1 Executive Summary

Modern flash memory is reliant on error-correcting code to ensure error-free operation. Current flash memory architectures often use linear block codes for this purpose (e.g. Reed-Solomon, Hamming or BCH codes). However, the recent rediscovery of LDPC (Low Density Parity Check) codes, which can achieve superior performance close to the Shannon Limit, has generated much interest in the NAND memory industry. These codes could be used to further improve error correction capability in flash memory, thus allowing for more densely packed memory cells and thus larger capacity drives.

The general aim of this project is to produce a MATLAB simulation of how an error correction system using LDPC codes would work for flash memory. By combining both an error generation and an error correction model, it will be possible to benchmark these rediscovered codes, and subsequently compare them to current generation technologies.

Contents

2 Introduction

3 Overview of Linear Block Codes

Linear block codes are one of the two main classes of forward error correction (FEC), with the other main type being convolutional coding. A linear block code essentially takes a block of binary data, and adds additional redundant data onto it. This block can then be transmitted over a noisy channel, and subsequently decoded at the receiver. The redundant bits in the block are used as parity check equations, which allows a linear block code to both detect and correct errors.

3.1 Definitions for Linear Block Codes

All linear block codes can be described using a set of standard terms and symbols. For this project, all codes use a binary alphabet of $\{0, 1\}$ and hence all operations are over this binary field. n is the block length, the total size of the output codeword. k is the message length, the size of the information vector prior to encoding. An (n, k) error correcting *code* $\mathcal{C}$, will produce a set of 2^k output *codewords* $\mathbf{c}$. Hence, $\mathbf{c} \in \mathcal{C}$.

The rate of any linear block code, R, is defined as:

$$R = \frac{k}{n} \tag{3.1.1}$$

The rate is a measure of the number of information bits compared to the total number of transmitted codeword bits. A high rate code will be more efficient in terms of useful information transmitted, but will have a poorer error correction capability. In Flash Memory, very high rate ($R > 0.9$) codes are used in order to maximise the amount of usable storage space. Conversely, an example use of low rate codes would be in deep-space probe transmissions, where receiving error-free data is more important than rate of transmission.

A linear block code can be represented in two ways: through the $k \times n$ generator matrix $\mathbf{G}$, or the $(n - k) \times n$ parity check matrix $\mathbf{H}$. Each is the null-space of the other, such that:

$$\mathbf{G}\mathbf{H}^{\top} = 0 \tag{3.1.2}$$

Unsurprisingly, the generator matrix $\mathbf{G}$ is used in the transmission side when encoding data, and the parity check matrix $\mathbf{H}$ is used at the receiver to detect and correct any errors.

At the transmitter, if we take a $1 \times k$ input vector of binary data $\mathbf{x}$, the method of encoding this data into a codeword $\mathbf{c}$, is simply a multiplication operation:

$$\mathbf{c} = \mathbf{x}\mathbf{G} \tag{3.1.3}$$

At the receiver, a similar operation is performed:

$$\mathbf{s} = \mathbf{c}\mathbf{H}^{\top} \tag{3.1.4}$$

where $\mathbf{s}$ is known as the *syndrome*. If the syndrome is the all zero vector, then error free transmission has occurred. Conversely, if any bit of the syndrome is 1, then this represents a particular error pattern. For small block lengths, these error pattern's can be pre-calculated and saved as a table, allowing for syndrome lookup decoding. The table identifies the exact location of a bit error in the codeword, which can then be 'flipped' in order to perform error correction.

An example of a *systematic* generator matrix, in this case a matrix known as the *Hamming(7,4) code*, takes the form:

$$\mathbf{G} = \left(\begin{array}{ccc|cccc} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ \underbrace{1 \quad 0 \quad 1}_{n-k} & & & \underbrace{0 \quad 0 \quad 0 \quad 1}_{k} & & & \end{array}\right) \tag{3.1.5}$$

In this code, 4 information bits are encoded into 7 output bits. Notice the identity matrix in the right portion of the generator matrix. This means that the 4 message (k) bits are always encoded at the end of the codeword, with the parity check ($n - k$) bits at the start of the codeword. This is why it is *systematic*. At the decoder, it is then easy to extract the (uncorrected) message bits from the codeword, simply by looking at the last 4 bits.

3.2 Hamming weight, distance, and error correction capability

An important metric when discussing error correcting codes is the concept of the Hamming weight. For any codeword, the Hamming weight is defined as the total number of non-zero elements in a given codeword. Another metric, the minimum weight (w_{min}), is simply the minimum value from the set of all

Hamming weight's for a given code, excluding the all-zero case.

Example: A fictional example (6,2) code could have the following codewords:

Message (2 bits)	Codeword (6 bits)	Hamming weight
0 0	0 0 0 0 0 0	0
0 1	1 0 1 0 1 0	3
1 0	0 0 1 1 0 0	2
1 1	1 0 0 1 1 0	3

For this code, the minimum weight (w_{min}) is 2, since that is the smallest value of the Hamming weight's excluding the all-zero case.

Another metric used is the Hamming distance. The Hamming distance defines how 'close' any two codewords are to each other. Codewords that are 'far away' from each other are less likely to be decoded in error, and hence the Hamming distance determines how 'good' a code is at error detection and correction. Formally, the Hamming distance is defined as the number of (binary) places that any 2 codewords differ. Analogous to the minimum weight, there is also a minimum distance (d_{min}), which is the minimum value from the set of all Hamming distances for a given code, excluding the trivial case of comparing a codeword to itself.

An important result arises because of the use of binary arithmetic, in that the minimum Hamming weight is in fact equal to the minimum Hamming distance:

$$d_{min} = w_{min} \tag{3.2.1}$$

It is now possible to present the results that describe, for linear block codes, their error correction and detection performance:

Error Detection Theorem: *A linear block code with minimum weight w_{min} is able to detect up to e errors:*

$$e_{detectable} = w_{min} - 1 \tag{3.2.2}$$

Error Correction Theorem: *A linear block code with minimum weight w_{min} is able to correct up to e errors:*

$$e_{correctable} = \frac{w_{min} - 1}{2} \tag{3.2.3}$$

3.3 Low Density Parity Check codes

A particular class of codes, known as “Low Density Parity Check” (LDPC) codes, are of particular interest and relevance to this project. LDPC codes are generally considered to be some of the best performing linear block codes available, in terms of error performance, with some codes getting within a fraction of the Shannon Limit. Additionally, LDPC codes have no patent and hence no licensing costs, making them attractive for real world use.(??!).

LDPC codes were originally discovered by R.G. Gallager in 1962. Then known as “Gallager codes”, they were defined by a sparse parity check matrix with low column weights. Gallager also worked on a probabilty based decoding method for these codes, which proved to have promising performance. However for various reasons, these codes were essentially lost in favour of other more practical codes. It is possible that the decoding complexity for LDPC was, at the time, too great for the computational power then available.[1]

Modern LDPC codes were re-discovered by J.C. MacKay in 1996. MacKay demonstrated that LDPC codes could be decoded using probabilistic methods, even beyond the bound set by their minimum distance. Today, LDPC codes are seeing a resurgence in various applications. Most notably in the DVB-S2 standards for digital HD satellite broadcast, 10GBase-T Ethernet and as optional ‘add-ons’ to the 802.11n/ac wi-fi standards.

Disadvantages of LDPC include the fact that there still exists a small (often in the region of 10^{-6} to 10^{-9}) probability of error after decoding, known as the ‘error-floor’. This can be avoided by using a second high rate, ‘inner’ Error Correcting Code such as BCH or Reed-Solomon to remove the last few bit errors. Other issues include decoding complexity. Whilst decoding time is linear with block length, decoding using the belief propagation algorithm is still problematic, especially for low power mobile devices. Most applications of LDPC so far have been on mains-powered equipment.

Example: A specific DVB-S2 code, with $n = 64800$ and $r = 0.9$, has a total of $194,399$ non-zero elements in it’s parity check matrix $\mathbf{H}$. However, the non-zero elements account for just 0.04% of the total matrix: The vast majority of $\mathbf{H}$ is empty. It is therefore easy to see why they are called “Low Density”. In this project, this specific high-rate code is used extensively, especially when modelling the memory-specific case in section ?????????.

[1] Personal opinion. Even today, decoding LDPC using near-optimum belief propagation on a PC is computationally expensive, whilst on dedicated ASIC hardware consumes large amounts of power.

3.4 Channel Capacity

Channel capacity stuff to go here

4 Overview of Flash Memory Technology

5 Decoding of LDPC Codes

Using syndrome lookup decoding as described in section 3.1 would be nearly impossible for longer block lengths. There is a much better, iterative decoding method that can be used for any linear block code, and which is linear in block length. It is called the belief propagation algorithm (also known as the sum-product algorithm or the message passing algorithm).

There are two distinct methods of belief propagation: Hard decision decoding and soft decision decoding. In hard decision decoding, the error correction algorithm only receives binary data (i.e. $\{1,0\}$). In soft decision decoding, the error correction algorithm receives a numerical likelihood of the data being either a 0 or 1. Soft decision decoding will therefore result in superior performance, since it is able to make use of the additional ’soft’ information that is otherwise discarded.

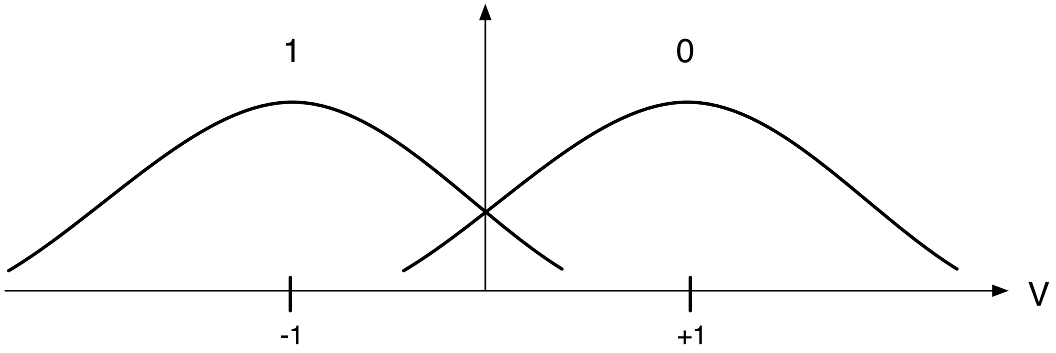

Figure 1: Received voltage probability distribution for AWGN channel

Figure 1 shows the typical probability distribution of a Binary Phase Shift Keying (BPSK) system with Additive White Gaussian Noise (AWGN). The x-axis is effectively a received voltage value from the demodulator. At transmission, a value of +1 volts corresponds to a binary 0, and a value of -1 volts corresponds to a binary 1. However, the additive noise in the channel results in the received voltage taking a range of values, and hence the received voltage is now defined as a probability distribution.

With hard decision decoding, the obvious boundary would be $x = 0$, half way between the +1 and -1 constellation symbols. Any value to the right side of this boundary would always be classified as binary 0, and anything to the left always binary 1. This means that a voltage value of 0.01 would be output as a binary 0, even though in practice it is almost equiprobable to be a binary 1. The fact that it could equally be a binary 0 or binary 1 is lost when making a hard decision, and the error correction decoder does not get that additional information.

Soft decision decoding seeks to improve on hard decision decoding, by making use of the actual received voltage value, rather than discarding it. The messages are now the conditional *probabilites* of being a 1 or 0, instead of being just binary values. This allows the error correction decoder to know the degree of certainty that the message sent was a 1 or a 0.

5.1 Hard decision decoding

The message passing algorithm can be best understood with hard decision decoding. As an example, eq 5.1.1 shows the parity check matrix $\mathbf{H}$ of an (8,4) code. This code can also be displayed, as in figure 2, as a visual graph representation known as a tanner graph. [Reference: LDPC Leiner tutorial]

$$\mathbf{H} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{5.1.1}$$

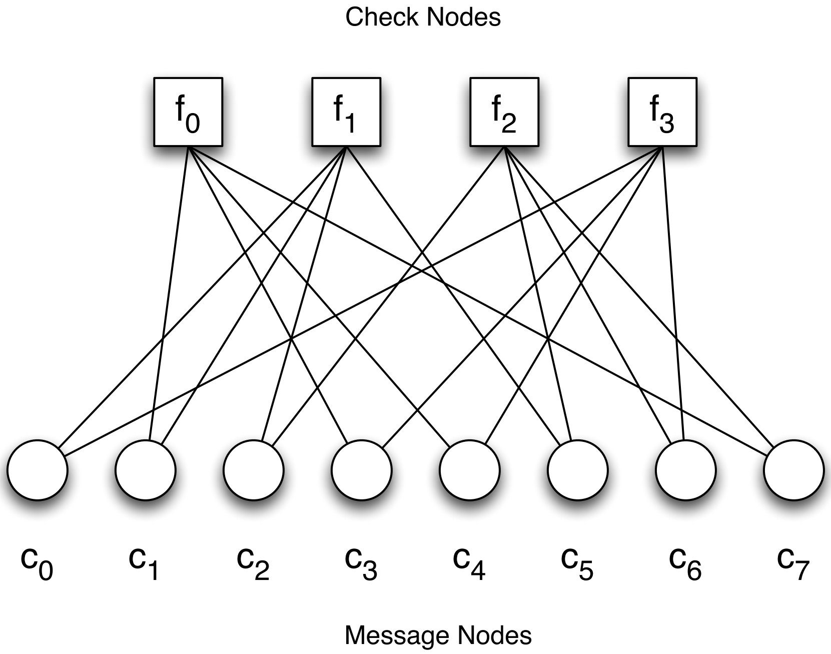

Figure 2: Tanner Graph

The tanner graph is a bipartite graph with 2 types of node: Check nodes and Message nodes. The check nodes represent the $(n-k)$ parity check equations, whilst the message nodes represent the n codeword bits. It is directly related to $\mathbf{H}$: Check node f_j connects to message node c_i if element h_{ji} is 1. Using the tanner graph, the hard decision decoding algorithm can now be explained as follows:

1. All message nodes c_i, having been initialised to the received codeword y, send their value to their connected check nodes f_j.

2. Each check node f_j calculates a separate reply back to each message node c_i, with the binary value that it believes the message node should be. The parity check equation at each check node must satisfy $|\sum f_j|_{mod2} = 0$. From the example, check node f_0 receives values from $c_{1,3,4,7}$. When sending it's reply back to variable node c_1, it uses the values from nodes 3, 4 & 7 along with the parity check constraint, to calculate the outbound message. Note that it does not use the information received from c_1 to send a reply back to c_1. This process continues: At each check node, a separate reply is calculated back to each connected message node.

3. The check nodes send their update back to the message nodes. Each message node in this example is connected to 2 check nodes, as well as having a previous value from step 1. As such, majority logic (with 3 bits in this case) can be used to decide whether the message node should be a 1 or a 0.

4. The process now loops until the parity check constraint is satisfied for all nodes, at which the process terminates.

A simple example describing the check node stage:

- Check node f_0 might receive values from c_1,c_3,c_4,c_7 = {1,1,0,1}.
- To calculate the reply message to each c_j, use the parity check constraint $|\sum f_0|_{mod2} = 0$:
  - Reply for $c_1 : x + 1 + 0 + 1 = 0 \therefore c_1 = 0$
  - Reply for $c_3 : 1 + x + 0 + 1 = 0 \therefore c_3 = 0$
  - Reply for $c_4 : 1 + 1 + x + 1 = 0 \therefore c_4 = 1$
  - Reply for $c_7 : 1 + 1 + 0 + x = 0 \therefore c_7 = 0$
- Repeat this process at all other check nodes $f_{1,2,3}$

5.2 The Log-likelihood ratio

Whilst hard decision decoding is a good way to demonstrate how the iterative message passing algorithm works, soft decision decoding yields substantially better error correction performance, and is therefore the main method used in decoding LDPC.

In soft decision decoding, the message nodes no longer represent binary 1's or 0's, but instead can take a continuous range of probability values. These probability values are initially calculated using two sources of information: The received voltage value of each bit, and the underlying probability distribution of the received bit. Before being able to describe the soft decision decoding algorithm, this information needs to be formed into a useful metric: the *Log-likelihood ratio*, [Reference: LLR computation.pdf]

$$\mathcal{L}(c|y) = log_e \left[\frac{f(c=+1|y)}{f(c=-1|y)} \right] \tag{5.2.1}$$

The term $\mathcal{L}(c|y)$ is the likelihood of c being transmitted given that y was received. The log-ratio is able to tell us if $c = +1$ or $c = -1$ was the most likely transmitted symbol, given the received value y. A positive LLR indicates it is more likely that $c = +1$ was transmitted, and a negative LLR that $c = -1$ was transmitted. Additionally, the LLR has a range of $-\infty$ to $+\infty$, which provides the degree of certainty of a given symbol.

To make use of the LLR, it is necessary to manipulate it into a more useful form. Using Bayes' rule[2], and the assumption that the transmitted bits are equiprobable[3],

$$\begin{aligned}\mathcal{L}(c|y) &= log_e \left[\frac{f(y|c=+1)\dfrac{f(c=+1)}{f(y)}}{f(y|c=-1)\dfrac{f(c=-1)}{f(y)}} \right] \\ &= log_e \left[\frac{f(y|c=+1)}{f(y|c=-1)} \right] \\ &\sim log_e \left[\frac{\textit{“Probability density function of +1”}}{\textit{“Probability density function of -1”}} \right]\end{aligned} \tag{5.2.2}$$

The LLR can now be calculated for each received value of y. Note that it is necessary to have the underlying density functions of the received symbols. By inserting the received value y into each PDF, you obtain a numerical probability of y representing either +1 or -1. The ratio of these 2 probabilities then gives us the likelihood ratio.

[2] $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$

[3] i.e. $f(c=+1) = f(c=-1)$

The LLR method applies in general to all noisy channels, that is, the probability density functions can take any form. However when dealing with the AWGN channel, converting from a received symbol amplitude y into the LLR is much simpler, since the probability density function in both cases is a gaussian,

$$\begin{aligned}\mathcal{L}(c|y) &= log_e\left[\frac{\mathcal{N}(1,\sigma^2)}{\mathcal{N}(-1,\sigma^2)}\right] \\ &= log_e\left[\frac{e^{-(y-1)^2/(2\sigma^2)}}{e^{-(y+1)^2/(2\sigma^2)}}\right] \\ &= log_e\left[e^{4y/(2\sigma^2)}\right] \\ &= \frac{2y}{\sigma^2}\end{aligned} \tag{5.2.3}$$

This result makes it very easy to take the output from the demodulator (the +1/-1 BPSK y symbols) and generate the appropriate LLR's for the AWGN channel.

5.3 Soft decision decoding

The Belief Propagation algorithm for soft decision decoding follows a similar process to that of hard decision decoding. Messages are passed between check and variable nodes defined by the tanner graph. The main difference is what happens at each of the nodes, since now the messages are LLR's rather than binary bits.

[Papers such as ??, ?? and ??] have all proved the set of equations that are used at both the message nodes and check nodes. The belief propagation algorithm can be distilled into just 2 equations: The message node update equation, and the check node update equation. The reason why it is often called the 'Sum-Product' algorithm, is since the message node update equation involves summing the LLR's, and the check node update equation involves taking their product.

$$m_{ij}^{(l)} = L_i + \sum_{j' \in C_i \neq j} m_{j'i}^{(l-1)} \tag{5.3.1}$$

Equation 5.3.1 is the message node update equation. $m_{ij}^{(l)}$ is the message sent from message node i to check node j, at iteration l. L_i is the initial LLR for message bit i. The expression $j' \in C_i \neq j$ is used to sum the incoming messages from all check nodes j' that are connected to C_i, except the current node j that is being sent to. This exclusion is the same as for hard decision decoding, whereby a message to a node is never a function of a message from that node (the extrinsic information rule).

Finally, $m_{j'i}^{(l-1)}$ indicates that the sum is of the received check node messages from the last $(l-1)$ iteration.

$$m_{ji}^{(l)} = 2\,\mathrm{arctanh}\left[\prod_{i' \in V_j \neq i} \tanh(\frac{m_{i'j}^{(l-1)}}{2})\right] \tag{5.3.2}$$

Equation 5.3.2 is the check node update equation. $m_{ji}^{(l)}$ is the message sent from check node j to message node i, at iteration l. As before, $i' \in V_j \neq i$ includes all message nodes i' that are connected to check node V_j, except the current node i that is being sent to (extrinsic information rule).

With both node equations defined, the iterative soft decision decoder proceeds as follows:

1. All message nodes c_i, having been initialised to the received LLR's L_i, send their value to their connected check nodes f_j.

2. At each check node j, using equation 5.3.2, the return message to each connected message node i can be calculated.

3. Sum the LLR's received at each message node, and obtain the binary value of the codeword $(l_i > 0 \rightarrow 0,\ l_i < 0 \rightarrow 1)$. Using the parity check equation $\mathbf{s} = \mathbf{c}\mathbf{H}^{\top}$, where c is the current binary value of the codeword, we obtain the syndrome. If the syndrome is all-zero, the algorithm terminates, since a valid (though not necessarily correct) codeword c has been found.

4. At each message node i, using equation 5.3.1, the return message to each connected check node j can be calculated.

5. Steps 2-4 repeat until a valid codeword is found, or up to a maximum number of iterations (Usually around $l = 50$).

5.4 Min-Sum approximation

Min sum to go in here

6 The AWGN channel: Simulation & Results

One of the most important noisy channels used in communications theory, and already mentioned previously, is the Additive White Gaussian Noise (AWGN) channel. Whilst this noise channel is not really applicable to model Flash Memory, which is the aim of this project, it is however a useful benchmarking tool to ensure that the MATLAB decoder is working correctly. It is also a good method of demonstrating the power of LDPC, soft decision decoding, and error-correcting codes generally. This section presents the work done in modelling the encoder, AWGN channel, and decoder, and compares the results to known data sets.

6.1 Simulation model

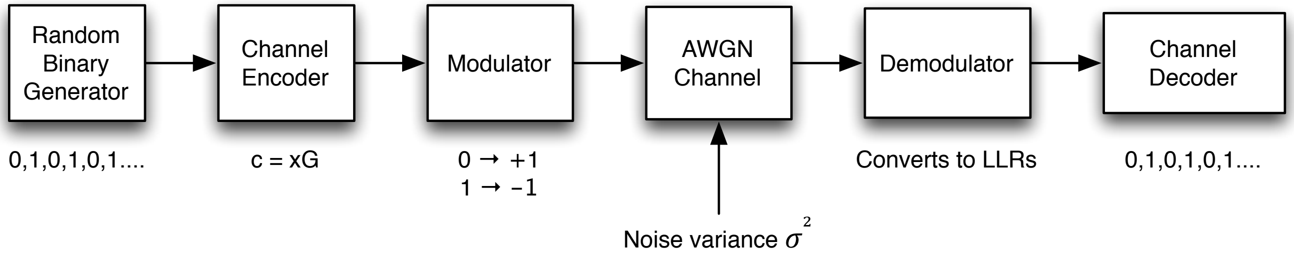

Figure 3: The AWGN simulation model

- The random generator produces a vector of pseudo-random, uniformly distributed values from the set {1,0}, of length k.
- The channel encoder takes the length k message, and encodes it using the generator matrix, into a block of length n.
- The modulator maps each binary bit, in this case using Binary Phase-Shift Keying (BPSK), onto a constellation symbol. These symbols represent a real voltage value.
- The channel is simulated by adding white gaussian noise onto each symbol. The output from the channel is $\mathbf{Y} = \mathbf{X} + \mathbf{N}$, where $\mathbf{X}$ is the input random variable (+/- 1), and $\mathbf{N}$ is the noise random variable ($\mathbf{N} \sim \mathcal{N}(0, \sigma^2)$)
- The demodulator, for soft decision decoding, calculates the LLR of each received symbol. From equation 5.2.3, $\mathcal{L} = \frac{2y}{\sigma^2}$, where y is the received value from the output of the channel, and σ^2 is the

AWGN channel noise variance. In a real system, this method therefore requires knowledge of the underlying noise parameters. For the simulation, the value is assumed to be known.

- Finally, the channel decoder performs error correction using the Belief Propagation algorithm, and outputs the corrected codeword (Note: The encoded message can be extracted from the codeword if a systematic generator matrix was used).

To simulate the error correction performance of this system, and compare it to other results, there needs to be a standardised quantity to describe the error rate and the channel noise. Whilst the noise variance is one such quantity, it doesn't take into account the relative difference between signal power and noise power. A far more useful metric would be Signal-to-noise ratio (SNR), or more specifically in this case $\frac{E_b}{N_0}$ (energy per bit/noise power). Like SNR, this is usually given in decibels[4].

Converting between σ^2 and $\frac{E_b}{N_0}$ is relatively trivial:

E_s	-	Energy per symbol (Always 1 for BPSK)
R_c	-	Code rate
R_m	-	Modulation rate (Always 1 for BPSK)
E_b	-	Energy per information bit

$$\sigma^2 = \frac{N_0}{2} \tag{6.1.1}$$

$$E_s = R_c R_m E_b \tag{6.1.2}$$

N_0, the noise power, is defined as in 6.1.1. The energy per symbol is defined in 6.1.2: the energy per bit multiplied by both the number of bits per symbol and the ratio of useful message data. By substituting E_s into $\frac{E_b}{N_0}$:

$$\begin{aligned} \frac{E_b}{N_0} &= \frac{E_s}{R_c R_m N_0} \\ &= \frac{E_s}{R_c R_m 2\sigma^2} \\ &= \frac{1}{2R_c\sigma^2} \end{aligned} \tag{6.1.3}$$

$E_s = R_m = 1$ in this case, further simplifying the equation. This allows the conversion between σ^2, the parameter used in the noise model, and $\frac{E_b}{N_0}$, the metric used to present the results.

[4] For the conversion calculations, $\frac{E_b}{N_0}$ must be a linear, not logarithmic, value. i.e: $\frac{E_b}{N_0}$(dB) $= 10\log_{10}\left[\frac{E_b}{N_0}\text{(lin)}\right]$

The dependent variable in the simulation is the Bit Error Rate (BER). The output from the channel encoder is compared to the output from the channel decoder, where both are blocks of length n. The number of bit errors is simply the difference between the two codewords (similar to the Hamming distance). This raw bit error number is then divided by the block length, to obtain a bit error rate.

Each instance of the block diagram in figure 3 generates a random binary block, and the noise added to that block is also random. A single sample will not be sufficient to get an accurate error rate of the system for a given noise value, since at very low error rates thousands of blocks may need to be decoded before a single bit error occurs. Performing a simulation of repeated sampling is known as a Monte Carlo simulation. The process in MATLAB is therefore designed to perform N trials of the simulation, with each trial outputting a bit error rate. After completing all trials, the bit error rate can be averaged over all blocks. In this project, for the lowest bit error rate experiments, up to 10^6 blocks were processed per noise value.

6.2 Simulation results

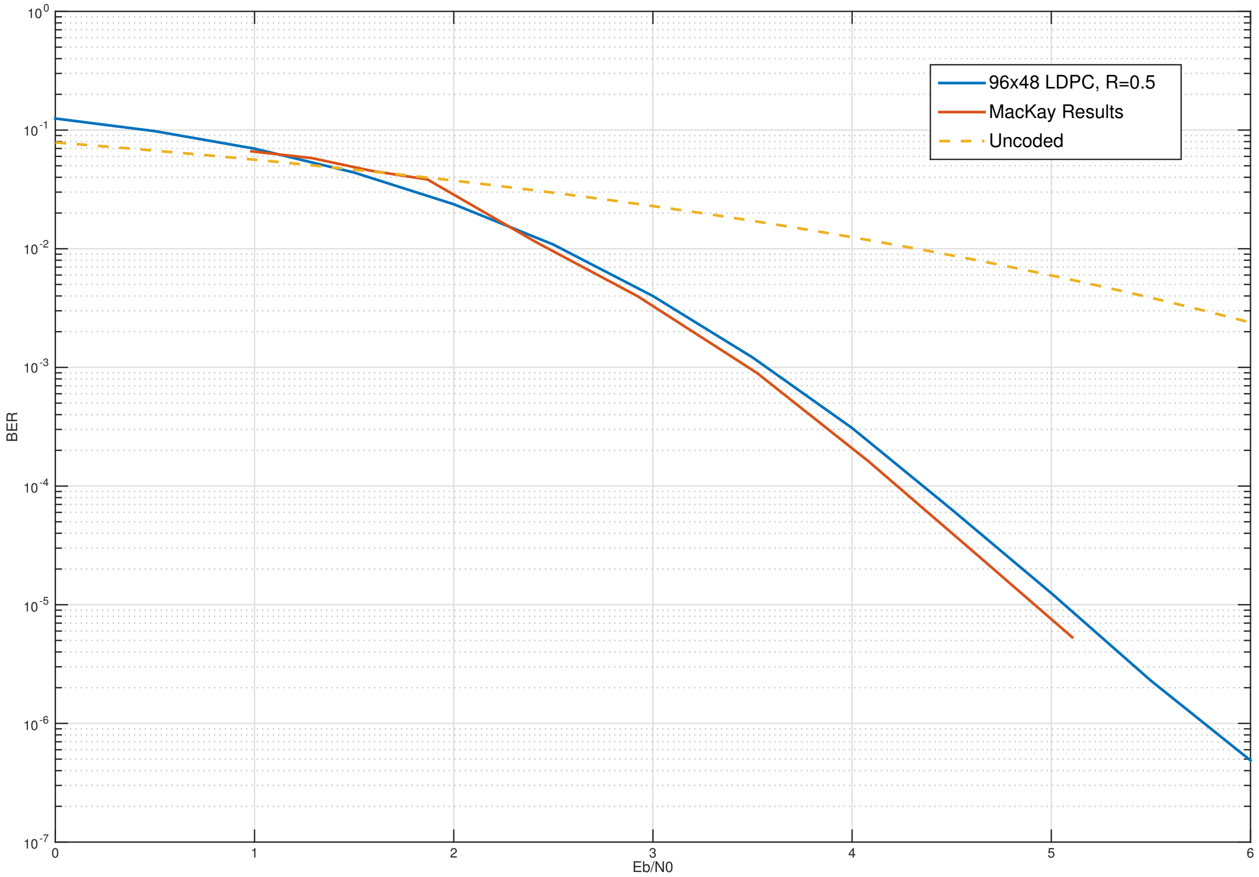

Figure 4: Simulated results of 96,48 LDPC code

A short block length, rate $\frac{1}{2}$ LDPC code was used to produce the results in figure 4, using the Sum-Product decoding algorithm. The figure also displays the results obtained by MacKay using the same LDPC code. There is a minor performance loss compared to MacKay, but otherwise the results are in agreement. Additionally, the dashed line indicates the expected bit error rate in the event that no error correction code was used. There is therefore a substantial coding gain in this system, for example to achieve a BER of 10^{-6} requires around 5.5 dB with coding, and nearly 10 dB without.

6.3 Random number generation

When performing any sort of Monte Carlo simulation, and particularly if running the same program in parallel across multiple computers, it is important to ensure that the random numbers being generated are as random as possible. Even more crucially, there must be no dependence between the parallel task's random numbers if we are to combine result sets.

MATLAB, like many other software packages, cannot generate truly random numbers. Instead, it uses a pseudo-random number generator, such as the Mersenne Twister algorithm. This is just a function that produces numbers which, for most purposes, are considered to be pseudo-random and pseudo-independent. That is, if you generate numbers from this algorithm, they will appear to be random samples from a uniform distribution.

However, the Mersenne Twister algorithm actually has a finite period. After generating 2^{19937} random numbers, the output begins to repeat itself. More importantly, every time MATLAB starts, the random generator is reset to the same position. This means if you try to generate a large set of random numbers in MATLAB, you will always get exactly the same numbers. Effectively, MATLAB uses the same *seed* to the random number generator on every start-up. This is meant to be useful for debugging purposes, however when running simulations in parallel, it causes all the result to no longer be independent. This means you cannot combine results made in parallel, since the output from each parallel stream will actually be identical.

The solution is to ensure that every task executed in parallel has a random seed fed into the random number generator. Effectively, the seed is used as the starting position for the Twister algorithm. If the seed is a truly random number, then each random generator should start in a different position. Any two random generators might start with billions of positions between them, or possibly right next to each other. But the chances of an identical start position are negligible.

The commands in codebox 1 were used to seed the random number generator every time a

Codebox 1: Seeding random generator

```
% Ensures truly random numbers for each process
% seed is now a random number that can be used to initialise rand
fid = fopen('/dev/random');
seed = fread(fid, 1, 'uint32');
RandStream.setDefaultStream(RandStream('mt19937ar','seed',seed));
```

new MATLAB process was created. On UNIX machines, there is a system random number source '/dev/random', which "gathers environmental noise from device drivers and other sources into an entropy pool [and] from this entropy pool random numbers are created." The numbers generated by this process originate from physical random processes such as hardware noise, and so it is assumed that when calling the function on a different machine, a different random number will be generated every time. These random numbers are then used to seed the pseudo-random stream in MATLAB.

7 Modelling a memory-specific noise channel

8 Decoding in the non-Gaussian case

9 The memory channel: Simulation & Results

10 Conclusions