MATH UN4042 Final Project
Computational Aspects of Algebraic Coding Theory

Ben Flin (brf2117), Max Helman (mhh2148), Ben Sherwin (bes2169)

April 23, 2021

1 Algebraic Coding Theory

1.1 Big Picture

The world of Computer Science and Electrical Engineering is binary. This is for good reason: consider sending a voltage over a noisy channel. Noise can and will disrupt analog signals, at least to a certain degree, but we do not know how much it will affect the signal by. Therefore, modern communication systems are digital: if the voltage received is above some threshold, typically $2V$, we interpret it as a logical 1, and if it is below some other threshold, typically $0.8V$, we interpret it as a logical 0 (if it is in between the two, we request that the signal is sent again). This model of communication is far more robust and noise-tolerant, but it comes at a cost: we must encode everything as 1's and 0's, which means our messages will be much longer. We also must come up with efficient algorithms for encoding and decoding, and we need to figure out how to handle errors, which still might occur.

Designing effective codes for such a scenario is the driving motivation behind Coding Theory, which has been around since 1948 when Claude Shannon authored his paper "A Mathematical Theory of Information". The field was further developed when Richard Hamming of Bell Labs became frustrated that his codes could not recover from errors caused by noise, and it is an active area of research today. Algebra in particular has been an incredibly important tool to design effective coding schemes, and the algebraic lens will be what we place our central focus on as we work our way up to developing algorithms for modern coding schemes, with a particular emphasis on BCH Codes.

1.2 Basic Principles and Maximum-Likelihood Decoding

Suppose we want to send some m-digit message across a noisy channel. To do so, we encode it into a n-digit codeword, send the codeword across the channel, and then decode it back into either a m-digit message or an "error received" message, which occurs when there is a change in one or more bits of the codeword. We want n to be as small as possible for obvious reasons, but we simultaneously want to ensure redundancy.

A naive approach would be to send a message several times and compare each bit with the other messages. We could perhaps send three messages and then take the bitwise majority, but this requires $n = 3m$, and we can do much better than this. A much more efficient scheme is to use a parity check bit: there are $2^7 = 128$ ASCII characters, so they can be encoded using seven bits, but if we add an eighth bit to the beginning as a "parity check bit" we can easily detect errors. The parity check bit is set such that the total number of 1's in the 8-tuple is even. Thus, if a bit is flipped, the recipient can easily check that an error has occurred since there will now be an odd number of 1's, and it can request the message to be sent again. However, this has its limitations: we cannot detect multiple errors, since if two errors occur, there will still be an even number of 1's. We also cannot correct errors: we know if an error has occurred, but we have no idea where.

We now define a few terms that will be used to quantify the properties of codes as we build more sophisticated machinery. Suppose we have codes x and y:

1. The Hamming Distance $d(x, y)$ between x and y is the number of bits where x and y differ.
2. The Minimum Distance is the minimum of $d(x, y)$ between all distinct codewords x and y.
3. The Weight $w(x)$ is the number of 1's in x.

The final prerequisite definition is the principle of Maximum-Likelihood Decoding. This principle bakes in the likelihood of errors. We observe that if we transmit a message of length $n = 10,000$ where the probability (per bit) that no error occurs is $p = .99$, the probability that at least one error occurs is $(.99)^{10,000} = 0.00005$. So errors are likely a fact of life - how should we decode them? We always assume that fewer errors occur when comparing a received message to possible codewords. For example, if we incorrectly receive 111, it is more likely that we meant to send 011 than 010. In fact, using this principle, we correct to the codeword with the Minimum Distance from the received message.

1.3 Block Codes

How, then, can we develop efficient error-detecting and error-correcting codes now that we understand the principle of Maximum-Likelihood Decoding? We need some more definitions first. We say a code is a (n, m)-block code if the information to be encoded can be divided into blocks of m binary digits, each of which are encoded into n binary digits (essentially, blocks act as messages here). A block code:

1. Has an injective encoding function $E : \mathbb{Z}_2^m \to \mathbb{Z}_2^n$.
2. Has a surjective decoding function $D : \mathbb{Z}_2^n \to \mathbb{Z}_2^m$.

We also note that computationally, weights are easier to compute than Hamming Distances. In order to determine error-detecting and correcting capabilities for a block code, we need to examine the Minimum Distance of the code. If x and y are codewords and $d(x, y) = 1$, then x can be changed to y with a single error and without an error message, but if $d(x, y) = 2$ a single error cannot change x to y. However, if we receive some message z such that $d(x, z) = 1$ and $d(y, z) = 1$, then then decoder cannot decide if it should correct to x or y. What happens if we set $d_{min} \geq 3$? We can now detect and correct any single error. More generally, if we have a code with $d_{min} = 2n + 1$, the code can correct up to n errors and detect up to $2n$ errors.

1.4 Linear Coding

With a little bit of Group Theory and Linear Algebra, we can add some additional structure to our codes that will make our lives much easier in the long run. A group code is a code that is also a subgroup of $\mathbb{Z}_2^n$ (under addition, which of course is done $\in \mathbb{Z}_2$). Since every code is its own inverse and 0 is therefore a codeword, we only need to check for closure under addition.

We can generate group codes easily with Linear Algebra. We define the kernel of a matrix $H \in M_{m\times n}(\mathbb{Z}_2)$, denoted $kerH$, to be the set of all binary n-tuples x such that $Hx = 0$. An amazing property of $kerH$ is that it is a group code: clearly $H0 = 0$, and every other element in $kerH$ is its own inverse. A code is said to be a linear code if it is determined by the null space of some matrix $\in M_{m\times n}(\mathbb{Z}_2)$. An especially nice property of linear codes is that to see if some x is a codeword, you just need to multiply by H, since if $Hx = 0$, then $x \in kerH$ and it is indeed a codeword. Otherwise, x is not a codeword.

We now need to systematize the generation of linear coding and decoding, which can be done by carefully choosing H. Take H to be a $m \times n$ matrix with entries in $\mathbb{Z}_2$ and $n > m$. If the last m columns of the matrix are the $m \times m$ identity matrix I_m, then H is called a Canonical Parity-Check Matrix (precisely, $H = (A|I_m)$, where A is a $m \times (n - m)$ matrix). Each such matrix has an associated $n \times (n - m)$ Standard Generator Matrix $G = \begin{pmatrix} I_{n-m} \\ A \end{pmatrix}$. With some mechanical matrix manipulation that is left as an exercise to the reader, we can verify that an x satisfying $Gx = y$ exists $\iff Hy = 0$.

This is an incredible result, since if we are given a message block x to encode, G allows us to quickly encode it into a linear codeword y. We also observe that the rows in H represent the parity checks on certain bit positions in a tuple. Perhaps noteworthy, however, is that if H is a canonical parity-check matrix, $kerH$ consists of $x \in \mathbb{Z}_2^n$ with arbitrary bits in the first $n-m$ positions and then bits determined by $Hx = 0$ in the last m positions. In fact, each of these final bits is a parity check bit for some of the first bits, and as such, H creates a $(n, n - m)$-code. Here, the first $n - m$ bits in x are called information bits and the last m bits are called check bits.

We can now easily generate linear codes. If G is a $n \times k$ Standard Generator Matrix, $C = \{y|Gx = y, x \in \mathbb{Z}_2^k\}$ is a (n, k)-block code and C is a group code. If G is the corresponding Standard Generator Matrix to a Canonical Parity-Check Matrix H, then we observe $HG = 0$ (again via manual matrix multiplication left as an exercise to the reader). Finally, if we let C be the code generated by G, $y \in C \iff Hy = 0$, so C is a linear code with Canonical Parity-Check Matrix H.

1.5 Decoding Schemes

Now that we have shown how to easily generate linear codes that detect and correct errors, we need to find an efficient decoding scheme. The naive approach to decoding involves comparing a received n-tuple to all of the codewords and then determining the closest codeword, but this is not optimal due to its time complexity for codes containing many codewords. We need to define the syndrome of a message to properly discuss decoding: if H is a $m \times n$ matrix and $x \in \mathbb{Z}_2^n$, the syndrome of x is Hx.

If H determine a linear code and x is a received n-tuple, we can write x as $x = c + e$, where c is the transmitted codeword and e is the transmission error. We see via the mechanics of matrix multiplication that the transmission of the received codeword x is also the syndrome of its error e, and as such, the syndrome of a received word depends only on the error rather than on the transmitted codeword. A nice corollary here is that if the syndrome of x is 0, no detectable error has occurred, so if we set the Minimum Distance to a large enough value, we can be fairly confident that no error occurs if the syndrome of x is 0.

Our next observation is that a linear code C is a subgroup of $\mathbb{Z}_2^n$. We can actually use the cosets of C in $\mathbb{Z}_2^n$ to efficiently implement maximum-likelihood decoding. If C is a (n, m)-linear code, any coset of C in $\mathbb{Z}_2^n$ is written as $x + C$, where

$x \in \mathbb{Z}_2^n$. As such, there are 2^{n-m} distinct cosets of C in $\mathbb{Z}_2^m$ (the proof involves Lagrange's Theorem and is beyond the scope of this survey). We let x be the original codeword sent, r be the received n-tuple, and e be the transmission error. We know $r = e + x$, and with just a tad of algebraic manipulation we see $x = e + r$. This implies that r is an element of the coset $e + C$. By the principle of Maximum Likelihood Decoding, we want e to be small (i.e. it has the least weight). We can quantify this by selecting the n-tuple of minimum weight in each coset; this is what is known as a coset leader. After determining a coset leader for each coset, all that is needed to decode is to calculate $r + e$ to get x. We note that two n-tuples are in the same coset $\iff$ they have the same syndromes. Computationally, we can associate a syndrome with each coset and build a lookup table with a syndrome and its coset leader. This is known as the decoding table and avoids repeated calculations, although its setup time is nontrivial.

2 Polynomial Codes

2.1 Overview of Polynomial Codes

Let $\varphi : \mathbb{Z}_2^k \to \mathbb{Z}_2^n$ be a binary (n, k)-block code. We say φ is a cyclic code if for every codeword $(a_1, ..., a_n)$, the cyclically shifted n-tuple $(a_n, a_1, ..., a_{n-1})$ is also a codeword. Cyclic codes are of particular interest to us since digital systems are implemented in such a way that bitwise operations, including bit shifts, are incredibly cheap operations. Since transmitters and receivers are usually not particularly powerful computers, we want to keep the computational overhead of encoding and decoding as low as possible, which can be done by using shift registers; this means that cyclic codes are certainly up for the challenge.

We want to generate cyclic linear codes easily, which can be done with finite fields and polynomial rings over $\mathbb{Z}_2$. In fact, any binary n-tuple can be interpreted as a polynomial in $\mathbb{Z}_2[x]$. This is done by a mapping $\Phi : \{0, 1\}^n \to \mathbb{Z}_2[x]$ where $\Phi((a_0, a_1, ..., a_{n-1})) = a_0 + a_1x + ... + a_{n-1}x^{n-1}$. In fact, this works for all 2^n polynomials in $\mathbb{Z}_2[x]$ of degree $< n$. Since this function is a bijection on this restricted codomain, we can assign a binary encoding to any polynomial in $\mathbb{Z}_2[x]$.

If we fix a nonconstant polynomial $g(x) \in \mathbb{Z}_2[x]$ such that $deg(g(x)) = n - k$, we can define a (n, k)-code C by finding a k-tuple to be encoded, changing it to its polynomial representation $f(x) \in \mathbb{Z}_2[x]$, and multiplying it by $g(x)$. The codewords in C are actually all polynomials in $\mathbb{Z}_2[x]$ of degree $< n$ that are divisible by $g(x)$. A code obtained in this manner is called a polynomial code.

We also know that Rings of polynomials have some nice underlying structure, and it would be nice if we could bring said structure into our coding scheme. We know that $x^n - 1 = (x - 1)(x^{n-1} + ... + x + 1)$, so the factor ring $R_n = \mathbb{Z}_2[x]/(x^n - 1)$ is the ring of polynomials of the form $f(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1}$ where $x^n = 1$. We can use this to interpret a linear code as a subset of $\mathbb{Z}[x]/(x^n - 1)$, and in fact, there are methods for converting codes from other fields into equivalent binary representations.

We also note that $\mathbb{Z}_2^n \cong R_n$ as vector spaces, and that we can interpret a linear code as a subset of $\mathbb{Z}[x]/(x^n - 1)$. Now, here comes the amazing fact made possible by the additional structure given by Rings of polynomials: a cyclic shift of a n-tuple is equivalent to polynomial multiplication. If $f(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1}$ is a code polynomial in R_n, then $xf(x) = a_{n-1} + a_0x + ... + a_{n-2}x^{n-1}$ is the cyclically-shifted word that we obtained by multiplying $f(x)$ by x.

Proof. A linear code $C \in \mathbb{Z}_2^n$ is cyclic $\iff$ it is an ideal in $R_n = Z[x]/(x^n - 1)$.

If C is a linear cyclic code and $f(x) \in C$, then $xf(x) \in C$; this directly implies that $\forall k \in \mathbb{N}$, $x^k f(x) \in C$. We also know that C is a linear code, so any combination of the codewords $f(x), xf(x), ..., x^{n-1}f(x)$ must also be a codeword, so for every polynomial $p(x)$, $p(x)f(x) \in C$. By definition, C is an ideal.
Now we suppose C is an ideal $\in \mathbb{Z}_2[x]/(x^n + 1)$. If $f(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1}$ is a codeword $\in C$, then $xf(x)$ is also a codeword in C, so $(a_1, ..., a_{n-1}, a_0) \in C$. □

As a result of this theorem, we observe that knowing the ideals of R_n is equivalent to knowing the linear cyclic codes in $\mathbb{Z}_2^n$. We can actually find the ideals in R_n pretty easily: $\varphi : \mathbb{Z}_2[x] \to R_n$ defined as $\varphi(f(x)) = f(t)$ is a surjective homomorphism with kernel $(x^n - 1)$. Therefore, every ideal C in R_n is of the form $\varphi(I)$, where I is an ideal in $\mathbb{Z}_2[x]$ containing $(x^n - 1)$. Furthermore, we know that every ideal I in $\mathbb{Z}_2[x]$ is a principal ideal (since $\mathbb{Z}_2$ is a field), $I = g(x)$ for some unique monic polynomial in $\mathbb{Z}_2[x]$, and since $x^n - 1$ is contained in I, $g(x)$ must divide $x^n - 1$. Therefore, every ideal C in R_n is of the form $C = (g(t)) = \{f(t)g(t) | f(t) \in R_n, g(x) | (x^n - 1)\}$. The unique monic polynomial of the smallest degree that generates C is called the minimal generator polynomial of C.

We can get a generator matrix for a (n, k)-code C in a similar way to which we encode the elements t^k. If we let $x^n - 1 = g(x)h(x) \in \mathbb{Z}_2[x]$ such that $g(x) = g_0 + g_1x + ... + g_{n-k}x^{n-k}$ and $h(x) = h_0 + h_1x + ... + h)kx^k$, then:

$$G = \begin{bmatrix} g_0 & 0 & \dots & 0 \\ g_1 & g_0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_{n-k} & g_{n-k-1} & \dots & g_0 \\ 0 & g_{n-k} & \dots & g_1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & g_{n-k} \end{bmatrix}$$

$$H = \begin{bmatrix} 0 & \dots & 0 & 0 & h_k & \dots & h_0 \\ 0 & \dots & 0 & h_k & \dots & h_0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ h_k & \dots & h_0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

are generator and parity-check matrices for C, respectively, provided $C = (g(t))$. We also observe that $HG = 0$.

2.2 The Fast Fourier Transform

While on the topic of polynomial multiplication, it would be inexcusable to leave out a discussion of the Fast Fourier Transform, an incredibly important algorithm for multiplying two polynomials. The Fast Fourier Transform has revolutionized and defined the field of signal processing, and it has tremendous applications in Electrical Engineering. We see that the product of two degree-n polynomials is a polynomial of degree $2n$. Computing c_i, the ith coefficient in the multiplied polynomial, takes time $\mathcal{O}(n)$, so computing all coefficients should take time $\mathcal{O}(n^2)$. However, we can do better! This hinges on the fact that a degree n polynomial is uniquely characterized by its values at any $n+1$ distinct points. Therefore, $f(x) = a_0 + a_1x + ... + a_nx^n$ can be specified either as its coefficients $a_0, a_1, ..., a_n$ or as the values $f(x_0), f(x_1), ..., f(x_n)$.

If we use the value representation of polynomials, we can multiply polynomials in time $\mathcal{O}(n)$ since the value representation of the product has $2n+1$ points, each of which are $f(x)g(x)$. However, we need to convert back to coefficient representation, which is a process known as interpolation. We want faster evaluation than $\mathcal{O}(n^2)$, which can be done with a clever choice of $x_0, ..., x_{n-1}$ since the computations required by individual points overlap.

Algorithm 1: Polynomial Multiplication

Result: The products of two degree-d polynomials $f(x)$ and $g(x)$
Select points $x_0, x_1, ..., x_{n-1}$ where $n \geq 2d+1$
Compute $f(x_0), f(x_1), ..., f(x_n)$ and $g(x_0), g(x_1), ..., g(x_n)$
Compute $f(x_k)g(x_k)\ \forall k \in [0, ..., n-1]$
Recover $f(x)g(x) = c_0 + c_1x + ... + c_{2d}x^{2d}$
return $f(x)g(x)$

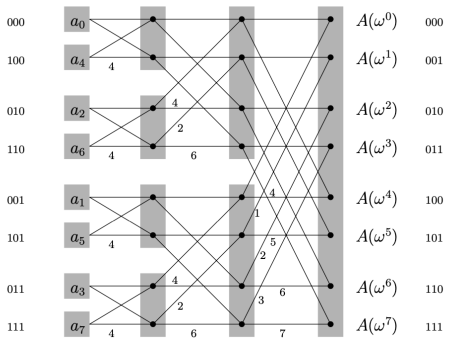
We now observe that if we pick n points such that they contain $n/2 \pm$ pairs, there is a lot of overlapped computation since even powers of x_i coincide with $-x_i$. We observe $f(x) = f_e(x^2) + xf_o(x^2)$, where $f_e(x)$ contains all of the even powers and $f_o(x)$ contains all of the odd powers, and both are of degree $\leq n/2 - 1$. So, we just need to evaluate $f_e(x)$ and $f_o(x)$ at $n/2$ points each; this gives us a divide and conquer algorithm, which is more efficient. However, we need to use complex numbers for the computation, since otherwise only the points will not be plus/minus pairs beyond the first level of recursion. We actually end up using nth roots of unity $1, \omega, \omega^2, ..., \omega^{n-1}$, since if n is a power of 2 (and even if it is not, we round up to the smallest power of 2, since this will save us time in the long run), the nth roots are $\pm$ pairs since $\omega^{n/2+j} = -\omega^j$ and squaring them yields $(n/2)$nd roots of unity, which have the same properties.

Algorithm 2: The Fast Fourier Transform

Result: The value representation $f(\omega^0), ..., f(\omega^{n-1})$
if $\omega = 1$ **then**
  return $f(1)$
end
Express $f(x)$ as $f_e(x^2) + xf_o(x^2)$
Call $FFT(f_e, \omega^2)$ to evaluate f_e at even powers of ω
Call $FFT(f_o, \omega^2)$ to evaluate f_o at even powers of ω
for $j = 0$ *to* $n-1$ **do**
  Set $f(\omega^j) = f_e(\omega^{2j}) + \omega^j f_o(\omega^{2j})$
end
return $f(\omega^0), ..., f(\omega^{n-1})$

This is amazing and beautiful and revolutionary, since we can switch from coefficients to values in time $\mathcal{O}(nlogn)$. Interpolation is actually nearly the exact same algorithm, but just with ω^{-1} instead of ω! This relies on a tremendous deal of

linear algebra; for a full derivation, please see Section 2.6.3 of Algorithms by Dasgupta, Papadimitirou, and Vazirani. Here is a graphic that illustrates the computation path of the Fast Fourier Transform, which gives good intuition as to how it can be directly implemented in hardware (the figure is from the same textbook):



3 BCH Codes

3.1 Big Picture

BCH codes were discovered independently by A. Hocquenghem in 1959 and R.C. Bose and D.V. Ray-Chaudhuri in 1960. They are incredibly prevalent today, 60 years later: they European and Transatlantic communications systems both use BCH codes, as do satellite communications, CD players, and quantum-resistant cryptography. Where BCH codes really shine is that we know of efficient error-correction algorithms for them. Other advantages of BCH codes is that they can be decoded easily via an algebraic method known as syndrome decoding; this simplifies the hardware design of the decoder and makes them feasible on small low-power hardware. BCH codes typically encode words of length 231; a polynomial of degree 24 is used to generate this code, and since $231 + 24 = 255 = 2^8 - 1$, this yields a $(255, 231)$-block code. Such a BCH code can detect 6 errors and correct 3, giving it a failure rate of 1 in 16 million.

The general idea behind BCH codes is to choose a generator polynomial of the smallest possible degree while maintaining large error detection and error correction capabilities; BCH codes are also cyclic. Define $d = 2r + 1$ for some $r \geq 0$ and let ω be a primitive nth root of unity over $\mathbb{Z}_2$, an element of $\mathrm{GF}(2^n)$, and let $m_i(x)$ be the minimal polynomial over $\mathbb{Z}_2$ of ω^i. If $g(x) = lcm(m_1(x), ..., m_{2r}(x))$, then the cyclic code $(g(t))$ in R_n is called the BCH code of length n and distance d.

Proof. The following three statements are equivalent if $C = (g(t))$, a cyclic code in R_n:

1. The code C is a BCH code whose minimum distance is at least d.
2. A code polynomial $f(x) \in C \iff f(\omega^i) = 0$ for $1 \leq i < d$.
3. The matrix

$$A = \begin{bmatrix} 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{(n-1)(2)} \\ 1 & \omega^3 & \omega^6 & \dots & \omega^{(n-1)(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{2r} & \omega^{4r} & \dots & \omega^{(n-1)(2r)} \end{bmatrix}$$

is a parity-check matrix for C.

We know that if $f(x) \in C$ that $g(x) | f(x) \in \mathbb{Z}_2$. Therefore, for $i = 1, ..., 2r$, we know $f(\omega^i) = 0$ since $g(\omega^i) = 0$ (and the converse holds as well), so $f(x)$ must be a codeword.
If we choose some $f(t)$ in R_n and pick its corresponding binary n-tuple, $Hx = (f(\omega), f(\omega^2), ..., f(\omega^{2r}))^T$, which is equal to 0 when $f(x) \in C$ by definition, so H is a parity-check matrix for C.
A code polynomial $f(x) \in C \iff f(\omega^i) = 0$ for $i = 1, ..., 2r$. The minimal polynomial such that this holds is $g(x) = lcm(m_1(x), ..., m_{2r}(x))$, so $C = (g(x))$. □

3.2 Primitive Narrow-Sense and General BCH Codes

We can construct what is known as a Primitive Narrow-Sense BCH code over $GF(q)$ with code length $n = q^m - 1$ and distance $\geq d$ when given a prime number p and a prime power p^m with $m, d \in \mathbb{Z}_{\geq 0}$ and $d \leq q^m - 1$. Let a be a primitive element of $GF(q^m)$. For any positive integer i, let $m_i(x)$ be the minimal polynomial with coefficients in $GF(q)$ of a^i. The

generator polynomial of the BCH code is the LCM of the minimal polynomials $\{m_1(x), ..., m_{d-1}(x)\}$, and we also know that $g(x) \in GF(q)[x]$ and $g(x)|x^n - 1$, so the polynomial code defined by $g(x)$ is a cyclic code.

General BCH codes differ from Primitive Narrow-Sense BCH Codes by relaxing the requirement that a must be a primitive element of $GF(q^m)$. Thus, the code length is $\mathcal{O}(a)$ rather than $q^m - 1$; if we play our cards right, this is a great improvement! We also see that the consecutive roots of the generator polynomial are $a^c, ..., a^{c+d-2}$ rather than $a, ..., a^{d-1}$. Fix a finite field $GF(q)$ where q is a prime power and choose positive integers m, n, d, c such that $2 \leq d \leq n$, $gcd(n, q) = 1$, and m is the multiplicative order of q (mod n). Then, let a be a primitive nth root of unity in $GF(q^m)$ and let $m_i(x)$ be the minimal polynomial over $GF(q)$ of a^i for all i. The generator polynomial of the BCH code is defined as the LCM of the minimal polynomials, this time $\{m_c(x), ..., m_{c+d-2}(x)\}$.

We observe that the generator polynomial of a BCH code has degree $\leq (d-1)m$; if $q = 2$ and $c = 1$, it has degree $\leq dm/2$.

3.3 Encoding

We know that any polynomial that is a multiple of the generator polynomial is a valid BCH codeword; therefore, BCH encoding mainly consists of finding some polynomial that contains the generator polynomial as one of its factors. Non Systematic Encoding is a naive encoding algorithm in which the encoding is computed as the product of some arbitrary polynomial and the generator. The arbitrary polynomial can, in fact, be chosen using the symbols of the message as coefficients, and using the Fast Fourier Transform, we can encode in time $\mathcal{O}(n \log n)$, where the polynomial is of degree n.

Systematic Encoding improves upon this process. A systematic code involves the message appearing verbatim somewhere in the codeword, so it involves first embedding the message polynomial within the codeword polynomial, and then adjusting the coefficients of the remaining non-message terms to ensure that the overall polynomial $s(x)$ is divisible by $g(x)$. Doing so leverages that subtracting the remainder from a dividend results in a multiple of the divisor. If we take $p(x)$ and multiply it by x^{n-k}, we observe $p(x)x^{n-k} = q(x)g(x) = p(x)x^{n-k} - r(x)$; the deeper underlying theory behind this involves cyclic redundancy checks, which are beyond the scope of this survey. However, what's incredible about systemic codes is that after error correction, the receiver can recover the original message by discarding everything after the first k coefficients; this is incredibly efficient!

3.4 Decoding

BCH Codes have a simple general algorithm for decoding, but each step is somewhat more complex. The general barebones algorithm is as follows:

Algorithm 3: Decoding BCH Codes

Result: A Decoded Code
Calculate the syndromes s_j of the received vector
Determine the number of errors t and the error locator polynomial from the syndromes
Calculate the roots of the error locator polynomial to find error locations X_i
Calculate the error values Y_i at these locations and correct the errors
return *Code with errors corrected*

It is possible for the decoding algorithm to determine that there are too many errors and that the code cannot be corrected. In this case, it will request for the code to be sent again. We start by calculating the syndromes, in which we consider R (the received code) as a polynomial and evaluate it at $a^c, ..., a^{c+d-2}$. As such, the syndromes are:

$$s_j = \{R(a) = C(a^j) + E(a^j) | j \in [c, c + d - 2]\}$$

Since a^j are the zeroes of $g(x)$ (which divides $C(x)$), we know $C(a^j) = 0$. Therefore, examining the syndrome values isolates the error vectors, and if there is no error, $s_j = 0 \ \forall j$, so we are done and can skip the next steps.

However, suppose there are nonzero syndromes. This tragically means we have errors, although we will see that it turns out to be not so tragic after all in most cases. The decoder must deal with the errors by figuring out how many errors there are, and what their magnitudes are. In the case of a single error, we can write $E(x) = zx^i$, where z is the magnitude of the error and i is its location. This means that the first two syndromes are $s_c = za^{ci}$ and $s_{c+1} = za^{c+1}i = a^i s_c$. However, it there are two or more errors, we have $E(x) = z_1 x^{i_1} + z_2 x^{i_2} + ...$, and it is less clear how to solve for z_k and i_k. We need a locator polynomial $\Lambda(x) = \prod_{j=1}^{t}(xa^{i_j} - 1)$. One way of doing this is with the Peterson-Gorenstein-Zierler algorithm:

Algorithm 4: Peterson-Gorenstein-Zierler Algorithm

Result: Coefficients $\lambda_0, ..., \lambda_v$ of Locator Polynomial $\Lambda(x)$

Generate a $S_{v\times v}$ matrix with elements that are syndrome values

$$S_{v\times v} = \begin{bmatrix} s_c & s_{c+1} & \cdots & s_{c+v-1} \\ s_{c+1} & s_{c+2} & \cdots & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ s_{c+v-1} & s_{c+v} & \cdots & s_{c+2v-2} \end{bmatrix}$$

Generate a $c_{v\times 1}$ vector:

$$C_{v\times 1} = \begin{bmatrix} s_{c+v} \\ s_{c+v+1} \\ \vdots \\ s_{c+2v-1} \end{bmatrix}$$

Generate a vector Λ of unknown polynomial coefficients

$$\Lambda_{v\times 1} = \begin{bmatrix} \lambda_v \\ \lambda_{v-1} \\ \vdots \\ \lambda_1 \end{bmatrix}$$

Consider the equation $S_{v\times v}\Lambda_{v\times 1} = -C_{v\times 1}$

if $det(S_{v\times v}) \neq 0$ **then**
  Solve for unknown Λ
end
else
  We cannot invert $S_{v\times v}$, so consider $S_{(v-1)\times(v-1)}$ and try again
end
return Λ

Now, we must find the roots of $\Lambda(x)$. Factoring is generally a hard computational problem, but we have a fast algorithm for finding roots of a polynomial over a finite field called Chien Search (in fact, BCH codes are the primary use of this algorithm). Chien search makes use of the generator in a finite field, and tests the elements in the generator's order. Thus, it only requires multiplication by constants and addition as its operations. Exhaustive search will also suffice for $GF(p^n)$ where p^n is small, but it requires multiplication of two variables. Chien search is more useful when implemented in hardware, though, since the multiplications consist of one variable and one constant rather than two variables in exhaustive search; this is computationally cheaper. $\Lambda(x)$ will ultimately factor into $(a^{i_1}x - 1)(a^{i_2}x - 1)...(a^{i_v}x - 1)$, where the powers of the primitive element are the positions where the errors occured. We also get the zeroes: $a^{-i_1}, ..., a^{-i_v}$.

We now need to determine the error values at their respective locations, which are used to correct the received values at the locations and recover the original codeword. If we have a binary code, this is super easy, since we just need to flip the bits at the error locations. In the non-binary case, we can solve a linear system of equations to determine the error weights, or we can use the Forney Algorithm, which is more efficient.

Algorithm 5: Forney Algorithm

Result: Error Weights e_j

$S(x) = s_c + s_{c+1}x + s_{c+2}x^2 + ... + s_{c+d-2}x^{d-2}$

$v \leq d - 1$ $\lambda_0 \neq 0$ $\Lambda(x) = \lambda_0 \prod_{k=0}^{v}(a^{-i_k}x - 1)$ We define an error evaluatior polynomial $\Omega(x) = S(x)\Lambda(x) \bmod x^{d-1}$

We denote $\frac{d\Lambda}{dx}$ as $\Lambda'(x)$

for *All errors* e_k **do**
  $e_k = -\frac{a^{i_k}\Omega(a^{-i_k})}{a^{ci_k}\Lambda'(a^{-i_k})}$
end

The Forney Algorithm relies on Lagrange Interpolation and Generating Functions, both of which are beyond the scope of this survey. If a simpler algorithm is preferred that is still more efficient than solving a system of linear equations, there is an extended version of the Euclidean Algorithm that will do. Regardless, after the magnitudes and locations errors are found, correct them by subtracting their values from their locations. Thus concludes our discussion of decoding BCH Codes.

3.5 Simulation of BCH Coding

We wrote a Python program that given a message, encodes the message into a 255 bit BCH code, introduces errors as specified by the user, and then detects and corrects errors using the Peterson-Gorenstein-Zierler Algorithm. It works well in practice and is computationally efficient, as it represents polynomials in $\mathbb{Z}_2$ and $\mathrm{GF}(2^n)$ as binary integers. The biggest challenges were doing bitwise manipulation on polynomials in the finite field and creating a Galois Field numerical type that can efficiently calculate addition, multiplication, and inverses in $GF(2^n)$ for any n. We also used the Numpy library for matrix multiplication, although all other functionality was written by us. This did not work for matrix inverses and determinants due to limitations of Numpy, so we wrote our own implementation of those algorithms, namely Laplace expansions and the method of minors and cofactors. Our code is extensively documented and can be viewed here.

4 Where to Next?

4.1 Coding Hardware

There was sadly not much room in this survey to discuss hardware-specific implementations of codes. Many real-world encoders and decoders are not general-purpose computers, but rather specific components within a system. For an excellent and entertaining video on building Hamming codes on breadboards, see this video by Ben Eater. Perhaps a bit closer to the frontiers of computing are this paper on Low Power BCH Decoding and section 10.3 of the Dasgupta, Papadimitriou, and Vazirani book, which covers the Quantum Fourier Transform for Quantum Computers.

4.2 Locally-Decodable Codes

Normal Error-Correcting Codes allow efficient random-access retrieval of information if you only need to decode the portion of the data you are interested in, but they have poor noise resilience, since if a single block is corrupted some information is lost. As such, it is preferable to encode the whole message into a single codeword, but now you need to look at the whole codeword to recover any particular bit of it. Locally decodable codes are Error-Correcting Codes that avoid this problem since they have sublinear-time decoding algorithms. Classical Error-Correcting Codes have query complexity and codeword length proportional to the message length, but Locally-Decodable Codes improve on this. For further reading on Locally Decodable Codes, see this survey.

4.3 Lossy Compression

So far, we have discussed lossless compression, where messages can always be precisely recovered from their codes. Lossy (or irreversible) compression is a probabilistic encoding scheme that trades a bit of accuracy for reduced code length. You have probably encountered lossy compression of images, which appear blurrier and lower in quality than their original uncompressed versions. However, in some scenarios we are willing to trade a bit of accuracy for a significantly reduced size; lossy compression is primarily used for audio, video, images, and streaming media. If there are any typos in this survey, perhaps we can blame them on lossy compression!

The primary algorithm used in lossy compression is the Discrete Cosine Transform, which was discovered by Nasir Ahmed, T. Natarajan, and K.R. Rao in 1974. It takes advantage of the fact that fewer cosines than sines are needed to approximate a signal. If this sounds like the previous discussion of the Fast Fourier Transform, it is because the Discrete Cosine Transform is just a modified Fast Fourier Transform. For further reading on lossy compression and the Discrete Cosine Transform, see this article from Towards Data Science, which has some illustrative code examples.

References

[1] Thomas W. Judson. *Abstract Algebra Theory and Applications*. Stephen F. Austin State University, 2020.

[2] Wikipedia. *BCH Codes*
`https://en.wikipedia.org/wiki/BCH_code`

[3] Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani. *Algorithms*. 2006.

[4] Ben Eater. *What is Error Correction? Hamming Codes in Hardware.*
`https://www.youtube.com/watch?v=h0jloehRKas` 2020.

[5] Sergey Yekhanin. *Locally Decodable Codes*. Microsoft Research, 2012.

[6] Wikipedia. *Lossy Compression*
`https://en.wikipedia.org/wiki/Lossy_compression`