# Error Correcting Codes

John McCall
Adviser: Peter Dolan
Second Reader: David Roberts

March 21, 2014

## 1 Abstract

[Coming Soon]

## 2 Introduction

Communication plays a huge role in today's society. Electronic forms of communication become more prevalent every year. Whether it is listening to a CD in the car, reading an article on the internet, or talking with a friend via a cell phone, electronic communication impacts our everyday lives. However, these forms of communication are not always perfect. For instance, that CD could become scratched, corrupting some of its data. Error correcting codes are a way of coping with corruption.

The idea behind error correcting codes is to encode and transmit the data with a sufficient amount of redundancy to be able to reconstruct any data that may become lost or corrupted. This is why that even when a CD gets some minor scratches it will still be able to play perfectly. Again, these codes are not perfect. If a scratch is too large, some of the data will be irrecoverable and there will be a noticeable effect on the music playing.

In section 1 we will provide the necessary abstract algebra background to understand the remainder of the paper. Section 2 will introduce coding theory and give examples of some codes. In section 3 we will go into detail about Generalized Reed-Solomon codes and will walk through an example.

## 3 Background

This section will provide all the necessary background in abstract algebra that is needed to understand Generalized Reed-Solomon codes. Specifically it will define groups, cyclic groups, rings and fields. Introduce the finite field $\mathbb{F}_2$ as well as the extension field $\mathbb{F}_{2^m}$. Then it will describe how addition and multiplication behave in this field.

## 3.1 Groups, Rings, and Fields

A *group* is a set, $G$, paired with a binary operation, usually denoted as $+$ or $\cdot$, and referred to as addition and multiplication. For now we will use the multiplicative notation. A group must have the following properties.

1. $G$ must be closed under the operation. In otherwords, for any $a, b \in G, a \cdot b \in G$ as well.

2. The operation must be associative. So for all $a, b, c \in G$ we have $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.

3. There must be an identity element. This means that there exists an element $e \in G$ such that for all $a \in G, a \cdot e = e \cdot a\ a$. Traditionally, $e$ is written as 0 if we are using $+$ to denote our operation or as 1 if we are using $\cdot$.

4. Identity elements must exist. So for all $a \in G$ there exists an element $a^{-1} \in G$ such that $a \cdot a^{-1} = a^{-1} \cdot a = e$, the identity element.

It is important to note that the operation does not need to be commutative. If it is commutative that is called an *abelian group*. An example of a group would be $\mathbb{Z}$ under addition. It is easy to see that the integers under normal addition satisfies all of the above properties and is a group. Another example would be $\mathbb{Z}^*$, the integers with zero removed, under multiplication. Zero needed to be removed in this case because it has no multiplicative inverse.

An element $a \in G$ is called a *generator* or a *primitive element* of $G$ if $\{a^n | n \in \mathbb{Z}\} = G$. In other words if we take $a$ and multiply it by itself repeatedly, the set of products will be equal to the set $G$. Any group which has a generator is called a *cyclic group*.

A *ring* is a set $R$, paired with two binary operations, usually denoted $+ and \cdot$ and referred to as addition and multiplication, such that the following properties hold.

1. $R$ under addition is an abelian group.

2. Multiplication is associative.

3. The two distributive laws always hold true. In other words, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$.

## 3.2 Addition and Multiplication in $\mathbb{F}_{2^m}$

# 4 Codes

A *code* is a rule for converting information (usually a letter or word) into another representation. Oftentimes, this other representation is of a different form. *Encoding* is the actual act of conversion. A simple code would be to map each letter of the alphabet to a number. For instance, "A" maps to "1", "B" maps to

“2”, etc. In this example, the letters are our *source alphabet*, and the numbers are our *target alphabet*. To encode the word “ALGEBRA” we would simply replace all the letters with the appropriate numbers. In this case, “112752171” is the result. The act of *decoding* reverses the code and returns the word to its original state.

Codes are very useful when storing data, or transmitting it over long distances, as they provide an easy, consistent, way of representing data. However, this data may be corrupted in transmission. *Random errors* are a type of error that corrupts individual symbols during transmissions. Corruption, in this context, means changing the symbol to another symbol in our alphabet, or deleting a symbol all together. Going back to our example, if two random errors occur during transmission, our word may arrive as: “117712171”. A “2” was replaced with a “7” and a “5” was replaced with a “1”. As a result our decoded word is: “ARGABRA”. A *burst error* is when a large chunk of symbols in a row become corrupted. These can be particularly troublesome as they can corrupt a large portion of the data, and are common in several types of communication channels.

Codes that can detect these error are called *error detecting* codes. Codes that can correct these errors are called *error correcting* codes. To reiterate, there is no perfect code which can correct or detect all errors. The ability to correct even a small amount of errors is better than the alternative of having to retransmit the data every time an error occurs.

## 4.1 Cyclic Redundancy Check

An example of an actual code, used in computer hard drives, is a *cyclic redundancy check (CRC)*. A CRC is a type of error detecting code used to detect accidental changes in data. To detect an error, a fixed length *check value* is appended to the end of the message. The check value is based on the remainder of a polynomial division of the messages contents. When the message is received the check value is recomputed and the appropriate measures to fix the corruption can be taken if the two values do not match.

CRCs are commonly used because they are easy to implement in binary hardware, easy to analyze, and are quite good at detecting burst errors. Usually, an $n$-bit CRC used on an arbitrary length message will detect any single burst error less than or equal to $n$ bits and will detect a fraction $1 - 2^{-n}$ of all longer burst errors.

The specification of a CRC code requires a *generator polynomial*. This polynomial is the divisor in polynomial long division, the dividend is the message, the quotient is thrown away, and the remainder becomes the result. It is important to note that the coefficients of the polynomial are computed using the carry-less arithmetic of a finite field. Most CRCs use the finite field $\mathbb{F}_2$ in practice as this field only has two elements, 0 and 1, which meshes nicely with the architecture of computers.

A CRC is called an $n$-bit CRC when its check value has $n$ bits. Any $n$ has multiple CRCs, each with a different polynomial. Any such polynomial has highest degree $n$, which means that it has $n + 1$ terms, and that its encoding requires $n + 1$ bits. A CRC and associated polynomial are given a name of the form CRC-$n$-XXX. The simplest CRC is CRC-1 and it uses the generator polynomial $x + 1$. This is also called a parity bit, which is the simplest form of error detection.

### 4.1.1 Application and Integrity

A device that implements a CRC calculates the check value for each data block to be sent or stored and appends it to the end of the block. This forms a *codeword.* When a codeword is read or received the device does one of two things. It either recalculates the check value of the data block and compares it to the check value from the codeword, or it performs a CRC on the entire codeword and compares the resulting check value with an expected *residue* constant. If the check values do not match the block contains some errors in its data and the device may take corrective actions. However, because of the nature of error-checking, it is possible that some errors may remain undetected, even if the check values match.

CRCs are designed to protect accidental errors incurred from communication channels, and can give assurance of the integrity of delivered messages. CRCs are not suitable when it comes to protecting against intentional modification of data for several reasons. There is no authentication so an attacker can edit the message and recompute the CRC without being detected. CRC is easily reversible which makes it unsuitable for use in digital signatures. Finally, CRC is a linear function with a property that $crc(x \oplus y) = crc(x) \oplus crc(y)$. Because of this, even if the CRC is encrypted with a stream cipher, the message and its CRC can be altered without knowledge of the encryption key.

### 4.1.2 Computing a CRC

To compute an $n$-bit binary CRC, start with the message encoded in binary:

```
11010011101100
```

This is padded with $n$ zeroes, corresponding to the bit length $n$ of the CRC. The first calculation for a 3-bit CRC is as follows:

```
11010011101100 000 <--- input padded with 3 bits
1011               <--- divisor (4 bits)
------------------
01100011101100 000
```

If the input bit above the leftmost divisor bit is 0, do nothing. If the bit is 1, the divisor is XORed with the input. The divisor is then shifted one bit to the right. This process is repeated until the divisor reaches the right most bit of the input. The calculation in its entirety is:

```
11010011101100 000 <--- input padded with 3 bits
1011               <--- divisor
01100011101100 000 <--- result
 1011              <--- divisor ...
00111011101100 000
  1011
00010111101100 000
   1011
00000001101100 000
       1011
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
-----------------
00000000000000 100 <--- remainder (3 bits)
```

In this case the leftmost bit in the divisor zeroes out all the bits in the input so all that's left is the remainder. To check the validity of the received message simply perform the computation again, but instead of padding the input with zeroes, add the check value instead. If there are no errors the remainder after this computation should be zero.

```
11010011101100 100 <--- input with check value
1011               <--- divisor
01100011101100 100 <--- result
 1011              <--- divisor ...
00111011101100 100

......

00000000001110 100
          1011
00000000000101 100
           101 1
-----------------
                 0 <--- remainder
```

## 4.2 Linear Codes

A *linear code* of length $n$ and rank $k$ is a linear subspace $C$ with dimension $K$ of the vector space $\mathbb{F}_q^n$ where $\mathbb{F}_q$ is the finite field with $q$ elements. For our purposes $q = 2$. The vectors in $C$ are called *codewords*. The *weight* of a codeword is the number of its nonzero elements. The *distance* between two codewords is the number of elements in which they differ. This is called the *Hamming distance*.

Since $C$ is a linear subspace of $\mathbb{F}_q^n$, $C$ can be represented by a basis. A martix can be constructed from this basis where each row in the matrix is an element from the basis set. This martix is called the *generator matrix* of $C$.

The *dual code* of a code $C$, denoted $C^\perp$ is the code

$$C^\perp = \{\mathbf{x} \in F^n \mid \mathbf{x} \cdot \mathbf{c} = 0, \text{ for all } c \in C\}$$

where $\mathbf{x} \cdot \mathbf{c}$ is the usual dot product. If $C$ is a linear code, then $C^{\perp\perp} = C$. Dual codes are useful because they possess the property that if $G$ is a generator matrix for $C$ then $\mathbf{x}$ is in $C$ if and only if $G\mathbf{x}^\mathsf{T} = \mathbf{0}$.

The generator matrix $H$ for the dual code $C^\perp$ of linear code $C$ is called a *check matrix* for $C$. Since $C^{\perp\perp} = C$, we can use the check matrix $H$ for $C$ to define $C$ as:

$$C = \{\mathbf{x} \mid H\mathbf{x}^\mathsf{T} = \mathbf{0}\}.$$

Often a code is defined using a check matrix. [Insert example here?]

## 5 Generalized Reed-Solomon Codes

Reed-Solomon codes were first introduced in 1961 by I.S. Reed and G. Solomon. These codes are somewhat rare in that they have both useful practical applications and an interesting mathematical foundation. Today, Reed-Solomon codes are used in many applications such as CD players and deep-space communications.

Reed-Solomon codes are a linear code. An $n$ length and $k$ dimension RS code is capable of correcting $\left\lfloor \frac{n-k}{2} \right\rfloor$ symbol errors. Because of this RS codes are quite adept at correcting burst errors. For instance, lets say we have an RS code of length $n = 128$ and dimension $k = 120$. Each symbol of our code will be made up of eight bits. Since $n - k = 128 - 120 = 8$ this means that we can correct any four symbol errors in our block of 128 symbols. This is true even if they are four consecutive symbols, as could happen due to a burst error.

J. Hall presents, in *Notes on Coding Theory*, generalized Reed-Solomon Codes. It is these generalized codes that we will be focusing on for the remainder of this paper. Below we will discuss the basic structure of these codes. After that we will go into detail about how to encode and decode a generalized Reed-Solomon Code.

[There will also be an example to go along with this section in the future, unfortunately it is incomplete in this version]

### 5.1 The Basics

Let $F$ be a field. Choose nonzero elements $v_1, ..., v_n \in F$ and distinct elements $\alpha_1, ..., \alpha_n \in F$. Set $\mathbf{v} = (v_1, ..., v_n)$ and $\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_n)$ For $0 \leq k \leq n$ *generalized Reed-Solomon codes* are defined as:

$$\mathrm{GRS}_{n,k}(\boldsymbol{\alpha},\mathbf{v}) = \{(v_1 f(\alpha_1), v_2 f(\alpha_2), ..., v_n f(\alpha_n)) \mid f(x) \in F[x]_k\}.$$

Here $F[x]_k$ is the set of polynomials in $F[x]$ with degree less than $k$. If $f(x)$ is a polynomial, then $\mathbf{f}$ is its associated codeword. This codeword is also dependent on $\boldsymbol{\alpha}$ and $\mathbf{v}$. We can write

$$\mathbf{ev}_{\boldsymbol{\alpha},\mathbf{v}}(f(x)) = (v_1 f(\alpha_1), v_2 f(\alpha_2), ..., v_n f(\alpha_n)),$$

where $f = \mathbf{ev}_{\boldsymbol{\alpha},\mathbf{v}}(f(x))$ when the polynomial $f(x)$ is evaluated at $\boldsymbol{\alpha}$ and scaled by $\mathbf{v}$.

The distance between two codewords is defined as the number of symbols in which the sequences differ, in other words it is the Hamming distance. The minimum distance between two codewords for GRS codes is

$$\mathrm{d}_{\min} = n - k + 1.$$

A key concept is that any codeword which has up to $k$ entries equal to 0 corresponds to a polynomial of degree less than $k$ whose values matching the 0 polynomial in $k$ points must be the 0 polynomial. This is true since any polynomial of degree less than $k$ is uniquely determined by its values at $k$ distinct points. Which means that for any $n$-tuple $\mathbf{f}$, we can reconstruct the polynomial $f(x)$ of degree less than $n$ such that $\mathbf{f} = \mathbf{ev}_{\boldsymbol{\alpha},\mathbf{v}}(f(x))$. Let

$$L(x) = \prod_{i=1}^{n} (x - \alpha_i)$$

and

$$L_i(x) = L(x)/(x - \alpha_i) = \prod_{j \neq i} (x - \alpha_j).$$

Both $L(x)$ and $L_i(x)$ are *monic* polynomials of degrees $n$ and $n-1$, respectively. A polynomial is monic if the leading coefficient is equal to 1. Since the $i^{\text{th}}$ coordinate of vector $\mathbf{f}$ is $v_i f(\alpha_i)$ we can use Lagrange interpolation [insert reference to Lagrange interpolation here] to calculate

$$f(x) = \sum_{i=1}^{n} \frac{L_i(x)}{L_i(\alpha_i)} f(\alpha_i).$$

The polynomial $f(x)$ has degree less than $k$, while the interpolation polynomial of the righthand side has degree of $n-1$. The solution to this problem allows us to calculate the dual of a GRS code more easily. Hall gives a theorem in Chapter 5 of *Notes on Coding Theory* stating that:

$$\mathrm{GRS}_{n,k}(\boldsymbol{\alpha},\mathbf{v})^{\perp} = \mathrm{GRS}_{n,n-k}(\boldsymbol{\alpha},\mathbf{u}),$$

where $\mathbf{u} = (u_1, ..., u_n)$ with $u_i^{-1} = v_i \prod_{j \neq i} (\alpha_i - \alpha_j)$.

To verify that $\mathbf{f}$ is a codeword in $C = \mathrm{GRS}_{n,k}(\boldsymbol{\alpha},\mathbf{v})$ it is not necessary to compare it to every $\mathbf{g}$ of $C^{\perp} = \mathrm{GRS}_{n,n-k}(\boldsymbol{\alpha},\mathbf{v})$. Instead, we can use a basis of $C^{\perp}$, which is also a check matrix for $C$. Using a check matrix to define a linear code has its benefits. One such benefit is that it will allow us to use *syndrome decoding*, which will be discussed in more detail in a later section.

## 5.2 Encoding GRS Codes

The field we are using to encode is $\mathbb{F}_{2^6}$. This is the field of all 6 bit binary strings. Addition and multiplication work as described above in this field. This gives us 64 different symbols we can use in our source alphabet. For our example we chose to map 64 ASCII characters to the elements in $\mathbb{F}_{2^6}$. Specifically, we mapped characters 32-95 inclusive. Because this map exists, we do not have to think directly about the binary strings underneath. We can write the elements of $\mathbb{F}_{2^6}$ as those characters as well as perform the field operations on those characters. A Ruby script was used to generate an addition and multiplication table for these operations. The tables as well as the script can be seen in appendix A.

A GRS code is composed of two parts. The first part is the actual message that we are sending. The message is $n$ symbols long and can be written as an $n-1$ degree polynomial. The second part is the parity symbols. There are $r = n - k$ parity symbols appended to the end of the message to be used to ensure correctness after transmission.

In order to find the parity symbols we need to use a generating polynomial. The generating polynomial for a Reed-Solomon code is of the following form:

$$g(x) = g_0 + g_1 x + g_2 x^2 + ... + g_{r-1} x^{r-1} + x^r$$

In order to calculate $g(x)$ we need to find an element, $a$ in our group which is a primitive element. Then,

$$g(x) = (x - a)(x - a^2)...(x - a^r)$$

The remainder of this process is similar to how a CRC is performed. Once, found, we can divide the original message polynomial by $g(x)$. The remainder of this division is the parity symbols which we then add to the end of the message. Now that the message is complete we can send it.

## 5.3 Decoding GRS Codes

Having an encoded message is useless without someway to decode it. Reed-Solomon codes take advantage of a technique called *syndrome decoding.* Syndrome decoding uses the dual code and check matrices to decode and to detect and correct errors that may have occurred in transmission.

The code $\mathrm{GRS}_{n,k}(\boldsymbol{\alpha}, \mathbf{v})$ over $F$ is equal to the set of all $n$-tuples $\mathbf{c} = (c_1, c_2, ..., c_n) \in F^n$ such that:

$$\sum_{i=1}^{n} \frac{c_i u_i}{1 - \alpha_i z} = 0 \ (\text{mod } z^r),$$

where $r = n - k$ and $u_i^{-1} = v_i \prod_{j \neq i} (\alpha_i - \alpha_j)$. This is known as the *Goppa formulation for GRS codes.* The Goppa formulation is simply another way of writing our code, but there are benefits to using this formulation as we will see below.

If $\mathbf{c} = (c_1, c_2, ..., c_n)$, a GRS code in the Goppa formulation, is transmitted, and $\mathbf{p} = (p_1, p_2, ..., p_n)$ is received, then there is some vector $\mathbf{e} = (e_1, e_2, ...e_n)$ such that $\mathbf{p} = \mathbf{c}+\mathbf{e}$. This vector is called the error vector. We can then calculate the *syndrome polynomial* of $\mathbf{p}$:

$$S_{\mathbf{p}}(z) = \sum_{i=1}^{n} \frac{p_i u_i}{1 - \alpha_i^z} \pmod{z^r}.$$

We can do the same for $\mathbf{c}$ and $\mathbf{e}$ as well. The following equation holds true:

$$S_{\mathbf{p}}(z) = S_{\mathbf{c}}(z) + S_{\mathbf{e}}(z) \pmod{z^r}.$$

Since $\mathbf{c}$ is in the Goppa formulation

$$S_{\mathbf{c}}(z) = \sum_{i=1}^{n} \frac{c_i u_i}{1 - \alpha_i z} = 0,$$

so we write:

$$S_{\mathbf{p}}(z) = S_{\mathbf{e}}(z) \pmod{z^r}.$$

Let $B$ be the set of error locations:

$$B = \{i \mid e_i \neq 0\}.$$

Then the syndrome polynomial reduces further to:

$$S_{\mathbf{p}}(z) = S_{\mathbf{e}}(z) = \sum_{b \in B} \frac{e_b u_b}{1 - \alpha_b z} \pmod{z^r}.$$

We can drop the subscripts and just write $S(z)$ for the syndrome polynomial.

The next step is to find the *Key Equation.* This is done by clearing the denominators:

$$\sigma(z)S(z) = \omega(z) \pmod{z^r},$$

where

$$\sigma(z) = \sigma_{\mathbf{e}}(z) = \prod_{b \in B} (1 - \alpha_b z)$$

and

$$\omega(z) = \omega\mathbf{e}(z) = \sum_{b \in B} e_b u_b \left( \prod_{a \in B, a \neq b} (1 - \alpha_a z) \right).$$

The polynomial $\sigma(z)$ is called the *error locator*, and the polynomial $\omega(z)$ is called the *error evaluator.*

We can determine the error vector $\mathbf{e}$ given $\sigma(z)$ and $\omega(z)$. If we assume for now that none of the $a_i$ equal 0, then:

$$B = b \mid \sigma(a_b^{-1}) = 0$$

and, for each $b \in B$,

$$e_b = \frac{-\alpha_b \omega(\alpha_b^{-1})}{u_b \sigma'(\alpha_b^{-1})},$$

where $\sigma'(z)$ is the derivative of $\sigma(z)$. It can be shown that the polynomials $\sigma(z)$ and $\omega(z)$ determine the error vector even when some $\alpha_i$ is 0.

This method of decoding requires us to solve the Key Equation in order to find the error vector. The following theorem gives us a characterization of $\sigma(z)$ and $\omega(z)$, helping us solve the Key Equation.

*Given $r$ and $S(z) \in F[z]$ there is at most one pair of polynomials $\sigma(z), \omega(z) \in F[z]$ satisfying*:

1. $\sigma(z)S(z) = \omega(z) \pmod{z^r}$;
2. $\deg(\sigma(z)) \leq r/2$ and $\deg(\omega(z)) < r/2$;
3. $\gcd(\sigma(z), \omega(z)) = 1$ and $\sigma(0) = 1$.

Using this characterization we can solve the Key Equation using the Euclidean algorithm. For a code $\mathrm{GRS}_{n,k}(\boldsymbol{\alpha}, \mathbf{v})$ over $F$, with $r = n-k$, given a syndrome polynomial $S(z)$, the following algorithm halts, producing $\tilde{\sigma}(z)$ and $\tilde{\omega}(z)$:

**Algorithm 1:** Decoding GRS using the Euclidean Algorithm

```
Set $a(z) = z^r$ and $b(z) = S(z)$.
Step through the Euclidean Algorithm until at Step $j$, $\deg(r_j(z)) < r/2$.
Set $\tilde{\sigma}(z) = t_j(z)$ and $\tilde{\omega}(z) = r_j(z)$.
```

If the error vector exists, then $\hat{\sigma}(z) = \tilde{\sigma}(0)^{-1}\tilde{\sigma}(z)$ and $\hat{\omega}(z) = \tilde{\sigma}(0)^{-1}\tilde{\omega}(z)$ are the error locator and error evaluator polynomials for $\mathbf{e}$. However, there are a few stipulations. The number of roots of $\hat{\sigma}(z)$ among the $\alpha_i^{-1}$ must be equal to the degree of $\hat{\sigma}(z)$. If this is not the case, that means we have detected errors which we cannot correct. Also, if $t_j(0) = 0$ we cannot perform the final division to determine $\hat{\sigma}(z)$.

If $t_j(0) \neq 0$ and $\hat{\sigma}(z)$ has the correct number of roots, then we can evaluate errors at each location to find a vector of weight at most $r/2$ with our original syndrome. At this point we have either decoded correctly, or we had more than $r/2$ errors and could not decode properly.

## 6 Conclusion

[Coming Soon]

## 7 References

[Coming Soon]