# Implementation of High Performance LDPC in a Communications Channel

Submitted to

Professor David Casasent
Department of Electrical and Computer Engineering
Carnegie Institute of Technology
Carnegie Mellon University
Pittsburgh, PA 15213

Prepared by

Irsal Mashhor
Laura Miyakawa
Steven Sherry
18-551 Group 8

May 8, 2000

**Table of Contents**

Abstract i

1.0 Introduction 1

1.1 Purpose 1

1.2 Findings 1

1.3 Results 1

2.0 Turbo Codes 1

3.0 LDPC Advantages for the TI C67 EVM 3

4.0 How LDPC Works 3

4.1 LDPC Encode 4

4.2 LDPC Decode 5

4.2.1 Initial Probabilities 5

4.2.2 Syndrome Check 6

4.2.3 Iterative Decoding 7

5.0 Project Results 9

5.1 Memory Optimization 10

5.2 Performance 12

5.2.1 Qualitative Analysis 12

5.2.2 Quantitative Analysis 14

6.0 Discussion 15

6.1 Performance 16

6.2 Real Time Transmission 17

7.0 Conclusion 18

8.0 Future Extensions 19

Sources Cited 20

## Abstract

This project achieved its goal of successfully implementing an LDPC code channel on a TI C67 EVM. Provided in this report is background information on Turbo and LDPC codes, results and discussion of the project, a conclusion, and suggestions for further extension. The background on turbo codes includes information on traditional turbo coding encoding and decoding functions. The report then describes the advantages of LDPC and how the algorithm encodes and decodes. Next, the report details the memory optimizations made and a qualitative and quantitative analysis of the results. A discussion of channel performance and real time transmission follows. Finally, the report presents conclusions and suggests avenues for further research.

## 1.0 Introduction

### 1.1 Purpose

Our goal was to implement LDPC codes on a TI C67 EVM processor. This project posed many challenges for us: turbo code research, converting Matlab code to C, debugging, and optimization.

### 1.2 Findings

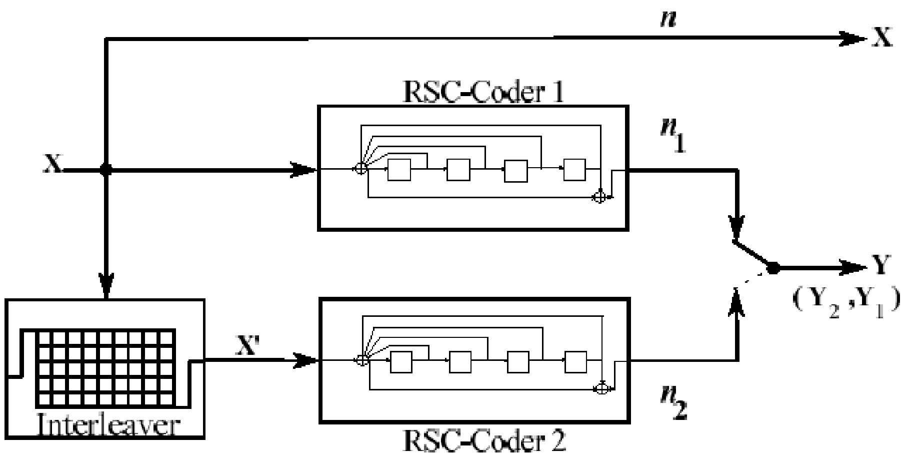
LDPC encoding can be simply implemented with xor matrix multiplication. Decoding is a more complicated and time-consuming iterative process. We felt that LDPC was a natural choice for implementation on the TI C67 EVM because the structure of the decoder lends itself to parallel operations.

### 1.3 Results

Although we had original hoped to attain real time video transmission, we are pleased with the successful implementation of LDPC codes on the TI C67 EVM. Our code achieves outstanding performance considering the memory constraints of the processor.

## 2.0 Turbo Codes

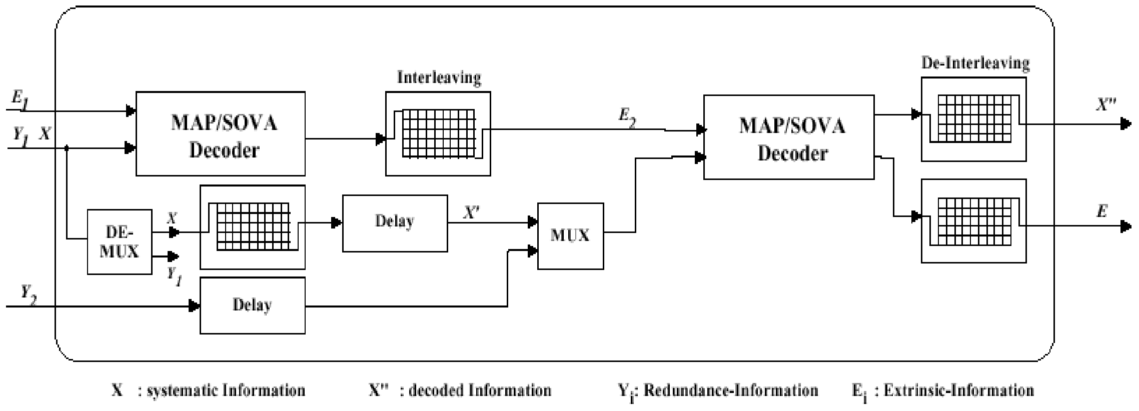
Turbo codes are to date the best encoding scheme available: they can achieve results closer to the Shannon Limit than any others. Turbo codes are a combination of convolutional codes with bit interleavers. A traditional Turbo code encoder is shown on the next page.



*TURBO - Coder*

Figure 1: Turbo Coder, Valenti

This encoder is far more complicated than the LDPC encoder, though it is not the hardest part of the turbo coding algorithm. The decoder is a combination of many interleavers as well as MAP/SOVA decoders to undo the convolutional encoding. The traditional decoder is shown below.



*Optimal Decoding of TURBO-Codes (one Iteration).*

Figure 2: Turbo Decoder, Valenti

Although the group wanted to strive for the results that turbo codes could offer, they decided that it would be beyond the scope of the course to implement them. With the advice of Professor Kumar, group 8 decided to implement LDPC codes. They are within the same family as turbo codes; however, they are much easier to implement.

## 3.0 LDPC Advantages for the TI C67 EVM

Our group chose to implement LDPC from the turbo code family because of its parallel processing possibilities, simple encode, and the fact that it does only error corrections. Since the TI C67 EVM can do up to eight operations in parallel, we felt it would be optimal to use the LDPC algorithm. The q and r computations in the decode function lend themselves easily to parallel processing. The encoding process is much simpler than that of the original turbo codes, which consisted of convolutional codes and interleavers. First, LDPC's encode function is a straight matrix multiplication. Second, in the binary case, the encode matrix multiplication is as easy as an xor of bits. Finally, many encoding schemes, like simple parity check, can only detect when an error is there not correct it. The receiver must then ask for the packet once again in order for the packet to be properly decoded. LDPC only does error correction. This means it can find the error in the packet and, given enough iterations, correct it. The only case where this fails is if the number of iterations is limited. If one were to implement LDPC with an acknowledge/repeat system or with infinitely many iterations, the transmission would be entirely without errors though very slow.

## 4.0 How LDPC Works

As in every coding scheme, LDPC has two stages: an encode step and a decode step. The LDPC encode is a simple matrix multiplication common to all sparse graph. The LDPC decode is where the difficulty and the strength of LDPC lies: it uses a decode matrix in an iterative decoding process on probabilities of specific symbols. This section will discuss a binary implementation of LDPC codes, covering the encode function, and the decode function in detail.

### 4.1 LDPC Encode

The best way to explain the encode step is by example. LDPC uses an encoding matrix, G. To encode a message word, one must simply multiply the message word by the G matrix. The result is a ready-to-be-transmitted codeword. G is typically made up of 2 parts: an identity and a parity check.

$$\mathbf{G = [\ I\ |\ P\ ]}$$

The purpose of the identity is to repeat the message word. This makes it simple to find the message word when the codeword has been recovered in the decode step. The parity check part of the matrix is what gives us the ability to correct errors. It simply takes a combination of bits and xors them to come up with another bit that is then tacked onto the end. This is illustrated by a message word, u, and the encoding matrix, G, below.

$$\mathbf{u = [1\ 0\ 0\ ]}$$

$$G = \begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{array}$$

When u is multiplied by G, the result is y, the codeword. The codeword y shows the message word, u, in the first 3 bits, and the parity check in the last 3 bits.

$$\mathbf{y= [\ 1\ 0\ 0\ 0\ 0\ 1\ ]}$$

Once y has been computed, it is ready to be sent through a communications channel to a receiver. The receiver takes the received codeword, s, which is y with noise added to it, and applies the decoding algorithm to it to get the message word back again.

**4.2 LDPC Decode**

The decoding step can be broken into 3 main parts: computing initial probabilities, syndrome check, and iterative decoding. Before anything else can be done, the received codeword, s, must be converted into symbol probabilities. Next a syndrome check is performed on the probabilities to determine if s is a valid codeword. If s is a valid codeword, the message is extracted from it, and the decoding step is finished. If we do not have a valid codeword, it is sent through the iterative decoder, which updates the probabilities by using knowledge of other probabilities. After each iteration the syndrome check is performed to see if a valid codeword has been decoded. Usually, a maximum number of iterations is set. Should the maximum number be reached, then the codeword is not decoded and there are possibilities for errors in the extracted message.

### 4.2.1 Initial Probabilities

The received codeword, s, is no longer a bit stream. It is a string of floating point numbers. For instance, although a one may have be sent through the channel, the noise may have made it into 1.2 or .2 depending on the noise strength. Using these values we can derive the probability that each bit was a one or a zero from the other side. The probability that a bit is a one is computed in $q^1$; the probability that it is a zero is set in $q^0$.

$$q^1 = \frac{1}{1+e^{-2y_k/\sigma^2}}$$

$$q^0 = 1 - q^1$$

Once all the $q_1$ and $q_0$ are computed the syndrome check and iterative decoding may begin.

### 4.2.2 Syndrome Check

The syndrome check takes the computed probabilities and decides whether the bit is a one or a zero based on the higher probability. It then uses a decode matrix, H, derived from the encode matrix, G, to decide if the codeword is valid or not. The codeword is simply multiplied by the transpose of H. If the result is an all zero vector, then the codeword is valid. H is of the form:

$$\mathbf{H = [\, P^T \mid I \,]}$$

Where $P^T$ is the transpose of the parity check from G, and I is the identity. For the previous example H is:

$$H = \begin{matrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{matrix}$$

Assuming our codeword comes through the channel with no errors the probabilities we can put y to the test.

$$\mathbf{Y * H^T = [\, 0\ 0\ 0 \,]}$$

In this case we exit the decoding process and take the first three bits of y since we know that they are the original message word. Suppose now that we receive the word m.

$$\mathbf{m = [\, 1\ 0\ 0\ 0\ 0\ 0 \,]}$$

When m is multiplied by the transpose of H the resulting vector is

$$\mathbf{[\, 0\ 0\ 1 \,]}$$

In this case we know that m is not a valid codeword, so we must go into the iterative decoding process to find a better codeword.

### 4.2.3 Iterative Decoding

The iterative decoding process takes the probabilities from the initial probabilities part and the H matrix from the syndrome check and updates the probabilities. From the H matrix a message-passing graph is formed. Here is the message-passing graph for the previous example.

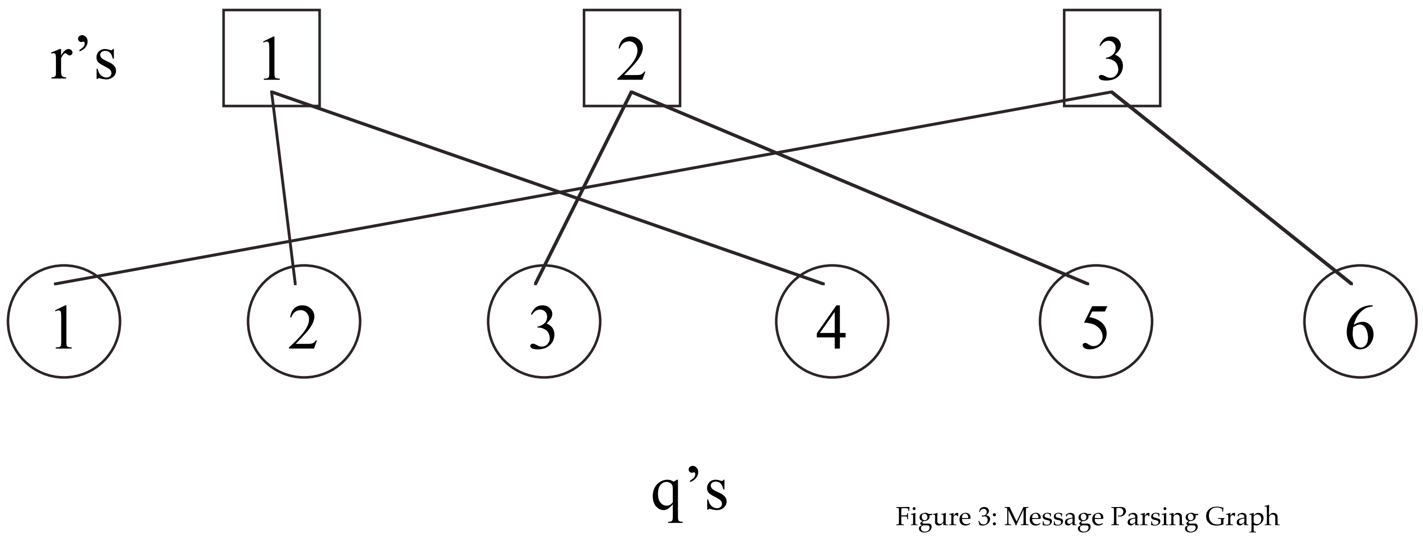


Figure 3: Message Parsing Graph

Looking back at the H matrix the first line contains ones in the second and the fourth places, and therefore, there is a connection between the first square and the second and fourth circles. The additional connections are made in similar manners. The idea in decoding is to iteratively go back and forth between the q probabilities and the r probabilities to get the best possible result. The q’s are set initially to be the initial probabilities. The r’s are then computed using the values of q and the following formulas.

$$r_k^0 = \_\ (1 + \Pi(q_k^0 - q_k^1\ ))$$

$$r_k^1 = \_\ (1 - \Pi(q_k^0 - q_k^1\ ))$$

In these formulas, $q_k^1$ is the probability that circle k is a one, and $q_k^0$ is the probability that circle k is a zero. These two should add to one. The differences of $q_k$’s are then multiplied together to form $r^1$ and $r^0$. For example to get $r_1^1$ one would multiply $(q_2^0 – q_2^1)(q_4^0 – q_4^1)$ then subtract that from 1 and divide by 2. Once the r’s have been calculated, it is possible to update the q’s. q’s are calculated using the following equations.

$$q_k^0 = \alpha_k p_j^0 \Pi r_k^0$$

$$q_k^1 = \alpha_k p_j^1 \Pi r_k^1$$

Here $\alpha$ is a normalization constant to ensure that $q_k^0$ and $q_k^1$ add to 1. p is the inverse of the previous value of q to avoid double counting. In the example $q_1^1$ is computed by $\alpha$ times the prior value of q times $r_3^1$.

From these 4 equations it is possible to iterate back and forth to get better and better values of q. Between iterations, it is customary to do a syndrome check on the values of q to see if a valid codeword has been determined. This enables the algorithm to do as much work as needed to retrieve the real codeword, while not wasting time doing extra iterations.

Once a valid codeword is determined, the message word is extracted by removing the first string of bits of length equal to the message length. Then the decoding algorithm is complete.

## 5.0 Results

In its final version, our project was fully functional on the EVM. We implemented the LDPC encoder, channel noise simulation, and the LDPC decoder, as described in the mid-semester project update. After completing the project, we next improved its speed and evaluated its functionality. Performance on the EVM was hindered by the requirements of the LDPC iterative decoding algorithm. Despite initial difficulties, we decreased decoder clock cycles used by 32%[1] at 3dB of SNR. Our group tested the final PC/EVM code at several SNR levels to perform a BER evaluation. We transmitted an image at 0dB and 1dB to gain a qualitative understanding of the LDPC channel. For a quantitative perspective, we performed a SNR vs. BER analysis.

[1] Our improvement efforts focused on the iterative decoder, as this is the bottleneck of the LDPC channel.

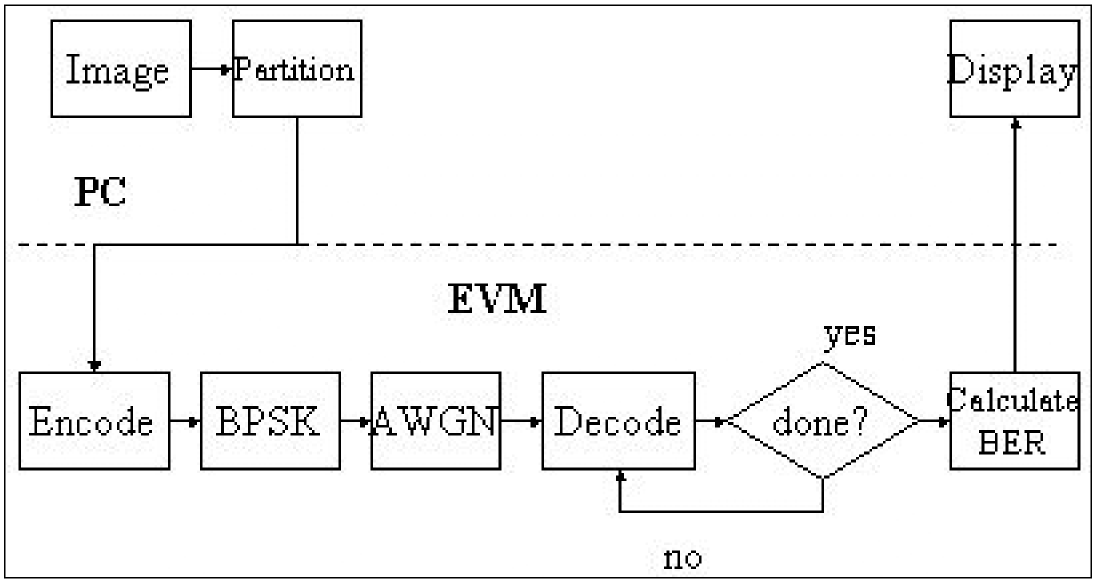


*Figure 4: Project Component Layout*

## 5.1 Memory Optimization

Figure 4 displays the layout of our project. In order to use the same EVM for encoding, adding noise, and decoding, we carefully controlled memory usage. One difference between our updated project proposal and our final project is the addition of noise on the EVM. We had initially envisioned sending the encoded signal form the EVM to the PC, adding noise on the PC, and passing back the noisy signal to the EVM to be decoded. An unforeseen consequence of simulating a channel with BPSK modulation and using a _ rate error correction code is that *four floating-point noise values must be generated for each bit of data transmitted.* Realizing this, we decided to send the noise values to the EVM, corrupt the encoded frames, and decode the frames serially.

A major drawback of the iterative LDPC algorithm is the accuracy that it requires. All of the probability-containing arrays must be double-precision, floating-point values. In an attempt to decrease on-chip memory used, we changed all double arrays to float arrays, but the algorithm could not decode properly. A C float value is accurate to $1*10^{-6}$ and the lower bound on the zero representation[2] in Igor's Matlab LDPC Simulation is $1*10^{-20}$, so the necessary accuracy for the algorithm lies between these bounds. In practice, many more values must be temporarily stored than many technical papers make apparent. Using a packet of 52 bits[3], over 3,000 double-precision, floating-point values need to be stored. These arrays alone take up 24KB of memory.

As the discussion below illustrates, it is possible to streamline the execution of the iterative decoding function, but one must remember that decoding times are highly dependent on the number of iterations necessary to decode a packet at given SNR. Since the decoder iterates on a packet until the packet has been successfully decoded,[4] there is a direct relationship between the BER, based on the SNR, and the time to decode a packet. Due to the highly variable nature of the channel rate and the limitations of the decoding matrix[5], an in-depth analysis of packet throughput is not appropriate, but for qualitative purposes we found that at 3dB the decoder processes about 280 bps. Our group chose to use 3dB of SNR as a benchmark because after two iterations at this power over 97% of noise is removed and almost all packets can be resolved in fewer than five iterations.

In its first operational version, the iterative decoder[6] exceeded 6.3 million clock cycles to decode a packet at 3dB. All operations were executed in the 8MB EVM external memory. Next, we determined that the Q, q, r, and check arrays were the most frequently used and placed them into on-chip memory. Moving these arrays decreased the average iteration to 5.2 million clock cycles. Still with some free space on the chip, we moved on the H matrix, which is used to index the q and r probability arrays, and the j and jv indexing arrays. This dropped the average decoding cycle to about 4.8 million clock cycles. Finally, we added the large dq and r_temp arrays into

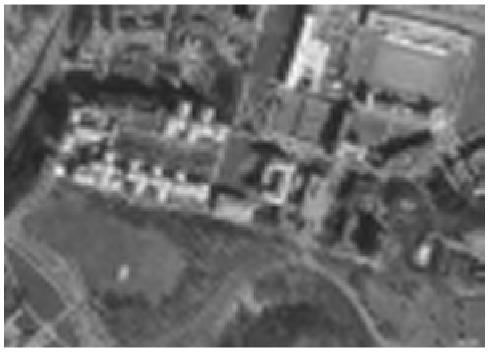
[2] Setting a non-zero, zero value prevents divide by zero errors.
[3] The 52-bit packet resulted from the dimensions of the H matrix.
[4] If the decoder reaches a max_iterations value, it declares the packet damaged.
[5] See Discussion: Performance
[6] See Appendix B: LDPC Decoder

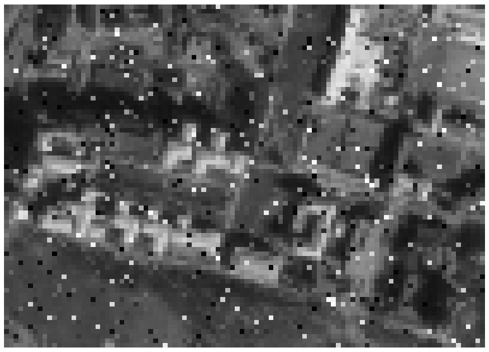
on-chip memory and removed the printf statements from the function, and reduced the average number of clock cycles to decode a packet at 3dB to 4.3 million. After completing memory optimizations, the maximum and average packet decoding attempts improved by 32% and the minimum packet decoding case improved by 34%.

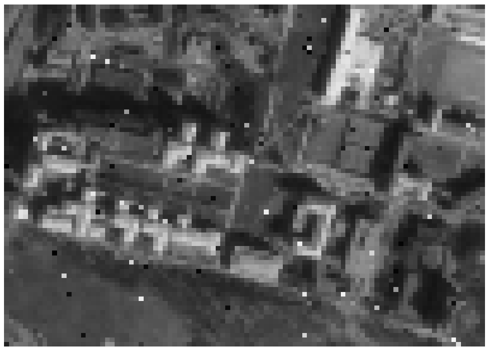
## 5.2 Performance

Our group performed both qualitative and quantitative analyses of the EVM LDPC Channel. For a qualitative understanding, we transmitted a RAW format image at several SNR levels, controlling the maximum number of iterations permitted. The uncompressed RAW format stores each pixel of a picture as an 8-bit value. It was necessary to use an uncompressed format in the qualitative test to prevent a single error in the header of a compressed file from allowing the file to open. Also, the image reflected every error introduced by the AWGN. For a quantitative analysis, we compared the code's BER at several SNR levels.

### 5.2.1 Qualitative Analysis

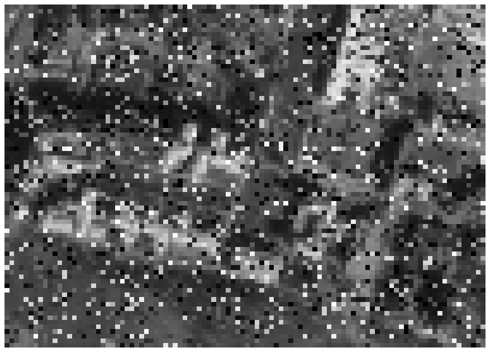
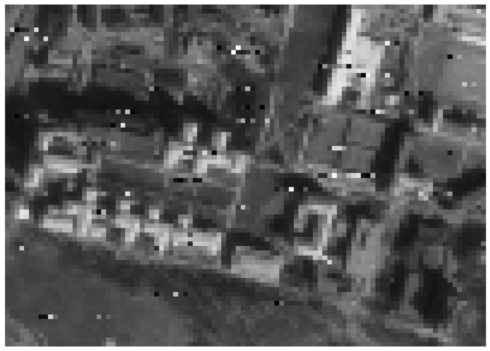
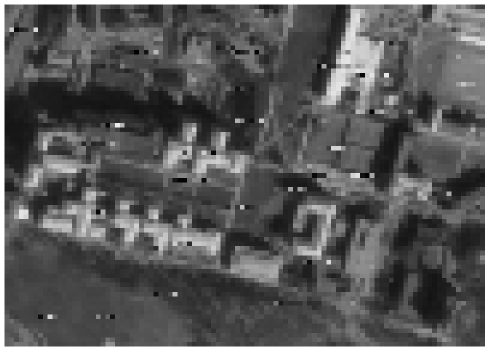






***Figures 5-7 (clockwise, from upper left):Original Image; SNR = 1dB, max_iterations = 1; SNR = 1dB, max_iterations = 2.***

At 1dB of SNR, the power of the LDPC code is clear. The original image in Figure 5 shows a satellite picture of Carnegie Mellon's campus taken during the 1995 US Geological Survey. The image is slightly distorted as it has been shrunken for transmission and blown up to three times its transmission size. The image was resized to transfer in a reasonable amount of time during the demonstration. The 232KB image accumulates over 18,000 errors in passing through the channel, or experiences an 8% error rate. In figure 6, after a single pass through the decoder, 76% of the noise has been removed. Figure 7 shows the same picture after two passes through the iterative decoder with 95% of noise removed.

***Figures 8-10 (clockwise, from upper left):***
***SNR = 0dB, max_iterations = 20; SNR = 0dB, max_iterations = 50, SNR = 0dB, max_iterations = 500.***

At 0dB of SNR, the encoding/decoding matrices are reaching the limits of their potential. However, the performance is still impressive when the size constraints are considered[7]. The unprocessed image contains over 37,000 errors, an error rate of 16%. In figure 8 with max_iterations at 20, 55% of the noise has been removed. Figure 9 shows a remarkable improvement from figure 8, with max_iterations increased to only 50. 96% of noise has been removed at this stage. If the decoder is permitted to continue to a maximum of 500 iterations, figure 10 shows that over 97% percent of errors can be removed.

It is important for the reader to consider that setting max_iterations to a certain value does not guarantee that this value will be reached even

[7] See Discussion: Performance

regularly. In practice, a histogram of the number of iterations to decode a packet shows that the tail of the graph falling off in an inverse power law relationship (i.e.: $1/x^n$). Some implementations of sparse graph ECCs have set a constant number of decoding iterations, but this configuration wastes both time, when processing an already decoded packet, and information, when a potentially decipherable packet is discarded. As both qualitative exercises above show, the great majority of noise can be removed with relatively little iteration. Considering this and weighting of the cost of each iteration, we determined that decoder efficiency is improved by several orders of magnitude by focusing decoding efforts on the few difficult-to-decode packets in each transmission.

### 5.2.2 Quantitative Analysis

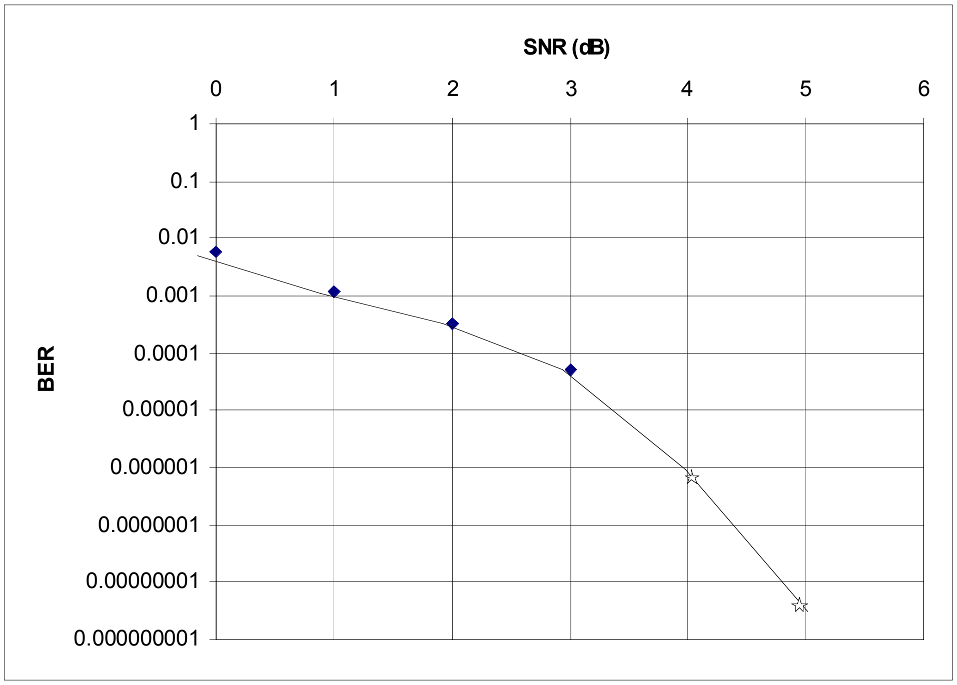


*Figure 11: SNR vs. BER Analysis and Expected Projection*

Our quantitative analysis of the LDPC channel consisted of an SNR vs. BER comparison. The figure 11 shows the relationship between SNR and BER we were able to determine, and the stars show the relationship we would expect to see, based on our knowledge of LDPC channels in general. By transmitting until 100 output errors[8] had

[8] LDPC decoding errors are always detected when using a check to end decoder iteration.

occurred,[9] we eliminated the possibility of random variance effecting the BER calculation.

## 6.0 Discussion

Despite the outstanding performance of our LDPC channel, two aspects did not perform as initially expected. The SNR vs. BER curve appeared fundamentally different from near Shannon Limit attaining codes, but a closer examination reveals that significant matrix size compromises had to be made to operate on the EVM. In light of these constraints, the matrix's performance is good. Also, we found it necessary to abandon real time video operation of the LDPC channel, but after more research, discovered that NASA itself rarely uses it.

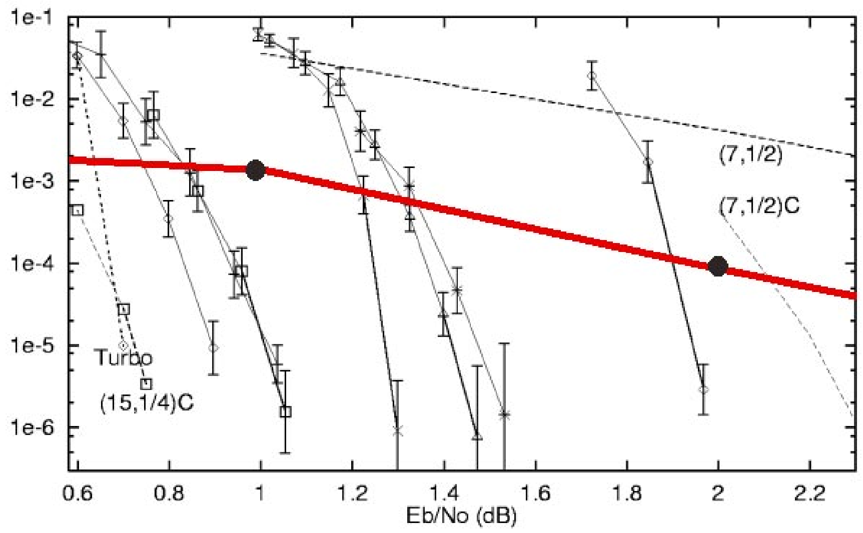


*Figure 12: Comparison of Results to Other Codes*

### 6.1 Performance

Although we successfully implemented the LDPC channel, the results of our SNR vs. BER plot were not in the range we expected. Figure 12 compares our results to other sparse graph and conventional codes. The bold line running through the middle of the graph indicates our results with a 52 by 200 H matrix. From left to right, the dashed line marked turbo is one of the best know turbo codes. The dashed line marked (15, 1/4)C is an extremely

[9] Professor Kumar suggested waiting for 100 output errors to be statistically confident of a calculated BER.

computationally expensive JPL concatenated code based on a convolutional code of constraint length 15 and rate _. The first group of solid lines represents several irregular, non-binary LDPC codes. The smallest H matrix in this group is 15000 by 5000 and rate 1/3. The next group of solid lines demonstrates the performance of regular, non-binary LDPC codes. The simplest code in this set uses a 13000 by 3000 H matrix at rate _. The last, lone solid line is a regular, binary LDPC code with matrix size of 30000 by 20000. The dashed line marked with (7, 1/2) shows the performance of a standard convolutional code with a constraint length of seven and a rate of _. (7, 1/2)C is the performance of the same convolutional code concatenated with a Reed-Solomon code.

The nature of the decoding matrix used appears to have the greatest effect on LDPC code performance. LDPC codes with irregular, non-binary matrices perform the best, followed by regular, non-binary matrices, irregular, binary matrices, and regular, binary matrices, respectively. Also, the randomly generated decoding matrices must be on the order of ten millions values to obtain optimal performance. Considering these issues, other factors being equal, one sees that the greater the entropy that a matrix has, the better it can be expected to perform. Intuitively, this makes sense; a large, irregular, non-binary matrix has a finer resolution than a single bit to contain randomness. We chose to use a relatively small regular, binary matrix of about 10,000 values for several reasons. A regular, binary matrix can be quickly encoded using David MacKay's alist matrix storage format. In an attempt to optimize our project for speed, we recognized that only a small matrix can fit into the EVM's on-chip memory and encoding/decoding times are directly related to matrix size. Due to speed and space constraints, the matrix we used was at least four orders of magnitude less than optimal regular, binary matrices. Viewed in this light, the performance of our LDPC channel is not surprising.

### 6.2 Real Time

Initially, our group aspired to better illustrate the real time properties of the channel we created. We found that decoding was the main bottleneck of our channel, and focused our efforts there. While we were able to transmit images in real time during our demonstration, a timing interdependence issue prevented us from achieving transfer rates sufficient for video. To better protect the image from the channel noise, we would need a larger, better-connected matrix. The sturdier packets would now need less iteration to decode, but the larger matrix would require more paging and memory transfer overhead for encoding and decoding. Performance increases gained in saving decoding iterations begin to offset memory transfer issues. By using a smaller matrix to reduce allocation overhead, fewer matrix interconnections mean that packets become more sensitive to noise. We chose to use a small decoding matrix, save clock cycles from transferring data, and iterate on each packet longer, and our transfers were limited to small images.

Even though real time video eluded our grasp, NASA itself rarely attempts it. The majority of deep-space transmissions NASA processes are images protected with the computationally expensive (15, 1/4) concatenated JPL code. Further, the propagation delay associated with sending these images can be up to several minutes. Consequently, it doesn't make sense for NASA to attempt to recover these images in real time because of the minimum amount of time required to recover them from the channel.

In summary, after we learned more about NASA's transmission strategies, and how infrequently they transmit real time video, our demonstration and transfer rates seem more appropriate.

## 7.0 Conclusion

In conclusion, our project was a success. We were able to implement the LDPC channel we envisioned in the beginning of the semester, and proposed in our project update. We successfully completed all programming, improved performance, and demonstrated our communications channel. When one considers that the Shannon Limit for the binary AWGN channel is around –1.5dB, our transmission at 0dB is quite impressive. Although we had initially hoped to accomplish real time transmission of video near the Shannon Limit, after we learned more about LDPC, we refocused and achieved more realistic goals.

Throughout the project, the novelty of the LDPC algorithm held us back. Instead of having readily available libraries of code, we had to write our own. Instead of working with an established algorithm, much of the theory behind sparse graph codes is still being developed. Instead of having previous 18-551 projects to base our work on, we struck out on our own. We understood the challenges that the project presented when we decided to undertake it, but feel that in the future, other groups will be able to use our project as a basis for further exploration.

## 8.0 Future Extensions

There are several areas in LDPC that have the potential for new 18-551 projects. With some research, it would be possible to write software to create custom size G and H matrices. These custom matrices could be geared towards noise robustness or speed efficiency, but their creation is non-trivial. Also, it has been suggested that LDPC codes are the best channel coding algorithms for the AWGN communications channel. A comparison of other channel codes with LDPC codes, using different matrices, code rates, and perhaps puncturing, would be an interesting exercise. However, the most exciting extension would be to separate the encoder and decoder to create a true communications channel. This would allow the transmission of data through the Internet, noise generating devices, and myriad other possibilities.

## Sources Cited

Bhagavatula, Vijayakumar, “An Introduction to Turbo Coding”, Carnegie Mellon University, 2000

Couleaud, Jean Yves, “High Gain Coding Schemes for Space communications” Signal Processing Research Institute, University of South Australia, 1995

Davey, Matthew C., “Error-correction using Low-Density Parity-Check Codes”, Cavendish Laboratory, University of Cambridge, 1999

Davey, Matthew C. and David J.C. MacKay, “Evaluation of Gallager Codes for Short Block Length and High Rate Applications”, University of Cambridge, 1999

Heegard, Chris and Stephen B. Wicker, “Turbo Coding”, Kluwer Academic Publishers, 1999

Levine, Benjamin R., R. Reed Taylor, and Herman Schmit, “Implementation of Near Shannon Limit Error-Correcting Codes Using Reconfigurable Hardware”, Carnegie Mellon University, (yet to be published)

MacKay, David J.C., “Good Error-Correcting codes based on Very Sparse Matrices”, University of Cambridge, 1998

Valenti, Matthew, “Turbo Codes and Iterative Processing”, Virginia Polytechnic Institute and State University, 1998