Communication Lab 2013 Final Report
Low-Density Parity Check Code

Group 3 陳彥均 蔡秉珈 莊皓翔

I. Abstract

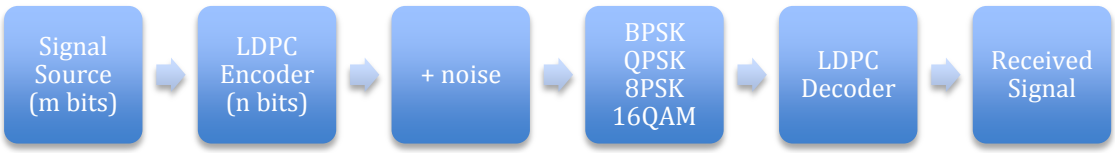
We learned the theory of low-density parity check (LDPC) code and implemented the code. By having several tests, we evaluated the performance of LDPC and compared it with the lab we have done before.

II. Introduction

All of us are very interested in the previous lab about convolutional code simulation. Therefore, we decided to do more research on this topic and found a very popular way of error correcting code simulation—LDPC. We decided to implement this technique based on the program of previous lab.

- Understanding the theory behind the code
- Implementation of the encoder and decoder of LDPC
- Combine this convolutional code technique along with BPSK, 8PSK and 16QAM
- Analyze the result with previous lab work

III. System model



IV. Theory and Principle

The Low-Density Parity Check code (LDPC code) is a kind of linear error correcting code. Basically it is a block code with a low-density parity check matrix H. The “low density” here means that there are only a few ones in the matrix H, and the other elements of H are all zeros. The LDPC code has the following advantages. First, it can achieve performance close to the Shannon limit provided that the codeword length is long. Second, it has a lower decoding complexity than that of the Turbo code. The commonly used decoding algorithm for LDPC is “belief propagation”, which is parallelizable and can be accomplished

at significantly greater speeds than the decoding of Turbo codes. Third, the decoding algorithm is verifiable in the sense that decoding to a correct codeword is a detectable event.

- Encoding

To get a (n, j, k) LDPC code, we can generate an m by n parity check matrix H (where m = n - k) and derive the generator matrix as follows:

1. Generate m x n matrix H by setting all of the elements to zero. Flipping the first k elements to 1 in the first row and the k+1 to 2k in the second and do this for the first m/j rows. Ex : (n, j, k)=(20,3,4)

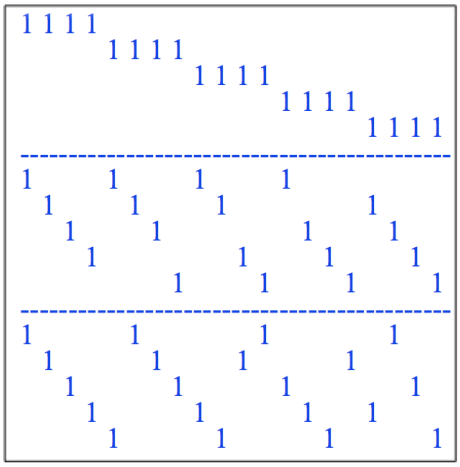


Fig. 1

The rest parts of the rows are just the permutation of the columns of the first part.

2. Shift the columns of H such that the right hand square sub-matrix of H has full rank, i.e. H = [A|B], where A is an m by k matrix, B is an m by m square matrix. Shift the columns of H to make B has full rank.
3. Use row operations to convert H to systematic form H_{sys}. Then derive the corresponding generator matrix G:

$$H_{sys} = [P|I_{m\times m}]$$

$$\mathrm{G} = [I_{k\times k}|P^T]$$

4. For a length-k information sequence s, the corresponding length-n coded word t can be calculated by:

$$\mathrm{t} = G^T s \ (mod\ 2)$$

- Decoding

The decoding algorithm iteratively computes the distributions of variables in graph-based model. Figure 2 shows the graph-based model. The

information of the probability of every variable is exchanged between the check nodes (lower blocks in fig. 2) and variable nodes (upper nodes in fig. 2). In our experiments, we used log-likelihood ratio (LLR) as the information of probability. Log-likelihood ratio (LLR):

$$\mathrm{L}(c_i) = \log\left(\frac{\Pr(C_i = 0 \mid y_i)}{\Pr(C_i = 1 \mid y_i)}\right)$$

Where c represents the code and y represents the received signal.

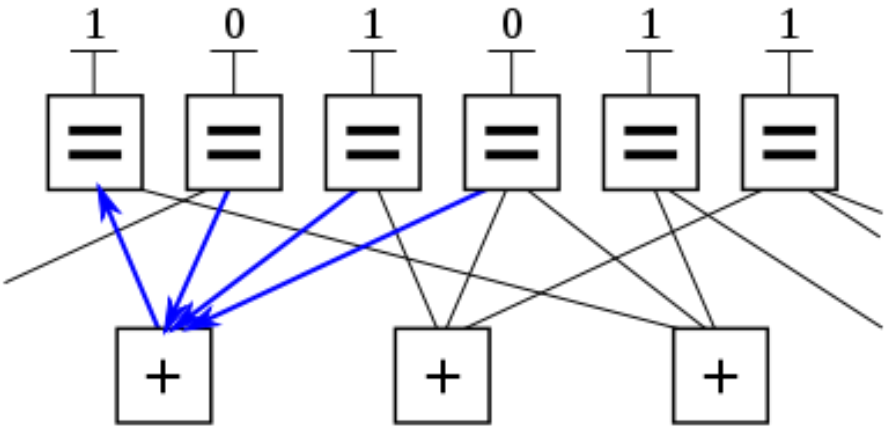


Fig. 2

By iteratively computing LLR between nodes, we can get the convergent result.

a. $\mathrm{L}(r_{ij}) = 2tanh^{-1}(\prod_{i' \in Row[j] \backslash \{i\}} \tanh(\frac{1}{2} L(q_{i'j})))$

b. $\mathrm{L}(q_{ij}) = \mathrm{L}(c_i) + \sum_{j' \in Col[i] \backslash \{j\}} \mathrm{L}(r_{ij'})$

After several iterations,

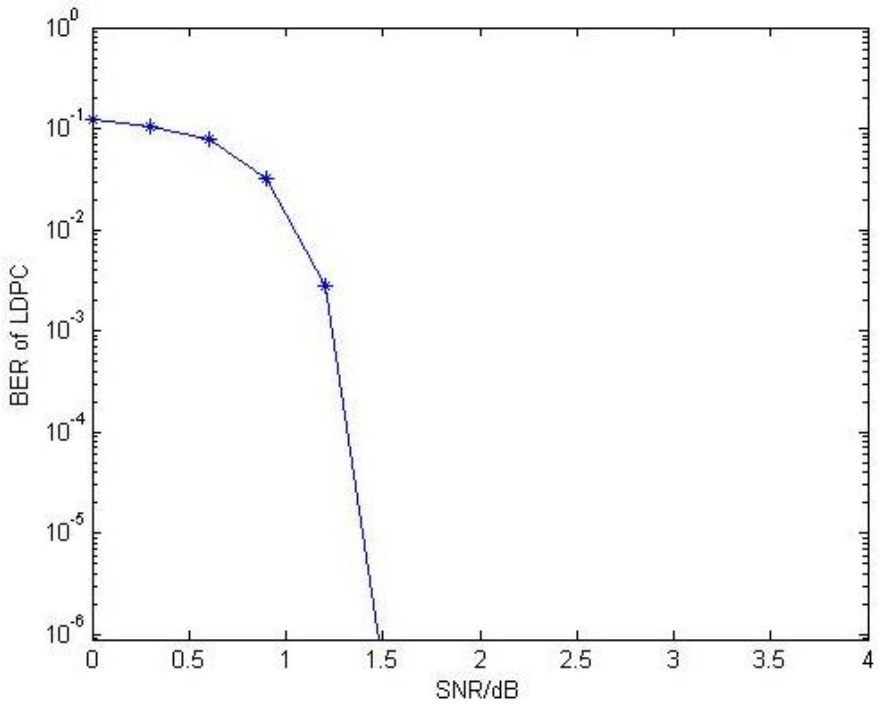
$$L_{posterior}(c_i) = \mathrm{L}(c_i) + \sum_{j' \in Col[i]} L(r_{ij'})$$

V. Results

First, we transmit random data to test the BER characteristics of LDPC code.

We plot the result as the BER curves below.

1. Comparison with Other's work



Reference:

http://www.mathworks.com/matlabcentral/fileexchange/28070-the-parity-check-matrix-of-ieee-8

Fig. 3

Figure 3 is a simulation having H with size =2256 × 4512.

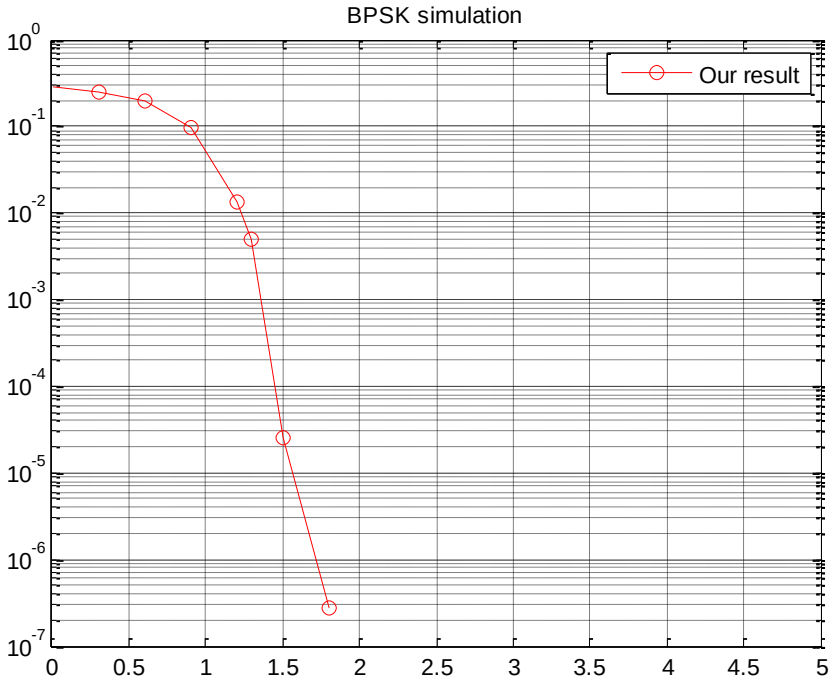


Fig. 4

2. Comparison with CC and BPSK without code

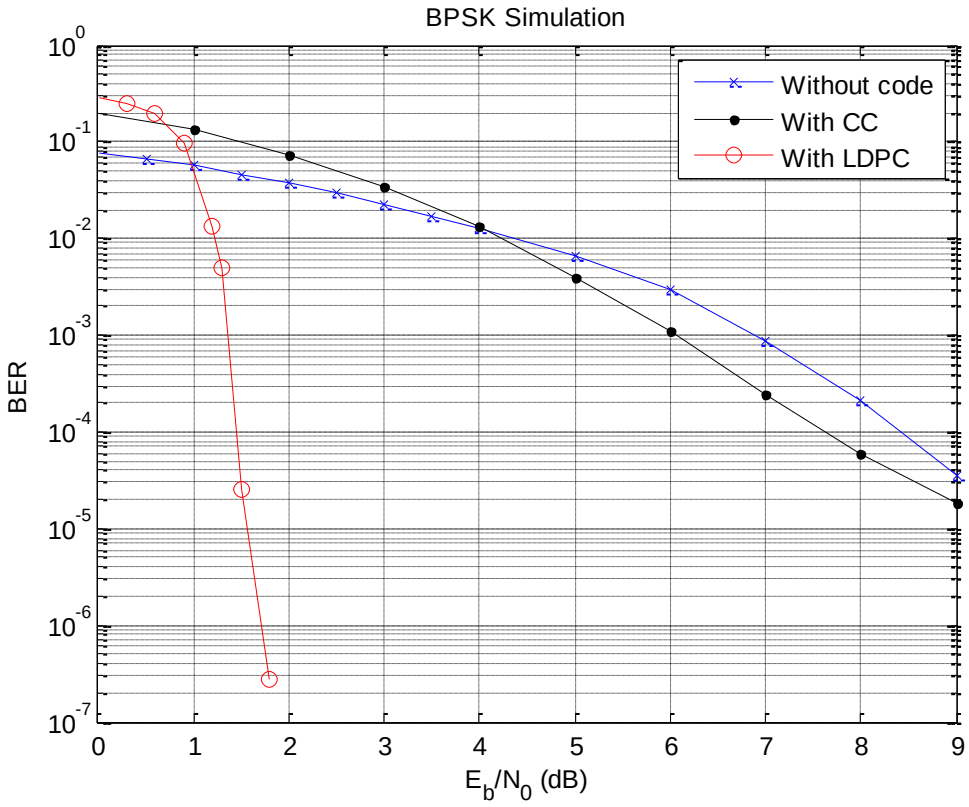


Fig. 5

3. LDPC Code Simulation with Different Numbers of Iterations

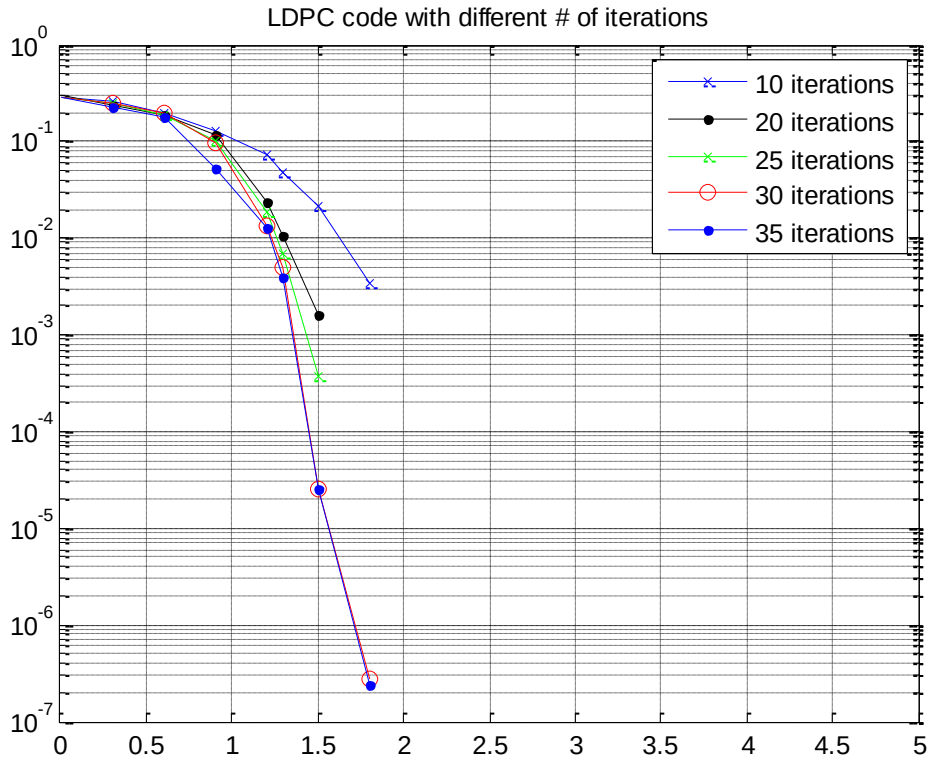
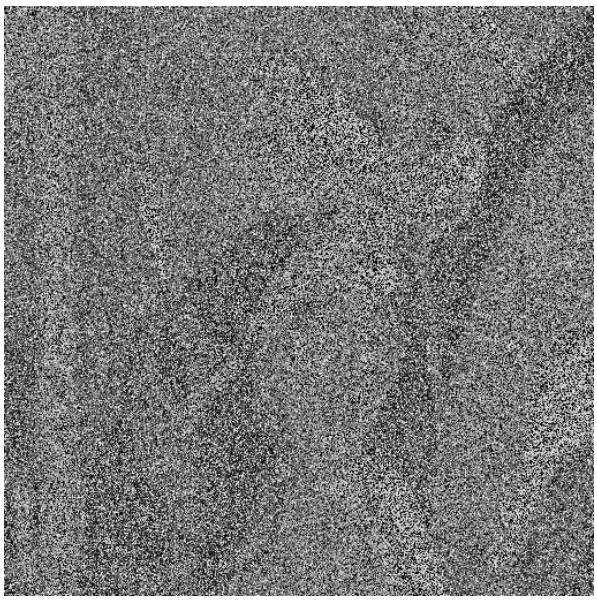


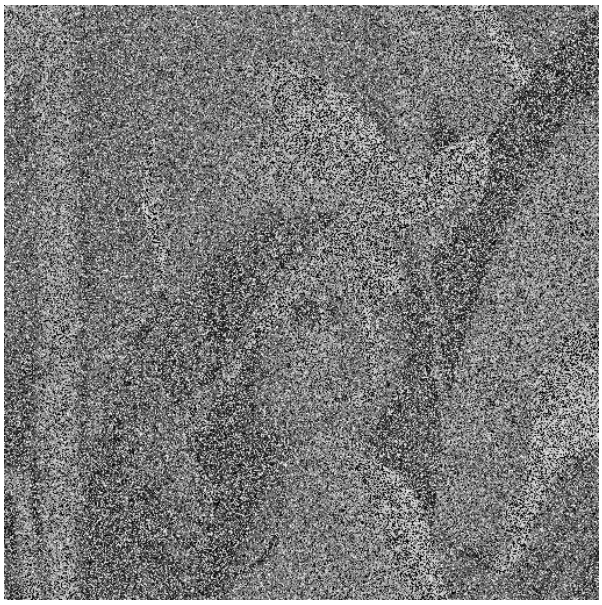
Fig. 6

4. Image Transmission under Different SNR

In order to visualize the BER and get some insight of the error distribution, we transmit the Lena image (BMP format; 512x512 pixels). The results are shown below.



SNR = 0dB



SNR = 0.3dB

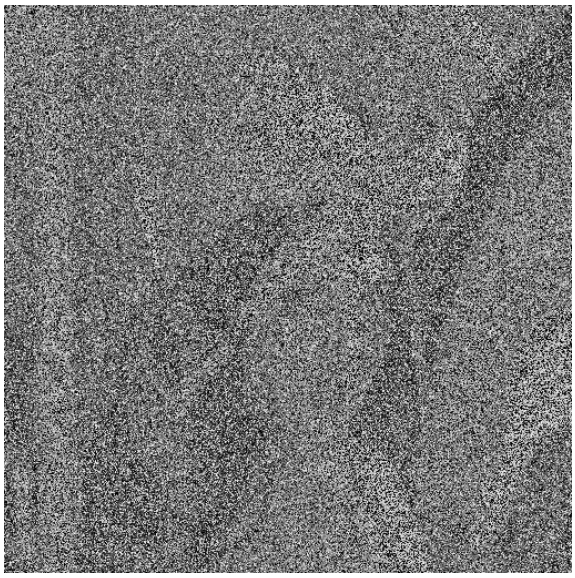
SNR = 0.6dB

SNR = 0.9dB

	
SNR = 1.2dB	SNR = 1.5dB
	
SNR = 1.8dB	SNR = 2.1dB

5. Image Transmission with/without LDPC Code

a. SNR = 0dB

	
Ordinary BPSK	BPSK with LDPC code

b. SNR = 0.9dB

	
Ordinary BPSK	BPSK with LDPC code

c. SNR = 1.2dB

	
Ordinary BPSK	BPSK with LDPC code

d. SNR = 1.8dB

	
Ordinary BPSK	BPSK with LDPC code

VI. Conclusions

First of all, we need to verify our simulation. We found a reliable simulation result on the internet. The curve is almost consistent as our result. We can see the sharp waterfall, the important characteristic of LDPC code. Therefore, we can convince ourselves that our implementation is correct.

In the second part, we put the BER curves together with the BER curves in previous labs. We can see that LDPC encoded BPSK performs much better than the ordinary BPSK and BPSK with CC in most of the SNR values. At low SNR, the energy will disperse into code with twice length. Also, if there exists a lot of error in the signal, there will be not enough correct bits for correcting the error. Due to the two main reasons, LDPC code performs worse when SNR is really low. As SNR goes high, LDPC code shows its power and has extremely better result.

In the third part, we can see that different times of iterations can cause different result of BER. The main idea of LDPC code is that the probability of every bit being 1 will close to 1 or 0 after several iterations. If we do is few times, the result will not converge to an ideal value. In our test, the simulation with more times of iterations has better results of BER. However, it converges at about 30 iterations, so we need only 30 iterations in the rest of our simulations.

In the fourth part, we can see the image gets clearer as the SNR goes high. Interestingly, we discover that the corrupted image pixels tend to be in the same row, especially in the image of SNR = 0.9 and SNR = 1.2. We guess this is due to the fact that we segment the data and encode them separately. If the received segment contains some large error, the error propagates in LDPC algorithm. Thus the decoded segment would contain lots off error bits. Because BMP format takes the pixels row by row and our segmenting algorithm segment the data as original order, we found the above observation.

In the last part, we compare the LDPC encoded BPSK and ordinary BPSK on image transmission. We found a consistent result as the random bit transmission: in low SNR channel, LDPC has poor performance; in high SNR channel, LDPC has significantly better quality. We also can see the different

error distribution characteristics. The ordinary BPSK error occurs uniformly randomly, on the other hand, LDPC error tends to concentrate in the same row. If we can shuffle the data before LDPC transmission and recover it back at the receiver, we may gain some visual quality on the received image.

VII. Reference

- http://en.wikipedia.org/wiki/Low-density_parity-check_code
- https://sites.google.com/site/zhenglu1986/implementation-of-ldpc-codes-in-labview
- http://www.jatit.org/volumes/Vol38No1/14Vol38No1.pdf
- http://www.telecom.tuc.gr/~alex/papers/ryan.pdf

VIII. Team member

陳彥均　　莊皓翔　　蔡秉珈

IX. Source code

We use mostly C++ to implement on Linux OS with g++ as the compiler together with small portion of MATLAB.

The source in “Source Code” file contains three parts. The first part is to find a good matrix H for parity check and the generator G for the encoder. We tried to construct one by ourselves but failed. Thus we decided to use the matrix for LDPC in IEEE 802.16e. Luckily we found the matrix in txt format.(802.16eH.txt) Then we use MATLAB to parse it and compute the

required G and output them together in our desire format as binary files.(H.matrix, G.matrix) In the second part, we write the encoder and decoder using C++. We define them as two solver class in ldpcCode.h and ldpcCode.cpp. The work is to parse the matrix file and perform the encoding and decoding operation. The last part is to unify the source code to the previous work of lab exercises. We choose to use inheritance to strengthen the transmitter and receiver class with the LDPC encode ability. The implantation details are omitted here.

Finally, we can perform simulation using the class in commSystem.h. The sample codes are ldpc_ber.cpp and ldpc_image.cpp. The Makefile is also included. Simply type mage and get the two executable ldpc_ber.exe and ldpc_image.exe for simulation.

X. Appendix

The result in this report is quite different from that in our demo at 6/25. The BER here is much lower, and we can see that the curve moves left. It is due to the correction of our code. In our decoder, one variable should have the type “double”, but we previously mistook it as “integer” and caused some error. The edition in this report is the correct one.