(I)=MOD(Z11(I)+E3,K)
            ENDIF
            ENDIF
            ENDIF
C           WRITE(9,71)Z11(I),I
C71         FORMAT(' ','NEW Z11 ==',2X,I3,2X,I3)
            ENDIF
            IF (XP2(I).EQ.NP-JJ)THEN
            ND=ND+1
            E1=MOD(K+MOD(PL1(I+13)-Z11(I+13),K),K)
            E2=MOD(K+MOD(PL2(I)-Z12(I),K),K)
            E3=MOD(K+MOD(PL3(I-1)-Z13(I-1),K),K)
C          WRITE(9,35)E1,E2,E3
C35          FORMAT(' ',9(I3,8X))
C           WRITE(9,66)Z12(I),I
C66         FORMAT(' ','OLD Z12=====',2X,I3,3X,I3)
            IF(E1.EQ.E2)THEN
            Z12(I)=MOD(Z12(I)+E1,K)
            ELSE
            IF(E1.EQ.E3)THEN
            Z12(I)=MOD(Z12(I)+E1,K)
            ELSE
            IF(E2.EQ.E3)THEN
            Z12(I)=MOD(Z12(I)+E2,K)
            ELSE
            Z12(I)=MOD(Z12(I)+E3,K)
            ENDIF
            ENDIF
            ENDIF
C           WRITE(9,61)Z12(I),I
C61         FORMAT(' ','NEW Z12 ==',2X,I3,2X,I3)
            ENDIF
            IF (XP3(I).EQ.NP-JJ)THEN
            ND=ND+1
            E1=MOD(K+MOD(PL1(I+26)-Z11(I+26),K),K)
            E2=MOD(K+MOD(PL2(I+2)-Z12(I+2),K),K)
            E3=MOD(K+MOD(PL3(I)-Z13(I),K),K)
C          WRITE(9,45)E1,E2,E3
C45          FORMAT(' ',9(I3,8X))
C           WRITE(9,55)Z13(I),I
C55         FORMAT(' ','OLD Z13=====',2X,I3,3X,I3)
            IF(E1.EQ.E2)THEN
            Z13(I)=MOD(Z13(I)+E1,K)
            ELSE
            IF(E1.EQ.E3)THEN
            Z13(I)=MOD(Z13(I)+E1,K)
            ELSE
            IF(E2.EQ.E3)THEN
```

```
            Z13(I)=MOD(Z13(I)+E2,K)
            ELSE
            Z13(I)=MOD(Z13(I)+E3,K)
            ENDIF
            ENDIF
            ENDIF
C           WRITE(9,51)Z13(I),I
C51         FORMAT(' ','NEW Z13 ==',2X,I3,2X,I3)
            ENDIF
 23        CONTINUE
           RETURN
           END
           SUBROUTINE STORE(X1,X2,X3,X4,X5,X6,X7,X8,X9,S1,S2,S3,
     @S4,S5,S6,S7,S8,S9,N,NL)
           INTEGER X1(10000),X2(10000),X3(10000),X4(10000),X5(10000)
           INTEGER X6(10000),X7(10000),X8(10000),X9(10000),S1(10000)
           INTEGER S2(10000),S3(10000),S4(10000),S5(10000)
           INTEGER S6(10000),S7(10000),S8(10000),S9(10000)
           NI=N-50
            DO 907 KL=1,N
              S1(KL)=X1(KL)
              S2(KL)=X2(KL)
              S3(KL)=X3(KL)
              S4(KL)=X4(KL)
              S5(KL)=X5(KL)
              S6(KL)=X6(KL)
              S7(KL)=X7(KL)
              S8(KL)=X8(KL)
              S9(KL)=X9(KL)
 907         CONTINUE
             RETURN
             END
            SUBROUTINE RAND(ZX,V,NS)
             INTEGER ZX(6000,12),V(6000),NNN
             NSC=NS+10
             V(1)=4096
             DO 5 I=2,NSC
           ZX(I,1)=MOD((ZX(I-1,2)+ZX(I-1,10)+ZX(I-1,11)+ZX(I-1,12)),2)
             DO 6 J=2,12
              ZX(I,J)=ZX(I-1,J-1)
 6           CONTINUE
             V(I)=ZX(I,1)*2048+ZX(I,2)*1024+ZX(I,3)*512+ZX(I,8)*16
     @+ZX(I,4)*256+ZX(I,5)*128+ZX(I,6)*64+ZX(I,7)*32+ZX(I,9)*8
     @+ZX(I,10)*4+ZX(I,11)*2+ZX(I,12)
C            PRINT,V(I)
 5           CONTINUE
             DO 4 KN=1,12
             ZX(1,KN)=ZX(NS,KN)
 4           CONTINUE
             RETURN
             END
             SUBROUTINE PSTO(PL1,PL2,PL3,SL1,SL2,SL3,N)
            INTEGER PL1(12000),PL2(12000),PL3(12000)
            INTEGER SL1(12000),SL2(12000),SL3(12000)
            DO 312 I=1,N
            SL1(I)=PL1(I)
            SL2(I)=PL2(I)
            SL3(I)=PL3(I)
C           PRINT,PL3(I),'YYYYYYYYYYYYYYYYYYYYYYY',I
 312        CONTINUE
```

8 APPENDIX B

In this appendix we present the simulation program for the rate 1/2 convolutional TASM code with 3-data line and 3-parity lines. This program simulates the encoder, the channel and the decoder and calculates the output error rate. It employs the new decoding algorithm described in section 3.1.1 .

```
**************$JOB
C*
C*
C*
C*
C*************************************************************
      INTEGER X1(2000),X2(2000),X3(2000),S1(2000),S2(2000),S3(2000)
      INTEGER XW1(2000),XW2(2000),XW3(2000),XP1(2000),XP2(2000)
      INTEGER PL1(3000),PL2(3000),PL3(3000),XP3(2000),SL3(3000)
      INTEGER Z11(3000),Z12(3000),Z13(3000),SL1(3000),SL2(3000)
      INTEGER ZX(6000,12),V(6000),SERR,PC,ERRO
     OPEN(9,FILE='NPSM')
      ZX(1,1)=1
      ZX(1,2)=1
      ZX(1,3)=1
      ZX(1,4)=1
      ZX(1,5)=1
      ZX(1,6)=1
      ZX(1,7)=1
      ZX(1,8)=1
      ZX(1,9)=1
      ZX(1,10)=1
      ZX(1,11)=1
      ZX(1,12)=1
      READ(9,42)NBIT,NS,NDLIN,NP
 42   FORMAT(' ',I4,2X,I4,2X,I4,2X,I4)
C     WRITE(9,88)NBIT,NS,NDLIN,NP
C88  FORMAT(' ','NBIT=',2X,I3,'NS=',2X,I3,'NDLIN',2X,I2,'NP=',2X,I3)
C     READ,NBIT
      K=2**NBIT
C     READ,NS
      NN=NS+40
      N1=NN+1
C     READ,NDLIN
      NL=NDLIN
C     READ,NP
      SERR=0
      PC=0
      NDDD=0
       DO 01 I=1,40
       X1(I)=0
       X2(I)=0
       X3(I)=0
       PL1(I)=0
       PL2(I)=0
       PL3(I)=0
 01    CONTINUE
        N=NN+40
       DO 02 J=N1,N
       X1(J)=0
       X2(J)=0
       X3(J)=0
       PL1(J)=0
       PL2(J)=0
       PL3(J)=0
 02   CONTINUE
      DO 499 I=1,20
      Z11(I)=0
      Z12(I)=0
      Z13(I)=0
```

```
  499      CONTINUE
           DO 021 J=N1,N
           Z11(J)=0
           Z12(J)=0
           Z13(J)=0
  021      CONTINUE
          NSS=N
           NNM=IJ+10
          DO 990 IJ=NS,N
           READ(9,*)(X1(I),I=IJ,NNM)
           READ(9,*)(X2(I),I=IJ,NNM)
           READ(9,*)(X3(I),I=IJ,NNM)
          DO 10 I=41,NS
          Z11(I-24)=MOD(K-MOD((X1(I)+X2(I-8)+X3(I-16)+Z12(I-32)+
     @Z13(I-40)),K),K)
          Z12(I-26)=MOD(K-MOD((X1(I-18)+X2(I-20)+X3(I-22)+Z11(I-24)+
     @Z13(I-28)),K),K)
          IF (I.NE.41)THEN
          Z13(I-27)=MOD(K-MOD((X1(I-22)+X2(I-23)+X3(I-24)+Z11(I-25)+
     @Z12(I-26)),K),K)
          ENDIF
  10      CONTINUE
          CALL PSTO(Z11,Z12,Z13,SL1,SL2,SL3,N)
           CALL STORE(X1,X2,X3,S1,S2,S3,N,NL)
  211     JJ=0
          NDDD=0
          NI=51
           CALL RAND(ZX,V,NS)
C       PRINT,ZX(1,1),ZX(1,2),ZX(1,3),ZX(1,4),ZX(1,5),ZX(1,6),ZX(1,7)
C     @,ZX(1,8)
            I=IJ
              NII=NI+MOD(I,4)
            IF(MOD(V(I-39),NII).EQ.0)THEN
C            WRITE (9,112)X1(I),V(I-40),I
C112         FORMAT(' ',2X,I7,2X,I7,2X,I7)
             X1(I)=ZX(I-40,1)*4+ZX(I-40,3)*2+ZX(I-40,4)
C            WRITE (9,113)X1(I),V(I-40),I
C113         FORMAT(' ',2X,I7,2X,I7,2X,I7,'X1')
             ENDIF
         CALL RAND(ZX,V,NS)
C       PRINT,ZX(1,1),ZX(1,2),ZX(1,3),ZX(1,4),ZX(1,5),ZX(1,6),ZX(1,7)
              NII=NI+MOD(I,4)
            IF(MOD(V(I-40),NII).EQ.0)THEN
             X2(I)=ZX(I-39,4)*4+ZX(I-39,11)*2+ZX(I-39,7)
C            WRITE (9,114)X2(I),V(I-40),I
C114         FORMAT(' ',2X,I7,2X,I7,2X,I7,'X2 ')
C            PRINT,X2(I),V(I-39),I
             ENDIF
         CALL RAND(ZX,V,NS)
C       WRITE(9,121)ZX(1,1),ZX(1,2),ZX(1,3),ZX(1,4),ZX(1,5)
C     @,ZX(1,6),ZX(1,7),ZX(1,8),ZX(1,9),ZX(1,10),ZX(1,11),ZX(1,12)
C121     FORMAT(' ',12(2X,I3))
              NII=NI+MOD(I,4)
            IF(MOD(V(I-41),NII).EQ.0)THEN
             X3(I)=ZX(I-40,9)*4+ZX(I-40,8)*2+ZX(I-40,6)
C            WRITE (9,115)X3(I),V(I-40),I
C115         FORMAT(' ',2X,I7,2X,I7,2X,I7,'KB ')
             ENDIF
         CALL RAND(ZX,V,N)
C       PRINT,ZX(1,1),ZX(1,2),ZX(1,3),ZX(1,4),ZX(1,5)
```

```
            NII=NI+MOD(I,4)
          IF(MOD(V(I-16),NII).EQ.0)THEN
          Z11(I)=ZX(I-15,5)*4+ZX(I-15,8)*2+ZX(I-15,2)
C            WRITE (9,117)Z11(I),V(I-16),I
C117         FORMAT(' ',2X,I7,2X,I7,2X,I7,'Z11')
C         PRINT,PL1(I),V(I-40),I,'PL1'
          ENDIF
          CALL RAND(ZX,V,N)
              NII=NI+MOD(I,4)
          IF(MOD(V(I-14),NII).EQ.0)THEN
          Z12(I)=ZX(I-13,1)*4+ZX(I-13,12)*2+ZX(I-13,8)
C            WRITE (9,118)Z12(I),V(I-14),I
C118         FORMAT(' ',2X,I7,2X,I7,2X,I7,'Z12')
C         PRINT,PL2(I),V(I-40),I,'PL2'
           ENDIF
          CALL RAND(ZX,V,N)
              NII=NI+MOD(I,4)
          IF(MOD(V(I),NII).EQ.0)THEN
          Z13(I)=ZX(I,3)*4+ZX(I,6)*2+ZX(I,9)
C            WRITE (9,119)Z13(I),V(I),I
C119         FORMAT(' ',2X,I7,2X,I7,2X,I7,'Z13')
          ENDIF
 198       CALL CODE(X1,X2,X3,PL1,PL2,PL3,Z11,Z12,Z13,NN,N,K)
C     @PL3(I),Z13(I)
          DO 333 I=1,N
          XW1(I)=0
          XW2(I)=0
          XW3(I)=0
          XP1(I)=0
          XP2(I)=0
          XP3(I)=0
 333      CONTINUE
           CALL DET(XW1,XW2,XW3,XP1,XP2,XP3,N,PL1,PL2,PL3,Z11,Z12,Z13,NC
C         DO 14 I=1,N
C          WRITE(9,15)XW1(I),XW2(I),XW3(I),XP1(I),XP2(I),XP3(I)
C15        FORMAT(' ',6(I3,2X))
C14        CONTINUE
          IF(NDDD.EQ.7)THEN
C         PRINT,'NNDDDDD---------------------- 7     '
          GO TO 197
          ENDIF
          IF(NC.NE.0)THEN
          CALL CORR(XW1,XW2,XW3,XP1,XP2,XP3,X1,X2,X3,PL1,PL2,
     @PL3,Z11,Z12,Z13,N,NL,K,JJ,ND,NP)
           NDDD=NDDD+1
           IF(ND.EQ.0)THEN
           JJ=JJ+1
           IF(NP-JJ.EQ.1)THEN
            ERRO=0
            GOTO 937
            ENDIF
            ENDIF
          GO TO 198
          ELSE
          GO TO 197
          ENDIF
 197      NDDD=0
          ERRO=0
          IF(NC.NE.0)THEN
          ERRO=2
```

```
            GO TO 937
            ENDIF
            I=IJ
            IF(X1(I).NE.S1(I))THEN
              ERRO=ERRO+1
             ENDIF
             IF(X2(I).NE.S2(I))THEN
              ERRO=ERRO+1
             ENDIF
             IF(X3(I).NE.S3(I))THEN
              ERRO=ERRO+1
             ENDIF
C            IF(X4(I).NE.S4(I))THEN
C             ERRO=ERRO+1
C            ENDIF
 937        N=NSS
            CALL STORE(S1,S2,S3,X1,X2,X3,N,NL)
           CALL PSTO(SL1,SL2,SL3,Z11,Z12,Z13,N)
C           PRINT,'ERROR===',ERRO
            IF(ERRO.EQ.0)THEN
            PC=PC+1
             IF(PC.GE.5000)THEN
             GO TO 301
            ENDIF
             GO TO 211
            ENDIF
             PC=PC+1
C           PRINT,'PC=   ',PC
            SERR=SERR+1
C            PRINT,'SERR======',SERR
             WRITE(9,322)PC,SERR,ERRO
 322         FORMAT(' ','PCCCCCCCCC=',I4,'SERRRRRR=',I4,'ERROR===',I4)
             IF(PC.GE.5000)THEN
             GOTO 301
             ENDIF
            IF(SERR.LT.5)THEN
            GO TO 211
            ENDIF
 301         WRITE(9,321)PC,SERR,ERRO
 321         FORMAT(' ','PCCCCCCCCC=',I4,'SERRRRRR=',I4,'ERROR===',I4)
 990        CONTINUE
            CLOSE(9)
            STOP
            END
C           END
        SUBROUTINE CODE(X1,X2,X3,PL1,PL2,PL3,Z11,Z12,Z13,NN,N,K,IJ)
       INTEGER X1(2000),X2(2000),X3(2000),PL1(3000),PL2(3000),PL3(3000)
       INTEGER Z11(3000),Z12(3000),Z13(3000)
         PL1(I-24)=MOD(K-MOD((X1(I)+X2(I-8)+X3(I-16)+Z12(I-32)+
     @Z13(I-40)),K),K)
         PL2(I-26)=MOD(K-MOD((X1(I-18)+X2(I-20)+X3(I-22)+Z11(I-24)+
     @Z13(I-28)),K),K)
         IF (I.NE.41)THEN
         PL3(I-27)=MOD(K-MOD((X1(I-22)+X2(I-23)+X3(I-24)+Z11(I-25)+
     @Z12(I-26)),K),K)
         ENDIF
       RETURN
       END
         SUBROUTINE DET(XW1,XW2,XW3,XP1,XP2,XP3,K,PL1,PL2,PL3,Z11,
     @Z12,Z13,NC,IJ)
```

```
      INTEGER XW1(2000),XW2(2000),XW3(2000),XP1(2000),XP2(2000)
      INTEGER XP3(2000),PL1(3000),PL2(3000),PL3(3000),Z11(3000)
      INTEGER Z12(3000),Z13(3000)
      NM=N-24
C     WRITE(9,20)N
C20   FORMAT(' ','NNNN=======',2X,I3)
        NC=0
      IF(PL1(J).NE.Z11(J))THEN
        NC=NC+1
        XW1(J+24)=XW1(J+24)+1
        XW2(J+16)=XW1(J+16)+1
        XW3(J+8)=XW3(J+8)+1
        XP1(J)=XP1(J)+1
        XP2(J-8)=XP2(J-8)+1
        XP3(J-16)=XP3(J-16)+1
      ENDIF
      IF(PL2(I).NE.Z12(I))THEN
        NC=NC+1
         XW1(I+8)=XW1(I+8)+1
         XW2(I+6)=XW2(I+6)+1
         XW3(I+4)=XW3(I+4)+1
         XP1(I+2)=XP1(I+2)+1
         XP2(I)=XP2(I)+1
         XP3(I-2)=XP3(I-2)+1
      ENDIF
      IF (PL3(I).NE.Z13(I))THEN
        NC=NC+1
         XW1(I+5)=XW1(I+5)+1
         XW2(I+4)=XW2(I+4)+1
         XW3(I+3)=XW3(I+3)+1
         XP1(I+2)=XP1(I+2)+1
         XP2(I+1)=XP2(I+1)+1
         XP3(I)=XP3(I)+1
     ENDIF
09   CONTINUE
     RETURN
     END
     SUBROUTINE CORR(XW1,XW2,XW3,XP1,XP2,XP3,X1,X2,X3,PL1,PL2,PL3,
    @Z11,Z12,Z13,N,NL,K,JJ,ND,NP,IJ)
      INTEGER X1(2000),X2(2000),X3(2000),XW1(2000),XW2(2000),XW3(200(
      INTEGER XP1(2000),XP2(2000),XP3(2000),Z11(3000),Z12(3000),Z13(.
      INTEGER PL1(3000),PL2(3000),PL3(3000),E1,E2,E3
       ND=0
        I=IJ
C      WRITE(9,99)N,K,NP,JJ
C99    FORMAT(' ','N=',2X,I3,'K=',2X,I3,'NP=',2X,I3,'JJ=',2X,I3)
       NM=N-24
        IF(XW1(I).EQ.NP-JJ)THEN
         ND=ND+1
         E1=MOD(K+MOD((PL1(I-24)-Z11(I-24)),K),K)
         E2=MOD(K+MOD((PL2(I-8)-Z12(I-8)),K),K)
         E3=MOD(K+MOD((PL3(I-5)-Z13(I-5)),K),K)
C        WRITE(9,27)X1(I),I,E1,E2,E3
C27      FORMAT(' ','OLD X1 ===',5(2X,I5))
         IF (E1.EQ.E2)THEN
          X1(I)=MOD(X1(I)+E1,K)
         ELSE
          IF(E1.EQ.E3)THEN
           X1(I)=MOD(X1(I)+E1,K)
          ELSE
```

```
                IF(E2.EQ.E3)THEN
                 X1(I)=MOD(X1(I)+E2,K)
                ELSE
                 X1(I)=MOD(X1(I)+E3,K)
                ENDIF
               ENDIF
              ENDIF
C             WRITE(9,26)X1(I),I
C26           FORMAT(' ','NEW X1 ==',2X,I3,2X,I3)
              ENDIF
              IF (XW2(I).EQ.NP-JJ)THEN
              ND=ND+1
              E1=MOD(K+MOD(PL1(I-16)-Z11(I-16),K),K)
              E2=MOD(K+MOD(PL2(I-6)-Z12(I-6),K),K)
              E3=MOD(K+MOD(PL3(I-4)-Z13(I-4),K),K)
C              WRITE(9,97)X2(I),I,E1,E2,E3
C97            FORMAT(' ','OLD X2 ===',5(2X,I3))
              IF(E1.EQ.E2)THEN
              X2(I)=MOD(X2(I)+E1,K)
              ELSE
              IF(E1.EQ.E3)THEN
              X2(I)=MOD(X2(I)+E1,K)
              ELSE
              IF(E2.EQ.E3)THEN
              X2(I)=MOD(X2(I)+E2,K)
              ELSE
              X2(I)=MOD(X2(I)+E3,K)
              ENDIF
              ENDIF
              ENDIF
C             WRITE(9,96)X2(I),I
C96           FORMAT(' ','NEW X2 ==',2X,I3,2X,I3)
              ENDIF
              IF (XW3(I).EQ.NP-JJ)THEN
              ND=ND+1
              E1=MOD(K+MOD(PL1(I-8)-Z11(I-8),K),K)
              E2=MOD(K+MOD(PL2(I-4)-Z12(I-4),K),K)
              E3=MOD(K+MOD(PL3(I-3)-Z13(I-3),K),K)
C              WRITE(9,37)X3(I),I,E1,E2,E3
C37            FORMAT(' ','OLD X3 ===',5(2X,I3))
              IF(E1.EQ.E2)THEN
              X3(I)=MOD(X3(I)+E1,K)
              ELSE
              IF(E1.EQ.E3)THEN
              X3(I)=MOD(X3(I)+E1,K)
              ELSE
              IF(E2.EQ.E3)THEN
              X3(I)=MOD(X3(I)+E2,K)
              ELSE
              X3(I)=MOD(X3(I)+E3,K)
              ENDIF
              ENDIF
              ENDIF
C             WRITE(9,36)X3(I),I
C36           FORMAT(' ','NEW X3 ==',2X,I3,2X,I3)
              ENDIF
              IF (XP1(I).EQ.NP-JJ)THEN
              ND=ND+1
              E1=MOD(K+MOD(PL1(I)-Z11(I),K),K)
              E2=MOD(K+MOD(PL2(I-2)-Z12(I-2),K),K)
```

```
          E3=MOD(K+MOD(PL3(I-2)-Z13(I-2),K),K)
C        WRITE(9,25)E1,E2,E3
C25         FORMAT(' ',9(I3,8X))
C         WRITE(9,77)Z11(I),I
C77       FORMAT(' ','OLD Z11=====',2X,I3,3X,I3)
          IF(E1.EQ.E2)THEN
          Z11(I)=MOD(Z11(I)+E1,K)
          ELSE
          IF(E1.EQ.E3)THEN
          Z11(I)=MOD(Z11(I)+E1,K)
          ELSE
          IF(E2.EQ.E3)THEN
          Z11(I)=MOD(Z11(I)+E2,K)
          ELSE
          Z11(I)=MOD(Z11(I)+E3,K)
          ENDIF
          ENDIF
          ENDIF
C         WRITE(9,71)Z11(I),I
C71       FORMAT(' ','NEW Z11 ==',2X,I3,2X,I3)
          ENDIF
          IF (XP2(I).EQ.NP-JJ)THEN
          ND=ND+1
          E1=MOD(K+MOD(PL1(I+8)-Z11(I+8),K),K)
          E2=MOD(K+MOD(PL2(I)-Z12(I),K),K)
          E3=MOD(K+MOD(PL3(I-1)-Z13(I-1),K),K)
C        WRITE(9,35)E1,E2,E3
C35         FORMAT(' ',9(I3,8X))
C         WRITE(9,66)Z12(I),I
C66       FORMAT(' ','OLD Z12=====',2X,I3,3X,I3)
          IF(E1.EQ.E2)THEN
          Z12(I)=MOD(Z12(I)+E1,K)
          ELSE
          IF(E1.EQ.E3)THEN
          Z12(I)=MOD(Z12(I)+E1,K)
          ELSE
          IF(E2.EQ.E3)THEN
          Z12(I)=MOD(Z12(I)+E2,K)
          ELSE
          Z12(I)=MOD(Z12(I)+E3,K)
          ENDIF
          ENDIF
          ENDIF
C         WRITE(9,61)Z12(I),I
C61       FORMAT(' ','NEW Z12 ==',2X,I3,2X,I3)
          ENDIF
          IF (XP3(I).EQ.NP-JJ)THEN
          ND=ND+1
          E1=MOD(K+MOD(PL1(I+16)-Z11(I+16),K),K)
          E2=MOD(K+MOD(PL2(I+2)-Z12(I+2),K),K)
          E3=MOD(K+MOD(PL3(I)-Z13(I),K),K)
C        WRITE(9,45)E1,E2,E3
C45         FORMAT(' ',9(I3,8X))
C         WRITE(9,55)Z13(I),I
C55       FORMAT(' ','OLD Z13=====',2X,I3,3X,I3)
          IF(E1.EQ.E2)THEN
          Z13(I)=MOD(Z13(I)+E1,K)
          ELSE
          IF(E1.EQ.E3)THEN
          Z13(I)=MOD(Z13(I)+E1,K)
```

```
            ELSE
            IF(E2.EQ.E3)THEN
            Z13(I)=MOD(Z13(I)+E2,K)
            ELSE
            Z13(I)=MOD(Z13(I)+E3,K)
            ENDIF
            ENDIF
            ENDIF
C           WRITE(9,51)Z13(I),I
C51         FORMAT(' ','NEW Z13 ==',2X,I3,2X,I3)
            ENDIF
           RETURN
           END
           SUBROUTINE STORE(X1,X2,X3,S1,S2,S3,N,NL,IJ)
         INTEGER X1(2000),X2(2000),X3(2000)
         INTEGER S1(2000),S2(2000),S3(2000)
C        NI=N-50
            KL=IJ
           S1(KL)=X1(KL)
           S2(KL)=X2(KL)
           S3(KL)=X3(KL)
           RETURN
           END
          SUBROUTINE RAND(ZX,V,NS)
           INTEGER ZX(6000,12),V(6000),NNN
           NSC=NS+10
           V(1)=4096
           DO 5 I=2,NSC
         ZX(I,1)=MOD((ZX(I-1,2)+ZX(I-1,10)+ZX(I-1,11)+ZX(I-1,12)),2)
           DO 6 J=2,12
            ZX(I,J)=ZX(I-1,J-1)
 6         CONTINUE
           V(I)=ZX(I,1)*2048+ZX(I,2)*1024+ZX(I,3)*512+ZX(I,8)*16
     @+ZX(I,4)*256+ZX(I,5)*128+ZX(I,6)*64+ZX(I,7)*32+ZX(I,9)*8
     @+ZX(I,10)*4+ZX(I,11)*2+ZX(I,12)
C           PRINT,V(I)
 5         CONTINUE
           DO 4 KN=1,12
           ZX(1,KN)=ZX(NS,KN)
 4         CONTINUE
           RETURN
           END
           SUBROUTINE PSTO(PL1,PL2,PL3,SL1,SL2,SL3,N)
          INTEGER PL1(3000),PL2(3000),PL3(3000)
          INTEGER SL1(3000),SL2(3000),SL3(3000)
          SL1(I)=PL1(I)
          SL2(I)=PL2(I)
          SL3(I)=PL3(I)
C         PRINT,PL3(I),'YYYYYYYYYYYYYYYYYYYYYYY',I
          RETURN
          END
```

9 REFERENCES

[1] D. Schilling, D. Manela, R. Pickholtz, and B. Newman. Jr. "Projection Codes- A New Burst and ARQ Codes" MILCOM'87, Oct 87.

[2] D. Schilling, D. Manela and G. Lomp "Projection Codes-A New Class of FEC, Burst Correction and ARQ Codes for Mobile Radio ", ICC'88

[3] D. Schilling, G. Lomp, and A. Pavelchek "Non Binary Forward error Correction for Military Communications" MILCOM'89 Boston, 1989

[4] D. Manela, "A New Class of Forward Error Correcting Codes for Burst and Random Errors" Ph.D. dissertation, City University of New York , June 1987.

[5] G. Lomp " Non-Binary Projection Code" NSF report 1990.

[6] E. Kanterakis " The Projection Code- A New Burst and Random Error Correcting Code" Final Technical report NYS Small Business

Innovation Research.

[7] P. Yu and S. Lin "An Efficient Selective-Repeat ARQ Scheme for Satellite Channels and Its Throughput Analysis" IEEE Transactions on Communications, Vol. Com-29. pp 353-362, March 1981.

[8] S. Lin ,D.J. Costello,Jr. and M.J. Miller "Automatic Repeat Request Error- Control Schemes" IEEE Communications Magazine, vol. com-22. pp 5-16, Dec. 1984.

[9] A.R.K. Sastry "Improving ARQ Performance on a satellite channel Under High Error Rate Conditions" IEEE Transactions on Communications, vol. COM-23, pp 436-439, April 1975

[10] D.M. Mandelbaum "An Adaptive-Feedback Coding Scheme Using Incremental Redundancy" IEEE Transactions on Information Theory, pp 388-389, May 1974.

[11] S. Lin and D.J. Costello "Error Control Coding: Fundamentals and Applications" Prentice Hall pub. New Jersey, 1983.

[12] A. Viterbi and J. Omura "Principles of Digital Communications and Coding" McGraw Hill ,New York, 1979.

[13] R. Blahut "Theory and Practice of Error Control Codes" Addison Wesley, Massachusetts, 1983.

[14] H. Taub and D. Schilling "Principles of Communication Systems 2nd Ed." McGraw Hill, New York, 1986.

[15] J. Proakis "Digital Communications" McGraw Hill, New York, 1983.

[16] J.M. Wozencraft and I.M. Jacobs, "Principles of Communication Engineering" John Wiley & Sons, New York, 1965.

[17] B. Sklar," Digital Communications- Fundamentals and Applications" Prentice Hall, New Jersey, 1988.

[18] Y Wang and S. Lin " A Modified Selective-Repeat Type II ARQ System and Its Performance" IEEE Transactions on Communications, vol. COM-31, pp 593-362, May 1983.

[19] S. Lin and P.S. Yu " A Hybrid ARQ Scheme with Parity Retransmission of Error Control of Satellite Channels" IEEE Transactions on Communications, vol. COM-30, pp 1701-1719, July 1982.

[20] Dong Li "Bounds on Convolutional Projection Codes" PhD dissertation City University of New York 1991.

[21] Yang Gang " Bound on Block Projection Codes" PhD Dissertation City University of New York 1991.

[22] C.B. Schlegel and M.A. Herro " A Burst-Error Correcting Viterbi Algorithm" IEEE Trencactions on Communications , March 1990

[23] T. Kasami, T. Klove and S. Lin, 'Linear Block Codes for Error Detection" IEEE Trans. Info.THeory, pp. 131-136, Jan. 1983.

[24] M. Baulm, P.G. Farrel, and H. Van Tilborg," A Class of Burst Error Correcting Array Codes', IEEE Trans. on Information Theory, pp 836-839, Nov. 1986.

[25] T Fuja, C. Heegard and M. Baulm," Cross parity Convolutional Codes" IEEE trans. on Information Theory, vol. com-38, pp 1904-1915, Nov. 1989.