UNIVERSITY OF OXFORD

HONOUR SCHOOL OF ENGINEERING SCIENCE (PART C)

4TH YEAR PROJECT REPORT

Making Flash Memory Work

Investigating error-correcting codes in Flash Memory devices

Author:

Henry FLETCHER

Supervisor:

Professor Justin COON

May 2015

FINAL HONOUR SCHOOL OF
ENG / EEM (delete as appropriate)

UNIVERSITY OF OXFORD

DECLARATION OF AUTHORSHIP

You should complete this certificate. It should be bound into your fourth year project report, immediately after your title page. Three copies of the report should be submitted to the Chairman of examiners for your Honour School, c/o Clerk of the Schools, examination Schools, High Street, Oxford.

Name (in capitals): ……

College (in capitals): …… **Supervisor:** ……

Title of project (in capitals): ……..……

Page count (excluding risk and COSHH assessments): ……

Please tick to confirm the following:

I have read and understood the University's disciplinary regulations concerning conduct in examinations and, in particular, the regulations on plagiarism (*Essential Information for Students. The Proctors' and Assessor's Memorandum*, Section 9.6; also available at www.admin.ox.ac.uk/proctors/info/pam/section9.shtml). ☐

I have read and understood the Education Committee's information and guidance on academic good practice and plagiarism at www.admin.ox.ac.uk/edc/goodpractice. ☐

The project report I am submitting is entirely my own work except where otherwise indicated. ☐

It has not been submitted, either partially or in full, for another Honour School or qualification of this University (except where the Special Regulations for the subject permit this), or for a qualification at any other institution. ☐

I have clearly indicated the presence of all material I have quoted from other sources, including any diagrams, charts, tables or graphs. ☐

I have clearly indicated the presence of all paraphrased material with appropriate references. ☐

I have acknowledged appropriately any assistance I have received in addition to that provided by my supervisor. ☐

I have not copied from the work of any other candidate. ☐

I have not used the services of any agency providing specimen, model or ghostwritten work in the preparation of this thesis/dissertation/extended essay/assignment/project/other submitted work. (See also section 2.4 of Statute XI on University Discipline under which members of the University are prohibited from providing material of this nature for candidates in examinations at this University or elsewhere: http://www.admin.ox.ac.uk/statutes/352-051a.shtml#_Toc28142348.) ☐

The project report does not exceed 50 pages (including all diagrams, photographs, references and appendices). ☐

I agree to retain an electronic copy of this work until the publication of my final examination result, except where submission in hand-written format is permitted. ☐

I agree to make any such electronic copy available to the examiners should it be necessary to confirm my word count or to check for plagiarism. ☐

Candidate's signature: …….. **Date:** ……..

Abstract

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 flash 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 was to produce a MATLAB simulation of how an error correction system using LDPC codes works for flash memory. By combining both an error generation and an error correction model, it was possible to benchmark these rediscovered codes against current coding schemes.

The results showed the importance of accurately modelling the underlying noise process, since by doing so increases the performance of any soft-decision decoder. However, even close approximations to the noise process result in improved performance over existing hard-decision schemes.

Acknowledgements

I would like to express my utmost gratitude towards my supervisor, Justin Coon, for all of the assistance he has provided over this past year, both for the project as well as in his related lecture series.

I would also like to thank Mohamed Ismail and Toshiba Research Europe Ltd. for providing the use of compute resources for this project.

Finally I wish to thank Hachem Yassine for his assistance in code debugging, and the discussions surrounding the project area.

Contents

1 Introduction

Flash Memory is a multi-billion dollar industry, and is almost certainly set to grow in the future [1]. It is also a highly competitive one, with consumers eager for low cost yet high volume storage. Innovation in the industry therefore seeks to increase the capacity and performance of these drives whilst keeping manufacturing costs low.

An obvious way to achieve low cost per Gigabyte is to increase the number of stored bits per cell without increasing the number of physical memory cells. Indeed, most consumer high capacity solid-state drives use this approach, making use of MLC (Multi-level cell) or TLC (Triple-level cell) technology where 2 or 3 bits of information are stored per cell, respectively [1]. However this comes at the cost of reducing read/write speeds, as well as increasing the occurrence of bit-errors on the device.

Current generation flash memory therefore makes extensive use of Error-Correcting Code, to offset the increased error rate caused by densely packed cells. However, recent advances in error-correction capability elsewhere have yet to be implemented in consumer flash memory devices. This project therefore aims to investigate the use of the latest generation error-correcting codes in flash memory.

Previous papers [1, 2] on this topic have investigated the use of more modern error-correcting codes, including a type known as Low Density Parity Check (LDPC) codes and have shown performance improvements over existing coding schemes. In addition, these papers present a more accurate noise model for flash memory that takes into account the program/erase nature of the memory cells. Previous work on LDPC codes is extensive [3, 4], particularly by MacKay who rediscovered them. The decoding method for these codes, the Belief Propagation algorithm, is also widely used [5, 6], and in addition methods for simplifying this decoding method through the use of approximations [7, 8, 9].

The majority of this project concerns the development of a MATLAB simulation, that includes both an error generation and error correction model. The flash memory error model, used to simulate the effects of noise and time-varying degradation in a memory cell, is mostly derived from existing work. So too are the methods for error correction, in the form of LDPC and the Belief Propagation algorithm. However, the novelty of this project is investigating how varying degrees of prior information provided to the decoder can alter error performance, through using approximations of the underlying noise assumptions. It also seeks to show the importance of modelling the noise correctly, since having an accurate noise model will result in superior system performance.

Firstly, this report presents an overview of flash memory and error-correction, in the form of linear

block codes. The, the process of decoding these error-correcting codes is presented. Subsequently, the simulation of one type of channel model, the Additive White Gaussian Noise (AWGN) channel, is performed and the results compared to a known data set. Finally, the remainder of the report focuses on creating a specific ‘memory noise model’, simulating it, and comparing the various decoding schemes used.

2 Overview of Flash Memory Technology

A flash memory cell is essentially just a floating gate transistor. The threshold voltage of each transistor can be individually programmed to determine what data is stored in that cell. A Single-Level Cell (SLC) has 2 states: erased and programmed. Whereas a Multi-Level Cell (MLC) often has 4 states (1 erased and 3 different programmed levels), allowing it to store 2 bits of information.

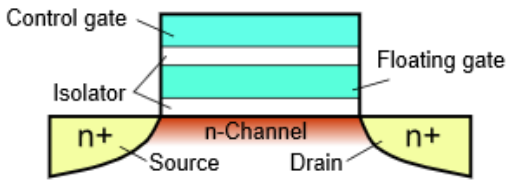

Figure 1: Floating gate transistor
[Image licensed under CC BY-SA 4.0, via Wikimedia Commons]

The floating gate transistor, shown here in figure 1, looks like a traditional MOSFET, except it has two gates, separated by an oxide layer. The principal of operation is that when charge is placed onto the floating gate, it shields the control gate from the channel of the device. If the floating gate is negatively charged through addition of electrons, to make the channel conductive, a larger gate voltage is now required. Therefore, the addition or removal of charge onto the floating gate "modulates" the effective threshold voltage (V_t) of the device [10].

To determine the threshold voltage of each cell, a voltage is applied to the control gate, and the subsequent drain current is then measured. A current comparator for each cell can then detect the current [2], and classify if the cell was in the conductive or cut-off region. If the channel becomes conductive after applying a specific gate voltage, it means that there was not sufficient charge on the floating gate, and so the effective threshold voltage of the device is low - the erased state. Conversely, if the channel does not conduct after applying the gate voltage, then the floating gate is charged and shielding the channel, and the effective threshold voltage of the device is now high - the programmed state.

Programming the memory cell is done through a process known as incremental step pulse programming (ISPP) [1]. A target programmed voltage, V_p, is reached through a repeated "program-and-verify" strategy: A high-voltage pulse, starting at some value below the target, is applied to the control gate. The read process as above then determines the effective threshold voltage. If the target voltage is not reached, this pulse is increased by an amount ΔV_{pp}, the "Pulse Width", and applied again. This

continues until the threshold voltage of the cell is correct. This programming process therefore results in the cell's threshold voltage taking on a uniform distribution, between $[V_p, V_p + \Delta V_{pp}]$.

Whilst the programmed state caused by ISPP takes on the shape of a uniform distribution, the erased state is assumed to be a wide Gaussian [1]. The voltage probability distribution for an ideal programmed MLC device is shown below in figure 2. An SLC device would only have a single uniform distribution, causing 2 discrete states, and therefore could only store 1 bit of information per cell.

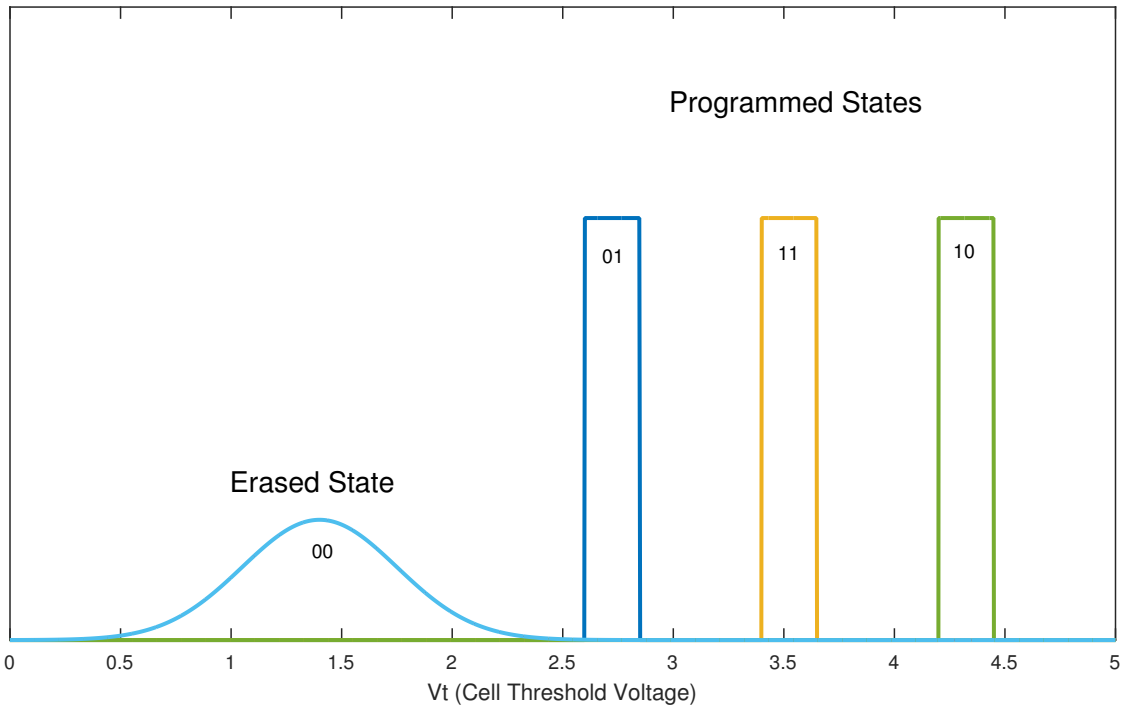

Figure 2: Ideal voltage distribution for MLC. This cell can store 2 bits of data, dependent on V_t being $\sim 1.4, 2.6, 3.4$ or 4.2 volts. Here, the pulse width ΔV_{pp} is 0.25 volts.

Since no dielectric is perfect, charge stored on the floating gate transistor will slowly "leak out" of the gate over time. This generally results in the threshold voltage of cells decreasing over time. This process is one of the causes of noise generation that is used in this project, and is described in full in section 6.

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 [11, 12, 13].

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 patterns 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:

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

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 [14]. 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 [14]. 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 core intellectual property restrictions [14, p.90], making them attractive for real world use.

LDPC codes were originally discovered by R.G. Gallager in 1962 [4]. Then known as “Gallager codes”, they were defined by a sparse parity check matrix with low column weights. Gallager also worked on a probability 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 David MacKay in 1996 [3]. 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’ [15]. 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 6.

[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.

4 Decoding of LDPC Codes

Using syndrome lookup decoding as described in section 3.1 would be nearly impossible for longer block lengths. There are much better, iterative decoding methods that can be used for any linear block code, and which are linear in block length. These are called 'belief propagation algorithms', and more specifically used here is the 'sum-product algorithm'. Other alternatives also include the 'min-sum 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.

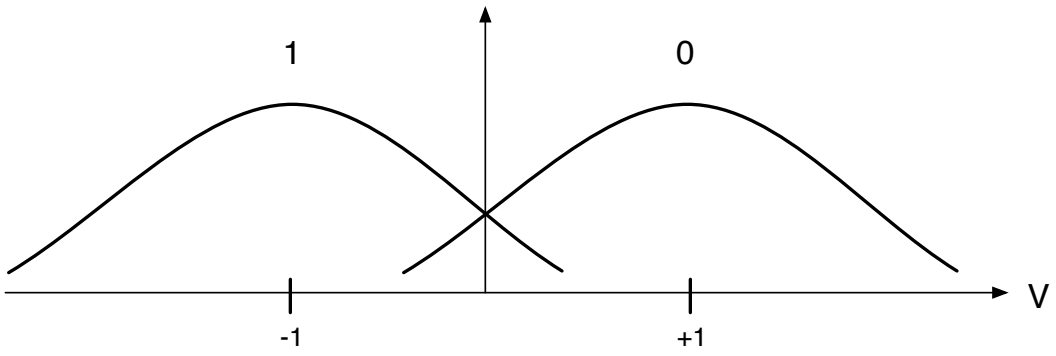

Figure 3: Received voltage probability distribution for AWGN channel

Figure 3 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 *probabilities* 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.

4.1 Hard decision decoding

The message passing algorithm can be best understood with hard decision decoding. As an example, eq 4.1.1 shows the parity check matrix $\mathbf{H}$ of an (8,4) code. This code can also be displayed, as in figure 4, as a visual graph representation known as a tanner graph. [16]

$$\mathbf{H} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{4.1.1}$$

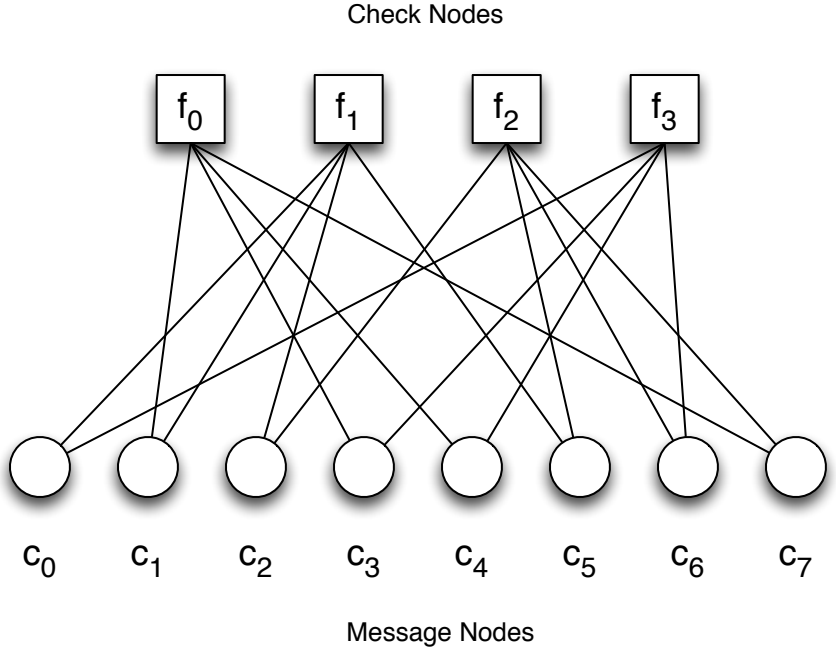

Figure 4: Tanner Graph

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 [17]:

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 its 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_i, 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 $f_{1,2,3}$

4.2 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*.

$$\mathcal{L}(c|y) = log_e \left[\frac{Pr(c = +1|y)}{Pr(c = -1|y)} \right] \tag{4.2.1}$$

The term $\mathcal{L}(c|y)$ [18] is the likelihood of c being transmitted given that y was received. $Pr(c = +1|y)$ is the conditional probability of obtaining $c = +1$ given the received value y. Thus, the log-likelihood ratio is able to tell us what the most likely transmitted symbol was. 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],

$$\begin{aligned} \mathcal{L}(c|y) &= log_e \left[\frac{Pr(y|c = +1)\dfrac{Pr(c = +1)}{Pr(y)}}{Pr(y|c = -1)\dfrac{Pr(c = -1)}{Pr(y)}} \right] \\ &= log_e \left[\frac{f(y|c = +1)}{f(y|c = -1)} \right] \\ &\sim log_e \left[\frac{\textit{“Probability density function of +1”}}{\textit{“Probability density function of -1”}} \right] \end{aligned} \tag{4.2.2}$$

where $f(y|c = +1)$ is now the Probability Density Function (PDF) of receiving $c = +1$. 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 LLR.

[2] $Pr(A|B) = \dfrac{Pr(B|A)Pr(A)}{Pr(B)}$

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

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 [18],

$$\begin{aligned}\mathcal{L}(c|y) &= log_e\left[\frac{\mathcal{N}(1,\sigma^2)}{\mathcal{N}(-1,\sigma^2)}\right] \\ &= log_e\left[\frac{e^{-(y-1)^2/(2\sigma^2)}}{e^{-(y+1)^2/(2\sigma^2)}}\right] \\ &= log_e\left[e^{4y/(2\sigma^2)}\right] \\ &= \frac{2y}{\sigma^2}\end{aligned} \tag{4.2.3}$$

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.

4.3 Soft decision decoding

The Sum-Product 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.

Previous work [3, 4, 17] has proved the set of equations that are used at both the message nodes and check nodes. The 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.

$$m_{ij}^{(l)} = L_i + \sum_{j' \in C_i \neq j} m_{j'i}^{(l-1)} \tag{4.3.1}$$

Equation 4.3.1 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.

$$m_{ji}^{(l)} = 2 \operatorname{arctanh} \left[\prod_{i' \in V_j \neq i} \tanh(\frac{m_{i'j}^{(l-1)}}{2}) \right] \tag{4.3.2}$$

Equation 4.3.2 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 4.3.2, 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 $\mathbf{s} = \mathbf{cH}^\top$, 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 4.3.1, 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 $l = 50$).

4.4 Min-Sum approximation

The full sum-product algorithm is computationally expensive, with the check-node update equation requiring calculation of hyperbolic tangents. A simplified version of Belief Propagation, known as the "min-sum" algorithm, reduces decoding complexity at the cost of slightly reduced performance [8].

$$m_{ji}^{(l)} = \alpha \times \left(\prod_{i' \in V_j \neq i} \operatorname{sign}(m_{i'j}^{(l-1)}) \right) \times \min_{i' \in V_j \neq i} \left| m_{i'j}^{(l-1)} \right| \tag{4.4.1}$$

Equation 4.4.1 is the check node update equation for the min-sum approximation [9]. The message node update equation is the same as the full sum-product algorithm (eq. 4.3.1). Essentially, when

a check node receives incoming messages, the return messages are just the minimum value of the received ones (bearing in mind however, the extrinsic information rule still applies). An additional term, $0 < \alpha < 1$, is the normalisation constant, which helps to counter the fact that the check node step in min-sum generally produces an over-estimate, and including it improves performance [9].

Min-sum is useful since implementation is meant to be less complex than the full sum-product algorithm, particularly on dedicated decoding hardware [8], such as ASIC devices. In addition, it has been shown that in cases where full soft information is unavailable, and quantisation error is introduced, min-sum can actually perform as well as the full sum-product algorithm [7].

Initial testing of using the min-sum approximation in MATLAB showed that it performed very closely to the full sum-product scheme (Figure 5; exact simulation method described in the subsequent section). However, this particular implementation of min-sum was actually slower in MATLAB compared to sum-product[4]. Because of this, and the fact that its performance is so similar, all subsequent simulations in this project used the full sum-product algorithm.

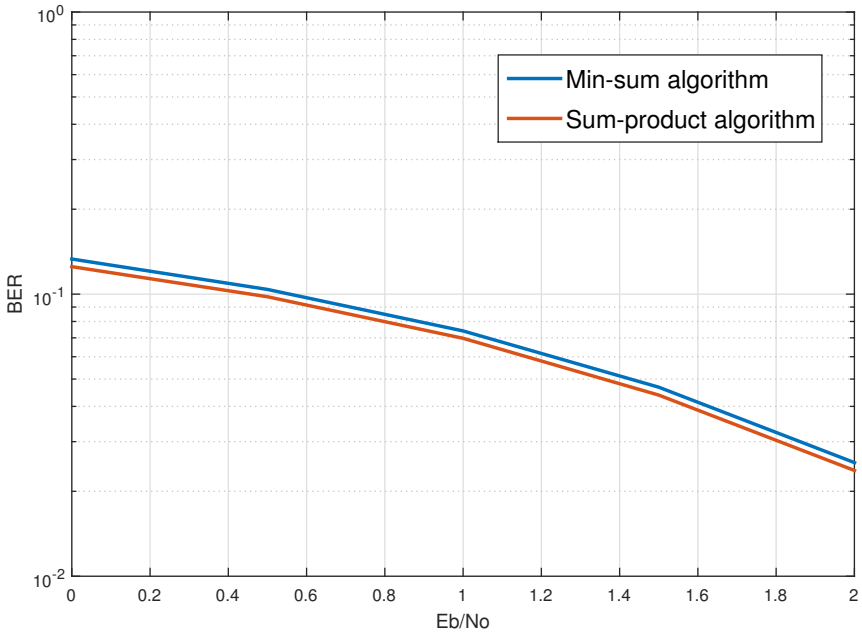

Figure 5: Bit Error Rate simulation comparing MS and SP algorithms

It is noted, however, that further research into using min-sum in the context of flash memory might be interesting, particularly with the constraints requiring quantisation of the soft information into discrete bins.

[4]MATLAB is able to calculate hyperbolic tangents in parallel very efficiently, whereas constructing the necessary matrices for min-sum was actually fairly slow

5 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 [11, 12, 13, 19]. 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.

5.1 Simulation model

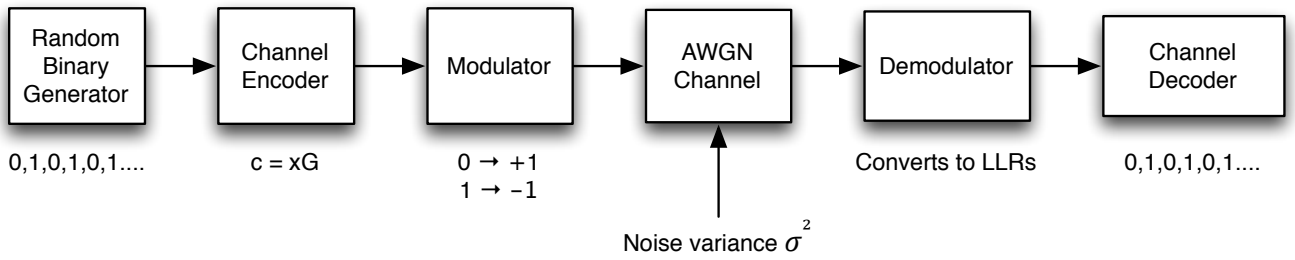

Figure 6: The AWGN simulation model

- 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 4.2.3, $\mathcal{L} = \dfrac{2y}{\sigma^2}$, 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[5].

Converting between σ^2 and $\frac{E_b}{N_0}$ is relatively trivial [20]:

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

$$\sigma^2 = \frac{N_0}{2} \tag{5.1.1}$$

$$E_s = R_c R_m E_b \tag{5.1.2}$$

N_0, the noise power, is defined as in 5.1.1. The energy per symbol is defined in 5.1.2: 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 $\frac{E_b}{N_0}$:

$$\begin{aligned} \frac{E_b}{N_0} &= \frac{E_s}{R_c R_m N_0} \\ &= \frac{E_s}{R_c R_m 2\sigma^2} \\ &= \frac{1}{2R_c\sigma^2} \end{aligned} \tag{5.1.3}$$

$E_s = R_m = 1$ 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.

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

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 6 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.

5.2 Simulation results

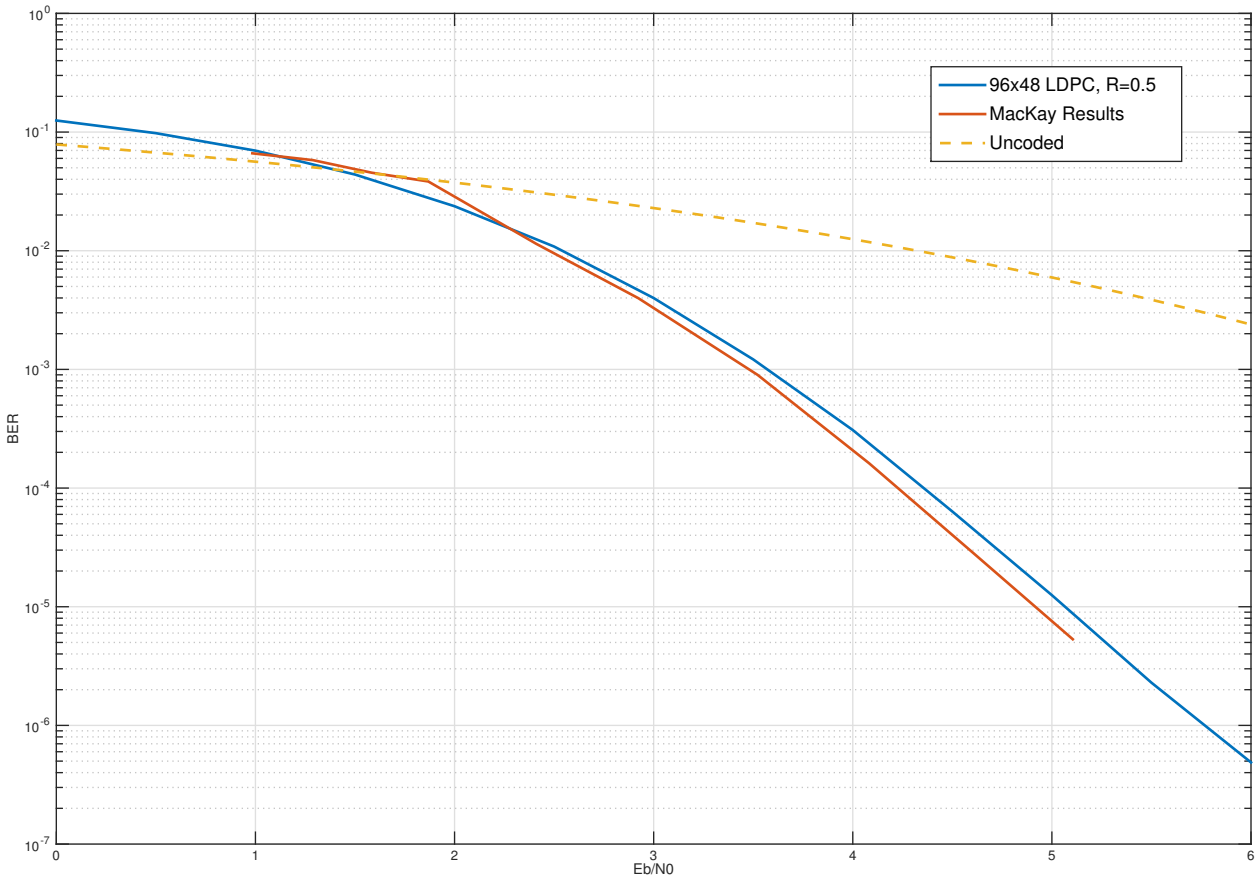

Figure 7: Simulated results of 96,48 LDPC code

A short block length, rate $\frac{1}{2}$ LDPC code was used to produce the results in figure 7, using the Sum-Product algorithm with soft decision decoding. The figure also displays the results obtained by MacKay using the same LDPC code and decoding method. There is a minor performance loss compared to MacKay, but otherwise the results are in agreement. Additionally, the dashed line indicates the theoretical expected bit error rate in the event that no error correction code was used [14]. 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.

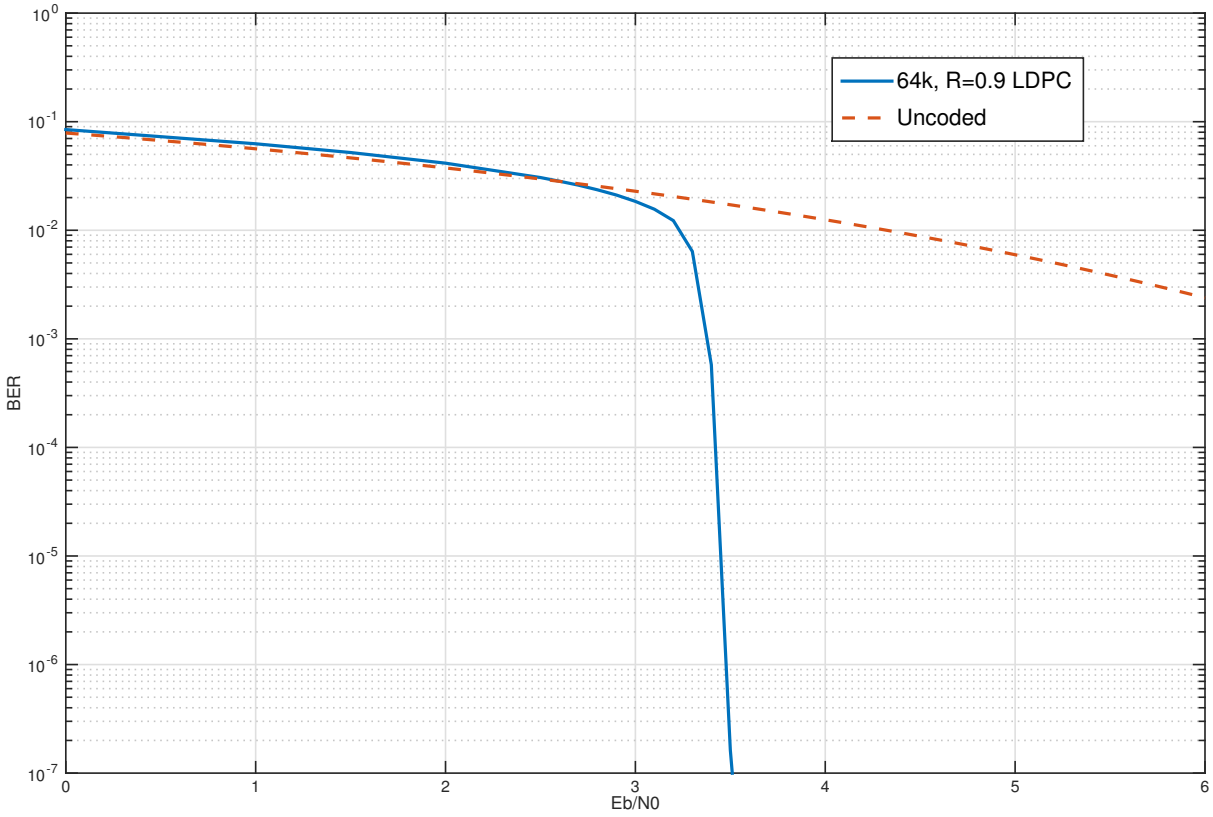

Figure 8: Simulated results of 64k DVB-S2 LDPC code

Longer block length codes generally achieve superior performance and greater 'cut-off' compared to shorter codes. Figure 8 demonstrates a code with a block length of 64000, and a substantially higher rate ($R = 0.9$). This particular code is a standardised construction, used in the DVB-S2 (Satellite broadcasting) specifications. This code is also the primary one used in this project for the memory error correction later. One benefit of using this code is its sharp 'cut-off', compared to the previous code in figure 7.

5.3 Conclusion

Simulating the AWGN channel, and then comparing the results to the known data published by MacKay, allows for some degree of verification of the MATLAB program. The AWGN channel is one of the common channels simulated when dealing with error correcting code, since it simulates the effect of random noise that often occurs in the environment.

One issue identified with the simulation program was the speed of the decoder. In the first version of the decoder program [Appendix A, extract 2], there were a substantial number of 'for' loops, which are generally slow in MATLAB. Subsequent versions of the decoder attempted to fix this by vectorising the code [Appendix A, extract 3], resulting in approximately a 2x speed increase. However, even with these improvements, the decoder was still much slower compared to MATLAB's built-in LDPC decoder [21]. In addition, freely available OpenCL accelerated decoders [22] can be used on a GPU (Graphics Processing Unit) which are even faster than MATLAB's implementation (See Appendix B for the benchmark comparisons). In the subsequent sections simulating the memory model, the MATLAB built-in implementation of the LDPC decoder was used, allowing for substantially more blocks to be decoded in the same amount of time.

Whilst subsequent simulations made use of the built-in MATLAB decoder, the error performance of the original decoder was identical to that of MATLAB's. Additional testing of the original decoder against the built-in decoder on the memory model also produced identical results. The only reason we are making use of a library function, in this case, is for the massive speed improvements it offers.

6 Modelling a memory-specific noise channel

There are many similarities between modelling a communications system and flash memory. Both are binary channels, subject to some binary input, distortion by some noise source, and subsequent decoding. However, the major difference in flash memory is the storage of binary data in the device for a length of time, unlike a communications system where the data is 'in transit'. When a bit is stored to a memory cell, the noise effect is cumulative over the lifetime of the stored bit. Therefore, two major parameters in modelling flash memory are time, and the number of read/write (or program/erase) cycles each cell is subject to.

As stated previously in section 2, it is the threshold voltage of each gate that defines the binary value of the memory cell. In this project, the flash memory is assumed to be of SLC design, that is 1 bit per cell. It is also assumed that a low gate threshold voltage corresponds to a binary 0, whilst a higher threshold voltage corresponds to a binary 1.

The primary noise source, which is caused by repeated P/E (program/erase) cycling, can be split into two physical phenomena [1]: Electron capture and emission events, which result in a random variation of the threshold voltage, and interface trap recovery and electron de-trapping, which gradually reduces the gate threshold voltage over time. The former is known as 'Random Telegraph Noise' (RTN), whilst the latter is often called the 'Retention Noise', since it sets a time limit on how long data can be stored in a cell until the threshold voltage becomes too low.

Another noise source, known as 'Cell-to-cell interference', which as the name implies is the result of neighbouring memory cells interfering with each other. This dependence between neighbouring cells adds additional complexity when modelling the system, since it is no longer possible to model one cell at a time. As a result, it was decided for this project not to include this noise source, instead focusing on an individual memory cell.

6.1 Description of Noise Sources

The first noise source in the memory channel, the random telegraph noise (RTN), can be modelled as a Laplacian distribution, $Y_{rtn} \sim \mathcal{L}(\lambda_{rtn})$, whose probability density function is [1]:

$$p_{rtn}(x) = \frac{1}{2\lambda_{rtn}} e^{-|x|/\lambda_{rtn}} \tag{6.1.1}$$

In equation 6.1.1, x is the value of the voltage fluctuation, and λ_{rtn} scales with N, the P/E cycling number. In this project, $\lambda_{rtn} = K\sqrt{N}$ where K is just a constant. Therefore as the memory cell is subject to more P/E cycles (N), the RTN distribution widens, as expected. Figure 9 shows the probability density function of the distribution for different values of N. The distribution is centred about $x = 0$, since the random effect of RTN can result in either an increase of decrease of the cell threshold voltage.

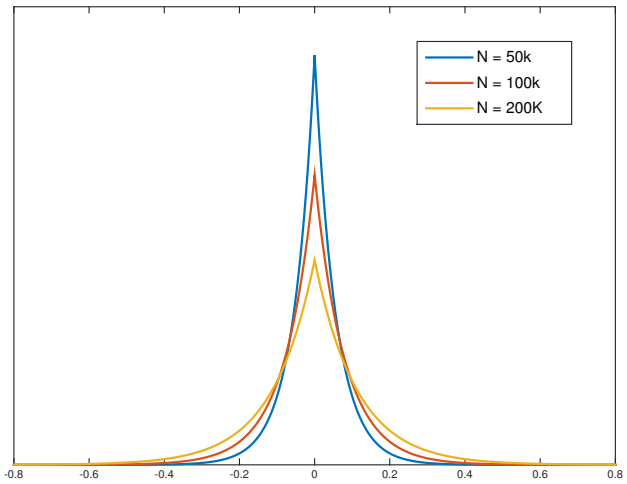

Figure 9: PDF of RTN distribution

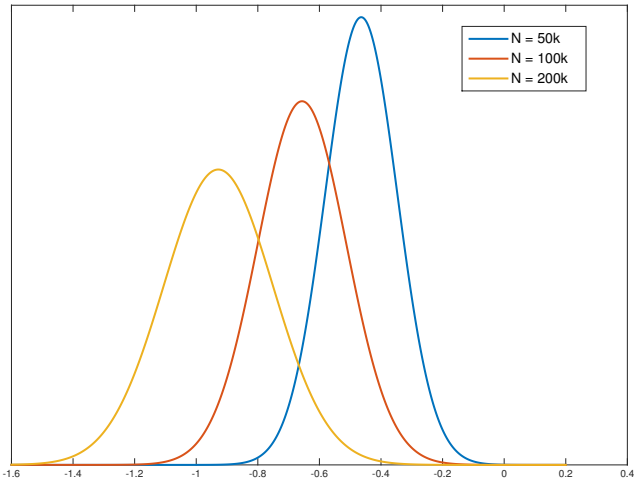

Figure 10: PDF of Retention noise distribution

The second noise source, the retention noise, is modelled as a Gaussian, $Y_{retention} \sim \mathcal{N}(\mu_r, \sigma_r^2)$, with the mean and variance defined as [1]:

$$\begin{aligned} \mu_r &= -K_s K_d (V_p - V_e) N^{0.5} \ln\left(1 + \frac{t}{t_0}\right) \\ \sigma_r^2 &= K_s K_m (V_p - V_e) N^{0.6} \ln\left(1 + \frac{t}{t_0}\right) \end{aligned} \tag{6.1.2}$$

In equation 6.1.2, V_p is the initial voltage of the programmed state, V_e is the (mean) voltage of the erased state, t is the current memory retention time, and t_0 is an initial starting time. $K_{s,d,m}$ are all constants. The retention noise therefore has a power-law relationship with the P/E cycling number N, a logarithmic relationship with retention time, and is also dependent on the initial threshold voltages. Figure 10 shows the probability density function of the retention noise for different values of N, with the remaining variables fixed. Also note that since the mean (μ_r) is always negative, the effect of the Retention noise is to always reduce the gate threshold voltage. As N increases, the programmed state voltage will therefore reduce and get closer to the erased state voltage. This is perhaps the main source of bit error: the interference between the erased and programmed states.

6.2 Channel Model

Both the RTN and retention noise are independent random variables. Therefore the threshold voltage of any given cell is simply the sum of the random variables. As explained previously in section 2, each cell is either a binary 0 or 1, with a corresponding threshold voltage distribution for each. Therefore in the ideal case each cell will have a single voltage value, either V_{p_0} for a programmed cell or V_{e_0} for an erased cell. After noise, the final threshold voltage of the cell is then

$$\begin{aligned} V_p &= V_{p_0} + Y_{rtn} + Y_{retention} \\ V_e &= V_{e_0} + Y_{rtn} \end{aligned} \tag{6.2.1}$$

where V_p and V_e are the final programmed and erased voltage states, respectively. Retention noise is not added onto the erased state in this model, since we are assuming that the erased state is the reference voltage level that the retention noise is derived from.

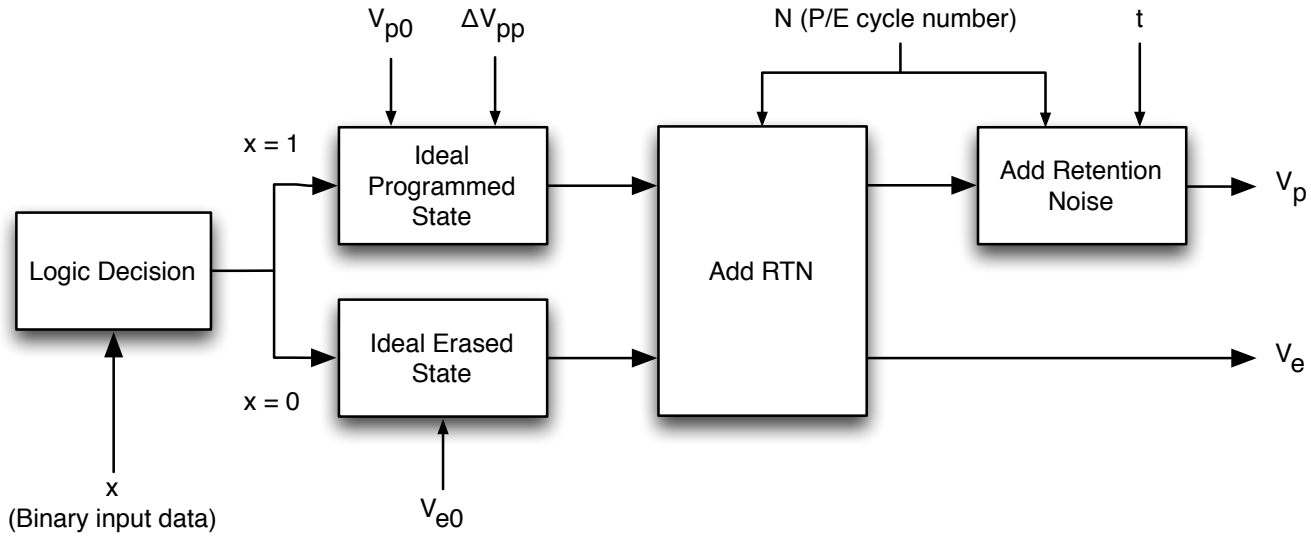

Figure 11: Memory channel noise model

Figure 11 shows the block diagram of the memory channel noise model. Unlike the AWGN channel, the noise added to each binary bit depends on whether it is a 0 or a 1. The input to the model is a binary value, whilst the output from the model is the final threshold voltage value of the memory cell, ready to be decoded. Effectively, the model simulates the programming of a memory cell, and then the subsequent noise that would be experienced if the cell were subjected to repeated program and erase cycles.

Figure 12 shows the probability density functions obtained from the memory model in figure 11. Note that the x axes are the gate threshold voltages. Various values for N are displayed as well as

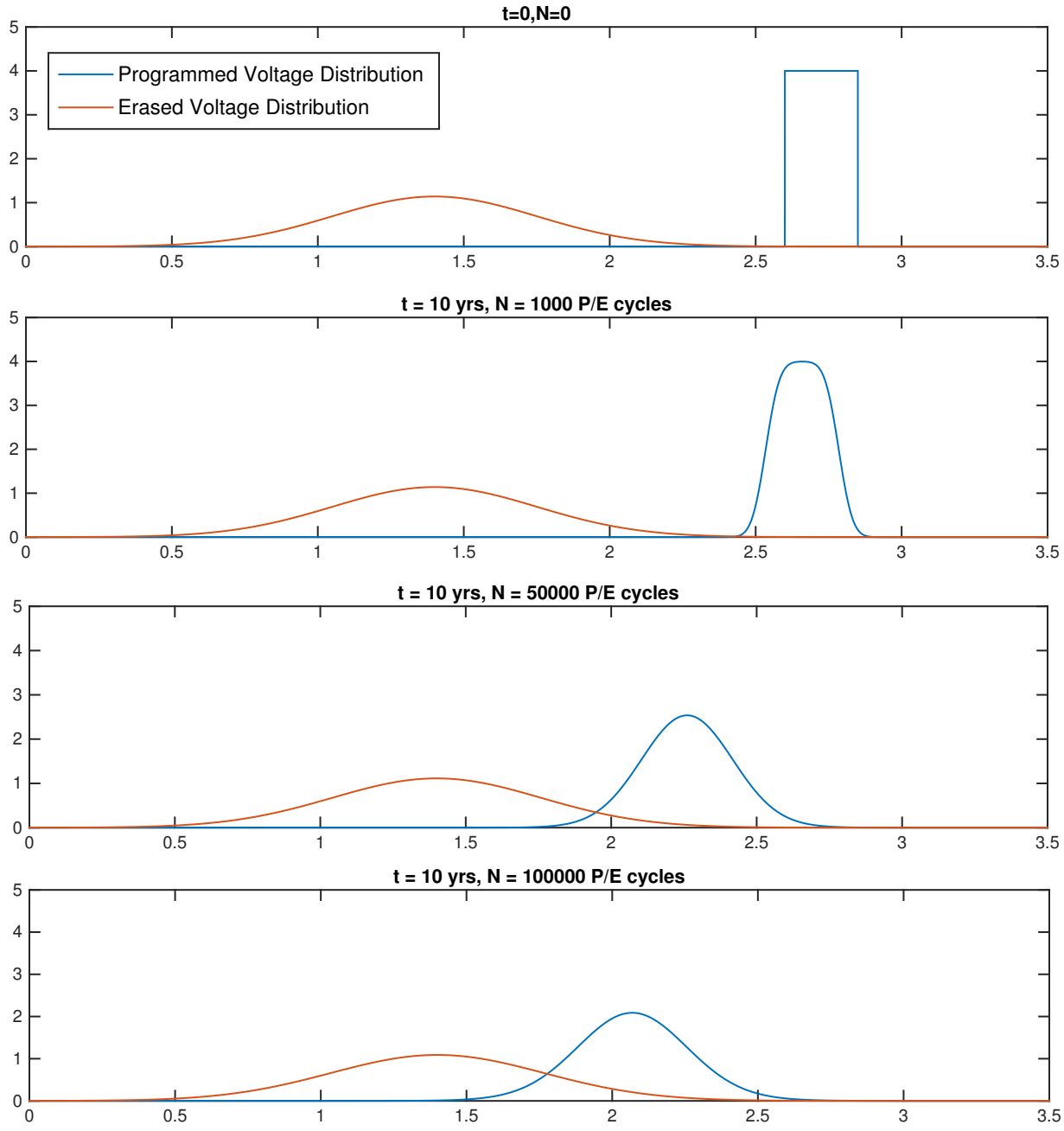

Figure 12: Density functions of memory threshold voltages for differing values of N

showing the initial programmed state (the uniform distribution) when $N = 0$. As the number of P/E cycles increases, the programmed distribution shifts left and widens, and interferes with the erased state. It is now clear that a simple hard-decision decoder without any error correction would generate a substantial number of errors when, in this case, $N > 50,000$. The distributions overlap, which would lead to incorrectly decoded bits.

7 Decoding for the memory model

In section 4, two primary methods for error correction decoding were presented: Soft decision decoding and hard decision decoding. For the memory model, both types of decoding can be used, however with some substantial changes compared to the simplistic AWGN case.

The AWGN model assumes that both symbols $\{$+1,-1$\}$, after noise, produce symmetric Gaussian distributions (Figure 3, page 13). For hard decision decoding, it is obvious where the boundary should be placed in this case: the midpoint between the two distributions. For soft decision decoding, obtaining the log-likelihood ratios is relatively simple, as explained previously in equation 4.2.3, whereby the raw voltage values are just divided by the noise variance.

The memory channel noise model is, however, not two symmetric Gaussians. As demonstrated in figure 12, the programmed distribution's mean is not static: it shifts to the left as N increases. Also, it is not immediately clear what the overall distribution is, since we have added 2 sources of noise to an originally uniform distribution.

7.1 Hard decision decoding: Variable boundary

To use hard decision decoding, it is only necessary to classify each symbol in the memory model as either a 0 or 1. To do this, the system needs to determine where the boundary between these regions should be. The main issue here is that as the number of program/erase cycles increases, both the shape and mean of the programmed distribution changes significantly. A static boundary between the two regions would therefore result in very poor performance. As an example, using figure 12, setting a crossover voltage of 2 volts may seem somewhat reasonable even up to 50,000 cycles. But at 100,000 cycles this decision region would result in nearly half of all programmed 1's being decoded as 0's.

The obvious solution is to have a variable decision boundary, that moves position dependent on the number of program/erase cycles the memory cell has undergone. Mathematically, this position should seek to ensure an equal probability of obtaining a 1 or 0 either side of the boundary. Or, the corollary of this is that the position of the boundary should ensure that the probability of *incorrectly* decoding a 1 or 0 is the same. This is effectively Maximum Likelihood decoding.

If we let $P(N, x)$ equal the probability density function of the programmed (1) state at N cycles,

and $E(N, x)$ be the density function of the erased (0) state at N cycles, then

$$\int_0^x \mathrm{P}(N, x)\,\mathrm{d}x = \int_x^\infty \mathrm{E}(N, x)\,\mathrm{d}x \tag{7.1.1}$$

This could be solved for x to obtain the ideal boundary value for each value of N. However, it is not possible to immediately obtain the expressions for each density function, since in reality it is the combination of many different functions. To solve this problem more quickly, the solution was obtained using numerical integration in MATLAB.

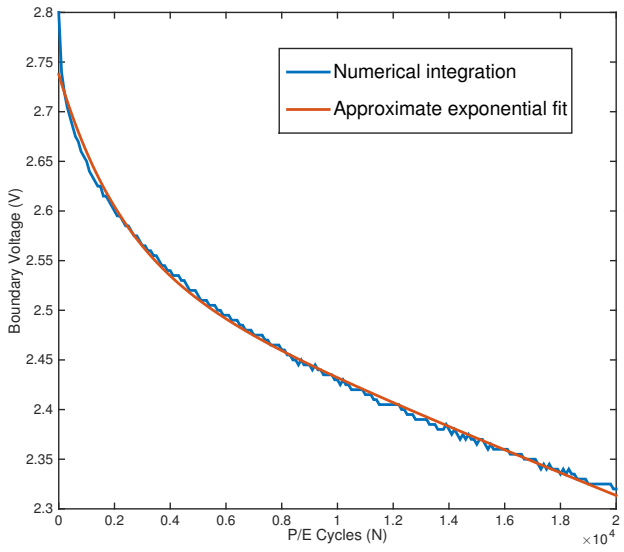

Figure 13: Results of numerical integration, showing ideal hard decision decoding boundary for given values of N

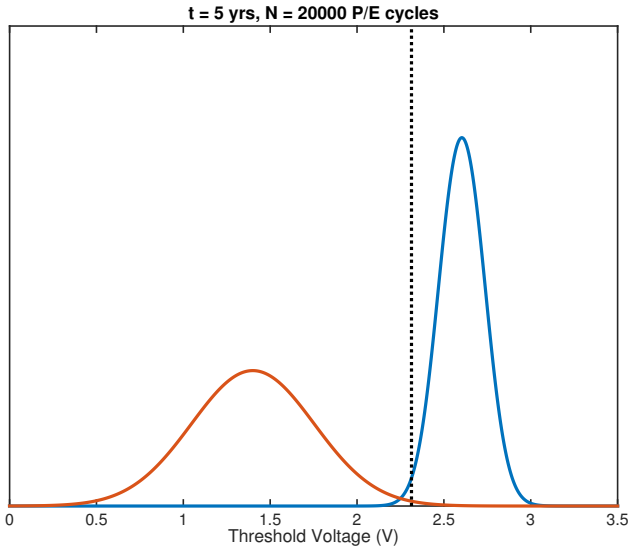

Figure 14: Example showing erased (red line) and programmed (blue line) distributions and the decision boundary (dotted line)

Figure 13 shows the results of the numerical integration, in order to obtain the optimal decision boundary for hard decision decoding. The boundary voltage, V, is the same as x for the analytic case. The results appear slightly jagged since the data was taken from histograms of each distribution, which themselves were created from random sampling. In order to make use of the data from the integration, in a closed form solution, a simple exponential fit was plotted over the results.

Figure 14 is an example of both the erased and programmed distributions for a specific P/E cycle number, $N = 20,000$. The dotted black line shows the optimal decision boundary, taken from the approximate exponential fit in figure 13 ($V \approx 2.31$). This boundary results in Maximum Likelihood decoding for both programmed and erased states.

7.2 Soft decision decoding: Gaussian approximations & non-Gaussian functions

As shown previously, soft decision decoding makes use of additional 'soft' information in order to improve the decoding performance. The main premise with SDD is to introduce probabilities into the decoding process, harnessing the *a priori* probabilities of the underlying distributions, in order to construct a probabilistic decoder.

Simply obtaining the exact threshold voltage of the memory cell is not sufficient to perform SDD. As with the AWGN model, some prior information (in that case the noise variance) is required in order to convert the received raw data into a likelihood value. Once a likelihood *ratio* of the received data has been calculated, exactly the same Sum-Product algorithm can be used as before. The difficulty in the memory model case, is obtaining this likelihood ratio.

$$\mathcal{L}(c|y) \sim log_e \left[\frac{\textit{"Probability density function of erased state, E(x)"}}{\textit{"Probability density function of programmed state, P(x)"}} \right] \tag{7.2.1}$$

Equation 7.2.1, which results from the earlier work in section 4.2, gives us the Log-Likelihood ratio that is required in order to perform soft decision decoding on the memory channel. It is clear that the necessary *a priori* information in this case is the entire Probability Density Function for both the erased and programmed states. Once the PDF's are known, the voltage value for the cell can be inputted and the requisite LLR obtained.

As said previously, obtaining an exact solution for the density functions is non-trivial. This arises since, as explained in equation 6.2.1, we are adding both Gaussian and Laplacian noise onto a Uniform distribution in the case of the programmed state, and adding Laplacian noise to a Gaussian distribution in the case of the erased state.

To evaluate the overall probability density functions, given the individual PDF's of each noise source, requires using the convolution property:

$$f_{1+2+...+N}(x) = (f_1 * f_2 * .... * f_N)(x) \tag{7.2.2}$$

That is, the probability density function resulting from the sum of independent random variables, is just the convolution of each random variable's own density function. In terms of the memory model, this means the overall density function representing the programmed state is the 3-way convolution of a Gaussian, Laplacian and Uniform distribution. For the erased state, it is the convolution between a Gaussian and

Laplacian.

Performing such a convolution analytically, whilst possible, results in a function with a large number of terms. For example, the resultant PDF of the programmed state contains 6 exponential and 6 error (erf) functions. Whilst using the full density function within the Log-Likelihood ratio gives the overall best performance, there may be ways to achieve similar performance by using a simpler approximation. In terms of the simulation, a simplified approximation would be able to run quicker in MATLAB, whilst in a real device, might be easier to implement.

7.2.1 Exact function: Retention Noise + RTN

The full convolution of the initial Uniform distribution with the retention noise and RTN density function, the result of which was obtained from [23], is shown below:

$$\begin{aligned} f_{prog}(\Delta V_T = x) = \frac{1}{4\Delta V_{pp}} e^{\frac{\sigma_r^2}{2\lambda^2}} \Bigg[& e^{-\frac{x-Vp-\Delta V_{pp}-\mu_r}{\lambda}} \cdot \mathbf{erfc}\left(\frac{\sigma_r}{\sqrt{2}\lambda} - \frac{x-Vp-\Delta V_{pp}-\mu_r}{\sqrt{2}\sigma_r}\right) \\ & - e^{\frac{x-Vp-\Delta V_{pp}-\mu_r}{\lambda}} \cdot \mathbf{erfc}\left(\frac{\sigma_r}{\sqrt{2}\lambda} + \frac{x-Vp-\Delta V_{pp}-\mu_r}{\sqrt{2}\sigma_r}\right) \Bigg] \\ - \frac{1}{4\Delta V_{pp}} e^{\frac{\sigma_r^2}{2\lambda^2}} \Bigg[& e^{-\frac{x-Vp-\mu_r}{\lambda}} \cdot \mathbf{erfc}\left(\frac{\sigma_r}{\sqrt{2}\lambda} - \frac{x-Vp-\mu_r}{\sqrt{2}\sigma_r}\right) \\ & - e^{\frac{x-Vp-\mu_r}{\lambda}} \cdot \mathbf{erfc}\left(\frac{\sigma_r}{\sqrt{2}\lambda} + \frac{x-Vp-\mu_r}{\sqrt{2}\sigma_r}\right) \Bigg] \\ + \frac{1}{2\Delta V_{pp}} \cdot \mathbf{erfc}\left(\frac{x-Vp-\Delta V_{pp}-\mu_r}{\sqrt{2}\sigma_r}\right) & - \frac{1}{2\Delta V_{pp}} \cdot \mathbf{erfc}\left(\frac{x-Vp-\mu_r}{\sqrt{2}\sigma_r}\right) \end{aligned} \tag{7.2.3}$$

Equation 7.2.3 is the probability density function of the cell threshold voltage V_T for the programmed state. V_p is the initial programmed voltage, ΔV_{pp} is the programming step voltage, μ_r and σ_r are the retention noise parameters from equation 6.1.2, and λ is the parameter describing the RTN.

7.2.2 Approximation: Retention Noise only

The magnitude of the RTN generated is substantially smaller than the magnitude of the retention noise. It therefore seems reasonable to ignore the effect of the RTN and perform the convolution of just the Uniform and Gaussian distributions for the programmed state. Starting with the definitions of both the retention noise and initial programmed state PDF's:

$$\text{Retention Noise: } Y_{retention} \sim \mathcal{N}(\mu_r, \sigma_r) = \frac{1}{\sigma_r\sqrt{2\pi}} \exp\left\{-\frac{(x-\mu_r)^2}{2\sigma_r^2}\right\} \tag{7.2.4}$$

$$\text{Initial State: } V_{p_0} \sim \mathcal{U}(a,b) = \begin{cases} \dfrac{1}{b-a}, & \text{for } x \in [a,b] \\ 0, & \text{otherwise} \end{cases} \tag{7.2.5}$$

The convolution property from equation 7.2.2 can now be used to obtain the overall PDF:

$$\begin{aligned} V_{p_0}(x) * Y_{ret}(x) &= \int_{-\infty}^{\infty} V(\tau) Y(t-\tau)\,\mathrm{d}\tau \\ &= \frac{1}{b-a}\int_a^b \frac{1}{\sqrt{2\pi\sigma_r^2}} \exp\left\{-\frac{(t-\tau-\mu_r)^2}{2\sigma_r^2}\right\}\mathrm{d}\tau \end{aligned} \tag{7.2.6}$$

By comparing the necessary convolution in 7.2.6 to that of the Normal distribution's cumulative distribution function (equation 7.2.7), the convolution can now be calculated as in equation 7.2.8:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-t^2/2}\,\mathrm{d}t \tag{7.2.7}$$

$$V_{p_0}(x) * Y_{ret}(x) = \frac{1}{b-a}\left[\Phi\left(\frac{(x-a)-\mu_r}{\sigma_r}\right) - \Phi\left(\frac{(x-b)-\mu_r}{\sigma_r}\right)\right] \tag{7.2.8}$$

This result shows that by adding Gaussian noise to a Uniform distribution, you obtain the difference between two normal CDF's, scaled by the Uniform width.

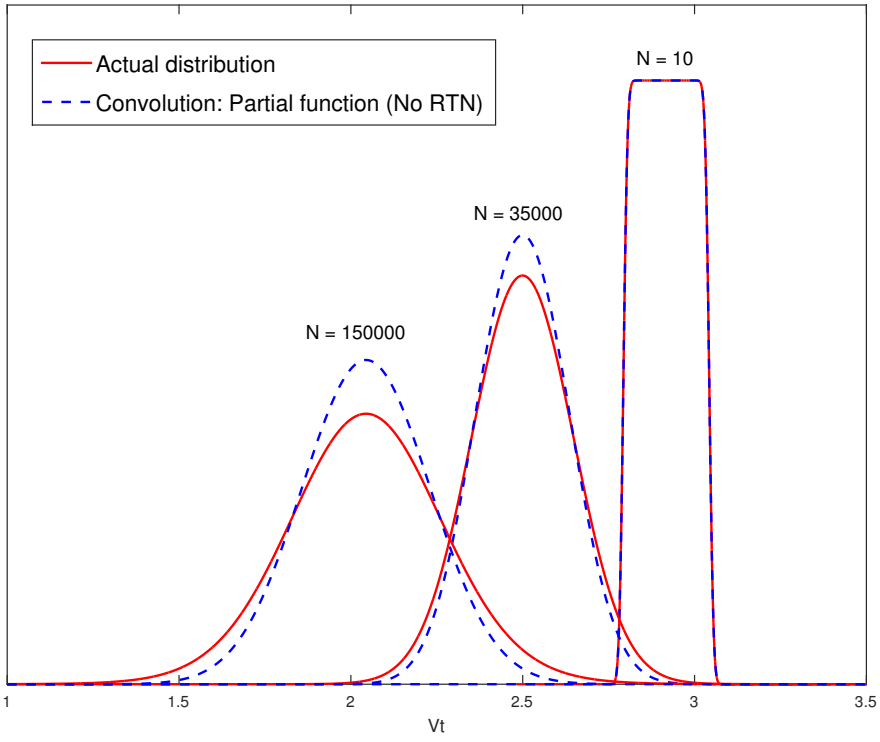

Figure 15: Comparison of the distributions for different values of N

Figure 15 shows that the convolution method, when excluding RTN, produces the correct shape of distribution for small N, but as N increases the difference between the distribution widens. This is because the RTN increases as N increases.

7.2.3 Approximation: Matched Gaussians

Another approach to obtain the distribution of the threshold voltage, is to use a reasonable approximation of the function, rather than trying to obtain it directly. Using a simplified approximation also turns out to be faster[6].

An obvious choice would be to use a Gaussian. Whilst the distribution starts off as a Uniform distribution, adding noise causes it to quickly degrade into a Gaussian-like shape. Since the memory noise model, as explained in equation 6.2.1, is addition of independent random variables, the mean and variance of each random variable can simply be added together to obtain the overall mean and variance of the distribution. These values can then be used to generate a 'Matched Gaussian', which will have the same mean and variance as the actual underlying function, but just with a slightly different shape.

$$\begin{aligned} \mu_p &= \mu_{p_0} + \mu_{rtn} + \mu_{retention} = \frac{2V_p + \Delta V_{pp}}{2} + 0 + \mu_r \\ \sigma_p^2 &= \sigma_{p_0}^2 + \sigma_{rtn}^2 + \sigma_{retention}^2 = \frac{\Delta V_{pp}^2}{12} + 2\lambda^2 + \sigma_r^2 \end{aligned} \tag{7.2.9}$$

Thus, the equations in 7.2.9 show that the overall mean is just the mean of the uniform distribution plus the mean of the retention noise μ_r, since the RTN distribution has a mean of zero. The variance is just the sum of the Uniform, Laplacian and Gaussian noise variances.

Figure 16 shows that using an approximation of the actual distribution can give, at least by inspection, very close results. Whilst the shape of the uniform distribution for small values of N is not successfully captured, for larger values of N the distributions appear to be a close match.

[6] Appendix B, table 2: LLR's are generated 5x faster in MATLAB compared to the full function in section 7.2.1

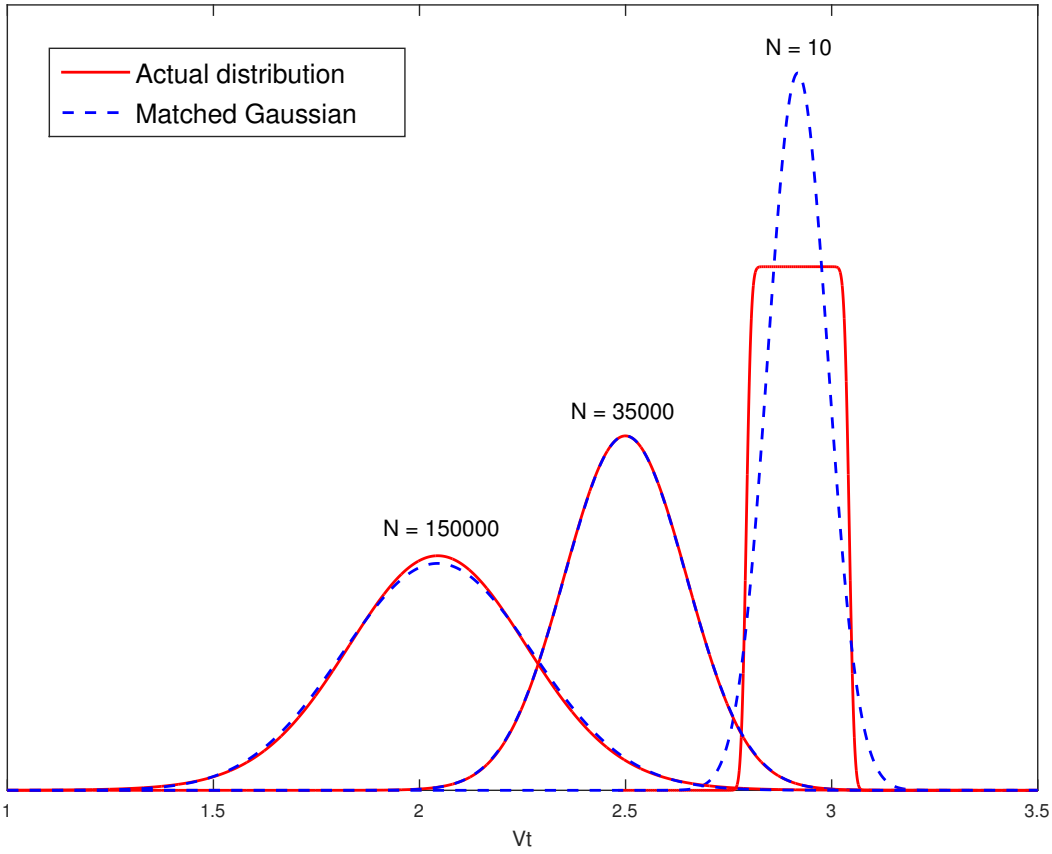

Figure 16: Comparison of the distributions for different values of N

8 The memory channel: Simulation & Results

8.1 Simulation Model

The process for simulating the error correcting performance of the flash memory model is similar to the AWGN channel model that was used earlier. For the memory channel, a block of random binary data was generated, encoded to form an LDPC codeword, mapped to a cell threshold voltage, subject to some form of noise, detected and then finally decoded by the Belief Propagation algorithm to obtain the output data. By performing repeated trials of this process (Monte-Carlo simulation), for any given value of N, we can count the total number of bit errors over all simulated blocks, and obtain a bit-error rate (BER).

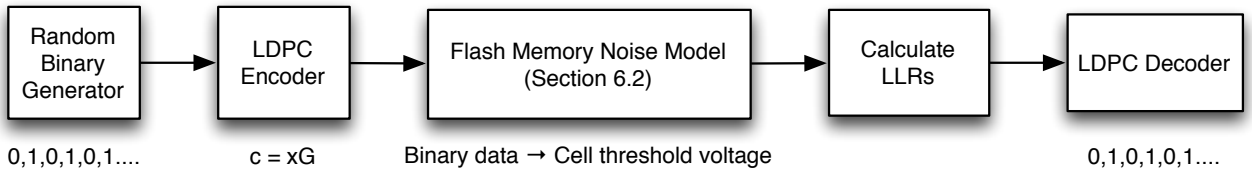

Figure 17: Memory simulation model

This simulation model is shown in figure 17. The 'Memory Noise Model' block encapsulates the entire process described in section 6.2, which includes mapping the input data to a cell threshold voltage, and then applying noise to that voltage, remembering that binary 1's and 0's are handled differently.

In this particular model, there are 2 independent variables that can be chosen: 'N', the number of program/erase cycles that the cell has undergone, and 't', the time in seconds that the data has been stored in that cell (the retention time). In addition, there are a number of initial conditions, which don't vary during the simulation [Appendix B, table 3]. For the remainder of this project, 't' was fixed to 5 years, which simulates data having been stored in the cell for that time. In initial testing, the variation of 't' had a substantially smaller effect on the overall bit-error rate of the system than varying 'N', so for simplicity 't' was fixed. This is shown in figure 18, where the uncoded (variable boundary hard decision method) bit-error rate of the memory cell is shown, and using a fixed 't' of 5 years has close results to using a 't' that varies as 'N' increases $(t_{yrs} = N/5000)$. Using a fixed value for 't' gives us a 'worst case scenario' of data undergoing a long retention time, even if this is unlikely in an actual scenario.

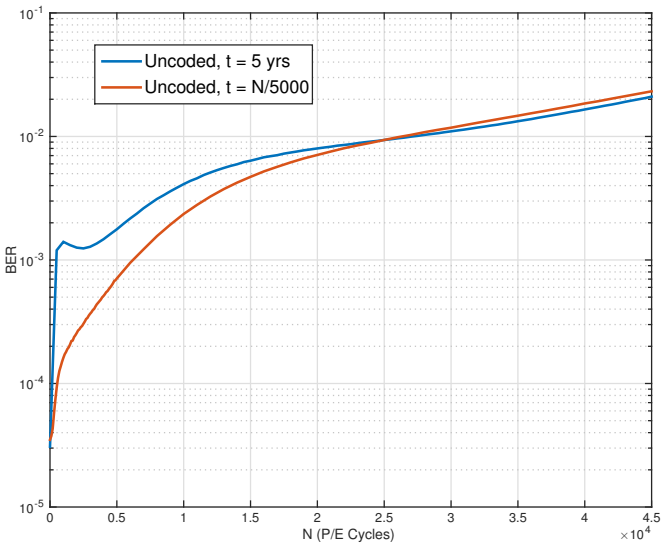

Figure 18: Comparing the effect of fixed t against using a variable t dependent on N

8.2 Simulation Results

Monte-Carlo simulations were run on a remote cluster computer for each of the different decoding schemes explained in section 7. The results are shown in both figures 19 and 20.

Figure 19 shows the bit-error rate against a given 'N', the number of cell program/erase cycles. The solid lines all represent soft-decision decoding, each using a different approach when forming the LLR that is fed to the decoder. The 'Static Gaussian', whilst not explicitly declared earlier, uses a Gaussian centred about V_{p0}, the initial programmed voltage. It is static since both the mean and variance of this Gaussian don't vary as N increases. This was the first attempt at performing soft-decision decoding for the system, and is essentially similar to the method of obtaining the LLR's for the AWGN channel (2 static Gaussian distributions).

The two matched Gaussians, one including the Random Telegraph Noise and one without, are the next best performing curves. It is clear that by adding the variance of the RTN to the Gaussian, it substantially improves performance of the decoder. However, the two functions obtained through the convolution of the various PDF's have the best overall performance. The green line, which is effectively making use of the exact underlying function in order to generate the LLR's, is therefore the upper bound on the performance of this system.

Finally, the dotted line shows the performance of hard decision decoding. It is substantially worse performing than any of the soft decision methods, and only improves error performance against the uncoded case (the blue dashed line) by a relatively small margin, and only up to $25,000$ cycles.

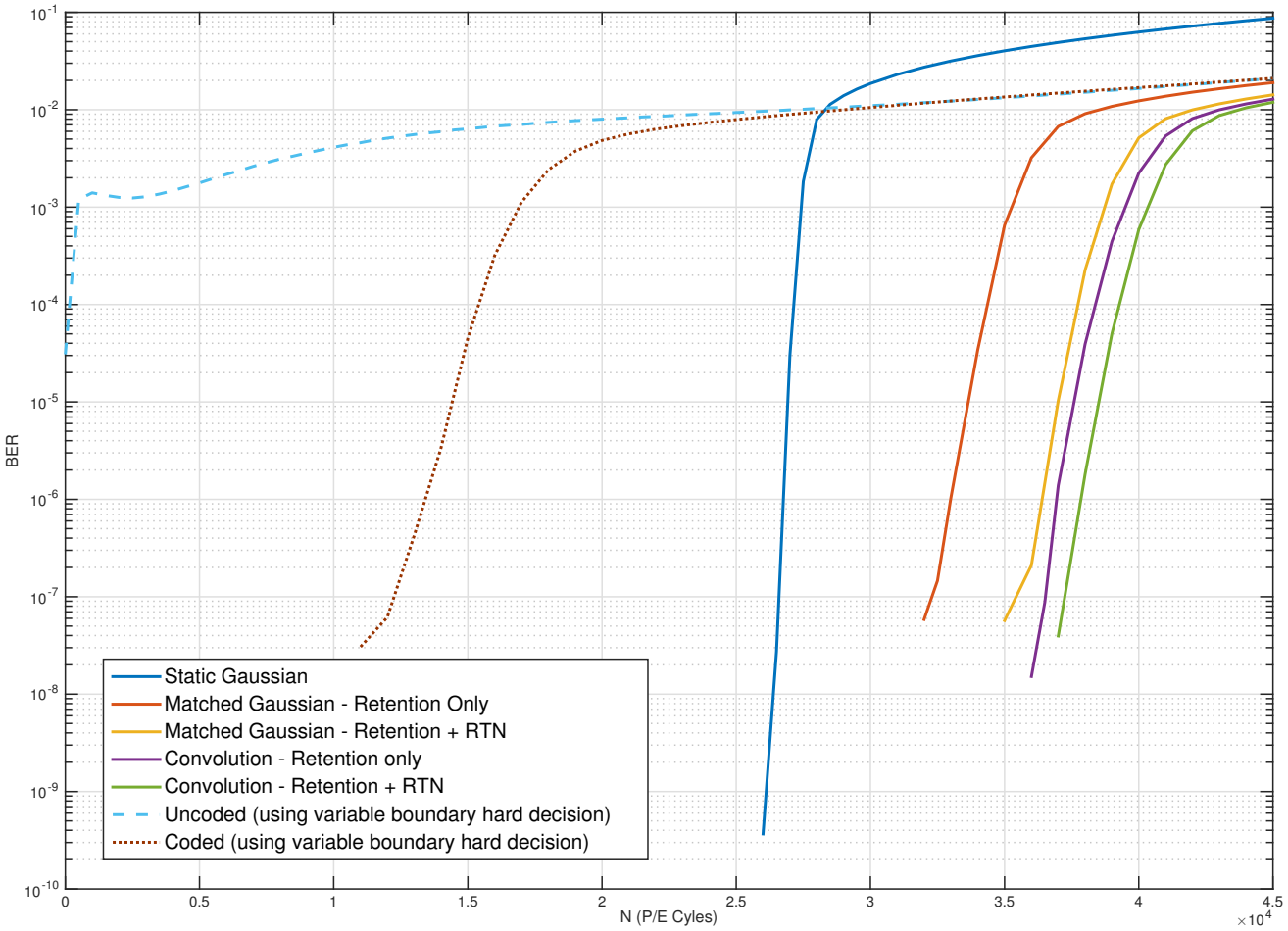

Figure 19: Graph of bit-error rate vs N for different decoding and LLR schemes

Figure 20 displays the same set of results for the soft decision schemes only, but in a different format. The Coded vs Uncoded graph is a standard method used in industry to compare relative performance of error correction schemes. Essentially, given an uncoded system's bit error rate, it is possible to determine what the BER would be if an error correcting scheme is used. The uncoded BER in this case is taken from the hard decision, variable boundary method (the blue dashed line in the previous graph).

This particular graph construction is useful in comparing different systems, since it does not depend on the exact type of noise used. Instead, it allows us to easily compare what the error rate of the memory cell would be given a specific input error rate and coding scheme. For example, if the target bit error rate at the output of the system is to be 10^{-6}, then an error rate of around 1% in the memory cell is acceptable if using the Static Gaussian model. However, using the full PDF of the underlying noise in the decoder, an error rate of around 1.5% is acceptable. Whilst this seems only a very small gain, by looking at figure 19 again, it actually results in the cell being able to endure an additional $10,000$ cycles.

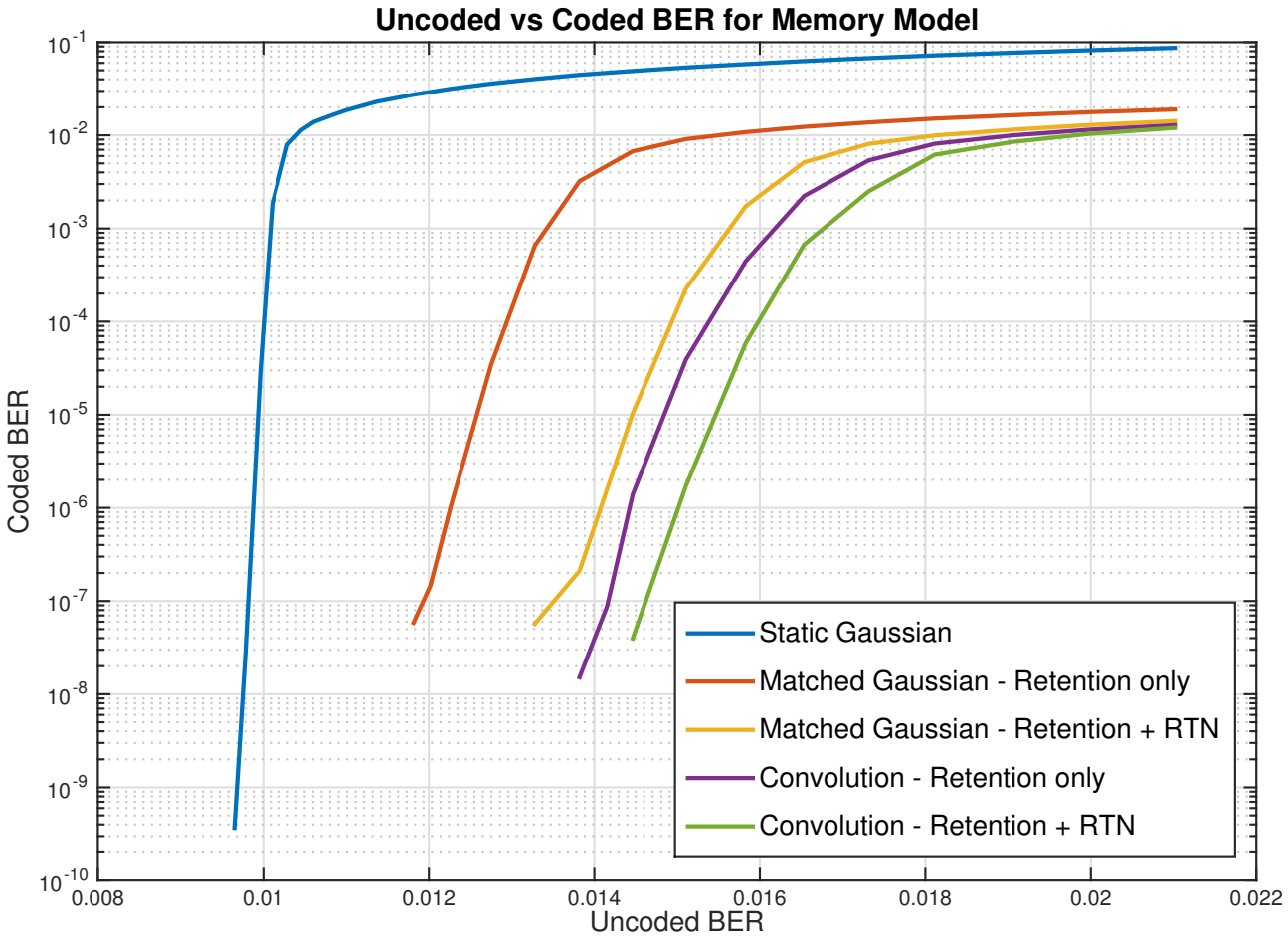

Figure 20: Graph of uncoded vs coded bit-error rate for soft-decision cases

9 Conclusions

It is clear that error-correcting code plays an important part in flash memory. Without it, modern multi-level cell devices would likely not be possible, or would have much tighter constraints on device lifetime. However, as said previously, current flash memory uses hard decision decoding. The difference between the hard-decision schemes currently used and the potential soft-decision schemes demonstrated in this project, as well as investigated in many other papers, are substantial. This is perhaps why there is a great deal of interest in this area by the memory manufacturers.

LDPC codes are widely known to achieve capacity approaching performance on the AWGN channel. However, it is clear that such a simplistic channel model is not appropriate when modelling a flash memory cell. This project has demonstrated the implementation of a soft-decision LDPC scheme in tandem with a more complex noise model. Admittedly, it is very hard to verify how accurate this model is compared to a real device, mainly since the data generated in industry is kept highly secretive. However, the importance of accurately modelling the underlying noise process is shown to be especially critical when dealing with soft-decision decoding. Even the most minor variations in the probability density functions can result in fairly large differences. So, whilst it wasn't possible to verify that the noise model is accurate compared to a real device, the project has shown the importance of matching the decoder to the assumed noise process.

Soft-decision decoding with the AWGN channel assumes that the received symbols generate Gaussian distributions, centred about the transmitted mean. However, this project has shown that the 'Static Gaussian' approach to decoding is not suitable when dealing with flash memory: the mean of the cell threshold voltage varies with the program/erase cycle number. The logical extension was to continue using a Gaussian for the decoding, but matching the mean and variance with the underlying noise process. The 'Matched Gaussians' had substantially improved performance over the static case, whilst being computationally inexpensive to implement [Appendix B, table 2].

In order to achieve the best soft-decision decoding performance, the exact probability density function of the cell threshold voltage for all values of 'N' is required. This was demonstrated by taking the two additive noise sources in the memory model, and producing the required PDF through the convolution operation. Whilst this process results in the best performance, it does so at a cost to both the complexity of the decoder as well as the work required to obtain the PDF in the first place. Since the actual real-world noise process is assumed to be unknown, or at least unverified, using this method seems

redundant unless the model is shown to be very accurate compared to an actual device.

An important, but so far completely ignored reality, is that full unlimited precision soft-information is impossible to obtain from a memory cell. The sense-amp comparator used in memory cells today generates at most 3-bits of information, in the case of triple-level cell memory. All of the simulations in this project have assumed double floating point precision, which is effectively 64 bits of information. Work is ongoing in investigating the effect of quantisation in soft-decision decoding of LDPC codes [7], as well as more specifically in the number of comparator reads required to achieve near-optimum levels of soft information in a flash memory cell [2]. Both have shown that even with limited precision soft information, the decoder retains the majority of its performance. For example, [7] shows that around 6 bits of soft information is required in the AWGN case to achieve near identical performance to the unlimited precision case. For the memory model, figure [??] demonstrates the effect of simple 2-bit and 3-bit quantisation when using the exact convolution PDF. It shows reasonable performance considering the reduction in precision. An area of future research would likely be investigating exactly how to create the optimum quantisation regions.

As well as looking at the effect of quantisation, future work in this area would almost certainly involve looking at code construction. This project made use of a specific LDPC code designed for digital video broadcast - not at all related to flash memory. It is known that “the optimal code for a given set of channel conditions may not resemble the optimal code for another” [24, p. 369], that is, codes perform differently dependent on application. What works for the AWGN channel may not work for any future memory channel, and the codes that perform well with unlimited soft information may not be the ones that perform well in the limited precision case.

This project only looked at modelling SLC memory (1 bit per cell), whereas in reality modern cells either store 2 or 3 bits of information. As the cell threshold voltages get closer and closer to each other, the importance of error correction will only increase in the future. One solution presented here is to make use of soft information in the cell, which results in substantial improvement over the hard-decision scheme. It was also shown that accurate modelling is vitally important to the error correcting performance. Future developments in flash memory will likely include QLC (Quad-level cell) [25] technology for solid-state drives. Modern error correction schemes, such as the ones presented here, will almost certainly be one of many areas of research in the wider semiconductor industry, with the goal of increasing storage capacity, whilst reducing device costs.

Appendix A: MATLAB code

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 [26]. 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. This 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.

Extract 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));
```

The commands in extract 1 were used to seed the random number generator every time a new MATLAB process was created [27]. 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" [28]. 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.

Sum-Product decoders

Extract 2 and extract 3 show the code generated in MATLAB to construct the sum-product decoders. The former is the first iteration of the program, that was un-vectorised, whilst the latter is a fully vectorised version.

Extract 2: Sum-Product decoder, first version

```
function [y,iterations] = BP_iterate(x,H,l)
iterations = l;
% x = Input LLR vector
% H = Graph connection matrix/Parity Check Matrix
% l = # Iterations
% y = Output LLR vector
%i = Message Nodes;
%j = Check Nodes;

% Need to calculate number of CHK and MSG nodes from H
[j_max,i_max] = size(H);
nonzeros = nnz(H);
%Preallocate M_IJ and M_JI as sparse matrices
m_IJ = spalloc(j_max,i_max,nonzeros);
m_JI = spalloc(j_max,i_max,nonzeros);

for iter = 0:l
    %All Message nodes:
    for i = 1:i_max
        % At each Message node:
        h = H(:,i);% Column vector of connections to check nodes
        if iter == 0 % on initial iteration:
            m_IJ(find(h),i) = x(i); % Message sent = initial conditions
        else % subsequently:
            w = m_JI(:,i);
            % This single line of code below was debugged by Hachem Yassine
            m_IJ(:,i) = h*x(i) + h*sum(w) - w; %SUM step
        end
    end

    m_IJ_2 = m_IJ'; %Transpose: Memory access is then much quicker!
```

```

    %All Check nodes:
    for j = 1:j_max
        % At each Check node:
        w = m_IJ_2(:,j);
        [row,~,v] = find(w); % v is non-zero elements. row is the data
        m_JI(j,row) = 2*atanh(prod(tanh(v./2))./(tanh(v./2))); % PROD step
    end

    %Clipping function: prevents overflow
    m_JI((m_JI) > 1000)=999;
    m_JI((m_JI) < -1000)=-999;

    %Get current variable node values
    sumVector = sum(m_JI);
    y = x + sumVector;

    %Check: Is a valid codeword? i.e. y*H' == 0
    %Hard Decision:
    for i = 1:length(y)
        if y(i) > 0
            y2(i) = 0;
        else
            y2(i) = 1;
        end
    end
    %Parity Check:
    test = sum(mod(y2*H',2));
    if test == 0
        iterations = iter;
        return
    end
end
end
```

Extract 3: Sum-Product decoder, second version (Vectorised)

```
function [y,iterations] = BP_iterate(x,H,l)
iterations = l;

% Need to calculate number of CHK and MSG nodes from H
[j_max,i_max] = size(H);
nonzeros = nnz(H);
%Preallocate M_IJ and M_JI as sparse matrices
m_IJ = spalloc(j_max,i_max,nonzeros);
m_JI = spalloc(j_max,i_max,nonzeros);
H_ = H';

for iter = 0:l
    %All Message nodes:
    if iter == 0 % on initial iteration:
        m_IJ = bsxfun(@times,x,H);
    else % subsequently:
```

```
        a = (x + sum(m_JI)); % SUM step
        b = bsxfun(@times,a,H);
        m_IJ = b - m_JI;
    end

    m_IJ_2 = m_IJ'./2; %Transpose: Memory access is then much quicker!

    %All check nodes:
    %Uses logarithm for element-wise division of 2 matrices
    %i.e. subtraction of logs is division
    c = tanh(m_IJ_2);
    d = spfun(@log,c);
    e = sum(d);
    f = bsxfun(@times,e,H_);
    g = f - d;
    g = spfun(@exp,g);
    m_JI = 2*atanh(g);
    m_JI = m_JI';

    % Clipping function
    m_JI((m_JI) > 1000)=999;
    m_JI((m_JI) < -1000)=-999;

    %Get current variable node values
    sumVector = sum(m_JI);
    y = x + sumVector;

    %Stop check: Is a valid codeword? i.e. y*H' == 0
    %Hard Decision:
    y2(y>0) = 0;
    y2(y<0) = 1;

    %Parity Check:
    test = sum(mod(y2*H_,2));
    if test == 0
        iterations = iter;
        return
    end
end
end
```

Simulation run script

Extract 4 shows the top level script when running locally. When submitting to a compute cluster, the code was changed slightly (e.g. to include the Random Number Generation setup). The outer 'for' loop selects the P/E cycle number, 'N', whilst the inner 'Parfor' loop performs the Monte Carlo simulation of generating many blocks, from which the error ratio is obtained.

Extract 4: Top-level Monte-Carlo simulation run script for the Memory channel

```
% LDPC Run Script for memory model
% FIXED t, VARIABLE N
% System Parameters
SystemParams.tYrs = 5;
SystemParams.Verased = 1.4;
SystemParams.Vp = 2.8;
SystemParams.deltaVp = 0.25;
SystemParams.tSecs = SystemParams.tYrs*365*24*3600;

%%% DVB-S2 CODES %%%
%Code Rate
Rc = 9/10;
%Code size
Nc = 64800;
%DVB-S2 Parity check matrix
H = dvbs2ldpc(Rc);

% MonteCarlo Simulation Runs
mc_iters = 100000; %Number of blocks
l = 50; %SumProduct max iterations

% Loop to go over all values of N, as well as perform MC Simulation
I = [];
for N = 0:1000:45000
    hEnc = comm.LDPCEncoder(H);
    hDec = comm.LDPCDecoder('ParityCheckMatrix',H,'
       IterationTerminationCondition',...
        'Parity check satisfied','OutputValue','Whole codeword');
    hError = comm.ErrorRate;
    SystemParams.N = N;

    %Parallel 'for' loop: Run simultaneously on multi-cores
    parfor i = 1:mc_iters
        errRatio(i) = ldpc_BER_memoryN_coded(Rc,Nc,hEnc,hDec,...
        hError,SystemParams,H,l);
    end

    %Output Matrix
    I = [I;N,mean(errRatio)];
end
```

Appendix B: Tables

Table 1: Speed benchmark comparison of various Sum-Product decoders (Timed over 100x64k blocks)

Decoder Type	Time to run (s)	Relative Speed
Matlab built-in (on Toshiba Cluster) 160x CPU	0.25	2430x
OpenCL, AMD GPU	2.02	301x
OpenCL, 8x CPU	6.32	96x
Matlab built-in, 8x CPU	7.63	80x
OpenCL, iGPU (Intel)	8.14	75x
Matlab built-in, 1x CPU	28.4	21x
Personal decoder v2, 8x CPU	123.4	5x
Personal decoder v1, 8x CPU	608.52	1x

Table 2: Speed benchmark comparison of the different schemes used to obtain the LLR's for the memory model decoding

Function Used	Time to run (s)	Relative Speed
Moment-Matched Gaussian	0.495	5.5x
Partial function (Convolution of Uniform & Gaussian)	1.841	1.5x
Full function (Convolution of Uniform, Laplacian & Gaussian [23])	2.732	1x

Table 3: Value of variables used in the memory simulations

Symbol	Name	Value
V_{p0}	Initial programmed voltage	2.8 V
ΔV_{pp}	ISPP step voltage	0.25 V
V_{e0}	Mean initial erased voltage	1.4 V
σ_e^2	Variance of erased state	0.35 V
t_{Yrs}	Retention time	5 Years
K_s	Retention noise parameter	0.38
K_d	Retention noise parameter	4×10^{-4}
K_m	Retention noise parameter	4×10^{-6}
t_0	Retention noise parameter	3600 s
K_{rtn}	RTN constant ($\lambda_{rtn} = K_{rtn}\sqrt{N}$)	0.00025

References

[1] Guiqiang Dong et al. “Estimating information-theoretical NAND flash memory storage capacity and its implication to memory system design space exploration”. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 20.9 (2012), pp. 1705–1714.

[2] Jiadong Wang et al. “LDPC Decoding with Limited-Precision Soft Information in Flash Memories”. In: *CoRR* abs/1210.0149 (2012). URL: http://arxiv.org/abs/1210.0149.

[3] D. J. C. MacKay. “Good Error–Correcting Codes based on Very Sparse Matrices”. In: *Proceedings of 1997 IEEE International Symposium on Information Theory. Ulm, Germany.* 1997, p. 113.

[4] Robert G Gallager. “Low-density parity-check codes”. In: *Information Theory, IRE Transactions on* 8.1 (1962), pp. 21–28.

[5] Matthew C Davey and David JC MacKay. “Low density parity check codes over GF (q)”. In: *Information Theory Workshop, 1998*. IEEE. 1998, pp. 70–71.

[6] Jinghu Chen and Marc PC Fossorier. “Near optimum universal belief propagation based decoding of low-density parity check codes”. In: *Communications, IEEE Transactions on* 50.3 (2002), pp. 406–414.

[7] Jianguang Zhao, Farhad Zarkeshvari, and Amir H Banihashemi. “On implementation of min-sum algorithm and its modifications for decoding low-density parity-check (LDPC) codes”. In: *Communications, IEEE Transactions on* 53.4 (2005), pp. 549–554.

[8] Jinghu Chen et al. “Reduced-complexity decoding of LDPC codes”. In: *Communications, IEEE Transactions on* 53.8 (2005), pp. 1288–1299.

[9] Mohammad Rakibul Islam et al. “Optimized Min-Sum Decoding Algorithm for Low Density Parity Check Codes”. In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 2.12 (2011).

[10] Roberto Bez et al. “Introduction to flash memory”. In: *Proceedings of the IEEE* 91.4 (2003), pp. 489–502.

[11] David Tse and Pramod Viswanath. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[12] J.G. Proakis. *Digital Communications*. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill, 2001.

[13] Simon Haykin. *Communication systems*. John Wiley & Sons, 2008.

[14] Justin Coon. *C16: Advanced Communications (Lecture Notes)*. Department of Engineering Science, University of Oxford, 2015.

[15] Tom Richardson. "Error floors of LDPC codes". In: *Proceedings of the annual Allerton conference on communication control and computing*. Vol. 41. 3. 2003, pp. 1426–1435.

[16] Bernhard MJ Leiner. *LDPC Codes–a brief Tutorial*. 2005. URL: http://www.bernh.net/media/download/papers/ldpc.pdf.

[17] Tuan Ta. "A Tutorial on Low Density Parity-Check Codes". In: *The University of Texas at Austrin* (2009).

[18] Junho Cho, Chongjin Xie, and Peter J Winzer. "Analysis of soft-decision FEC on non-AWGN channels". In: *Optics express* 20.7 (2012), pp. 7915–7928.

[19] Sarah J Johnson. *Iterative error correction: Turbo, low-density parity-check and repeat-accumulate codes*. Cambridge University Press, 2009.

[20] James E Gilley. "Bit-error-rate simulation using Matlab". In: *Transcrypt International, Inc* (2003).

[21] Mathworks. *MATLAB Documentation: LDPC Decoder*. URL: http://www.mathworks.com/help/comm/ref/comm.ldpcdecoder-class.html (visited on 04/25/2015).

[22] "Levyfan". *OpenCL LDPC Decoder*. URL: https://github.com/levyfan/opencl-LDPC-decoder (visited on 04/25/2015).

[23] Hachem Yassine. *Convolution of Uniform, Gaussian and Laplacian distributions*. Communications Research Group, Department of Engineering Science, University of Oxford.

[24] S.W. Golomb, R.E. Peile, and R.A. Scholtz. *Basic Concepts in Information Theory and Coding: The Adventures of Secret Agent 00111*. Applications of Communications Theory. Springer, 1994.

[25] Young Choi. *Under the Hood: NAND flash preps for 4 bits per cell*. URL: http://www.eetimes.com/document.asp?doc_id=1167325 (visited on 05/02/2015).

[26] Mathworks. *MATLAB Documentation: Random number generator algorithms*. URL: http://www.mathworks.com/help/matlab/ref/randstream.list.html (visited on 04/27/2015).

[27] Mike Croucher. *Parallel Random Numbers in MATLAB*. URL: http://www.walkingrandomly.com/?p=2755 (visited on 04/27/2015).

[28] Ubuntu Manpage Repository. *Manual page on: dev/random*. URL: http://manpages.ubuntu.com/manpages/lucid/man4/random.4.html (visited on 04/27/2015).

Some LaTeX code, used to format this report, obtained from: http://en.wikibooks.org/wiki/LaTeX (Licensed under CC BY-SA 3.0).

Generic Display Screen Equipment Risk Assessment – "Making Flash Memory Work" - 4YP MATLAB based project		
Assessment undertaken by: Henry Fletcher	**Signed**:	**Date:**
Assessment supervisor: Dr. Justin Coon	**Signed:**	**Date:**

Hazard	**Persons at Risk**	**Risk Controls In Place**	**Further Action Necessary To Control Risk**
Eyestrain/ Headaches	User	**Take regular breaks every hour**. - undertake a different task. - adjust screen location to prevent glare or bright reflections. - Angle screen downwards to prevent reflection. - ensure no screen flicker. - ensure screen surface is clean. - ensure lighting is adequate for the task. - have an eye test if problems persist. - close blinds to prevent glare (as appropriate)	Consult Supervisor and advise Departmental Safety Officer (DSO) if problems persist. Please refer to the following link for a picture of good posture: http://www.hse.gov.uk/pubns/indg36.pdf
Back pain	User	**Ensure Workplace is correctly set up** - E.g. height of chair needs to be set so that forearms are parallel to desk. - ensure good posture at all times, sitting upright or slightly reclining. - Lower back supported to maintain natural curves.	Refer any medical issues to Supervisor or Departmental Safety Officer (DSO)
Aching shoulders, wrists	User	**Check seat height is correct** - forearms horizontal, level with top of desk. - keep wrists straight, use wrist rest. - No overreaching, exercise muscles. - Arms relaxed by side.	Refer any medical issues to Supervisor or Departmental Safety Officer (DSO)
Aching neck	User	**Check screen height is correct** - Eyes level with top of screen. - use document holder. - exercise muscles. - Check chair height e.g. forearms horizontal, level with top of desk	

Aching legs	User	**Check space under desk** to stretch legs, feet rest comfortably on floor otherwise get footrest. - exercise muscles. - Knees level with pelvis or slightly below. - Feet flat on the floor or use a footrest.	Remove items under desk which are preventing correct use e.g. boxes.
Water/Liquids	User	Please ensure that no liquids are sat on your hard drive or near to your monitor.	Building Inspections.
240 VAC Electrical shock	User	User to check that all electrical leads to their PC are in good working order. Contact Electronics (Thom 5th floor) if Portable Appliance Label 'out of date' or not visible.	Supervisor/Student to check validity of PAT test label.