Design of an ARQ/AFEC Link in the Presence of Propagation Delay and Fading

by

Bradley Bisham Comar

B.S., Electrical Engineering (1995)

Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering

at the

Massachusetts Institute of Technology

June 2000

Signature of Author: _____

Department of Electrical Engineering
May 17, 2000

Certified by: _____

Dr. Steven Finn
Principal Research Scientist
Thesis Supervisor

Accepted by: _____

Arthur C. Smith
Professor of Electrical Engineering
Chairman, Committee on Graduate Students

Design of an ARQ/AFEC Link in the Presence of Propagation Delay and Fading

by

Bradley Bisham Comar

Submitted to the Department of Electrical Engineering
on May 18, 2000 in Partial Fulfillment of the
Requirements for the Degree of Master of Engineering in
Electrical Engineering

ABSTRACT

This thesis involves communicating data over low earth orbit (LEO) satellites using the Ka frequency band. The goal is to maximize throughput of the satellite link in the presence of fading and propagation delay. Fading due to scintillation is modeled with a log-normal distribution that becomes uncorrelated with itself at 1 second. A hybrid ARQ/AFEC (Adaptive Forward Error Correction) system using Reed-Muller (RM) codes and the selective repeat ARQ scheme is investigated for dealing with errors on the link. When considering links with long propagation delay, for example satellite links, the delay affects the throughput performance of an ARQ/AFEC system due to the lag in the system's knowledge of the state of the channel.

This thesis analyzes various AFEC RM codes (some using erasure decoding and/or iterative decoding) and two different protocols to coordinate the coderate changes between the transmitter and receiver. We assume the receiver estimates the channel bit error rate (BER) from the data received over the channel. With this BER estimation, the receiver decides what the optimal coderate for the link should be in order to maximize throughput. This decision needs to be sent to the transmitter so that the transmitter can send information at this optimal coderate. The first protocol investigated for getting this information back to the transmitter involves attaching a header on every reverse channel packet to relay this information. The second protocol we explore leaves the information packets unchanged and makes use of special control packets to coordinate the rate changes between the transmitter and receiver.

Thesis Supervisor: Dr. Steven Finn
Title: Principal Research Scientist

[This page is intentionally left blank]

TABLE OF CONTENTS

[This page is intentionally left blank]

1.0 INTRODUCTION

This thesis involves communicating data over low earth orbit (LEO) satellites using the Ka frequency band. The goal is to maximize the throughput of the satellite link in the presence of long propagation delay. It is assumed that the communication system uses power control to negate the effects of fading due to rain attenuation. However, fading due to scintillation must be dealt with. This fading is modeled with a log-normal distribution that becomes uncorrelated with itself at 1 second.

When communicating digital information over noisy channels, the problem of how to handle errors due to noise must be addressed. The two major schemes for dealing with errors are forward error correction (FEC) and automatic retransmission request (ARQ) [8]. Combining these schemes into a hybrid ARQ/FEC system can be used to make more efficient communications over the link [8]. Additional performance is gained by using an adaptive FEC, or AFEC, scheme which changes coding rate as the signal to noise ratio in the channel changes in order to maximize throughput performance. Throughput performance is also gained by using the selective repeat ARQ scheme, which our system uses. Throughput is defined as the inverse of the average number of transmitted bits required to successfully send one bit of information. When considering links with long propagation delay, for example satellite links, the delay affects the performance of an ARQ/AFEC system due to the lag in the system's knowledge of the state of the channel.

This thesis addresses some of the analysis that a designer of such an ARQ/AFEC system should consider. The two main phases of this thesis are 1) to analyze various AFEC codes and 2) to analyze different protocols to coordinate the coderate changes between the transmitter and receiver. A good AFEC code is determined from phase 1 of this project and then a good coordination protocol for this chosen AFEC is determined in phase 2. Our hybrid ARQ/AFEC scheme uses a CRC check code for error detection in addition to the AFEC code for error correction.

In phase one, a set of codes is chosen to construct the AFEC code. Here, the major focus is on linear Reed-Muller (RM) codes and two-dimensional (2-D) product codes constructed with these RM codes. 2-D product codes are

explored because literature indicates that they are efficient and powerful [9]. Because the sizes of 2-D codes are the square of the sizes of the codes that construct them, these constructing codes should be limited to reasonably small sizes. RM codes are chosen to construct these 2-D codes because they are optimal at smaller sizes [5] and they have the added benefit of a known hard decision decoding algorithm that is faster and simpler than other comparable block codes [12]. In addition to hard decision decoders, another set of decoders implements soft decision (erasure) decoding. This decoding method allows for three decision regions. Erasure decoding is proposed because this decoding method allows for easy and efficient hardware or software implementation [12]. Therefore, the four sets of codes explored are normal RM codes with hard decision decoders, 2-D RM codes with hard decision decoders, normal RM codes with erasure decoders, and 2-D RM codes with erasure decoders.

The second phase of this thesis involves incorporating the AFEC code chosen in phase one with the selective repeat ARQ scheme. Sufficient buffering to avoid overflow is assumed on both the transmit and receive sides of the channel. The fading on our channel is assumed to be log-normally distributed. Two different protocols to handle the change in the AFEC coderate are analyzed. We assume the receiver estimates the channel bit error rate (BER) from the data received over the channel. With this BER estimation, the receiver decides what the optimal coderate for the link should be in order to maximize throughput. This decision needs to be sent to the transmitter so that the transmitter can send the information at the optimal coderate.

The first protocol for getting this information back to the transmitter involves attaching a header on every reverse channel packet to relay this information. The header is encoded with the rest of the packet during transmission. The receiving modem tries decoding every packet it receives with decoders set at each possible coderate. If none or more than one receiver decodes a packet that passes the CRC check, a failure is declared and a request for retransmission is issued. If exactly one decoder has a packet that passes the CRC check, that packet is determined good and is passed to the higher layer. A disadvantage of using this scheme is that every packet has this additional overhead. This might be a heavy price to

pay if the BER is fairly constant for most of the time and the coderate changes infrequently.

The second protocol we explore leaves the information packets unchanged and makes use of special control packets to coordinate the rate changes between the transmitter and receiver. When the receiver wants the coderate of the transmitter to change, it sends a "change coderate" control packet to the transmitter telling it what rate to change to. These packets can be encoded using the lowest coderate which gives the lowest probability of error. Again, the receiver can track the transmitter by using several decoders and choosing the correctly decoded packet as determined by the CRC code. The benefit of this second protocol over the first protocol described above is more efficient use of the link if the coderate does not change often.

The ARQ/AFEC system under these protocols and log-normal fading is analyzed for throughput. For a fading model with a mean of 5dB E_b/N_0 and a standard deviation of 2, the better protocol depends on the data rate. For data rates higher than 10 Mbps, the second protocol, which uses control packets, is more throughput efficient. For data rates lower than 2 Mbps the first protocol, which places coderate information in the header, is more throughput efficient. For data rates in between, the choice of protocols does not effect throughput significantly.

[This page is intentionally left blank]

2.0 PHASE I : FINDING AN ADAPTIVE CODE

An adaptive FEC code is a code that can change its coderate in response to the noise environment. In this work, an adaptive ARQ/FEC Hybrid system is designed in order to maximize throughput. During periods of low noise, the system can transmit information at a high coderate and not waste efficiency transmitting too many redundancy bits. During high noise time periods, the system can transmit at a lower coderate and not waste efficiency by re-transmitting packets too many times. The receiver will monitor incoming packets, estimate the noise environment, choose which FEC code should be used to maximize throughput, and relay this information to the transmitter.

The first step in designing such a system is to choose an adaptive code. This adaptive code will be a collection of several individual codes with different coderates. There are many varieties of codes to choose from. This project concentrates on two-dimensional iterative block codes constructed with Reed-Muller codes. See Appendix 1. After looking at different RM codes, the set of codes of length n=64 is chosen. The RM(1,6), RM(2,6), RM(3,6) and RM(4,6) codes are used. The values of k for these codes are 7, 22, 42, and 57 respectively. Their coderates are 7/64, 22/64, 42/64 and 57/64 and the coderates of the respective 2-D codes they construct are $7^2/64^2$, $22^2/64^2$, $42^2/64^2$ and $57^2/64^2$. The methods of iterative decoding in this thesis are greatly simplified from traditional iterative block decoding which is described in Appendix 1g.

The RM codes mentioned above are used to create four 2-D codes that use hard decision decoding. These codes are encoded in the normal manner for 2-D codes. See Appendix 1g. The decoding algorithm is as follows. Hard decode all the columns in the received n x n matrix $\mathbf{R}_0$ to create a new k x n matrix $\mathbf{R}_V'$. Re-encode $\mathbf{R}_V'$ to create a new n x n matrix $\mathbf{R}_V$. $\mathbf{R}_V$ now has valid codewords for each of its columns that $\mathbf{R}_0$ may not necessarily have. Subtract $\mathbf{R}_V$ from $\mathbf{R}_0$ to get $\mathbf{E}_V$. $\mathbf{E}_V$ is the error pattern matrix. It will have 1s in the locations where the bits in $\mathbf{R}_V$ and $\mathbf{R}_0$ differ and 0s in the locations where they agree. Run this same procedure with the rows of $\mathbf{R}_0$ to create the matrix $\mathbf{E}_H$. Element-wise multiply the two error pattern matrices to create the matrix $\mathbf{E}$. $\mathbf{E}$ represents the locations in $\mathbf{R}_0$ where both horizontal and vertical decoding agree are corrupted. Change these bits by subtracted $\mathbf{E}$ from $\mathbf{R}_0$. Thus $\mathbf{R}_1 = \mathbf{R}_0 - \mathbf{E}$.

See Figure 1. This procedure can be iterated as many times as desired. After iterating x times, decode $\mathbf{R}_x$ horizontally to get the k x n matrix $\mathbf{R}_x'$ and decode the columns of $\mathbf{R}_x'$ to get the k x k matrix of message bits. These codes will be referred to as 2-D Hard codes in this paper.

Another set of codes to be analyzed is similar to the set of 2-D Hard codes described above except that decoding the rows and columns of $\mathbf{R}_0$ is done with erasure decoding as described in Appendix 1f. The decoding procedure is as follows. Erasure decode all the columns in the received n x n matrix $\mathbf{R}_0$, which may have erasure (?) bits, to create a new k x n matrix $\mathbf{R}_V'$. Re-encode $\mathbf{R}_V'$ to create a new n x n matrix $\mathbf{R}_V$, which will not have any erasure (?) bits. Perform a special element-wise "erasure addition" on the two matrices $\mathbf{R}_0$ and $\mathbf{R}_V$ to get a new horizontal matrix $\mathbf{R}_{VX}$. This special addition is labeled as (+) and defined in table 2.1.

1 (+) 0 = ?	? (+) 0 = 0	0 (+) 0 = 0
0 (+) 1 = ?	0 (+) ? = 0	1 (+) 1 = 1
	? (+) 1 = 1	
	1 (+) ? = 1	

Table 2.1: The (+) operation.

This addition is intuitively structured so that if one matrix says a bit should 0 and the other says it should be 1, $\mathbf{R}_{VX}$ will determine it undecided or ?. If one matrix is undecided and the other has a solution, that solution is accepted. Finally, if both matrices have solutions that match, the solution is accepted. Repeat this procedure on the rows of $\mathbf{R}_0$ to create $\mathbf{R}_{HX}$. Erasure add $\mathbf{R}_{HX}$ and $\mathbf{R}_{VX}$ to create $\mathbf{R}_1 = \mathbf{R}_{HX}$ (+) $\mathbf{R}_{VX}$. This procedure can be iterated as many times as desired. After iterating x times, erasure decode $\mathbf{R}_x$ horizontally to get the k x n matrix $\mathbf{R}_x'$ and erasure decode the columns of $\mathbf{R}_x'$ to get the k x k matrix of message bits. These codes will be referred to as 2-D Soft codes in this paper. The decision regions that define when a received bit should be decided as a 0, 1 or ? are determined experimentally.

The last two sets of codes use the same k x k matrix of k^2 message bits. These matrices are encoding by encoding each row of the message matrix with the appropriate RM code to create a k x n matrix that gets transmitted to the

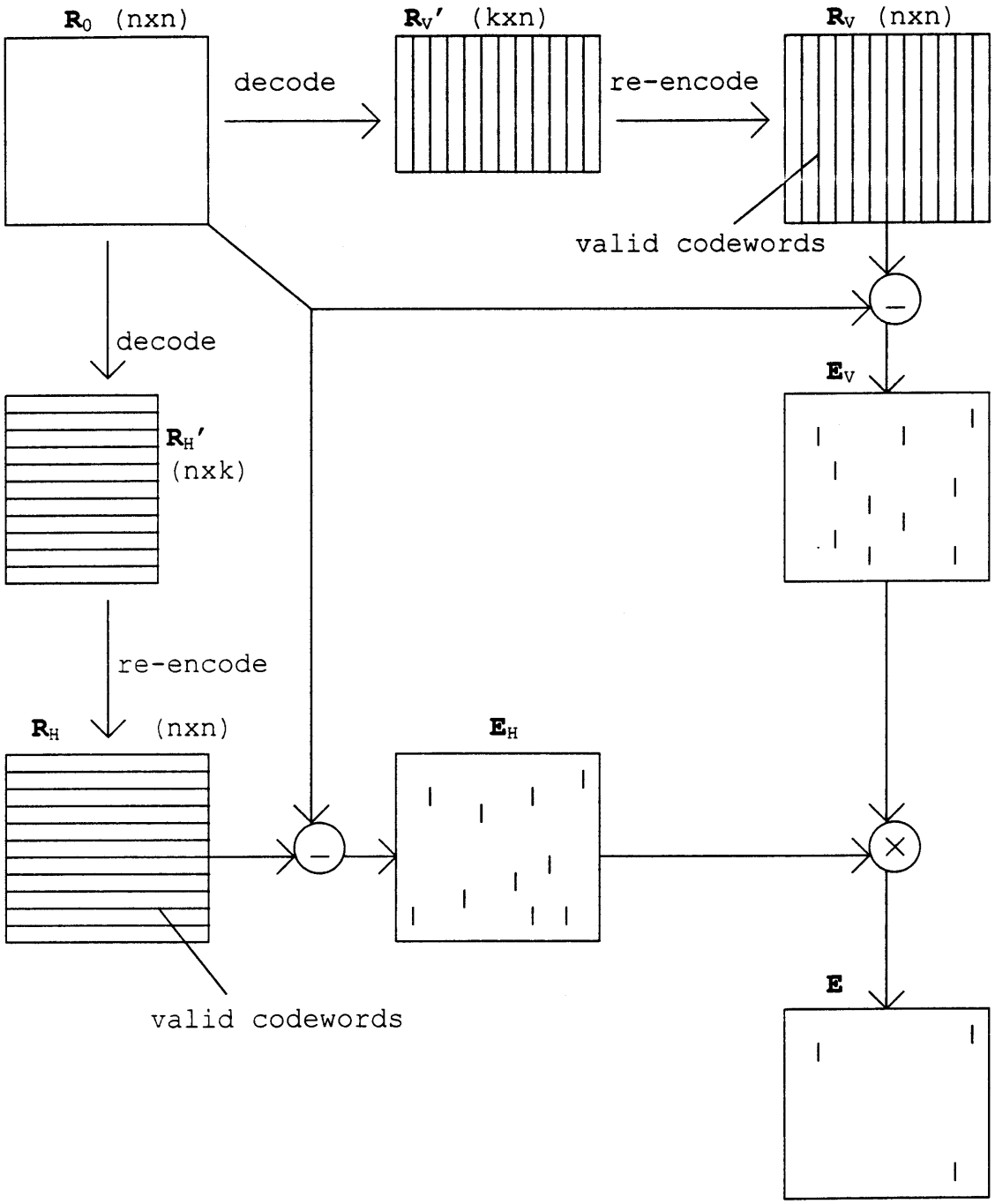
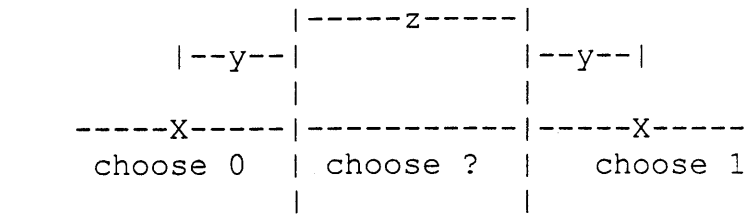


FIGURE 1: This figure shows the procedure for making matrix **E** in the iterative decoding algorithm. $\mathbf{R}_1 = \mathbf{R}_0 - \mathbf{E}$. Note that binary field subtraction is equivalent to binary field addition. (See Appendix 1b).

receiver. The received matrix $\mathbf{R}_0$ can be decoded by hard decoding each row of $\mathbf{R}_0$ to regenerate the k x k message matrix. This set of codes is referred to as 1-D Hard codes. See Appendix 2a. $\mathbf{R}_0$ can also be decoded by erasure decoding each row of $\mathbf{R}_0$ to regenerate the k x k message matrix. This set of codes is referred to as 1-D Soft codes. These 1-D codes are used as baseline codes that are used to evaluate the performance of the 2-D codes described above.

2.1 Procedure and Results of Phase I

Before analyzing these codes, decision regions for the soft codes must be determined. The first regions in signal space that are tried are: (See Appendix 1e)



Here, the length of z equals two times the length of y. Letting a = sqrt(E_b/N_0) and b = sqrt(energy of signal):

P(error)	= Q(distance to travel / s)
z + 2y = 3z = 2b	=> z = 2b/3
P(erasure or error)	= Q(2b/3s) = Q(2a/3)
P(error)	= Q(4a/3)
P(erasure)	= Q(2a/3) - Q(4a/3)

When these decision regions are tried, the soft decision decoders perform worse than the hard decision decoders in both the 1-D and 2-D codes. Therefore, we tried other size regions to see if performance can be improved. We tried the following region sizes:

P(error)=Q(0.80a)	P(erasure)=Q(1.20a)-Q(0.80a)
P(error)=Q(0.85a)	P(erasure)=Q(1.15a)-Q(0.85a)
P(error)=Q(0.90a)	P(erasure)=Q(1.10a)-Q(0.90a)
P(error)=Q(0.95a)	P(erasure)=Q(1.05a)-Q(0.95a)

The third row of probabilities works best for the 2-D Soft codes. This corresponds to a value of z that is (0.22…)y. For 1-D soft codes, no decision region works well. When the decision region for ? shrinks to zero, the performance

of this code approaches the performance of 1-D hard codes. The third row of probabilities is used to calculate the performance of 1-D soft codes.

Due to a programming mistake found while debugging C code, a peculiar property with the 2-D soft codes is discovered. When calculating $\mathbf{R}_1 = \mathbf{R}_{HX}$ (+) $\mathbf{R}_{VX}$, better performance is gained defining this erasure addition as shown in table 2.2.

1 (+) 0 = 1	? (+) 0 = 0	0 (+) 0 = 0
0 (+) 1 = 0	0 (+) ? = 0	1 (+) 1 = 1
	? (+) 1 = 1	
	1 (+) ? = 1	

Table 2.2: A new (+) operation.

When the horizontal and vertical matrices disagree, side with the horizontal matrix. This new calculation, labeled $\mathbf{R}_1 = \mathbf{R}_{HX}$ $(+)_H$ $\mathbf{R}_{VX}$, does not get better with more iterations while $\mathbf{R}_1 = \mathbf{R}_{HX}$ (+) $\mathbf{R}_{VX}$ does. Thus, the best performance of these codes is gained when doing $\mathbf{R}_a = \mathbf{R}_{HX}$ (+) $\mathbf{R}_{VX}$ on the first n-1 iterations and doing $\mathbf{R}_n = \mathbf{R}_{HX}$ $(+)_H$ $\mathbf{R}_{VX}$ on the last iteration. A value of n=6 seems to work reasonable well. Going beyond this point requires more calculations and achieves less benefit form the extra work. In this thesis, the first five iterations use (+) while the last iteration used $(+)_H$. See Appendix 2b. Note that this problem is not horizontal specific. If the rows are decodes first and the columns second during the iterations, then the best codes requires siding with the vertical matrix. Therefore, it does not matter whether the columns or rows are decoded first as long as $(+)_X$ is consistent. In this project, the columns are decoded first.

Calculating the throughput of these codes under the selective repeat ARQ scheme assuming sufficient buffering on both ends to avoid overflow makes use of the throughput equation: $T_{sr} = P_c R$ where R is the coderate and P_c is the probability that the matrix is decoded correctly. See Appendix 1h. The simulated performance of 1000 matrices transmitted for each code under each of several different E_b/N_0s is run. Table 2.3 shows the simulated error probabilities and Figure 2 shows the throughput performance graph calculated from these error probabilities.

Codes constructed by RM(1,6):

Eb/N0	1-D Hard	1-D Soft	2-D Hard	2-D Soft
0.2	95.5	95.2	.1	0
0.4	29	26.7	0	0
0.6	1.9	1.8	0	0
0.8	0	0	0	0

Codes constructed by RM(2,6):

Eb/N0	1-D Hard	1-D Soft	2-D Hard	2-D Soft
0.4	100	100	100	100
0.6	100	100	82.5	12.3
0.8	99.6	100	0	0
1.0	90.2	92.8	0	0
1.2	49.2	54.4	0	0
1.4	16.1	19.0	0	0
1.6	3.8	5.4	0	0
1.8	0.9	2.2	0	0
2.0	0.1	0.5	0	0
2.5	0	0	0	0

Codes constructed by RM(3,6):

Eb/N0	1-D Hard	1-D Soft	2-D Hard	2-D Soft
1.2	100	100	100	100
1.4	100	100	99.9	92.4
1.6	100	100	75.4	25.7
1.8	99.1	99.6	15.8	1.3
2.0	91.8	96.3	0.9	0
2.5	31.0	43.8	0	0
3.0	5.6	10.0	0	0
3.5	0.8	1.5	0	0
4.0	0	0.1	0	0
4.5	0	0	0	0

Codes constructed by RM(4,6):

Eb/N0	1-D Hard	1-D Soft	2-D Hard	2-D Soft
2.0	100	100	100	100
2.5	100	100	100	99.6
3.0	98.8	99.9	92.4	69.8
3.5	78.7	91.6	43.2	14.7
4.0	45.0	61.7	10.0	2.0
4.5	17.3	32.2	1.7	0.2
5.0	6.1	14.0	0.2	0
5.5	2.8	6.1	0.1	0
6.0	1.3	2.2	0	0
6.5	0.3	0	0	0

Table 2.3: P(Matrix error) with 1000 packets passed. E_b/N_0 is in the linear scale.

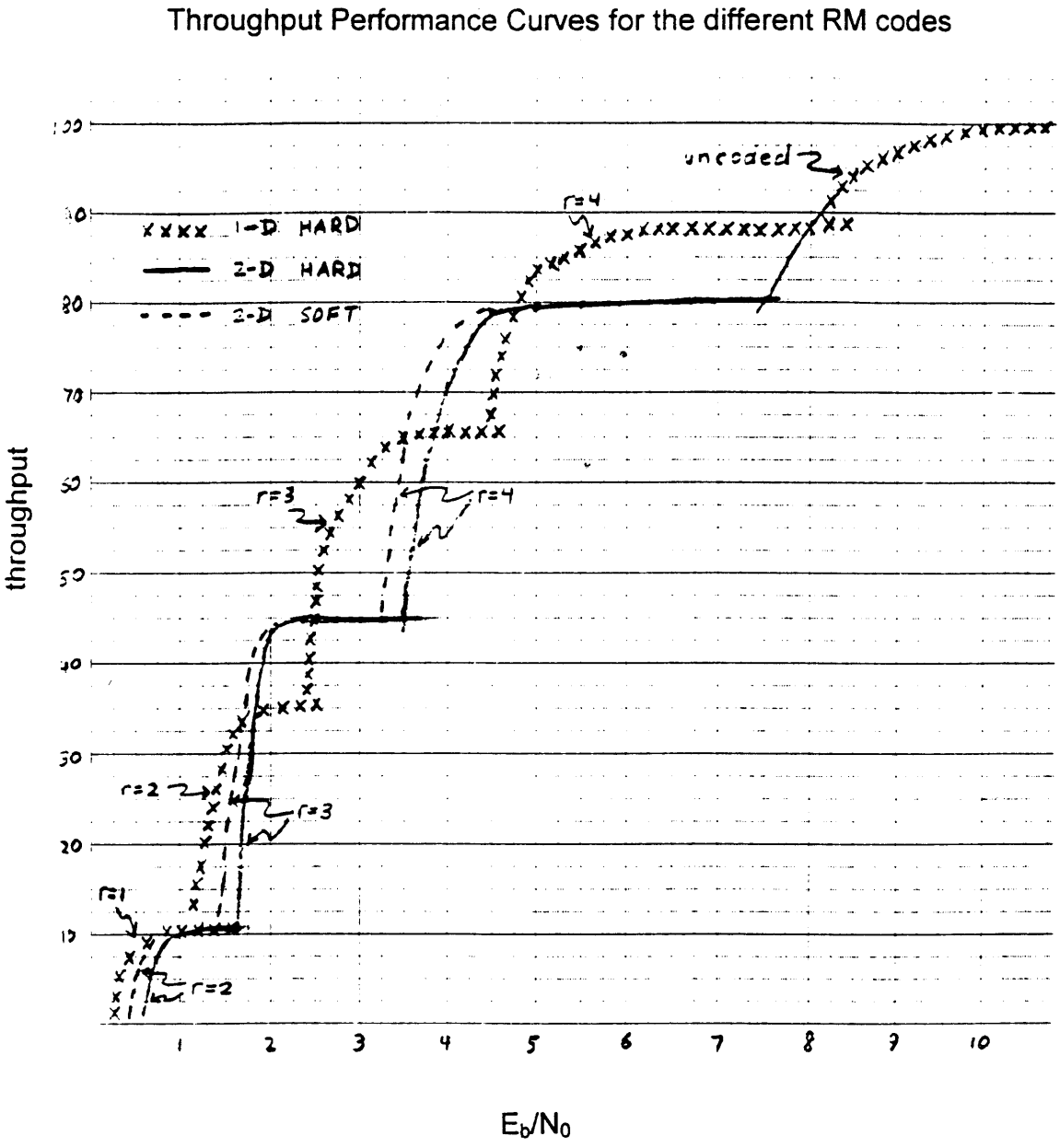


FIGURE 2: These throughput performance curves are obtained by taking the error probability information in table 2.3 and multiplying the probabilities by the coderates of the respective codes.

From the information in Figure 2, the 1-D hard set of codes is chosen to make the adaptive FEC code. This code does not always perform as well as the 2-D codes, but it comes close. The 2-D codes require much more calculations, and thus they become unattractive. It should be noted that the 2-D codes give much better BER performance, but they operate with lower coderates which pushes their throughput performance back to levels similar to those of the 1-D hard code.

3.0 ANALYSIS OF THE 1-D HARD ADAPTIVE CODE

The set of 1-D hard codes is modified so that all the codes are the same size. This is a good idea so that the transmitter and receiver do not lose synchronization when coderate information is miss-communicated. Currently, the k x n matrix sizes vary because the values of k vary. Therefore, the number of rows needs to be fixed to a value ℓ. The size of the ℓ x n matrix is $\ell \cdot n$ and it contains $\ell \cdot k$ message bits. Another factor to consider is that extra overhead bits are needed. In this project, 64 bits are allocated for a 32 bit CRC-32 checksum, a 24 bit packet number field, and 8 bits for other control information. Therefore, the new coderate of the individual 1-D hard codes is $R=(k\ell-64)/n\ell$ and throughput of these codes is $P_cR=(1-P_e)(k\ell-64)/n\ell$.

In order to verify the results of the simulated throughput performance of the chosen code, the union bound estimate is used.

$$P(\text{row correctly decoded}) = P(0 \text{ bit errors}) + P(1 \text{ bit error}) + P(2 \text{ bit errors}) + \ldots + .5P(d_{min}/2 \text{ errors})$$

The last term is added into the equation because when exactly $d_{min}/2$ errors occur, the decoder picks one of the two nearest codes and is correct half the time. The decoding is deterministic, but the probabilities of occurrence of the two codes to decide upon are equal. Note that the codes used in this project all have even d_{min}s. Also note that these equations assume only one nearest neighbor for each codeword. If there is more than one nearest neighbor, the probabilities calculated will be less than the actual probabilities. Even if there is only one nearest neighbor, other neighbors a bit further away may make the union bound estimate smaller that the actual probabilities. The calculated probabilities of correctly decoded rows are:

$$P(\text{for RM}(4,6)) = x^{64} + \text{Comb}(64\ 1)(1-x)x^{63} + .5\text{Comb}(64\ 2)(1-x)^2x^{62}$$

$$P(\text{for RM}(3,6)) = x^{64} + \text{Comb}(64\ 1)(1-x)x^{63} + \text{Comb}(64\ 2)(1-x)^2x^{62} + \text{Comb}(64\ 3)(1-x)^3x^{61} + .5\text{Comb}(64\ 4)(1-x)^4x^{60}$$

$$P(\text{for RM}(2,6)) = x^{64} + \text{Comb}(64\ 1)(1-x)x^{63} + \ldots$$

$$P(\text{for RM}(1,6)) = x^{64} + \text{Comb}(64\ 1)(1-x)x^{63} + \ldots$$

The solutions for these calculations appear in table 3.1. Also in this table is the simulated calculations for 1000 matrices of one row for comparison. These matrices have ℓ=1 and are traditional RM codes.

Eb/N0	RM(1,6)	RM(2,6)	RM(3,6)	RM(4,6)
0.2	36.6%/58.8%	98.7/99.6	100 / 100	100 / 100
0.4	4.8/ 9.8	85.7/89.5	99.9/99.7	100 / 100
0.6	0.2/ 7.8	56.0/59.0	96.7/96.5	99.9/99.7
0.8	0/ 0.4	26.2/27.3	88.9/85.4	98.2/97.9
1.0	0/ 0.2	9.2/ 9.5	72.9/66.2	94.6/92.7
1.2	0/ 0	3.2/ 2.7	54.8/44.8	88.3/82.9
1.4	0/ 0	0.6/ 6.6	34.1/27.0	80.2/69.8
1.6	0/ 0	0/ 1.4	21.0/14.8	68.7/55.4
1.8	0/ 0	0/ 0.3	10.8/ 7.6	56.0/41.9
2.0	0/ 0	0/ 0	5.4/ 3.7	41.1/30.4
2.5	0/ 0	0/ 0	1.0/ 0.5	19.8/12.1
3.0	0/ 0	0/ 0	0.2/ 0.1	7.8/ 4.4

Table 3.1: 1000 Packet Simulation vs. Analytic Solution. The following table shows the simulated probability of error P(e) data run on the 1-D Hard C program in Appendix 2a with a slight modification. The number of rows is changed from k to 1. A C program is used to calculate the union bound estimate as shown in section 3. The format for the table is (simulated P(e))/ (analytic P(e)).

Note that the analytic solutions are close to the simulated solutions. Given that the analysis is an approximation, the simulation algorithm will be used to obtain values for the probability of matrix error. To get more accurate calculations, 100000 matrices (data packets) are transmitted in the simulation program for each RM code as a function of E_b/N_0. The results of these simulations on packets of ℓ=1 appear in table 3.2.

For the packets constructed with the RM(2,6) code:
P(e)=0.000 for all E_b/N_0 values tested.

For the packets constructed with the RM(2,6) code:

E_b/N_0	P(e)	E_b/N_0	P(e)	E_b/N_0	P(e)
0.25dB	6.852%	1.75	0.401	3.25	0.001
0.50	4.779	2.00	0.190	3.50	0.001
0.75	3.207	2.25	0.097	>3.5	0.000
1.00	2.013	2.50	0.038		
1.25	1.256	2.75	0.014		
1.50	0.706	3.00	0.006		

For the packets constructed with the RM(3,6) code:

E_b/N_0	P(e)	E_b/N_0	P(e)	E_b/N_0	P(e)
0.25dB	67.611%	2.25	16.186	4.25	0.460
0.50	61.298	2.50	11.895	4.50	0.244
0.75	54.542	2.75	8.414	4.75	0.103
1.00	47.494	3.00	5.803	5.00	0.040
1.25	40.414	3.25	3.722	5.25	0.020
1.50	33.500	3.50	2.370	5.50	0.007
1.75	27.204	3.75	1.413	5.75	0.003
2.00	21.333	4.00	0.815	6.00	0.001
				>6.0	0.000

For the packets constructed with the RM(4,6) code:

E_b/N_0	P(e)	E_b/N_0	P(e)	E_b/N_0	P(e)
0.25dB	95.026%	3.25	36.460	6.25	0.603
0.50	93.160	3.50	30.134	6.50	0.361
0.75	90.860	3.75	24.323	6.75	0.184
1.00	87.981	4.00	19.111	7.00	0.096
1.25	84.420	4.25	14.644	7.25	0.042
1.50	80.180	4.50	10.962	7.50	0.022
1.75	75.262	4.75	7.871	7.75	0.015
2.00	69.611	5.00	5.515	8.00	0.006
2.25	63.344	5.25	3.783	8.25	0.004
2.50	56.850	5.50	2.517	8.50	0.001
2.75	50.018	5.75	1.645	>8.5	0.000
3.00	43.186	6.00	1.038		

Table 3.2: 100000 Packet Simulation. The following table shows the simulated probability of error P(e) data run on the 1-D Hard C program in Appendix 2a with 1 row. This simulation differs from the one recorded in table 3.1 because it is run on 100000 packets instead of 1000 packets and E_b/N_0 is in the dB scale here while it is in a linear scale above. The probabilities of errors here are used to characterize the performance of the 1-D Hard codes.

Our adaptive code, which can code messages at any of the four coderates discussed above, can also choose not to code the message at all. To determine the probability of error for uncoded packets, the simulation program is run with 100000 packets without encoding or decoding. The

following calculations are made to compare with the simulations. Note that this calculation does not depend on approximations as did the calculations for coded data.

$$\begin{aligned} P(\text{row error}) &= 1-P(\text{good row}) = 1-(1-P(\text{bit error}))^{64} \\ [\text{no coding}] &= 1-(1-Q(\sqrt{2E_b/N_0}))^{64} \\ &= 1-(1-.5\,\text{erfc}(2E_b/N_0)^{64}) \\ &= 1-(1-.5\,\text{erfc}(\exp(.1(E_b/N_0 \text{in dB})\ln 10))^{64}) \end{aligned}$$

The last line is a trick used because MS Excel does not have an antilog function. The analytic solutions and simulations at 100000 passed packets match to within %0.1. See table 3.3. This case validates the accuracy of the simulations. For the case where there is no coding, the throughput performance curve uses the analytic solutions and not the simulations. Now that the probability of an error in a row is determined for all codes, the probability of an ℓ row packet error is: P(packet error) = 1 - P(good packet) = 1-P(every row is good) = 1 - [1-P(row error)]$^{\ell}$.

E_b/N_0	Sim'd P(e)	Analytic P(e)	E_b/N_0	Sim'd P(e)	Analytic P(e)
0.25dB	99.183%	99.205%	5.25	26.684	26.610
0.50	98.762	98.824	5.50	22.026	21.938
0.75	98.205	98.287	5.75	17.751	17.791
1.00	97.471	97.546	6.00	14.145	14.190
1.25	96.464	96.545	6.25	11.065	11.127
1.50	95.175	95.220	6.50	8.613	8.575
1.75	93.437	93.507	6.75	6.425	6.492
2.00	91.347	91.341	7.00	4.765	4.827
2.25	88.690	88.665	7.25	3.465	3.522
2.50	85.559	85.436	7.50	2.508	2.520
2.75	81.623	81.634	7.75	1.745	1.768
3.00	77.279	77.264	8.00	1.194	1.214
3.25	72.341	72.362	8.25	0.784	0.816
3.50	66.919	66.998	8.50	0.581	0.536
3.75	61.147	61.268	8.75		0.344
4.00	55.217	55.295	9.00		0.215
4.25	49.087	49.219	9.25		0.131
4.50	43.121	43.181	9.50		0.077
4.75	37.287	37.321	9.75		0.045
5.00	31.864	31.763	…		

Table 3.3: Analytic Solution & Simulation for No Coding. The simulation for no coding is run for 100000 packets and is compared to the calculations as shown in section 3. Simulations are stopped after E_b/N_0 of 8.5dB while the calculations continue until P(e) rounds to 0.000.

4.0 PHASE II: ARQ/AFEC IN A FADING ENVIRONMENT

A log-normal fading environment with an E_b/N_0 that is constant over the duration of one packet transmission is assumed. The overall throughput over a long period of time that includes fading is the average of the throughput performance curve (with E_b/N_0 in dB) scaled by the Gaussian PDF curve of a certain mean $\boldsymbol{m}$ and standard deviation $\boldsymbol{s}$.

MATLAB code is written to calculate this throughput for values of $\boldsymbol{m}$ that range from 1 to 10 in steps of 1 and values of $\boldsymbol{s}$ from .25 to 2.5 in steps of .25. Values of ℓ ranging from 1 to 500 are calculated for each case and the value that gives the optimal throughput for each $\boldsymbol{m}$ and $\boldsymbol{s}$ for each code is recorded in figure 3. The graphs have sections with a ceiling of 500, but the actual optimal values of ℓ will be higher in these regions. The optimal throughputs at these values of ℓ are shown in figure 4. Figure 5a shows the best throughput for any of the individual codes at each value of $\boldsymbol{m}$ and $\boldsymbol{s}$. This graph represents the best that can be done in a location with noise characteristics $\boldsymbol{m}$ and $\boldsymbol{s}$ where the system designer can choose any fixed code but not an adaptive code. Figure 5b has a graph that shows which code gives that optimal performance. Notice that in figure 5, the values of $\boldsymbol{s}$ are extended to 5.0 and the step size of $\boldsymbol{m}$ is decreased to 0.5. Figure 6 shows the performance of an adaptive code and the value of ℓ that should be used for that code. All the constituent fixed codes that make up the adaptive code have ℓ rows at each value of $\boldsymbol{m}$ and $\boldsymbol{s}$ so that all packets are the same size at each value of $\boldsymbol{m}$ and $\boldsymbol{s}$. Therefore, picking a value ℓ for the adaptive code is a compromise that results in each constituent code not necessarily having the optimal size. Figure 7 shows what is gained by using the adaptive code of figure 6a instead of the best fixed code of figure 5a. The graph shows percent change by plotting throughput of the adaptive code divided by throughput of the fixed code. As expected, when the standard deviation is small, the impact of using the adaptive code is minimal because only one of its constituent codes is running most of the time. As s increases, the adaptive code becomes a better solution. Appendix 3 shows the program that is used to generate the graphs in figures 3-7.

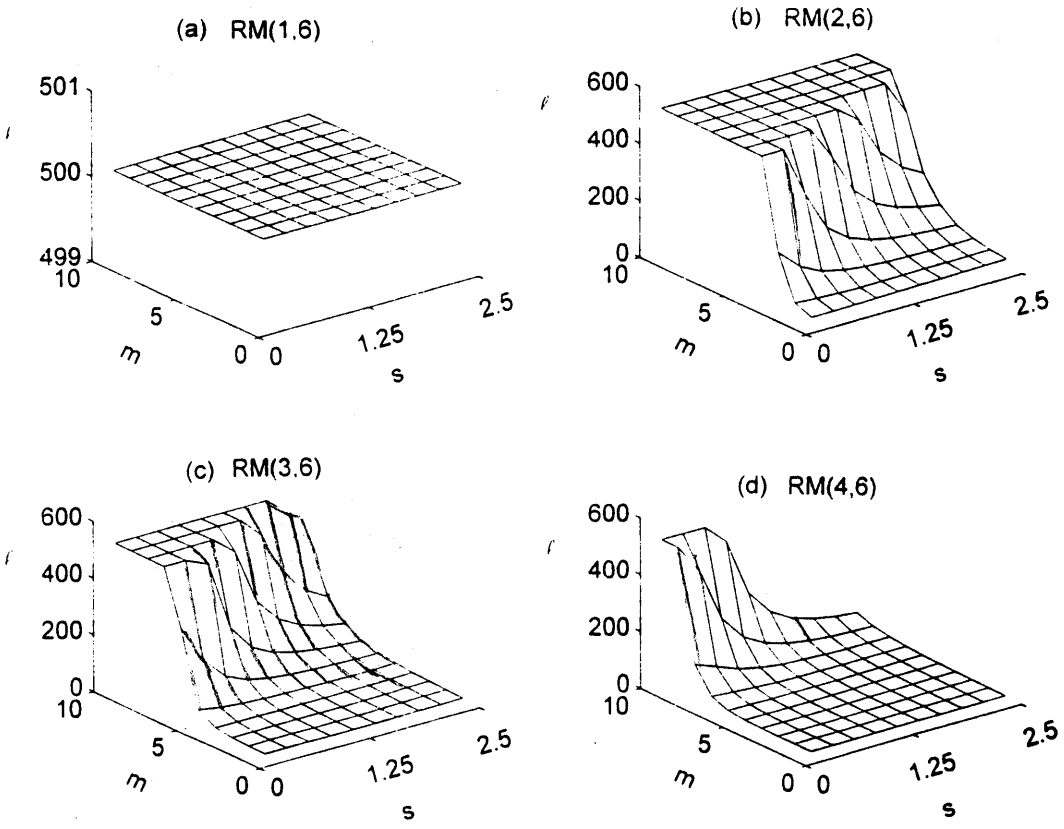


FIGURE 3: These graphs show the values of ℓ which give the optimal throughput for 1-D Hard codes in a hybrid selective repeat ARQ scheme. Values of ℓ are shown for fading environments with mean values of m ranging from 0dB to 5dB and standard deviation values s ranging from .25 to 2.5. Note that values of ℓ are restricted to a ceiling of 500.

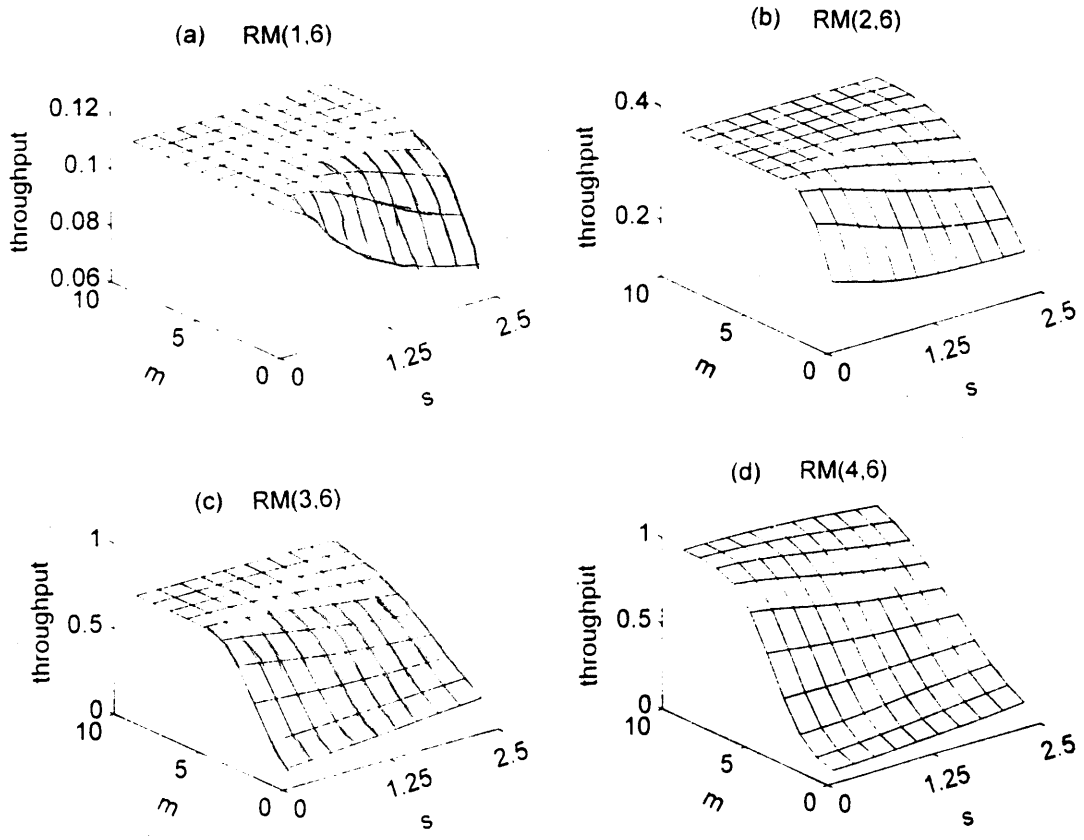
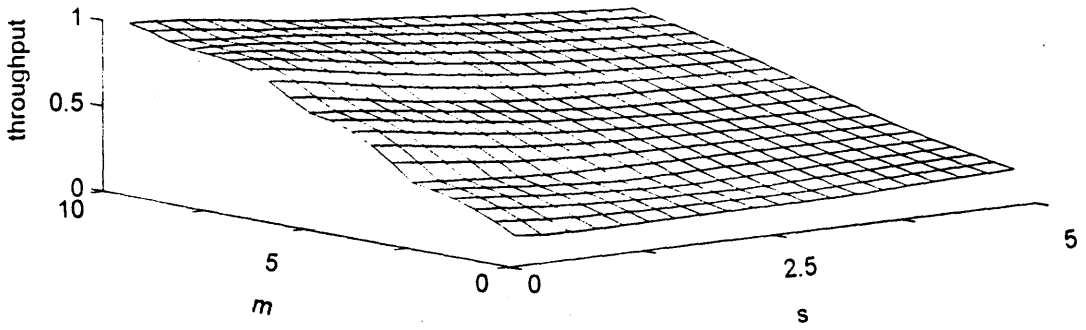


FIGURE 4: These graphs show the optimal throughput of the constituent codes at the values of ℓ in figure 3.

a) Non-Adaptive Code Throughput



b) Non-Adaptive Code: Constituent Code Selection

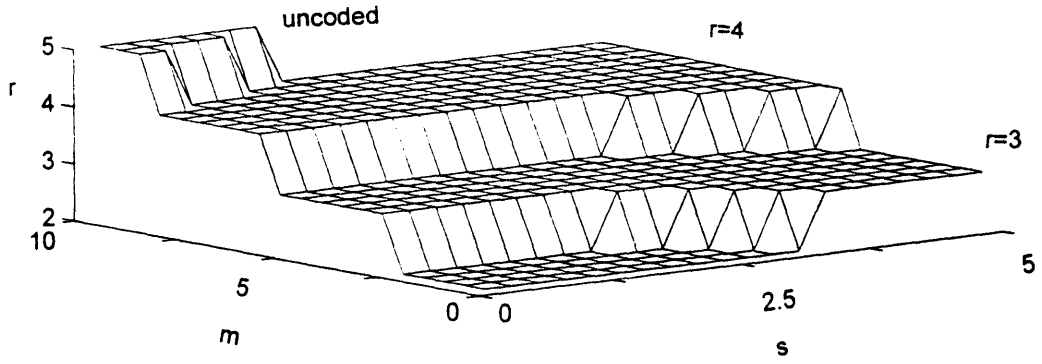


FIGURE 5: The first graph, figure 5a, is the composite of the optimal throughputs of the constituent codes in figure 4. This graph represents the best throughput that can be achieved using a non-adaptive FEC code with selective repeat hybrid system. Figure 5b is a graph that shows which of the constituent codes is used by the adaptive code at each of the different fading environments (at each of the values of ***m*** and ***s***.) Note that the step size of ***m*** is decreased to .5dB and the range of ***s*** is extended to 5.

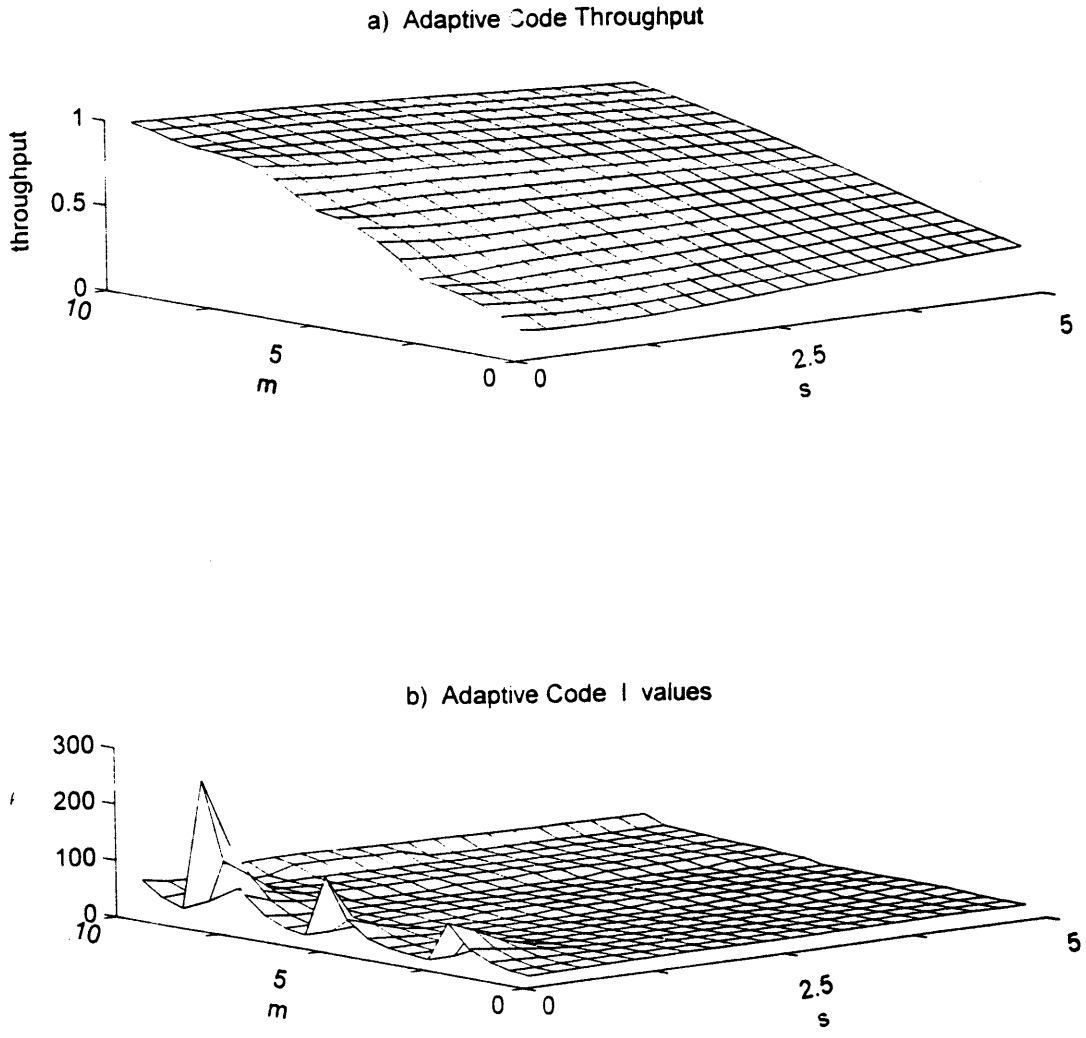


FIGURE 6: The throughputs that are achieved with our adaptive FEC code and selective repeat hybrid system are shown in figure 6a. Figure 6b is a graph showing the optimal values of l at each of the different fading environments. Notice the spikes in this graph where the noise PDF is sharp (s is low) and is centered on an E_b/N_0 value that is near a transition point of constituent codes in the AFEC.

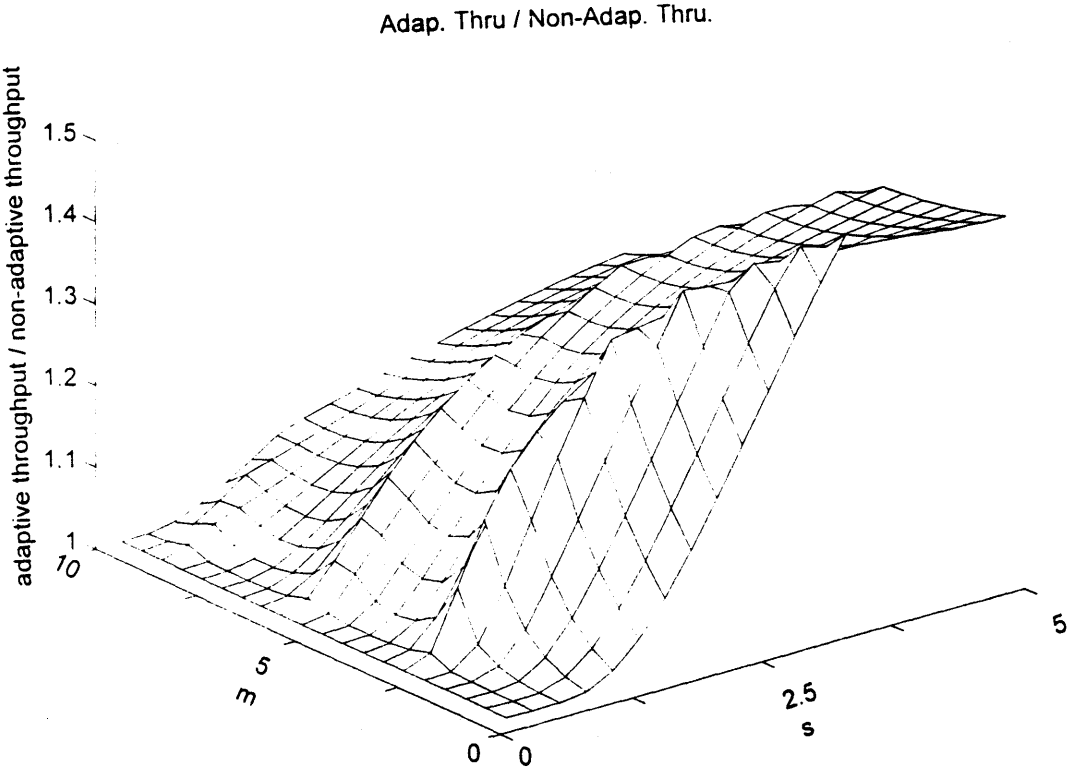


FIGURE 7: This graph shows the ratio of the optimal non-adaptive hybrid system throughput in figure 5a to the adaptive hybrid system throughput in figure 6a. This graph represents how much better an adaptive hybrid code works in different fading environments. Notice that the benefit of using an adaptive hybrid system increases as the standard deviation of the fading environment increases.

To study the effects of propagation delay in the link, we focus on a specific fading model. We choose the fading model with mean μ = 5dB and standard deviation σ = 2. The optimal value of ℓ in this case is 18. Figure 8 shows how different values of ℓ affect the throughput of this system. Figure 9 shows the performance curves at μ=5dB, σ=2 and ℓ=18 for the constituent codes that make the AFEC. The performance of the AFEC is the maximum of these curves. Also shown in this figure is the Gaussian PDF curve of the fading. This curve is not plotted on the same vertical scale as the throughput performance curves. It is included in the figure to give a sense of how often each coderate is used. Notice that the curves are cut off at E_b/N_0=0dB. At zero E_b/N_0 and below, it is assumed that the modem loses its lock on the signal and thus the throughput is zero.

To explore how a delay in the estimated BER of the noise in the channel affects the overall throughput of the system, a fading model for E_b/N_0 with respect to time must be determined. This delay can be caused by the propagation delay while the estimated coderate information is sent from the receiver to its transmitter. The model used in this thesis is a Markov Chain model with a Gaussian probability density function. The pertinent Markov Chain equation is: $P_{j+1} = (\lambda_j/\mu_{j+1})P_j$. The probability of being in state j, P_j, is determined by the Gaussian function. The probability of staying in the current state is some fixed probability for all states. In this thesis, the probability of staying in any state is 0.9. This is all the information needed to determine values for λ and μ. Appendix 4 has a program that used these Markov chain calculations. Notice that a chain is created which has a PDF of only the right side of the Gaussian function. At state 0, a coin is flipped to determine whether the states that the chain visits should be interpreted as left or right of the center of the Gaussian function. This coin is re-flipped each time state 0 in entered. This trick creates a Markov Chain with a full Gaussian PDF. Each time the chain makes a decision to enter a new state or stay in the current state, an iteration occurs. The chain is allowed to run for various numbers of iterations to determine how many iterations it takes for the PDF to look like a Gaussian function. See figures 10-12.

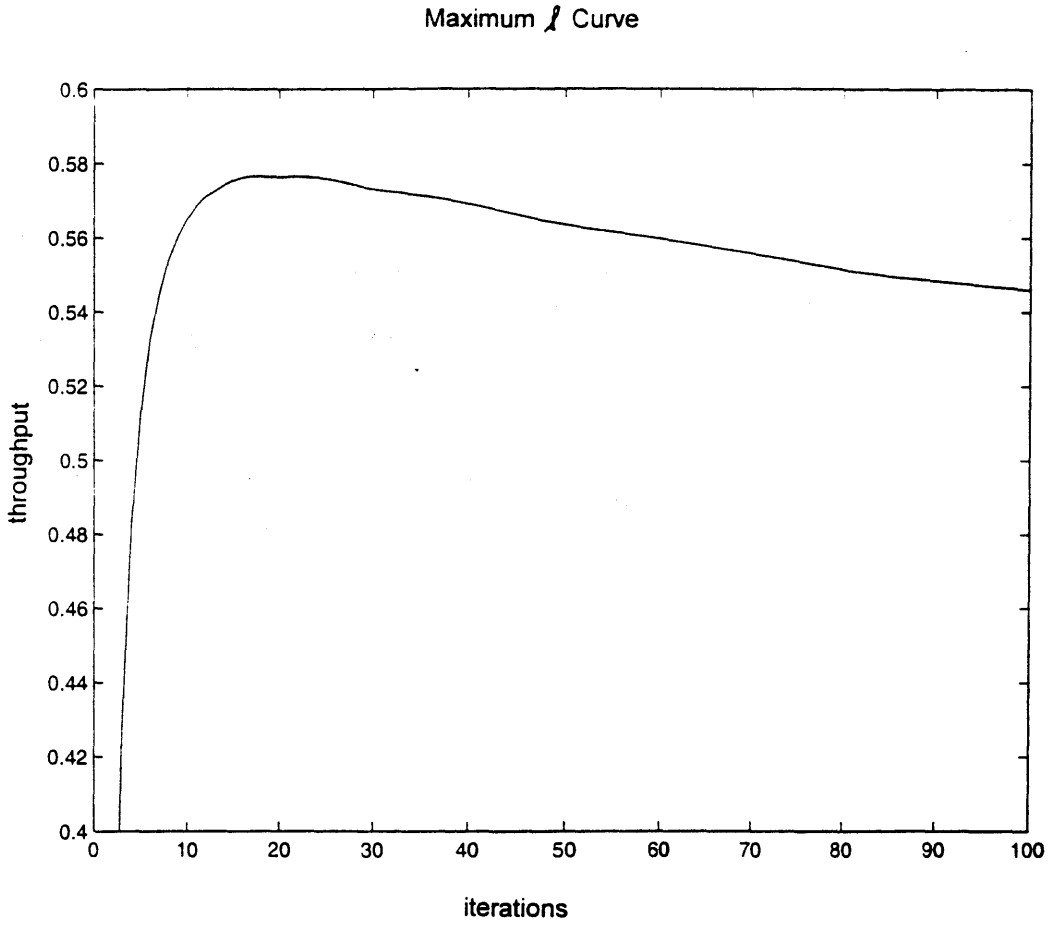


FIGURE 8: This graph shows the throughput performance of an adaptive FEC selective repeat ARQ hybrid system with the specific fading characteristics of m = 5dB and s = 2 for different values of ℓ. Throughput is maximized for ℓ = 18. Therefore, the packet size of our system is 64*18 = 1152.

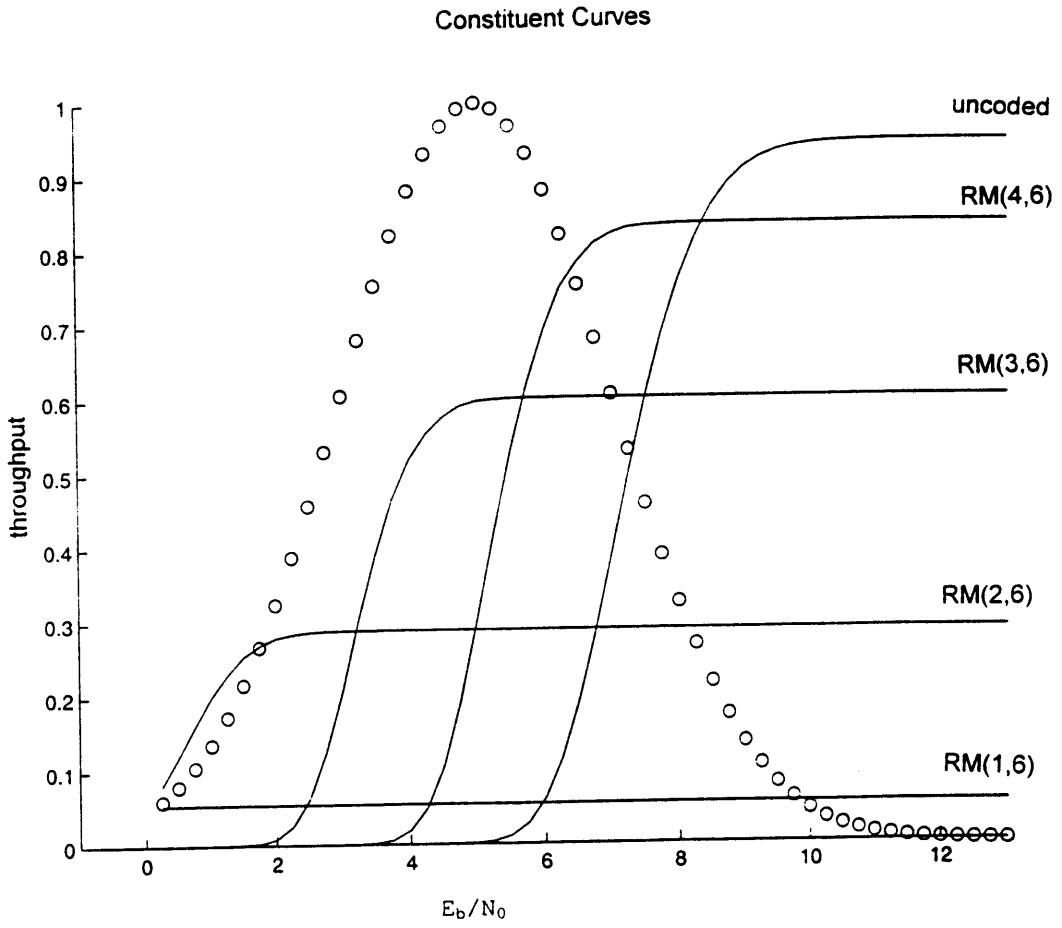
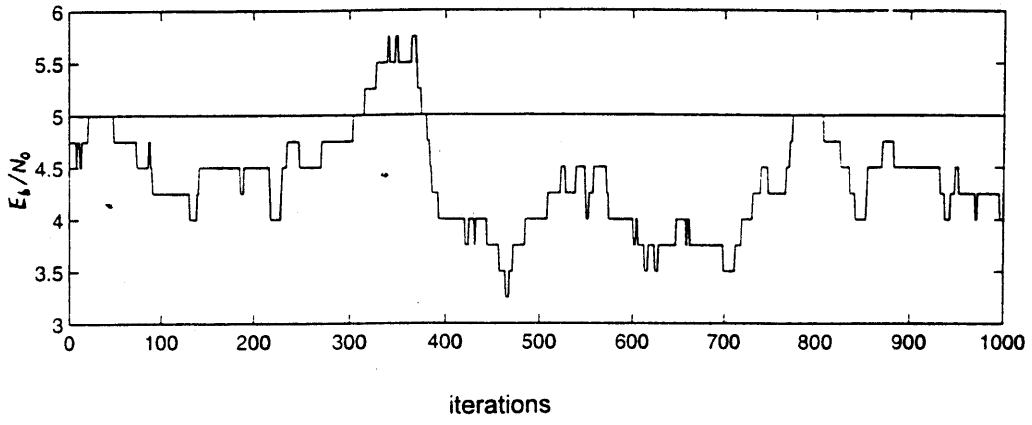


FIGURE 9: This graph shows the performance curves of the constituent codes that make up the AFEC at ℓ=18. The performance of the AFEC is simply the maximum of these curves at each E_b/N_0 value. The curve drawn in the circled line is the PDF of the fading at mean m = 5dB and standard deviation s = 2. This PDF curve is not drawn to scale vertically.

a) Fade



b) Prob. Density

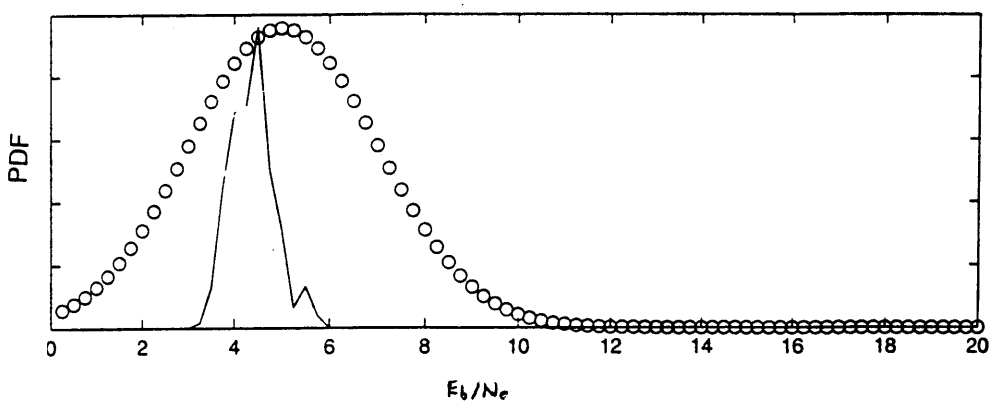
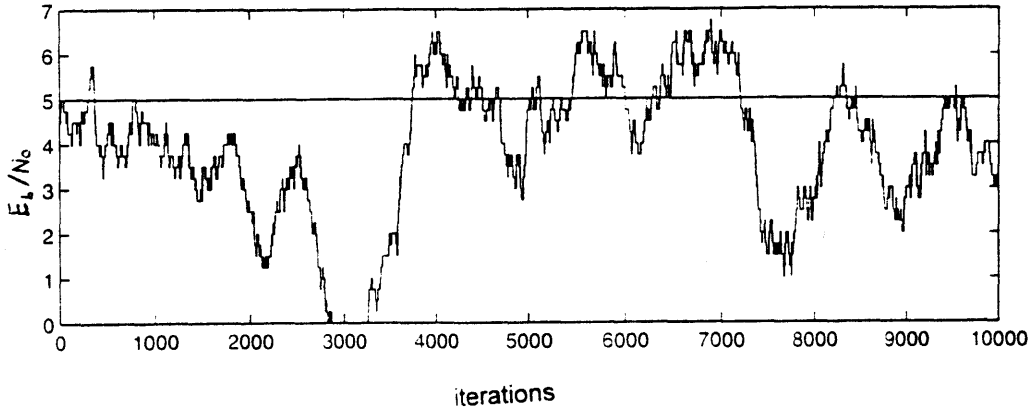


FIGURE 10: The output of the Markov Chain run for 1000 iterations modeling a fade with m=5dB and s=2 is shown in figure 10a. Figure 10b is the calculated PDF for this chain. Because it does not look like the expected Gaussian curve, more iterations should be run.

a) Fade



b) Prob. Density

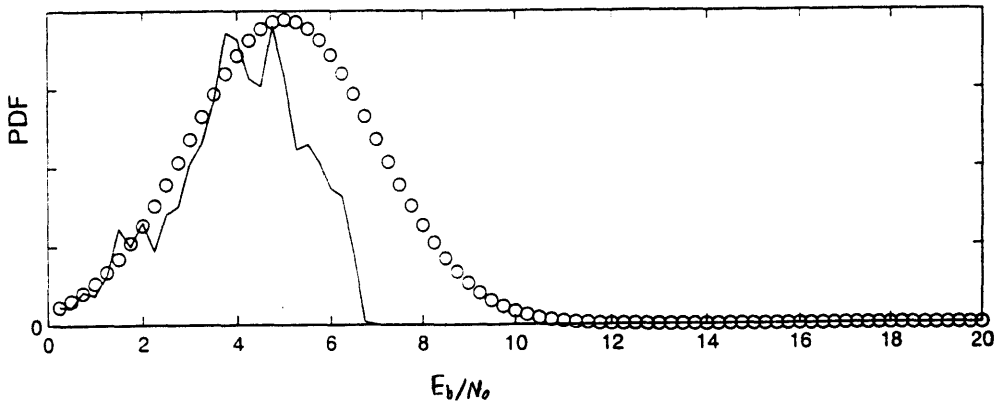
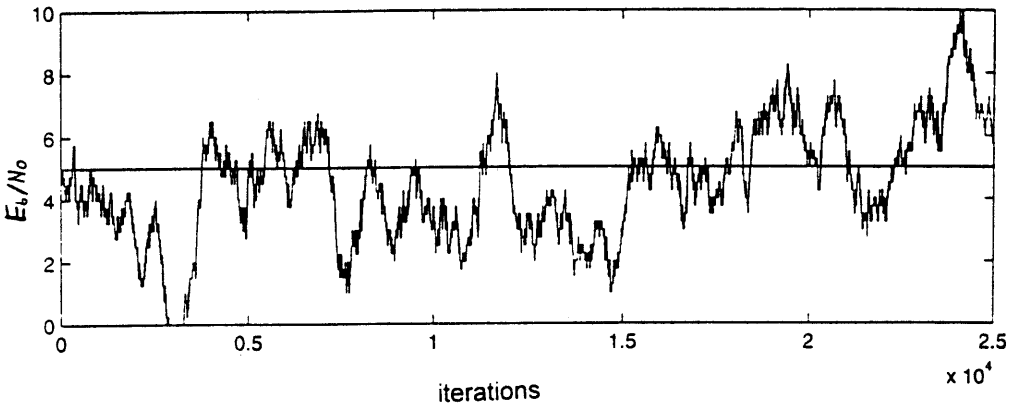


FIGURE 11: The output of the Markov Chain run for 10,000 iterations modeling a fade with m=5dB and s=2 is shown in figure 11a. The PDF curve in figure 11b looks more like the Gaussian curve. However, more iterations should be run.

a) Fade



b) Prob. Density

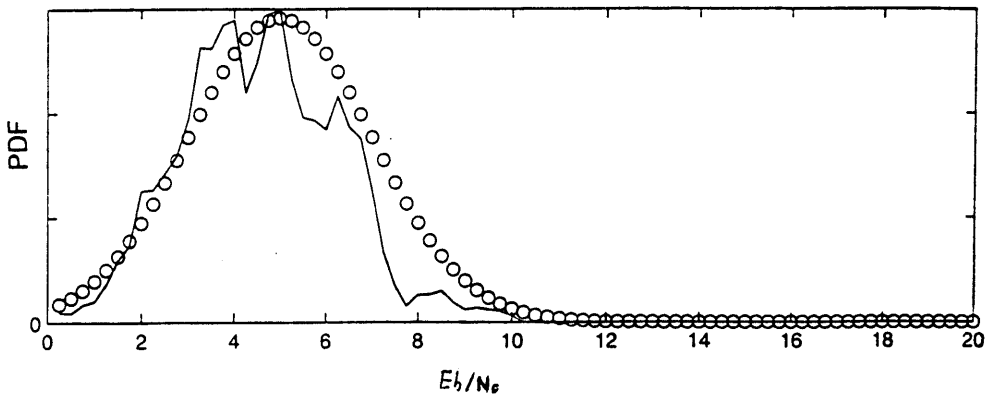


FIGURE 12. The output of the Markov Chain run for 25,000 iterations modeling a fade with m=5dB and s=2 is shown in figure 12a. The PDF curve in figure 12b looks more like the Gaussian curve.

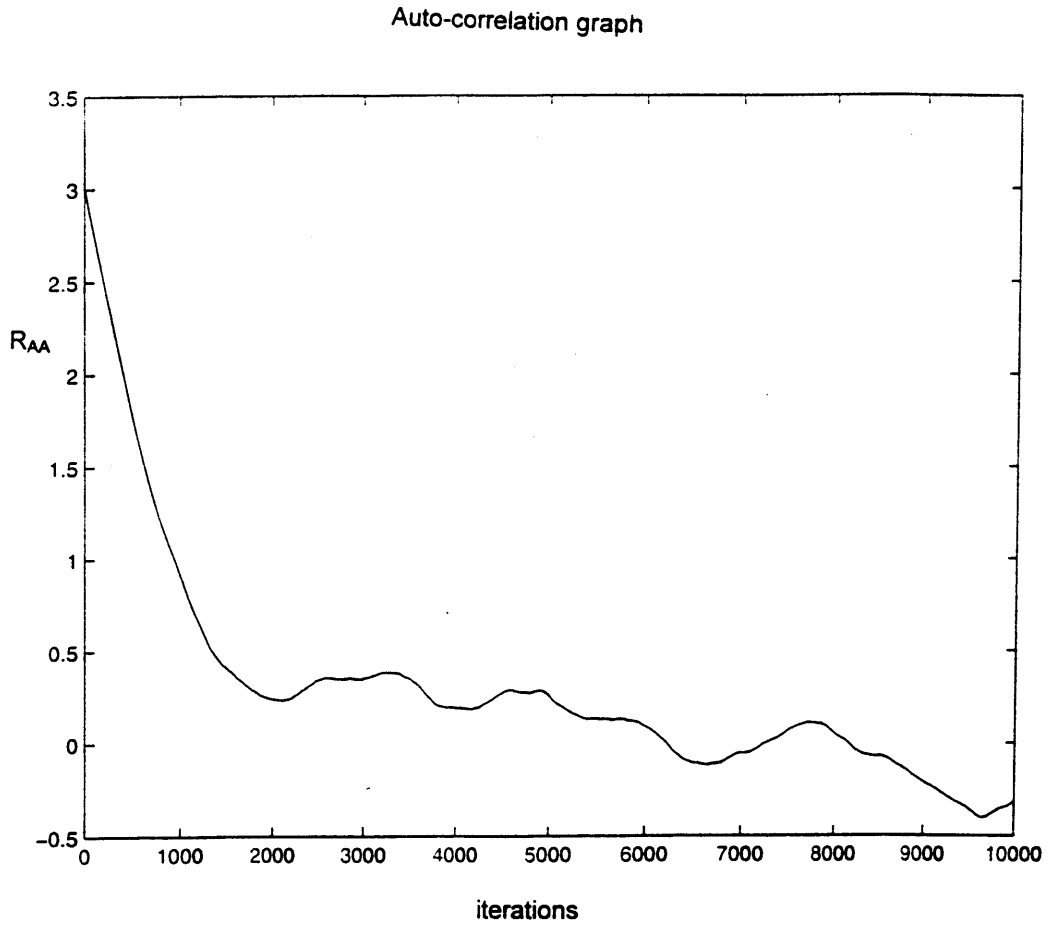


FIGURE 13: This graph shows the auto-correlation $R_{AA} = E[A(t) \cdot A(t+\tau)]$ of the output of the Markov Chain run for 100,000 iterations. The expected value is taken by averaging over 90,000 iterations. τ ranges from 0 to 10,000 iterations. The output becomes uncorrelated with itself at about 2000 iterations.

At this point in the thesis, an absolute time scale for the iterations of the Markov Chain model is set using an auto-correlation function, $R_A(\tau) = E[A(t+\tau)A(t)]$. See Appendix 4. This function determines how well the E_b/N_0 at one point in time is correlated with the E_b/N_0 τ iterations later. The results are shown in figure 13. The noise becomes uncorrelated with itself at approximately 2000 iterations. This cutoff will be defined as 1 second to approximate the affects of scintillation on Ka band signals. It is assumed that fading due to rain attenuation is handled by power control or some other means. Thus, 2000 iterations of a Markov Chain corresponds to 1 second of time.

The affects of delay and the different protocols to relay channel E_b/N_0 information from the receiver to the transmitter can now be simulated. MATLAB code is run to simulate the affects of delay on throughput. See Appendix 5. The choice of running this simulation for 1 million iterations or 500 seconds is determined by running simulations with zero delay for varying amounts of time and seeing how close the simulations' throughputs come to the calculated throughputs of the program in Appendix 3. The results are listed in table 4.1.

#iteratons	throughput	#iteratons	throughput
100K	.5868	850K	.5927
400K	.6099	1000K	.5871
600K	.5890	1500K	.5837

Table 4.1: This table shows the number of iterations for which the Markov Chain fading model is run and the associated system throughput calculated. m = 5dB, s = 2, ℓ = 18.

From this table, it looks as though 1000K iterations is where the throughput starts to stabilize near the .5766 calculated throughput target. The calculated PDF for 1 million iterations also looks Gaussian. The program in Appendix 5 is run for several different delays and the results are listed in table 4.2. This information is graphed in figure 14. These throughputs do not take into account the BER information that gets passed over the link from the receiver to its transmitter. This plot represents the baseline curve because any protocol for transferring BER information will not be able to perform better than the throughputs in this curve. Also shown on this graph is another simulation that only differs in the number of overhead bits, which is changed from 64 to 72. The extra 8

bits of information represent the extra space in the header of each packet that would be used to relay coderate information from the receiver back to the transmitter in the return link. Because the link is bi-directional, the extra overhead will be added on packets in both directions.

To simulate the protocol in which control packets are used to relay coderate information as opposed to an increased header, the data rate for the link must be known. This simulation is done for data rates of 1 Gbps, 10 Mbps, and 2 Mbps. Because a packet is roughly 1000 bits $(\ell=18)\cdot(n=64)$, 1 Gbps represents 1 million packets per second or 500 packets per iteration. Thus, each iteration represents the passing of 500 packets while sending a control packet only costs 1 packet worth of information. Following the same reasoning, the 10 Mbps data rate leads to 5 packets per iteration and the 2 Mbps leads to 1 packet per iteration. The throughputs for these links are shown in table 4.2.

		header	control packet protocol		
	baseline	protocol	1 Gbps	10 Mbps	2 Mbps
delay	thruput	thruput	thruput	thruput	thruput
0	.5871	.5809	.5871	.5806	.5817
25	.5788	.5726	.5788	.5777	.5735
50	.5719	.5658	.5719	.5709	.5667
75	.5653	.5593	.5653	.5643	.5602
100	.5592	.5532	.5592	.5581	.5541
200	.5387	.5330	.5387	.5378	.5338
300	.5246	.5190	.5246	.5237	.5198
400	.5127	.5072	.5126	.5117	.5080
500	.5029	.4975	.5029	.5020	.4983
1000	.4772	.4721	.4772	.4763	.4729
1500	.4639	.4590	.4639	.4631	.4597
2000	.4547	.4498	.4546	.4538	.4505
2500	.4513	.4465	.4513	.4504	.4472
3000	.4486	.4438	.4586	.4478	.4445
3500	.4493	.4445	.4493	.4485	.4452
4000	.4487	.4439	.4586	.4487	.4445

Table 4.2: This table shows the throughput of the baseline curve in column 2 with respect to the delay in iterations of the Markov Chain. This column does not use any protocol to get coderate information back to the transmitter. Column 3 uses an extra 8 bits in the header to relay coderate information. The last three columns use control packets to relay coderate information with system data rates of 1 Gbps, 10 Mbps, and 2 Mbps respectively. For all these columns, m = 5dB, s = 2, and ℓ = 18.

The performance plot of the control packet protocol at 1Gbps is indistinguishable from the plot for the baseline delay curve (figure 14). At 10 Mbps, the performance plot of the this protocol is worse than the baseline curve, but still better than the plot of the curve which represents passing coderate information in each packet header (figure 15). At 2 Mbps, the performance of this control packet protocol is worse than the performance of the header protocol (figure 16). However, all these curves are close to the baseline curve. Minimizing delay is more important than the protocol used to relay coderate information from the receiver back to the transmitter. This delay is approximately twice the round trip propagation delay of LEO satellite communications, and therefore is on the order of .025 seconds or 50 iterations.

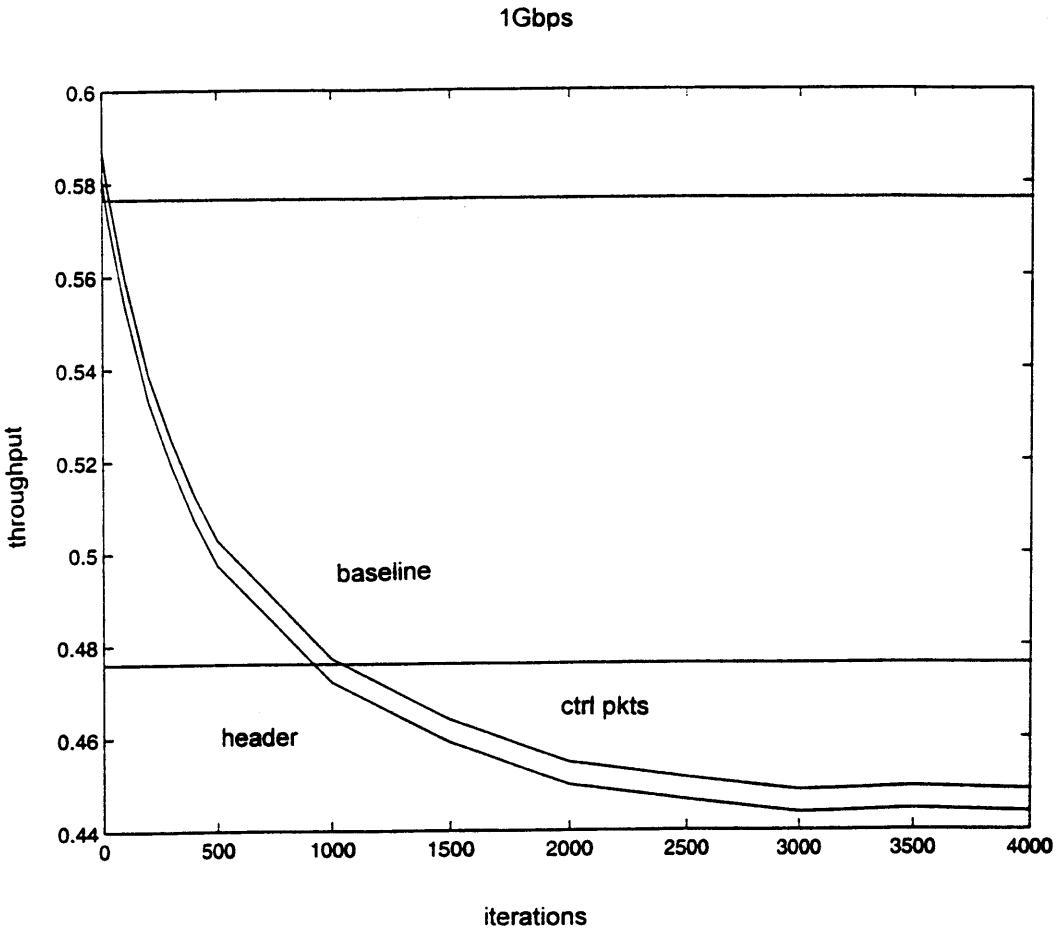


FIGURE 14: This graph shows how delay of the channel noise information affects throughput for a system at bit rate 1Gbps with a fading environment of m=5dB and s=2. The upper curve is the baseline curve and also shows the throughput of the system using the control packet protocol. The second lower curve shows the throughput of the system using the header protocol. Also shown is the throughput of the calculated adaptive hybrid system at zero delay and the throughput of the non-adaptive hybrid system at zero delay. From these zero delay lines, we see that the throughput performance of the adaptive system degrates to that of the best non-adaptive system if the channel information is delayed for about 1000 iterations or 1 second.

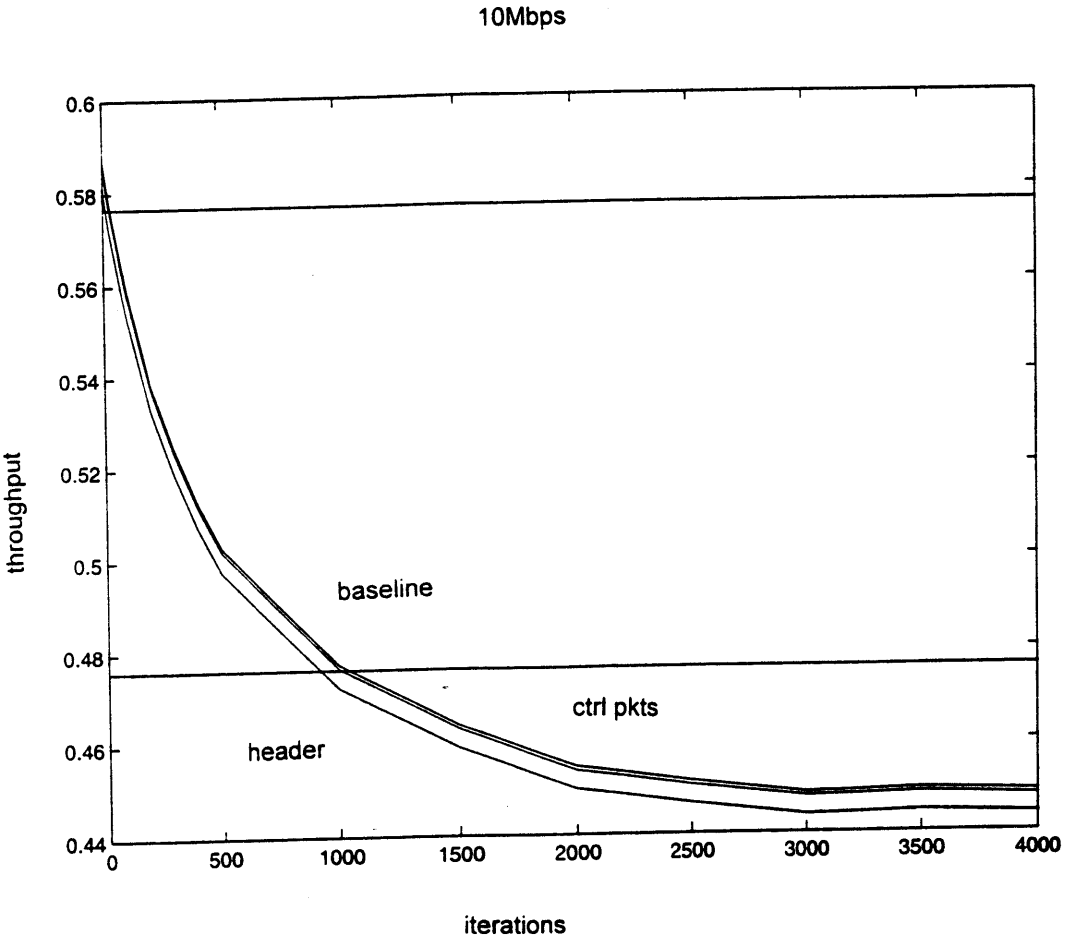


FIGURE 15: This graph shows how delay of the channel noise information affects throughput for a system at bit rate 10Mbps with a fading environment of m=5dB and s=2. The baseline curve (upper curve) and the throughput curve of the system using the header protocol (lower curve) do not change. The throughput curve of the system using the control packet protocol (middle curve) starts to move lower. However, this system still performs better than the system that uses the header protocol.

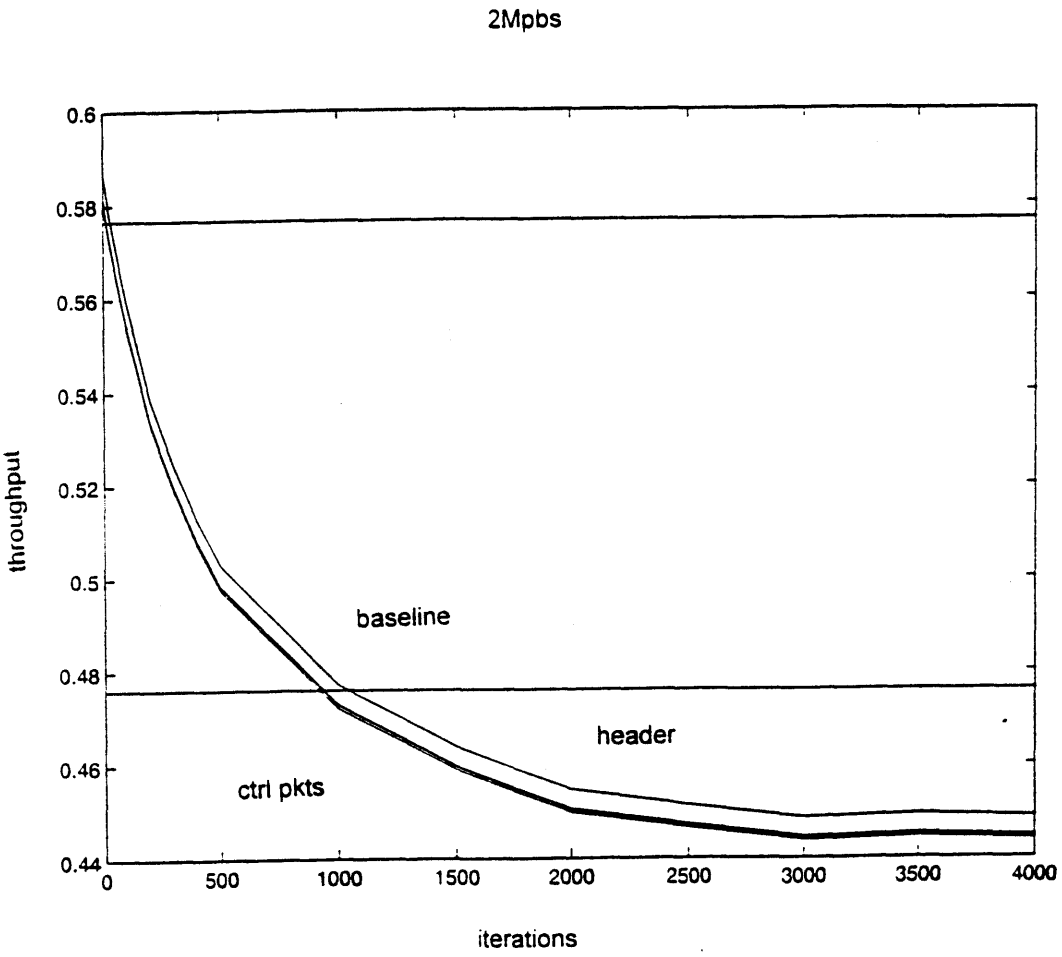


FIGURE 16: This graph shows the same throughput performance curves when the systems are operating at 2Mbps. The curve of the system that uses control packets has now dropped below the curve of the system that uses the header protocol.

[This page is intentionally left blank]

5.0 CONCLUSION

When designing an ARQ/AFEC system, many choices of codes are available to construct the AFEC. If RM codes are chosen, then erasure decoding and/or simple 2-D iterative block decoding do not buy enough throughput performance to make them worth their added complexity. One reason for this may be due to the fact that in an AFEC, codes are run on the edges of their performance curves. The range of E_b/N_0 values for which a code is used starts near to where that code's throughput performance falls off to zero. This most likely explains why a delay in updating coderate information in our AFEC has such a dramatic negative effect on throughput. If fading causes the E_b/N_0 to decrease, then the throughput of the system is falling to zero for a period of time until the new code is selected.

The ARQ/AFEC system under the two protocols and log-normal fading was analyzed for throughput. First a comparison of the ARQ/AFEC with an ARQ/Non-Adaptive FEC was made to determine whether a system designer should bother with an adaptive code. Adaptive codes become more worth while as the standard deviation of the fading environment increases. With communications over a LEO satellite, which has a round trip delay on the order of .013 seconds, and a fading model with a mean of 5dB E_b/N_0 and a standard deviation of 2, an ARQ/AFEC system makes sense. With this fading model, the better protocol for sending channel information from the receiver to its transmitter depends on the data rate. For data rates higher than 10 Mbps, the protocol that uses control packets is more throughput efficient. For data rates lower than 2 Mbps the protocol that places coderate information in the header is more throughput efficient. For data rates in between, the choice of protocols does not seem to affect throughput significantly.

These observations introduce two interesting extensions to this thesis. First, the erasure decoding and 2-D iterative decoding can be tested for throughput performance at much higher E_b/N_0 values that have error probabilities on the order of 10^{-6} and 10^{-9}. This is the usual region in which non-adaptive codes are used and analyzed. Our ARQ/AFEC does not run codes in this region because throughput is not maximized by doing so. Better throughput is achieved by switching to a code with a higher coderate and letting the ARQ system handle much of the

error control. It would be interesting to see if the erasure decoding and iterative decoding methods greatly outperform the normal RM decoding at these higher E_b/N_0 regions. If they do, one conclusion that may be drawn is that when constructing an ARQ/AFEC system, a designer should not bother with complex codes that approach the Shannon limit. Simple codes like normal RM codes may be used with minimal loss in throughput performance.

The second interesting extension to this thesis is to use margin when switching from one code to the next code so that the throughput of the system does not fall to zero as quickly during the delay in switching coderates. A range of margins can be analyzed to find the margin that produces the highest throughput.

APPENDIX 1: BACKGROUND

To understand the error correcting codes used in this thesis, binary fields are first explained. Next, vector spaces that are defined over these fields are introduced. With this knowledge, linear FEC codes can be explained. The Reed-Muller code, a specific binary linear code, is then introduced in detail because this is the code that is focused upon in this thesis. After this code is explained, bit error rate performance is explained. This section is then followed by short introductions to erasure and iterative decoding. Finally, the selective repeat ARQ scheme is introduced.

[This page is intentionally left blank]

1a) Binary Fields

A binary field, GF(2), is a set of two elements commonly labeled 0 and 1 with 2 operations. This is the simplest type of field. The two operations, addition and multiplication, are under modulo 2 arithmetic. To understand fields, abelian groups must first be understood. An abelian group, G, is a set of objects and an operation "*" which satisfy the following:

1. Closure: if a,b in G, then a*b = c in G.
2. Associativity: (a*b)*c = a*(b*c) for all a,b,c in G.
3. Identity: there exists i in G such that a*i = a for all a in G.
4. Inverse: for all a in G, there exists a^{-1} in G such that $a*a^{-1} = i$.
5. Commutativity: a*b = b*a for all a,b in G.

Examples of abelian groups are the set of integers and the set of real numbers. An example of a finite group is the set of integers from 0 to m-1 with the operation of addition modulo m. For example, the group {0,1,2,3} is a group under modulo 4 addition. This table defines the group:

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

All the rules above are met. For example, 0 is the identity element, the inverse of 3 is 1, 2+1 = 1+2, etc. Note that this same set of elements under modulo 4 multiplication would also meet these rules and would also be an abelian group.

A field, F, is a set of objects with two operations, addition (+) and multiplication (*), which satisfy:

1. F forms an abelian group under + with identity element 0.
2. F-{0} forms an abelian group under * with identity element 1.
3. The operations + and * distribute: a*(b+c) = (a*b)+(a*c).

The example above of {0,1,2,3} is not a field under addition and multiplication modulo 4 because 2*2 = 0 which is not in F-{0}. F-{0} is not closed and is not a group under *. Notice that the set of integers {0,1,2 … (p-1)} under addition and multiplication modulo p where p is a prime number does not fall into this trap. This set under these two modulo p operations does define a field. The simplest field is the set {0,1} under addition and multiplication modulo 2. The tables below define the field.

+	0	1
0	0	1
1	1	0

*	0	1
0	0	0
1	0	1

1b) Vector Spaces

Letting V be a set of elements called vectors and F be a field of elements called scalors, the operations vector addition "+" and scalor multiplication "*" can be introduced. V is said to form a vector space over F if:

1. V forms an abelian group under +.
2. For any a in F and $\mathbf{v}$ in V, $a*\mathbf{v} = \mathbf{u}$ in V.
3. The operations + and * distribute: $a*(\mathbf{u}+\mathbf{v}) = a*\mathbf{u} + a*\mathbf{v}$ and $(a+b)*\mathbf{v} = a*\mathbf{v} + b*\mathbf{v}$. (Here (a+b) is the additive field operation and not the additive vector operation. These operations can be distinguished because vectors are in bold and scalors are not.)
4. Associativity: for all a,b in F and $\mathbf{v}$ in V, $(a*b)*\mathbf{v} = a*(b*\mathbf{v})$.
5. The multiplicative identity 1 in F is also the multiplicative identity in scalor multiplication: for all $\mathbf{v}$ in V, $1*\mathbf{v} = \mathbf{v}$.

The simplest example of a vector space is the set of binary n-tuples. Here, vector addition is defined as element wise modulo 2 addition and multiplication is normal scalor multiplication.

$$\mathbf{u}+\mathbf{v} = (u_0+v_0,\ u_1+v_1,\ \ldots\ ,u_{n-1}+v_{n-1})$$
$$a*\mathbf{v} = (a\mathbf{v}_0,\ a\mathbf{v}_1,\ \ldots\ ,a\mathbf{v}_{n-1})$$

For example, (1,1,0,0,0,1) + (1,0,0,1,0,0) = (0,1,0,1,0,1) and 0 * (1,1,0,0,0,1) is (0,0,0,0,0,0). A spanning set is a set of vectors such that the linear combination of these vectors creates all the vectors in the vector space. A basis is a spanning set of minimal cardinality. For example, if $\mathbf{v}_0$=(1,0,0,0), $\mathbf{v}_1$=(0,1,0,0), $\mathbf{v}_2$=(0,0,1,0) and $\mathbf{v}_3$=(0,0,0,1), then the set $\{\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2,\mathbf{v}_3\}$ is a basis for the set of binary 4-tuples because any binary 4-tuple can be expressed as a linear combination of these vectors. For example, (0,0,1,1) = $0*\mathbf{v}_0 + 0*\mathbf{v}_1 + 1*\mathbf{v}_2 + 1*\mathbf{v}_3$. The dimension of a vector space equals the cardinality of its basis. A subspace is a subset of V which also satisfies the properties of a vector space. The inner product is defined as:

$$\mathbf{u}\cdot\mathbf{v} = u_0*v_0 + u_1+v_1 + u_2*v_2 + \ldots + u_{n-1}*v_{n-1}$$

Let S be a k-dimensional subspace of vector space V, and let S^D be the set of all $\mathbf{v}$ in V such that for all $\mathbf{u}$ in S and

for all $\mathbf{v}$ in S^D, $\mathbf{u}\cdot\mathbf{v} = 0$. Subspace S^D is the dual space of subspace S. Note that S and S^D are disjoint subspaces in V and thus the dimension of S^D is:

$$\dim(S^D) = \dim(V) - \dim(S).$$

Finally, note that vector subtraction is the same as vector addition in binary vector spaces because of the group property of V under vector addition. Every vector is its own inverse and thus $-\mathbf{v} = \mathbf{v}$ because $\mathbf{v}+\mathbf{v} = 1*\mathbf{v} + 1*\mathbf{v} = (1+1)\mathbf{v} = 0*\mathbf{v} = 0$ (the zero vector). Thus, $\mathbf{v}+\mathbf{v} = \mathbf{v}+(-\mathbf{v}) = \mathbf{v}-\mathbf{v}$.

1c) Linear Block Codes

A block code **C** is a set of m codewords $\{\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{m-1}\}$ where each codeword is an n-tuple of elements from a finite field, $\mathbf{c} = \{c_0, c_1, \ldots, c_{n-1}\}$. If these elements are from the finite field GF(q), then code **C** is q-ary. For binary block codes, the elements of **c** are form {0,1}, i.e. from GF(2). Block codes work in the following way. A data stream is broken up into message blocks of length k bits. Each block is encoded into an n bit codeword from **C** and transmitted. The receiver may then correctly decode the n bit block back into the original k bit message block, even if some of the bits in the codeword are corrupted by noise during transmission. This decoding is possible because r = n-k redundant bits have been added to the codeword. The rate of a code is R = k/n. Thus, using a half rate code requires the system to transmit twice as many bits as an uncoded system.

There are several definitions used to describe codes and their codewords. The weight of a codeword is the number of non-zero elements in that codeword. In the case of binary codes, the weight is the number of 1 coordinates. The Hamming distance between two codewords is the number of coordinates in which the two codewords differ. For example, the Hamming distance between (0,1,1,1,1) and (0,1,0,0,1) is 2. The minimum distance, d_{min}, of a code is the minimum Hamming distance between any of its codewords. An error pattern is an n-tuple of 0s and 1s in which 1s are placed in the coordinate positions of corrupted bits and 0s fill the rest of the n-tuple. For example, if (0,1,1,1,1) is transmitted and (0,1,0,0,1) is received due to noise, the error pattern is (0,0,1,1,0). Notice that the receiver can detect all errors in a received word whose error pattern weights are $<= d_{min}-1$ because a corrupted codeword could not be disguised as another valid codeword. Also, the receiver can correct all errors in a received word whose error pattern weights are $<= (d_{min}-1)/2$ because the received word will be closest in Hamming distance to the correct codeword. In this case, the receiver would simply replace the received word with this closest codeword and continue decoding.

A linear q-ary block code **C** is a specific type of q-ary block code in which **C** forms a vector subspace over GF(q). In the case of binary linear block codes, q = 2. The dimension of a linear code is the dimension of the

subspace which equals k, the number of coordinates in the message block for that code. Thus, there are 2^k codewords in the code. Properties of linear block codes include:

1. Any linear combination of codewords is also a codeword.
2. The minimum distance of the code is the weight of a lowest weight non-zero codeword.

Both properties follow directly from the group property of vector addition. The second property can be understood as follows: let $\mathbf{c}$ and $\mathbf{c}'$ be codewords with the minimum Hamming distance. $\mathbf{c} = \mathbf{c}-\mathbf{c}'$ is a codeword which has this same minimum Hamming distance from the all 0 codeword. $\mathbf{c}$ is also a lowest weight non-zero codeword. Note that the all 0 codeword must be included in any linear code because 0 is always a valid element over any field which the vector space can be formed over and $0*\mathbf{c}$ = the all 0 codeword.

A generator matrix $\mathbf{G}$ for a linear block code can be constructed from a basis $\{\mathbf{g}_0, \mathbf{g}_1, \ldots, \mathbf{g}_{k-1}\}$ of its vector space.

$$\mathbf{G} = \begin{vmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \ldots \\ \mathbf{g}_{k-1} \end{vmatrix} = \begin{vmatrix} g_{0,0} & \ldots & g_{0,n-1} \\ g_{1,0} & & \\ \ldots & & \ldots \\ g_{k-1,n} & \ldots & g_{k-1,n-1} \end{vmatrix}$$

The k bit message block is then encoded into the n bit codeword as follows: $\mathbf{c} = \mathbf{mG} = m_0\mathbf{g}_0 + m_1\mathbf{g}_1 + \ldots + m_{k-1}\mathbf{g}_{k-1}$. Note that any basis of the vector space can be used to construct this generator. The resulting codes will have the same performance and codewords, but the messages may be represented by different codewords in the code. A parity check matrix $\mathbf{H}$ for a code $\mathbf{C}$ is constructed using the basis $\{\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_{n-k-1}\}$ of the dual of $\mathbf{C}$'s vector space $\mathbf{C}^D$.

$$\mathbf{H} = \begin{vmatrix} \mathbf{h}_0 \\ \mathbf{h}_1 \\ \ldots \\ \mathbf{h}_{k-1} \end{vmatrix} = \begin{vmatrix} h_{0,0} & \ldots & h_{0,n-1} \\ h_{1,0} & & \\ \ldots & & \ldots \\ h_{n-k-1,n} & \ldots & h_{n-k-1,n-1} \end{vmatrix}$$

If a codeword is multiplied by its parity check matrix, the result is the all 0 n-tuple, $\mathbf{cH}^T = 0$. This follows directly from the property of dual spaces. Finally, a syndrome is calculated by multiplying the received word $\mathbf{r}$ by $\mathbf{H}^T$. Note that $\mathbf{s} = \mathbf{rH}^T = (\mathbf{c}+\mathbf{e})\mathbf{H}^T = \mathbf{cH}^T + \mathbf{eH}^T = 0 + \mathbf{eH}^T = \mathbf{eH}^T$. For many decoders, the syndrome is used to look up the error pattern $\mathbf{e}$ which can be used to correct the errors in $\mathbf{r}$.

1d) Reed-Muller Codes

Reed_Muller (RM) codes are considered "quite good" in terms of performance vs. complexity. [5] They are not the most powerful codes. However, they have an "extremely fast maximum likelihood decoding algorithm." [12] To understand RM codes boolean functions are first introduced. An example of boolean functions of four variables is shown in the following truth table.

v_4 =	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
v_3 =	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
v_2 =	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
v_1 =	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---
f_1 =	0	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1
f_2 =	1	0	1	0	1	0	1	1	1	0	1	0	0	1	0	0

In this example, $f_1 = v_3+v_2+v_1$ and $f_2 = v_3v_2v_1+v_4v_3+v_1+1$. These boolean functions and variables can be associated with vectors. For example, $\mathbf{f}_1 = (0,1,1,0,\ldots,1)$. Notice that the matrix $[\mathbf{v}_4,\mathbf{v}_3,\mathbf{v}_2,\mathbf{v}_1]^T$ is arranged such that the columns are the binary numbers from 0 to 2^4-1. Any boolean function can be represented as:

$$\mathbf{f} = a_0\mathbf{1} + a_1\mathbf{v}_1 + \ldots + a_m\mathbf{v}_m + a_{12}\mathbf{v}_1\mathbf{v}_2 + \ldots + a_{12\ldots m}\mathbf{v}_1\mathbf{v}_2\ldots\mathbf{v}_m$$

where m represents the number of variables that the functions are defined over. Note that the associated vectors are binary 2^m-tuples. There are 2^{2^m} distinct vectors/functions.

The RM code R(r,m) is the set of all 2^m-tuples that is associated with boolean functions in m variables of order r, i.e. boolean function that are polynomials of degree r or less. For example, R(2,4) has the generator matrix:

$$
\mathbf{G} = \begin{vmatrix} \mathbf{1} \\ \mathbf{v}_4 \\ \mathbf{v}_3 \\ \mathbf{v}_2 \\ \mathbf{v}_1 \\ \mathbf{v}_3\mathbf{v}_4 \\ \mathbf{v}_2\mathbf{v}_4 \\ \mathbf{v}_1\mathbf{v}_4 \\ \mathbf{v}_2\mathbf{v}_3 \\ \mathbf{v}_1\mathbf{v}_3 \\ \mathbf{v}_1\mathbf{v}_2 \end{vmatrix} = \begin{vmatrix} 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1 \\ 0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1 \\ 0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1 \\ 0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&0&0&1&1&0&0&1&1 \\ 0&0&0&0&0&0&0&0&0&1&0&1&0&1&0&1 \\ 0&0&0&0&0&0&1&1&0&0&0&0&0&0&1&1 \\ 0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1 \\ 0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1 \end{vmatrix}
$$

Note that to create the R(3,4) generator matrix, this matrix would be extended down to by adding the rows for $\mathbf{v}_2\mathbf{v}_3\mathbf{v}_4$, $\mathbf{v}_1\mathbf{v}_3\mathbf{v}_4$, $\mathbf{v}_1\mathbf{v}_2\mathbf{v}_4$, etc. The matrix above can be more compactly expressed using sub-matrices where $\mathbf{G}_0 = \mathbf{1}$, $\mathbf{G}_1 = [\mathbf{v}_4,\mathbf{v}_3,\mathbf{v}_2,\mathbf{v}_1]^T$, and $\mathbf{G}_2 = [\mathbf{v}_3\mathbf{v}_4,\mathbf{v}_2\mathbf{v}_4, \ldots ,\mathbf{v}_1\mathbf{v}_2]^T$. Note that the indices indicate the order of the collection of rows. The message vector can also be expressed in a similar way: $\mathbf{m}_0 = m_0$, $\mathbf{m}_1 = [m_4,m_3,m_2,m_1]$, and $\mathbf{m}_2 = [m_{34},m_{24},\ldots,m_{12}]$. To encode a message, into a codeword:

$$
\mathbf{c} = \mathbf{mG} = [\ \mathbf{m}_0 \mid \mathbf{m}_1 \mid \mathbf{m}_2\ ] \begin{vmatrix} \mathbf{G}_0 \\ -- \\ \mathbf{G}_1 \\ -- \\ \mathbf{G}_2 \end{vmatrix}
$$

So for an RM(r,m) code, $n = 2^m$ and $k = \text{SUM Comb}(m,j)$ where the summation variable j goes from 0 to r inclusive and Comb represents m things taken j at a time. For example, k of RM(2,6) is Comb(6,0)+Comb(6,1)+Comb(6,2) = 1+6+15 = 22.

Decoding an RM code back into a message $\mathbf{m}$ is done by estimating $\mathbf{m}_x$ from the highest order to $\mathbf{m}_0$. For our RM(2,4) example, this means estimating $\mathbf{m}_2$, then $\mathbf{m}_1$, then $\mathbf{m}_0$. Notice that for RM(2,4), $c_0 = m_0$, $c_1 = m_0+m_1$, $c_2 = m_0+m_2$ and $c_3 = m_0+m_1+m_2+m_{12}$. These equations can be directly read off the generator matrix by looking at the positions of 1s down its columns associated with each c_x. These equations can be added to get $m_{12} = c_0+c_1+c_2+c_3$. This procedure can be repeated to get the other four expressions for m_{12}: $m_{12} = c_4+c_5+c_6+c_7 = c_8+c_9+c_{10}+c_{11} = c_{12}+c_{13}+c_{14}+c_{15}$. Thus, to estimate m_{12} from the received word $\mathbf{r}$, a majority vote of the four corresponding estimations are taken:

$\hat{m}_{12}^{(1)} = r_0+r_1+r_2+r_3$ $\quad$ $\hat{m}_{12}^{(2)} = r_4+r_5+r_6+r_7$

$\hat{m}_{12}^{(3)} = r_8+r_9+r_{10}+r_{11}$ $\quad$ $\hat{m}_{12}^{(4)} = r_{12}+r_{13}+r_{14}+r_{15}$

$\hat{m}_{12} = maj\{ \hat{m}_{12}^{(1)}, \hat{m}_{12}^{(2)}, \hat{m}_{12}^{(3)}, \hat{m}_{12}^{(4)} \}$

$\hat{m}$ is used here to indicate an estimation of m. This same procedure can be followed to estimate $\hat{m}_{13}$. The result would be:

$\hat{m}_{13}^{(1)} = r_0+r_1+r_4+r_5$ $\quad$ $\hat{m}_{13}^{(2)} = r_2+r_3+r_6+r_7$

$\hat{m}_{13}^{(3)} = r_8+r_9+r_{12}+r_{13}$ $\quad$ $\hat{m}_{13}^{(4)} = r_{10}+r_{11}+r_{14}+r_{15}$

$\hat{m}_{13} = maj\{ \hat{m}_{13}^{(1)}, \hat{m}_{13}^{(2)}, \hat{m}_{13}^{(3)}, \hat{m}_{13}^{(4)} \}$

Once the components of $\mathbf{m}_2$ are all estimated, it is multiplied by $\mathbf{G}_2$ and subtracted form $\mathbf{r}$ to get $\mathbf{r}' = \mathbf{r}-\hat{\mathbf{m}}_2\mathbf{G}_2$. The entire procedure above can be repeated to get an estimate for $\hat{\mathbf{m}}_1$. Continuing with this RM(2,4) example, $m_1 = c_0+c_2 = c_2+c_3 = c_4+c_5 = \ldots = c_{14}+c_{15}$. Again, a majority vote of the estimates is taken: $\hat{m}1 = maj\{\hat{m}_1^{(1)}, \hat{m}_1^{(2)},\ldots, \hat{m}_1^{(8)}\}$. Once the four components of $\hat{\mathbf{m}}_1$ are estimated $\mathbf{r}''$ can be obtained by $\mathbf{r}'' = \mathbf{r}'-\hat{\mathbf{m}}_1\mathbf{G}_1$ which is also equal to $m_0\mathbf{1}+\mathbf{e}$. Thus, the estimate for $\hat{m}_0$ is $\hat{m}_1 = maj\{r_0'', r_1'', \ldots, r_{15}''\}$. All the components of the message vector $\mathbf{m}$ are now estimated and this estimate is the decoded message.

Running through an example of this decoding process, let the received vector $\mathbf{r}$ = (0101101100011011). $\hat{m}_{12}$ = maj{0,1,1,1} = 1, $\hat{m}_{13}$ = maj{1,1,1,1} = 0, $\hat{m}_{14}$ = 0, and all the the rest of the $\hat{m}_{xy}$s are 0. Thus $\hat{\mathbf{m}}_2$ = (000011) and

$$\begin{aligned}\mathbf{r}' = \mathbf{r}-\hat{\mathbf{m}}_2\mathbf{G}_2 = \quad & (0101101100011011)\\ & -(0001010000010100)\\ & \text{----------}\\ & (0100111100001111)\end{aligned}$$

The first order estimate are all 0 except $\hat{m}3$. Thus $\hat{\mathbf{m}}1$ = (0100) and

$$\begin{aligned}\mathbf{r}''= \mathbf{r}'-\hat{\mathbf{m}}_1\mathbf{G}_1 = \quad & (0100111100001111)\\ & -(0000111100001111)\\ & \text{----------}\\ & (0100000000000000)\end{aligned}$$

The estimate for $\hat{m}_0$ is clearly 0. Thus, $\mathbf{r}$ is decoded as $\mathbf{m}$ = $(\hat{m}_0, \hat{m}_4, \hat{m}_3, \hat{m}_2, \hat{m}_1, \hat{m}_{34}, \hat{m}_{24}, \hat{m}_{14}, \hat{m}_{23}, \hat{m}_{13}, \hat{m}_{12})$ = (00100000011).

There is a general method of finding the checksums upon which the majority decisions are taken. Let P_x be the 1s complement of the binary number for x. For example, P_3 = 1100 because 3 is 0011 in binary. To find the checksums for estimating $m_{i1,i2,\ldots,ik}$, let S be the set of P_xs indexed by the 1 positions of the basis vector $v_{i1}v_{i2}\ldots v_{ik}$. Let T be the set of P_xs indexed by the 1 positions in the complementary subspace to S, $v_{\{x:\ \{1,2,\ldots,m\}-\{i1,i2,\ldots ik\}\}}$. Expressing T in P_xs, the indices of P indicate the first checksum. Translating T by each vector in S, expressing the results in P_xs and taking the indices for each translation indicates the other checksums. For example, to find the checksums for m_{34} on code R(2,4), first look at basis vector $\mathbf{v}_3\mathbf{v}_4$ = (0000000000001111). Thus, S = $\{P_{12}, P_{13}, P_{14}, P_{15}\}$. The complementary subspace to S is $\mathbf{v}_1\mathbf{v}_2$ since {1,2,3,4} - {3,4} = {1,2}. $\mathbf{v}_1\mathbf{v}_2$ = (0001000100010001). Thus, T = $\{P_3, P_7, P_{11}, P_{15}\}$. To find the translations of T:

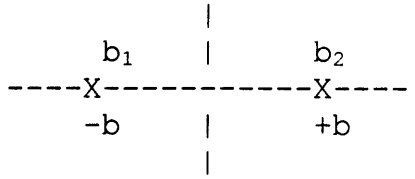
S={(0011),(0010),(0001),(0000)}
T={(1100),(1000),(0100),(0000)} = $\{P_3, P_7, P_{11}, P_{15}\}$
Translating T by (0011) = {(1111),(1011),(0111),(0011)}
= $\{P_0, P_4, P_8, P_{12}\}$
Translating T by (0010) = {(1110),(1010),(0110),(0010)}
= $\{P_1, P_5, P_9, P_{13}\}$
Translating T by (0001) = {(1101),(1001),(0101),(0001)}
= $\{P_2, P_6, P_{10}, P_{14}\}$

Looking at the indices of the Ps in T and its translations, the checksums for $m_{34} = c_0+c_4+c_8+c_{12} = c_1+c_5+c_9+c_{13} = c_2+c_6+c_{10}+c_{14} = c_3+c_7+c_{11}+c_{15}$.

1e) Bit Error Rate Performance

The noise signal, which gets added to the transmitted signal during its propagation through the channel, is modeled as White Gaussian Noise, WGN in this thesis. That is, it is modeled with a Gaussian distribution of the noise amplitude and a constant power spectral density over the bandwidth of interest. The Gaussian probability distribution f(x) will be abbreviated as ~N(u,s) where u is the mean and s is the standard deviation. It is assumed that the mean is zero because if it were not, that mean could simply be subtracted off the incoming signal and the new signal would be zero mean. The probability that the random variable x is between y and z, P(y<x<z) is the integral of f(x) with respect to x from y to z.

The simplest modulation scheme to analyze is BPSK. It is analyzed over a signal space which is a vector space over the infinite field of real numbers. Signals are represented as vectors where the length of the vector is the square root of the energy in the signal and an inner product is defined. BPSK uses two antipodal signals, representing 0 and 1. It is always assumed that each signal is transmitted with equal probability. The two signals have equal energy and are 180° out of phase. Let b_x be the head of the vector representing signal x whose tail is at the origin of the signal space. $b_0 = -b_1 = b$. The 2 dimensional signal space looks like:

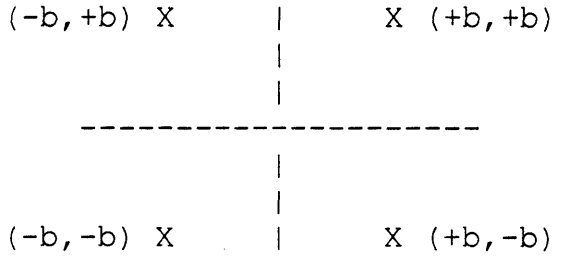


The received signal is $Z_i = b_i + n_i$. The optimal decision region for each bit is the Y axis and decisions are made by comparing the energy of the received signal with the energies of the two possible transmitted signals. Note that the distance between the possible transmitted signals in the signal space is $d = 2b = b_1 - b_0$. Assuming that a 1 is sent, an error in the decision of that bit is made when:

$$
\begin{aligned}
|Z_i - b_0|^2 &< |Z_i - b_1|^2 \\
|b_1 - b_0 + n_i|^2 &< |b_1 + n_i - b_1|^2 \\
|b_1-b_0|^2 + 2n_i|b_1-b_0| + |n_i|^2 &< |n_i|^2 \\
n_i d &< -d^2/2 \\
n_i &< -d/2
\end{aligned}
$$

Because the problem is symmetric, redoing it assuming a 0 is sent results in the same answer. Thus, the probability of signal error is the probability that the noise vector is greater that d/2.

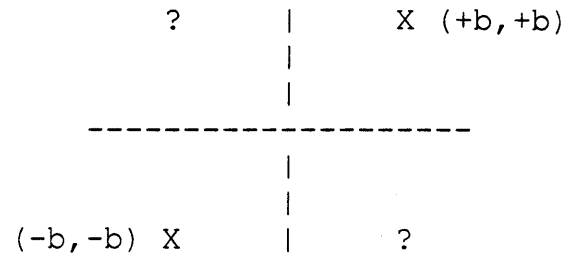
The function Q(y) is defined as the integral of ~N(0,1) from y to infinity. The probability that a Gaussian random variable x with zero mean and variance = s^2 is P(x > X) = Q(X/s). Thus, P(signal error) = P(n_i > d/2) = Q(d/2s). For WGN, the variance of the random variable n_i is the power spectral density $N_0/2$. The energy per signal E_s = energy per bit E_b (because there is one signal per bit in BPSK) = b^2 = $(d/2)^2$. Thus P(signal error) = P(bit error) = Q(d/2s) = Q'($2E_b/N_0$) where Q'(x) is defined as Q(sqrt(x)). For QPSK, the signal space looks like:



The decision regions are defined by both the X and Y axes. The distance of these signals to the origin is d/sqrt(2). Thus, E_b = $d^2/2$. Assuming that the top right signal is sent, P(signal error) is the probability that noise pushed the received signal outside its region. Therefore, P(signal error) = P(signal is push left beyond Y axis) + P(signal is pushed below the X axis) - P(both events)[the area that is counted twice]. Again, because of the symmetry of this problem, assuming other signals were sent results in the same answer. Thus P(signal error) = $2Q(d/2s) - Q^2(d/2s)$. Because Q(d/2s) is small, $Q^2(d/2s)$ is very small and can be discarded. Since there are 2 bits representing each signal, P(bit error) = P(signal error)/2 = Q(d/2s) = Q'($2E_b/N_0$) = same as for BPSK. Q'($2E_b/N_0$) is the probability of bit error that is used in this project. Thus it can be assumed that the modulation scheme being used in this thesis is BPSK or QPSK.

1f) Erasures and Soft Decision

It may seem as though information is being lost when each signal is simply decoded to the nearest bit before the set of n bits is decoded into a message block for coded signals. The minimum distance decoding in Hamming space takes as much as a 3dB loss in coding gain when compared to the optimum decoding in signal (Euclidean) space. For example, a (2,1,2) code, which is a shorthand way of writing a linear block code with ($n=2, k=1, d_{min}=2$) can be view in a 2-D space as:



where ? represents a received hard decision that is not a valid codeword. In this region, the decoder fails because simply knowing that the received signal is ? does not help the decision of which bit was sent. Notice that a valid codeword is distance b away from entering a ? region in Hamming space. For Euclidean space however, the decision region is the -45° diagonal through the origin. Thus the distance from one codeword to the other is sqrt(2)b. P(entering a ? region) = 2Q(b/s) while P(the optimal Euclidean decision) = Q(sqrt(2)b/s), which is 3dB better. As the Hamming space is taken to higher dimensions (as the coderate decreases), P(Error) for the optimum Hamming space decoder approaches P(Error) for the optimum Euclidean space decoder.

In soft decision, the receiver knows how far each signal is from the decision regions (knows the quality of its decision) and is able to use this information to help in the decoding process. The better the soft decision, the closer the Hamming space decoder mimics the Euclidean space decoder and thus the better its performance. Erasure decoding is the crudest form of soft decision. Here, there are three decision regions for each bit, 0, 1, or ?. ? represents a received signal that is close enough between a 0 and 1 that the receiver does not want to make a guess as to which one it is. Once all the bits of the codeword are received in this way, decoding goes as follows: Put 0s in

place of all the ?s and decode the word as normal to get $\mathbf{c}_0$. Put 1s in place of all the ?s and decode the word as normal to get $\mathbf{c}_1$. Compare the Hamming distance of $\mathbf{r}$ and $\mathbf{c}_0$ with that of $\mathbf{r}$ and $\mathbf{c}_1$ and choose the codeword closest to $\mathbf{r}$.

1g) Iterative Decoding

Iterative block codes, also referred to as turbo product codes, have their bits arranged in a 2 or higher dimensional matrix. Only the 2-D case is dealt with in this paper. Using an (n,k) code, a 2-D block can be created. Starting with a k X k matrix of k^2 message bits, each row can be encoded separately to create a new k X n matrix. Then each column of the k X n matrix can be encoded to create an n X n matrix which is ready for transmission across the channel. Note that different codes with different rates can be used to encode horizontally and vertically to form a rectangular matrix instead of simply a square matrix. In this paper, only square matrices using the same code horizontally and vertically are considered. The rate of such a code is k^2/n^2.

The decoder receives a matrix $\mathbf{R}_0$ and applies soft decision decoding to each row to get a reliability y_j and a decision d_j for each received bit r_j. The receive vector is mapped to the closest codeword in Euclidean space and the following d_js are determined from that codeword. To find the reliability y_j, the closest codeword which has a 0 in the j^{th} position is found and labeled $\mathbf{c}^0$. The closest codeword which has a 1 in the j^{th} position is found and labeled $\mathbf{c}^1$. Reliability $y_j = |\,|\mathbf{r}-\mathbf{c}^0|^2-|\mathbf{r}-\mathbf{c}^1|^2\,|$. Thus, if $\mathbf{c}^0$ and $\mathbf{c}^1$ are approximately the same distance from $\mathbf{r}$, then bit j is not relied upon much and y_j is low. If the distances of $\mathbf{c}^0$ and $\mathbf{c}^1$ from $\mathbf{r}$ vary greatly, then bit j is relied upon more in the decision process and y_j is higher. Let $\mathbf{r}_j' = y_j d_j$ which moves $\mathbf{r}_j$ in the direction of the decision d_j scaled by its reliability y_j. This is the extrinsic information that is important for iterative decoding. Running through this procedure for each row generates the matrix of $\mathbf{r}_j'$s labeled $\mathbf{W}_0$. The receive matrix $\mathbf{R}_0$ can be modified to $\mathbf{R}_1 = \mathbf{R}_0 + a_0\mathbf{W}_0$. Doing the same thing for the columns on the new matrix $\mathbf{R}_1$ to generate $\mathbf{W}_1$ allows the calculation of $\mathbf{R}_2 = \mathbf{R}_1 + a_1\mathbf{W}_1$. Using $\mathbf{R}_2$ the procedure on the rows can be repeated to find and even better $\mathbf{W}_2$. This operation can be iterated for as many times as desired and the final iteration makes a hard decision. The best scaling factors a_x here are found experimentally. These factors usually start small because the early calculations for $\mathbf{W}_x$ are less reliable and these factors get larger when the calculations for $\mathbf{W}_x$ become more reliable.

1h) ARQ and Hybrid Systems

An ARQ system handles errors by requesting retransmissions of packets in the form of negative acknowledgments (NAKs) which are usually piggybacked on packets traveling in the reverse direction. In a selective repeat scheme, packets are transmitted continuously and only those packets that are NAKed (or timed out) are retransmitted. This scheme, which is used in this thesis, is throughput efficient but requires buffering at the transmitter and receiver. Here throughput is defined as the inverse of the average number of bits needed to correctly transmit one bit of information. Assuming sufficient buffering to avoid overflow at both ends, the average number of transmitted packets required for one packet to be correctly received is:

$$N_{sr} = 1 \cdot P_c + 2 \cdot P_c(1-P_c) + 3 \cdot P_c(1-P_c)^2 + \ldots = 1/P_c$$

where P_c is the probability of a packet being transmitted correctly. The throughput of this system is $T_{sr} = 1/N_{sr} = P_c$. If this ARQ system also uses a linear block code of rate k/n to encode each packet, then P_c will increase. The throughput of this hybrid system is then $T_{sr} = P_c(k/n)$ where P_c is the new probability which takes linear block coding into account.

APPENDIX 2: C PROGRAMS TO SIMULATE PHASE I CODES

2a) Abbreviated C program for 1-D Hard code simulation: this code simulates the performance by generating a random message, encoding it as specified in Appendix 1d, adding noise (flipping bits with a probability specified in Appendix 1e), decoding the result and then checking whether this new message is the same as the original message. This procedure can be repeated many times to estimate a probability of a frame error.

```
const int n=64;          //Global Variables
int r; int k;
bool v6[]  ={0,0,0,0,0,…,1);          //these are the vectors
bool v5[]  ={0,0,0,0,0,…,1);          //comprise the rows of
… bool v1[]  ={0,1,0,1,0,…,1);        //the generator matrix.
bool v6v5[]={0,0,0,0,0,…,1);
… bool v5v4v3v2v1[]={0,0,0,0,0,…,1);

void CreateGeneratorMatrix(bool gen[n][n])
{     int col;
      for (col=0; col<n; col++)       //This procedure creates
      {     gen[col][0]  = 1;         //the generator matrix
            gen[col][1]  = v6[col];
            … gen[col][56] = v4v3v2v1[col];}}

void CreateRandomMessage(bool message[n][n])
{
      int col,row;              //matrix message gets modified
      for (col=0; col<k; col++)
            for (row=0; row<k; row++)
                  if (rand() > 16384) message[col][row]=0;
                  else message[col][row]=1;}

// This procedure will only encode 1 row or 1 column of the
// nXk message.  It is called by procedure EncodeMessage.
void EncodeLine(bool message_line[n], bool gen[n][n],
                        bool cw_line[n])
{
      int col, row;       //variables message & gen not modified
      bool sum;           //variable cw_line is modified

      for (col=0; col<n; col++)//This segment encodes our
      {                              //message by multiplying
        sum = 0;                 //message vector with
        for (row = 0; row<k; row++)//generator matrix.
          sum = sum ^ (message_line[row] && gen[col][row]);
        cw_line[col] = sum;}}

void EncodeMessage(bool message[n][n], bool gen[n][n], bool cw[n][n])
{     int col, row;               //message & gen not modified
      bool message_line[n]; //cw is modified
      bool cw_line[n];

      for (row=0; row<k; row++)
```

```
      {
        for (col=0;col<k;col++) //Grab each row from message
          message_line[col] = message[col][row]; //and
          EncodeLine(message_line,gen,cw_line);  //encode it
          for (col=0; col<n; col++)               //into cw
             cw[col][row] = cw_line[col];}}

void AddNoiseToCodeword(bool cw[n][n], int error)
{  int count,col,row,error_val;
   count=0; //error_val = (32767 * error_percentage);
   if     (error==1) error_val=8636; //P(e)=.2635, EbNo=0.2
   else if(error==2) error_val=6080; //P(e)=.1855, EbNo=0.4
   else if(error==3) error_val=4478; //P(e)=.1367, EbNo=0.6
   …else if(error==19) error_val= 5; //P(e)=.00015,EbNo=6.5

   for (row=0; row<k; row++)    //Above segment adds noise
      for (col=0; col<n; col++) //to the codeword cw.
        if (rand() < error_val)
            {count = count +  1;  //With certain probability
             if (cw[col][row]==1) cw[col][row]=0; //the bits
             else cw[col][row]=1;}}      //in cw are swapped

int CountBitErrors(bool message[n][n],bool decode[n][n])
{int sum=0, col, row;
 for (row=0; row<k; row++)
   for (col=0; col<k; col++)
      if (message[col][row] != decode[col][row]) sum=sum+1;
 return sum;}

bool CheckSum(bool rcv_cw[n], bool S[n], bool T[n])
{ int zerosum=0, onesum=0;    col;
  int s_count=0, t_count=0, count, count2;
  int s_element[n], t_element[n];
  bool sum[n];

  for (col=0; col<n; col++)
  { if (S[col] == 1)          //gather S subspace points
      {s_element[s_count] = (n-1) - col;  //and store in
       s_count = s_count + 1;}                  //array s_element
    if (T[col] == 1)          //gather T subspace points
      {t_element[t_count] = (n-1) - col;  //and store in
       t_count = t_count +1;}}                  //array t_element

  for (count=0; count<s_count; count++) sum[count] = 0;
  for (count=0; count<s_count; count++)
    for (count2=0; count2<t_count; count2++)
       sum[count] = sum[count] ^ rcv_cw[(t_element[count2] +
                                        s_element[count])];
  for (count=0; count<s_count; count++)
      if (sum[count] == 0) zerosum = zerosum + 1;
      else if (sum[count] == 1) onesum = onesum + 1;

  if (zerosum > onesum) return 0;
      else if (onesum > zerosum) return 1;
      else return 0;}

void DecodeLine(bool cw_line[n], bool gen[n][n], bool m[n])
```

```
{     bool sum;   rcv_cw[n];
     int maj,col,row;

      for (col=0; col<n; col++)  //Creating copy of cw_line
        rcv_cw[col] = cw_line[col]; //in local variable.          if
(r==4)
       {m[42]=CheckSum(rcv_cw,v6v5v4v3,v2v1);  //compute
        m[43]=CheckSum(rcv_cw,v6v5v4v2,v3v1);    //value of
       m[44]=CheckSum(rcv_cw,v6v5v4v1,v3v2);  //vector m4
        …m[56]=CheckSum(rcv_cw,v4v3v2v1,v6v5);
        for (col=0; col<n; col++)   //Calculate m4 times G4
          {sum=0;                   //and subtact off of r.
            for (row=42; row<k; row++)
            sum = sum ^ (m[row]&&gen[col][row]);
            rcv_cw[col] = rcv_cw[col] ^ sum;}}
      if (r>=3)
       {m[22]=CheckSum(rcv_cw,v6v5v4,v3v2v1);    //compute m3
        …m[41]=CheckSum(rcv_cw,v3v2v1,v6v5v4);
        for (col=0; col<n; col++)  //Calculate m3 times G3
            {sum=0;                       //and subtact off of r'.
             for (row=22; row<42; row++)
             sum = sum ^ (m[row]&&gen[col][row]);
             rcv_cw[col] = rcv_cw[col] ^ sum;}}
      if (r>=2)
       {m[7] =CheckSum(rcv_cw,v6v5,v4v3v2v1);    //Compute m2
        …m[21]=CheckSum(rcv_cw,v2v1,v6v5v4v3);
        for (col=0; col<n; col++)  //Calculate m2 times G2
            {sum=0;                       //and subtact off of r''.
             for (row=7; row<22; row++)…}}
      if (r>=1)
       {m[1] =CheckSum(rcv_cw,v6,v5v4v3v2v1);   //Compute m1
        …m[6] =CheckSum(rcv_cw,v1,v6v5v4v3v2);
        for (col=0; col<n; col++) //Calculate m1 times G1
{sum=0;                     //and subtact off of r'''.
             for (row=1; row<7; row++)…}}
      if (r>=0)
      {maj=0;                           //For m0, we simply do a
for (col=0; col<n; col++)  //majority vote on r''''
         if (rcv_cw[col] == 1) maj = maj + 1;
       if (maj > 32) m[0]=1; else m[0]=0;}}

void DecodeBlock(bool cw[n][n], bool gen[n][n], bool decode[n][n])
{int col, row;
 bool decode_line[n];
 bool cw_line[n];

 for (row=0; row<k; row++)
   {for (col=0; col<n; col++)      //Grab each row from
       cw_line[col] = cw[col][row]; //kXn decoded rows
    DecodeLine(cw_line,gen,decode_line);//and decode them
    for (col=0; col<k; col++)       //into a final block.
       decode[col][row] = decode_line[col];}}

//******************** MAIN PROGRAM ******************
int main()
{int test;
 int error, bit_error_count, frame_error_count;
```

```
long running_bit_error_count;
float BER,k2,running_error;

bool message[n][n];     //size nXn but only use kXk of it
bool decode[n][n];      //decoded message (also kXk bits)
bool cw[n][n];          //codeword
bool gen[n][n];  //generator matrix

CreateGeneratorMatrix(gen);
for (error=1; error<20; error++)
 {printf ("\nerror%d ",error);
  for (r=1; r<5; r++)
   {if      (r==1) k= 7; else if (r==2) k=22;
     else if (r==3) k=42; else if (r==4) k=57;
     running_bit_error_count=0;
     frame_error_count=0;
     for (test=0; test<1000; test++)//Runs 1000 test frames
       {srand(test);                // for each value of r
                                          // and prints results
        CreateRandomMessage(message);
        EncodeMessage(message,gen,cw);
        AddNoiseToCodeword(cw,error);
        DecodeBlock(cw,gen,decode);
        bit_error_count=CountBitErrors(message,decode);

        if (bit_error_count != 0) frame_error_count += 1;
        running_bit_error_count += bit_error_count;}
        printf ("%d ",frame_error_count);
        printf ("%d ",running_bit_error_count);
        running_error = running_bit_error_count;
        k2 = k*k*10;
        BER = running_error / k2;
        printf ("%.4f ",BER);} }
return 0;}
```

2b) Abbreviated C program for 2-D Soft code simulation: this code simulates iterative erasure decoding as specified in Appendix 1 sections f and g. When a procedure is identical to one in Appendix 2b, it is specified as such and thus not repeated.

```
void CreateGeneratorMatrix(bool gen[n][n])
{…same as 2a}

void CreateRandomMessage(bool message[n][n])
{…same as 2a}

void EncodeLine(bool message_line[n], …)
{…same as 2a}

void EncodeMessage(bool message[n][n], …)
{ int col, row;
  bool message_line[n], cw_line[n];

  for (row=0; row<k; row++)
   {…same as above}
  for (col=0; col<n; col++)
   {for (row=0; row<k; row++)          //Grab each col from
       message_line[row] = cw[col][row];  //kXn encoded rows
    EncodeLine(message_line,gen,cw_line);//and encode
    for (row=0; row<n; row++)            //them into an
      cw[col][row] = cw_line[row];}}     //nXn block

void AddNoiseToCodeword(bool cw[n][n], …)
{ if      (error==1)  error_val = 9326;    //P(e)=.2846;
  else if (error==2)  error_val =   6895; //P(e)=.2104;
  else if (error==3)  error_val =   5311; //P(e)=.1621;
  …else if(error==18) error_val = 30;      //P(e)=.0009;

  if      (error==1)  error_val2 = 7972; //P(e)=.2433;
  else if (error==2)  error_val2 = 5328; //P(e)=.1626;
  …else if(error==18) error_val2 =    2; //P(e)=.0001;

  for (row=0; row<n; row++)
      for (col=0; col<n; col++) //add noise to codeword
      { number = rand();
        if (cw[col][row]==1) corrupt_cw[col][row]='1';
        else corrupt_cw[col][row]='0';
        if (number < error_val)
        { corrupt_cw[col][row]='X'; //bits are undecided.
          sumX+=1;}
        if (number < error_val2)    //bits are swapped
        { if (cw[col][row]==1) corrupt_cw[col][row]='0';
          if (cw[col][row]==0) corrupt_cw[col][row]='1';
          sum+=1;}}}

int CountBitErrors(bool message[n][n],bool decode[n][n])
{…same as 2a}

bool CheckSum(bool rcv_cw[n], bool S[n], bool T[n])
{…same as 2a}
```

```
void DecodeLine(bool cw_line[n], bool gen[n][n], bool m[n])
{…same as 2a}

void ErasureDecodeLine(char era_cw_line[n], …)
{int col, distance0=0, distance1=0;
 bool cw_line[n],temp_decode_line0[n],temp_decode_line1[n];

 for (col=0; col<n; col++)    //replace Xs with 0s
 {                                    //and hard decode
   if      (era_cw_line[col]=='0') cw_line[col]=0;
   else if (era_cw_line[col]=='1') cw_line[col]=1;
   else if (era_cw_line[col]=='X') cw_line[col]=0;}
 DecodeLine(cw_line, gen, temp_decode_line0);
 for (col=0; col<n; col++)    //replace Xs with 1s
 {                                    //and hard decode
   if      (era_cw_line[col]=='0') cw_line[col]=0;
   else if (era_cw_line[col]=='1') cw_line[col]=1;
   else if (era_cw_line[col]=='X') cw_line[col]=1;}
 DecodeLine(cw_line, gen, temp_decode_line1);
 for (col=0; col<n; col++)    //determine Hamming distances
 {if      ((era_cw_line[col]=='0') &&
      (temp_decode_line0[col]==1)) distance0 +=1;
  else if ((era_cw_line[col]=='1') &&
      (temp_decode_line0[col]==0)) distance0 +=1;}
 for (col=0; col<n; col++)    //Note that we consider
 {          //distances from X to be zero for both cases.
   if      ((era_cw_line[col]=='0') &&
      (temp_decode_line1[col]==1)) distance1 +=1;
   else if ((era_cw_line[col]=='1') &&
      (temp_decode_line1[col]==0)) distance1 +=1;}

 if (distance0 < distance1)
   for (col=0; col<n; col++)
      decode_line[col] = temp_decode_line0[col];
 else
   for (col=0; col<n; col++)
      decode_line[col] = temp_decode_line1[col];}

void DecodeBlock(char corrupt_cw[n][n], …)
{int col, row, count;
 bool decode_line[n], re_encode_line[n], cw_line[n];
 char era_cw_line[n], vert_correct_grid[n][n];
 char hori_correct_grid[n][n];
// We do this procedure 2ce.  First we do iterations
//impartial, then we do last iteration siding with hori.
 for (count=0; count<5; count++)
 {for (col=0; col<n; col++)
    {for (row=0; row<n; row++)      //Grab each col***
        era_cw_line[row] = corrupt_cw[col][row];
      ErasureDecodeLine(era_cw_line,gen,decode_line);
      EncodeLine(decode_line,gen,re_encode_line);
      for (row=0; row<n; row++)
      {if (corrupt_cw[col][row]=='0')
        {if (re_encode_line[row]==0)
              vert_correct_grid[col][row]='0';
           else if (re_encode_line[row]==1)
```

```
                vert_correct_grid[col][row]='X';}
            else if (corrupt_cw[col][row]=='1')
            { … follow impartial rules}
            else if (corrupt_cw[col][row]=='X')
            { … follow impartial rules}}
        for (row=0; row<n; row++)
          {for (col=0; col<n; col++)  //Grab each row***
             … do the same as did for columns}
         for (col=0; col<n; col++)//Combine to make corrupt_cw
            for (row=0; row<n; row++)//the overall correct grid
               {if (vert_correct_grid[col][row]=='0')
                     {if      (hori_correct_grid[col][row]=='0')
                           corrupt_cw[col][row]='0';
                      else if (hori_correct_grid[col][row]=='1')
                           corrupt_cw[col][row]='X';
                      else if (hori_correct_grid[col][row]=='X')
                           corrupt_cw[col][row]='0';}
                 else if (vert_correct_grid[col][row]=='1')
                     {…continue with impartial calculations}}}
 for (count=0; count<1; count++)
 {…repeat procedure above expect do the calculations
    that involve sizing with the horizontal.}
// Now we simply do the normal decode.
 for (col=0; col<n; col++)
  {for (row=0; row<n; row++)  //Grab each col
      era_cw_line[row] = corrupt_cw[col][row];
   ErasureDecodeLine(era_cw_line,gen,decode_line);
   for (row=0; row<k; row++)
      decode[col][row] = decode_line[row];}
 for (row=0; row<k; row++)
  {for (col=0; col<n; col++)  //Grab each row
      cw_line[col] = decode[col][row];
   DecodeLine(cw_line,gen,decode_line);
   for (col=0; col<k; col++)
      decode[col][row] = decode_line[col];}}

//********************* MAIN PROGRAM ******************
int main()
{int test;
 int error, bit_error_count, frame_error_count;
 long running_bit_error_count;
 float BER,k2,running_error;
 bool message[n][n];     //size nXn but only use kXk
 bool decode[n][n];           //decoded message (also kXk bits)
 bool cw[n][n];               //codeword
 bool gen[n][n];              //generator matrix
 char corrupt_cw[n][n];

 CreateGeneratorMatrix(gen);
 for (error=1; error<19; error++)
   {for (r=1; r<5; r++)
      {if      (r==1) k= 7; else if (r==2) k=22;
       else if (r==3) k=42; else if (r==4) k=57;
       running_bit_error_count=0;
       frame_error_count=0;
       for (test=0; test<1000; test++) //Runs 1000 frames
       { srand(test);
```

```
        CreateRandomMessage(message);
        EncodeMessage(message,gen,cw);
        AddNoiseToCodeword(cw,corrupt_cw,error);
        DecodeBlock(corrupt_cw,gen,decode);
        bit_error_count=CountBitErrors(message,decode);
       if (bit_error_count != 0) frame_error_count += 1;
        running_bit_error_count += bit_error_count;}
      printf ("%d ",frame_error_count);
      printf ("%d ",running_bit_error_count);
      running_error = running_bit_error_count;
      k2 = k*k*10;
      BER = running_error / k2;
      printf ("%.4f",BER);}}
return 0;}
```

APPENDIX 3: PHASE II THROUGHPUT CALCULATION PROGRAM

This is the MATLAB script file that is used to generate graphs for various throughputs and ℓ values. Using the probabilities calculated in section 3, this file determines optimal packet sizes and throughputs of the adaptive code for various fading environments. This file also calculates the throughput of the optimal non-adaptive code.

```
a1 = zeros(1,120);              %a = Pr(frame decodes incorrectly)
a2 = [06.825 04.779 03.207 02.013 01.256  0.706   … 0];
a3 = [67.611 61.298 54.542 47.494 40.414 33.500   … 0];
a4 = [95.026 93.160 90.860 87.981 84.420 80.180   … 0];
a5 = [99.205 98.824 98.287 97.546 96.545 95.220   … 0];

b1 = 1 - (a1 / 100);    %b = Pr(frame decodes correctly)
b2 = 1 - (a2 / 100);  b3 = 1 - (a3 / 100);
b4 = 1 - (a4 / 100);  b5 = 1 - (a5 / 100);

x = (.25:.25:30);                    % x = x-axis
k1 = 7; k2 = 22; k3 = 42; k4 = 57; k5 = 64;

for m2 = 1:20
   m = .5 * m2;
   for s2 = 1:20
      s = .25 * s2;
      for l = 1:500
      c = b1 .^ l;   %c=Pr(l frame block decodes correctly)
      y = (1/(((2*pi)^.5)*s)) * exp(-1 * (x-m).^2 /
                  (2*s^2));  % y = Norm(m,s) curve
      t = c*((k1*l)-64)/(64*l);      % t = throughput
      t(t<0) = - t(t<0)*0;           % set neg thrpts to 0
      tnorm = t .* y;                % normalize throughputs
      tnorml(l) = .25 * (t * y');    % overall thpts in tnorml
      end
      [thruput1(m2,s2),thruput1l(m2,s2)] = max(tnorml);
      % store best thrpt in thruput & best l in thruput1
   end
end

for m2 = 1:20
   m = .5 * m2;
   for s2 = 1:20
      s = .25 * s2;
      for l = 1:500
       {… repeat above expect c = b2 .^ l and store in
            thruput2(m2,s2) }
   end
end

{… repeat again for b3 and b4}
```

```
for m2 = 1:20
   for s2 = 1:20
      d = [thruput1(m2,s2) thruput2(m2,s2) thruput3(m2,s2)
                  thruput4(m2,s2) thruput5(m2,s2)];
      [thruputx(m2,s2),thruputx2(m2,s2)] = max(d);
   end                    %This segment picks the best
end                       %non-adaptive code for a given m & s.

for m2 = 1:20
   m = .5 * m2;
   for s2 = 1:20
      s = .25 * s2;
      for l = 1:500
      c1 = b1 .^ l;      %c=Pr(l frame block decodes correctly)
      t1 = c1*((k1*l)-64)/(64*l);   % t = throughput
      c2 = b2 .^ l;
      t2 = c2*((k2*l)-64)/(64*l);
      c3 = b3 .^ l;
      t3 = c3*((k3*l)-64)/(64*l);
      c4 = b4 .^ l;
      t4 = c4*((k4*l)-64)/(64*l);
      c5 = b5 .^ l;
      t5 = c5*((k5*l)-64)/(64*l);
      for e = 1:120
        t(e) = max([t1(e) t2(e) t3(e) t4(e) t5(e)]);
      end;
      t(t<0) = - t(t<0)*0;          %set any negative thrpts to 0
      y = (1/(((2*pi)^.5)*s)) * exp(-1 * (x-m).^2 /
                        (2*s^2));         % y = Norm(m,s) curve
      tnorm = t .* y;                     % normalize throughputs
      tnorml(l) = .25 * (t * y');         % store overall thrpts
      end
      [thruputa(m2,s2),thruputal(m2,s2)] = max(tnorml);
      %store best throughput in thruput & best l in thruputl
   end
end

%mesh(thruputa-thruputx);     %adaptive - non-adaptive
%mesh(thruputa ./ thruputx);  %adaptive / non-adaptive

subplot(2,1,1);
mesh(thruputx);   %thruputs for best non-adaptive code
subplot(2,1,2);
mesh(thruputx2);  %the specific code used

%subplot(2,1,1);
%mesh(thruputa);  %thruputs for adaptive code
%subplot(2,1,2);
%mesh(thruputal); %value of l at those thruputs

%subplot(2,2,1);  %we can also get thruputs and l values
%mesh(thruput1);  %for individual codes.
%subplot(2,2,2);
%mesh(thruput2);
% … mesh(thruput4);
```

APPENDIX 4: MARKOV CHAIN MODEL AND ITS AUTO-CORRELATION

This MATLAB script file runs the Markov Chain model of fading. It also calculates the auto-correlation of the state of the chain as a function of iterations. This function is used to determine how many iterations equal 1 second of time.

```
m = 5;      % range from 0 to 10 in steps of 0.5
s = 2;      % range from 0 to 5 in steps of 0.25
lambda = zeros(1,80);
mu = zeros(1,80);
sumstate = zeros(1,80);

rand('seed',0);          % set rand # seed to startup value
x = (.25:.25:20);        % x = x-axis (index for Eb/Nos)
y = (1/(((2*pi)^.5)*s)) * exp(-1 * (x-m).^2 / (2*s^2));
                  % y = Norm(m,s) curve
state = m*4;             % state = y's index corresponding to m
lambda(state) = .05;
while ( (lambda(state)>=0)&(mu(state)>=0)&(state<80) )
  mu(state+1) = lambda(state)*y(state)/y(state+1);
  lambda(state+1) = .1 - mu(state+1);
  state = state+1;       % setting up lambda and mu vectors
end
lambda(state)=0; mu(state)=.1;
state = m*4;
lambda(state) = 2*lambda(state);     % readjust lambda and mu at
mu(state) = 2*mu(state);             % mean value

for t = 1:100000 %-----------Creates Fading--------------------------
  if (state==(m*4))                  % m*4 b/c x in .25 increments
    if (rand > 0.5) dirgauss = 1;
    else dirgauss = 0; end;
  end;
  staterand = rand;
  if (staterand < lambda(state)) state=state+1;
  elseif (staterand < (lambda(state)+mu(state))) state=state-1; end;

  if (dirgauss==1)
    statex(t) = .25 * state;
  else st_ref = (m*8)-state;  % reflect state to other side of chain
    statex(t) = .25 * st_ref; end;
end;
save mc100k

AmtTau = 10000;                % How far do we calculate Rt?
Amt_t = 90000;                 % How many points do be average out?
for tau = 1:AmtTau             %------Calculates coorlelation--------
  SumR = 0;
  for t = 1:Amt_t
    SumR = SumR + (statex(t+tau)-m)*(statex(t)-m);
  end;
  R(tau) = SumR / Amt_t;
```

```
end;
save mc90k;
subplot(2,1,1); plot(R);
%plot(x,y);
subplot(2,1,2); plot(statex);
%subplot(2,1,2); %plot(x,sumstate);
```

APPENDIX 5: PHASE II THROUGHPUT SIMULATION PROGRAM

This script file is used to simulate the throughput performance of the ARQ/AFEC system. The UsingCtrlPkts variable is set appropriately to simulate the desired protocol.

```
clear;                          % clear all variables
%variables to input
m = 5;                          % mean of gaussian fade
s = 2.0;                        % standard deviation of gaussian fade
l = 18;                         % length of block
delay = 0;
iterations = 1000000;           % # of packets to transmit
UsingCtrlPkts = 1;              % 1=ctrl pkts, 0=code field in header
pkts_per_iter = 5;              % 500 for 1 GHz, 5 for 10 MHz

for trials = 1:5

delay = 25 * (trials-1);        % delays 0 25 50 75 100
%delay = 100 * trials + 100;    % delays 200 300 400 500
%delay = 500 * trials + 500;    % delays 1k 1.5k 2k 2.5k 3k 3.5k 4k

rand('seed',0);                 % set rand # seed to startup value
if (UsingCtrlPkts==0)
  overhead = 72;                % # of bits for CRC, pkt #, control
else overhead = 64; end;        %using 64 again to compare to
                                %NoDelay...will change to 72

a1 = zeros(1,80);               % a = Pr(frame decodes incorrectly)
a2 = [06.825 04.779 03.207 … 0];
a3 = [67.611 61.298 54.542 … 0];
a4 = [95.026 93.160 90.860 … 0];
a5 = [99.205 98.824 98.287 … 0];
b1 = 1 - (a1 / 100);            % b = Pr(frame decodes correctly)
b2 = 1 - (a2 / 100);
…b5 = 1 - (a5 / 100);
c1 = b1 .^ l; c2 = b2 .^ l;     % c = Pr(block decodes correctly)
…c5 = b5 .^ l;

x = (.25:.25:20);           % x = x-axis (index for Eb/Nos)

k1 = 7;  k2 = 22;           % # info bits per line of block
k3 = 42; k4 = 57;           %  k*l - overhead = # info bits per block
k5 = 64;
n = 64;
rate1 = ((k1*l)-overhead)/(n*l);
rate2 = ((k2*l)-overhead)/(n*l);
..rate5 = ((k5*l)-overhead)/(n*l);
                            % create vectors that store which code
for c = 1:80                % gives max throughput & value for that Pr
  [maxPrC(c),maxCode(c)] =max([c1(c)*rate1 c2(c)*rate2…c5(c)*rate5]);
end;

actual_EbNo = m;            % determines noise to pick Pr(success) from
assumed_EbNo = m;           % determines code transmitted & received
```

```
trans1 = 0; trans2 = 0;        % # of times transmitter switches code
trans3 = 0; trans4 = 0;        %  while system using code x
trans5 = 0;
num_code1 = 0; num_code2 = 0; % running sum of the number of
num_code3 = 0; num_code4 = 0; %   times that code x is transmitted
num_code5 = 0;
sum_success = 0;               % running sum of # of correct decodes

for d = 1:10000                % initialize delay matrix to the
  Delay(d)=m;                  %  mean EbNo value.
end;
assumed_Ptr = 1;               % initialize delay pointers
actual_Ptr = assumed_Ptr + delay;

% Setting up variables for Markov Chain
lambda =   zeros(1,80);        % lambda & mu used for Markov Chain.
mu =       zeros(1,80);        % sumstate used to check chain
sumstate = zeros(1,80);        %  distribution (make sure Gaussian)
y = (1/(((2*pi)^.5)*s)) * exp(-1 * (x-m).^2 / (2*s^2));
                               % y = Norm(m,s) curve
state = m*4;                   % state = y's index corresponding to m
lambda(state) = .05;
while ( (lambda(state)>=0)&(mu(state)>=0)&(state<80) )
  mu(state+1) = lambda(state)*y(state)/y(state+1);
  lambda(state+1) = .1 - mu(state+1);
  state = state+1;             % setting up lambda and mu vectors
end
lambda(state)=0; mu(state)=.1;% terminate positive end properly
state = m*4;
lambda(state) = 2*lambda(state); % readjust lambda and mu at mean
mu(state) = 2*mu(state);       %  value (b/c counted 2ce, 1 pos, 1 neg)

t = 0;
while (t < iterations)
  t = t + 1;

  %------------Gaussian Procedure: uses and updates------------------
  %------------variable state to update variable   ------------------
  %------------actual_EbNo                         ------------------
  if (state==(m*4))                % When state at mean, it has a
    if (rand > 0.5) dirgauss = 1;  %  50/50 chance of going positive/
    else dirgauss = 0; end;        %  normal or going negative/
  end;                             %  reflection of positive

  staterand = rand;                % Calculate next normal state
  if     (staterand < lambda(state)) state=state+1;
  elseif (staterand < (lambda(state)+mu(state))) state=state-1; end;

  if (dirgauss==1)
    sumstate(state) = sumstate(state)+1; % State positive
    actual_EbNo = state;
  else
    st_ref = (m*8)-state;      % reflect state to other side of chain
    if (st_ref>0)
      sumstate(st_ref) = sumstate(st_ref)+1;
      actual_EbNo = st_ref;
    else
```

```
      actual_EbNo = 0;              % when state < 0, modem unlock and
    end;                        %  Pr(success) = 0;
  end;

  %--------Delay procedure for picking code that system ------------
  %--------thinks is current                            ------------
  Delay(actual_Ptr) = actual_EbNo;        % use variable assumed_EbNo2
  assumed_EbNo2 = Delay(assumed_Ptr);     %  to compute transitions
  actual_Ptr = actual_Ptr + 1;                  %  info for throughput
  if (actual_Ptr==10001) actual_Ptr = 1; end;
  assumed_Ptr = assumed_Ptr + 1;
  if (assumed_Ptr==10001) assumed_Ptr = 1; end;
  if (assumed_EbNo2==0) assumed_EbNo2=1; end; %send at least code1

  %--------Collect the variables that are needed to -----------------
  %--------calculate the throughput                 -----------------
  if (actual_EbNo==0)
    success = 0;
  else
    if (maxCode(assumed_EbNo2) ~= maxCode(assumed_EbNo))
      if (UsingCtrlPkts==1)
        %tracks # times code changed while system in code x
        if     (maxCode(assumed_EbNo)==1) trans1=trans1+1;
        elseif (maxCode(assumed_EbNo)==2) trans2=trans2+1;
        …elseif (maxCode(assumed_EbNo)==5) trans5=trans5+1; end;
      end;
      assumed_EbNo = assumed_EbNo2; % start using assumed_EbNo now
    end

    if     (maxCode(assumed_EbNo)==1)
      PrSucc=c1(actual_EbNo);       % calculate correct probability
      num_code1=num_code1+1;        %  of success and calculate running
    elseif (maxCode(assumed_EbNo)==2) % sum of # of times each code
      PrSucc=c2(actual_EbNo);         % is transmitted
      num_code2=num_code2+1;
    …elseif (maxCode(assumed_EbNo)==5)
      PrSucc=c5(actual_EbNo);
      num_code5=num_code5+1;
    end;
    if (PrSucc > rand) success=1;
    else success=0; end;
  end;

  if (success)                            %  sum of the number of
    sum_success = sum_success + 1;        %  successful decodes
  end;

end;

%-----------------------------------------
% Now we calculate throughput.  For system using ctrl info in packets,
% increase overhead and zeroize transitions information.  For system
% using control packets, subtract transitions info because these use
% control pkts and are not counted in throughput.
%-----------------------------------------

PrCorrect = sum_success / iterations;
```

```
num_code1 = num_code1 * pkts_per_iter;     %Mult by pkts_per_iter and
num_code2 = num_code2 * pkts_per_iter;     %thus each numcode actually
…num_code3 = num_code5 * pkts_per_iter;    % represents x amt of pkts.
num_code_t = num_code1 + num_code2 + … + num_code5;

%# of info bits txmtd by each code in the session
% …divide sum by the total # of bits txmtd in the session
Rate1 = (num_code1-trans1)*((k1*l) - overhead);
Rate2 = (num_code2-trans2)*((k2*l) - overhead);
…Rate5 = (num_code5-trans5)*((k5*l) - overhead);
OverallRate = (Rate1+Rate2+Rate3+Rate4+Rate5) / (n*l*num_code_t);

Throughput = PrCorrect * OverallRate       % Thrput for Selective Repeat

end;

%subplot(2,1,1);
%plot(y);
%subplot(2,1,2);
%plot(sumstate);
```

REFERENCES:

[1] Buch, Gabriele; Burkert, Frank, "Concatenated Reed-Muller Codes for Unequal Protection," IEEE Communications Letters, vol. 3, no. 7, July 1999.

[2] Chang, Yuwei, "A New Adaptive Hybrid ARQ Scheme," IEEE Transactions on Communications, vol. 43, no. 7, July 1995.

[3] Efficient Channel Coding, Inc. "Technical Description of Turbo Product Codes." Eastlake, OH, Version 3.1, June 1998.

[4] Evans, John, "New Satellites for Personnal Communications," Scientific American, pp 71-77, April 1998.

[5] Forney, David, Class Notes for 6.451: Principles of Digital Communications. MIT Spring 2000.

[6] Howald, Rob, "Symbol Error Expressions Demystified," Communication Systems Design, vol. 6, no. 2, Feb 2000.

[7] Kallel, Samir, "Efficient Hybrid ARQ Protocols with Adaptive Forward Error Correction," IEEE Transactions on Communications, vol. 42, no. 2/3/4, Feb/Mar/Apr 1994.

[8] Lin, Shu; Costello, Daniel; Miller, Michael, "Automatic-Repeat-Request Error Control Schemes," IEEE Communications Magazine, vol.22 no.12, pp 5-17, Dec. 1984.

[9] Pyndiah, Ramesh, "Near-Optimal Decoding of Product Codes: Block Turbo Codes," IEEE Transactions on Communications, vol.46 no.8, pp 1003-1010, Aug. 1998.

[10] Shiozaki, Akira, "Adaptive Type-II Hybrid Broadcast ARQ System," IEEE Transactions on Communications, vol. 44, no. 4, April 1996.

[11] Shiozaki, Akira; Okuno, Kiyoshi; Suzuki, Katsufumi; Segawa, Tetsuro, "A Hybrid ARQ Scheme with Adaptive Forward Error Correction for Satellite Communications," IEEE Transactions on Communications, vol. 39, no. 4, April 1991.

[12] Wicker, Stephen B, Error Control Systems for Digital Communication and Storage. New Jersey, Prentice Hall, 1995.