Boise State University
ScholarWorks

Mathematics Undergraduate Theses | Department of Mathematics

8-2014

A Brief Encounter with Linear Codes

Brent El-Bakri
Boise State University, brentelbakri@boisestate.edu

Follow this and additional works at: http://scholarworks.boisestate.edu/math_undergraduate_theses

Recommended Citation
El-Bakri, Brent, "A Brief Encounter with Linear Codes" (2014). *Mathematics Undergraduate Theses.* Paper 3.

A Brief Encounter with Linear Codes

Brent El-Bakri
Boise State University

August 21, 2014

Abstract

While studying irreducible polynomials and fields our abstract algebra professor briefly mentioned that they play a role in error correcting and detecting codes. It was this comment that fostered the idea of further study on this topic. It would give us the opportunity to to apply some abstract concepts to the physical realm. The information contained herein is from researching the topic. Most of the material relies upon matrices and polynomials. While it is brief in nature it does lay a simple foundation of the some basic concepts of linear block blocks along with a subset known as cyclic codes. These cyclic codes have a great deal of structure and we further bring the abstract into the physical world by showing how cyclic codes can be implemented in hardware. The information here is just the tip of iceberg so there is so much more!!

MSC Primary 94B05, 94B15.
Keywords and phrases: Linear Codes, Cyclic Codes

1 Introduction

When was the last time someone ask you to repeat something? We all undoubtedly have had conversations where we had to repeat ourselves due to the environment around us, whether it was from an unexpected noise or perhaps we were in a noisy situation to begin with like a concert. This repeating allows us to get our message across and hopefully the reception is interpreted properly. This is a crude example of a simple communication system that exemplifies the need for some form of error correcting or detecting. In today's society most information is transmitted or stored in digital form. We see this every day such as online transactions, images, songs, and even our TV broadcasts. Safeguards need to be in place so what is received is what was sent without errors. We don't want our online transaction billing our credit card a $100 for a $10 dollar item, or our songs lyrics to be unrecognizable.

Suppose you are printing a file wirelessly from your laptop to a printer. It is possible to model the data being transferred as string of 0's and 1's. Usually when a 0 is sent the printer will receive a 0 and the same thing with a 1. But occasionally noise (hardware failures, electromagnetic interference and etc) will cause the 0 to become a 1 or a 1 to become a 0. We would like to create ways to overcome these errors that can happen during transmission.

Or what if you were streaming a video wirelessly to your laptop. You would hate for your laptop to issue vertical syncs at the wrong times, it would make viewing unpleasant.

Error correcting codes are used to correct and detect errors that can occur when data is being transmitted across some noisy channel or stored on some medium. When images are transferred back to earth from deep space probes, error codes are used to help fight the noise caused by atmospheric conditions such as lighting and other sources. Music CD's use error codes so that a CD player can read data from CD even if the data has been corrupted by a defect in the CD.

In systems where data is encrypted, correcting errors is even more important. Having one wrong bit in the received ciphertext can cause a lot of changes in the decrypted message. We would hate to be charged for $100 for that $10 purchase we made online!

1.1 Some Examples

In this section we will introduce a few example of error-control codes. This will help us see the difference between error-detecting and error-correcting codes.

Example 1.1. (The Repetition Code) *This particular example will illustrate a simple error-correction code. Suppose we only have two messages to send, 'YES' or 'NO'. We will let 0 represent 'YES' and a 1 to represent 'NO'. If we send a 0 it is possible to receive a 1 due to the noise in the channel. What we want to do is increase the probability that the received message is correct. Instead of sending a single bit we send 00000 for 'YES' and 11111 for 'NO' and the receiver will use a majority rule to decode the result. So if zero, one or two bits are in error we will still decode the proper message.*

Example 1.2. (ISBN) *This simple example that is often shown in textbooks is the ISBN code and the reader is encourage to look at [3] [4]. The first 9 digits of an ISBN (x_1 x_2...x_9) gives us information about the book, like language, publisher and etc. These numbers are bound between 0 and 9 inclusive. To guard against errors, the nine-digit number is encoded to a ten-digit number where the appended tenth digit is chosen so that all 10 digits satisfy*

$$\sum_{i=1}^{10} ix_i \equiv 0 \; mod \; 11$$

This tenth digit also known as a check digit *may equal 10 and if this happens an 'X' is used. This code can detect any single digit error.*

The main purpose of error coding theory is to encode channel data with enough redundancy and in an efficient manner so any errors can be corrected/detected. This next example will provide a simple method to encode messages.

Example 1.3. *Let* $C = \{000, 110, 101, 011\}$ *be the set used to transmit* $\{00, 11, 10, 01\}$. *Then it will be able to detect a single bit error. The following table will help show this:*

Data	*Encoded Data*	*Received Data, single bit error*
00	*000*	*001, 100, 010*
01	*011*	*001, 010, 111*
10	*101*	*001, 100, 111*
11	*110*	*100, 010, 111*

As can be seen any single bit error will be detected since the received words are not valid messages (or codewords*).*

The interesting thing about 1.3 is that each encoded message can be shifted right by one and still be a valid message. It's also possible to add any of the codewords together and still get a codeword. This type code is known as a *cyclic code* We will give a more formal definition of a cyclic code shortly. These codes are important for two main reasons:

I Encoding and decoding computations can be implemented with shift and feedback registers.

II They have considerable inherent algebraic structure.

The rest of this paper will be provide the reader with the necessary information to understand the math that is used to create such codes (mainly linear and abstract algebra). Along with this math background, we will also look at the ideas behind basic linear codes since cyclic codes are a subset of linear block codes. We will then examine the structure of cyclic codes along with some examples of hardware implementation.

1.2 Definitions and Terminology

The figure below shows the flow of information in a typical digital communication system [6].

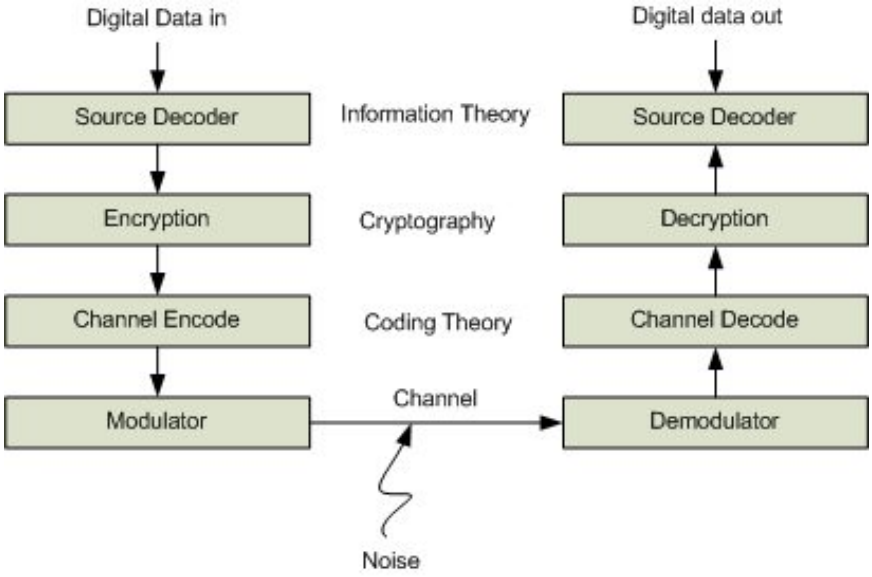


Figure 1: A simple communication system

Throughout this paper we model our *digital data in* as strings of discrete symbols such as the binary symbols 0,1. Each string will be referred to as a message or just data. Once the message travels through the *channel encoder* it becomes a codeword. The second column of example 1.3 shows the codewords for the messages in the first column. The complete set of codewords is what will call a code $\mathbf{C}$. The *channel* will be modeled as additive white Gaussian noise, which we model as string of symbols that get added symbol wise to each packet of data sent across the channel. In our case we are using binary symbols so addition is done modulo 2. For example if our message $\mathbf{m} = 01$ as in 1.3, we would transmit a codeword $\mathbf{c} = 011$. If we had a noisy channel and the receiver received $\mathbf{r} = 001$, we would say noise word or vector $\mathbf{e} = 010$. In other words :

$$m \rightarrow Encode \rightarrow c \rightarrow Noise \rightarrow c + e = r \rightarrow Decode \rightarrow \hat{m}$$

Where m is the message and $\hat{m}$ is whats received. We hope $m = \hat{m}$.

A *q-ary code* is given set of sequences of symbols from where each symbol is from a set F_q of q distinct elements. F_q is called the *alphabet*. Example 1.1 and 1.3 above can also be referred to as a 2-ary code. Example 1.2 is 11-ary.

The efficiency rate of a code is defined as a ratio of the number of symbols in the message to the number of symbols in the codeword. In Example 1.3 our messages are 2 bits and the each code word is 3 bits for the rate is 66.67

2 Introduction to Linear Block Codes

Let's start with a vector space $\mathbf{V}_n$ of all n-tuples over $GF(q)$,

$$\mathbf{V}_n = \{(a_0, a_1..., a_{n-1}) | a_i \in GF(q)\}$$

and define what a linear code is.

Definition 2.1. *A subset* $\mathbf{C}$ *of* $\mathbf{V}_n$ *is a* **linear code** *if and only if* $\mathbf{C}$ *is a subspace.*

With $\mathbf{C}$ defined as a subspace we can represent all the vectors (or codewords) in the subspace by the row space of a $(k \times n)$ matrix $\mathbf{G}$, called a generator matrix of $\mathbf{C}$. We may use the term *code space* interchangeably with subspace when referring to $\mathbf{C}$.

$$\mathbf{G} = \begin{bmatrix} g_{00} & g_{01} & \cdots & g_{0,n-1} \\ g_{10} & g_{11} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_{k-1} \end{bmatrix}$$

The codespace $\mathbf{C} = span(\mathbf{g}_0, \mathbf{g}_1 \cdots \mathbf{g}_{k-1})$ so if these row vectors are linearly independent $\mathbf{C}$ has a *rank* or *dimension* of k. We will assume for this paper that rows of $\mathbf{G}$ are linearly independent. And since we are over a field $GF(q)$ that has q elements we will have q^k *codewords* in $\mathbf{C}$ and we will often refer to it as an (n, k) code.

Example 2.1. *Let the matrix* $\mathbf{G}_1$ *, of a (5,3) code given as follows:*

$$G_1 = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \quad \textit{over GF(2)}$$

Since the rows of $\mathbf{G}_1$ *are linearly independent, we see that we have* $2^3 = 8$ *codewords for this code which are*

$$C_1 = \{00000, 10110, 11011, 01010, 01101, 11100, 10001, 00111\}$$

Example 2.2. *Let* $\mathbf{G}_2$ *be a (4,2) linear code* $\mathbf{C}_2$ *over GF(3) :*

$$G_2 = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \end{bmatrix} \quad \textit{over GF(3)}$$

$$C_2 = \{0000, 1011, 0112, 2022, 0221, 1120, 2210, 1202, 2101\}$$

Again with the rows of $\mathbf{G}_2$ *being linearly independent we see that we get* $3^2 = 9$ *codewords [4].*

One of the important concepts about codes is related to the next two definitions and we will see shortly why this is so.

Definition 2.2. *The* Hamming weight *of a code vector* $\mathbf{c} = (c_0, c_1 ..., c_{n-1})$, $w(\mathbf{c})$ *is the number of nonzero components in* $\mathbf{c}$.

Definition 2.3. *The* Hamming distance *between two vectors* c_1 *and* c_2, $d(c_1, c_2)$, *is the Hamming weight of* $c_1 - c_2$ *or the number of positions in which the two codewords differ.*

Looking back at the $\mathbf{C}_2$ code we see that the Hamming weight for any nonzero codeword is 2. The *minimum distance* (d_{min} or just *distance*) of a code $\mathbf{C}$ is the minimum distance between all distinct pairs of codewords. The distance of a linear code is also the minimum weight of (nonzero) codewords.

Since we are using linear combinations of the rows of $\mathbf{G}$ to also represent codewords or the fact that these combinations are still in the code space , we can say it's a *linear code*. We also say that it is a *linear block code* since all codes are fixed length, namely n. In this paper we may use code, linear code or linear block code to mean the same thing.

Definition 2.4. *A* $(n \times k)$ *matrix* $\mathbf{G}$ *is said to be in* systematic form *if*

$$\mathbf{G} = \begin{bmatrix} \mathbf{I}_k & \mathbf{P} \end{bmatrix} \text{ or } \mathbf{G} = \begin{bmatrix} \mathbf{P} & \mathbf{I}_k \end{bmatrix}$$

Theorem 2.1. *Let* $\mathbf{G}$ *be an* $(n \times k)$ *matrix in* systematic form *as*

$$\mathbf{G} = \begin{bmatrix} \mathbf{I}_k & \mathbf{P} \end{bmatrix}$$

then its null space *is given by the row space of* $\mathbf{H}$,

$$\begin{bmatrix} -\mathbf{P}^T & \mathbf{I}_{n-k} \end{bmatrix} \text{ or } \begin{bmatrix} \mathbf{I}_{n-k} & -\mathbf{P}^T \end{bmatrix}$$

where $\mathbf{P}^T$ *is the transpose of* $\mathbf{P}$ *and* $\mathbf{I}_{n-k}$ *is the identity matrix of rank* $n-k$, *and just negate* $\mathbf{P}^T$ *to get* $-\mathbf{P}^T$.

Proof. The proof of this is straight forward. If we expand $\mathbf{G}$ and $\mathbf{H}$ out and then evaluate $\mathbf{g}_i \cdot \mathbf{h}_j$ for all i, j where $\mathbf{g}_i \cdot \mathbf{h}_j$ represents the dot product of the rows of $\mathbf{G}$ and $\mathbf{H}$, we see that each product is equal to zero.

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & p_{0,0} & p_{0,1} & \cdots & p_{0,n-k-1} \\ 0 & 1 & 0 & \dots & 0 & p_{1,0} & p_{1,1} & \cdots & p_{1,n-k-1} \\ & \vdots & & & & \vdots & & & \vdots \\ 0 & 0 & 0 & \dots & 1 & p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} \end{bmatrix} = \begin{bmatrix} \mathbf{g}_1 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_k \end{bmatrix}$$

$$\mathbf{H} = \begin{bmatrix} -p_{0,0} & -p_{1,0} & \dots & -p_{0,n-k-1} & 1 & 0 & 0 & \dots & 0 \\ -p_{0,1} & -p_{1,1} & \dots & -p_{1,n-k-1} & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots & & \vdots & & & \vdots \\ -p_{0,n-k-1} & -p_{1,n-k-1} & \dots & -p_{k-1,n-k-1} & 0 & 0 & 0 & \dots & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{h}_0 \\ \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_{n-k} \end{bmatrix} \tag{2.1}$$

In other words

$$\mathbf{G} \cdot \mathbf{H}^T = \mathbf{0}.$$

One final thing we need to show is the rank($\mathbf{H} = n$ - rank$\mathbf{G}$. And this follow from the fact that each matrix is in reduced echelon form. □

2.1 Dual code

We know from linear algebra that given a subspace of dimension k in a vector space $\mathbf{V}_n$ of dimension n, there exists a complement of $\mathbf{V}_n$ with a dimension of $r = n - k$ which is also a subspace. So in effect we could also have defined a code $\mathbf{C}$ in terms of this null space matrix.

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_0 \\ \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_{r-1} \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & \cdots & h_{0,n-1} \\ h_{10} & h_{11} & \cdots & h_{1,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ h_{r-1,0} & h_{r-1,1} & \cdots & h_{r-1,n-1} \end{bmatrix}$$

$\mathbf{H}$ is called the *parity check* matrix or the $\mathbf{H}$ matrix of the code. We will also require the r rows of $\mathbf{H}$ to be linearly independent, so its rank for an (n, k) code is $r - n - k$.

By the definitions above we have for any $\mathbf{u} \in \mathbf{C}$,

$$\mathbf{u} \cdot \mathbf{H}^T = \mathbf{0}$$

This equation will play in important part in our decoding of received codewords.

Finding $\mathbf{H}$ from $\mathbf{G}$ is pretty straight forward. We know that reducing $\mathbf{G}$ to its canonical for by row operations will preserve its row and null spaces. So if we reduce $\mathbf{G}_1$ from example 2.1 we get,

$$G_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{I}_3 \ \mathbf{P} \end{bmatrix}$$

And

$$\mathbf{H}_1 = \begin{bmatrix} -\mathbf{P}^T \ \mathbf{I}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Where the $\mathbf{I}_k$ is the $(k \times k)$ identity matrix and the above are in *systematic forms*.

2.2 Encoding and Decoding

What purpose do the generator matrix $\mathbf{G}$ and the parity check matrix $\mathbf{H}$ serve? Simply put $\mathbf{G}$ allows us to encode a message and $\mathbf{H}$ helps us determine if the received message has any errors.

If we let $\mathbf{G}$ be in *systematic form*,

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & p_{0,0} & p_{0,1} & \cdots & p_{0,n-k-1} \\ 0 & 1 & 0 & \dots & 0 & p_{1,0} & p_{1,1} & \cdots & p_{1,n-k-1} \\ & \vdots & & & & \vdots & & & \vdots \\ 0 & 0 & 0 & \dots & 1 & p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} \end{bmatrix}$$

then we can encode a data $\mathbf{d} = (d_0, ...d_{k-1})$ into a codeword $\mathbf{u}$ by

$$\mathbf{u} = \mathbf{dG} = (d_0, ...d_{k-1})\,[\mathbf{I}_k\mathbf{P}] = (d_0, ..., d_{k-1}, p_k, ..., p_{n-1})$$

Here the parity bits p_i are:

$$p_k = d_0 \cdot p_{0,0} + d_1 \cdot p_{1,0} + \cdots d_{k-1} \cdot p_{k-1,0}$$

$$p_{k+1} = d_0 \cdot p_{0,1} + d_1 \cdot p_{1,1} + \cdots d_{k-1} \cdot p_{k-1,1}$$

$$\vdots$$

$$p_{n-1} = d_0 \cdot p_{0,n-k-1} + d_1 \cdot p_{1,n-k-1} + \cdots + d_{k-1} \cdot p_{k-1,n-k-1}$$

This type of encoding is really easy to implement in hardware.

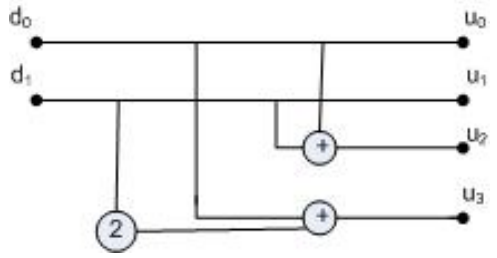


Figure 2: Simple hardware encoding for the $\mathbf{C_2}$ code.

Like the encoding process above, we also decode the received codeword in a parallel fashion by using the $\mathbf{H}$ matrix of the code. Suppose $\mathbf{v} = (v_0, v_1, ..., v_{n-1})$. Then let

$$\mathbf{s}(\mathbf{v}) = \mathbf{v} \cdot \mathbf{H}^T$$

be called the *syndrome*. If $\mathbf{s}(\mathbf{v}) = \mathbf{0}$ then $\mathbf{v}$ is a codeword and no errors are detected. If $\mathbf{s}(\mathbf{v}) \neq \mathbf{0}$ then errors exist. Remember that $\mathbf{C}$ is the null space of $\mathbf{H}$ so if the syndrome is equal to zero than we know $\mathbf{v}$ was a codeword. So the syndrome detects errors. The detecting and correcting of errors is handled by the decoder. If we now

let the actual codeword that was transmitted without errors be $\mathbf{u} = (u_0, u_1, ..., u_{n-1})$ and let the error word be $\mathbf{e} = (e_0, e_1, ..., e_{n-1})$ then we can write

$$\mathbf{v} = \mathbf{u} + \mathbf{e}$$

and

$$\mathbf{s}(\mathbf{v}) = \mathbf{v} \cdot \mathbf{H}^T = \mathbf{u} \cdot \mathbf{H}^T + \mathbf{e} \cdot \mathbf{H}^T = \mathbf{0} + \mathbf{e} \cdot \mathbf{H}^T = \mathbf{e} \cdot \mathbf{H}^T = \mathbf{s}(\mathbf{e})$$

This means that the syndrome of $\mathbf{v}$ is a linear combinations of the columns of $\mathbf{H}$.

It also shows that if there is only one single bit error, in the i^{th} position of $\mathbf{v}$, then the corresponding syndrome vector $\mathbf{s(e)}$ will be nonzero and will be equal to the i^{th} column of $\mathbf{H}$, so we know what location the error occurred in. The following example will help illustrate this.

Example 2.3. *Consider the code* $\mathbf{C}_1$ *in Example 2.1, let* $\mathbf{u} = (11011) \in \mathbf{C}_1$ *be a codeword and let the error be* $\mathbf{e}$= (01000) *so that*

$$\mathbf{v} = \mathbf{u} + \mathbf{e} = (10011)$$

And we know $\mathbf{H}$ *for* $\mathbf{C}_1$ *is*

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad = \quad \left[\mathbf{h}_0, \mathbf{h_1}, \mathbf{h}_2, \mathbf{h}_3, \mathbf{h}_4\right].$$

Thus the syndrome is

$$\mathbf{s}(\mathbf{v}) = \mathbf{v} \cdot \mathbf{H}^T = (10011) \begin{bmatrix} \mathbf{h}_0 \\ \mathbf{h}_1 \\ \mathbf{h}_2 \\ \mathbf{h}_3 \\ \mathbf{h}_4 \end{bmatrix} = \mathbf{h}_0 + \mathbf{h}_3 + \mathbf{h}_4 = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

while

$$\mathbf{e} \cdot \mathbf{H}^T = (01000) = \mathbf{h}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

As seen the syndrome is not equal to zero so the code correctly detects that the error occurred in the received word $\mathbf{v}$.

Another decoding process commonly known as *coset* or *standard array decoding* can decode not just single-bit errors but also multi-bit errors.

Remember that $\mathbf{C}$ (n, k) code, is a a vector subspace. In particular it's an additive subgroup of $\mathbf{V}^n$. We know from abstract algebra that subgroups can be used to partition the whole group [3], [5]. In this case we can use $\mathbf{C}$ which has q^k elements to partition $\mathbf{V}^n$ into q^{n-k} cosets ($|\mathbf{V}^n/\mathbf{C}| = q^n/q^k = q^{n-k}$). We form an array where rows are the cosets in the following way.

1. Write down all the codewords of the code in a row starting with the zero codeword in column 1.

2. Select from the remaining unused vectors of $\mathbf{V}^n$ one of minimal weight, say $\mathbf{e}$. Write $\mathbf{e}$ in the column under the zero codeword, then add $\mathbf{e}$ to each codeword in the first row and place each sum under the corresponding codeword.

3. Continue the process until all vectors in $\mathbb{F}_q^n$ have been placed in the array.

Each row of the array lists the elements in a coset of $\mathbf{V}^n$. The first element in each row is called a *coset leader*. They represent the error patterns that can be corrected by this code and horizontal lines indicate where the error weight of $\mathbf{e}$ changes.

Example 2.4. *Let a (7,3) be given by the following generator matrix*

$$\mathbf{G}_3 = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 &) & 1 \end{bmatrix}$$

the code words are then

$C_3 = (0000000,\ 0111100,\ 1011010,\ 110010,\ 1101001,\ 1010101,\ 0110011,\ 0001111)$

By following the steps above we can create what's called a standard array. *We first select one of the remaining 7-tuples from $\boldsymbol{F}_2^7$ of minimal weight, say (1000000), and build the second row. Once this row is completed, we pick the next minimal weight 7-tuple word and repeat the process until all words are accounted for.*

row 1	*0000000*	*0111100*	*1011010*	*1100110*	*1101001*	**1010101**	*0110011*	*0001111*
row 2	*1000000*	*1111100*	*0011010*	*0100110*	*0101001*	*0010101*	*1110011*	*1001111*
row 3	*0100000*	*0011100*	*1111010*	*1000110*	*1001001*	*1110101*	*0010011*	*0101111*
row 4	*0010000*	*0110100*	*1001010*	*1110110*	*1111001*	*1000101*	*0100011*	*0011111*
row 5	*0001000*	*0110100*	*1010010*	*1101110*	*1100001*	*1011101*	*0111011*	*0000111*
row 6	*0000100*	*0111000*	*1011110*	*1100010*	*1101101*	*1010001*	*0110111*	*0001011*
row 7	*0000010*	*0111110*	*1011000*	*1100100*	*1101011*	*1010111*	*0110001*	*0001101*
row 8	*0000001*	*0111101*	*1011011*	*1100111*	*1101000*	*1010100*	*0110010*	*0001110*
row 9	*1100000*	*1011100*	*0111010*	*0000110*	*0001001*	*0110101*	*1010011*	*1101111*
row 10	*1010000*	*1101100*	*0001010*	*0110110*	*0111001*	*0000101*	*1100011*	*1011111*
row 11	*0110000*	*0001100*	*1101010*	*1010110*	*1011001*	*1100101*	*0000011*	*0111111*
row 12	**1001000**	*1110100*	*0010010*	*0101110*	*0100001*	**0011101**	*1111011*	*1000111*
row 13	*0101000*	*0010100*	*1110010*	*1001110*	*1000001*	*1111101*	*0011011*	*0100111*
row 14	*0011000*	*0100100*	*1000010*	*1111110*	*1110001*	*1001101*	*0101011*	*0010111*
row 15	*1000100*	*1111000*	*0011110*	*1000010*	*0101101*	*0010001*	*1110111*	*1001011*
row 16	*1110000*	*1001100*	*0101010*	*0010110*	*0011001*	*0100101*	*1000011*	*1111111*

If the received codeword is $\mathbf{v}$*=(0011101) (shown in bold) then the coset leader is* $\mathbf{e}$*=(1001000) and the codeword sent is* $\mathbf{c}$*=(1010101).*

The standard array can be used to decode linear codes but it's not very efficient. It suffers from a major problem and that's the amount of memory required to hold the array. In our example above it's not an issue but if our code is say (256,200), which isn't very big by today's standards, it would require $2^{256} \approx 1.2 \times 10^{77}$ vectors each 256 bits in width. That is a lot of storage, not to mention a lot of entries to compare against. This table can be reduced to just two columns by using what's called *syndrome decoding* [6] but it it can still be impractical for today's codes.

As we have seen the decoder has two main functions. And since it doesn't know what $\mathbf{u}$ or $\mathbf{e}$ is it must first determine whether $\mathbf{v}$ has any transmission errors. If it has errors, the decoder will take action to locate and and correct them (if it can) or request a re-transmission of $\mathbf{v}$.

2.3 Hamming Weight and Distance

We mentioned earlier that the properties of the linear codes such as *Hamming weight* and *Hamming distance* play an important role in these codes. We also

mentioned that the *distance* of a code $\mathbf{C}$ is referred to as d_{min} and it equals the w_{min}, minimum weight of the linear code. This parameter determines the error-correcting and the error-detecting capabilities of the code [3], [6].

It turns out that :

Theorem 2.2. *If the* Hamming weight *of a linear code is at least* $2t+1$*, then the code can correct any* t *or fewer errors, or the same code can detect any* $2t$ *or fewer errors.*

The proof may be found in [3,§31] or [6, §3.4].

If s, t are non-negative integers and $d_{min} \geq 2t + s + 1$ then the code can detect any s errors and correct any t errors. This means that for instance if the $d_{min} = 6$ for a code $\mathbf{C}$ we would have the following options:

1. Detect any 5 errors ($t = 0, s = 5$).
2. Correct any one error and detect any four errors ($t = 1, s = 4$).
3. Correct any two errors and any detect any two errors ($t = 2, s = 2$).
4. Correct any 3 errors ($t = 3, s = 0$).

Since there are q^k codewords, obtaining all the nonzero codewords and finding their minimum weight could be tedious. However there is a simpler way to determine the distance and that is from the $\mathbf{H}$ matrix. This method is based upon the following theorems [6].

Theorem 2.3. *If there is a codeword of weight* d *in a linear code* $\mathbf{C}$*, then its parity check matrix has* d *columns that are linearly dependent.*

Proof. Let $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1}) \in \mathbf{C}$ and $\mathbf{w}(c) = d$. This means there are d symbols in $\mathbf{c}$ that are nonzero and $n-k$ symbols that are 0's. We know that $\mathbf{c} \cdot \mathbf{H}^T = 0$ so

$$\begin{aligned} \mathbf{c} \cdot \mathbf{H}^T &= (c_0, c_1, \ldots, c_{n-1}) \cdot (\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_{n-1}) \\ &= c_0 \cdot \mathbf{h}_0 + c_1 \cdot \mathbf{h}_1 + \cdots + c_{n-1} \cdot \mathbf{h}_{n-1}, \end{aligned}$$

where $\mathbf{h}_i$ is the i^{th} column vector of $\mathbf{H}$. Since there are only d entries in $\mathbf{c}$ that are nonzero the d columns of $\mathbf{H}$ are linearly dependent. $\square$

Theorem 2.4. *A linear code* $\mathbf{C}$ *has a distance at least* d *if and only if every* $d-1$ *or fewer columns of its* $\mathbf{H}$ *matrix are linearly independent.*

Proof. The first part of the proof follows immediately from theorem 2.3. Meaning that a codeword of weight d implies that d columns of $\mathbf{H}$ are dependent.

Now suppose we have d columns $\mathbf{h}_{i_0}, \mathbf{h}_{i_1}, \ldots, \mathbf{h}_{i_{d-1}}$ of $\mathbf{H}$ such that

$$\mathbf{h}_{i_0} + \mathbf{h}_{i_1} + \cdots + \mathbf{h}_{i_{d-1}} = 0$$

Now if we form an n-tuple $\mathbf{v} = (v_0, v_1, \ldots, v_{n-1})$ with nonzero components $v_{i_0}, v_{i_1}, \ldots v_{i_{d-1}}$, it's Hamming weight is d. Now consider the product

$$\begin{aligned} \mathbf{v} \cdot \mathbf{H}^T &= v_0 \cdot \mathbf{h}_{i_0} + v_1 \cdot \mathbf{h}_{i_1} + \cdots + v_{n-1} \cdot \mathbf{h}_{i_{n-1}} \\ &= v_{i_0} \cdot \mathbf{h}_{i_0} + v_{i_1} \cdot \mathbf{h}_{i_1} + \cdots + v_{i_{d-1}} \cdot \mathbf{h}_{i_{d-1}} \\ &= \mathbf{h}_{i_0} + \mathbf{h}_{i_1} + \cdots + \mathbf{h}_{i_{d-1}} \end{aligned}$$

This last sum is equal to zero, so $\mathbf{v}$ is a codeword. □

Two important classes of linear block codes are the Hamming code and Reed-Muller codes. The Hamming codes were discovered by Richard Hamming just a couple of years after Shannon published his famous paper. The Hamming codes have been widely used for error control in digital communication and data storage systems due to their high efficiency rates and decoding simplicity. The Reed-Muller codes form a large class of codes for error correction. They are simple in construction and have good structural properties. To learn more about these code see [6], [7].

3 Cyclic Codes

As mentioned earlier, cyclic codes are important for two main reasons:

1. They have considerable inherent algebraic structure.
2. Encoding and decoding computations can be implemented with shift and feedback registers.

These codes are a subclass of linear block codes [6] and because of the reasons above, we can find practical methods to decode them. The codes are often used in communications systems for detecting errors. In this section we will examine cyclic codes in more detail. We will examine some properties of these codes that help understand how to create a generator and parity check matrix for such a code.

Definition 3.1. *A linear code* $\mathbf{C}$ *is a* cyclic code *if for any codeword (vector)* $\mathbf{c} = (c_0, \ldots, c_{n-1}) \in \mathbf{C}$*, the vector* $\mathbf{c}' = (c_{n-1}, c_0 \ldots, c_{n-2})$ *obtained by right logical shift of its components is also a code vector i.e.,* $c' \in \mathbf{C}$*.*

That is if $\mathbf{C}$ is a $((n, k))$ linear code, then $\mathbf{C}$ is a cyclic code provided that if $(c_0, \ldots, c_{n-1})$ is a codeword then

$$\begin{array}{cccccc} c_0 & c_1 & c_2 & \ldots & c_{n-2} & c_{n-1} \\ c_{n-1} & c_0 & c_1 & \ldots & c_{n-3} & c_{n-2} \\ c_{n-2} & c_{n-1} & c_0 & \ldots & c_{n-4} & c_{n-3} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ c_1 & c_2 & c_3 & \ldots & c_{n-1} & c_0 \end{array}$$

are all codewords.

3.1 Cyclic Codes and Polynomial Math

Polynomial math plays an important role in cycle codes so let's review a bit.

Consider the ring $R_n[x]$= GF$(q)[x]/(x^n - 1)$ of polynomials over GF(q) modulo $x^n - 1$. Every polynomial in this ring has the form:

$$r(x) = r_0 + r_1 * x + \ldots + r_{n-1}x^{n-1} \qquad r_i \in GF(q)$$

Since $x^n \equiv 1 \bmod(x^n - 1)$, we can write $x \cdot x^{n-1} = x^n \equiv 1$ in R_n[x]. So we have

$$\begin{aligned} x \cdot r(x) &= a_0 \cdot x + a_1 \cdot x_2 + \ldots + a_{n-2} \cdot x^{n-1} + a_{n-1} \cdot x^n \\ &= a_{n-1} + a_0 \cdot x + a_1 \cdot x_2 + \ldots + a_{n-2} \cdot x^{n-1} \end{aligned}$$

So multiplying a polynomial $r(x) \in R_n[\mathrm{x}]$ by x^i is equivalent to cyclically shifting the coefficients of $r(x)$ to the right by i places. Recall in GF(2), $-1 = 1$, so $R_n[\mathrm{x}]$ is also called algebra of polynomials modulo $x^n + 1$. In the rest of this section $R_n[\mathrm{x}]$ will represent the polynomials modulo $x^n + 1$ over GF(2).

Now let's relate a code vector $\mathbf{c} = (c_0 c_1 c_2 \ldots c_{n-1})$ to a *code polynomial*:

$$\mathbf{c}(x) = c_0 + c_1 \cdot x + c2 \cdot c^2 + \ldots + c_{n-1} \cdot x^{n-1}$$

where the exponent powers correspond to the bit positions and the coefficients are 0's and 1's.

Example 3.1. *Here are two codewords and their polynomial representation.*

$$c = (1001) \textit{ becomes } 1 + x^3$$
$$c = (010101) \textit{ becomes } x + x^3 + x^5$$

Each codeword is represented by a polynomial of degree less than or equal to $n - 1$.

Why do we want to represent a code vector with a polynomial? Well, the following will help answer this. If we let

$$\begin{aligned}
\mathbf{c} &= (c_0 c_1 c_2 \ldots c_{n-1}) \\
\mathbf{c}^{(i)} &= (c_{n-i} c_{n-i+1} \ldots c_{n-1} c_0 \ldots c_{n-i-1}) \\
\text{then} & \\
\mathbf{c}(x) &= c_0 + c_1 \cdot x + c2 \cdot x^2 + \cdots + c_{n-1} \cdot x^{n-1} \\
\mathbf{c}^{(i)}(x) &= c_{n-i} + c_{n-1+1} \cdot x + \cdots + c_{n-1} \cdot x^{i-1} + c_0 \cdot x^i + \cdots + c_{n-i-1} \cdot x^{n-1}
\end{aligned}$$

What is the relation between $\mathbf{c}(\mathrm{x})$ and $\mathbf{c}^{(i)}(\mathrm{x})$? Just like we saw above in our polynomial math in $\mathbf{R}_n[\mathrm{x}]$ a shift by one place to the right is equivalent to multiplying by x.

$$x^i \cdot \mathbf{c}(x) = c_0 \cdot x^i + c_1 \cdot x^{i+1} + \cdots + c_{n-i-1} \cdot x^{n-1} + c_{n-i} \cdot x^n + \cdots + c_{n-1} \cdot x^{n-i-1}$$

The last i terms have powers $\geq n$ and can't be represented by bit locations so let's pull a trick out of our hat add a zero-value sum to this polynomial. Let the zero-value sum be:

$$(c_{n-i} + c_{n-i}) + (c_{n-i+1} + c_{n-i+1}) \cdot x + \cdots + (c_{n-1} + c_{n-1}) \cdot x^{i-1}$$

and when we add it to $x^i \cdot \mathbf{c}(x)$ and arrange the terms a little differently we get:

$$\begin{aligned} x^i \cdot \mathbf{c}(x) &= c_{n-i} \cdot (x^n + 1) + c_{n-i+1} \cdot x \cdot (x^n + 1) + \cdots + c_{n-1} \cdot x^{i-1}(x^n + 1) \\ &+ c_{n-i} + c_{n-i+1} \cdot x + \cdots + c_{n-1} \cdot x^{i-1} \\ &+ c_0 \cdot x^i + c_1 \cdot x^{i+1} + \cdots + c_{n-i-1} \cdot x^{n-1} \end{aligned}$$

This can reduce to :

$$x^i \cdot \mathbf{c}(x) = \mathbf{q}(x) \cdot (x^n + 1) + \mathbf{c}^{(i)}(x)$$

with $\mathbf{c}^{(i)}(x)$ being the remainder from dividing $x^i \cdot \mathbf{c}(x)$ by $(x^n + 1)$. In other words, $\mathbf{c}^{(i)}(x) = x^i \cdot \mathbf{c}(x) \bmod (x^n + 1)$ which says that multiplying a code-word $\mathbf{c}(x)$ by x^k creates a new code word by cyclically shifting $\mathbf{c}(x)$ right by k places.

The above analysis brings us to the following theorem.

Theorem 3.1. *A subset* $\mathbf{C}$ *of polynomials in* $R_n[x]$ *is a cyclic code if and only if it is an ideal.*

Proof. Suppose $\mathbf{C}$ is a cyclic code in $\mathbf{R}_n[x]$. By definition $\mathbf{C}$ is a linear code and therefore a subspace and it's an additive subgroup of $\mathbf{R}_n[x]$. We also know for any code polynomial $\mathbf{c}(x) \in \mathbf{C}$, $x \cdot \mathbf{c}(x)$ is also a code polynomial. Since $x \cdot \mathbf{c}(x) \in \mathbf{C}, x^2 \cdot \mathbf{c}(x)$ is also in $\mathbf{C}$. If we use the fact that $\mathbf{C}$ is linear then any linear combination of these code polynomials is a code polynomial. So this implies that for all $\mathbf{c}(x) \in \mathbf{C}$ and all $\mathbf{r}$(x) in $\mathbf{R}_n[x]$.

$$(r_0 + r_1 \cdot x + \ldots) \cdot \mathbf{c}(x) = \mathbf{r}(x) \cdot \mathbf{c}(x) \in \mathbf{C}.$$

So by definition $\mathbf{C}$ is an ideal. We could also prove this by using the ideal test as given in [3].

Conversely if $\mathbf{C}$ is an ideal $R_n[x]$ then by definition it is a subgroup and for all $\mathbf{c}$(x) in $\mathbf{C}$,

$$\mathbf{r}(x) \in \mathbf{R}_n[x], \mathbf{r}(x) \cdot \mathbf{c}(x) \in \mathbf{C}$$

By setting $\mathbf{r}(x) = a \in GF(q)$ and using the fact $\mathbf{C}$ is a subgroup we see that $\mathbf{C}$ is linear code. Also if we set $\mathbf{r}(x) = x$, we get

$$x \cdot \mathbf{c}(x) = \mathbf{c}'(x) \in \mathbf{C}$$

So $\mathbf{C}$ is a cyclic code. □

Recall that $R_n[x]$ is a principal ideal domain so all the properties of PIDs apply here.

Since $\mathbf{C}$ is ideal in $R_n[x]$ we know there exists a polynomial $\mathbf{g}(x)$ that generates this ideal (or cyclic code). We also know that this $\mathbf{g}(x)$ is the smallest degree monic polynomial in $\mathbf{C}$.

3.2 Generator Polynomial

Let's take a closer look at this generator polynomial and see what properties it has. These properties will aid us in finding a generator and parity matrices for cyclic codes.

Theorem 3.2. *The nonzero code polynomial of minimum degree in a cyclic code is unique.*

Proof. Let $\mathbf{g}(x) = g_0 + g_1 \cdot x + \cdots + g_r \cdot x^{r-1} + x^r$ in $\mathbf{C}$ and suppose it's not unique. Then there exists another $\mathbf{g}'(x) = g'_0 + g'_1 \cdot x + \cdots + g'_r \cdot x^{r-1} + x^r$ of the same degree in $\mathbf{C}$. Since both are codewords which means $\mathbf{g}(x) + \mathbf{g}'(x)$ is a codeword, which means

$$\mathbf{g}(x) + \mathbf{g}'(x) = (g_0 + g'_0) + (g_1 + g'_1) \cdot x + \cdots + (g_{r-1} + g'_{r-1}) \cdot x^{r-1}$$

If $\mathbf{g}(x) + \mathbf{g}'(x) \neq 0$ then we have a contradiction since the degree of $\mathbf{g}(x) + \mathbf{g}'(x)$ is less than r. So $\mathbf{g}(x) + \mathbf{g}'(x) = 0$ so $\mathbf{g}(x) = \mathbf{g}'(x)$. □

Theorem 3.3. *Let $\mathbf{g}(x) = g_0 + g_1 \cdot x + \cdots + g_r \cdot x^{r-1} + x^r$ be the nonzero code polynomial of minimum degree that generates the (n, k) cyclic code $\mathbf{C}$, then g_0 is equal to 1.*

Proof. If not then one shift to the left (this is equal to $n-1$ shifts to the right) of $\mathbf{g}(x)$ will produce another code polynomial but of smaller degree, a contradiction. □

Since $\mathbf{C}$ is cyclic we know that $x \cdot \mathbf{g}(x), x^2 \cdot \mathbf{g}(x), \ldots, x^{n-r-1} \cdot \mathbf{g}(x)$ are also code polynomials in $\mathbf{C}$. We also know that the degree of $x^{n-r-1} \cdot \mathbf{g}(x)$ is $n-1$. With $\mathbf{C}$ being linear, we can form linear combinations of $\mathbf{g}(x), x \cdot \mathbf{g}(x), x^2 \cdot \mathbf{g}(x), \ldots, x^{n-r-1} \cdot \mathbf{g}(x)$

$$\begin{aligned} \mathbf{c}(x) &= u_0 \cdot \mathbf{g}(x) + u_1 \cdot x \cdot \mathbf{g}(x) + \cdots + u_{n-r-1} \cdot x^{n-r-1} \cdot \mathbf{g}(x) \\ &= (u_0 + u_1 \cdot x + \cdots + u_{n-r-1} \cdot x^{n-r-1}) \cdot \mathbf{g}(x) \end{aligned}$$

is also a code polynomial with the coefficients u_i equal to 0 or 1. This brings us to the next theorem.

Theorem 3.4. *A binary polynomial of degree $n-1$ or less is a code polynomial if and only if it is a multiple of $\mathbf{g}(x)$.*

Proof. We have already shown the first part of the proof. For the second part, if a code polynomial $\mathbf{c}(x)$ is not a multiple of $\mathbf{g}(x)$ then the remainder of $\mathbf{c}(x)$ divided by $\mathbf{g}(x)$ is a code polynomial of degree less then $\mathbf{g}(x)$ and this is a contradiction. □

Since the degree of $\mathbf{g}(x)$ is r and we know the largest degree of a code polynomial $\mathbf{c}$(x) is $n-1$. This means that since $\mathbf{c}(x) = \mathbf{r}(x) \cdot \mathbf{g}(x)$ then the degree of $\mathbf{r}$(x) can be taken to be less than or equal to $n-r-1$. This means that there are 2^{n-r} ways of forming $\mathbf{r}$(x). We also know that an (n,k) code $\mathbf{C}$ has 2^k codewords so $n-r=k$ or $r=n-k$.

We can summarize all the results above in the following theorem.

Theorem 3.5. *If $\mathbf{C}$ is an (n,k) cyclic code, then there exists one and only one code polynomial of degree $n-k$,*

$$\mathbf{g}(x) = 1 + g_0 \cdot x + g_2 \cdot x^2 + \cdots + g_{n-k-1} \cdot x^{n-k-1} + x^{n-k}$$

Every code polynomial is a multiple of $\mathbf{g}(x)$ and every binary polynomial of degree $n-1$ or less that is a multiple of $\mathbf{g}(x)$ is a codeword.

It follows from theorem 3.5 that $\mathbf{g}(x)$, *generator polynomial*, defines the cycle code. In other words, every code polynomial $\mathbf{c}(x)$ in an (n,k) cyclic code is a multiple of $\mathbf{g}(x)$.

Theorem 3.6. *The generator polynomial $\mathbf{g}(x)$ of an (n,k) cyclic code is a factor of x^n+1.*

Proof. If we multiply $\mathbf{g}(x)$ by x^k we get a polynomial of degree n. Dividing by $x^n + 1$ gives

$$x^k\mathbf{g}(x) = (x^n + 1) + \mathbf{r}(x)$$

where the degree of the remainder $\mathbf{r}(x)$ is either less than n or is equal to zero.

From our discussion after Example 3.1, we know that $\mathbf{r}(x)$ is a code polynomial. Specifically, $\mathbf{r}(x) = \mathbf{g}^k(x)$. So $\mathbf{r}(x) = \mathbf{a}(x)\mathbf{g}(x)$ for some polynomial $\mathbf{a}(x)$. Therefore

$$x^n + 1 = x^k\mathbf{g}(x) + \mathbf{a}(x)\mathbf{g}(x) = (x^k + \mathbf{a}(x))\mathbf{g}(x)$$

which shows that $\mathbf{g}(x)$ is a factor of $x^n + 1$, as claimed. □

The following theorem (see [6] , [7] for a proof), tells us when we can generate a cyclic code from $\mathbf{g}$(x).

Theorem 3.7. *If* $\mathbf{g}(x)$ *is polynomial of degree* $n-k$ *and is a factor of* x^n+1 *then* $\mathbf{g}(x)$ *generates an (n,k) cyclic code.*

An example will put all of this into perspective.

Example 3.2. *Suppose we wanted to construct a cyclic code of length 7. We know that* $x^7 + 1 = (1 + x) \cdot (1 + x + x^3) \cdot (1 + x^2 + x^3)$*, so we can pick a factor (or product of factors) of degree* $n - k$ *as the generator polynomial of an (n,k) cyclic code. Lets pick* $\mathbf{g}(x)$*=* $1 + x + x^3$ *for a (7,4) code.*

Message $(m_0m_1m_2m_3)$	**Code**	**Polynomial**	**Codeword** $(c_0c_1c_2\ldots c_{n-1})$
0000	$0 \cdot \mathbf{g}(x)$	$= 0$	*0000000*
1000	$1 \cdot \mathbf{g}(x)$	$= 1 + x + x^3$	*1101000*
0100	$x \cdot \mathbf{g}(x)$	$= x + x^2 + x^4$	*0110100*
1100	$(1 + x) \cdot \mathbf{g}(x)$	$= 1 + x^2 + x^3 + x^4$	*1011100*
0010	$x^2 \cdot \mathbf{g}(x)$	$= x^2 + x^3 + x^5$	*0011010*
1010	$(1 + x^2) \cdot \mathbf{g}(x)$	$= 1 + x + x^2 + x^5$	*1110010*
0110	$(x + x^2) \cdot \mathbf{g}(x)$	$= x + x^3 + x^4 + x^5$	*0101110*
1110	$(1 + x + x^2) \cdot \mathbf{g}(x)$	$= 1 + x^4 + x^5$	*1000110*
0001	$x^3 \cdot \mathbf{g}(x)$	$= x^3 + x^4 + x^6$	*0001101*
1001	$(1 + x^3) \cdot \mathbf{g}(x)$	$= 1 + x + x^4 + x^6$	*1100101*
0101	$(x + x^3) \cdot \mathbf{g}(x)$	$= x + x^2 + x^3 + x^6$	*0111001*
1101	$(1 + x + x^3) \cdot \mathbf{g}(x)$	$= 1 + x^2 + x^6$	*1010001*
0011	$(x^2 + x^3) \cdot \mathbf{g}(x)$	$= x^2 + x^4 + x^5 + x^6$	*0010111*
1011	$(1 + x^2 + x^3) \cdot \mathbf{g}(x)$	$= 1 + x + x^2 + x^3 + x^4 + x^5 + x^6$	*1111111*
0111	$(x + x^2 + x^3) \cdot \mathbf{g}(x)$	$= x + x^5 + x^6$	*0100011*
1111	$(1 + x + x^2 + x^3) \cdot \mathbf{g}(x)$	$= 1 + x^3 + x^5 + x^6$	*1001011*

Example 3.2 showed us just one $\mathbf{g}(x)$. There are several we could have chosen.

$\mathbf{g}(x)$	**Code**	$\mathbf{g}(x)$	**Code**
$(1 + x)$	(7,6)	$(1 + x) \cdot (1 + x + x^3)$	(7,3)
$(1 + x + x^3)$	(7,4)	$(1 + x) \cdot (1 + x^2 + x^3)$	(7,3)
$(1 + x + x^2 + x^3)$	(7,4)	$(1 + x + x^3) \cdot (1 + x + +x^3)$	(7,1)

We need to keep in mind that even though we were able to generate a cyclic code by choosing a $\mathbf{g}(x)$, we know nothing about its minimum distance. So, this code could be good or it could be bad. How do we select a good generator? Well that's a hard question to answer and beyond the scope of this paper.

Since we now know how we can find a generating polynomial $\mathbf{g}(x)$ for a cyclic code, let's see how we can create the generator and parity-check matrices like we did for the general linear block codes.

3.3 Generator and Parity-Check Matrices

We know from our excursion above that given an (n, k) code $\mathbf{C}$ and a generator polynomial $\mathbf{g}(x) = g_0 + g_1 \cdot x + \cdots + g_r \cdot x^r$ that the dimension of $\mathbf{C}$ is $n - r$. We also know that $\mathbf{g}(x), x \cdot \mathbf{g}(x), x^2 \cdot \mathbf{g}(x), \ldots, x^{n-r-1} \cdot \mathbf{g}(x)$ are codewords. So if we combine this into a matrix (keep in mind that $r = n - k$),

$$\mathbf{G} = \begin{bmatrix} g_0 & g_1 & g_2 & \ldots & g_r & 0 & 0 & \ldots & 0 \\ 0 & g_0 & g_1 & g_2 & \ldots & g_r & 0 & \ldots & 0 \\ 0 & 0 & g_0 & g_1 & g_2 & \ldots & g_r & \vdots & 0 \\ \vdots & \vdots & & & & & & & 0 \\ 0 & 0 & \ldots & 0 & g_0 & g_1 & g_2 & \ldots & g_r \end{bmatrix} \tag{3.1}$$

We also know that the rows of $\mathbf{G}$ are linearly independent since the matrix is in echelon form [8], [10].

We know a couple of things that can help us find the parity-check matrix. First given a $\mathbf{g}(x)$ we know it is a factor $x^n + 1$, say

$$x^n + 1 = \mathbf{g}(x) \cdot \mathbf{h}(x)$$

and the degree of $\mathbf{h}(x)$ is k,

$$\mathbf{h}(x) = h_0 + h_1 \cdot x + \cdots + h_{k-1} \cdot x^{k-1} + h_k \cdot x^k$$

We also know that codewords $\mathbf{c}(\mathrm{x})$ are exactly multiples of $\mathbf{g}(\mathrm{x})$ or $\mathbf{c}(x) = \mathbf{m}(x) \cdot \mathbf{g}(x)$ where $\mathbf{m}(\mathrm{x})$ is the message and has the form of $\mathbf{m}(x) = m_0 + m_1 \cdot x + \cdots + m_{k-1} \cdot x^{k-1}$. So

$$\mathbf{c}(x) \cdot \mathbf{h}(x) = \mathbf{m}(x) \cdot \mathbf{g}(x) \cdot \mathbf{h}(x) = \mathbf{m}(x)(x^n + 1) = \mathbf{m}(x) + \mathbf{m}(x) \cdot x^n$$

Since the degree of $\mathbf{m}(\mathrm{x})$ is $k - 1$ or less then $x^k, x^{k+1}, ..., x^{n-1}$ terms will not appear in this $\mathbf{m}(x) + \mathbf{m}(x) \cdot x^n$ sum. Thus the coefficients of $x^k, x^{k+1}, ..., x^{n-1}$ in the product of $\mathbf{c}(x) \cdot \mathbf{h}(x)$ must be equal to zero (do a simple example here and you will see what's below is correct),

$$\begin{array}{ccccccccc}
c_0 \cdot h_k & + & c_1 \cdot h_{k-1} & + & \cdots & + & c_k \cdot h_0 & = 0 \\
c_1 \cdot h_k & + & c_2 \cdot h_{k-1} & + & \cdots & + & c_{k+1} \cdot h_0 & = 0 \\
\quad \ddots & & & & & & \ddots & \vdots \\
 & & c_{n-k-1} \cdot h_k & + & \cdots & + & c_{n-1} \cdot h_0 & = 0
\end{array}$$

And this can be written as

$$\sum_{i=0}^{k} h_i \cdot c_{l-i} = 0 \qquad \text{for} \qquad l = (k, k+1, \ldots, n-1)$$

Thus we have any codeword $(c_0 c_1 \ldots c_{n-1})$ is orthogonal to $(h_k h_{k-1} \ldots h_0 00 \ldots 0)$ (dimension $= n$ so $n-k$ zeros).

And we can finally express this as

$$\begin{bmatrix}
h_k & h_{k-1} & h_{k-2} & \ldots & h_0 & 0 & 0 & \ldots & 0 \\
0 & h_k & h_{k-1} & h_{k-2} & \ldots & h_0 & 0 & \ldots & 0 \\
0 & 0 & h_k & h_{k-1} & h_{k-2} & \ldots & h_0 & \ldots & 0 \\
0 & 0 & 0 & h_k & h_{k-1} & h_{k-2} & \ldots & h_0 & 0 \\
 & & & \ddots & & & \ddots & & \\
0 & 0 & \ldots & 0 & h_k & h_{k-1} & h_{k-2} & \ldots & h_0
\end{bmatrix}
\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-2} \\ c_{n-1} \end{bmatrix}
= \begin{bmatrix} \mathbf{0} \end{bmatrix}$$

so our parity-check matrix is

$$\mathbf{H} = \begin{bmatrix}
h_k & h_{k-1} & h_{k-2} & \ldots & h_0 & 0 & 0 & \ldots & 0 \\
0 & h_k & h_{k-1} & h_{k-2} & \ldots & h_0 & 0 & \ldots & 0 \\
0 & 0 & h_k & h_{k-1} & h_{k-2} & \ldots & h_0 & \ldots & 0 \\
0 & 0 & 0 & h_k & h_{k-1} & h_{k-2} & \ldots & h_0 & 0 \\
 & & & \ddots & & & \ddots & & \\
0 & 0 & \ldots & 0 & h_k & h_{k-1} & h_{k-2} & \ldots & h_0
\end{bmatrix}$$

Is this parity-check matrix $\mathbf{H}$ cyclic? The answer is yes by Theorem 3.6 if we can show $\mathbf{h}(x)$ is a factor of $x^n + 1$. So here's how we do this. Let

$$\overline{\mathbf{h}}(x) = h_k + h_{k-1} \cdot x + \cdots + h_1 \cdot x^{k-1} + h_0 \cdot x^k$$

and then notice

$$\overline{\mathbf{h}}(x) = x^k \cdot \mathbf{h}(x^{-1})$$

And if we manipulate these equations

$$\begin{aligned}
\mathbf{h}(x^{-1}) \cdot \mathbf{g}(x^{-1}) &= (x^{-1})^n + 1 \\
x^k \cdot \mathbf{h}(x^{-1}) \cdot x^{n-k} \cdot \mathbf{g}(x^{-1}) &= x^n \cdot (x^{-n} - 1) \\
\overline{\mathbf{h}}(x) \cdot x^{n-k} \cdot \mathbf{g}(x^{-1}) &= 1 - x^n \\
&= x^n + 1 \text{ since } +1, -1 \text{ are the same in GF(2)}
\end{aligned}$$

So we have that $\overline{\mathbf{h}}(x)$ is indeed a factor of $x^n - 1$ and therefore the code generated by $\mathbf{H}$ is cyclic.

Notice that the $\mathbf{G}$ and $\mathbf{H}$ matrices of cyclic codes are normally not in systematic form so the codewords obtained from

$$\mathbf{c}(x) = \mathbf{d}(x) \cdot \mathbf{g}(x)$$

with data $\mathbf{d}(x)$ are non systematic as example 3.2 shows. The generator matrix for this code in the form of 3.1 is

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

and we can use row reductions methods to create a systematic form

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

However, the following method of encoding messages into a systematic form will aid us in our understanding of the next section when we see how to implement hardware encoders and decoders.

If we have a message $\mathbf{m}$ and multiply its polynomial representation by x^{n-k} we end up with

$$x^{n-k} \cdot \mathbf{m}(x) = m_0 \cdot x^{n-k} + m_1 \cdot x^{n-k+1} + \cdots + m_{k-1} \cdot x^{n-1}$$

(basically just shift $\mathbf{m}$ right by $n-k$ places) which in vector form is

$$(\underbrace{0,0,0,\ldots,0}_{n-k},m_0,m_1,\ldots,m_{k-1})$$

If we divide $x^{n-k}\cdot\mathbf{m}(x)$ by $\mathbf{g}$(x) we end up with a quotient and remainder, $x^{n-k}\cdot\mathbf{m}(x)=\mathbf{q}(x)\mathbf{g}(x)+\mathbf{d}(x)$. And by subtracting the remainder we see that $x^{n-k}\cdot\mathbf{m}(x)-\mathbf{d}(x)$ is a codeword i.e.

$$x^{n-k}\cdot\mathbf{m}(x)-\mathbf{d}(x)=\mathbf{q}(x)\cdot\mathbf{g}(x)$$

$x^{n-k}\cdot\mathbf{m}(x)-\mathbf{d}(x)$ is equal to $(-d_0,-d_1,\ldots,d_{n-k-1},m_0,m_1,\ldots,m_{k-1}$ in vector form.

In a similar manner we can create a systematic form for the generator matrix simply by

$$x^{n-k+i}\cdot\mathbf{g}(x)-\mathbf{b}_i(x)=\mathbf{q}(x)\cdot\mathbf{g}(x)$$

with $i=0,1,\ldots,k-1$. So

$$\mathbf{G}=\begin{bmatrix}-b_{0,0} & -b_{0,1} & \ldots & -b_{0,n-k-1} & 1 & 0 & 0 & \ldots & 0\\ -b_{1,0} & -b_{1,1} & \ldots & -b_{1,n-k-1} & 0 & 1 & 0 & \ldots & 0\\ -b_{2,0} & -b_{2,1} & \ldots & -b_{2,n-k-1} & 0 & 0 & 1 & \ldots & 0\\ \vdots & & \vdots & & & & & \vdots & \\ -b_{k-1,0} & -b_{0,1} & \ldots & -b_{k-1,n-k-1} & 0 & 0 & 0 & \ldots & 1\end{bmatrix}$$

Example 3.3. *Consider the code generated by* $\mathbf{g}(x)=1+x+x^3$,

$$\begin{array}{llll} i=0: & x^3 & =\mathbf{g}(x)+(1+x) & \mathbf{b}_0 & =1+x\\ i=1: & x^4 & =x\mathbf{g}(x)+(x+x^2) & \mathbf{b}_1 & =x+x^2\\ i=2: & x^5 & =(1+x^2))\mathbf{g}(x)+(1+x+x^2) & \mathbf{b}_2 & =1+x+x^2\\ i=3: & x^6 & =(1+x+x^3)\mathbf{g}(x)+(1+x^2) & \mathbf{b}_3 & =1+x^2\end{array}$$

and the generator and parity matrices are

$$\mathbf{G}=\begin{bmatrix}1&1&0&1&0&0&0\\0&1&1&0&1&0&0\\1&1&1&0&0&1&0\\1&0&1&0&0&0&1\end{bmatrix}\qquad\mathbf{H}=\begin{bmatrix}1&0&0&1&0&1&0\\0&1&0&1&1&1&0\\0&0&1&0&1&1&1\end{bmatrix}$$

We can clearly see that the row reduction systematic form (3.2) is the same as the above $\mathbf{G}$.

4 Hardware Implementation

We've seen that the memory requirements for linear block codes can make them inefficient for decoding. One of the justifications in using cyclic codes with polynomial representation is that they can be implemented in hardware very effectively. This is where some of the beauty of cyclic codes come through. Let's see how this is done [6], [7].

4.1 Polynomial Multiplication and Division

We start with some simple components such as:

1.  D flip flop. This a simple storage element that will hold its content (in our case a 0, 1 in GF(2)) until a new value is clocked into it. We don't show the clock signal just to keep things simple. We can create a shift register by connecting several of these together in a serial fashion.

2.  An adder. This has two inputs and one output which is the sum of the inputs.

3.  A multiplier. Multiply the input by the $\mathbf{g_i}$ to create the output.

If we want to multiply two polynomials together say $a(x) = a_0 + a_1 \cdot x + \ldots + a_k \cdot x^k$ and $g(x) = g_0 + g_1 \cdot x + \cdots + g_r \cdot x^r$ we get a product of

$$\begin{aligned} b(x) &= a(x) \cdot g(x) \\ &= a_0 \cdot g_0 + (a_0 \cdot g_1 + a_1 \cdot g_0) \cdot x \\ &\cdots + (a_k \cdot g_{r-1} + a_{k-1} \cdot g_r) \cdot x^{r+k-1} + a_k \cdot g_r \cdot x^{r+k} \end{aligned}$$

and it can be represented by the circuit in figure 3. The operation is straight forward. First the registers are cleared. The most significant bit (msb) or symbol, a_k is input first. The first output is $a_k \cdot g_r$ which is the last term in product above. Since all the other registers are 0 they contribute nothing to the sum. The next step is to clock the registers so a_k loaded into the first register and a_{k-1} is presented at the input. The output is now $a_{k-1} \cdot g_r + a_k \cdot g_{r-1}$. This process is continued until a_0 is clocked in , then the system is clocked r times more to produce all the terms ($k + r - 1$ of them).

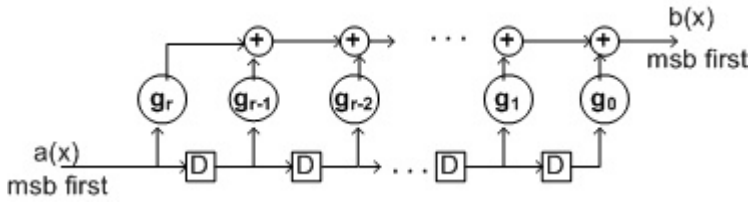


Figure 3: A polynomial multiplication circuit

Suppose on the other hand we were dividing two polynomials

$$d(x)/g(x)$$

where

$$d(x) = d_0 + d_1 \cdot x + \cdots + d_n \cdot x^n$$

and the divisor

$$g(x) = g_0 + g_1 \cdot x + \cdots + g_p \cdot x^p$$

Figure 4 shows how this can be done in hardware. For binary polynomials the coefficient $-g_p^{-1}$ has a value of 1. Some other things we know are, the remainder must be of degree $\leq p - 1$ since the divisor has degree p so,

$$r(x) = r_0 + r_1 \cdot x + \cdots + r_{p-1} \cdot x^{p-1}$$

The quotient

$$q(x) = q_0 + q_1 \cdot x + \cdots + q_{n-p} \cdot x^{n-p}$$

The division circuit of figure 4 operates as follows:

1. All the memory devices (registers) are cleared to 0.

2. The coefficients of $d(x)$ are clocked into the left registers for p steps starting with d_n. This step initializes the registers.

3. The coefficients of $d(x)$ continue to be clocked in on the left. The bits shifted to the right represent the coefficients of the quotient $d(x)/g(x)$ with the highest-order coefficient first.

4. After all the coefficients of $d(x)$ have been shifted in the contents of the register elements are the remainder of the division, with the highest coefficient on the right.

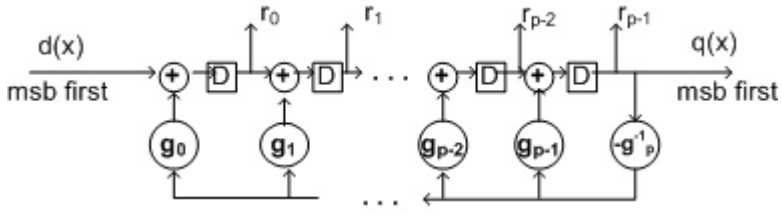


Figure 4: A polynomial divsion circuit

The following example will help illustrate how division is accomplished with a circuit like that in 4.

Example 4.1. *Let's divide* $d(x) = x^8 + x^7 + x^5 + x + 1$ *by* $g(x) = x^5 + x + 1$, *with the polynomial long division shown below (keep in mind we are over GF(2) so* $+1 = -1$*). The division circuit for g(x) is in figure 5.*

$$
\begin{array}{rl}
 & \quad x^3+\quad x^2+\qquad\qquad\quad 1 \\
x^5+x+1 \;\big)\; & \overline{x^8+\;x^7+\quad x^5{}_{\mathbf{A}}+\qquad\qquad\qquad\quad x+\;1} \\
 & \quad x^8+\qquad\qquad\quad x^4+\quad x^3 \\
 & \quad\overline{\qquad x^7+\quad x^5+\;x^4+\;x^3{}_{\mathbf{B}}+\qquad\quad x+\quad 1} \\
 & \qquad\quad x^7+\qquad\qquad\quad x^3+\qquad x^2 \\
 & \qquad\quad\overline{\qquad\qquad x^5+\;x^4+\qquad x^2{}_{\mathbf{C}}+\;x+\quad 1} \\
 & \qquad\qquad\quad x^5+\qquad\qquad\qquad\quad x+\quad 1 \\
 & \qquad\qquad\quad\overline{\qquad\qquad\quad x^4+\qquad\quad x^2{}_{\mathbf{D}}}
\end{array}
$$

Initially the contents of the registers are zeroed out. Then after 5 shifts, the first 5 terms of d(x) will be loaded into the registers with x^8 *on the right. The corresponding entry in the long division is the line with* $_{\mathbf{A}}$ *in the line with*

most significant terms are on the left. On the next shift g(x) is subtracted (or added) to create the value in line **B**. *This process continues until all the bits of d(x) have been shifted in as shown in* **C** *and* **D**. *The remainder of* $d(x)/g(x)$ *is contained in the registers.*

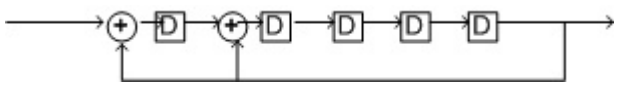


Figure 5: A polynomial divsion circuit for $g(x) = x^5 + x + 1$

The following table shows how the circuit operates for each shift.

Shift j	*Input bit*	*Polynomial term*	*Register bits*	*polynomial representation*	*Output bit*	*polynomial term*
0	-	-	*0 0 0 0 0*			
1	*1*	(x^8)	*1 0 0 0 0*			
2	*1*	(x^7)	*1 1 0 0 0*			
3	*0*	(x^6)	*0 1 1 0 0*			
4	*1*	(x^5)	*1 0 1 1 0*			
5	*0*	(x^4)	*0 1 0 1 1*	**A**: $x^5 + x^7 + x^8$	*1*	x^3
6	*0*	(x^3)	*1 1 1 0 1*	**B**: $x^3 + x^4 + x^5 + x^7$	*1*	x^2
7	*0*	(x^2)	*1 0 1 1 0*	**C**: $x^2 + x^4 + x^5$	*0*	x^1
8	*1*	(x^1)	*1 1 0 1 1*	$x + x^2 + x^4 + x^5$	*1*	x^0
9	*1*	*1*	*0 0 1 0 1*	**D**: $x^2 + x^4$		

We now have the building blocks to encode and decode messages.

4.2 Cyclic Encoding and Decoding

We learned in section 3.2 of a simple way to create a systematic form for a cyclic code. There were basically three steps:

1. Calculate $x^{n-k} \cdot m(x)$.
2. Divide by $g(x)$ to create remainder $d(x)$.
3. Compute $x^{n-k} \cdot m(x) - d(x)$.

Figure 6 shows a circuit that accomplishes this. The division circuit is very similar to the one in figure 4 except that the message is fed into the right instead of the left to reflect the shift by x^{n-k}. This shifted signal is then divided by the feedback structure of the circuit.

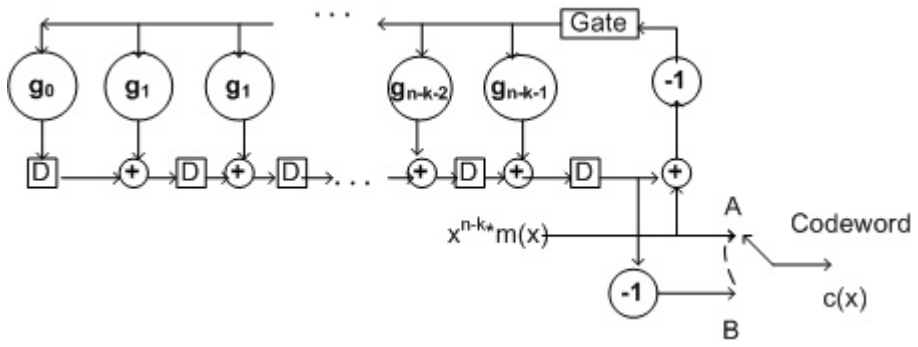


Figure 6: A circuit for systematic encoding using $g(x)$.

The operation of the circuit is as follows:

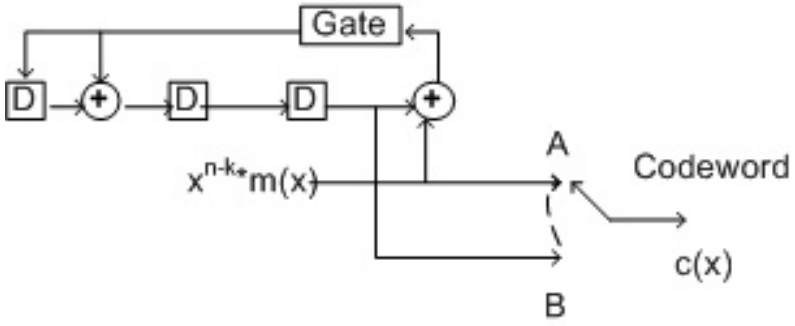
1. With the gate open (meaning data is allow to propagate through) and the switch on position A, the message is fed into the channel msb first (i.e. $m_{k-1}, m_{k-2}, \ldots, m_0$). When all the message bits have been sent, the contents to the $n-k$ registers form the remainder (they are the parity bits).

2. The gate is then closed. This removes any feedback from entering the circuit. The switch is then moved to the B position. Remember we need to subtract the remainder hence the reason for multiplying by -1. However in GF(2) it's not needed since $-1 = 1$.

3. The circuit is then clocked $n-k$ times to shift the parity symbols out into the channel.

Example 4.2. *Let $g(x) = 1 + x + x^3$, the systematic encoder is shown below.*

For a message $\mathbf{m} = (0, 1, 0, 1) \leftrightarrow m(x) = x + x^3$, the contents of the registers are:



Input	*Register contents*	
	0 0 0	*(initial state)*
1	*1 1 0*	
0	*0 1 1*	
1	*1 1 1*	
0	*0 1 1*	*(parity bits, d(x)=$x^2 + x$*

So the output is **c***=(0,1,1,0,1,0,1).*

We've been able to take simple messages and encode them with a few simple basic hardware components. Now the question is, how do we decode these binary cyclic codes and determine if there are any errors? We could use the standard array decoding as we discussed in 2.2 or even the reduced standard array by syndromes. [6] [7]. What we are going to do here is define the syndrome in a slightly different way. We know that a codeword must be a multiple of $g(x)$. If we divide $r(x)$ (the received codeword) by $g(x)$ then the remainder is equal to zero if $r(x)$ is a codeword. If use the division algorithm, we can write

$$r(x) = q(x) \cdot q(x) + s(x)$$

where $s(x)$ is the syndrome (and the remainder) such that

$$s(x) = s_0 + s_1 \cdot x + \cdots + s_{n-k-1} \cdot x_{n-k-1}$$

To compute the syndrome we use polynomial division and a circuit like the one in Figure 4 will work. Remember that a syndrome can be used to detect and even correct errors in received codewords. The circuit below is an example.

The following theorem is an important one. The proof is in [6]. It will aid us greatly in our endeavor to find decoding/correcting circuit.

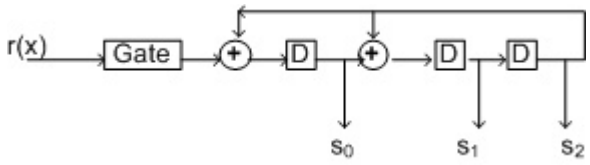


Figure 7: A syndrome generation circuit for $g(x) = 1 + x + x^3$.

Theorem 4.1. *Let* $\mathbf{s}(x)$ *be the syndrome of a received polynomial* $\mathbf{r}(x) = r_0 + r_1 \cdot x + \cdots + r_{n-1} \cdot x^{n-1}$. *Then, the remainder* $\mathbf{s}^1(x)$ *resulting from dividing* $x \cdot \mathbf{s}(x)$ *by the generator polynomial* $\mathbf{g}(x)$ *is the syndrome of* $\mathbf{r}^1(x)$, *which is a cycle shift of* $\mathbf{r}(x)$.

What this means is that $\mathbf{s}^i$(x) that represents the cyclical shift of $\mathbf{r}$(x) by i places to produce $\mathbf{r}^i$(x) is obtained from the remainder of $x^i \cdot \mathbf{s}(x)$ when divided by $\mathbf{g}(x)$. This can be accomplished by circuit in figure 7 by opening the gate to shift $\mathbf{r}(x)$ in (n shifts), then close the gate and shift i more times.

The beauty of this is that if there is an error in a bit in $\mathbf{r}$(x) say r_2, the syndrome $\mathbf{s}$(x) will be non zero. And by the theorem, if we loaded a shifted version of $\mathbf{r}$(x) into the syndrome circuit, the syndrome will also be shifted and non zero. This example will bring this to light along with giving us an idea on how we can used for error correcting.

Example 4.3. *Let's consider the syndrome circuit in Example 7 for the code with the generator* $\mathbf{g}(x)$=$x^3 + x + 1$. *The table below shows single bit error and the syndromes associated with those errors. If the received codeword* $\mathbf{r}_1(x)$ *with an error in bit location 1 (i.e.* $\mathbf{e}_1(x)$ = *0100000) the syndrome would tells us we have an error but not the location. Shifting* $\mathbf{s}_1(x)$ *by 5 places would produce the syndrome for* $\mathbf{r}_1(x)$ *shifted by 5 places or* $x^5 \cdot \mathbf{r}_1(x)$, *with* $x^5 \cdot \mathbf{s}_1(x)$ = *101. Now suppose we get receive* $\mathbf{r}_2(x)$ *with an error in the 4th bit,* $\mathbf{e}_2(x)$ = *0000100, so* $\mathbf{s}_2(x)$ = *011, and we began to shift* $\mathbf{s}_2(x)$. *The first shift would give us* $x \cdot \mathbf{s}_2(x) = 111$, *the second shift* $x^2 \cdot \mathbf{s}_2(x) = 101$. *Notice that the syndrome for the two error patterns matched when we moved the error to the msb location.*

error	*error polynomial*	*syndrome*	*syndrome polynomial*
0000000	$e(x) = 0$	*000*	$x(x) = 0$
1000000	$e(x) = 1$	*100*	$x(x) = 1$
0100000	$e(x) = x$	*010*	$x(x) = x$
0010000	$e(x) = x^2$	*001*	$x(x) = x^2$
0001000	$e(x) = x^3$	*110*	$x(x) = 1 + x$
0000100	$e(x) = x^4$	*011*	$x(x) = x + x^2$
0000010	$e(x) = x^5$	*111*	$x(x) = 1 + x + x^2$
0000001	$e(x) = x^6$	*101*	$x(x) = 1 + x^2$

This example which is made possible by Theorem 4.1 is very important. It not only allows us to compute one syndrome for an error and just create cyclic shifts to cover the other errors, but the syndrome table will only need to store n entries instead of 2^n. And we can use the same hardware to create those syndromes.

This example also shows a way to create error correcting hardware. Figure 8 shows a decoder structure that is known as Meggitt Decoder [6], [7].

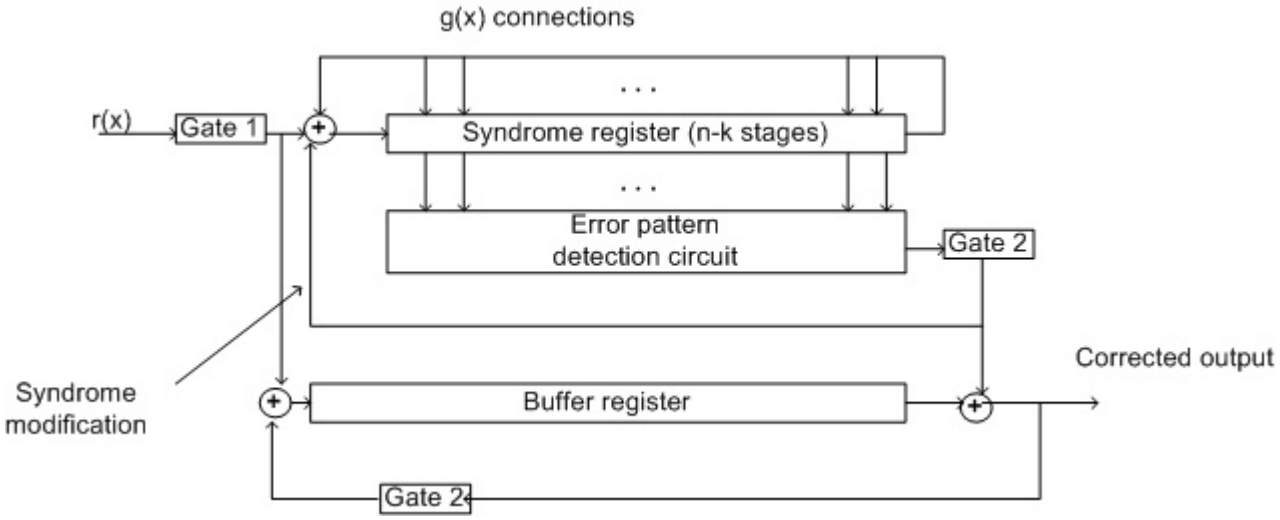


Figure 8: A cyclic decoder when $\mathbf{r}$(x) is shifted in the left end of the syndrome register

Let's examine how this operates.

1. With Gate 1 opened and Gate 2 closed and the syndrome register is cleared, the received vector is shifted into syndrome register and

the buffer register. After n shifts the syndrome register will hold the syndrome for $\mathbf{r}(x)$. Like we describe in Figure 4.3.

2. The error pattern detection circuit will indicate there is an error in the most significant bit (msb) in the received word (i.e. it will output a 1 when an error is detected). So if it outputs a 1, then the msb in the buffer register is in error so it is complemented (1 is added) to correct the error. This correction must also be fed back to the syndrome register so it knows an error is being corrected. Now a cyclical shift is applied and the corrected received bit is loaded into the lsb of the buffer register.

3. The above step will be repeated for each bit in the received word (a total of n times). When done the, the buffer register will hold the corrected received word.

The key to decoding process is generating the correct error pattern detection circuit. As we mentioned in 4.3, the syndrome for the error bit when it was shifted to the last position is the same. So the key is to generate what the syndrome would be if the error was in msb of the received word. If we take $\mathbf{g}(x)$ as in example 4.3 the detection of error pattern is simple as shown in the following figure.

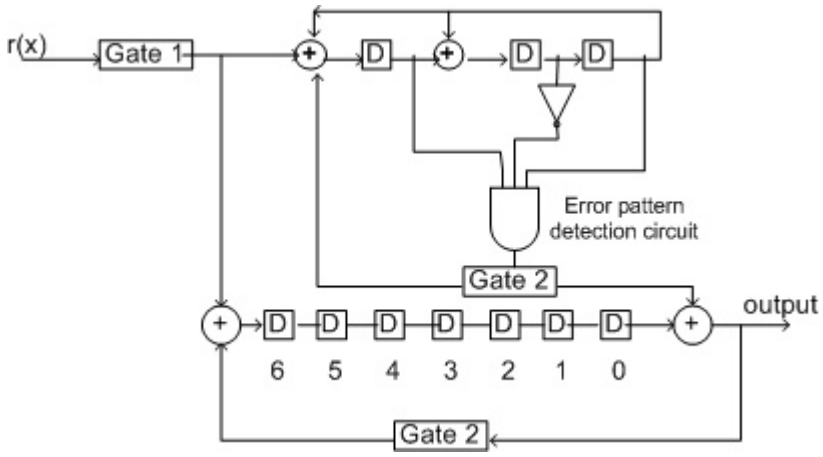


Figure 9: A Meggitt decoder for example 4.3.

We can see the error pattern detection circuit is a simple inverter and a 3-input and gate.

5 Conclusions

Without even knowing it, when we were kids we used error correction and detection when we said our first "What did you say"? Often times we can't ask for data to be re-transmitted so we build some redundancy in the data as it is transmitted that enables us detect and even correct errors in the received data. We've seen that we can use simple linear and abstract algebra concepts to build the idea of a linear block codes. These linear block codes have several characteristics that made it fairly easy to construct, that of generator matrix and parity check matrix. They also have an important property known as the weight of the code. This feature determines how many errors the code can detect or correct. The detection and correction is often done by standard arrays,cosets or syndrome decoding.

We then used our extended knowledge of abstract algebra to construct a subset of linear block codes known as cyclic codes. These codes are based upon a generator polynomial $\mathbf{g}(x)$, which is a factor of $x^n - 1$ for a (n, k) code. These codes are rich in algebraic structure and are easily implemented. Like the basic linear block code detecting and correcting errors in cyclic codes can also be done with syndromes. Error correction capability is inferred from the roots of $\mathbf{g}(\mathrm{x})$ which comes from even more abstract algebra, that of finite fields.

We have shown that syndromes are an easy way to determine if an error occurred in the received codeword as well as correcting some of those errors. This type of detecting and correcting errors works for the general linear block code as well as cyclic codes. The main draw back with using the standard array decoding is that it requires too much memory to be really practical for large codes. Using syndromes can also be inefficient with memory. However if we use cyclic codes, we can use hardware to efficiently generate those syndromes. And with a little extra hardware, we can not only detect errors but correct them as was shown by the simple Meggitt Decoder. Computer systems and networking protocols often employ cyclic codes (known as Cyclic Redundancy Checks, CRC) to detect errors in data packets received. The concept of generating syndromes for CRCs is similar to the single bit at a time that we generated but bytes are used to improve efficiency. The reader is encouraged to see [6], [7] or other literature to see the further explore CRCs or other block codes. We have only scratched the surface!

Acknowledgments

I would like to express my thanks to the math department at Boise State University for having such knowledgeable and capable staff. These last two years have broadened my horizons and helped me deepen my knowledge and appreciation for such a rich subject.
I also want to personally thank Dr. Zach Teitler for his assistance in helping with this paper. His time, guidance and direction are and ever will be greatly appreciated!

References

[1] Axler, Sheldon *Linear Algebra Done Right Second Ed*, **Springer**, 1997

[2] Burton, David M. *The History of Mathematics An Introduction, 7th Ed*, **McGraw-Hill**, 2011

[3] Gallian, Joseph, *Contemporary Abstract Algebra Sixth Ed.*, **Houghton Mifflin Company**, 2006

[4] Hill, Raymond *A First Course in Coding Theory*, **Oxford University Press**, 1986

[5] Hungerford, Thomas W. *Abstract Algebra An Introduction , Second Ed*, **Cengage Learning**, 1997

[6] Lin, Shu; Costello Jr, Daniel *Error Control Coding, Second Ed.*, **Pearson Publications**, 2005

[7] Peterson, W. Wesley; Weldon Jr, E.J. *Error Correcting Codes Second Ed*, **MIT Press**, 1972

[8] Poole, David *Linear Algebra A modern Introduction, 3rd Ed*, **Brooks/Cole**, 2011

[9] Roman, Steven *Advanced Linear Algebra Third Ed*, **Springer**, 2010

[10] Strang, Gilbert *Introduction to Linear Algebra.*, **Wellesley-Cambridge Press**, 2009