Generalized Reed-Solomon Codes

John McCall
Course: Math 4901
Adviser: Peter Dolan
Second Reader: David Roberts

April 24, 2014

1 Abstract

Error correcting codes are used to correct any errors which may occur in a data transmission. Generalized Reed-Solomon (GRS) codes are a class of error correcting code. They are utilized in real world applications such as CD-roms and deep space communications, due to their ability to recover even when a large chunk of data has been compromised. While no error correcting code is perfect, the ability to correct any number of errors during data transmission is incredibly useful. This paper describes generalized Reed-Solomon codes and goes into detail about the process of encoding and decoding a GRS code.

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 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. 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.

Reed-Solomon codes are error correcting codes. They were first introduced in 1960 by I.S. Reed and G. Solomon [2]. 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.

In Section 3 we will provide the necessary abstract algebra background to understand the remainder of the paper. Section 4 will introduce coding theory and give examples of some codes. In Section 5 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.

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. Inverse 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 $\mathbb{Q}$ under addition. It is easy to see that the rational numbers under normal addition satisfies all of the above properties and is a group. Another example would be $\mathbb{Q}^*$, the rationals 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 $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$.

The multiplication does not have to be commutative. If it is, then the ring is called a *commutative ring*. Another thing to notice is that multiplicative inverses are not required in a ring. An element that does have a multiplicative inverse is called a *unit*. If a ring is commutative and every non-zero element is a unit, that is, every element has a multiplicative inverse then that ring is called a *field*. The integers, along with normal addition and multiplication, are an example of a commutative ring. However, they are not a field since not every element has a multiplicative inverse. The solution to this is to add fractions, giving every element a multiplicative inverse. Thus, the rational numbers are a field.

We use the notation $\mathbb{F}[x]$ to denote all polynomials with coefficients in the field $\mathbb{F}$. Polynomials with rational coefficients would be denoted $\mathbb{Q}[x]$, polynomials with real coefficients would be denoted $\mathbb{R}[x]$. We write $\mathbb{F}[x]_k$ to denote all polynomials with coefficients in $\mathbb{F}$ with degree less than k.

3.2 Defining $\mathbb{F}_{64}$

The field which we will focus on in this paper is $\mathbb{F}_{64}$. The finite field of 64 elements. For our purposes, the elements of this field will be 64 ASCII characters, shown in Appendix A. The elements include all capital letters, the numbers zero through nine, as well as some punctuation. These characters where chosen specifically to make encoding easier. Since these symbols can be hard to distinguish from actual text, the symbols will be given a grey background. For example: `A`, `1`, `#` are all elements in $\mathbb{F}_{64}$.

These elements can be represented as binary strings, which in turn represent polynomials in $\mathbb{F}_2[x]_6$. In other words, polynomials with coefficients that are either zero or one that are of degree five or less. For example, `#` can be written as the binary string `11`. This binary string represents the polynomial $x+1$. The symbol `$` corresponds to the binary string `100`, which gives us the polynomial x^2.

3.2.1 Addition in $\mathbb{F}_{64}$

To perform addition in $\mathbb{F}_{64}$, we first convert the elements being added to their polynomial form. Then we simply add those polynomials, keeping in mind that the coefficients are in $\mathbb{F}_2$. To illustrate this, take `!` + `#`. This becomes: $1 + (x + 1)$. Doing the addition in $\mathbb{F}_2$ yields x, which represents the element `"`. So, `!` + `#` = `"`. A complete addition table for $\mathbb{F}_{64}$ can be found in Appendix B.

3.2.2 Multiplication in $\mathbb{F}_{64}$

Similarly to addition, we use the polynomial representation of our elements to multiply them. Let us multiply `#` by itself.

$$\texttt{\#} * \texttt{\#} = (x+1)*(x+1) = x^2+x+x+1 = x^2+1 = \texttt{\%}.$$

A problem that arises when doing multiplication is that it is possible to produce a polynomial that is higher than degree 5. For instance:

$$\texttt{\%} * \texttt{0} = (x^2+1)*(x^4) = x^6+x^4$$

When this occurs we use the the irreducible polynomial x^6+x+1 to reduce the power. In $\mathbb{F}_2$ we can rewrite that polynomial as $x^6 = x+1$. We can then substitute this value into our product to lower the degree.

$$x^6+x^4 = (x+1)+x^4 = x^4+x+1 = \texttt{3}$$

A complete multiplication table can be found in Appendix B. In addition to this there is also a grey-scale multiplication table included in Appendix B. The grey-scale table helps to illustrate patterns that appear in the multiplication that are hard to see using the normal table. Most notably, it is easy to see multiplication is commutative using the grey-scale table because the table is symmetrical along the main diagonal.

3.2.3 Polynomials over $\mathbb{F}_{64}$

Let us define $\mathbb{F}_{64}[x]$ to be all polynomials with coefficients in $\mathbb{F}_{64}$. We can perform all of the usual polynomial operations using the addition and multiplication methods defined above including: addition, multiplication, substitution, and taking the derivative. For example, let

$$f(x) = \texttt{!}\,x^3 + \texttt{!}\,x^2 + \texttt{\#}\,x + \texttt{/}$$

then

$$\begin{aligned} f(\texttt{A}) &= (\texttt{!} * \texttt{A}^3) + (\texttt{!} * \texttt{A}^2) + (\texttt{\#} * \texttt{A}) + \texttt{/} \\ &= (\texttt{!} * \texttt{Q}) + \texttt{!} + (\texttt{\#} * \texttt{Y}) + \texttt{/} \\ &= \texttt{Q} + \texttt{Y} + \texttt{@} + \texttt{/} \\ &= \texttt{G} \end{aligned}$$

The derivative, $f'(x)$, would equal

$$\begin{aligned} f'(x) &= 3 * \texttt{!}\,x^2 + 2 * \texttt{!}\,x + \texttt{\#} \\ &= (\texttt{!} + \texttt{!} + \texttt{!})x^2 + (\texttt{!} + \texttt{!})x + \texttt{\#} \\ &= \texttt{!}\,x^2 + \texttt{\#} \end{aligned}$$

4 Codes

A *code* is a rule for converting information (usually a letter or word) into another representation. *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, "1,12,7,5,2,17,1" 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: "1,17,7,1,2,17,1". 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 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. 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.

Before examining Reed-Solomon codes, we will discuss a simpler code. Cyclic Redundancy Check is a good code to start with because it introduces several important ideas in coding theory.

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 [4].

The specification of a CRC code requires a *generator polynomial*, which is an

irreducible polynomial of degree n. 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. When a block 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 received, or it performs a CRC on the entire block and checks that the result is zero. If the check values do not match, then errors have been successfully detected in the block. Due to 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. One reason is that there is no authentication so an attacker can edit the message and recompute the CRC without being detected.

4.1.2 Computing a CRC

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

```
1100101
```

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:

```
1100101 000 <--- input padded with 3 bits
1011        <--- divisor (4 bits)
------------------
0111101 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:

```
1100101 000 <--- input padded with 3 bits
1011        <--- divisor
0111101 000 <--- result
 1011       <--- divisor ...
0010001 000
  1011
0000111 000
    101 1
0000010 100
     10 11
-------------
0000000 010 <--- 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. If the remainder is not zero that means that there were errors in the received message.

```
1100101 010  <-- No errors          Errors -->  1010101 010
1011                                            1011
0111101 010                                     0001101 010
 1011                                              1011
0010001 010                                     0000110 010
  1011                                              101 1
0000111 010                                     0000011 110
    101 1                                             10 11
0000010 110                                     0000001 000
     10 11                                              1 011
-------------                                   -------------
0000000 000           <-- Remainder -->         0000000 011
```

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 $q = 2^m$. 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 matrix can be constructed from this basis where each row in the matrix is an element from the basis set. This matrix 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, generalized to $\mathbb{F}_q^n$. 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. For us, $C^{\perp\perp} = C$, so 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.

5 Reed-Solomon Codes

Reed-Solomon codes are a linear code. An n length and k dimension Reed-Solomon 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 a Reed-Solomon 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.

A message encoded using a Reed-Solomon code is a polynomial with the message embedded in the coefficients. This is called the *message polynomial*, $m(x)$. In general, $m(x) \in F[x]_k$ where F is a field. However, we will be working in $\mathbb{F}_{64}$ for the remainder of the paper so we will have $m(x) \in \mathbb{F}_{64}[x]_k$. Say the message we want to encode is: "THIS IS MAJOR TOM". Then our message polynomial is:

$$m(x) = \texttt{T} + \texttt{H}\,x + \texttt{I}\,x^2 + \texttt{S}\,x^3 + \texttt{ }x^4 + \texttt{I}\,x^5 + \texttt{S}\,x^6 + \texttt{ }x^7 + \texttt{M}\,x^8 + \texttt{A}\,x^9 + \texttt{J}\,x^{10} + \texttt{O}\,x^{11} + \texttt{R}\,x^{12} + \texttt{ }x^{13} + \texttt{T}\,x^{14} + \texttt{O}\,x^{15} + \texttt{M}\,x^{16}.$$

Similar to the CRC example, the message polynomial is divided by a generator polynomial. The degree of $m(x)$ is then upshifted by $n-k$ and the remainder of the division is added to it. These extra terms function like the check value did for CRC. Unlike CRC, the extra terms provide enough information that we are able to fix a certain number of errors, instead of just detecting them.

There is an alternative representation of Reed-Solomon codes, known as generalized Reed-Solomon codes [1]. Where Reed-Solomon codes use the message polynomial and some parity terms as the codeword, a generalized Reed-Solomon code evaluates the message polynomial at n distinct points. The resulting vector is used as the codeword. We will be focusing on the generalized case.

5.1 Generalized Reed-Solomon Codes

Let F be a field. F can be any field, but we will use F_{64}. Choose nonzero elements $\hat{v} = (v_1, ..., v_n) \in F^n$ and distinct elements $\hat{\alpha} = (\alpha_1, ..., \alpha_n) \in F^n$. For $0 \leq k \leq n$ *generalized Reed-Solomon codes* are defined as:

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

For our example, $k = 17$, $n = 23$, $\hat{v} = (\texttt{!}, \texttt{!}, ..., \texttt{!})$, $\hat{\alpha} = (\texttt{A}, \texttt{B}, ..., \texttt{W})$, and $f(x) = m(x)$ is our message polynomial. Since ! is the multiplicative identity, this essentially means that we are scaling everything by "1". The error correcting capacity of this example code is $\frac{23-17}{2} = 3$, so we can correct up to three symbol errors. The codeword associated with a message polynomial is dependent on both $\hat{v}$ and $\hat{\alpha}$ and is defined as

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

where the message polynomial is evaluated at $\hat{\alpha}$ and scaled by $\hat{v}$.

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. Given as a theorem in [1], the minimum distance between two codewords for GRS codes is

$$\mathrm{d}_{\min} = n - k + 1 \text{ where } k \neq 0.$$

A key concept is that any codeword which has more than 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 $\hat{c}$, we can reconstruct the message polynomial $f(x)$ of degree less than or equal to k such that $\hat{c} = \mathbf{ev}_{\boldsymbol{\alpha},\mathbf{v}}(f(x))$. 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^{th} coordinate of vector $\hat{c}$ is $v_i f(\alpha_i)$ we can use Lagrange interpolation 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, $L_i(x)$, 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:

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

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

To verify that $\hat{c}$ is a codeword in $C = \mathrm{GRS}_{n,k}(\hat{\alpha}, \hat{v})$ it is not necessary to compare it to every $\hat{g}$ of $C^{\perp} = \mathrm{GRS}_{n,n-k}(\hat{\alpha}, \hat{v})$. 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 Section 5.2.

A program written in a language called Ruby was used to perform the actual evaluations for the example. The source code for this program can be seen in Appendix C. The message to be encoded is "THIS IS MAJOR TOM", recall that

$$f(x) = \texttt{T} + \texttt{H}x + \texttt{I}x^2 + \texttt{S}x^3 + \texttt{ }x^4 + \texttt{I}x^5 + \texttt{S}x^6 + \texttt{ }x^7 + \texttt{M}x^8 + \texttt{A}x^9 + \texttt{J}x^{10} + \texttt{O}x^{11} + \texttt{R}x^{12} + \texttt{ }x^{13} + \texttt{T}x^{14} + \texttt{O}x^{15} + \texttt{M}x^{16}$$

and that $\hat{v} = (\texttt{!}, \texttt{!}, ..., \texttt{!})$ and $\hat{\alpha} = (\texttt{A}, \texttt{B}, ..., \texttt{W})$. The program was told to calculate the codeword for the specified $\hat{v}, \hat{\alpha},$ and $f(x)$. The resulting codeword is

$$\hat{c} = (\texttt{T}, \texttt{(}, \texttt{U}, \texttt{8}, \texttt{P}, \texttt{K}, \texttt{G}, \texttt{N}, \texttt{W}, \texttt{P}, \texttt{4}, \texttt{K}, \texttt{'}, \texttt{ }, \texttt{N}, \texttt{(}, \texttt{M}, \texttt{H}, \texttt{K}, \texttt{1}, \texttt{"}, \texttt{<}, \texttt{K}).$$

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

$$\sum_{i=1}^{n} \frac{c_i u_i}{1 - \alpha_i z} = 0 \pmod{z^r},$$

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 message that was sent is $\hat{c}$. Let $\hat{p}$ be the message that was received. For our example

$$\hat{p} = (\texttt{T}, \texttt{(}, \texttt{U}, \texttt{8}, \texttt{P}, \texttt{K}, \texttt{Z}, \texttt{N}, \texttt{W}, \texttt{P}, \texttt{\&},$$
$$\texttt{K}, \texttt{'}, \texttt{\_}, \texttt{N}, \texttt{(}, \texttt{M}, \texttt{H}, \texttt{K}, \texttt{1}, \texttt{"}, \texttt{R}, \texttt{K}).$$

Three errors were introduced during transmission, highlighted in red.

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 α_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 $\mathrm{GRS}_{n,k}(\hat{\alpha}, \hat{v})$ 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 α_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

In this paper we introduced important abstract algebra concepts, such as groups and fields, discussed linear codes, such as Cyclic Redundancy Check, and gone into detail about how GRS codes are formed. Using an example, we showed the process of encoding and decoding a GRS code and highlighted how the error correction process works. Although, we cannot correct every error, having the ability to automatically correct some errors is undeniably useful.

References

[1] J. Hall. *Notes on Coding Theory.* Michigan State University, 2012.

[2] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, June 1960.

[3] B. Sklar. Reed solmon codes. 2001.

[4] Wikipedia. Cyclic redundancy check — wikipedia, the free encyclopedia, 2014.

7 Appendix A: The elements of $\mathbb{F}_{64}$

8 Appendix B: Addition and Multiplication Tables for $\mathbb{F}_{64}$

Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
32	20	040		Space	64	40	100	@	@
33	21	041	!	!	65	41	101	A	A
34	22	042	"	"	66	42	102	B	B
35	23	043	#	#	67	43	103	C	C
36	24	044	$	$	68	44	104	D	D
37	25	045	%	%	69	45	105	E	E
38	26	046	&	&	70	46	106	F	F
39	27	047	'	'	71	47	107	G	G
40	28	050	(	(	72	48	110	H	H
41	29	051	)	)	73	49	111	I	I
42	2A	052	*	*	74	4A	112	J	J
43	2B	053	+	+	75	4B	113	K	K
44	2C	054	,	,	76	4C	114	L	L
45	2D	055	-	-	77	4D	115	M	M
46	2E	056	.	.	78	4E	116	N	N
47	2F	057	/	/	79	4F	117	O	O
48	30	060	0	0	80	50	120	P	P
49	31	061	1	1	81	51	121	Q	Q
50	32	062	2	2	82	52	122	R	R
51	33	063	3	3	83	53	123	S	S
52	34	064	4	4	84	54	124	T	T
53	35	065	5	5	85	55	125	U	U
54	36	066	6	6	86	56	126	V	V
55	37	067	7	7	87	57	127	W	W
56	38	070	8	8	88	58	130	X	X
57	39	071	9	9	89	59	131	Y	Y
58	3A	072	:	:	90	5A	132	Z	Z
59	3B	073	;	;	91	5B	133	[	[
60	3C	074	<	<	92	5C	134	\	\
61	3D	075	=	=	93	5D	135	]	]
62	3E	076	>	>	94	5E	136	^	^
63	3F	077	?	?	95	5F	137	_	_

Figure 1: Taken from: http://www.asciitable.com/

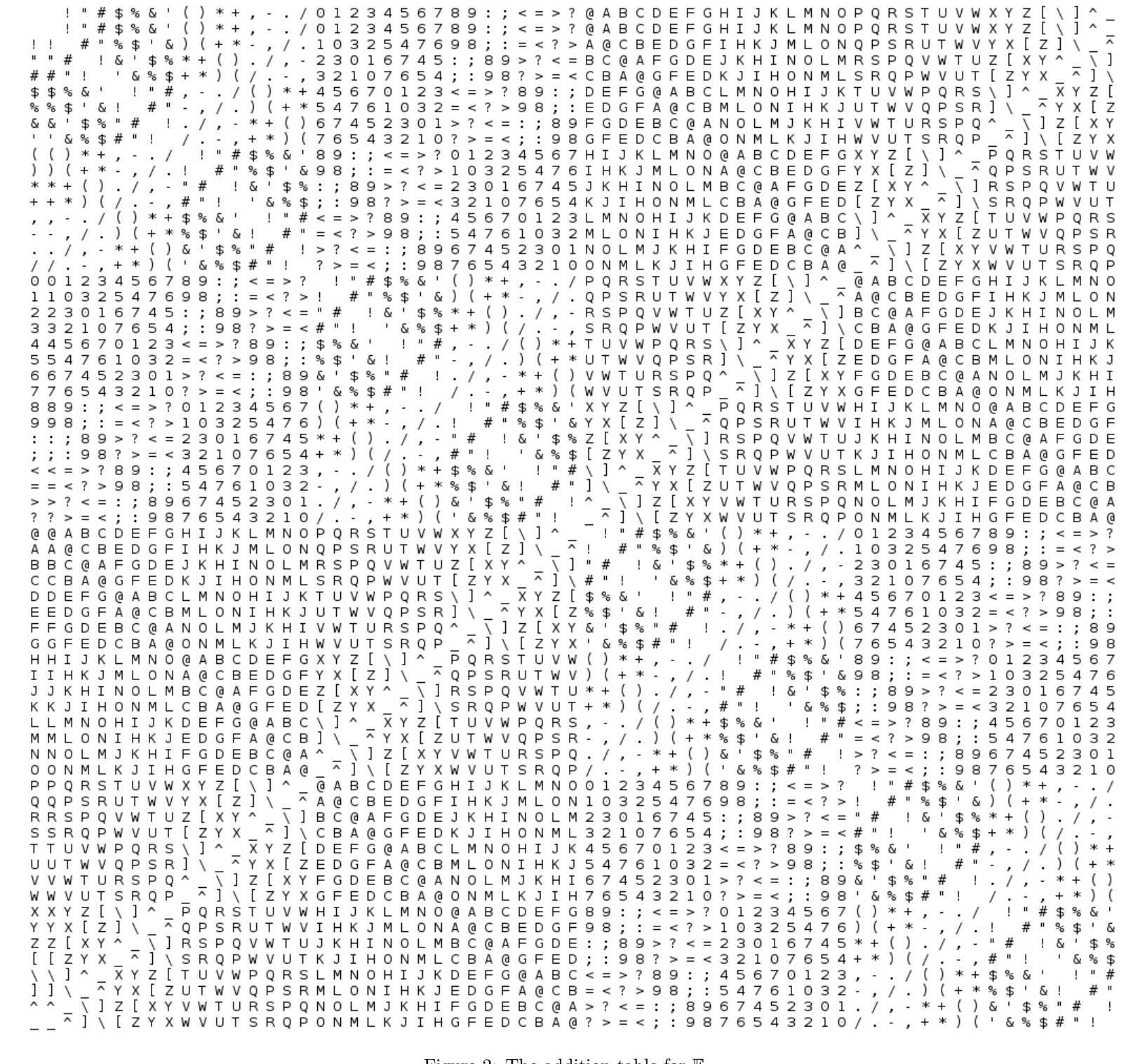

Figure 2: The addition table for $\mathbb{F}_{64}$

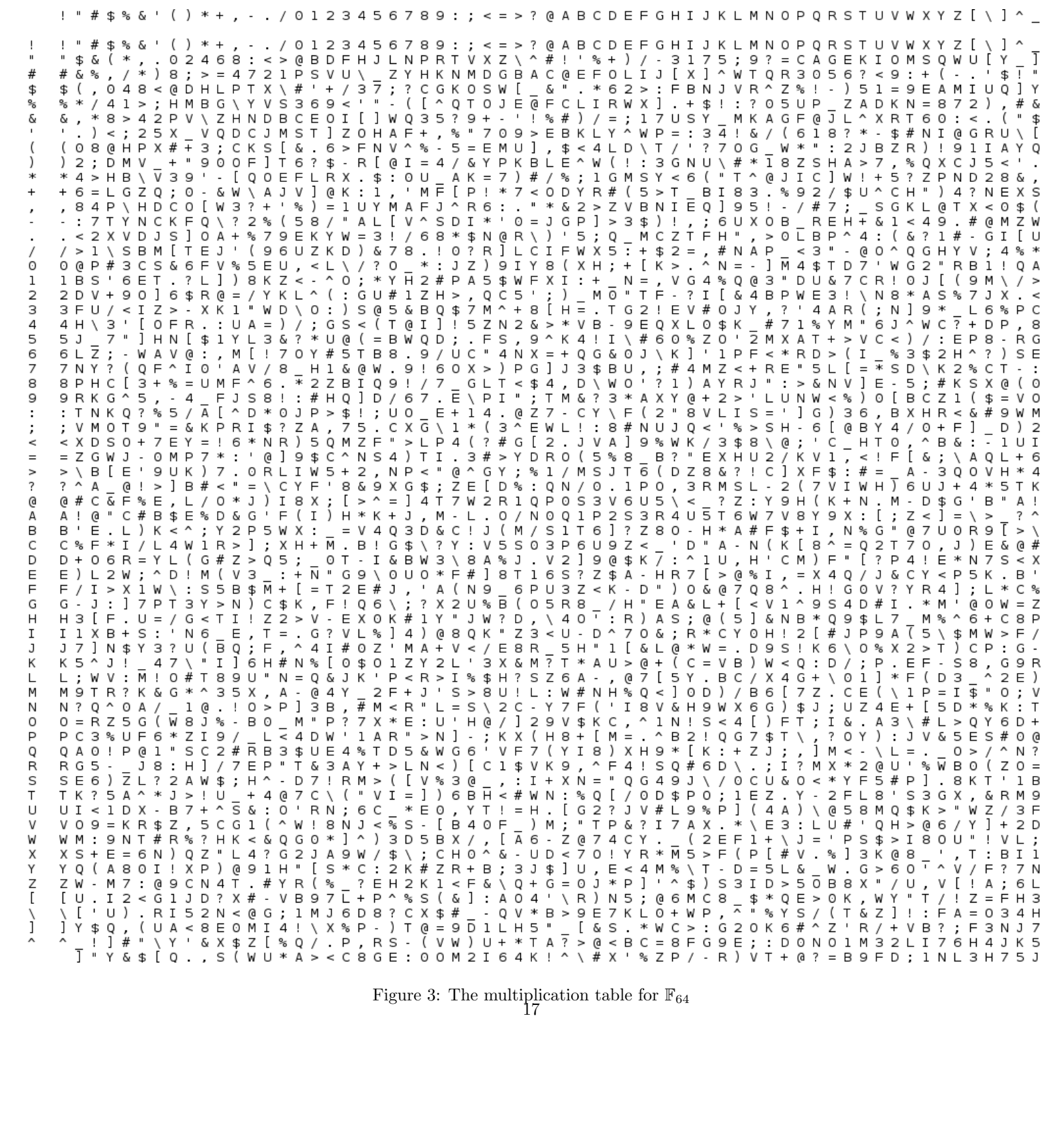

Figure 3: The multiplication table for $\mathbb{F}_{64}$

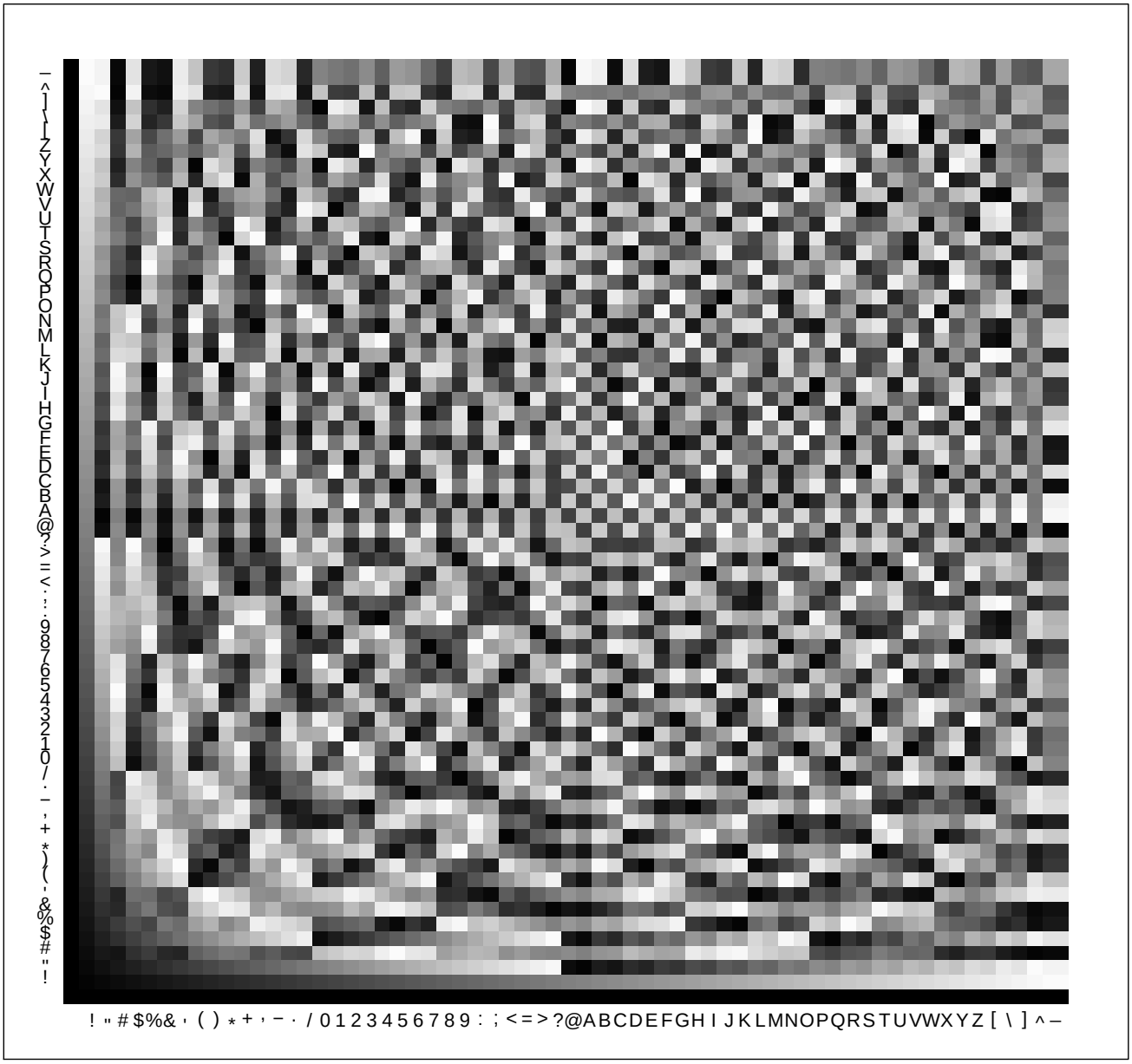

Figure 4: A grey-scale multiplication table for $\mathbb{F}_{64}$