Benedictine College

An Introduction to Coding Theory

with Reed-Solomon Codes as a Real-World Example

March 4, 2019

Augustine Calvino

supervised by
Dr. Nickolas Hein

Abstract

Communication of data is ubiquitous in today's world. Whether within a device or between devices, reliable data transfer is essential to proper function of the databases, applications, servers, and computers that comprise today's technological systems. Errors, however, do occur. They must be corrected. And math can help.

Here we present an introduction to *coding theory*. We will use Reed-Solomon Codes as a real-world example demonstrating some of its fundamental principles.

1 Introduction

The term **Coding Theory** is loosely used to describe any mathematical methods of manipulating data, including cryptography and data compression, but it has come to refer primarily to the study of error correcting codes. For the rest of this paper, this is what we mean by any references to 'codes' or 'coding theory.' This study contrasts with cryptography in that while cryptography aims to make data undecodable (to unintended recipients), the aim of coding theory is to make data intelligible despite interference. It contrasts with data compression in that while data compression compacts data by eliminating redundancy, coding theory makes data bigger by adding redundancy. [C, 1] Like cryptography and data compression, coding theory is used in the contexts of data storage and transmission.

The need for reliable transmission of data is rapidly increasing as the number of types of technological systems in use escalates. Watches, phones, radios, tablets, speaker systems, computers, cars, satellites, and routers must communicate with each other over various channels ranging from a few feet to thousands of miles. Sometimes these channels have the assistance of wires, but sometimes they do not. Data transmission also occurs within devices, such as computer systems busing information between the hard drive, main memory, CPU, keyboard, and CD reader. Some data transmission is by humans directly, such as typing products into an on-line catalog or database.

But just as how in a loud room you may mishear what your friend is saying due to the *noise*, various real-world factors can disrupt the integrity (accuracy) of data. Incidentally, we use the same term, **noise**, to describe the interference that occurs on data. In the case of wi-fi, such interference may be frequencies emitted by a microwave; in the case of satellites, it may be a cloud in the sky; in the case of a CD reader it may be scratches or dust on the CD; and in the case of a human it may just be sloppy typing fingers. Due to these and other factors, transmission cannot be guaranteed to proceed without error.

Consequently, a whole branch of Mathematics and Theoretical Computer Science has been developed to deal with this issue effectively and efficiently.

2 Background

To understand the motivation and terminology behind this study, let us return to the example of a noisy room. Consider such circumstances under which you hear

your friend say "Hă-foo seen my —llet?" where the dashes represent muffled sounds that were covered up by a shout in the room. Immediately you can detect an error simply from the fact that the sounds 'Hă-foo' do not form a word in your vocabulary. In considering the sounds, however, you easily recognize them as being similar to the sounds 'Have you' and interpret them as such. Similarly, though you didn't catch the whole of the last word, in considering the words you know that end in '-llet,' and perhaps some other context (such as a wallet sitting on the table in front of you), you correctly interpret the complete phrase as "Have you seen my wallet?"

In coding theory the set of possible symbols from which a message can be comprised is called the **alphabet** (denoted A), and is analogous to the set of possible sounds in the previous example. Sequences of symbols from A are elements of the set A^* (or A^n for sequences of a specific length n) and are called **strings**. They correspond to combinations of sounds. Just as words are valid combinations of sounds in a vocabulary, **codewords** c are valid strings in a **code** C. This of course suggests that there are non-valid combinations of sounds in a vocabulary and that not every string in A^n is also in C. In other words, $C \subset A^n$, where n is the **length** of the code. When a string s is received such that $s \notin C$, we detect that at least one symbol of that string is an **error**, just as you knew at least one sound in 'Hă-foo' was heard in error because it is not a word. Sometimes, due to the way hardware works, we can know it pretty likely that a specific symbol is in error and treat it as a dropped symbol, or **erasure** in the received string. This correlates to how a shout created enough disruption in the communication channel to cause you to regard the first sounds of 'wallet' as missing rather than being 'brace-' or 'in-.' Erasures are actually preferable to errors because they tell us with which character in a string an error occurred. Given knowledge of errors and erasures, we can often correct a string to the codeword c that is most similar.

For example, let $A = \{0-9, a-z, A-Z\}$, the set of legal symbols to use in a certain password. If the password must be eight characters and contain at least one of each type, then C is a subset of A^8, and it consists of all codewords c that fulfill the above condition. One possible such codeword would be $I8tb4uop$, and if it was received from transmission as $i8tb4uop$, we could detect that the string is not a codeword because it has no upper-case letter. In other words, we detect an error. Although error correction will rarely, if ever, be used on passwords, this demonstrates how a code is a subset of the possible strings over an alphabet.

In some sense, these examples demonstrate the inefficiency of data encoding and

language. That is, a code of just as many elements could be generated using fewer symbols per codeword and a language of just as many words could be generated using fewer sounds per word. Returning to the password example, we could use every possible combination of symbols from the alphabet and not restrict the criteria for a valid password to generate just as many passwords as in the previous example, while needing fewer than eight characters. In the case of language, we could shorten the length of words by designating meaning to syllables like 'bu' or 'ek' before having words like 'compartmentalization' in our vocabulary. Now, that is not the way language develops, but these example demonstrate the way that the length of the medium used to convey an idea (or password) is often longer than the minimum possible length of the idea expressed by that message.

In coding theory we actually intentionally lengthen our initial information by adding data symbols to the end or by systematically manipulating the data string to something wholly different and longer. Just as the function $y = 2x$ transforms elements of $\mathbb{Z}$ to elements of $2\mathbb{Z}$ (the even integers), coding theory transforms information from the set of possible data strings to elements of C. This functional transformation is also known as **mapping**, and the rule which governs it is a **mapping function**. The length of the data before mapping it to the code is typically denoted k, the length of the code is typically denoted n, and a code that is designed around such a transformation is denoted $C(n, k)$. The difference in length that this process adds without adding new information is known as **redundancy**.

This lengthening of the data, however, is exactly what enables us to detect and correct errors. Recall the example where you correctly interpreted what you misheard from your friend. If all sound combinations had meaning, you would never even have realized that you misheard your friend. Similarly, if all strings were codewords, we could never detect an error, much less correct one. A typical way of correcting is to find the codeword with the least difference from the received string. This amount of difference between particular codewords, measured simply as the number of symbols which differ between them, is called the **distance**, and for a code C, the least distance that exists between distinct elements of C is the **minimum distance** d_{min}. If the number of errors that occur on a codeword is not less than d_{min}, there is a risk that the received string is another codeword and that the error will not be detected. Similarly, if the number of errors is not less than $\frac{d_{min}}{2}$, the received string may be more similar to a different codeword than the one it is supposed to be, and correction techniques will not be effective. Hence (as shown in [H, 2]), a code can correct up to

t errors, where

$$t \leq \lfloor \frac{d_{min} - 1}{2} \rfloor. \tag{1}$$

For these reasons, a good code will have a large minimum distance relative to the amount of redundancy. The ratio of a code's error correcting and detecting capability to the amount of redundancy is its **efficiency**.

For example, let $A = \{1, 2, 3\}$, $C = \{1231, 1312, 2132, 3113, 2223, 3321\} \subset A^4$, and $a, b \in C$. For $a = 1231$ and $b = 3113$, the distance $dist(a, b) = 4$ because every corresponding symbol differs between those codewords. The minimum distance of this code is $d_{min}(C) = 3$ because any pair of codewords have at least three symbols which differ. If b was received with one symbol error as $b' = 3123$, we could successfully correct it to the most similar codeword. If two symbol errors occurred such that $b' = 3323$, the most similar codeword would be 3321, not 3113, and error correction would fail. This reflects that the max number of errors this code can handle is $\lfloor \frac{d_{min} - 1}{2} \rfloor = \lfloor \frac{3 - 1}{2} \rfloor = 1$.

3 Abstract Algebra Background

Coding Theory builds heavily upon abstract algebra. Consequently, this section gives the mathematical background necessary to understand certain abstract structures sufficiently for this introductory paper. See [W, 45-60] for more review.

3.1 Fields

Consider the sets $\mathbb{R}, \mathbb{C},$ and $\mathbb{Q}$ which have the usual arithmetic properties of addition, subtraction, multiplication and division. **Fields** are simply a generalization from sets with these properties. The formal set of properties for a field $\mathbb{F}$ is as follows.

- Existence of Additive Identity: There is an element $0 \in \mathbb{F}$ such that $0 + a = a = a + 0$ for all $a \in \mathbb{F}$.
- Existence of Additive Inverses: For all $a \in \mathbb{F}$, there is an element $b \in \mathbb{F}$ such that $a + b = 0 = b + a$.
- Commutativity of Addition: For all $a, b \in \mathbb{F}$, we have $a + b = b + a$.
- Associativity of Addition: For all $a, b, c \in \mathbb{F}$, we have $(a + b) + c = a + (b + c)$.

- Existence of Multiplicative Identity: There is an element $1 \in \mathbb{F}$ such that $1a = a = a1$ for all $a \in \mathbb{F}$.
- Existence of Multiplicative Inverses: For all $a \in \mathbb{F}\ \{0\}$, there is an element $b \in \mathbb{F}$ such that $ab = 1 = ba$.
- Commutativity of Multiplication: For all $a, b \in \mathbb{F}$, we have $ab = ba$.
- Associativity of Multiplication: For all $a, b, c \in \mathbb{F}$, we have $(ab)c = a(bc)$.
- Distributive Laws: For all $a, b, c \in \mathbb{F}$, we have $a(b + c) = ab + ac$ and $(a + b)c = ac + bc$.

Note that $\mathbb{Z}$ is not a field because it lacks existence of multiplicative inverses. That is, for $5 \in \mathbb{Z}$, there is no $b \in \mathbb{Z}$ such that $5b = 1$. Some less familiar fields which can be shown to possess all these properties are $\mathbb{Q}(x)$ (the set of ratios of polynomials with coefficients in $\mathbb{Q}$) and $n \times n$ diagonal matrices with entries from $\mathbb{Q}$.

3.2 Finite Fields

A **finite field** is simply a field with a finite number of elements. The simplest example of this is the integers modulo a prime p, represented $\mathbb{Z}/p\mathbb{Z}$ or $\mathbb{F}_p$. This set can be represented as the numbers 0 through $p - 1$.

The arithmetic operations of such fields work by 'wrapping around.' For example, adding 1 to $p - 1$ results in 0, and subtracting 1 from 0 results in $p - 1$. In general, for a result r from an arithmetic operation in $\mathbb{Z}$, the result of that operation in $\mathbb{F}_p$ is the remainder of $\frac{r}{p}$. So while $4+3 = 7$ in $\mathbb{Z}$, $4+3 = 2$ in $\mathbb{F}_5$ because $remainder(\frac{r}{p}) = remainder(\frac{7}{5}) = 2$. Formally, we write this $7 \equiv 2$ **mod** 5. Similarly, while $4 \cdot 4 = 16$ in $\mathbb{Z}$, $4 \cdot 4 = 1$ in $\mathbb{F}_5$ because $16 \equiv 1 \bmod 5$.

A slightly more complex finite field is the set of polynomials of degree less than some d with coefficients in some $\mathbb{F}_p$. There are p^d elements because each of the d coefficients can take on one of p values. Consequently, this field is represented $\mathbb{F}_{p^d}$. As an example, let us consider $\mathbb{F}_{2^4}$ which contains elements such as $a = x^3+x^2+1$ and $b = x^2 + x$. Addition occurs coefficient-wise, so the sum of these two elements would be $x^3 + x + 1$. Note that addition of the x^2 coefficient resulted in a 'wrapped around' result because it was reduced to its equivalent modulo 2. Results of multiplication are also modularly reduced to keep the degree less than d, but by reduction modulo

a fixed irreducible polynomial that degree. An **irreducible polynomial** is a non-constant polynomial that cannot be factored into the product of two non-constant polynomials. In $\mathbb{F}_{2^4}$, one such polynomial is $x^4 + x + 1$. Using $a \cdot b$ as an example of this reduction, we have

$$(x^3 + x^2 + 1)(x^2 + x) = x^5 + x^3 + x^2 + x \equiv \underline{x^3} \bmod (x^4 + x + 1)$$

because

$$x^5 + x^3 + x^2 + x = \underline{x^3} + (x)(x^4 + x + 1).$$

We will see an example of a code that takes its symbols from the field $\mathbb{F}_{2^m}$. In such real-world cases, transmitting the full polynomials would be terribly inefficient, so we abbreviate them to their coefficient lists and *interpret* them as polynomials. Thus $x^3 + x^2 + 1$ becomes 1101 and $x^2 + x$ becomes 0110.

3.3 Vector Spaces

A **vector space** is a set closed under vector addition and scalar multiplication. Each element of the vector space is called a **vector**, and is typically represented as a list of **coordinates**, usually enclosed by brackets or parenthesis. Each of these coordinates is a member of the field over which multiplication is defined. For example, the real vector space of dimension n, denoted $\mathbb{R}^n$, has elements $(.3, 0)$ for $n = 2$ and $(-5, \sqrt{2}, 0, 1)$ for $n = 4$.

A **linear subspace** is simply subset of a vector space which is still a vector space. It must contain the zero vector $\mathbf{0}$ and be closed under addition and scalar multiplication. A subspace of $\mathbb{R}^3$ (three-dimensional real space) would be a plane going through the origin, such as $\{(x, y, 0) \text{ for all } x, y \in \mathbb{R}\}$.

Every codeword and data string is a vector-like structure in that its symbols serve as coordinates of the larger object. However, because a code need not be algebraic (if its alphabet was $\{a - z\}$, addition would not really be defined), it is not necessarily a vector space. When the alphabet of a code is a field $\mathbb{F}$ and the code is closed under addition and multiplication by a constant, the code is a vector space and we call it a **linear code**. In this case, C is not just a subset of A^n, but a subspace of A^n. Throughout the rest of this paper, codewords and data strings will often be represented as vectors or vector-like structures $v = (v_0, v_1, ...)$, where the coordinates v_i are the letters, digits, or symbols that comprise the strings.

4 Codes

We now turn back to codes, and examine a few examples of how these mathematical techniques can be used for error detection and correction.

4.1 Repetition

The r-Repetition Codes are a simple but terribly inefficient set of codes used mostly for explanatory purposes. For initial data string $d = d_0 d_1 ... d_{n-1}$ over A^n where A is a field, the codeword $c = c_0 c_1 ... c_{n \cdot r - 1}$ is defined as d repeated r times. For example, for $r = 3, A = \mathbb{F}_2,$ and $d = 1011, c = 1011\ 1011\ 1011$. The length of the data is $k = 4$, so the code C has length $n = k \cdot r = 12$. It is the set of twelve digit binary strings c such that each section is identical: $c_0 c_1 c_2 c_3 = c_4 c_5 c_6 c_7 = c_8 c_9 c_{10} c_{11}$. Let a *position* be the set of places relative to a section. For example, the underlined places in $c = 10\underline{1}1\ 10\underline{1}1\ 10\underline{1}1$ represent the third position. Because changing one codeword to another requires changing every digit in a given position, $d_{min} = r$. This means that if less than r errors occur within a given position, we can detect the error. If less than $\frac{r}{2}$ errors occur in a given position, then over half the elements in that position will still be correct and a majority vote will determine what the correct value should be for all elements in that position. In other words, this code can correct up to $\lfloor \frac{r-1}{2} \rfloor$ errors. This corresponds with (1) because $r = d_{min}$. We can tell that the repetition code is linear because

- the zero vector consists of identical, repeated sections, and is therefore in the code,
- the alphabet A is a field,
- and adding two codewords together or multiplying a codeword by a constant from A results in a vector which also has identical sections, and is therefore a member of the code.

Recall that because the list of digits that comprise a codeword are mathematically being treated as elements of the vector space $\mathbb{F}_2$, addition and multiplication do not carry to the next digit, but wrap around to zero. Repetition codes, while able to correct an arbitrary number of errors by repeating more times, are too inefficient for much actual use because they require adding $2k$ data symbols to guarantee correction of an arbitrary error. [W, 2]

4.2 ISBN-10

Found on the back of any book, the International Standard Book Number (ISBN) is used to uniquely identify books around the world. We are going to focus on the ISBN-10 code. As might be guessed, it consists of ten-symbol codewords, $c_1c_2c_3c_4c_5c_6c_7c_8c_9c_{10}$. The first nine symbols are regular digits and are actually enough to identify a book, while the tenth is a redundant check symbol from $\{0-9, X\}$. It is computed as a weighted sum:

$$\text{Let } c'_{10} \equiv c_1 + 2c_2 + ... + 9c_9 \bmod 11.$$

$$\text{Then } c_{10} = \begin{cases} c'_{10}, & c'_{10} < 10 \bmod 11 \\ X, & c'_{10} \equiv 10 \bmod 11 \end{cases}.$$

For example, if the first nine digits are 123456789, $c'_{10} \equiv 285 \equiv 10 \bmod 11$, and $c_{10} = X$. If the first nine digits are 987654321, $c'_{10} \equiv 165 \equiv 0 \bmod 11$, and $c_{10} = 0$. This code is not a vector space because the elements do not admit multiplication by members of a field (the first nine values are from a different set than the last), so the code is not linear.

This code has the ability to detect all single digit and transposition errors, that is, all cases where one digit is mistyped or two digits are typed in transposed order. It does not offer the ability to correct errors. Rather, its strength lies in how, with only one redundant digit and a simple encoding scheme, it can detect all the most common errors. [W, 4]

4.3 Reed-Solomon Codes

Some particular applications which use Reed-Solomon codes are DVD's, CD's, barcodes, cell phones, satellite communications, digital television, and high speed modems. [RR] Denoted $RS(n, k)$ for codewords of length n derived from initial data of length k, they are non-binary, linear codes over some alphabet $A = \mathbb{F}_{2^m}$. Non-binary means that although it is used in electronic contexts where all data is a string of **bits** (ones and zeros), each 'character' or 'symbol' in a data string or codeword is not just a bit, but a grouping of m bits. Because grouping eight bits into a **byte** is standard in computer science, $m = 8$ would be a logical choice for many applications. Examples of such symbols in $\mathbb{F}_{2^m}$ are 101 for $m = 3$ and 11011001 for $m = 8$. Recall that because these are elements of a field, these values are not decimal numbers or binary values. That is, $101 \in \mathbb{F}_{2^3}$ is not one hundred one, nor is it the binary value

equivalent to five. More accurately, it represents the polynomial $1 \cdot x^2 + 0 \cdot x + 1$, where the coefficients are elements of $\mathbb{F}_2$ and multiplication over x will 'wrap around' by reduction modulo a fixed irreducible polynomial of degree 3.

Consider the data string $d = (d_0, d_1, ..., d_{k-1})$ of length k. Note that each of these symbols d_i in the data string is not a single digit, but an element of $\mathbb{F}_{2^m}$, and therefore has a multi-digit representation (i.e. '101'). Define

$$F(x) = d_0 + d_1 \cdot x + ... + d_{k-1} \cdot x^{k-1}, \text{ with } x \in \mathbb{F}_{2^m}.$$

The codeword corresponding to d is

$$c = (c_0, c_1, ..., c_{n-1}) = (F(0), F(\alpha), F(\alpha^2), ..., F(\alpha^{n-1}))$$

where $0, \alpha, \alpha^2, ..., \alpha^{n-1}$ are n elements of $\mathbb{F}_{2^m}$. By taking the equation associated with a coordinate of the codeword for each coordinate, we can form the system

$$\begin{aligned}
F(0) &= d_0 \\
F(\alpha) &= d_0 + d_1 \cdot \alpha + d_2 \cdot \alpha^2 + ... + d_{k-1} \cdot \alpha^{k-1} \\
F(\alpha^2) &= d_0 + d_1 \cdot \alpha^2 + d_2 \cdot \alpha^{2 \cdot 2} + ... + d_{k-1} \cdot \alpha^{2(k-1)} \\
&\vdots \\
F(\alpha^{n-1}) &= d_0 + d_1 \cdot \alpha^{n-1} + d_2 \cdot \alpha^{2(n-1)} + ... + d_{k-1} \cdot \alpha^{(n-1)(k-1)}.
\end{aligned}$$

This gives us a system of n equations in k variables (solving for the d_i's), but $n > k$, so the system is over-determined. This means that we can select a subset of k of these equations and still solve for d, as long as the coefficient matrix determined by those equations is nonsingular (invertible). Luckily, it has been proved that any selection of k equations from this set always generates such a matrix. Putting this system in matrix form we have $Md = c$ with solution $d = M^{-1}c$. Taking the first k equations as an example, we have

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & \alpha & \alpha^2 & \dots & \alpha^{k-1} \\ 1 & \alpha^2 & \alpha^{2 \cdot 2} & \dots & \alpha^{2(k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{k-1} & \alpha^{2(k-1)} & \dots & \alpha^{(k-1)(k-1)} \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{k-1} \end{bmatrix} = \begin{bmatrix} F(0) \\ F(\alpha) \\ F(\alpha^2) \\ \vdots \\ F(\alpha^{k-1}) \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{k-1} \end{bmatrix}.$$

This example uses only the first k symbols from the codeword to generate an equation which is solvable for the original data string. [KK, 257-260]

Now, when errors or erasures occur, they correspond to symbols of the codeword being wrong or missing, respectively. This means that any system which includes the missing or erroneous symbol c_i in its c vector will fail to correctly decode what d is. So just how many errors and erasures can occur before this method will fail?

Theorem. *Reed-Solomon Codes can correct t errors and v erasures as long as $2t + v \leq n - k$.*

Proof. As long as more systems decode to d than any other single data string, we can take a majority vote to successfully determine the correct data string. This will be the case when the erroneous codeword is closer to (has a smaller distance from) the original codeword than any other $c \in C$. This suggests making use of (1). We would rather not check every pair of codewords to determine the minimum distance, so instead we will use the code's algebraic properties. Because $RS(n, k)$ is a linear code, for $a, b \in C$ we know also that $a - b \in C$. Because identical corresponding symbols will cancel to zero in the subtraction,

$$dist(a, b) = dist(a - b, \mathbf{0}) = weight(a - b).$$

So the minimum distance between every pair of codewords equals the weight of some other codeword. This simplifies the problem because now we only need to find the minimum weight of C, that is, the least number of non-zero coordinates in a codeword. Now the generator polynomial for the c_i's, $F(x) = d_0 + d_1 \cdot x + ... + d_{k-1} \cdot x^{k-1}$, has degree of at most $k - 1$, and therefore has at most $k - 1$ roots. So the codeword $(F(0), F(\alpha), F(\alpha^2), ..., F(\alpha^{n-1}))$ has at most $k - 1$ coordinates which are zero. [G, 2] Thus at least $n - (k - 1) = n - k + 1$ coordinates are non-zero, which means

$$weight_{min}(RS(n, k)) = d_{min}(RS(n, k)) \geq n - k + 1.$$

Plugging this into (1) yields

$$t \leq \lfloor \frac{n - k + 1 - 1}{2} \rfloor = \lfloor \frac{n - k}{2} \rfloor \implies 2t \leq n - k.$$

When v erasures occur, up to v of the non-zero coordinates are erased, thereby decreasing the effective minimum weight and distance. Consequently, for a system with

v erasures, we adjust our previous equation by replacing n with $n - v$:

$$2t \leq n - v - k \implies 2t - v \leq n - k.$$

This gives the maximum number of errors and erasures that can occur such that $RS(n, k)$ can still properly decode. □

Now the method of correcting by decoding all possible systems and comparing the results, while mathematically sound, is temporally inefficient. It turns out to be the simpler way of understanding the code, but know that in practice, there is a different, more efficient way of doing this [KK, 262].

The fact that $RS(n, k)$ has minimum distance $n - k + 1$ is actually optimal in a way. [KK, 261] That is, relative to fixed values of n and k, it is impossible to design a code with a smaller d_{min}. This makes Reed-Solomon codes a good choice for many applications. In particular, consider circumstances where **burst error**, that is, high noise occurring over several successive data bits, occurs. Because any number of error bits within a symbol of $\mathbb{F}_{2^m}$ will only count as one error, Reed-Solomon codes are particularly suited for these situations. On the other hand, a single bit of a symbol being wrong means the whole m-bit symbol is erroneous. So while Reed-Solomon codes are optimal at their symbol-wise error correction rate, they are not optimal from a bit-wise perspective. This makes Reed-Solomon codes a bad choice for other applications.

5 Conclusion

Thus, while they are a good example and are widespread, Reed-Solomon codes remain only one of the many codes in use. Some of the other codes may be used because they are more efficient at correcting certain error patterns, just as Reed-Solomon codes are efficient for burst errors. Other codes may be preferable because the simplicity of their underlying algorithms makes them faster to encode and decode. We saw that ISBN codes are one case where a simple algorithm is preferable and good enough. Sometimes different codes are layered on top of one another to correct a combination of error patterns well, or to balance simplicity with efficiency. Whatever code or codes are used, the mathematics beneath is what is responsible for the integrity of the data transfer.

References

[C] Kenyon College. "Introduction to Coding." Accessed April 17, 2017. http://www2.kenyon.edu/Depts/Math/Aydin/Teach/Fall06/328/Intro.pdf.

[G] Venkatesan Guruswami. "Reed-Solomon, BCH, Reed-Muller, and concatenated codes" Accessed April 17, 2017. https://www.cs.cmu.edu/∼venkatg/teaching/codingtheory/notes/notes6.pdf.

[H] Raymond Hill. *A First Course in Coding Theory*. New York: Oxford University Press, 1991.

[KK] Dave K. Kythe and Prem K. Kythe. *Algebraic and Stochastic Coding Theory*. Boca Raton: CRC Press, 2012.

[RR] Martyn Riley and Iain Richardson. "Reed-Solomon Codes." Accessed April 17, 2017. https://www.cs.cmu.edu/∼guyb/realworld/reedsolomon/reed_solomon_codes.html.

[W] Judy L. Walker. *Codes and Curves*. Providence, RI: American Mathematical Society, 2002.