BLIND CONSTRUCTION OF A PARITY CHECK MATRIX FOR ARBITRARY LINEAR BLOCK CODES

by

Aaron J. Womack

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Todd Moon, Ph.D. Major Professor	Jacob Gunther, Ph.D. Committee Member
Jared Jensen, M.S. Committee Member	D. Richard Cutler, Ph.D. Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2025

ABSTRACT

Blind Construction of a Parity Check Matrix for Arbitrary Linear Block Codes

by

Aaron J. Womack, Master of Science

Utah State University, 2025

Major Professor: Todd Moon, Ph.D.
Department: Electrical and Computer Engineering

Since Claude Shannon introduced his channel capacity theorem, mathematicians and engineers have discovered many techniques to accomplish Forward Error Correction (FEC). For decades, algorithms have been proposed with parameters that can be configured to the specific needs of the communication system. Because of this, there are countless variations of FEC that use different approaches to accomplish a similar goal. In a typical communication system, the transmitter and receiver are both aware of the FEC type and parameters, so they agree on the encoding and decoding process. However, if the encoder parameters are unknown to the receiver, decoding the FEC codewords is challenging due to the large variety of FEC code types and parameters.

Many FEC codes used today fall under the category of linear block codes. Linear block codes can be represented with a generator matrix for the encoder and a parity check matrix for the decoder. The parity check matrix often provides insight into the structure of the code, which could indicate the type of FEC that was applied and how to decode it. Therefore, a logical first step in blindly decoding a stream of encoded bits is to determine a parity check matrix.

This thesis offers a method to blindly construct a parity check matrix for a linear block code given a stream of received soft-decision bits. The method does not require codelength

or dimension to be known a priori. The method uses rank deficiency to search for a nullspace of a group of potential codewords.

(57 pages)

PUBLIC ABSTRACT

Blind Construction of a Parity Check Matrix for Arbitrary Linear Block Codes

Aaron J. Womack

Communication systems are systems in which a transmitter sends data across a channel to a receiver. In any modern communication system, Forward Error Correction (FEC) is used to detect and correct errors that occur during transmission. As communication systems have developed over the years, many FEC algorithms have been proposed. Each FEC algorithm has parameters that can be selected based on the needs of the communication system. Typically, the transmitter and receiver are both aware of the FEC parameters, but in some cases the receiver may not know how the transmitter encoded its message. In this case, the receiver must blindly determine the FEC parameters.

This thesis offers a method to blindly recognize an important parameter of the FEC code, called a parity check matrix. The parity check matrix helps the receiver to decode the messages sent by the transmitter. The method described in this paper takes a noisy transmission and determines a parity check matrix without any prior knowledge of the FEC code. While the method outlined in this thesis does not work for every type of FEC code, it works for a category of FEC commonly used in modern communication systems called linear block codes.

ACKNOWLEDGMENTS

I want to thank my wife, Kaitlyn, for her patience and support through my education. I could not have done this without her. I want to thank my parents and grandparents for teaching me the value of hard work, and my children for providing me the motivation to put it into practice.

I am grateful for my professors, especially Dr. Scott Budge, Dr. Jake Gunther, and Dr. Todd Moon. Through the lessons I learned in your classrooms, I have grown to appreciate signal processing and communications. You shaped the path of my education and career.

I would like to thank all those who risked hiring me for internships, research, and full-time work even though I was far from qualified. I would not be the engineer I am now without those opportunities.

Finally, I thank my God who gives me strength to do all things.

Aaron J. Womack

CONTENTS

Page

ACRONYMS

AWGN	Additive White Gaussian Noise
BCH	Bose, Chaudhuri, Hocquenghem
BPSK	Binary Phase Shift Keying
BSC	Binary Symmetric Channel
CRC	Cyclic Redundancy Check
FEC	Forward Error Correction
GF	Galois Field
LDPC	Low-Density Parity Check
LLR	Log-Likelihood Ratio
PC	Parity Check
RREF	Reduced Row-Echelon Form
RS	Reed-Solomon
SNR	Signal to Noise Ratio

CHAPTER 1

INTRODUCTION

Forward Error Correction (FEC) is the process of encoding redundancy into a message so that errors can be detected and corrected after errors are introduced into the communication channel. FEC is used in some form in almost any communication system. In typical communication systems, the transmitter and receiver are designed with mutual agreement on the FEC code and parameters. However, in non-cooperative environments, the receiver has no prior knowledge of the FEC. This thesis focuses on the application of FEC diagnosis in a non-cooperative environment.

Although FEC codes are difficult to detect and decode blindly, the purpose of FEC is generally not to obfuscate the message. If a communication system requires secrecy, a separate encryption process is typically used. FEC codes must add structure to the code in order to detect and correct errors. Blind diagnosis exploits the structure encoded into the message through FEC.

This chapter gives an introduction to some important principles of FEC. Section 1.1 gives some background on FEC concepts that will be used in the thesis. Section 1.2 explains noisy environments and how soft-decisions are used to decode. Section 1.3 describes the purpose and use of interleavers in communication systems, and Section 1.4 provides the scope of the research.

1.1 Forward Error Correction Background

This section provides background on error correction theory that will be important throughout the paper. The two main categories of FEC are linear block codes and convolutional codes. Section 1.1.1 gives background on linear block codes. Section 1.1.2 defines Hamming distance and Hamming weight, two concepts that describe how effective a linear block code is. Convolutional codes are briefly described in Section 1.1.3. Section 1.1.4

explains equivalent codes, and Section 1.1.5 describes how error correction can be used for larger Galois Fields (GF).

1.1.1 Linear Block Codes

Linear block FEC codes are memoryless systems which process a block of message bits to produce a codeword. Linear block codes are typically represented with a dimension k, corresponding to the number of bits in a message. The length of the codeword, also known as the codelength, is represented by n. The encoder can be represented with a generator matrix $G_{k\times n}$, and the decoder can similarly be represented by a Parity Check (PC) matrix $H_{(n-k)\times n}$. To encode a message vector m, the vector is applied to the generator matrix as follows:

$$c = m * G \tag{1.1}$$

The codeword c is a vector of n bits. The PC matrix is designed such that each column in H is orthogonal to any codeword c. In other words, the PC matrix satisfies the following equation for any codeword c:

$$0 = c * H^T \tag{1.2}$$

The generator matrix is often designed with a structure to allow for simple decoding. Some examples of matrix structure include systematic, cyclic, and low-density matrices. Some common examples of linear block codes include Hamming codes, Bose-Chaudhuri-Hocquenghem (BCH) codes, Reed-Solomon (RS) codes, and Low-Density Parity Check (LDPC) codes. Although the decoding process looks different for each code, all codes can be represented by their generator or PC matrices.

1.1.2 Hamming Weight and Hamming Distances

Hamming distances and Hamming weights are metrics used to describe how effective a FEC code is. The Hamming distance is a metric used to determine how similar two

codewords are. The Hamming distance refers to the number of coefficients in two codewords that are different from each other. The number of nonzero coefficients in a single codeword is called the Hamming weight. The Hamming weight can be thought of as the Hamming distance between any codeword and the all-zero codeword. Because each row in a generator matrix is a valid codeword, each row has a Hamming weight.

The minimum Hamming distance of a code is the smallest distance between any two codewords in the set of codewords. If a FEC code has a large minimum Hamming distance, the code is able to correct and detect more errors. One way to find the minimum Hamming distance is to find the smallest Hamming weight of any codeword in the set other than the all-zero codeword.

1.1.3 Convolutional Codes

Convolutional codes differ fundamentally from linear block codes. Linear block codes are memoryless from one message to the next. In contrast, the convolutional encoder holds memory of previous inputs. The output of the encoder is a function of both the current input and the current state of the memory. Many assumptions used to blindly diagnose or reconstruct linear block codes are based on linearity, and therefore the techniques do not apply to convolutional codes.

1.1.4 Equivalent Codes

An error correction code is defined by its mapping from messages to codewords. It is possible for two different encoders to map the same set of messages to the same set of codewords, even if the mapping between individual messages to codewords is not the same. In this case, codes are equivalent. A generator matrix can be transformed into an equivalent matrix by applying elementary row operations. For the purpose of this research, finding an equivalent PC matrix for the decoder satisfies the search.

1.1.5 Larger Fields

The most common example of larger fields in FEC is the RS code, although many other codes can be adapted to use larger fields as well. For a code operating over GF(2), each message and codeword is a vector of bits. For codes over a larger field, the messages and codewords are vectors of symbols whose coefficients live in the larger field. The generator matrix and PC matrix for a linear block code over a larger field are also represented by symbols, where each element in the matrix is a coefficient from the larger field.

There is no restriction on the Galois Fields that a FEC code can operate over, but in practice codes always lie on GF(2^m) for an integer m. This is because the symbols must still represent a set of bits in order to communicate information efficiently, and bits are best represented in powers of 2.

1.2 Noise and Soft-Decisions

The purpose of this section is to explain different representations of noise in a communication system. Section 1.2.1 describes the concept of hard-decision and soft-decision decoding. Section 1.2.2 explains how Log-Likelihood Ratios (LLR) are used in soft-decision decoding.

1.2.1 Hard-Decision vs. Soft-Decision Decoding

There are two general approaches to correct errors in a noisy environment: hard-decision decoding and soft-decision decoding. In hard-decision decoding, the values of the bits are determined before attempting to decode and correct errors. In this approach, a bit error occurs if the wrong decision is made about a bit. The hard-decision decoder attempts to locate that bit and flip it to the correct value.

A common noise model for hard-decision decoding is the Binary Symmetric Channel (BSC). The BSC gives each bit has an equal probability of being received incorrectly. The BSC takes clean bits as input and flips the value of some bits to generate the output.

Soft-decision decoding uses received values from the output of a communication system's matched filter to decode. No decisions are made about the bit values, so more

information is available for the decoder to use for correcting errors. For example, in the case of Binary Phase Shift Keying (BPSK) modulation, if the soft-decision value is close to zero then it is likely to have an error. On the other hand, if the received bit has a large magnitude, then it is more likely to be the correct value.

The most common noise model for soft-decisions is the Additive White Gaussian Noise (AWGN) channel. The AWGN channel assumes that the noise has consistent power across all frequencies with a zero-mean Gaussian distribution in the time-domain. However, some channels are better modeled with bursts of noise. For a bursty channel, the power of the noise jumps at random moments in time. While errors are typically distributed uniformly across the message in an AWGN channel, the errors in a bursty channel are grouped together in clusters.

Soft-decision decoders are more effective than hard-decision decoders because they use more information, but soft-decision decoders also require more computations and memory. Hard-decision decoders can store many bits very efficiently and use binary math for sums and products of bits. Soft-decision decoders must store a floating point or integer value for every received symbol, and the math for a soft-decision decoder is often more complex.

1.2.2 Log-Likelihood Ratios

Many soft-decision decoders use LLR to correct errors. LLR are a common representation used to show the likeliness that a bit is either a 1 or a 0. The equation for LLR of a bit b is often given as

$$L_b = \log \frac{P(b=0)}{P(b=1)} \tag{1.3}$$

where L_b is the LLR value for the bit b [1]. Approximate LLR are sometimes calculated instead to improve computation time.

LLR calculation for BPSK modulation is simple because each bit has a unique modulated symbol. When multiple bits are sent in a single symbol, the LLR calculation requires extracting the bit probabilities for each bit in the symbol. This probability can be complicated for some modulations, but the result is that all modulations can be converted into

a common soft-decision representation. Therefore, soft-decision decoders that use LLR as inputs are typically generic to any type of modulation.

Soft-decision decoders can perform better than their hard-decision counterparts, but the LLR do not have the simple binary calculations that many hard-decision decoders use. For example, the addition of two bits over GF(2) can be done with an XOR operation. The equivalent operation is

$$L_1 + L_2 = 2\tanh^{-1}(\tanh(\frac{L_1}{2})\tanh(\frac{L_2}{2})) \tag{1.4}$$

for LLR [1]. In order to improve computation time for LLR-based decoders, approximations and lookup tables can be used in place of the computations. One example of an optimization is using the function $\phi(x)$ which is defined as

$$\phi(x) = 2\tanh^{-1}(\log^{-1}(-x)) \tag{1.5}$$

Equation 1.4 can be replaced by the function $\phi(x)$ with the following operation:

$$L_1 + L_2 = \text{sign}(L_1) * \text{sign}(L_2) * \phi(\phi(|L_1|) + \phi(|L_2|)) \tag{1.6}$$

A lookup table is often used for $\phi(x)$ to avoid calculating it each time. Although this is still not as fast as a binary XOR operation, the function $\phi(x)$ and other approximations make LLR reasonable to work with.

1.3 Interleavers

This section explains the use of interleavers in communication systems. Section 1.3.1 describes the purpose of interleavers and a few common types. Section 1.3.2 describes how interleavers can be represented in matrix form and combined with a generator matrix.

1.3.1 Interleaver Overview

Many forms of error correction work better on errors spread out across the message than they do on bursts of errors. Interleavers are commonly used in communication systems to spread out bursts of errors so that the decoder is more effective. Interleavers take multiple codewords at a time and permute the bits before the message is transmitted. A known permutation is used so that a deinterleaver can undo the operation at the receiver. If a burst of errors occurrs on the transmitted message, the deinterleaver will shuffle the errors around to make the errors appear more uniform.

A simple example of interleaving is the matrix interleaver. A matrix interleaver is defined by the number of rows and number of columns the matrix contains. The bits on the input of the interleaver are written into the matrix row-by-row. The bits are then read out of the matrix column-by-column. The corresponding matrix deinterleaver writes the received values into a matrix column-by-column and reads the bits out row-by-row. Some other examples of interleavers include convolutional interleavers and helical interleavers. Although interleavers with particular qualities are chosen for efficiency, any permutation of a set of bits can be used as an interleaver.

1.3.2 Matrix Representation of Interleavers

An interleaver is simply a permutation of bits, so any block interleaver can be described as a permutation matrix. For example, the simple 2×3 matrix interleaver can be represented in matrix form as

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.7}$$

This permutation matrix can be combined with an error correction code's generator matrix to form a permuted generator matrix. Suppose the 2×3 matrix interleaver above

is fed by two codewords from a (3, 1) repetition code. The (3, 1) repetition code can be represented with the generator matrix

$$G = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \tag{1.8}$$

Since two repetition codewords are inputs to the matrix interleaver, a generator matrix that forms both codewords can be represented as

$$G_2 = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{1.9}$$

If the interleaver follows immediately after the repetition encoder, the generator matrix and permutation matrix can be multiplied together to form a modified generator matrix, G_p, where

$$G_p = G_2 * P = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{1.10}$$

In this case, the input to G_p would be two bits, b_1 and b_2, and the output is alternating between them three times in a row: $b_1, b_2, b_1, b_2, b_1, b_2$. While this code and interleaver may not be useful in practice, it serves as a trivial example. Using the same logic, any linear block code followed by a block interleaver can be represented with a single, permuted generator matrix.

1.4 Scope

This thesis offers a new algorithm to determine an equivalent generator matrix and PC matrix for a stream of encoded bits from linear block codes. The algorithm should not rely on any prior knowledge about the encoder. This thesis assumes that soft-decisions are available from the receiver. This thesis also assumes that if encryption or a scrambling sequence is applied in the communication system of interest, it has been applied over the message bits and not the encoded bits. The focus of this project is to discover information about the FEC, although information about the interleaver can be learned if any type of block interleaver has been applied.

CHAPTER 2

BACKGROUND

Blind recognition and diagnosis of error correction codes is not a new problem. This chapter will discuss the previous research conducted in this field. Section 2.1 discusses current research related to FEC diagnosis. Section 2.2 describes the process of finding the nullspace of a matrix over GF(2), an important step in Chapter 3. Section 2.3 looks at gaps in current research that the thesis seeks to fill.

2.1 Literature Review

The research on FEC diagnosis can be divided into two categories. The first category is research focused on efficient algorithms to blindly diagnose a specific code type. The second category focuses on general algorithms to diagnose a large set of codes. This thesis falls in the second category as it seeks a general algorithm of FEC diagnosis.

The methods outlined in research on individual codes provide more complete solutions but fail to generalize to other codes. Some examples of this research include blindly diagnosing convolutional [2], cyclic [3], LDPC [4], BCH [5], and RS [6] [7] codes. Research is more prevalent in the context of individual codes, likely because unique algorithms can exist for every type of error correction. Papers on individual codes tend to exploit unique mathematical properties of the code.

The research focused on general FEC diagnosis is often less efficient and less complete than the research on individual codes. The results of this research tend to provide solutions for some parameters but not enough information to decode the signal. Many papers focus on a single code parameter.

A few papers focus on blindly identifying the codelength and synchronization of any linear block code. For example, one paper had success identifying codelength and synchronization of binary codes with length 256 or less and with LDPC codes [8]. Another paper

was able to successfully find codelengths for non-binary codes [9]. A third paper was able to discover dimension as well as codelength and synchronization [10], but the results given were only for codelengths of 6, 7, and 8. The author did not state a maximum codelength the algorithm could discover, but it is not clear from the results if the algorithm works for larger codes.

One paper focused on blindly recovering a PC matrix of a linear block code if the codelength and dimension were already known [11]. This approach assumed that an equivalent systematic form of the PC matrix existed. A systematic code is any code that contains the original message somewhere inside the codeword. While most linear block codes can be manipulated into a systematic code, this is not a guarantee. In addition, this paper focuses on binary codes only, and no explanation is given on how to handle non-binary codes.

One research paper explores the use of recurrent neural networks to identify which type of code is used [12]. Identifying the type of code is important because the PC matrix alone does not equate to a functioning decoder. It is essential to use properties of the FEC code to efficiently correct errors and decode messages. In the research paper on code type identification, the only input to the neural network is a stream of noisy bits. The research used a small set of codes and code parameters, so it is not clear how well this concept would generalize to all codes.

2.2 Finding Nullspace of a Matrix over GF(2)

This section describes the mathematics used to find a nullspace of a matrix over GF(2). Section 2.2.1 gives some background on why Gaussian Elimination works over GF(2). Section 2.2.2 explains how the nullspace of a Reduced Row-Echelon Form matrix is determined in two cases: a matrix with a systematic form and a matrix without one. Examples are provided as well as references to previous research in Section 2.2.2.

2.2.1 Gaussian Elimination over GF(2)

There are many methods to find the nullspace of a matrix over the real numbers. Some of these methods are the Singular Value, QR, and LU decompositions. Each method has

attributes that make it more suitable in some cases than others. Gaussian Elimination is one method of finding the nullspace of larger matrices, but it is not used often because it can cause numerical instability. Numerical instability is not a concern over GF(2) since there are only two possible coefficients: 0 and 1. The decompositions over real matrices do not apply over GF(2), but Gaussian Elimination works because numerical instability cannot occur.

Gaussian Elimination is the process of using elementary row operations to reduce a matrix to its Reduced Row-Echelon Form (RREF). The coefficients of the RREF matrix can be formed into a set of linear equations that are all equal to zero. For a $k \times n$ matrix over GF(2) with $k < n$, there can be k linearly independent columns at most. According to the Rank-Nullity Theorem, a $k \times n$ matrix with k linearly independent columns will have $n - k$ vectors that span the nullspace of the matrix.

Gaussian Elimination is an order $O(n^3)$ computation, where n is the number of columns in the matrix. Over GF(2), Gaussian Elimination only uses XOR operations and swapping of rows, so it can be more efficient than Gaussian Elimination over the real numbers.

2.2.2 Determining Nullspace over GF(2) Given an RREF Matrix

When a $k \times n$ matrix has been reduced to RREF, each non-zero coefficient can be classified as a pivot variable or a free variable. The pivot variables are the leading non-zero coefficient of each row, and all other non-zero coefficients are free variables. If a generator matrix in RREF has an identity matrix in the first k columns, this matrix is systematic. Otherwise, it is non-systematic. This section will provide examples of how the nullspace of a systematic or a non-systematic RREF generator matrix can be calculated.

Suppose there is a matrix A and the RREF has been computed and labeled A_r, given by

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}, A_r = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{2.1}$$

Note that the matrix A_r is in systematic form and can be written as $A_r = [I|P]$ where I is the 4×4 identity matrix and P is an arbitrary 3×4 matrix. The system of equations corresponding to A_r is formed by multiplying the coefficients of each column c with x_c, summing across the row, and setting the row equal to 0. For A_r, the system of equations is

$$\begin{aligned} x_1 + x_5 + x_6 &= 0 \\ x_2 + x_6 + x_7 &= 0 \\ x_3 + x_5 + x_6 + x_7 &= 0 \\ x_4 + x_5 + x_7 &= 0 \end{aligned} \tag{2.2}$$

Using the system of equations, solve for each of the pivot variables: x_1, x_2, x_3, and x_4. Then form a column vector of all values of x_c, but substitute the pivot variables for the equivalent sum of free variables. For the matrix A_r, the column vector would be

$$\begin{bmatrix} x_5 + x_6 \\ x_6 + x_7 \\ x_5 + x_6 + x_7 \\ x_5 + x_7 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \tag{2.3}$$

Now separate the column vector into individual column vectors for each free variable, listing the coefficient instead of the variable. The span of these column vectors is the nullspace, and a nullspace basis matrix can be formed by combining the vectors into a single matrix. For the example matrix in Equation 2.1, the resulting nullspace basis matrix

is

$$B = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

Note that the matrix B is of the form $B = \begin{bmatrix} P \\ I \end{bmatrix}$ using the same 3×4 matrix P found in A_r. The nullspace for a systematic matrix can be determined simply by rearranging the location of P and the dimension of I in the RREF matrix A_r. This comes from solving the system of equations in Equation 2.2, and the same steps work for matrices that are not systematic.

If the RREF generator matrix is not in systematic form, obtaining the nullspace result is not quite as convenient as rearranging an I and P. However, following the same steps can still provide a simple method to determine the nullspace. Suppose there is a non-systematic RREF matrix named A'_r, defined as

$$A'_r = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{2.5}$$

The system of equations for this matrix A'_r is

$$\begin{aligned} x_1 + x_4 + x_7 &= 0 \\ x_2 + x_4 + x_6 &= 0 \\ x_3 + x_4 + x_6 + x_7 &= 0 \\ x_5 + x_6 + x_7 &= 0 \end{aligned} \tag{2.6}$$

For Equation 2.2, the pivot variables were x_1, x_2, x_3, and x_4. In Equation 2.6, x_4 is a free variable while x_5 is a pivot instead. After solving for the pivot variables, forming a column vector, and converting this into a matrix, the resulting nullspace for A'_r is

$$B' = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

Analyzing the structure of the matrix B', there are still some similarities with A'_r. Rows 4, 6, and 7 in B', corresponding to the free variables x_4, x_6, and x_7, form an identity matrix. Rows 1, 2, 3, and 5, corresponding to the pivot variables, can be combined to form a matrix P. This P is the same matrix formed by taking the free variable columns in A'_r. To visualize this, A'_r and B' are shown again below, with the columns and rows of P highlighted in yellow. The rest of the matrices are highlighted in blue to represent the identity matrix.

$$A'_r = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}, B' = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

In conclusion, the nullspace can be calculated for any GF(2) matrix with the following steps:

1. Convert to RREF matrix
2. Form matrix P from the free variable columns of the RREF matrix
3. Assign rows of P to pivot variable rows of the nullspace basis matrix
4. Assign free variable rows of the nullspace basis matrix to the rows of an identity matrix

Researchers have looked at alternative methods to calculate the nullspace of a matrix over GF(2). One paper proposed an algorithm designed to limit storage usage compared to Gaussian elimination [13], and another built on this algorithm by making it parallel [14]. For the calculations in this thesis, a matrix will already be RREF, so calculating the nullspace requires only rearranging some rows and columns.

2.3 Gaps in Current Literature

Although there is extensive research surrounding the blind identification of linear block codes, the author is not aware of an algorithm that determines the PC matrix for a linear block code without prior knowledge such as codelength and dimension. In addition, current research on blind recovery of a PC matrix has not been shown to generalize for codes that do not have an equivalent systematic representation.

One gap in the current literature is an algorithm that takes only a stream of encoded bits from a linear block code and produces a PC matrix, without any prior knowledge of the code. The research in this document seeks to fill that gap in current literature.

CHAPTER 3

METHODS

The end goal of blind FEC identification is to obtain decoded message bits. For a linear block code, the decoder can be represented through a PC matrix. From the PC matrix, it may be easier to construct a decoder to obtain message bits. This chapter defines the method that will be used to identify the generator matrix and PC matrix for any linear block code using only a noisy stream of LLR. The algorithm is given in the context of binary codes in Section 3.1. Section 3.2 explains how the concept can be extended to apply for codes over a larger field as well. Section 3.3 shows how an interleaved code is handled given a generator or PC matrix.

3.1 Description of Blind PC Matrix Algorithm

This section describes the algorithm used to blindly identify a PC matrix for a code. Algorithm 3.1 provides an overview of the methods. Algorithm 3.1 relies heavily on Gaussian Elimination to find the RREF of a matrix of LLR. The important steps of the algorithm will be described in detail in Sections 3.1.1, 3.1.2, and 3.1.3.

The input of Algorithm 3.1 is a vector of LLR values for a set of noisy bits that have been encoded with a linear block code. The demodulation of the signal should occur prior, although deinterleaving is not required beforehand. The first LLR should correspond to the first bit of a codeword, although Chapter 4 will discuss the scenario when the start of the LLR is not aligned with the start of a codeword.

The bulk of the algorithm is seeking a generator matrix for the code, and a PC matrix is derived from the generator matrix at the end. The PC matrix, H, and the codelength that produced H are outputs of the algorithm.

Algorithm 3.1 Blind Parity Check Identification

Input:
LLR of noisy bits
Output:
Parity Check Matrix, H
Codelength, N
Begin
Let $max\_codelength$ be square root of number of LLR
For n=3:1:$max\_codelength$
Form codewords from LLR
Sort codewords based on minimum LLR magnitude
Let A be RREF of matrix formed by best $n + \lceil \sqrt{n} \rceil$ codewords
If rows(A) $< n - 1$
Let G be the GF(2) conversion of A
For rows of G with Hamming weight 1 or 2
Look for and correct error if possible
End
Let H be nullspace of corrected G matrix
Let N be current codelength n
Exit
End
End
End

3.1.1 Selecting Potential Codewords and Sorting

Since the codelength is not known for the stream of bits, a loop passes over possible codelengths. At each codelength, divide the stream of LLR into a set of potential codewords which will be used throughout the rest of the steps.

From the available codewords, fewer than $2 * n$ codewords are needed. The LLR indicate how likely it is an error occurred on a bit. By extension, the LLR can indicate which codewords are most likely to contain errors. To get the best performance possible, the best quality codewords should be used.

If a LLR value has a large magnitude, then it has high confidence. Therefore, the corresponding bit is more likely to be correct. A codeword contains an error if any bit is incorrect, so the best codeword has high magnitudes for all LLR. The codewords are sorted based on the minimum LLR magnitude.

The average magnitude of a codeword's LLR could be used as a metric instead, but the average does not capture the likeliness of an error occurring on a codeword. If one LLR is very large and another in the same codeword is close to zero, the average would still look decent. However, one of the bits in the codeword is very likely to have an error. The minimum LLR magnitude is a better metric because it will ensure all bits have a higher confidence of being correct.

To select the best possible codewords, label all available codewords by the lowest LLR magnitude contained in that codeword. Sort the codewords from highest minimum to lowest minimum.

3.1.2 Using Gaussian Elimination to Find Rank Deficiency

Gaussian Elimination uses more computations for a matrix over LLR than a matrix over GF(2). The general steps are the same, but the XOR operation for two GF(2) values is replaced with an atan operation for LLR values. If the atan operation is substituted with the function $\phi(x)$ using a lookup table as described in Section 1.2.2, the Gaussian Elimination for a matrix of LLR is reasonable to compute.

After codewords have been formed and sorted at the current potential codelength, form a matrix A from the best available codewords. The number of codewords used to form A should ideally eliminate false positives while avoiding unnecessary errors. If n codewords are used to form A, any potential codeword that randomly lands on the linear combination of other codewords will cause the rank of A to be less than n. This results in a false positive. If A is formed out of $2 * n$ codewords to avoid false positives, there are twice as many codewords that may contain errors. The number $n + \lceil \sqrt{n} \rceil$ was found to be a good balance between eliminating false positives and reducing the amount of errors in the matrix A. Therefore, the matrix A is formed from the best $n + \lceil \sqrt{n} \rceil$ codewords.

The goal is to perform Gaussian Elimination on A to check the rank of the matrix. If the rank is n, it means that there is no linear structure in the codewords for the current codelength. This either means the data is too noisy to recognize the structure, or the wrong codelength has been selected. In this case, move on to the next potential codelength and return to the codeword selection step.

If the rank is less than n, the LLR are likely to have structure at the same boundary as the selected codelength. This points to the bits being encoded by a linear block code with the codelength n. The matrix A in RREF is considered a generator matrix if the rank is less than n.

3.1.3 Removing Initial Errors

After a codelength has produced a generator matrix, errors should be reduced to confirm that this potential codelength and generator matrix are correct. This section will discuss techniques to correct errors and get closer to the correct generator matrix. Each error in a codeword that goes into the Gaussian Elimination can result in an additional row in the generator matrix. Adding rows to the generator matrix increases the dimension of the code, which can make the structure of a code difficult to interpret. Therefore, a generator matrix is not as useful if it contains errors.

One simple technique to correct errors is to scan a row for the lowest magnitude LLR value, reverse the sign, and try Gaussian Elimination again. If the resulting RREF matrix

is smaller than the original one, this bit contained an error that has now been corrected. If the resulting RREF matrix is the same size, it is not known whether an error existed or not. The LLR value is restored to its original sign to avoid introducing additional errors. This approach is a form of educated guessing and checking, where the LLR are used to guess where errors occurred.

While this first approach can work, it is computationally expensive for a low yield of corrected errors. It is also impossible to correct cases of two or more errors on a single codeword, since only one bit is flipped at a time.

A better technique to correct errors is to use information provided by the RREF generator matrix. If a bit has been flipped in a codeword, it will result in a linearly independent row in the generator matrix. This row will have a coefficient with the value 1 in the column where the error occurred. All other coefficients will be zero. Since the row has Hamming weight of 1 and FEC codes should have a higher weight, this row can be flagged as an error. If two bit errors occurred on the same codeword, the linearly independent row would have two coefficients equal to 1 and the rest would be zero.

After discovering that a bit error has occurred and finding the position of the error, the codeword containing the error must be located. Sort the codewords based on the magnitude of the LLR in the column where the error occurred. Starting with the smallest magnitude LLR, reverse the sign of the LLR and perform Gaussian Elimination again. If Gaussian Elimination returns an RREF matrix of a smaller dimension, the error has been corrected. If the matrix is the same size, return the LLR to its original sign and try the next codeword. As long as only one codeword had a bit error on that particular column, the error will be corrected.

If all single-bit errors have been corrected such that no more rows have Hamming weight 1, the same concept can be applied to the rows with weight 2. For each row with Hamming weight of 2, suppose the two indices with a coefficient equal to 1 are both errors. Sort the codewords based on the average magnitude of the LLR in the two indices, and flip the sign of the LLR in both locations for each codeword. If the Gaussian Elimination with

the modified codeword produces a smaller dimension matrix, then an error has likely been corrected. Otherwise, revert the LLR for the modified codeword back to the original signs.

While this concept could theoretically extend to rows of even higher Hamming weights, this should be done with caution. Since a code can have a minimum distance of 3 and still correct errors, it is possible to remove valid codewords with Hamming weight 3 from the code. By treating a valid codeword as an error and converting it to a different valid codeword, the generator matrix would become smaller and not represent the real code.

After the errors have been removed, the PC matrix can be found from the generator matrix. To do this, find the nullspace of G as described in Section 2.2, and take the transpose of the result.

3.2 Extending to Larger Fields

The methods in this chapter work on elements from the field GF(2). Some popular FEC such as RS codes use larger fields for each symbol, so adapting to larger fields is necessary. In practice, any FEC code using a larger field uses a power of 2.

A PC matrix over a larger field can be broken down into an equivalent PC matrix over GF(2) as long as the larger field is GF(2^m) for some integer m. If the GF(2^m) PC matrix has dimensions $(n-k) \times n$, the GF(2) PC matrix will have dimensions $(n-k)*m \times n*m$. By treating the matrix over a larger field as a larger matrix over GF(2), the same methods should offer results for FEC codes on a larger field. A post-processing step would restore the code to the larger field.

3.3 Extending to Interleaved Codes

The methods in this chapter work for an arbitrary generator matrix. As explained in Section 1.3.2, a generator matrix G that is interleaved can be represented by $G_p = G_2 * P$, where G_2 represents a generator matrix for multiple codewords and P is a permutation matrix representing the interleaver. If a code has gone through an interleaver, the methods given in the previous sections of this chapter can find G_p but not G. However, some post-processing techniques can get closer to the matrix G. Section 3.3.1 describes the method used to discover interleaver patterns, and Section 3.3.2 provides an example of the method.

3.3.1 Method for Discovering Interleaver Patterns

Let a set of rows be linked with each other if they both have a 1 in the same column. If one row is linked to a second row through a column, and the second row is linked to a third row through a different column, all three rows will be considered linked. The concept of linked rows will be used to discover interleaver patterns.

Suppose the blind discovery techniques from this chapter are used to find G_p without any errors. Create a set of linked rows, called S_1, and add the first row in G_p to the set. Find every row in G_p that is linked to the first row, and add these rows to S_1. If at least one more row has been added to S_1, find every row in G_p that is linked to any row in the set S_1. Repeat this step recursively until S_1 does not grow. The set S_1 contains all rows in G_p that could be linked to the first row.

Let G'_p be the matrix formed when the rows in S_1 are removed from G_p. If G'_p is empty, this means that G_p is not the result of interleaving multiple codewords together. Otherwise, G_p has multiple interleaved codewords and S_1 contains the rows associated with one of the codewords.

Suppose G_p is the result of multiple interleaved codewords, and the rows in S_1 have been removed to form G'_p. Repeat the process used to find S_1 on the matrix G'_p, but name the set S_2. Continue to repeat this process until a set S_l contains all remaining rows in G_p. At this point, G_p has been divided into l sets corresponding to l codewords that were interleaved together.

Let c_1 be the codeword associated with the set of rows in S_1. Form a matrix out of the rows in S_1. For each column in the S_1 matrix that contains a 1, the column index reveals the location to which the bits of c_1 were interleaved. Using these column indices, the bits can be gathered to form c_1', which is a permutation of the original codeword c_1. This same process can be done for all other sets to form permutations of the other codewords.

For some interleavers, all permuted codewords c' may be equal to the original codewords c. However, this is not guaranteed for all interleavers. The process of linking rows together is not guaranteed to return the original codewords, but it does group the bits of each codeword together to provide some information about the interleaver.

3.3.2 Interleaved Matrix Example

To visualize this concept of linked rows, an example matrix G_p is shown in Equation 3.1. The matrix G_p was generated by interleaving 3 Hamming(7,4) codewords together using a 3×7 matrix interleaver and taking the RREF form of the matrix. The first row in G_p and all rows linked to it have been highlighted. The columns used to link the rows have been highlighted as well.

$$G_p = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} \tag{3.1}$$

The highlighted rows are added to the set S_1. The rows in the set S_1 are removed from G_p to form G'_p. After highlighting the rows and columns used to find S_2, the resulting matrix G'_p is

$$G'_p = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} \tag{3.2}$$

By removing the rows in G'_p that lie in S_2, the final set S_3 is formed. Each set is shown

below. The columns with a 1 coefficient in each set are highlighted. The highlighted columns represent the locations to which the bits of the corresponding codeword were interleaved.

$$
S_1 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}
$$

$$
S_2 = \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}
\tag{3.3}
$$

$$
S_3 = \begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

In this example, all three sets become identical when the all-zero columns are removed. Combining the highlighted columns into a matrix creates the RREF of the original G used for the encoder. For other interleavers, the highlighted columns may need to be permuted to get the RREF of the original G. In those cases, the author is not aware of any method to determine what the permutation of the columns is. The result of this method is that the bits can be gathered into groups for each codeword, but the order of the bits may not be correct.

CHAPTER 4

RESULTS

The following linear block codes have been tested and proven to work with this algorithm: Hamming codes, BCH codes, RS codes, LDPC codes, and Cyclic Redundancy Checks (CRC). Linear block codes that have been passed into an interleaver have also been shown to work. Not every code will have an example in this thesis, but the same steps work to produce a PC matrix.

This chapter looks at some results of Algorithm 3.1. Section 4.1 provides examples of the algorithm when applied to a Hamming code. Section 4.2 shows the performance of the algorithm based on parameters such as codelength, rate, and Signal to Noise Ratio (SNR). Section 4.3 gives the order of computations for the algorithm.

4.1 Example on a Hamming Code

This section provides an example of Algorithm 3.1 when it is applied to a Hamming(7,4) code. Let the generator matrix G for a cyclic (7, 4) Hamming code be

$$G = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{4.1}$$

An array of 1000 bits is generated at random, divided into 4-bit messages, and encoded with the generator matrix G. The resulting codewords are formed into a one-dimensional array of bits. Gaussian noise is applied to the bits, with noise applied to create a 3dB SNR. After noise has been applied, the LLR are calculated from the noisy bits assuming a BPSK modulation. The stream of noisy bits is passed into the algorithm with no other information.

For each potential codelength, codewords are formed and sorted based on the smallest magnitude LLR values of each codeword. The best codewords are arranged into a matrix A that has dimensions $n+\lceil\sqrt{n}\rceil \times n$. The RREF of the matrix is found using LLR operations, and only the rows with at least one negative element are returned. For the LLR calculation used, a negative value is likely to be a 1 in GF(2), so this is the equivalent of keeping only rows with at least one nonzero element in them. The RREF matrix is an $n \times n$ identity matrix for the codelengths 3, 4, 5 and 6 because there is no structure that falls along those potential codewords. When the potential codelength is set to 7, a $k \times n$ matrix is returned where k is less than n. In RREF form, the generator matrix produced is

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{4.2}$$

In this example, a generator matrix with an error was chosen to show how errors are corrected. The fourth row has a Hamming weight of 1, with all zero coefficients except for the fourth column. Because this is likely an error, loop over the codewords in the matrix A, and try flipping the sign of one LLR in the fourth column. After flipping the sign of the LLR, compute the RREF, take the rows that have at least one negative value, and check the new dimensions of the matrix. If the dimension is smaller than it was before flipping the LLR, the error was corrected. The generator matrix G' after correcting the bit error is

$$G' = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{4.3}$$

After correcting the error, no other rows have a small Hamming weight. All rows in G' are linked, so the codewords have not passed through an interleaver. The PC matrix H is

formed and given by

$$H = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

This is a valid PC matrix for the generator matrix used to generate the Hamming codewords. The algorithm was successful at finding a codelength and PC matrix for the bits.

4.1.1 Handling Unsynchronized Bit Streams

In blind decoding, there is no guarantee that the receiver knows when a packet of information starts. Therefore, it is possible that the algorithm could start forming codewords that are misaligned with the real codewords. If Algorithm 3.1 incorrectly assumes the first bit is the start of a codeword, then the bit stream is considered unsynchronized. This can create problems for the algorithm.

Suppose the bit stream is unsynchronized by one bit for a Hamming(7,4) code. The algorithm forms codewords that contain six bits from one codeword and another bit from a different codeword. The six bits from the same codeword are related, so there will be rank deficiency among those six bits when Gaussian Elimination is performed on the misaligned codewords. The bit that belongs to a different codeword has no relationship with the six bits from the adjacent codeword, so one linearly independent row will be added to the generator matrix. The linearly independent row will have Hamming weight of 1 and look like an error, but it cannot be corrected by flipping bits.

If multiple bits are misaligned, the algorithm will not recognize rank deficiency as easily. It becomes increasingly difficult for any generator matrix to be produced, meaning the

algorithm will likely fail. The goal of the algorithm was to be fully flexible and generic with no inputs other than the bits. However, to obtain the correct PC matrix, synchronization is required. No other assumptions were found that are necessary for the algorithm to work.

4.2 Performance on Varying Parameters

This section will discuss how a few parameters affect the performance of the algorithm. Each section compares the performance at various SNR when the parameter of focus varies. Section 4.2.1 discusses the effects of codelength, Section 4.2.2 discusses the effects of rate, and Section 4.2.3 discusses the number of bits used.

For each test in this section, noise was applied to create SNR from 15dB down to 0dB. The noise was applied according to BPSK modulation. At each SNR, the code was tested 20 times. Each iteration of the test checked two things: if the PC matrix is the same as the noiseless case, and if the codelength was the correct value. A counter for each check increased if it was true, and then the counter was divided by 20 at the end to get a percentage that each check passed. The bits were passed into the algorithm with the assumption that the first bit was the start of a codeword, but no other information or assumptions were provided to the algorithm.

4.2.1 Codelength vs. SNR

To see the effects of codelength on the performance of the algorithm, a test varied the codelength and SNR while all other parameters were fixed. The test used the following BCH codes: BCH(31, 21), BCH(63, 39), BCH(127, 81), BCH(255, 171), and BCH(511, 340). The dimension for each BCH code was selected to provide a rate near $\frac{2}{3}$ so the rate could be fixed as well.

The number of bits for each codelength was selected to be $4 * n^2$. By making the number of bits relative to n, the number of bits should not be a contributing factor in the algorithm's performance as the codelength changes. Figure 4.1 shows the results for both the correct H and the correct codelength tests.

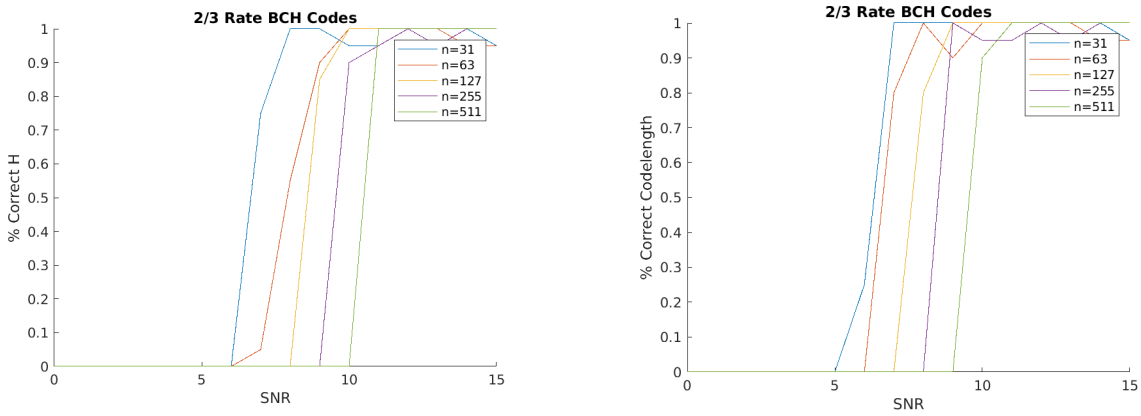

Fig. 4.1: Plots showing the how effective Algorithm 3.1 is at various codelengths for a $\frac{2}{3}$ BCH code. The percentage represents how often the algorithm identified (a) the correct PC matrix and (b) the correct codelength.

The results of this graph are expected. As the codelength increases, the codewords have more bits where an error could occur. The codewords are more likely to have errors, so it is harder to locate a rank deficient matrix. When the codelength is fixed, the ability to decode becomes easier as the SNR increases.

Based on the plots in Figure 4.1, there seems to be a direct relationship between the codelength and the minimum SNR required to find a PC matrix. To find a PC matrix that contains some errors, the SNR should be about log2(n)dB for a codelength n. To find a PC matrix without errors, the SNR should be 1dB higher, or $1 + \log2(n)$dB.

A BCH(903, 1023) was tested a few times as well. This BCH code was omitted from the full Codelength vs. SNR test because it was not reasonable to run the algorithm for hundreds of iterations with the available test equipment. Algorithm 3.1 was tested three times on the BCH(903, 1023) code, twice with an SNR of 10dB and once with 11dB. The 10dB SNR tests both failed to find any PC matrix. The 11dB SNR test produced the correct codelength, but the PC matrix had some errors. Further testing would need to be performed to know if 11dB is an accurate threshold for the algorithm to work on a code that large.

4.2.2 Rate vs. SNR

To test the algorithm's performance at various rates, a fixed codelength of 255 was selected for multiple BCH codes. The code's dimension was changed to vary the rate. The dimensions selected are 231, 207, 179, 155, and 131. With each change in dimension, the rate decreases by about $\frac{1}{10}$. The number of bits used to test the codes was 255,000. Figure 4.2 shows the results of the Rate vs. SNR test.

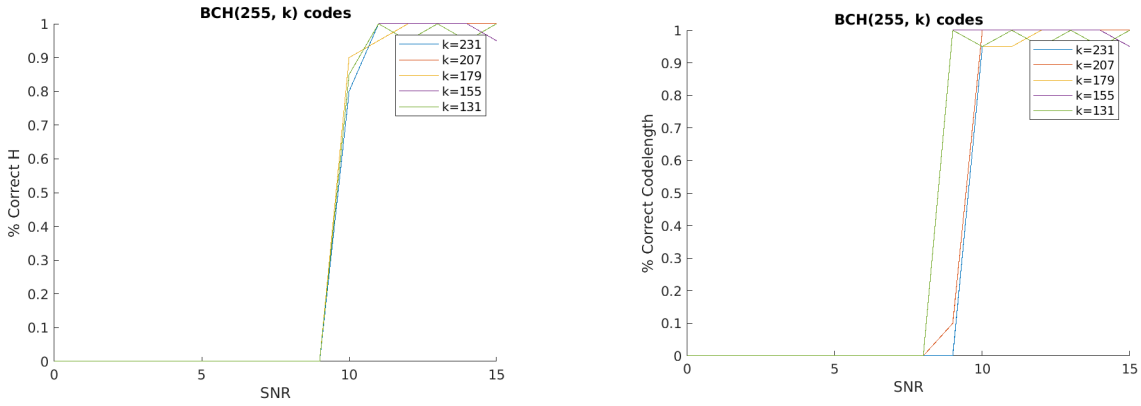

Fig. 4.2: Plots showing the how effective Algorithm 3.1 is at various rates for a BCH(255, k) code. The percentage represents how often the algorithm identified (a) the correct PC matrix and (b) the correct codelength.

Using steps of 1dB in SNR, the rate did not have a large impact on the ability of the algorithm. The algorithm found the correct codelength a little less often than average for the BCH code with dimension 231, but all other results were similar at each rate. Theoretically, the rate should have an impact on the performance of the algorithm. When the code is high rate, a few errors can hide the rank deficiency in the codewords, causing the algorithm to fail. In practice, the plot shows that rate does not have an obvious impact on the performance.

It is possible finer steps in SNR would show the exact impact that rate has on the algorithm. Also, further testing may be able to uncover a threshold at which the rate becomes important.

4.2.3 Number of Bits vs. SNR

To test the relationship between the number of bits and the SNR, a fixed code was used for all iterations. The only variable that changed was the number of bits passed into the algorithm. A BCH(63, 39) was used to allow for many iterations to be tested with a number of bits that is large compared to the codelength. Figure 4.3 shows the results of the test on the number of bits.

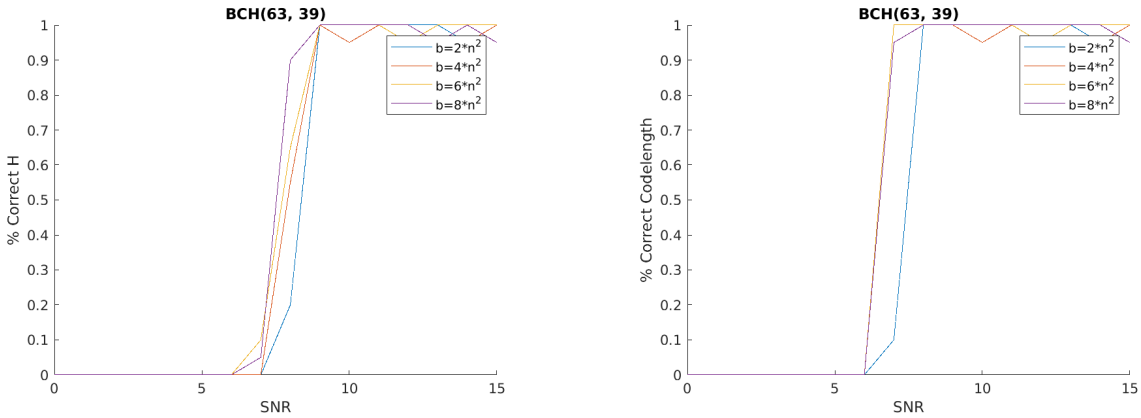

Fig. 4.3: Plots showing the how effective Algorithm 3.1 when different number of bits are available for a BCH(63, 39) code. The percentage represents how often the algorithm identified (a) the correct PC matrix and (b) the correct codelength.

The plots do not show an exact trend, but the plot containing $2{*}n^2$ bits underperformed compared to the rest of the plots. There is about a 1dB difference in the performance of the algorithm when increasing from $2 * n^2$ to a larger number of bits. After that initial increase, the performance does not appear to have a significant change when the number of bits varies.

4.3 Order of Computations

This algorithm relies heavily on Gaussian Elimination, which is a calculation of order $O(m^3)$ for an $m \times m$ matrix. It is assumed that the codelength will be at least 3 since no useful code can be smaller. If no prior information is available about the codelength, then Gaussian Elimination will be performed on all possible codelengths from 3 up to n.

This means the order of computations for Algorithm 3.1 is approximately $\sum_{k=3}^{n} k^3$. Since $\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$, the order of computations for the algorithm can be approximated as $O(n^4)$.

After locating the generator matrix, finding the PC matrix is only a matter of rearranging rows and columns. To determine if the code has been interleaved or not, this takes a small recursive algorithm that is negligible compared to the $O(n^4)$ computations to find the generator matrix. However, removing errors from the generator matrix requires a significant number of computations. Using the technique of small Hamming weight rows described in Section 3.1.3, each potential error would take up to n Gaussian Eliminations that are each $O(n^3)$ operations. This is an additional $O(n^4)$ operations to correct each potential error. Therefore, a more accurate order of operations can be given by $O(n^4 * (N_e + 1))$, where N_e is the number of errors on the generator matrix.

CHAPTER 5

DISCUSSION

This chapter evaluates the effectiveness of Algorithm 3.1. Section 5.1 discusses the strengths and weaknesses of the algorithm. Section 5.2 looks at how this algorithm fits into the big picture of blind FEC diagnosis.

5.1 Evaluation of the Algorithm

This section will evaluate scenarios when Algorithm 3.1 performs particularly well or struggles to provide an answer. Section 5.1.1 will focus on the strengths and Section 5.1.2 will focus on the weaknesses of the algorithm.

5.1.1 Strengths

In the results of Chapter 4 and through other testing, Algorithm 3.1 has been proven to work over a range of linear block codes. All linear block codes that have been tested so far could return a valid PC matrix when a higher SNR is used. The codes that were tested include Hamming, BCH, RS, and LDPC codes as well as CRCs. Codes that have been passed through an interleaver were tested as well. Although not all linear block codes have been tested, the algorithm is expected to work as long as a generator matrix exists for the code. The only requirement is that the data has been synchronized such that the first bit entering the algorithm is the start of a codeword. No other prior information is needed, and the algorithm does not assume a particular structure such as systematic or cyclic structures exist.

Based on the results from various tests, the algorithm performs well in environments with log2(n)dB of SNR where n is the codelength. If the codelength is small and a large amount of data is available, the algorithm can work even around 2 or 3dB SNR.

While only AWGN channels were used to gather results, the algorithm would likely

perform better in a burst noise channel. The potential codewords are sorted before they are used, so the codewords during a burst of noise would be avoided. The sorting gives the algorithm an opportunity to blindly discover a generator matrix even if the bursts of noise contain a lot of power. Further testing could confirm if this is the case.

5.1.2 Weaknesses

While the algorithm succeeded in many scenarios, there are obstacles to consider as well. One weakness of the algorithm is larger codes. While the algorithm can accomplish the task, identifying a PC matrix for larger codes requires a high SNR, a lot of computations, and a large dataset.

While the algorithm works well for linear block codes, it fails to provide a valid result for convolutional codes. The algorithm can produce a nullspace for convolutional encoded data at almost any potential codelength. Because the current state of a convolutional encoder relies on previous states as well, local nullspaces can exist for a small set of bits. The algorithm finds these local nullspaces and exits early but fails to capture the full structure of the code. The author is not aware of a method to generalize this algorithm for convolutional codes in addition to linear block codes.

Modulation types that send multiple bits per symbol could be another weakness of the algorithm. For BPSK modulation, the 1 symbol is a set distance away from the 0 symbol. All bits have the same probability that noise could cause the wrong symbol to be received. For modulations with larger constellations, multiple bits are sent per symbol. Some symbols lie in the center of the constellation while others lie on the edge. This means some symbols are easier to mistake for different symbols than others. Therefore, some bits have a higher probability of flipping to the wrong value. It is not known how this would affect the sorting performed at the beginning of the algorithm. Additional testing is required to have an answer on the effects the modulation has on Algorithm 3.1.

5.2 Using Results for Blind Diagnosis

The goal of this thesis was to find an algorithm that locates a PC matrix for a FEC code. The context of this goal is blind diagnosis, where the receiver is not aware of the encoder's parameters. This section will discuss how the located PC matrix fits into the context of blind diagnosis.

Decoding codewords with only the PC matrix is not feasible for larger codes. Specific structures are applied to the bits by the encoder, and the decoder must use this structure to efficiently decode. To solve this problem, the type of FEC code used on the encoder must be identified. The code identification should be easier with a PC matrix, but it still requires some work.

A few parameters are easily available from the PC matrix. The number of columns in the PC matrix matches the codelength. Subtracting the number of rows from the codelength gives the message length or dimension of the code. The rate is found from $\frac{k}{n}$ where k is dimension and n is codelength. These parameters from the size of the matrix may be able to give clues about the type of FEC code applied.

Common tables of codelengths and dimensions for various codes are widely available, such as a list of narrow sense BCH codes. There are also standards that have been established for various communication protocols. If the codelength and dimension match the parameters for a common or standard code, the user can check if the PC matrix works for that code.

If the PC matrix does not match any common codes, the matrix can be manipulated to equivalent forms that may identify structures. The equivalent matrix is found by performing row operations on the PC matrix produced by Algorithm 3.1. Some possible identifying structures include cyclic rows, symbols over larger fields, or a sparse matrix. A future step after the algorithm in this thesis would be to find which structures exist and which one best represents the correct decoder.

CHAPTER 6

CONCLUSION

This chapter provides a conclusion of the concepts discussed in the thesis. Section 6.1 gives an overview of the proposed algorithm in relation to previous research. Section 6.2 describes future work beyond the scope of this thesis.

6.1 Overview of the Proposed Algorithm

In a non-cooperative environment, it is difficult to determine what error correction has been applied to a signal. Blind diagnosis of error correction codes requires identifying the type of FEC code and finding specific parameters for the decoder. A good first step to blind diagnosis of linear block codes is to find an equivalent PC matrix for a code. Since the PC matrix is a way to represent a linear block decoder, it provides rich information about the code.

One previous research publication provided methods that determine a linear block code's PC matrix given a set of noisy bits [11]. This research was effective but required a known codelength and dimension and assumed that the code had an equivalent systematic generator matrix. In contrast, the method proposed in this thesis can discover a generator matrix and PC matrix for any linear block code without prior knowledge of the codelength or dimension. This includes codes that have passed through an interleaver after the encoding process. The proposed method only requires that the bit stream has been synchronized so that the first bit of the stream is the start of a codeword.

Algorithm 3.1 uses rank deficiency to determine when a generator matrix might exist. The only input to the algorithm is LLR, which are used to sort potential codewords for the best candidates. At each potential codelength, the algorithm relies on Gaussian Elimination over the sorted LLR codewords to look for rank deficiency. If no rank deficiency exists, the algorithm advances to a new codelength. Otherwise, the algorithm attempts to correct

errors in the generator matrix. The PC matrix is found from the nullspace of the corrected generator matrix.

A post-processing step checks for an interleaving pattern. If a pattern is found, the bits can be separated into groups based on the original codewords to which the bits belonged. The order of the bits within the codeword is not known. Other post-processing steps, such as analyzing the PC matrix for certain structures, are out of the scope of this thesis.

Algorithm 3.1 was tested on many linear block codes and found to work when noise is applied. The performance varies most based on the codelength, with some variance based on the number of available bits as well. The algorithm found a PC matrix at the correct codelength but with some errors when the SNR is about log2(n). The algorithm was consistently able to correct all errors on the PC matrix when the SNR was increased by 1dB.

6.2 Future Work

This research fills a gap in current literature by generalizing more than other algorithms and by not requiring prior information. By generalizing to any linear block code, even codes that have been interleaved can be discovered. However, there is still work that can expand the current research.

Any algorithm that is implemented in a real system must consider computation time as a factor. The algorithm outlined in this thesis will be more useful with a smaller computation time. A simple method to reduce computation time is to parallelize the algorithm. This requires access to more hardware resources, but the answer would be produced faster. One place to start parallelizing the algorithm is trying multiple potential codelengths at a time. Each step taken to eliminate or parallelize operations will make the algorithm more practical.

Another way to improve the algorithm in this thesis is by reducing the required SNR for the algorithm to succeed. The current approach sorts through the LLR to find the best possible candidates for codewords, while other potential codewords go unused. Ideally, information from all codewords would be used to search for a generator matrix. If the

algorithm was modified to retain the information from all codewords instead of only the selected codewords, the algorithm would discover a generator matrix in lower SNR scenarios.

If the algorithm could generalize to work on convolutional codes as well as linear block codes, this would have a significant impact. The author is not aware of any research algorithm that can blindly decode both linear block codes and convolutional codes.

Finally, further research includes post-processing algorithms that use the generator matrix and PC matrix to determine more information about the code. Some post-processing examples include searching for an equivalent matrix that is low-density or cyclic. Another post-processing step is to find an equivalent matrix that can be returned to a larger field in the case of Reed-Solomon matrices. These post-processing steps would create a higher demand for the proposed algorithm, because it would become easier to use the discovered PC matrix.

By researching these items of future work, the proposed algorithm could become a practical tool for anyone doing blind FEC diagnosis.

REFERENCES

[1] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. 111 River Street, Hoboken, NJ 07030, USA: Wiley, 2021.

[2] J. Dingel and J. Hagenauer, "Parameter estimation of a convolutional encoder from noisy observations," in *2007 IEEE International Symposium on Information Theory*, 2007, pp. 1776–1780.

[3] Z. Jing, H. Zhiping, S. Shaojing, and Y. Shaowu, "Blind recognition of binary cyclic codes," *EURASIP Journal on Wireless Communications and Networking*, vol. 2013, p. 218, 12 2013.

[4] L. Longqing, S. Fangqi, and Z. Jing, "Recovering the parameters of an ldpc code from noisy intercepted sequences," *IEEE Signal Processing Letters*, vol. 29, pp. 617–621, 2022.

[5] Y. Chang, W. Zhang, H. Wang, and Y. Liu, "Fast blind recognition of bch code based on spectral analysis and probability statistics," *IEEE Communications Letters*, vol. PP, pp. 1–1, 07 2021.

[6] S. Ramabadran, A. Madhukumar, G. Wang, and T. Kee, "Blind reconstruction of reed-solomon encoder and interleavers over noisy environment," *IEEE Transactions on Broadcasting*, vol. 64, pp. 830–845, 12 2018.

[7] L. Shi, W. Zhang, Y. Chang, H. Wang, and Y. Liu, "Blind recognition of reed-solomon codes based on galois field fourier transform and reliability verification," *IEEE Communications Letters*, vol. PP, p. 1, 06 2023.

[8] M. Cluzeau and M. Finiasz, "Recovering a code's length and synchronization from a noisy intercepted bitstream," vol. ISIT-09, pp. 2737–2741, 2009.

[9] E. R. M. M. Yasamine Zrelli, Roland Gautier and E. Radoi, "Blind identification of code word length for non-binary error-correcting codes in noisy transmission," *EURASIP Journal on Wireless Communications and Networking*, vol. 2015, p. 43, 2015. [Online]. Available: https://doi.org/10.1186/s13638-015-0294-5

[10] A. Sharma and N. R. Pillai, "Blind recognition of parameters of linear block codes from intercepted bit stream," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, 2016, pp. 1262–1266.

[11] P. Wang, Y. L. Guan, L. Wang, and P. Cheng, "Fast blind recovery of linear block codes over noisy channels," 2023. [Online]. Available: https://arxiv.org/abs/2305.04190

[12] F. Mei, H. Chen, and Y. Lei, "Blind recognition of forward error correction codes based on recurrent neural network," *Sensors*, vol. 21, no. 11, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/11/3884

[13] D. Coppersmith, "Solving homogeneous linear equations over gf (2) via block wiedemann algorithm," *Mathematics of Computation*, vol. 62, pp. 333–350, 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:1328407

[14] I. Flesch, "A new parallel approach to the block lanczos algorithm for finding nullspaces over gf (2)," 2002. [Online]. Available: https://api.semanticscholar.org/CorpusID:18503104

APPENDIX: MATLAB Code for Algorithms

```
function [h, codelength] = find_H_noisy(data_in)
   num_bits = length(data_in);
   max_codelength = floor(sqrt(num_bits));
   codelength = 0;
   h = [];
   for n=7:1:max_codelength
       % Loop through the possible codelengths and find a possible parity
       % check matrix for each one
       total_codewords = floor(num_bits / n);
       num_codewords = n + ceil(sqrt(n));
       M = reshape(data_in(1:1:total_codewords*n), [n, total_codewords])';
       % Sort the rows by confidence of LLR
       M_sorted = sort_llr_confidence(M);
       M_subset = M_sorted(1:num_codewords, :);
       A = llr_rref(M_subset);
       [rows, cols] = size(A);
       if(rows < n-1)
           G = A < 0
           [rows, cols] = size(G)

           % Check for small Hamming weights, remove if they exist
           for weight=1:1:2
               hamm_weights = sum(G');
               inds = find(hamm_weights == weight);
               idx = 1;
               while (length(inds) >= idx)
                   %Try flipping the bit in the related column
                   [rows, cols] = size(G);
                   [G, M_subset, corrected] = flip_bits(M_subset, G(inds(idx),:),
                       rows);
```

```
                % If this error was not corrected, the row will still
                % show up in inds but we want to move on to next one
                if(~corrected)
                    idx = idx + 1;
                else
                % If error was corrected, recheck row weights
                    idx = 1;
                    hamm_weights = sum(G');
                    inds = find(hamm_weights == weight);
                end
            end
        end

        corrected_G = G
        [rows, cols] = size(G)
        h = gf2_nullspace(G, true);
        [rows, cols] = size(h);
        codelength = n;
        break;
    end
  end
end
```

```
function M_out = sort_llr_confidence(M_in)

    abs_M = abs(M_in);
    min_rows = min(abs_M, [], 2);
    [~,idx] = sort(min_rows, 1, "descend");
    M_out = M_in(idx, :);

end
```

```
function A = llr_rref(A)
```

```
% This function finds Row-Reduced Echelon Form for a matrix of LLR


    if(~isempty(A))
        [rows, cols] = size(A);
        r = 1;
        for c=1:1:cols
            % If the current bit is positive (equivalent to '0' bit) swap
            % with another row
            if(A(r, c) > 0)
                inds1 = find(A(:,c) < 0);
                for l=1:1:length(inds1)
                    if(inds1(l) > r)
                        % Swap rows
                        tmp = A(r,:);
                        A(r,:) = A(inds1(l),:);
                        A(inds1(l),:) = tmp;
                        break;
                    end
                end
            end
            % Now that the current bit is negative, do forward elimination
            if(A(r, c) <= 0)
                inds1 = find(A(:,c) < 0);
                for l=1:1:length(inds1)
                    if(inds1(l) ~= r)
                        A(inds1(l),:) = llr_add(A(inds1(l),:), A(r,:));
                    end
                end
                r = r + 1;
            end
        end
        A = A(1:r-1, :);
```

```
    end
end
```

```
function [G_prime, M_prime, corrected] = flip_bits(M, row, dimension);

    G_prime = M < 0;
    M_prime = M;
    corrected = false;
    [num_rows, num_cols] = size(M);
    reverse_row = (-2*row) + 1;
    % loop through all rows and check if there is an error that can be
    % corrected
    for r=1:1:num_rows
        original_row = M(r,:);
        M(r,:) = M(r,:) .* reverse_row; % Flips the bits of the potential error
        G_prime = gf2_rref(M < 0);
        [new_num_rows,~] = size(G_prime);
        if new_num_rows < dimension
            M_prime = M;
            corrected = true;
            break;
        else
            M(r,:) = original_row;
        end
    end
end
```

```
function A = gf2_rref(A)
% This function finds Row-reduced Echelon Form of matrices over GF(2)

    % Confirm matrix is not empty
    if(~isempty(A))
        [rows, cols] = size(A);
```

```
7         r = 1;
8         %For our problem, there should always be more rows than columns
9         if(cols > rows)
10            cols = rows;
11        end
12        for c=1:1:cols
13            % If there is not already a 1 in the diagonal, switch with a row
14            % with a 1 in this column
15            if(~A(r,c))
16                inds1 = find(A(:,c) == 1);
17                for l=1:1:length(inds1)
18                    if(inds1(l) > r)
19                        tmp = A(r,:);
20                        A(r,:) = A(inds1(l),:);
21                        A(inds1(l),:) = tmp;
22                        % break % This break statement slows things down
23                    end
24                end
25            end
26            % Do the forward and backward elimination
27            if(A(r,c))
28                inds1 = find(A(:,c) == 1);
29                for l=1:1:length(inds1)
30                    %if(inds1(l) > r) % use this line for forward only
31                    if(inds1(l) ~= r)
32                        A(inds1(l),:) = bitxor(A(inds1(l), :), A(r,:));
33                    end
34                end
35                r = r + 1;
36            end
37        end
38        A = A(1:r-1, :);
```

```
    end
end
```

```
function H = gf2_nullspace(G, rref_form)
% This function finds the nullspace for a matrix over GF(2). It is assumed
% the matrix is a GF(2) matrix. The matrix must be full rank if rref_form
% is true

    [num_rows, num_cols] = size(G);
    if (~rref_form)
        G = gf2_rref(G);
    end
    % Every pivot column will be labeled with a 1, others with a 0
    pivot_vs_free = zeros(1,num_cols);
    for row=1:1:num_rows
        pivot_idx = find(G(row,:),1);
        pivot_vs_free(pivot_idx) = 1;
    end
    I = eye(num_cols - num_rows);
    P = G(:, find(pivot_vs_free == 0));
    H = zeros(num_cols, num_cols - num_rows);
    H(find(pivot_vs_free),:) = P;
    H(find(pivot_vs_free == 0),:) = I;
end
```