

# Improving Low Density Parity Check Codes Over the Erasure Channel

## The Nelder Mead Downhill Simplex Method

Scott Stransky

Programming in conjunction with: Boris Cukalovic

18.413 Final Project

Spring 2004

## **Abstract**

Error correcting codes prevent loss of integrity in data transmission. Low Density Parity Check codes are a family of codes that are specified by sparse matrices. Using the Nelder-Mead Downhill Simplex Evolution to design an irregular Low Density Parity Check code, we hope to improve upon the accuracy of decoding.

## **Introduction**

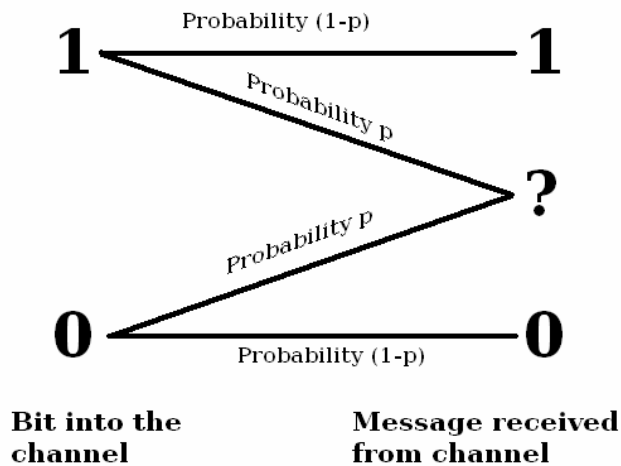
Loss of data integrity during transmission has always been a problem, since there is no way of guaranteeing accuracy in the bits received. Error correcting codes are a family of methods that correct this problem. In an error-correcting paradigm, extra bits are sent along with the message. These extra bits are carefully calculated using various methods. On the receiving end of the transmission, there must be a decoding algorithm to analyze all the bits that were received and try to deduce the bits that were sent. In this project, we will look at a specific type of code – the Low Density Parity Check (LDPC) code. We will try to use the Nelder-Mead optimization method to determine an optimal LDPC code, and thereby improve upon the number of errors that can be corrected using normal LDPC's.

## **Channels**

Channels are a representation of a path that bits (0's and 1's) can travel over. Bits will usually travel in packets or codewords (messages that have been encoded by the encoder of an error correcting algorithm), denoted by the symbol  $C$ . While in the channel, there is a chance that the bit will lose some or all of its integrity. A major type of channel is the erasure channel, which this project will deal with. This channel will be discussed in the next section.

## The Erasure Channel

The erasure channel with erasure probability  $p$  erases transmitted bits with probability  $p$  and accurately transmits bits with probability  $1-p$ . The chart below details this process:



Therefore, upon receiving a 0 or a 1 from the channel, there is a probability of 1 that that is the bit that was sent. If a “?” is received, there is a 0.5 probability of a 1 having been sent, and a 0.5 probability of a 0 having been sent. These properties will be important in the next section. The following chart explains this:

Bit Received	Bit Sent	Probability
1	1	1
	0	0
0	1	0
	0	1
?	1	0.5
	0	0.5

# Error Correcting Codes

Error correcting codes provide a way of reliably transmitting data over a channel. The general idea of an error correcting code is to add extra bits to the message (the encoding process), based on the message bits themselves, to create a codeword  $C$ . Once passed through a channel, a decoding algorithm can be applied to correct incorrect bits in the received message. We can define the “rate” as the number of message bits per bit sent, so if there are twice as many bits sent as are in the message, the rate is  $\frac{1}{2}$ . An example of an error correcting code is a Low Density Parity Check code.

## Regular Low Density Parity Check Codes (LDPC Codes)

Regular Low Density Parity Check codes are capable of decoding received messages with a considerable number of erasures. In fact, a normal LDPC can decode a message properly even if about 42% of the bits are lost over an erasure channel. These codes are made from graphs of message and parity nodes. The message nodes contain the bits of the message from the channel; while the parity nodes ensure that the parity of all the message nodes connected to it by the graph are 0. The procedure for this type of code can be broken down into three separate stages: generating a matrix and dual, encoding codewords, and decoding.

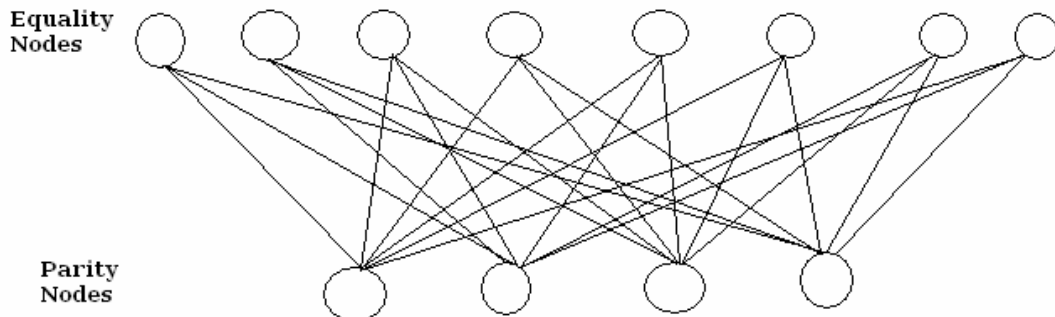
- Generating a matrix and dual: the matrix itself will represent a bipartite graph of decoding nodes – both parity check and message (also known as equality nodes) – that the decoding algorithm will use. All of the 1’s in the matrix will represent a mapping from an equality node (the columns) to a parity check node (the rows). First, we need to decide on what qualifies a code as normal, and for the purposes of this paper, a “3-6” graph will be normal. This means that there are 3 edges out of each message node and 6 edges out of each parity check node. Given that each node must connect to only one other node, there must be twice as many message

nodes as parity check nodes. This can be represented in a sparse matrix. A sparse matrix is a matrix with few 1's; this allows a computer to only store the position of the 1's. In a 5000x10000 matrix, this can save a great deal of memory. To generate a "3-6" sparse matrix, there must be three 1's per column and six 1's per row. These should be placed as randomly as possible. An example of this is (where every position represented by the dots would be a zero):

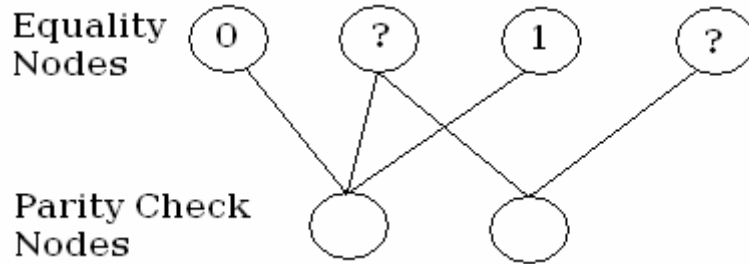
$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & \dots & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & \dots & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & \dots & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & \dots & 1 & 1 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 1 & 0 & 1 & 1 & \dots & 0 & 0 & 1 & 1 \end{bmatrix}$$

The next step is to generate the dual of this matrix. This can be accomplished with a C routine that we were provided with. The dual is what will be used to encode.

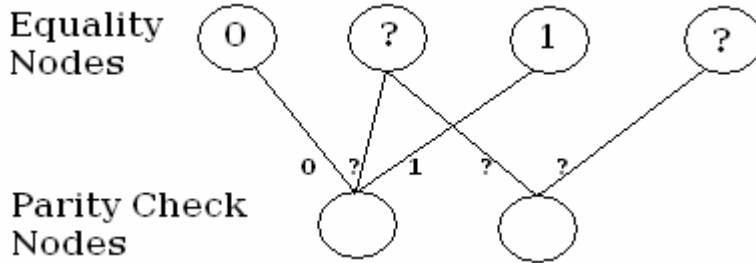
- Encoding: using the dual. To encode a 5000-bit message with a 5000x10000 dual matrix, use multiplication and multiply the message as a vector by the dual matrix. The resultant will be a 10000-bit long codeword vector.
- Decoding: using the bipartite node graph. Each message node is connected to three unique parity check nodes, while each parity check node is connected to six unique message nodes. This is shown in the following diagram (though there are only 8 message nodes):



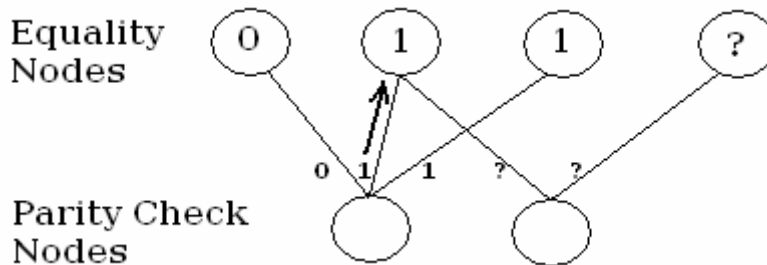
The decoding process over the erasure channel is simplified, as opposed to over any other channel, because if a bit is received, it is guaranteed that that bit was sent. To begin decoding, the message from the channel is placed in the equality nodes:



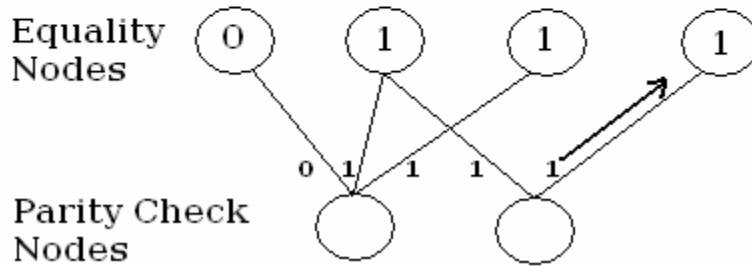
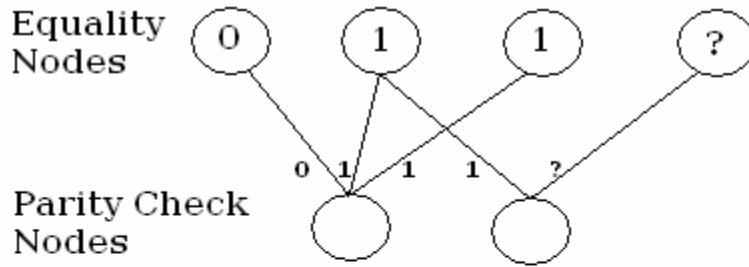
These nodes then transmit their messages through all of their outgoing edges:



Next, the parity check nodes try to determine the value of any “?” messages in the following way: If, at a parity check node, there is one question mark and all other bits are known, then the “?” bit is equal to the sum (mod 2) of all the other bits. If there is more than one “?”, then nothing can be determined. Following this stage, the newly known bits are sent back to the equality nodes:

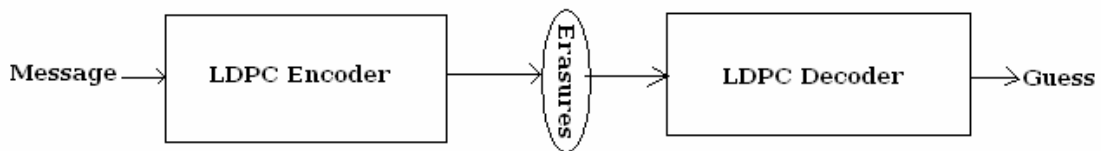


This process then repeats until all bits are known, or no new bits are learned on any given iteration (and therefore errors cannot be corrected):



At termination, the corrected message will appear in the equality nodes.

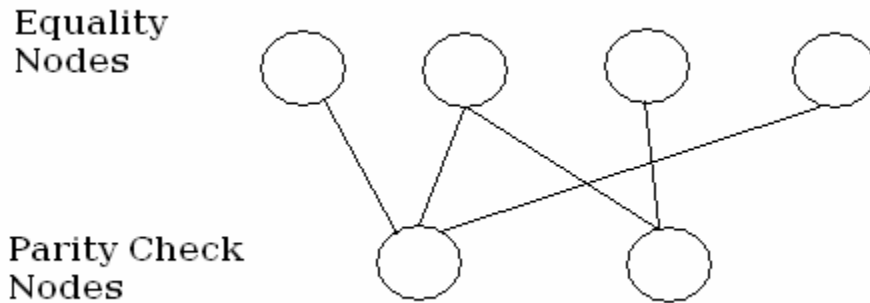
This entire process can be represented in the following way:



## Irregular LDPC's (and their specification)

In general, LDPC's can be improved by changing the arrangement of the edges between the equality nodes and the parity check nodes. This improvement can allow upwards of 49% of the bits to get erased by the channel, yet decoding will still be feasible. To begin thinking about changing the number of edges from each node, some notation must be introduced. We can define  $\lambda_i$  to be the fraction of edges that are connected to equality nodes with degree  $i$ . We will also define  $\rho_i$  to be the fraction of

edges that are connected to parity check nodes with degree  $i$ . The following diagram shows an example of an irregular LDPC graph:



In this graph,  $\lambda_1$  is equal to  $3/5$  because 3 out of the 5 edges are exiting equality nodes with degree one;  $\lambda_2$  is equal to  $2/5$ . All other  $\lambda_i$ 's are equal to 0 because there are no equality nodes of degree higher than 2. Similarly,  $\rho_2$  is equal to  $2/5$  and  $\rho_3$  is  $3/5$ , while all the other  $\rho_i$ 's are 0. This also implies that for a typical “3-6” LDPC, the following conditions hold:  $\lambda_i=0$  for  $i \neq 3$ ,  $\lambda_3=1$ ,  $\rho_i=0$  for  $i \neq 6$ , and  $\rho_6=1$ .

It is important to determine how many of the lambda's and rho's are free to change. We must introduce some constraints for two reasons. Firstly, they ensure that the lambda's and rho's will generate feasible graphs capable of decoding. Secondly, they will lower the number of free variables, making optimization easier. We will start off with  $L + R$  free variables, where  $L$  is the highest degree of the lambda's and  $R$  is the highest degree of the rho's. The first two simple constraints are:  $\sum_{i=0}^n \lambda_i = 1$  and  $\sum_{i=0}^n \rho_i = 1$ ,

where  $n$  is an arbitrarily chosen number representing the highest allowed node degree. These ensure that the probabilities add to 1. In addition,  $\forall i, \lambda_i, \rho_i \geq 0$ . This must be true so that there are no negative values of the lambda's or rho's; if there were, you could not construct a graph from them. Obviously, the number of edges out of the equality nodes must equal the number of edges into the parity check nodes, which is given by

$$\sum_i \frac{\rho_i}{i} = \beta \sum_i \frac{\lambda_i}{i},$$

where  $\beta$  is equal to 1 minus the rate of the code. We also must assure

that  $\lambda_1=0$  and  $\rho_1=0$ , because if this is not true, then the code may not work properly, or at all. This lowers the number of free variables to  $L + R - 2$ . We can remove two more free



variables by defining one of the lambda's and one of the rho's in terms of all the other ones, since their sums must be one:  $\lambda_2 = 1 - \sum_{i=3}^L \lambda_i$ ,  $\rho_2 = 1 - \sum_{i=3}^R \rho_i$ ; this lowers the free variables to  $L + R - 4$ . To lower the number of free variables to  $L + R - 5$ , you can combine these previous relations with  $\sum_i \frac{\rho_i}{i} = \beta \sum_i \frac{\lambda_i}{i}$  to determine that

$$\lambda_L = \frac{\frac{.5 + \sum_{i=3}^R \rho_i}{\beta} - .5 - \sum_{i=3}^{L-1} \lambda_i}{\frac{1}{L} - .5}. \text{ For simplicity, we can set } L = R \text{ and determine that in}$$

terms of  $D$ , the total number of dimensions – or free variables – in the problem,  $L$  and  $R$  are equal to  $(D + 5)/2$ . A final constraint that must be satisfied is the following inequality<sup>1</sup>:  $f(\lambda_i, \rho_k, \delta; x_j) ::= \delta \lambda(1 - \rho(1 - x_j)) < x_j$ ,  $x_j = \frac{\delta j}{N}$ ,  $j = 1, \dots, N$ , where  $N$  is the number of equidistant, discrete points on the interval  $(0, \delta]$ . According to the “Efficient Erasure Correcting Codes” paper, if at most a  $\delta$ -fraction of the codeword  $C$  is erased, independently and at random, there is a high probability that the decoding algorithm will terminate successfully.

Using various methods, radically different  $\lambda$ 's and  $\rho$ 's can be arrived at. Methods suggested by previously written papers include differential evolution, an algorithm that produces very good results. It has been suggested that the Nelder-Mead method might produce reasonable results, so this is the method that we are using.

---

<sup>1</sup> “Design of Efficient Erasure Codes”, equation 5

## The Nelder-Mead Method<sup>2</sup>

The Nelder-Mead method will allow us to try to find an ideal  $\lambda$  and  $\rho$  to decode and fix as many errors as possible. The Nelder-Mead method is typically applied to problems that are unconstrained, but in this case, there are all the constraints that were discussed in the previous section.

To use Nelder-Mead, we will try to maximize the value of  $\delta$  (or, since Nelder-Mead is a minimizer, we will minimize the value of  $-\delta$ ) in  $f(\lambda_i, \rho_k, \delta; x_j) < 0$  while trying to avoid violating any of the constraints. To deal with the constraints, we are going to use a penalty function for all solutions that are in violation of the constraints. This was a constant on the order of a billion, thereby almost never allowing infeasible points.

Another consideration is the starting values, because for Nelder-Mead to operate, there must be  $N+1$  starting points where  $N$  is the number of dimensions. In our case, this will be the value of  $L + R - 5$ . The points create a simplex (an  $n$  dimensional object with segments connecting all points and faces between all segments). The 2-dimensional simplex is a triangle; the 3-dimensional one is a tetrahedron.

The way that we created our starting simplex is as follows

- Create an initial feasible (valid) point using the following procedure<sup>3</sup>:
  - Set  $\lambda_3$  through  $\lambda_{L-1}$  equal to  $\frac{1}{L-1}$ .
  - Set  $sL$  equal to  $\frac{.5}{L-1} \sum_{i=2}^L \frac{1}{i}$ .
  - Set  $sR$  equal to  $\frac{1}{R-2} \sum_{i=2}^{R-1} \frac{1}{i}$ .
  - Define  $\rho_R$  to be  $\frac{sL - sR}{\frac{1}{R} - sR}$ .

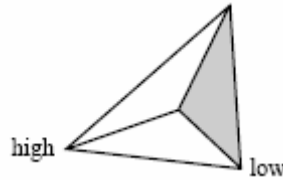
---

<sup>2</sup> C code for this method, and a description were supplied in “Numerical Recipes in C”

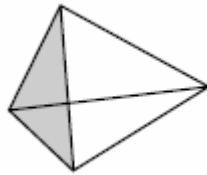
<sup>3</sup> Suggested by Prof. Dan Spielman

- Set  $\rho_3$  through  $\rho_{L-1}$  equal to  $\frac{1 - \rho_R}{R - 2}$ .
- Compute  $\lambda_2$ ,  $\lambda_L$ , and  $\rho_2$  using the constraints.
- This point corrects about a 30-35% bit erasure rate.
- Pick a random direction in the space, and travel from the initial feasible point in that direction until an infeasible point is reached. Add the last feasible point that was transversed to the simplex.
- Repeat the previous step until the simplex is completed (there are N+1 points in it).

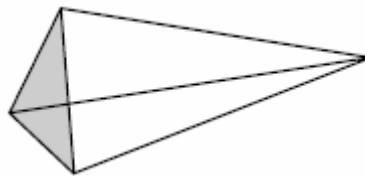
At each iteration, one of the following four things happens (generalized to 3 dimensions so it can be visualized). The starting stage might look like this:



In this case, high represents the worst point, based on the minimization, and low is the best. The first possibility is that the high point is reflected through all the other points in the simplex resulting in:



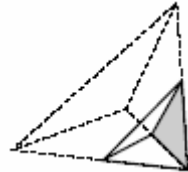
A second alternative is for the high point to be reflected and expanded through the simplex:



The third option is a contraction away from the high point:



The final choice is a contraction around the lowest point by all other points:



The option that is chosen is whichever gives the lowest value of the function that is trying to be minimized.

The method terminates when none of the choices improve the value of the function by more than a tolerance value that we set.

At the end of an individual Nelder-Mead run, the results that are produced are not very good. A reason for this could be that the method itself took an anomalous step at some point in the run. Our way of solving this was as follows: Upon completion of a Nelder-Mead run, we took the best point in the simplex to be our new starting point. We then perturbed this point into a new starting simplex, and ran the Nelder-Mead method again. This process was repeated until the optimization improved the function by less than a second tolerance value. To continue to improve the percentage of erasures that our code could cope with, we repeated this entire process for as long as we had time.

Our program displays (via a System.out) the current value of delta, which represents the percentage of erasures that can be corrected with the current polynomials.

## Results and Data

To standardize our results with the literature, our lambda and rho values will be shown as polynomials given by the generating functions:  $\lambda(x) = \sum_{i=2}^L \lambda_i x^{i-1}$  and

$$\rho(x) = \sum_{i=2}^R \rho_i x^{i-1}.$$

The first stage in producing high  $\delta$  values was an overnight run. During this run, the dimensions of the simplexes were randomly chosen from the odd integers between 33 and 55. (They were chosen to be odd so that the formula  $L = R = (D + 5)/2$  held with integer values of  $L$  and  $R$ .) Results of this first stage ranged from  $\delta = 0.45127$  to  $\delta = 0.49001$ . The highest value of delta was for a 49-dimensional run. This first stage allowed weak codes to be “weeded” out manually.

The second stage of my procedure was taking the polynomial for the best  $\delta$  value and using it as a new starting point. I eliminated all the other polynomials that I had generated. After each run of the Nelder-Mead algorithm, the new best  $\delta$  valued polynomial was taken to restart the method. In this way, the algorithm can be run indefinitely, beginning with an excellent polynomial, though with the improvement decaying somewhat per run.

The following is the best polynomial that we generated after stage two – it is in 49 dimensions, therefore,  $L = R = 27$ . This means that our code would correct errors on a message where over 49.12% of the bits were erased:

Polynomial coefficients	$\delta$ -value
$\lambda(x) = 0.0990299219062345x + 0.05322429856029543x^2 +$ $0.055277203574016234x^3 + 0.02844910724153615x^4 +$ $0.02623612386335452x^5 + 0.00974806698319024x^6 +$ $0.043625918242555586x^7 + 0.03731895663295148x^8 +$ $0.03031150462120532x^9 + 0.037309402010523x^{10} +$ $0.029597164315280516x^{11} + 0.021166600459527427x^{12} +$ $0.03415225624197976x^{13} + 0.041320608127954954x^{14} +$ $0.009263507615627463x^{15} + 0.018522044947541907x^{16} +$ $0.008887897283874187x^{17} + 1.8583708480587282E-6x^{18} +$ $0.0024302037195027106x^{19} + 4.875780862482407E-6x^{20} +$ $1.4847922748886244E-5x^{21} + 1.278790267836231E-4x^{22} +$	0.49123208

$1.8314221090588438E-6x^{23} + 4.255863080467144E-8x^{24} + 1.7067227839464666E-5x^{25} + 0.4139608113430263x^{26}$ $\rho(x) = 7.632676934932192E-9x + 8.540515922834572E-9x^2 + 1.347032203221241E-7x^3 + 0.018706481130486223x^4 + 0.14176365914019248x^5 + 0.048483176456814533x^6 + 0.018207299247641195x^7 + 0.0019723149738321006x^8 + 0.03351827937913176x^9 + 3.5897497511212437E-4x^{10} + 2.192759770277235E-4x^{11} + 3.094560606056156E-4x^{12} + 3.6525537046007547E-4x^{13} + 6.241422929251609E-4x^{14} + 0.0024904768128954646x^{15} + 0.0010594188309099165x^{16} + 2.447810869179167E-4x^{17} + 0.0019071855087696428x^{18} + 9.728748941992965E-4x^{19} + 0.0035607085025847664x^{20} + 8.074336119955476E-4x^{21} + 0.002270527070515199x^{22} + 0.0014808462429242576x^{23} + 3.4174652757254845E-4x^{24} + 0.005969329475817297x^{25} + 0.714366205554256x^{26}$	
--	--

The next section will be devoted to the analysis of this above polynomial.

After the first stage, the above polynomials had the following values:

Polynomial coefficients	$\delta$ -value
$\lambda(x) = 0.09913944229931859x + 0.05276368935878339x^2 + 0.055972280575682924x^3 + 0.029674260338107834x^4 + 0.017260488750503292x^5 + 0.025448559738954105x^6 + 0.04515638495454221x^7 + 0.04030333756774207x^8 + 0.023131373103644444x^9 + 0.033836826128835755x^{10} + 0.024174237516130185x^{11} + 0.010565020729052915x^{12} + 0.015051741615809872x^{13} + 0.022709753544240166x^{14} + 0.008548520034345693x^{15} + 0.015265273263572396x^{16} + 0.02854094895193963x^{17} + 0.02263104851684256x^{18} + 0.013927898276636303x^{19} + 0.07122136853087355x^{20} + 0.02153165264315276x^{21} + 0.040808253617859384x^{22} + 0.010714065924932058x^{23} + 0.002728620676797656x^{24} + 0.015153512175180905x^{25} + 0.2537414411665194x^{26}$ $\rho(x) = 3.6513918555414193E-9x + 5.062364731594058E-7x^2 + 0.00574684527124305x^3 + 0.028838659620213876x^4 + 0.11906353643723967x^5 + 0.0480945379927659x^6 + 0.022608297965228147x^7 + 0.0010237641058132966x^8 + 0.0336234867070067x^9 + 1.5748300415421974E-6x^{10} + 0.003049627548126765x^{11} + 0.001942286492248885x^{12} + 7.533265390057562E-5x^{13} + 2.917591878192492E-7x^{14} + 8.144447734694117E-4x^{15} + 0.001000888374498451x^{16} + 6.469655073957167E-4x^{17} + 7.214491255120121E-4x^{18} + 0.0014739803468346929x^{19} + 0.0026517581811077x^{20} + 0.0026718342756574406x^{21} + 0.0026424921242295726x^{22} + 0.0011782712186155905x^{23} + 1.1747323851536007E-4x^{24} + 0.007483472621880171x^{25} + 0.7145282189414026x^{26}$	0.49001044

For comparison, the following are the polynomials for the second best delta value before the “weeding” process (this is in 43 dimensions):

Polynomial coefficients	$\delta$ -value
$\lambda(x) = 0.11274587736727337x + 0.06498455715355583x^2 + 0.03969428250127628x^3 + 0.048961038412381516x^4 + 0.028189656203807884x^5 + 0.03628565340232763x^6 + 0.039921651057134x^7 + 0.048561615545643276x^8 + 0.01035366914158521x^9 + 0.03930103848736154x^{10} + 0.01204419343378802x^{11} + 0.0026297657792068107x^{12} + 0.007466072446663316x^{13} + 1.1199455054028839E-6x^{14} + 8.821403829892515E-5x^{15} + 1.7479984313786059E-4x^{16} + 0.03459305011604132x^{17} + 0.021583402329676413x^{18} + 0.004596632861824698x^{19} + 0.044462415543915126x^{20} + 0.04567687942127912x^{21} + 0.12447510904948544x^{22} + 0.23320930591883104x^{23}$	0.48937541
$\rho(x) = 2.3126878301304998E-7x + 1.1951953253748004E-8x^2 + 2.2733910865119922E-4x^3 + 0.030190568328413908x^4 + 0.1772726933792846x^5 + 0.016180678882956462x^6 + 0.0060195214101926246x^7 + 0.003991365160585282x^8 + 0.01969672231078712x^9 + 0.004628767552064218x^{10} + 0.001422882353708355x^{11} + 0.008258629068767844x^{12} + 0.01995513667205749x^{13} + 9.895079953920287E-4x^{14} + 0.0017074274635243986x^{15} + 2.6541462280539835E-4x^{16} + 1.345166853321739E-6x^{17} + 0.0014414245548813587x^{18} + 0.0019873824492244265x^{19} + 0.0010053649143326932x^{20} + 1.205146724011483E-4x^{21} + 7.158366228945121E-4x^{22} + 0.7039212340894853x^{23}$	

Given time, this polynomial could have had stage two applied to it; in which case, the delta value could have been considerably higher.

## Analysis

There are two main ways for us to analyze our  $\lambda$ 's and  $\rho$ 's.

### 1. Theoretically

- a. To determine whether a codeword is capable of being decoded after having passed through the erasure channel with probability of erasure  $p_0$ , we need to make sure that the plot of  $\rho(x) = \sum_i \rho_i (1 - (1 - x)^{i-1})$  lies

above the plot of  $\lambda(y) = p_0 \sum_i \lambda_i y^{i-1}$  at all points<sup>4</sup>. These two formulas have been derived for us in Lecture Notes for lecture 17. This is an extremely accurate way of determining whether all the errors can be corrected.

## 2. Experimentally

- a. This can be done simply, by using our  $\lambda$ 's and  $\rho$ 's to create a random encoding matrix. Using this matrix, we can encode some codewords to send over the erasure channel with various probabilities of erasure. Upon running the decoding algorithm, it would not be hard to tell if the decoder was capable of decoding at a given probability of erasure.

For the sake of this paper, I shall test the polynomials using the first method. In addition, I will make use of the program's outputted value of delta. This should be the first probability where the lines cross.

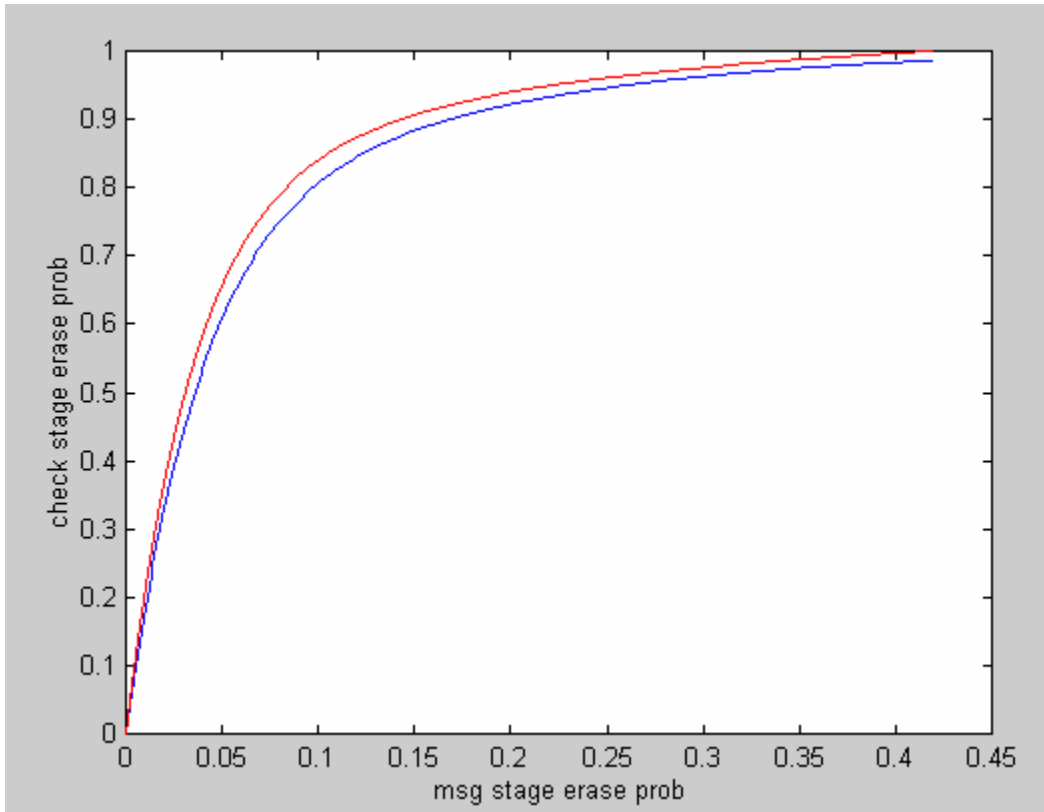
The polynomial that we generated can correct messages with up to a 49.12% erasure rate, as outputted by the program.

Here is the graph of  $\lambda(x)$  and  $\rho(x)$  at  $p = .42$ , the value where a normal LDPC begins to fail:

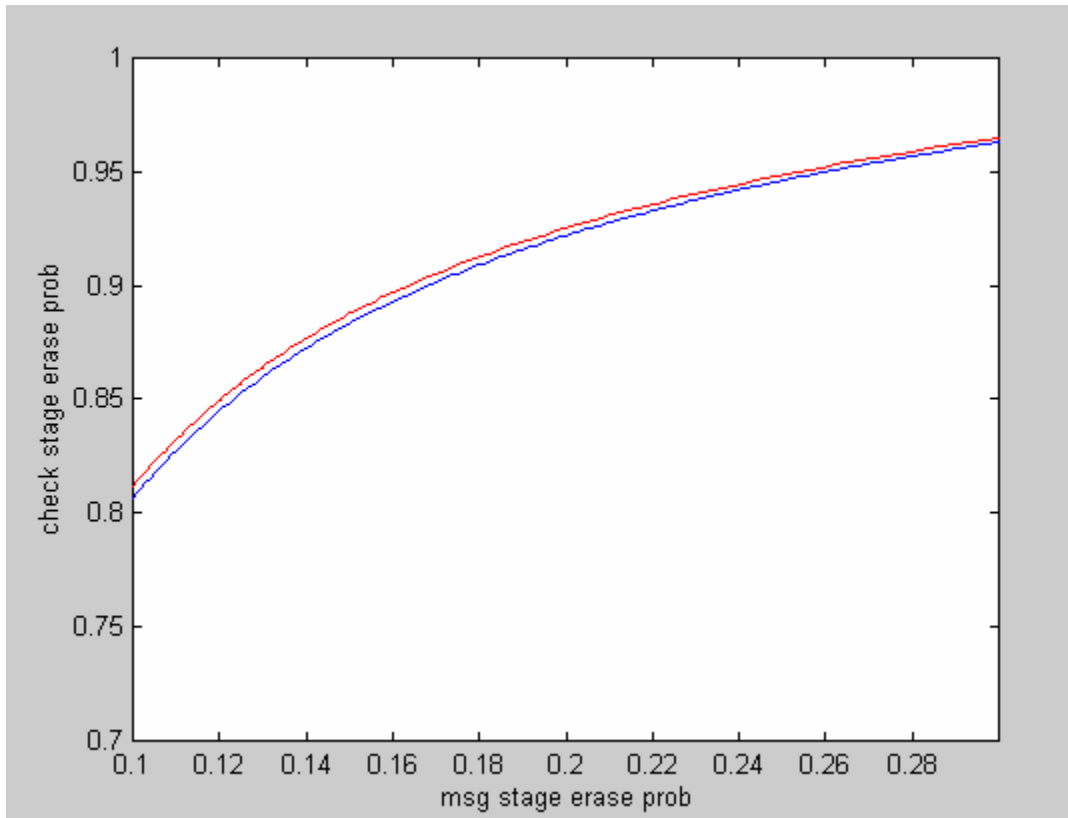
---

<sup>4</sup> Lecture Notes: Lecture #17

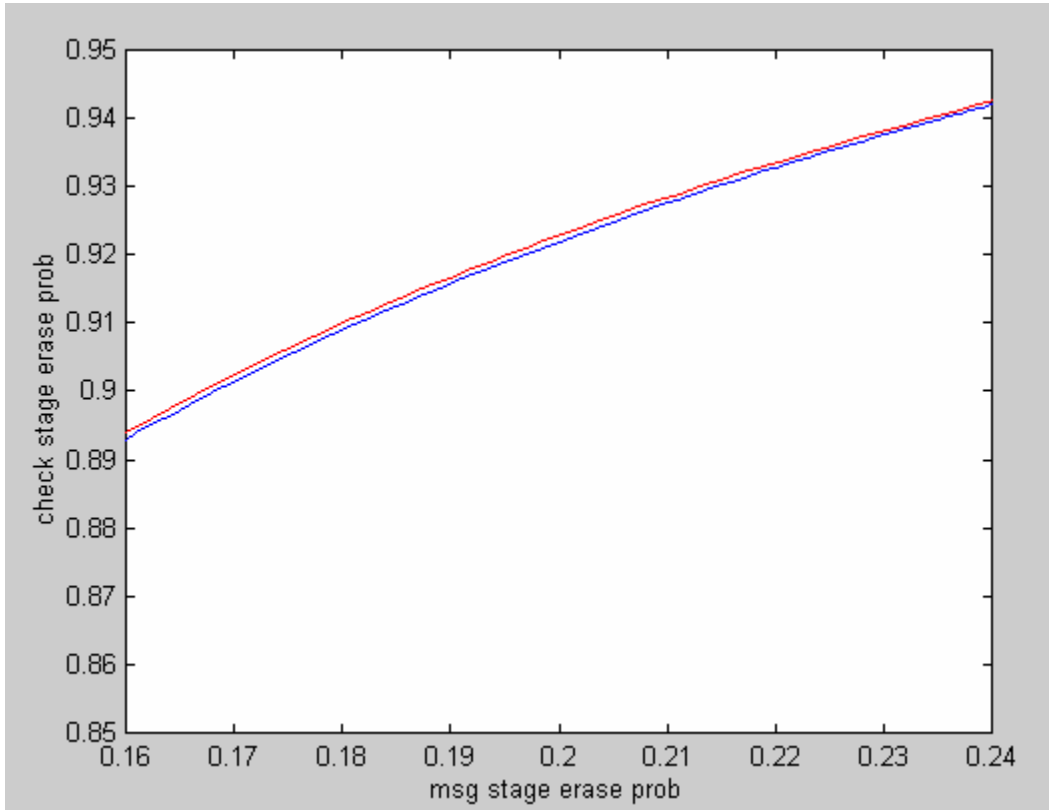




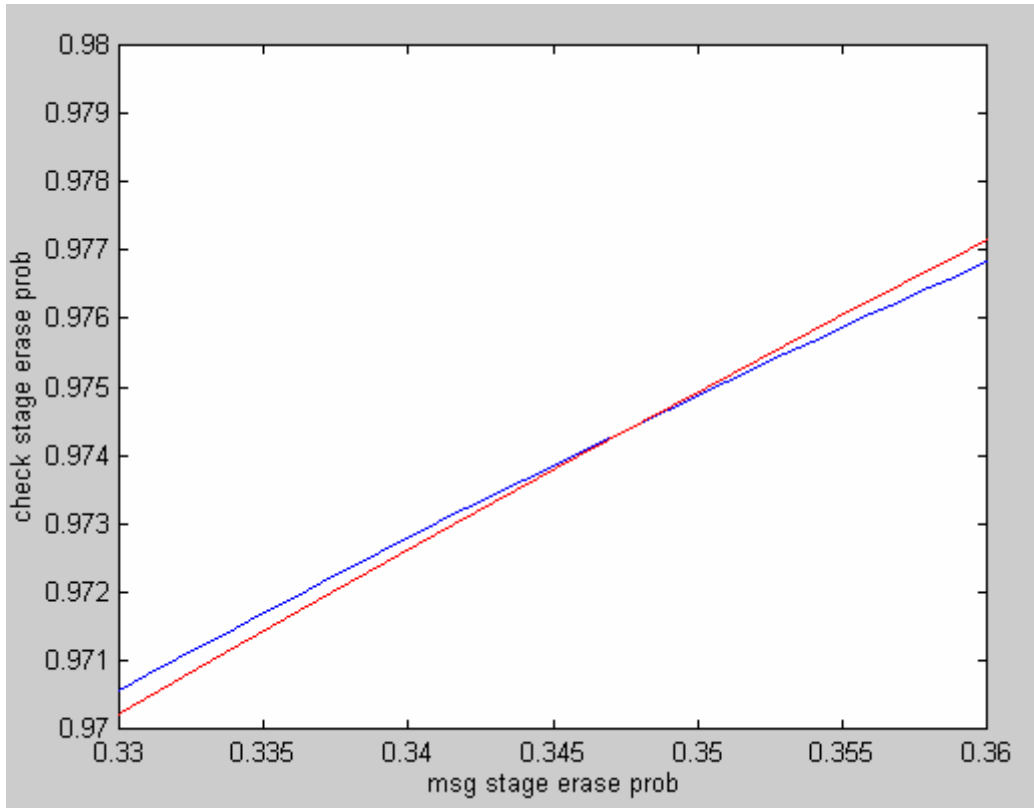
Here is the graph at  $p = .48$  (zoomed in on the closest point):



Here is the graph at  $p = .49$  (again zoomed in, this time even further), a successful value, yet somewhat close to the failure point:



Finally, here is the graph at  $p = .495$ , a value at which correction will fail; the lines cross:



## Conclusion

Given an ample amount of time to run the algorithm, I believe that the Nelder-Mead method will produce extremely good polynomials for Low Density Parity Check coding. At the time that I had to stop the algorithm to compile the results, delta was still increasing by a few thousandths of a percent per iteration.

The polynomials that our program did generate are capable of correcting more than a 49.12% bit erasure rate. This value is better than the linear programming approach that was used in the “Design of Efficient Erasure Codes with Differential Evolution” paper; none of these codes had a delta greater than 48.86%. Our codes do not appear to be as good as those that used differential evolution. This may not be the case, as the differential evolution codes were run on the order of 2,000,000 iterations, while ours only ran a few hundred thousand iterations. This is especially true because the Nelder-Mead

method is known to be one of the slowest optimization methods<sup>5</sup>. Another way for us to improve our codes would be allowing infeasible points during the iterations, and then filtering them out before outputting the polynomials. Finally, a different algorithm for determining the starting simplex could give drastically different results. Our starting simplex, though in the middle of the dimensional space, could only correct in the mid 30%'s erasure rate.

## References

Luby, Mitzenmacher, Shokrollahi, Spielman. Efficient Erasure Correcting Codes. IEEE Transactions on Information Theory, Vol 47, No 2. February 2001.  
Press, et al. Numerical Recipes in C. Cambridge Press. 1992.  
Shokrollahi, Storn. Design of Efficient Erasure Codes with Differential Evolution.  
Spielman. 18.413 Error Correcting Codes Lecture Notes.

## Appendix: Java Code

Our Java program's source code is attached.

---

<sup>5</sup> Numerical Recipes in C