

Project 1

In this project, you are going to implement the simplest low-density parity-check codes, and see how well they work.

Due: Thursday, April 1, 2004

This project has three phases

- Implementation
- Generating Data
- Explaining the data and implementation

I'll explain what you should do for each in turn.

1 Implementation

Your implementations will have six parts:

- A. sparse graph generator
- B. encoder
- C. Channel simulators,
- D. decoder,
- E. evaluator that compares the output of the decoder to the input to the encoder (or output of the encoder).
- F. code to drive all these

These parts should all be separable so that you can test each individually. This will also allow you change components. Many of you will be using parts of these components in your final projects.

Let's examine each in more detail.

A. sparse graph generator

A low-density parity-check code is specified by a sparse graph. You should create code that generates this graph. In this project, it will be a graph between n bit nodes of degree 3 and $n/2$ check nodes of degree 6. n should be 10,000 if possible, and 3,000 if that is too slow. Note that very different results may be seen for different n . Your code should save this graph so that other components may pick it up. You should also save the graphs you use so that your experiments can be reproduced. If you use matlab, you will find the `save` command useful for storing data. (type `help save` to learn more) The functional form of `save` is especially useful (like `save('graph.dat', 'G')`)

If you code in C, don't use the routine `rand()` to generate your random graph. The resulting graph turns out not to be very random. (and probably won't work for our purposes) Try instead the pseudo-random number generator I posted on the course web page.

If you use Matlab, you might find the commands `repmat` or `randperm` useful.

B. encoder

To encode a LDPC code, it is easiest to make a generator matrix that is dual to your low-density check matrix. You can then generate a random codeword by multiplying a random vector by the matrix. It would be best if the matrix were systematic (contain the identity matrix as a sub-matrix) so that the message bits appear in the codeword. The most natural way to compute the generator matrix is to observe that it is a dual of the check matrix, and then compute it using my code `FormDualMod2.c`. This code is described in a separate document. You can either use `FormDualMod2.c` as a C program that can be called from matlab, or you can rip routines from it to incorporate into you own C code.

It will take a lot of time to encode your LDPC codes. So, you might want to initially test your code on the all-0 or all-1 codewords (note that both of these will be codewords). But, to generate reasonable results, you should really do tests on random codewords. That said, generating codewords is so expensive that it would not make sense to generate a fresh codeword for each test. My suggestion would be to generate a small set of codewords, say 10 to 50, save them, and use them for your experiments. In this case you should be sure to cycle through each codeword the same number of times in each experiment. That is, if you are doing 1000 runs at 1.4 dB, then you should use each codeword the same number of times in these experiments. Some of you will note that if you use an ideal implementation of the decoder, then your simulations will give the same performance if you start with random codewords or the all-0 codeword. However, numerical errors could skew the results of your simulation in unpredictable ways.

C. channel simulators,

We'll test these codes on 3 different channels: the Binary Erasure Channel with erasure probability p , the Binary Symmetric Channel with crossover probability p , and the Gaussian channel with ± 1 signaling. You should design a separate module for each that takes as input a codeword and outputs the result of passing the codeword through the channel. At some point (here or in the decoder), you will need to estimate the probability that each codeword bit was 1 (or 0) given the output of the channel. If you like, you may make this output the result of the channel simulator. For example, if you are transmitting over the BSC_p , and the channel outputs 0, 0, 1, then you could report this as $p, p, 1 - p$.

D. decoder

This is where most of the work will be. The decoder should take as input the graph defining the check matrix of the low-density parity-check code, as well as the output from the channel. The decoder should then output a guess for the value of each bit of the codeword (or, just a guess for each message bit).

Let me re-state how the decoder should work. The decoder proceeds in stages, and at each stage it passes a message along the edges of the graph. It begins with an initial stage, alternates between check node and bit node stages, and finishes with a terminal stage.

- **Initial stage:** in the initial stage, each bit node sends a message along each of its outgoing edges indicating the probability that the bit at that node was 1 given the output of the channel for that bit.
- **Check node stage:** in a check-stage, each check node receives a message coming in on each of its edges, and sends a message back out on each edge. Let's say that a check node is a neighbor of bit nodes m_1, \dots, m_6 , and that it receives the messages p_1, \dots, p_6 . Let q_1, \dots, q_6 denote the messages that the check node sends out. Then, q_1 should be the probability that bit b_1 is 1 given that the sum mod 2 of b_1, \dots, b_6 in the original codeword was 0, that the probability that b_i equals 1 is p_i , for $i \geq 2$, and that these probabilities are independent. (while the values we have for p_2, \dots, p_6 might not actually represent independent events, we define our calculation by assuming that they are independent). That is q_1 is a function of p_2, \dots, p_6 but not of p_1 . The other q_i 's are similarly defined. If you study the computations for q_1, \dots, q_6 , you should find that it is possible to compute these jointly much faster than computing each one at a time (*i.e.*, their expressions share common terms).
- **bit node stage:** in a bit node stage, each bit node receives a message coming in on each of its edges, and sends a message back out on each edge. Let's say that a bit node represents bit b , is a neighbor of check nodes c_1, \dots, c_3 , and that it receives the messages q_1, \dots, q_3 . Let p_1, \dots, p_3 denote the messages that the bit node sends out. Also, let q_0 denote the probability that the message bit itself is 1, given the output of the channel for that bit. (yes, we used that information in the initial stage, and we'll use it again here). We will compute p_1 to be the probability that bit b is 1 given the observations q_0, q_2, q_3 . That is, we imagine that b was transmitted in the (1,4) repetition code, that q_0, q_2, q_3 are the channel outputs from three of the repetitions, and we transmit the maximum likelihood estimator obtained from these three estimates as p_1 . We similarly compute p_2 and p_3 , again using the rule that p_i is a function of all the q s by q_i . As with the check-node stage, it should be possible to compute these much faster jointly than individually.
- **terminal stage** A terminal stage should follow a check stage, and is used to make the final guesses about the values of the bits. It does this by taking *all* incoming messages for each bit, say q_1, q_2, q_3 and the output of the channel for that bit, q_0 , assumes that these are independent observations of the bit, and uses them to make the best estimate of the value of the bit. That is, we assume that the bit was transmitted using a repetition code. This is just like what we do in a bit-node stage, except that we don't exclude any of the messages. In fact, this operation can be incorporated into a bit node stage.

The output of the decoder should be a vector of 0s and 1s consisting of the guesses.

There are many ways to decide how many rounds of this algorithm to run. One way is to just fix some number (make it at least 20). Another is to incorporate a terminal stage into each bit-node stage, and check how many bits are changing value with each iteration. One could decide to stop when none, or very few, change for a few rounds. You should do some experiments to get a feel for what works.

While I have described the messages being passes along the edges as probabilities, you should feel free to encode them however you like (e.g., as likelihoods, log likelihoods, log probabilities, etc.)

One thing to be careful of when implementing the decoder is that many of the probabilities will approach 0 or 1. However, this may cause division-by-zero errors, or trouble if you ever take a logarithm of 0. There are many ways to circumvent these problems. One is to force all probabilities to lie between ϵ and $1 - \epsilon$ for some $\epsilon > 0$ that you pick. I'm sure you can think of other ways as well. There will often be a tradeoff between speed and precision in how you handle this.

- E. **evaluator**: this will compare the output of the decoder to the input to the encoder (or output of the encoder), and compute the word error rate and bit error rate. If you used a systematic generator matrix, then you should just score bit errors on the message (systematic) bits.

If you did not use a systematic generator matrix, then count the fraction of bit errors among all the bits. There is some theory that indicates that this value shouldn't be all that different.

- F. **driver code**: I'm sure that you will want to generate some scripts to tie all these together. Of course, when you first test your code you will want to do this part manually. Make sure that your driver code occasionally reports what it is doing. This way you will still save some information even if your job dies.

A random thought: if you'd like to test your system on a small code, you could always use the (9,4) product code from Small Project 2. It has 9 bit nodes of degree 2 and 6 check nodes of degree 3.

Arvind makes the following suggestions regarding longjobs:

- Make sure that your script runs in the t-shell, tsh. This is the default on athena. But, if you use bash, you can have trouble.
- Your script should add `matlab`. You should then probably call matlab like `matlab -nodisplay < your_script.m`, where `your_script` is a matlab script file (just a bunch of commands that don't begin with `function`).
- Don't forget to add a `quit` command at the end of your script so that the job will stop.
- Test your longjobs on a short job so you can be sure it works!
- Time your code on small runs so that you can figure out how many runs you can fit into long bursts (of say 8 or 24 hours).

2 Generating Data

If your code is fast enough, then you should work with codes of block-length $n = 10,000$. If that is too slow, try $n = 3,000$.

Note that your results will be different with different block-lengths.

You will generate various types of reports. The first type is the standard channel vs. BER plots. For these, you should simulate your system over

- The Binary Erasure Channel with erasure probabilities $p = .4, .42, .44, .46$. If you have more time, feel free to try a finer gradation.
- The BSC with crossover probabilities $p = .6, .7, .8$. If you have time, also try $.65$ and $.75$.
- The Gaussian channel at $E_b/N_0 = 1.2, 1.3, 1.4$ and $1.5dB$. Note that $E_b/N_0 = E_s/(RN_0)$, so for $R = 1/2$ we have $E_b/N_0 = 1/\sigma^2$.

Run at least 100 iterations for each. Try to run 1000 iterations for those on which the system performs better (we need to work harder to observe failures when they are rare).

The next type of output we would like to generate is some that will help us understand how our algorithm is behaving. One thing we would like to understand is how much progress the algorithm is making at each iteration. We can measure this by simulating a terminal stage after each check-node stage, and counting how many bits it gets right. We can then plot the number of iterations on the X-axis and the BER if we were to stop at that point on the Y-axis. Do this twice: once at a noise level for which the algorithm tends to work, and once at a noise level for which it tends to fail. You could combine many runs to do this. Be sure to explain how you derived the data for your plot.

Finally, we would like to understand how the messages sent by the algorithm are distributed at each iteration (or maybe every 5 iterations). So, we will extract this data for a few simulations. Of course, there is a question about how one should report this data. A histogram would be natural. However, it might not be meaningful if most of the data is near 0 or 1. A more useful approach might be to output some *order statistics* for the data. One way of obtaining these is to sort the data, and, if there are n items, report the 1st, $n/10$ th, $n/5$ th, $3n/10$ th, etc. data point in the sorted order. This gives one some feel for distribution. Generate data like this for each iteration (or every 2nd or 5th) of the algorithm on some run on which it succeeds, and on some run on which it fails. It would be most interesting to see it for a run on which it barely fails. Don't forget to explain how you got the data.

3 Reporting

Submit all code, plots, and the data used to generate plots (unless it is absurdly big). Start with an overview explaining what is to follow, and how your code is organized. To the extent that you generated data by interaction, rather than merely running the code you've submitted, explain what you did.

Note that I'm going to actually try to read your code, so please document it well!

4 Collaboration

For this project you are free to discuss how you do this with others in the class (and especially to get technical help). But, you should write your own code. You must acknowledge your collaborators.