

# Robust Communication: How Math Is Used to Encode and Decode Messages

Riley Debski

Department of Mathematics, University of Pittsburgh, Pittsburgh, PA

*E-mail:* rjd65@pitt.edu

## 1. Introduction

In today's society, effective communication is vital for everything people do from texting and video chatting to shopping online. Whenever a transfer of information happens, there is the possibility that the information will encounter interference that would disrupt its transmission. To protect messages from corruption, a system of *encoding* is used to ensure that a message can be recovered from corrupted transmission. Encoding and *decoding* are done using vector spaces called codes that strategically lengthen a message to protect it from errors. By encoding and decoding a message, effective and reliable transmission can be ensured.

The information presented in this paper follows from the book *Fundamentals of Error-Correcting Codes* by Huffman and Pless [1]. A nine-week independent study was done on the topics of this book. This paper covers the important topics from chapter one of Huffman and Pless. For a more in-depth understanding of the material presented here, refer to [1].

The following is a review of the encoding and decoding process. First, we present a review of the linear and abstract algebra needed to comprehend codes. Then we introduce codes, basic coding theory, and the example of the Hamming Codes. Finally, we present the process of encoding, introducing errors, and decoding. This process is illustrated throughout with the example of the  $[8, 4, 4]$  binary extended Hamming Code. A full example of the entire encoding and decoding process is presented at the end through a Python program written to simulate the outlined steps.

## 2. Mathematics Background

There are many applications of mathematical concepts in the process of encoding and decoding for the transmission of information. Topics include finite fields, vector spaces, and a basis of a vector space. If the reader is already familiar with these topics, this

section may be skipped. Presented is a general review of the topics mentioned above and their relation to codes.

## 2.1. Fields and Finite Fields

The most commonly known *field* is  $\mathbb{R}$ , the real numbers. There are many other fields that are useful for the study of codes.

**Definition 1.** A *field* is an abelian group that is closed under addition and scalar multiplication, where additive and multiplicative inverses are present. The additive identity 0 and multiplicative identity 1 must also be present. The group also follows the rules of associativity, commutativity, and distributivity.

A helpful field for working with codes is  $\mathbb{F}_2$ . This is the finite field of only two elements: 0 and 1. This binary field is the language of computers which makes encoding and decoding practical in this field for application in the real world. Being a finite field,  $\mathbb{F}_2$  is a set with addition and multiplication modulo 2, satisfying the above definition. Modulo 2 refers to taking the answer of the operation, dividing it by 2, and using the remainder as the number in the field.

**Example 1.** Again, all examples will be worked in  $\mathbb{F}_2$ . The addition and multiplication tables for  $\mathbb{F}_2$  are presented below to understand calculations in this finite field. Calculations in  $\mathbb{F}_2$  are simple as there are only two elements. For other examples of working in a finite field, reference [2].

$$\begin{array}{c|cc} + & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array} \quad \begin{array}{c|cc} \cdot & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$$

The typical fields used for encoding are  $\mathbb{F}_2$ ,  $\mathbb{F}_3$ , or  $\mathbb{F}_4$ , but any field can be used to form a code. For information about codes over other fields, see [1]. This paper presents theorems and processes as a general case for any field, but all examples worked will be in  $\mathbb{F}_2$ . The notation used throughout this paper will denote finite fields as  $\mathbb{F}_q$ , and all these fields follow **Definition 1** under addition and multiplication of integers modulo  $q$ . For  $\mathbb{F}_q$  to be a field,  $q$  must be an integer that is prime or a power of a prime, else the definition of a field will not hold. Therefore, all  $q$  referred to throughout this paper will be a prime or a power of a prime.

## 2.2. Vectors and Vector Spaces

The most common vector space used in classrooms is  $\mathbb{R}^n$ , but vector spaces can be over  $\mathbb{F}_q$  as well.

**Definition 2.** A *vector space* is the set of all vectors of a certain length  $n$  with elements that are a member of a field  $\mathbb{F}$ .

In this paper, vector spaces are denoted as  $\mathbb{F}_q^n$  and examples are in  $\mathbb{F}_2^n$ . Elements of a vector in  $\mathbb{F}_q^n$  are indexed from 0 to  $n - 1$ ; the first element of each vector is in the zeroth index. The key operations for working in a vector space are addition and scalar multiplication. Combining those two, we can also use linear combinations.

To write down a vector space, a *basis* is used.

**Definition 3.** A *basis* of a vector space is a linearly independent set of vectors that span the space. A set of vectors span a space when any linear combination of the vectors is in the vector space.

These basis vectors can be put into a matrix where each row is a vector from the basis. This matrix represents the vector space, as the rows of a matrix can be linearly combined to form any other vector of the vector space.

**Example 2.** To form a basis for  $\mathbb{R}^5$ , take vectors with a 1 in each index:  $[10000]$ ,  $[01000]$ ,  $[00100]$ ,  $[00010]$ ,  $[00001]$ . The vectors can be added and scaled by any number in  $\mathbb{R}$  to form any other vector in  $\mathbb{R}^5$ . This is called the standard basis of  $\mathbb{R}^5$ .

The *dot product* acts as another calculation on vectors. The dot product inputs two vectors, outputs a scalar, and is defined as

$$[v_1 v_2 \dots v_n] \cdot [u_1 u_2 \dots u_n] = v_1 u_1 + v_2 u_2 + \dots + v_n u_n. \quad (1)$$

The dot product multiplies a vectors and and another vector. To multiply a vector by a matrix, we treat each column of the matrix as a vector, and the dot product of the vector and column of the matrix is an element of the resulting vector i.e.

$$[v_1 v_2 \dots v_m] \begin{bmatrix} m_{11} & m_{12} & \dots & m_{1n} \\ m_{21} & m_{22} & \dots & m_{2n} \\ \dots & \dots & \dots & \dots \\ m_{n1} & m_{n2} & \dots & m_{nn} \end{bmatrix} = [(v \cdot \vec{m}_{1n})(v \cdot \vec{m}_{2n}) \dots (v \cdot \vec{m}_{nn})] \quad (2)$$

where  $\vec{m}_{1n}$  denotes the first column vector of the matrix,  $\vec{m}_{2n}$  the second column vector, and so on. Performing this calculation on vectors changes the length of the vectors, i.e. it sends vectors from  $\mathbb{F}^m \rightarrow \mathbb{F}^n$ , where  $n$  is the length of the  $n \times n$  matrix.

Another helpful concept for understanding codes is *subspaces*, which are smaller vector spaces contained in a larger vector space.

**Definition 4.** A *subspace* is a set of vectors, closed under addition and multiplication, contained within a larger vector space.

Subspaces reduce the amount of vectors in a space and shorten the basis for a space. Subspaces have the same properties and act the same way as vector spaces as mentioned above, just with fewer vectors. Subspaces have cosets within the larger vector space. Cosets of a subspace are the subspace added to a vector not in that subspace. These

subspaces can include the trivial subspace where you add  $\vec{0}$  to the subspace. Each subspace has multiple cosets that form the larger vector space when joined together.

Subspaces also have a dimension to describe how large the subspace is.

**Definition 5.** The *dimension* of a subspace is the number of vectors in the subspace's basis.

The dimension of a vector space is important for knowing the properties of a subspace and for calculating how many vectors are in the subspace.

### 3. Introduction To Codes

Interruption of communication is due to “noise.” Noise in the form of wind, machine malfunctions, or damage to the transmitter, like a scratch on a CD, corrupts the information being sent. If this incorrect message is received, the communication process fails to transmit the intended message.

#### 3.1. Codes and Their Representations

To protect messages against noise, messages are encoded. To achieve this process, we use *codes*. We let  $\mathbb{F}_q^n$  denote the standard vector space of dimension  $n$ , whose elements are vectors of length  $n$  with elements in the field  $\mathbb{F}_q$ .

**Definition 6.** A *code* is a sub-vector space of  $\mathbb{F}_q^n$ .

As a code is only a subspace of a vector space, it has a dimension which we denote by  $k$ .  $C$  denotes a  $[n, k]$  code. Since the code is a subspace of dimension  $k$ , it has  $q^k$  *codewords*. A codeword is any vector in the code.

To express a code in matrix form, one must choose a basis for the code.

**Definition 7.** Let  $G_{k \times n}$  denote the *generating matrix* of a code, where the rows of  $G$  consist of a basis for the code.

The generating matrix of a code can take many forms, as there is no one basis to express a vector space. The *standard form* of a generating matrix is  $G = [I_k | A]$  where  $I_k$  is the square identity matrix of size  $k$ , and  $A$  is any other matrix that makes the rows of  $G$  a basis for the code.

Another type of matrix representing a code that is important to discuss is the parity check matrix.

**Definition 8.** The *parity check matrix*,  $H_{(n-k) \times n}$ , is such that  $C = \{\vec{x} \in \mathbb{F}_q^n : H\vec{x} = \vec{0}\}$ .

The rows of  $H$  are independent, just like  $G$ . If  $G$  is written in standard form, the parity check matrix is found by  $H = [-A^T | I_{n-k}]$ . Both matrices are good candidates to be used in the encoding and decoding process. Depending on the type of code used, either the generator or parity check matrix is more efficient for encoding.

### 3.2. Minimum Distance

A helpful measurement of a code is the code's *minimum distance*. The *distance* between two codewords is the number of elements by which the codewords differ.

**Definition 9.** The *minimum distance* of a code is the smallest amount of elements by which any two codewords differ.

**Example 3.** Consider the binary code with codewords [1000010] and [1100011]. These codewords have a distance of 2 because they differ only in the second and seventh elements. Now consider the codeword obtained by summing the first and second codewords above [0100001]. As the third codeword is the sum of the first and second, codewords 1, 2, 3, and [0000000] form a code under addition and scalar multiplication in  $\mathbb{F}_2^n$ . Now, the second and third codewords are only different in 2 elements, so the minimum distance of this code is 2. Other codewords in the code could have a distance of 3 and 4, but the minimum distance is the smallest distance between any two codewords of a code.

Minimum distance is an important concept, as it determines the error-correcting capability of a code. This idea will be explored further in the *nearest neighbor decoding* section of this paper. To denote the minimum distance of a code, the variable  $d$  is used and placed in the denotation of a code:  $[n, k, d]$ .

### 3.3. Weight

Another helpful measurement of a code is its weight.

**Definition 10.** A codeword's *weight* is its distance from  $\vec{0}$ .

Much like the minimum distance of a code, a code also has a minimum weight which is the smallest weight of any codeword in the code. The weight of codewords will become useful when discussing *syndrome decoding* later in the paper.

### 3.4. Hamming Codes

The example of the Hamming Code is used to illustrate the process of encoding and decoding. The Hamming Codes follow the structure that  $[n, k, d] = [2^r - 1, 2^r - 1 - r, 3]$  for some integer  $r$ . The significance of the Hamming Codes' minimum distance of 3 is developed later in the paper.

To correct more than one error, the Extended Hamming Code is used. This becomes a  $[2^r, 2^r - 1 - r, 4]$  code with the ability to correct up to two errors. This is done by adding a parity check to the matrix. A *parity check* adds a column to the matrix, where the element for each row is the opposite of the sum of the elements of the row modulo  $q$ . This new column increases the length of the codewords in the code.

**Example 4.** Take the example of the binary Hamming Code [7, 4, 3]. A generating matrix for this code is:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \quad (3)$$

To add a parity check, the elements of each row are added modulo 2 and added in a new column. Since  $-1 = 1$  in binary, the sum of the elements is added as the parity check, as that is also the opposite of the sum. The first row has a sum of 3, which is equal to 1 modulo 2. So a 1 would be added to the end of the first row like so:  $[1, 0, 0, 0, 1, 1, 0] \rightarrow [1, 0, 0, 0, 1, 1, 0, 1]$ . Therefore, adding a parity check to  $G$ , gives

$$\hat{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \quad (4)$$

$\hat{G}$  is then the generating matrix for the  $[8, 4, 4]$  extended Hamming Code. The minimum distance of this extended code increases due to the parity check. This increase in distance allows the code to detect two errors and correct up to two errors, which will be explained later.

Since  $\hat{G}$  is in standard form, the parity check matrix for the  $[8, 4, 4]$  binary Hamming code is easily obtained as

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (5)$$

Since  $H$  has independent columns, permuting the order of the columns does not change the code and preserves the independence of the columns. So by permuting the columns of  $H$  and performing basic row operations,  $H$  becomes

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad (6)$$

As shown above,  $H$  has the first row of all ones. The bottom three rows form the numbers 0 through  $2^r - 1$  in binary in each column. This representation of the  $[8, 4, 4]$  binary Hamming Code will simplify the decoding process.

## 4. Encoding

Encoding a message before transmission protects the message against noise. Encoding is done using either the generating or the parity check matrix of a code. To illustrate the encoding process, the parity check matrix  $H$  with columns of binary numbers as shown in Equation (6) is used. The same process of encoding is used for every code, given a generating or parity check matrix.

To encode a message, the message is multiplied by the matrix. Since a matrix has a specific size, the message must have the same length as the dimension  $k$  of the code. To work with the  $[8, 4, 4]$  binary Hamming Code, messages will have length 4, and elements of the message will be either 1 or 0, as the code is in binary.

**Example 5.** Let  $\vec{m} = [1011]$ . To encode  $\vec{m}$ , multiply it by  $H$  to obtain the codeword for the message.

$$\vec{m}H = [1011] \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} = [10011001] \quad (7)$$

By multiplying the message by the matrix  $H$ , the message is encoded into a length 8 codeword. This codeword is in the code, and as stated earlier, a code has  $q^k$  codewords. For this code, there are  $2^4 = 16$  codewords, because  $q = 2$  and  $k = 4$ , giving messages of length 4. Since each message is in binary and of length 4, there are  $2^4 = 16$  messages that this code can encode. Therefore, there is a bijection between the messages the code can encode and the codewords within a code. So each message has a unique codeword that it is encoded to. This fact is important for decoding because each codeword can be decoded into a single message.

By extending the length of a message when making it into a codeword, encoding protects a message from noise. The extra bits in a codeword allow noise to change a codeword without affecting the code's ability to accurately decode the originally sent message. For the extended Hamming Code, two errors can be corrected when the errors are in specific indices of the codeword. Not all errors can be corrected, but one more can be detected and corrected if it is the appropriate index. This process is illustrated in Example 9.

## 5. Error Introduction From Noise

Before error correcting using  $H$  is explained in depth, some examples of a codeword being corrupted are explored. When noise enters a channel, it disrupts the transmission of the codeword, causing the codeword to have different elements than it did before transmission.

**Example 6.** Consider the codeword  $[10011001]$  from Equation (7). Say the second and fifth elements are corrupted by noise. This means that the second element would flip from a 0 to a 1, and the fifth element would flip from a 1 to a 0. The change in the codeword would be

$$[10011001] \rightarrow [11010001] \quad (8)$$

with the red elements being what was corrupted during transmission.

An error can occur anywhere in a codeword during transmission, and we assume that the probability that each bit of information is corrupted is equal and independent of other bits. As was stated earlier, the  $[8, 4, 4]$  extended Hamming Code can only correct up to two errors, so if a channel has too much noise, the number of errors will be too much for the Hamming Code to handle. Other codes are better equipped to correct more errors, the number of which is determined by the minimum distance of the code.

## 6. Decoding

There are multiple ways to decode, including nearest neighbor decoding and syndrome decoding. There is also a final step after both processes to regain the originally sent message. All of these steps are explained in this section.

### 6.1. Nearest Neighbor Decoding

The minimum distance  $d$  of a  $[n, k, d]$  code describes how different two distinct codewords must be in a code. Hence, the larger the minimum distance of a code, the further apart codewords are from each other. Since these codewords have a minimum distance between them, they are separated by a certain number of elements. This means that other binary vectors of length  $n$ , which are not in the code, are closest to some codeword, as the codewords are separated from each other. Finding this codeword that is closest to a transmitted message with errors is called *nearest neighbor decoding* [1] (Section 1.11.2). This process is the simplest way to decode messages and find which codeword was originally sent. Each binary vector of length  $n$  is sorted into a sphere of radius, labeled  $r$ , which separates vectors into groups of which codeword they are closest to. As long as the spheres of radius for each codeword are disjoint, i.e. a vector is not equally distant from two different codewords, nearest neighbor decoding is simple.

To achieve disjoint spheres, choose the radius labeled  $r$  such that  $r = (d - 1)/2$ . This ensures that the spheres are disjoint. If the spheres are disjoint, like above, at most one sphere can contain the codeword with errors, and the corrected codeword is the codeword in  $C$  for that sphere. If the spheres are not disjoint, a codeword with errors could be the nearest neighbor to two or more codewords which would not effectively decode messages. Therefore nearest neighbor decoding is only helpful for certain codes where spheres are disjoint.



**Example 7.** To see how nearest neighbor decoding works, take the received vector [11110001] that has introduced errors. Using the extended [8, 4, 4] Hamming Code example, it is known that the code has the codeword [11110000]. As this is only one element different from the received vector, this must have been the intended codeword, as the distance between these two vectors is less than or equal to  $r = (4 - 1/2) = 1.5$ . There is no other codeword that has a smaller distance from the received vector. Therefore, [11110000] is the originally sent codeword.

If there are a large amount of codewords, the process of finding the nearest neighbor of each codeword is an inefficient method for decoding. Each vector has to be compared to every codeword to find the closest match or one must find a codeword with a distance less than  $r$  to the vector, which is highly inefficient even though the process is simplistic.

The extra decoding capability of the extended Hamming Code is possible because the minimum distance increases from three to four when extending the code. Increasing the minimum distance of the code increases the spheres of radius of the code which separates codewords further apart than they were in the [7, 3, 3] Hamming Code. This allows for better decoding capabilities of the [8, 4, 4] extended Hamming Code.

## 6.2. Syndrome Decoding

Another way to decode is by syndromes. *Syndrome decoding* is a version of nearest neighbor decoding that uses cosets of  $C$  rather than finding the nearest neighbor of every codeword. To begin, consider a  $[n, k, d]$  code  $C$  over  $\mathbb{F}_q$ . Then every codeword is an element of  $\mathbb{F}_q^n$ . Since  $C$  has the properties of an abelian group,  $\mathbb{F}_q^n$  can be broken up into cosets of  $C$ , each with the same number of vectors as  $C$  according to the theory of Lagrange [2]. Each coset is defined as  $\vec{x} + C = \{\vec{x} + \vec{c}, \vec{c} \in C\}$  for some  $\vec{x}$  not in  $C$  or  $x = \vec{0}$  which would result in the trivial coset. This process is done for each coset with a different  $\vec{x}$  not in  $C$ .

Each of these cosets can be defined by a *coset leader*  $\vec{s}$  [1], which is a vector of smallest weight in the coset. If a coset has multiple vectors of the smallest weight, the decoder may choose which vector to use as the coset leader. Doing this breaks  $\mathbb{F}_q^n$  into  $q^{n-k}$  cosets of  $C$ , which is much more manageable than inspecting each vector individually in nearest neighbor decoding.

**Example 8.** Take the code used in Example 3. The codewords in this code are [0000000], [1000010], [1100011], and [0100001]. To find a coset, take the vector [1111111], which is not in the code, and add it to every codeword. This coset now contains the codewords [1111111], [0111101], [0011100], and [1011110]. The vector [0011100] has the smallest weight of 3 and is this coset's coset leader. This process is done to find each coset in  $\mathbb{F}_2$  with a new vector that is not in any previously found coset.

Once the cosets and coset leaders are found, a syndrome is associated with each coset. To find the syndrome of a vector, multiply the parity check matrix by the transpose of

the vector:

$$\text{syn}(\vec{v}) = H\vec{v}^T \quad (9)$$

This calculation gives a vector in  $\mathbb{F}_q^{n-k}$ , and every vector in  $\mathbb{F}_q^{n-k}$  is a syndrome for vectors in  $\mathbb{F}_q^n$ . Since there are  $q^{n-k}$  cosets and  $q^{n-k}$  syndromes in  $\mathbb{F}_q^{n-k}$ , there is a one-to-one correspondence between the cosets of  $C$  and the syndromes found by Equation (9). It also happens that two codewords are in the same coset if and only if they have the same syndrome. For a proof of this, reference [1] (p. 41). Therefore, each coset has a unique associated syndrome and coset leader. Creating a table of cosets, coset leaders, and associated syndromes will make the decoding process quicker, as the table can be referenced for each calculation.

To decode a vector, one must:

- i Find the syndrome of a received vector with error corruption
- ii Locate the coset leader associated with the syndrome
- iii Calculate the originally sent codeword as the errored vector minus the coset leader

$$\vec{c} = \vec{v} - \vec{s} \quad (10)$$

If  $\vec{v}$  is an element of  $C$  or the trivial coset, then its syndrome is  $\vec{0}$ , and the sent vector is the original codeword with no errors to correct.

This process of decoding to find the original codeword sent is more efficient in that it works with  $q^{n-k}$  vectors instead of  $q^n$  vectors. Unfortunately, this process does entail an initial setup of cosets and associated syndromes and a matrix multiplication for each transmitted vector.

### 6.3. Hamming Codes and Decoding

Fortunately, Hamming Codes are especially efficient at decoding using the syndrome method.

Once a syndrome for the sent codeword is found, the syndrome will tell where the error in the code is based on the binary number the syndrome expresses. If the syndrome is  $\vec{0}$ , there is no error. Any other binary representation shows the place in the codeword where the bit was flipped.

**Example 9.** To illustrate the process of decoding with the extended [8, 4, 4] Hamming Code, take  $H$  from equation (6) and the codeword from Equation (7).

- i Introduce errors to [10011001] to obtain the vector [11011001]

ii Compute the syndrome of this vector using the formula from Equation (9):

$$H\vec{v} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} [11011001] = [1001] \quad (11)$$

Ignoring the first element for the moment, the binary number in the last three bits of the syndrome indicates an error in the first index of the vector. Flipping the bit in index one of the vector obtains the codeword [10011001], which was shown earlier to be the encoded version of the message [1011]. The last three bits of the syndrome indicate the index where an error has occurred. If just the Hamming Code was used, the syndrome would only be three elements long and would indicate where the error is by those three bits.

iii By using the extended Hamming Code, a second error can be detected and corrected, as long as that error is in the zeroth or leftmost index of the vector. Consider the errored vector [01011001], which is the same vector as above with an error added in the zeroth index. Calculating the syndrome for this equation results in:

$$H\vec{c} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} [01011001] = [0001] \quad (12)$$

Equation (12) gives a nearly identical syndrome to that of Equation (11). The zero in the zeroth index of this syndrome indicates a second error, and the bit in the zeroth index of the vector is incorrect. Having a 1 in said bit of the syndrome indicates there is only one error in the vector, meaning there is no error in the zeroth bit of the vector.

#### 6.4. Finishing Decoding

The final step for decoding is recovering the original message that was encoded. Once the error-corrected codeword is found, multiply it by the inverse of the matrix that was used to encode. Doing this multiplication results in the original message.

**Example 10.** Take the matrix  $K$  below for the running example of the [8, 4, 4] binary extended Hamming Code.

$$K = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (13)$$

$HK = I_4$ , where  $I_4$  is the  $4 \times 4$  identity matrix. To recover  $\vec{m}$ , multiply  $K$  by  $\vec{m}$  to obtain:

$$\vec{m}K = [10011001] \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = [1011] \quad (14)$$

This recovers the original message  $\vec{m} = [1011]$  that was encoded in Equation (7). This step finishes the process of encoding and decoding a message.

Encoding a message protects the message from errors during transmission. If the original message is sent without encoding and adding extra parity bits, the receiver is much more likely to receive the wrong message because there is no method to correct errors without encoding.

## 7. Python Program

To illustrate the entire process with the same original message and random errors, the following Python output is presented. A program was written to illustrate the encoding and random error input such that the Hamming Code could then error correct and decode the message. The following example works with the message  $\vec{m} = [1001]$  and shows three different encoding and decoding processes.

```

'''
RESTART: C:\Users\Riley\Dropbox\Summer Cryptography\riley python\encodingError:
ngProgram.py
The 8-bit encoded version of [1, 0, 0, 1] is :
[1, 0, 1, 0, 1, 0, 1, 0]
The 8-bit encoded version of [1, 0, 0, 1] with random errors introduced is :
[1, 0, 1, 0, 1, 0, 1, 0]
The syndrome of the codeword with errors is :
[0, 0, 0, 0]
The corrected codeword is :
[1, 0, 1, 0, 1, 0, 1, 0]
The decoded message is :
[1, 0, 0, 1]
>>>
RESTART: C:\Users\Riley\Dropbox\Summer Cryptography\riley python\encodingError:
ngProgram.py
The 8-bit encoded version of [1, 0, 0, 1] is :
[1, 0, 1, 0, 1, 0, 1, 0]
The 8-bit encoded version of [1, 0, 0, 1] with random errors introduced is :
[0, 0, 1, 0, 1, 0, 0, 0]
The syndrome of the codeword with errors is :
[0, 1, 1, 0]
The corrected codeword is :
[1, 0, 1, 0, 1, 0, 1, 0]
The decoded message is :
[1, 0, 0, 1]
>>>
RESTART: C:\Users\Riley\Dropbox\Summer Cryptography\riley python\encodingError:
ngProgram.py
The 8-bit encoded version of [1, 0, 0, 1] is :
[1, 0, 1, 0, 1, 0, 1, 0]
The 8-bit encoded version of [1, 0, 0, 1] with random errors introduced is :
[1, 0, 0, 0, 1, 0, 1, 0]
The syndrome of the codeword with errors is :
[1, 0, 1, 0]
The corrected codeword is :
[1, 0, 1, 0, 1, 0, 1, 0]
The decoded message is :
[1, 0, 0, 1]
>>>

```

This program is hard coded to encode and decode messages based on the  $[8,4,4]$  Extended Binary Hamming Code. Error introduction is semi-random. Errors are coded to possibly happen in the first bit of the vector and one of the other seven bits. As much as this program mimics real-world error introduction, it is coded to only introduce errors that the code can handle. More errors can not be corrected by the code and are therefore omitted in the program. Overall, the program aims to simulate real-world error introduction and the process of decoding for each error.

## 8. Conclusion

The coding process is essential for effective communication in today's society where communication is highly digital. Without these processes, naturally occurring errors corrupt messages, and messages become difficult to interpret. Using the Hamming Codes is imperfect and will encounter problems when codewords have too many errors to easily be decoded. Increasing the distance between codewords in a code aids the decoding process by making it easier to find the intended message. Using coding theory protects messages from corruption that happens in everyday communication.

## References

- [1] W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge, Cambridge University Press, 2003.
- [2] John B. Fraleigh. *A First Course in Abstract Algebra*. Addison-Wesley, 2003.