LDPC Codes: Design and Shortcomings

Kenneth Andrews

March 26,2001

1 Introduction

The literature consistently reports that LDPC codes do not have an error floor, but my simulations always find one. Curiously, researchers report no difficulty in this matter, and yet never provide sufficient information to duplicate their results. Feeling frustrated, this memo is intended to concisely document my algorithms so others can consider them and provide comments.

2 LDPC Decoding Algorithm and Clipping

Let the transmitted symbols be $x \in \{+1, -1\}$ and the received symbols be $y = x + n$ where n is a zero-mean Gaussian random variable with variance σ^2. The standard decoding equations are:

$$v_j = w + \sum_{i \neq j} u_i$$

$$u_j = 2 \tanh^{-1} \left(\prod_{i \neq j} \tanh \frac{v_i}{2} \right)$$

where v_j and u_j are the messages sent from the variable nodes and from the check nodes along edge j, respectively. When using double precision arithmetic, $\tanh^{-1}(1 - 5.56 \times 10^{-17}) = 18.715$, and for arguments closer to 1, it evaluates to infinity. Hence messages u_j are constrained in magnitude to $\pm 2(18.715) = \pm 37.43$.

We may improve this situation with a change of variables. Let

$$\bar{w} = e^w, \bar{u}_j = e^{u_j}, \text{and} \bar{v}_j = e^{v_j}.$$

Then

$$\bar{v}_j = \bar{w} \prod_{i \neq j} \bar{u}_j$$

$$\bar{u}_j = \mathop{\#}_{i \neq j} \bar{v}_i \text{ where } x \# y \triangleq \frac{1 + xy}{x + y}$$

Because the largest finite value in double precision arithmetic is about 1.797×10^{308} and the evalution of $\bar{v}_x \# \bar{v}_y$ requires computing the product $\bar{v}_x \bar{v}_y$, messages $\bar{v}_j$ are constrained

to $\sqrt{1.797 \times 10^{308}} = 1.34 \times 10^{154}$. A similar constraint applies to small numbers to avoid denormalized double precision values, so we restrict $\bar{v}_j$ to be larger than the reciprocal of this number as well. In the original variables, v_j is effectively constrained to $\pm\log(1.34\times10^{154}) = \pm354.89$. While this constraint applies to v_j rather than u_j, it clearly provides a substantially larger range than the standard equations. Simulations of Gaussian density evolution behavior confirms this analysis.

3 Parity Check Matrix Generation

The behavior of the code is determined by the parity check matrix, so the algorithm used to construct it is critically important. I've done my best to duplicate the work of [1]. Richardson, Shokrollahi, and Urbanke state on page 624 that for frame length 10^4, the maximum variable degree used is 20. From their Table II on the same page, they list their suggested degree distribution with this maximum degree for the BIAWGNC (Binary Input Additive White Gaussian Noise Channel), and I used this distribution. The authors give some further clues on the H matrix construction:

> For length 10^3, the error rates are given for systematic bits. (A specific encoder was constructed.) For length 10^4 and above, the error rate is given over all of the bits in the codeword. These graphs were not chosen entirely randomly. The degree-two nodes were made loop-free for lengths less than 10^6 and, in the length 10^3 case, all of them correspond to nonsystematic bits. ... For shorter lengths [$< 10^6$] some small loop removal was performed.

My H matrix generation algorithm performs all of these optimizations as follows. First, sequences of variable nodes and check nodes are generated. The edge degree distributions in [1] are translated into node degree distributions, multiplied by the block length N, and rounded as required to match the sums and first moments of the edge and node degree distributions. A set of variable nodes are generated with these degrees and put in random order; the same is done for the check nodes. The result is a sequence of variable nodes and a sequence of check nodes, each node with a set of "sockets" where edges are to be attached.

Next, the degree-2 variable nodes are connected to check nodes in an entirely loop-free way. Imagine that each check node is an island, each with a unique name. The first degree-2 variable node is taken, two check node sockets are selected at random, and if they belong to different islands, edges are drawn to the variable node. These edges connect the islands (via the variable node) into one island, so the old names are discarded and they are given one new unique name. The next degree-2 variable node is taken, this island-connecting process is repeated, and so on. When complete, the resulting "degree-2 subgraph" is a forest, i.e. it consists of one or more trees. These edges remain fixed throughout the rest of the design process.

Next, the remaining edges are placed and parallel edges (length-2 loops) are removed. The sequence of remaining variable node sockets are connected to the remaining check node sockets by picking a random permutation. Parallel edges are identified (by sorting the edges according to the variable node and check node each connects and finding matches), and all but the first edge of each parallel set is flagged for rerouting. To increase the number of

options, a few more ($0.01N+5$) of the edges placed in this paragraph (those not connected to degree-2 variable nodes) are also flagged. The flagged edges are deleted, and a new random permutation is used to reconnect the now-vacant sockets. This process is repeated until no parallel edges remain.

Finally, length-4 loops are removed. Length-4 loops are identified by computing HH^T and looking for off-diagonal entries $M_{i,j}$ greater than 1 (each entry represents $\frac{1}{2}\binom{M_{i,j}}{2}$ length-4 loops). For each loop found, one of the four edges is selected; if a degree-2 variable node is involved, one of the other two edges is selected. One other edge in the graph (with terminal nodes different from that of the selected edge, and not connected to a degree-2 variable node) is also selected at random, and the check node connections of the two are reversed. When complete, HH^T is recomputed and the process is repeated until no more length-4 loops are found.

Because this algorithm and its implementation has proven error-prone, all claimed properties have been verified by entirely different methods. First H is constructed explicitly (most of the design algorithm does not use H, but operates on lists of edges and their terminal nodes). Degree distributions are verified using histograms of the row and column sums of H. Parallel edges are trivially detected by finding entries ≥ 1. The degree-2 subgraph criterion is verified by taking only the columns of H with sum 2, and iteratively pruning the leaves of this subgraph by alternately deleting rows with sum 1 and columns with sum 1. If this process fails to terminate with a null matrix, then a loop must exist. Finally, for each variable node, the length of the smallest loop to which it belongs is found (by an algorithm like that used to determine $d_{\min}$ from a convolutional code's trellis), and verified to be greater than 4.

4 LDPC Encoder

In [1], the authors did not construct an encoder for codes longer than 1000 symbols. For the codes presented here, of length up to $10,000$ symbols, I have found no problem constructing a low-complexity encoder using Algorithm CHT from [2]. The result is a parity check matrix H' derived from H by a permutation of its rows and columns, and a systematic encoder with linear-time encoding complexity. The algorithm was implemented in such a way that all degree-2 variable nodes are placed in the parity portion of H'.

5 Simulation Results

LDPC codes and encoders of rate 1/2 with block lengths $N = 10^3$ and $N = 10^4$ were generated, and simulations were run, using the algorithms described in the previous sections. Encoded random binary messages were used because biased decoders may generate erroneous statistics when run exclusively with the all-0 codeword. The equations used by the decoder are not exactly symmetric (due to details of the internal floating point arithmetic) so this concern must be taken seriously, though simulations have found no measurable bias. A stopping rule was used to increase the average decoder speed, which terminated the iterative decoding when a codeword was found.

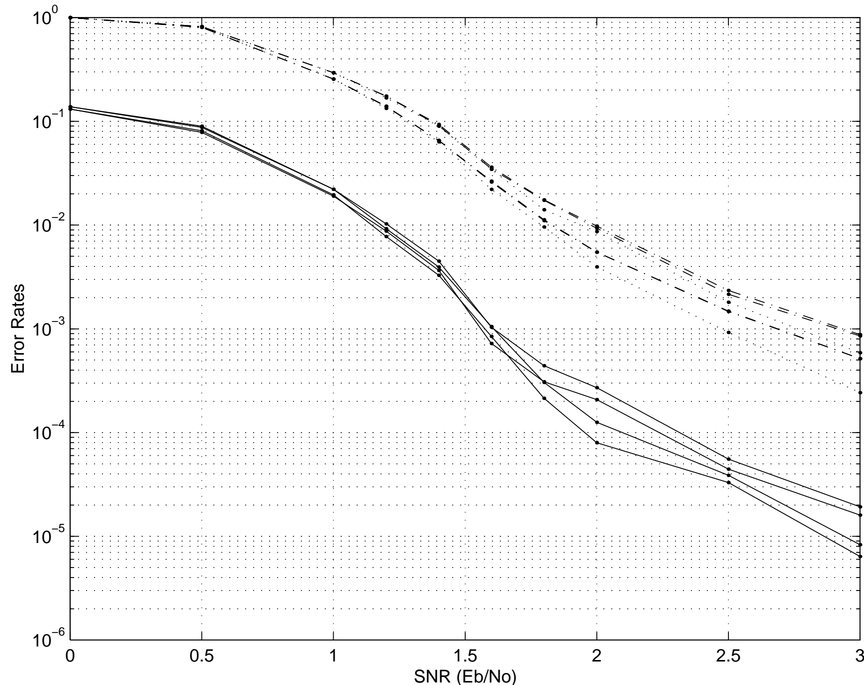

Figure 1: (N=1000, K=500) LDPC code, 50 and 200 iterations

Figure 1 shows performance curves for a code of length $N = 1000$ with a maximum of 50 and 200 iterations; tables 1 and 2 show the raw data. From top to bottom, the solid curves show symbol error rate (SER) with 50 iterations, bit error rate (BER) with 50 iterations, SER with 200 iterations, and BER with 200 iterations. Note that the BER curves are a little better than the SER curves because symbol errors are more likely to occur in the parity portion of the codeword than in the systematic portion. From top to bottom, the dashed curves show codeword error rate (CER, the fraction of decoded frames that have one or more code symbols in error) with 50 iterations, frame error rate (FER, the fraction of decoded frames that have one or more message bits in error) with 50 iterations, and the virtually coincident CER and FER curves with 200 iterations. The dotted curves show the detected codeword error rate (DCER, when the LDPC decoder completes its maximum number of iterations without finding any codeword), again for 50 and 200 iterations. The significant gap between the DCER and CER curves show that many of the codeword errors occur when the decoder finds a valid but incorrect codeword.

Figure 2 shows performance curves for a code of length $N = 10,000$ with a maximum of 50 and 200 iterations; tables 3 and 4 show the raw data. The curves are ordered in the same way as those for the shorter frames, and show the same trends.

6 Shortcomings

The most notable characteristic of the simulations is a sharp error floor. Analysis shows that in part, this is due to low weight codewords, despite David MacKay's statements, "Gallager

dB	blocks	blk_ers	bit_ers	cw_ers	sym_ers	fails
0.000000	100	100	6572	100	13824	100
0.500000	122	100	4780	100	10660	100
1.000000	340	100	3227	100	7529	100
1.200000	571	100	2496	100	5860	96
1.400000	1101	100	2015	103	4940	99
1.600000	2870	100	1036	104	2970	99
1.800000	5760	100	890	101	2542	81
2.000000	10735	100	1115	105	2903	93
2.500000	46514	100	1031	109	2577	84
3.000000	117313	100	939	103	2257	69

Table 1: (N=1000, K=500) LDPC code, 50 iterations

dB	blocks	blk_ers	bit_ers	cw_ers	sym_ers	fails
0.000000	100	100	6569	100	13840	100
0.500000	124	100	5032	100	11138	100
1.000000	391	100	3821	100	8674	100
1.200000	720	100	2790	101	6634	96
1.400000	1527	100	2508	100	6018	97
1.600000	3850	100	1624	102	4045	85
1.800000	9055	100	969	102	2761	87
2.000000	18212	100	725	100	2287	72
2.500000	68152	100	1122	101	2641	63
3.000000	193690	100	618	100	1608	47

Table 2: (N=1000, K=500) LDPC code, 200 iterations

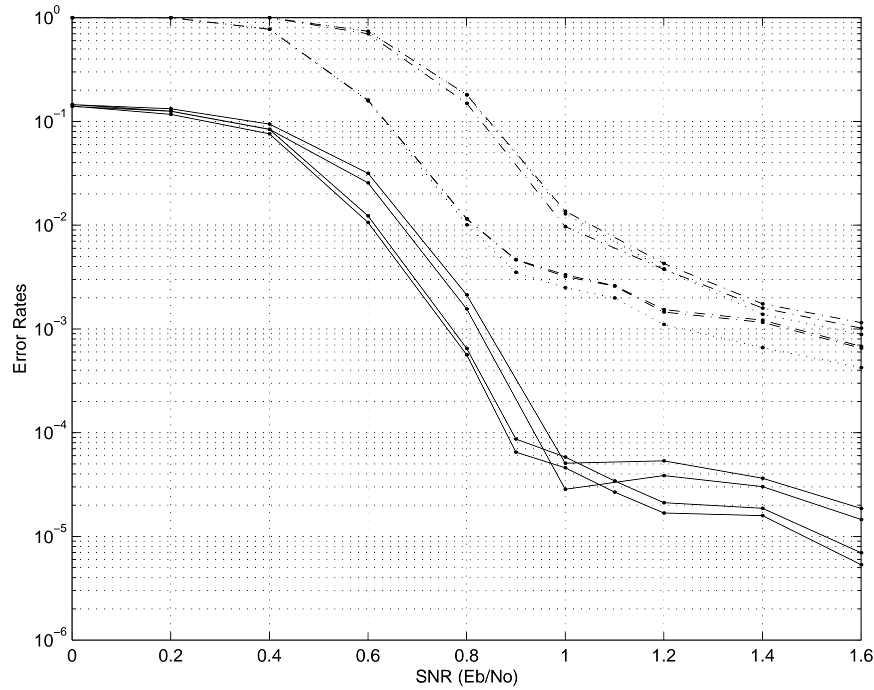

Figure 2: (N=10,000, K=5000) LDPC code, 50 and 200 iterations

dB	blocks	blk_ers	bit_ers	cw_ers	sym_ers	fails
0.000000	100	100	69800	100	145085	100
0.200000	100	100	62687	100	132409	100
0.400000	100	100	42044	100	94341	100
0.600000	143	100	18303	106	45213	106
0.800000	668	100	5202	121	14221	120
0.850000	1464	100	2534	137	8533	137
0.900000	2976	100	2556	155	8693	153
0.950000	5318	100	2845	138	8015	131
1.000000	10319	100	1468	141	5236	133
1.050000	14530	100	3248	131	8948	119
1.100000	19144	100	6921	123	17091	106
1.150000	24684	100	7963	122	19156	108
1.200000	26679	100	5138	114	14250	101
1.400000	62902	100	9490	110	22847	87
1.600000	98300	100	7151	113	18307	87

Table 3: (N=10,000, K=5000) LDPC code, 50 iterations

dB	blocks	blk_ers	bit_ers	cw_ers	sym_ers	fails
0.000000	100	100	70182	100	145255	100
0.200000	100	100	58490	100	125637	100
0.400000	129	100	49067	100	108058	100
0.600000	629	100	33408	101	77360	99
0.800000	8727	100	24552	101	56645	88
0.900000	21598	100	7001	101	18769	76
1.000000	31259	100	7171	104	18194	78
1.100000	38672	100	5176	101	13281	77
1.200000	68843	100	5789	106	14531	76
1.400000	86454	100	6850	105	16112	57
1.600000	153325	100	4084	104	10660	65

Table 4: (N=10,000, K=500) LDPC code, 200 iterations

codes do not typically show such an error floor" [3], "Gallager codes, in contrast [to turbo codes], show no such error floor, and it has been proved that they have asymptotically good distance properties." [4], and "In all our experiments with Gallager codes of block length greater than 1000 and column weight at least 3, undetected erorrs have never occured" [5]. Other researchers (Hui Jin, Flarion published curves, [1]) also claim without providing conclusive results that LDPC codes either have no error floor, or that it is very far down.

It is not clear how to systematically identify low weight codewords in an LDPC code. One method is to use an LDPC simulation, with the all-0 codeword and the SNR set near the low end of the error floor, and log undetected decoder errors, i.e. where the decoder selected an incorrect codeword. This procedure, applied to the $(N = 10000, K = 5000)$ rate 1/2 LDPC code used in the simulations, identified one codeword of weight 9, one of weight 11, two of weight 12, and one of weight 18. Because the decoder was only run for about 18 hours, there may be more unidentified codewords with weights of 12 or above, but probably no unidentified ones of smaller weight. This distribution of low weight codewords depends on the particular random code constructed, but this result appears to be typical. The $(N = 1000, K = 500)$ code has codewords of weights 7,10,11,12,13,14,16,17,19, and 21, and may have others of low weight.

The low weight codewords analyzed so far all involve two variable nodes of degree 3 and all remaining variable nodes of degree 2. The two degree-3 nodes are connected to six check nodes of course, and those check nodes are interconnected by three chains built from degree-2 variable nodes. Various modifications to the LDPC code design algorithm could probably prevent patterns of this particular type, potentially lowering the error floor. Alternatively, the check matrix could be built without using any degree-2 variable nodes, with a probable penalty in E_b/N_0 threshold.

Unfortunately, these modifications would lower but not eliminate the apparent error floor, as illustrated by the DCER curves. The undetected codeword errors cause very few symbol errors, so these modifications would likely have very little impact on the SER curves (this cannot be stated with certainty because the presence of low weight codewords might "confuse" the decoder and thereby affect its more general convergence behavior as well).

Perhaps the SER curves could be lowered by performing more iterations with these frames, but extrapolation from the 50 and 200 iteration results is not encouraging.

References

[1] Richardson, Shokrollahi, Urbanke, *Design of Capacity-Approaching Irregular Low-Density Parity-Check Codes*, IT Trans, Feb 2001, pg. 619.

[2] Richardson, Shokrollahi, *Efficient Encoding of Low-Densith Parity-Check Codes*, IT Trans, Feb 2001, pg. 638.

[3] MacKay, *Gallager Codes - Recent Results*, ftp://www.inference.phy.cam.ac.uk/pub/mackay/sparsecodes

[4] MacKay, *Turbo Codes are Low Density Parity Check Codes*, http://www.inference.phy.cam.ac.uk/mackay/turbo-ldpc.ps.gz

[5] MacKay, Wilson, Davey, *Comparison of Constructions of Irregular Gallager Codes*, http://www.inference.phy.cam.ac.uk/mackay/turbo-ldpc.ps.gz