On the family of BCH codes

Pedro Costa
Mathematical Institute, University of Oxford

February 16, 2025

Contents

1. Introduction

Suppose we have a communication channel where we are sending a sequence of symbols, each chosen from a set $\{0, 1, \ldots, q-1\}$. In an ideal world, the symbols would be received exactly as they were sent. However, real-world channels are rarely perfect — noise or interference can corrupt the data. For instance, you might send the sequence 101, but the receiver ends up with 111, where the second symbol was mistakenly flipped from a 0 to a 1.
Even small errors like this can lead to big problems, especially in applications such as telecommunications, data storage, or even space exploration. That is where error-correcting codes come in. By adding carefully designed redundancy to the data, usually in the form of extra symbols, these codes enable the receiver not only to spot that something went wrong but also to figure out exactly what the error was and fix it.

The challenge usually lies around designing a scheme which allows the receiver to detect and correct up to t errors in the transmitted message, using the least possible amount of redundancy. Among the many error-correcting codes out there, **Bose – Chaudhuri – Hocquenghem (BCH) codes** stand out as a family of cyclic error-correcting codes with powerful error detection and correction capabilities, which are constructed using polynomials over a finite field.

This report delves into BCH codes, focusing on their structure and how they work in practice. It covers the mathematical foundations behind their design, provides algorithms for encoding and decoding messages, and demonstrates their error correction capabilities through examples. The report aims to provide sufficient detail for the codes to be fully understood and reproduced, with an appendix included for the implementation code.

2. Construction of BCH Codes

2.1. Mathematical Foundations

We assume the reader is familiar with the usual definitions of a field, ring and group from abstract algebra. In any case, [1] is an excellent reference for all the results presented in this subsection.

Some of the most well known examples of fields are the field of rational numbers, the field of real numbers and the field of complex numbers, endowed with the usual sum and multiplication operations. There are also finite fields, i.e. fields with a finite number of elements. In this case we say the *order* of a finite field is its number of elements. Common examples are the integers modulo a prime number p: it is not hard to check they satisfy the field axioms if addition and multiplication are performed modulo p. However, not all finite fields are of this form. In fact, the following holds:

Theorem 2.1. *For every prime number p and positive integer k, there are finite fields of order p^k. All finite fields of a given order are isomorphic.*

Remark. *One may identify all finite fields with the same order q and denote them by $\mathbb{F}_q$ or $GF(q)$, where the letters GF stand for Galois Field.*

We have seen how to construct $GF(p^k)$ when $k = 1$. For $k > 1$, let $q = p^k$. The field $GF(q)$ can be explicitly constructed but requires a few more definitions.

Definition 2.1. *Let $\mathbb{F}$ be a field. The polynomial ring in X over $\mathbb{F}$, which is denoted by $\mathbb{F}[X]$, is the set of expressions, called polynomials in X, of the form*

$$P = c_0 + c_1X + c_2X^2 + \cdots + c_{m-1}X^{m-1} + c_mX^m,$$

where $c_0, c_1, \ldots, c_m$, the coefficients of P, are elements of $\mathbb{F}$, $c_m \neq 0$ if $m > 0$, and $X, X^2, \ldots$ are symbols, which represent powers of X. These symbols follow the usual rules of exponentiation: $X^0 = 1, X^1 = X$ and $X^aX^b = X^{a+b}$ for any non-negative integers a, b. Addition and multiplication are defined according to the ordinary rules for manipulating algebraic expressions, making it a commutative algebra.

Definition 2.2. *Let $\mathbb{F}$ be a field and $P \in \mathbb{F}[X]$. P is irreducible over $\mathbb{F}$ if it cannot be factored into the product of two non-constant polynomials with coefficients in $\mathbb{F}$.*

Now, $GF(q)$ may be explicitly constructed in the following way. One first chooses an irreducible polynomial P in $GF(p)[X]$ of degree k (such an irreducible polynomial always exists). Then, the quotient

$$GF(q) = GF(p)[X]/(P)$$

of the polynomial ring $GF(p)[X]$ by the ideal generated by P is a field of order q. This quotient identifies each polynomial $Q \in GF(p)[X]$ with the remainder of its euclidean division by P. This remainder necessarily has degree less than k, meaning $GF(q)$ has exactly $p^k = q$ elements. The addition and multiplication are those of $GF(p)[X]$, reducing modulo P in the end of each operation. Furthermore, since P was chosen to be irreducible, every non-zero element of $GF(q)$ is invertible.

Remark. *Except in the construction of $GF(4)$, there are several possible choices for P.*

The following result about finite fields will be useful.

Theorem 2.2. *The multiplicative group of the non-zero elements in $GF(q)$ is cyclic, i.e., there exists an element α such that the $q-1$ non-zero elements of $GF(q)$ are $\alpha, \alpha^2, \ldots, \alpha^{q-2}, \alpha^{q-1} = 1$. Such an element is called a primitive element of $GF(q)$.*

Remark. *The number of primitive elements of $GF(q)$ is $\phi(q-1)$, where ϕ is Euler's totient function.*

The last mathematical foundation we will need is that of a minimal polynomial.

Definition 2.3. *Let $GF(p^k)$ be a finite field with $k > 1$ and $\alpha \in GF(p^k)$. The minimal polynomial of α is the monic polynomial of least degree among all polynomials in $GF(p)[X]$ having α as root.*

Remark. *Note that since the multiplicative group of $GF(p^k)$ is cyclic, $\alpha^{p^k-1} - 1 = 0$ so there is always at least one polynomial in $GF(p)[X]$ having α as root, namely $X^{p^k-1} - 1$.*

It can be shown that the minimal polynomial of α is unique, irreducible over $GF(p)$ and generates the ideal of all polynomials which have α as a root. In addition, the degree of the minimal polynomial of α divides k [4].

2.2. Code definition

BCH codes make use of the concepts introduced in the previous subsection to create a polynomial code defined over a finite field $GF(q)$. First we define **primitive narrow-sense BCH codes**.

Let q be a prime number and m, d positive integers such that $d \leq q^m - 1$. Let α be a primitive element of $GF(q^m)$ and $n = q^m - 1$.

1. Compute the minimal polynomial $m_i(x)$ (with coefficients in $GF(q)$) of α^i, $i = 1, \dots, d-1$.

2. Compute $g(x) = lcm(m_1(x), m_2(x), \dots, m_{d-1}(x))$. Since each $m_i(x)$ is irreducible, this corresponds to multiplying all distinct $m_i(x)$ together.

3. The primitive narrow-sense BCH code is the polynomial code generated by $g(x)$, i.e., the set of codewords

$$\Big\{ g(x)h(x) \colon h(x) \in GF(q)[x] \wedge deg(h(x)) \leq n - deg(g(x)) - 1 \Big\}. \tag{1}$$

Remark. *Since $(\alpha^i)^n - 1 = 0$, $m_i(x) \mid x^n - 1$ for all $i = 1, \dots, d-1$. Thus, $g(x) \mid x^n - 1$. This means the polynomial code generated by $g(x)$ is a cyclic code.*

General BCH codes differ from primitive narrow-sense BCH codes in two ways. Firstly, the block length n need not equal $q^m - 1$, but instead n can be any divisor of $q^m - 1$. Secondly, the minimal polynomials computed can be for any contiguous segment of powers of α, starting at α^c: $\alpha^c, \alpha^{c+1}, \dots, \alpha^{c+d-2}$.
This way, the previous construction also works if α is chosen to be a primitive n^{th} root of unity in $GF(q^m)$. If we are given, instead, the values of q and n, we may choose m to be the multiplicative order of q modulo n so that $n \mid q^m - 1$. We are then able to compute $g(x)$ and the polynomial code generated by $g(x)$ is a BCH code.

- A BCH code with $c = 1$ is called a *narrow-sense BCH code.*

- A BCH code with $n = q^m - 1$ is called *primitive.*

2.2.1 Example

Suppose $q = 2$ and $n = 7$ (which implies $m = 3$) and take $c = 1$. We will construct the generator polynomial for different values of d for $GF(2^3)$ modulus the irreducible polynomial $P(x) = z^3 + z + 1$. To avoid confusion, polynomials in $GF(2^3)$ will use the letter z and minimal polynomials will use the letter x.

It turns out that the polynomial $\alpha(z) = z$ is a primitive n^{th} root of $GF(2^3)$, as can be seen by the following table.

$\alpha(z)^i$	$\alpha(z)$	$\alpha(z)^2$	$\alpha(z)^3$	$\alpha(z)^4$	$\alpha(z)^5$	$\alpha(z)^6$	$\alpha(z)^7$
Poly	z	z^2	$z+1$	z^2+z	z^2+z+1	z^2+1	1

Table 1: Elements of $GF(2^3)$ as powers of the primitive element $\alpha(z)$.

Note that since $P(x) = z^3 + z + 1$ is used to construct $GF(2^3)$, and we are working over $GF(2)$, we have $z^3 = -z - 1 = z + 1$. The minimal polynomial of α, α^2 and α^4 is

x^3+x+1 and of α^3, α^6 and α^5 is x^3+x^2+1. Then, the BCH code with $d = 2, 3$ has the generator polynomial

$$g(x) = lcm(m_1(x), m_2(x)) = m_1(x) = x^3 + x + 1.$$

It has minimal Hamming distance at least 3 and corrects up to one error. Since $g(x)$ has degree 3, this code has 4 data bits and 3 checksum bits, achieving the same redundancy as Hamming's $(7, 4)$ code. It is also denoted as a $(7, 4)$ **BCH code**.

Increasing d to $4, 5, 6$ or 7, we get

$$g(x) = lcm(m_1(x), m_2(x), m_3(x), m_4(x), m_5(x), m_6(x)) = m_1(x)m_3(x)$$
$$= (x^3 + x + 1)(x^3 + x^2 + 1) = x^6 + x^5 + x^4 + x^3 + x^2 + x + 1.$$

This code has minimal Hamming distance 7 and corrects three errors. It has 1 data bit and 6 checksum bits. It is also denoted as a $(7, 1)$ **BCH code**. In fact, this corresponds to a trivial repetition code: the only two codewords are 0000000 and 1111111.

Another example with $q = 2$, $m = 4$ and $n = 15$ can be seen in [6].

2.3. Properties

BCH codes have several interesting properties which make them powerful error-correcting codes.

Property 1. *A BCH code is cyclic.*

Proof. As seen in a previous remark, $g(x)$ divides $x^n - 1$, which implies the polynomial code generated by $g(x)$ is cyclic. □

Lemma 2.1. *Each minimal polynomial $m_i(x)$ has degree at most m.*

Proof. This can be seen by a simple application of the pigeonhole principle: there are $q^{m+1} - 1$ non-zero polynomials $P \in GF(q)[x]$ of degree at most m, and each of them maps a given $\alpha^i \in GF(q^m)$ to some element of $GF(q^m)$. Thus, there are two distinct P_1, P_2 of degree at most m such that

$$P_1(\alpha^i) = P_2(\alpha^i) \Rightarrow (P_1 - P_2)(\alpha^i) = 0,$$

proving the existence of at least one polynomial in $GF(q)[x]$ of degree at most m, $P_1 - P_2$, which has α^i as a root[1]. □

Lemma 2.2. *If $q = 2$, $m_i(x) = m_{2i}(x)$ for all i.*

Proof. Since $q = 2$, $(a + b)^2 = a^2 + b^2$. In other words, evaluating a polynomial in x^2 corresponds to evaluating it in x, and then squaring the result. Then,

$$m_i(\alpha^{2i}) = m_i(\alpha^i)^2 = 0^2 = 0.$$

We just showed that m_i has α^{2i} as root, so $m_{2i} \mid m_i$. Since m_i is irreducible, $m_{2i} = m_i$. □

Property 2. *The generator polynomial of a BCH code has degree at most $(d - 1)m$. If $q = 2$ and $c = 1$, then the degree is bounded by $dm/2$.*

[1]In fact, we saw earlier that the degree of $m_i(x)$ divides m so this result can be made stronger.

Proof. Each minimal polynomial $m_i(x)$ has degree at most m. Since we are taking the least common multiple of $d-1$ such polynomials, $g(x)$ has degree at most $(d-1)m$.
If $q = 2$, $m_i(x) = m_{2i}(x)$ for all i, which means $g(x)$ is the least common multiple of at most $d/2$ minimal polynomials. □

Property 3. *A BCH code has minimal Hamming distance at least d.*

Proof. Suppose that $p(x)$ is a codeword with less than d non-zero terms. Then

$$p(x) = b_1 x^{k_1} + \cdots + b_{d-1} x^{k_{d-1}}, \text{ where } k_1 < k_2 < \cdots < k_{d-1}.$$

Recall that $\alpha^c, \ldots, \alpha^{c+d-2}$ are roots of $g(x)$, hence of $p(x)$. This implies that $b_1, \ldots, b_{d-1}$ satisfy the following equations, for each $i \in \{c, \ldots, c+d-2\}$:

$$p\left(\alpha^i\right) = b_1 \alpha^{ik_1} + b_2 \alpha^{ik_2} + \cdots + b_{d-1} \alpha^{ik_{d-1}} = 0$$

In matrix form, we have

$$\begin{bmatrix} \alpha^{ck_1} & \alpha^{ck_2} & \cdots & \alpha^{ck_{d-1}} \\ \alpha^{(c+1)k_1} & \alpha^{(c+1)k_2} & \cdots & \alpha^{(c+1)k_{d-1}} \\ \vdots & \vdots & & \vdots \\ \alpha^{(c+d-2)k_1} & \alpha^{(c+d-2)k_2} & \cdots & \alpha^{(c+d-2)k_{d-1}} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{d-1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

The determinant of this matrix equals

$$\left(\prod_{i=1}^{d-1} \alpha^{ck_i}\right) \det \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha^{k_1} & \alpha^{k_2} & \cdots & \alpha^{k_{d-1}} \\ \vdots & \vdots & & \vdots \\ \alpha^{(d-2)k_1} & \alpha^{(d-2)k_2} & \cdots & \alpha^{(d-2)k_{d-1}} \end{pmatrix} = \left(\prod_{i=1}^{d-1} \alpha^{ck_i}\right) \det(V)$$

The matrix V is seen to be a Vandermonde matrix, and its determinant is

$$\det(V) = \prod_{1 \le i < j \le d-1} \left(\alpha^{k_j} - \alpha^{k_i}\right)$$

which is non-zero. It therefore follows that $b_1, \ldots, b_{d-1} = 0$, hence $p(x) = 0$. □

Remark. *If an error-correcting code has minimal Hamming distance at least d, it can correct up to $t = \lfloor (d-1)/2 \rfloor$ errors.*

Property 2 guarantees that, given n, m, d, the number of message symbols we can fit into a code block of length n is $n - deg(g) \geq n - m(d-1)$. In the case of primitive BCH codes, $n = q^m - 1$ so for each additional error we wish to correct, we increase d by two and use about $2m \sim 2\log_q(n)$ extra redundancy symbols. If $q = 2$, this is improved to $m \sim \log_2(n)$ redundancy bits per additional error.

3. Encoding Process

Given the set (1) of all codewords, encoding a message comes down to finding a suitable multiple of the generating polynomial to encode the message. There are multiple ways to do this, which can be broken down into two types:

- Systematic encoding, where the message appears verbatim somewhere within the codeword.
- Non-systematic encoding, where the above condition is not enforced.

3.1. Non-systematic encoding algorithm

A simple non-systematic encoding algorithm is to take the symbols of the message as coefficients and form a polynomial $m(x)$. Then, we multiply

$$s(x) = m(x)g(x)$$

and take the coefficients of $s(x)$ as the codeword.

As an example, let us revisit Section 2.2.1, where $g(x) = x^3 + x + 1$ was obtained as the generating polynomial of a BCH(7,4) code. In order to encode the message $\{0101\}$, we first turn it into the polynomial

$$m(x) = 0x^3 + 1x^2 + 0x + 1 = x^2 + 1$$

and calculate

$$s(x) = (x^2 + 1)(x^3 + x + 1) = x^5 + x^3 + x^2 + x^3 + x + 1 = x^5 + x^2 + x + 1.$$

Thus, the transmitted codeword is $\{0100111\}$.

The receiver can then, after correcting any possible errors that might have occurred during transmission, divide $s(x)$ by $g(x)$ to retrieve $m(x)$ and thus the original message.

3.2. Systematic encoding algorithm

In order for the message (of length k) to appear verbatim in the codeword (of length $n > k$), we will leverage the fact that we have total freedom to choose the $n - k$ remaining symbols that go into the codeword. In fact, there is so much freedom that we can start our codeword with the message, and still have enough symbols left to choose so that it is possible to force the codeword polynomial to be divisible by $g(x)$.

Formally, this means that we can calculate the remainder of the euclidean division of $x^{n-k}m(x)$ by $g(x)$ to get

$$x^{n-k}m(x) = q(x)g(x) + r(x).$$

Since $r(x)$ is a polynomial of degree strictly lower than that of $g(x)$ (which is $n - k$), subtracting $r(x)$ from $x^{n-k}m(x)$ does not change any of its message coefficients. Therefore, subtracting $r(x)$, we get

$$s(x) = q(x)g(x) = x^{n-k}m(x) - r(x).$$

As an example, let us try to encode the same message $\{0101\}$ as in the previous subsection. Since $n - k = 3$,

$$x^3m(x) = x^5 + x^3$$

and we calculate

$$\begin{aligned} x^5 + x^3 &= (x^2)(x^3 + x + 1) + (x^2) \\ \Rightarrow s(x) &= (x^2)(x^3 + x + 1) = x^5 + x^3 + x^2 \end{aligned}$$

Thus, the transmitted codeword is $\{0101100\}$. Note that the message appears verbatim in the first 4 bits.

The receiver can then, after correcting any possible errors that might have occurred during transmission, simply take the first k bits in the codeword as the original message.

4. Decoding Process

Decoding is, in general, a lot more complicated than encoding. This is mainly due to the task of detecting and correcting the errors in the received vector. Once this is done, however, our life is quite simple as we saw in Section 3.
Henceforth, we focus solely on finding and correcting the errors in the received vector. This could, of course, be done via a brute-force method: simply iterate over the number of errors t, then all possible sets of t positions in our vector, and all possible ways to change the symbols in those positions. Finally, in each iteration, check if we got a valid codeword. Due to the properties of BCH codes, if at most $\lfloor (d-1)/2 \rfloor$ errors occurred during transmission, there will be exactly one match and that is the correct codeword.

However, this very quickly becomes computationally unfeasible as the code parameters grow. Luckily, BCH codes have efficient decoding algorithms to help in this task. The high level description of the decoding algorithm goes as follows:

1. Calculate the syndromes s_j for the received vector.
2. Determine the number of errors t and the error locator polynomial $\Lambda(x)$ from the syndromes.
3. Calculate the roots of the error location polynomial to find the error locations X_i.
4. Calculate the error values Y_i at those error locations.
5. Correct the errors.

The received vector R can be though of as the sum of the correct codeword C and an unknown error vector E. Interpreting R as a polynomial, the syndrome s_j is defined as

$$s_j := R(\alpha^j) = C(\alpha^j) + E(\alpha^j),$$

where $j = c, c+1, \ldots, c+d-2$.
Since C is a valid codeword, $C(x)$ is a multiple of the generator polynomial $g(x)$. By construction, $g(\alpha^j) = 0$ for $j = c, c+1, \ldots, c+d-2$. As a result, $C(\alpha^j) = 0$ and the syndrome simplifies to

$$s_j = E(\alpha^j).$$

This way, we are able to isolate the error vector E. If the syndromes are all zero, the decoding is done since there was no error. Otherwise, assume $E(x)$ can be written as

$$E(x) = e_1 x^{i_1} + e_2 x^{i_2} + \ldots$$

where $i_1, i_2, \ldots$ are the error locations and $e_1, e_2, \ldots$ the error values.

4.1. Locating the errors

The first step in locating the errors is finding, with minimal possible t, the **error locator polynomial**

$$\Lambda(x) = \prod_{j=1}^{t} \left(x\alpha^{i_j} - 1 \right),$$

where $i_1, i_2, \ldots$ are the error locations. Note that we do not know yet how many errors occurred. We will describe the Peterson-Gorenstein-Zierler algorithm [5], but popular alternatives exist such as the Berlekamp–Massey algorithm and the Sugiyama Euclidean algorithm. Peterson's algorithm is used to calculate the error locating polynomial coefficients $\lambda_1, \lambda_2, \ldots, \lambda_t$ of

$$\Lambda(x) = 1 + \lambda_1 x + \lambda_2 x^2 + \cdots + \lambda_t x^t.$$

Initially, t is set to $\lfloor (d-1)/2 \rfloor$. We will use the syndromes $s_c, \ldots, s_{c+2t-1}$.

1. Calculate the matrix $S_{t\times t}$ where $S_{i,j} = s_{c+i+j-2}$

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

2. Generate a $c_{t\times 1}$ vector with elements

$$c_{t\times 1} = \begin{bmatrix} s_{c+t} \\ s_{c+t+1} \\ \vdots \\ s_{c+2t-1} \end{bmatrix}.$$

3. Let Λ denote the unknown polynomial coefficients, which are given by

$$\Lambda_{t\times 1} = \begin{bmatrix} \lambda_t \\ \lambda_{t-1} \\ \vdots \\ \lambda_1 \end{bmatrix}.$$

4. Form the matrix equation
$$S_{t\times t}\Lambda_{t\times 1} = -C_{t\times 1}.$$

5. If the determinant of matrix $S_{t\times t}$ is non-zero, then we can actually find an inverse of this matrix and solve for the unknown values in Λ. The algorithm stops here.

6. If $\det(S_{t\times t}) = 0$, decrease t by one. If t becomes zero, the algorithm stops and declares an empty error locator polynomial. Otherwise, go back to step 1.

The proof that the coefficients of the error locating polynomial must satisfy the matrix equation in step 4 is omitted for brevity[2] but boils down to a clever manipulation of the error locating polynomial and the definition of the syndromes. The interested reader can find a proof in [3].

Once we have the error locator polynomial $\Lambda(x)$, we can finds its roots by testing successive powers of the primitive element α. The roots of $\Lambda(x)$ are $\alpha^{-i_1}, \ldots, \alpha^{-i_t}$, so we find the values of i_j and are able to locate the errors in the received vector.

[2]The author also considered it would not add much insight in the general context of BCH codes.

4.2. Correcting the errors

In the case of binary BCH codes ($q = 2$), locating the errors is enough to correct them since there are only two possibilities for each symbol: just flip the bits of the received vector at the error positions to retrieve the correct codeword. In the general case, the error values e_j can be determined by solving the linear system

$$\begin{aligned} s_c &= e_1\alpha^{ci_1} + e_2\alpha^{ci_2} + \cdots + e_t\alpha^{ci_t} \\ s_{c+1} &= e_1\alpha^{(c+1)i_1} + e_2\alpha^{(c+1)i_2} + \cdots + e_t\alpha^{(c+1)i_t} \\ &\vdots \\ s_{c+t-1} &= e_1\alpha^{(c+t-1)i_1} + e_2\alpha^{(c+t-1)i_2} + \cdots + e_t\alpha^{(c+t-1)i_t} \end{aligned}$$

Similarly to the proof of Property 3, the square matrix of this linear system has non-zero determinant (since all i_j are distinct) and it can be inverted to give the solution for the error values.

More efficient algorithms, like the Forney algorithm, exist. It is based on Lagrange interpolation and techniques of generating functions. Define

$$\begin{aligned} &S(x) = s_c + s_{c+1}x + s_{c+2}x^2 + \cdots + s_{c+d-2}x^{d-2} \\ &t \leqslant d-1, \lambda_0 \neq 0 \quad \Lambda(x) = \sum_{i=0}^{t} \lambda_i x^i = \lambda_0 \prod_{k=0}^{t} \left(\alpha^{-i_k}x - 1\right). \end{aligned}$$

And the error evaluator polynomial

$$\Omega(x) \equiv S(x)\Lambda(x) \bmod x^{d-1}$$

Finally, we take a formal derivative:

$$\Lambda'(x) = \sum_{i=1}^{v} i \cdot \lambda_i x^{i-1}$$

where

$$i \cdot x := \sum_{k=1}^{i} x$$

It can be shown that if the syndromes could be explained by an error vector, which could only be non-zero on positions i_k, then the error values are given by

$$e_k = -\frac{\alpha^{i_k}\Omega\left(\alpha^{-i_k}\right)}{\alpha^{c\cdot i_k}\Lambda'\left(\alpha^{-i_k}\right)}.$$

For narrow-sense BCH codes, $c = 1$, so the expression simplifies to:

$$e_k = -\frac{\Omega\left(\alpha^{-i_k}\right)}{\Lambda'\left(\alpha^{-i_k}\right)}.$$

The interested reader can see a derivation for the formula of the Forney algorithm in [2], [6].

4.3. Example

Consider a BCH code in $GF(2^4)$ with $d = 7$ and $g(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$. We want to transmit the message [01011], or in polynomial notation, $m(x) = x^3 + x + 1$. The redundancy symbols are calculated by dividing $x^{10}m(x)$ by $g(x)$ and taking the remainder, resulting in $x^7 + x^3 + x^2 + x + 1$ or [0010001111]. These are appended to the message, so the transmitted codeword is [010110010001111].

Now, suppose there are two bit-errors in the transmission, so the received codeword is [110110010101111]. In polynomial notation:

$$R(x) = C(x) + x^{14} + x^5 = x^{14} + x^{13} + x^{11} + x^{10} + x^7 + x^5 + x^3 + x^2 + x + 1$$

In order to correct the errors, first we calculate the syndromes. Taking $\alpha = 0010$,[3] we have $s_1 = R(\alpha^1) = 1111, s_2 = 1010$, $s_3 = 1110, s_4 = 1000, s_5 = 0000$, and $s_6 = 1011$. Next, apply the Peterson procedure by computing the determinant of the matrix.

$$S_{3\times 3} = \begin{bmatrix} s_1 & s_2 & s_3 \\ s_2 & s_3 & s_4 \\ s_3 & s_4 & s_5 \end{bmatrix} = \begin{bmatrix} 1111 & 1010 & 1110 \\ 1010 & 1110 & 1000 \\ 1110 & 1000 & 0000 \end{bmatrix}$$

The determinant turns out to be zero, which is no surprise since only two errors were introduced into the codeword. However, when we decrease the size of S by one, we get

$$S_{2\times 2} = \begin{bmatrix} s_1 & s_2 \\ s_2 & s_3 \end{bmatrix} = \begin{bmatrix} 1111 & 1010 \\ 1010 & 1110 \end{bmatrix}$$

which has determinant $s_1 s_3 - s_2 s_2 = \alpha^{12}\alpha^{11} - \alpha^9\alpha^9 = \alpha^8 - \alpha^3 = \alpha^{13} \neq 0$. Note that this can be easily checked with a lookup table of all powers of α such as table 1, which we calculated for $GF(2^3)$ earlier. Solving for λ_i, we get $\lambda_2 = 0011, \lambda_1 = 1111$. The resulting error locator polynomial is $\Lambda(x) = 0011x^2 + 1111x + 0001$, which has zeros at $0010 = \alpha^{-14}$ and $0111 = \alpha^{-5}$. The exponents of α correspond to the error locations. There is no need to calculate the error values here, as the only possible value is 1.

Another example of decoding with the same BCH Code in $GF(2^4)$ can be seen in [6].

5. BCH Codes in practice

A significant portion of the time and effort spent producing this report were directed towards a full Python implementation of **general BCH codes**, which we now present. We made extensive use of the *SymPy* and *galois* libraries. *SymPy* is a Python library for symbolic mathematics, while the *galois* library is a Python 3 package that extends NumPy arrays to operate over finite fields.

5.1. Common subroutines

While working on the implementation, it became clear that certain subroutines would be frequently used and it would be useful to implement them efficiently in separate functions.

- `fastExp` performs fast exponentiation of a given Sympy polynomial, modulo a reducing polynomial `irr_poly`. It does so by using binary exponentiation to reduce the number of polynomial multiplications needed from $\mathcal{O}(\texttt{exp})$ to $\mathcal{O}(log_2(\texttt{exp}))$.

[3] This corresponds to $\alpha(z) = z$, if we think of the elements of $GF(2^4)$ as polynomials in z of degree at most 3, with coefficients in $GF(2)$, as presented in subsection 2.1.

- `findNthRoot` finds a primitive n^{th} root of $GF(q^m)$, assuming reducing polynomial `irr_poly`. It randomly generates non-zero elements of $GF(q^m)$ until it finds one whose order r is divisible by n. Then, by raising it to the power $\frac{r}{n}$ we get a primitive n^{th} root of $GF(q^m)$.

- `evaluatePoly` takes two SymPy polynomials `poly` and `min_poly` and evaluates the first into the second one. E.g. x^2 evaluated into $2\alpha^3 + 3\alpha$ yields $2x^6 + 3x^2$. It assumes both polynomials have coefficients in $GF(q)$ and takes the result modulo `irr_poly`.

- `minPoly` calculates the minimum polynomial of `poly` in $GF(q^m)$ with reducing polynomial `irr_poly`.

- `polyToInt` and `intToPoly` convert SymPy polynomials to their integer representation and back, respectively. E.g. `polyToInt(Poly(2*x**3+x+2, x, domain = GF(3)),3)` returns $2 \cdot 3^3 + 0 \cdot 3^2 + 1 \cdot 3^1 + 2 \cdot 3^0 = 59$.

5.2. Class BCHCode

The class `BCHCode` was defined to handle all major computations required to create a BCH code and perform encoding and decoding of messages. The attributes of this class are:

- `q,n,d,c,m,t`, the parameters of the code. t denotes the maximum number of errors the code will be able to correct.

- `g_poly, irr_poly, nth_root` the generating polynomial, the degree m irreducible polynomial that reduces $GF(q^m)$ and a primitive n^{th} root of unity, respectively.

- `elem` a list of length n with all the powers of the `nth_root` listed in order.

It has 5 methods, the first three being private:

- `__init__` receives q, n, d, c and calculates m to be the order of q modulo n, as well as $t = \lfloor (d-1)/2 \rfloor$. It creates a FieldArray subclass for $GF(q^m)$ and calls `__gen` to initialize the remaining attributes.

- `__gen` computes `irr_poly` and `nth_root` by using the subroutines `findNthRoot`, `fastExp` and `minPoly`. It also calls `__calcElem` and computes the generating polynomial `g_poly`.

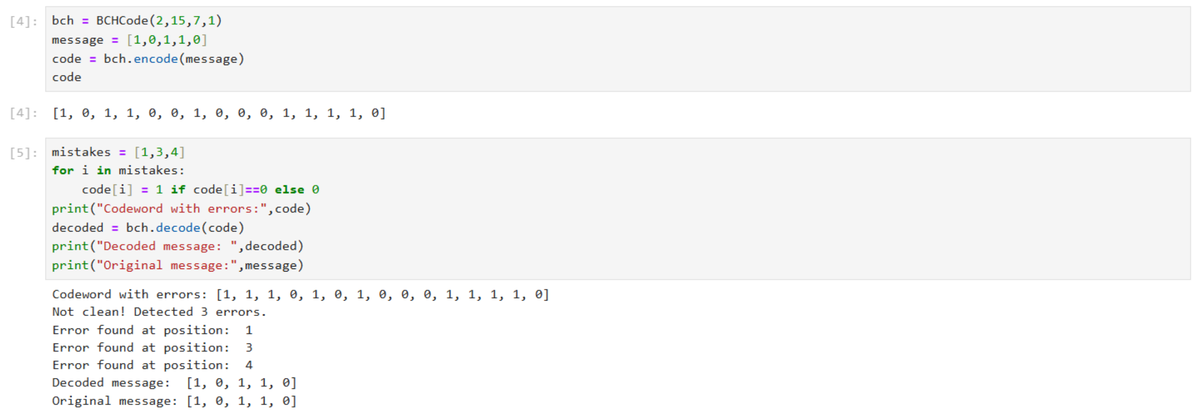
- `__calcElem` calculates the successive powers of `nth_root`, modulo `irr_poly`.

- `encode` receives a q-ary list (each entry is an integer between 0 and $q-1$) of size at most $n - deg(g\_poly)$, representing the message, and encodes it systematically (3.2). The message is automatically padded with 0's to the right until its length is $n - deg(g\_poly)$. The output codeword is a q-ary list of size n.

- `decode` receives a q-ary list of size n and performs the decoding algorithm. It calculates the syndromes, follows Peterson's algorithm (4.1) and solves the final linear system for the error values. After correcting them, it returns the original message.

5.3. Test Cases

After a detailed theoretical study, and careful implementation, we are ready to see BCH codes in action. Basic usage of the `BCHCode` class can be seen in Figure 1.



```
[4]: bch = BCHCode(2,15,7,1)
     message = [1,0,1,1,0]
     code = bch.encode(message)
     code

[4]: [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0]

[5]: mistakes = [1,3,4]
     for i in mistakes:
         code[i] = 1 if code[i]==0 else 0
     print("Codeword with errors:",code)
     decoded = bch.decode(code)
     print("Decoded message: ",decoded)
     print("Original message:",message)

     Codeword with errors: [1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0]
     Not clean! Detected 3 errors.
     Error found at position:  1
     Error found at position:  3
     Error found at position:  4
     Decoded message:  [1, 0, 1, 1, 0]
     Original message: [1, 0, 1, 1, 0]
```

Figure 1: BCH code with $q = 2, n = 15, d = 7$ and $c = 1$ being used to encode 10110 and correct three errors in the codeword.

Of course, having a working implementation of **general BCH codes**, we can go much further since we are not restricted to working modulo 2. To automate the process of testing the code, the function `testBCH` was implemented. It receives as inputs a `BCHCode` object, a message and a boolean `english`, with default value `False`.

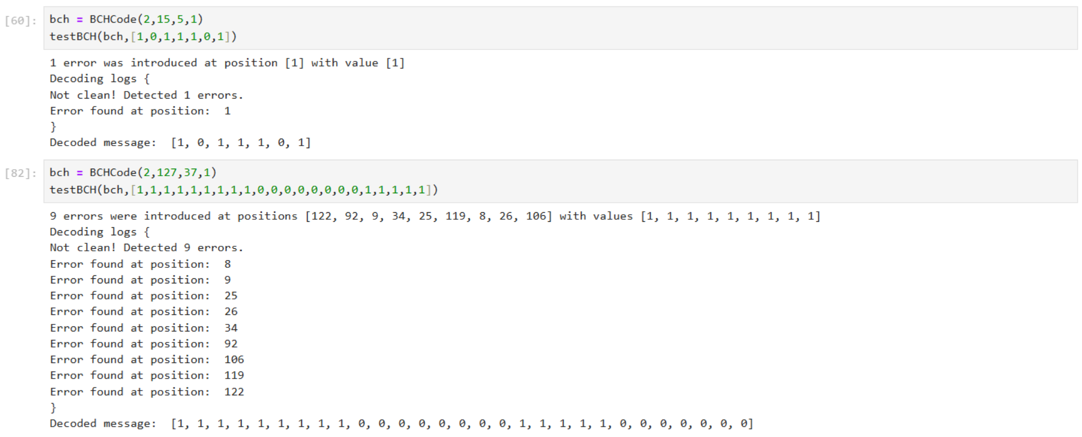
If `english` is `False`, after encoding the message the function randomly selects a number of errors between 1 and t, where those errors will occur and their values. After corrupting the data, it decodes it and prints the result, which should match the initial message.

If `english` is `True`, the function behaves similarly with the exception that now the message is a string consisting of upper-case English letters, commas, full stops, exclamation marks, question marks and empty spaces. This choice is made precisely so that there are $q = 31$ possible characters, a prime number. This string is converted to a 31-ary list which is then encoded as in the previous case. Additionally, in this case we make it so that all errors occur in the initial positions of the codeword, where the message appears. This is so the error correcting capabilities of the code are more obvious to the naked eye, and changes nothing about the structure of the algorithm.

Examples of testing binary BCH codes can be seen in Figure 2, and non-binary codes in Figure 3. The expressivity of the 31 chosen English text characters also allows us to encode almost any piece of text and see it being decoded in real time! Check Figures 4, 5 for examples.

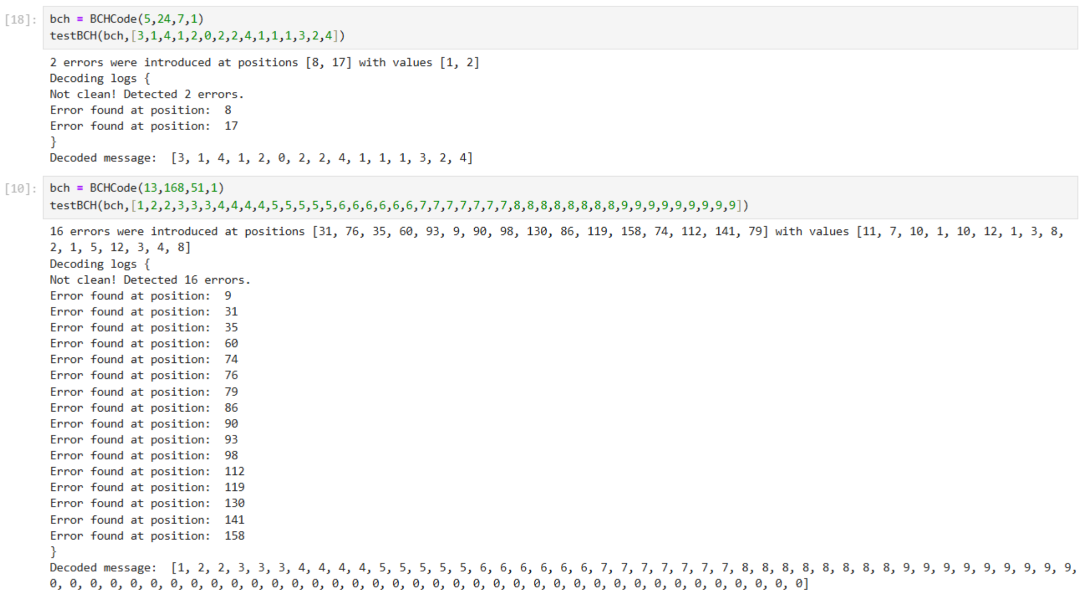
6. Conclusion

In this report, we introduced BCH codes, a family of cyclic error correcting codes with powerful error detection and correction capabilities. After covering the necessary mathematical foundations, we explained the code design process and explored its key properties. Subsequently, efficient algorithms for both encoding and decoding messages were given in detail, with relevant examples where needed. Finally, these codes were seen in action



```
[60]: bch = BCHCode(2,15,5,1)
      testBCH(bch,[1,0,1,1,1,0,1])

      1 error was introduced at position [1] with value [1]
      Decoding logs {
      Not clean! Detected 1 errors.
      Error found at position:  1
      }
      Decoded message:  [1, 0, 1, 1, 1, 0, 1]

[82]: bch = BCHCode(2,127,37,1)
      testBCH(bch,[1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,1])

      9 errors were introduced at positions [122, 92, 9, 34, 25, 119, 8, 26, 106] with values [1, 1, 1, 1, 1, 1, 1, 1, 1]
      Decoding logs {
      Not clean! Detected 9 errors.
      Error found at position:  8
      Error found at position:  9
      Error found at position:  25
      Error found at position:  26
      Error found at position:  34
      Error found at position:  92
      Error found at position:  106
      Error found at position:  119
      Error found at position:  122
      }
      Decoded message:  [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
```

Figure 2: Examples of error correction of binary BCH codes over $GF(2^4)$ and $GF(2^7)$.

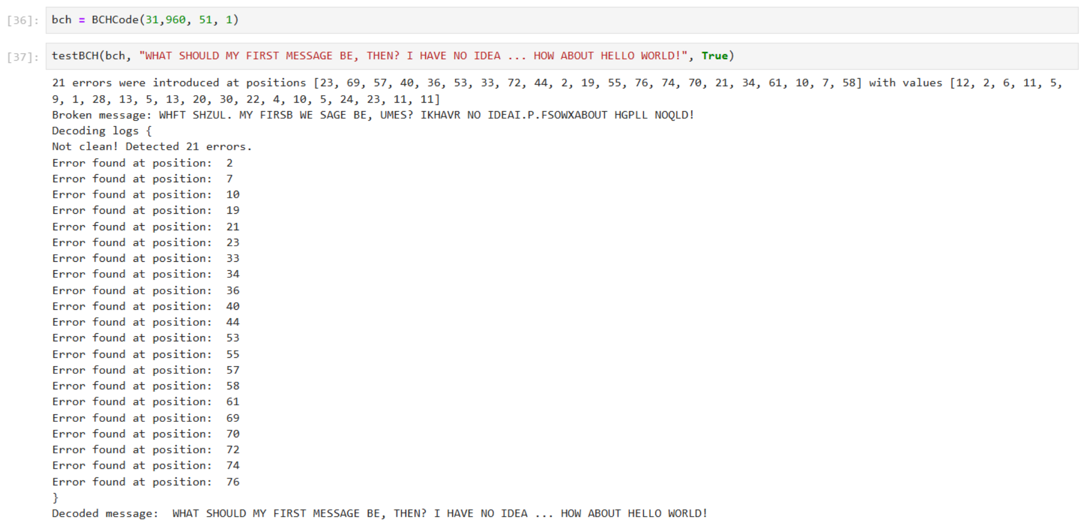


```
[18]: bch = BCHCode(5,24,7,1)
      testBCH(bch,[3,1,4,1,2,0,2,2,4,1,1,1,3,2,4])

      2 errors were introduced at positions [8, 17] with values [1, 2]
      Decoding logs {
      Not clean! Detected 2 errors.
      Error found at position:  8
      Error found at position:  17
      }
      Decoded message:  [3, 1, 4, 1, 2, 0, 2, 2, 4, 1, 1, 1, 3, 2, 4]

[10]: bch = BCHCode(13,168,51,1)
      testBCH(bch,[1,2,2,3,3,3,4,4,4,4,5,5,5,5,5,6,6,6,6,6,6,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,9])

      16 errors were introduced at positions [31, 76, 35, 60, 93, 9, 90, 98, 130, 86, 119, 158, 74, 112, 141, 79] with values [11, 7, 10, 1, 10, 12, 1, 3, 8,
      2, 1, 5, 12, 3, 4, 8]
      Decoding logs {
      Not clean! Detected 16 errors.
      Error found at position:  9
      Error found at position:  31
      Error found at position:  35
      Error found at position:  60
      Error found at position:  74
      Error found at position:  76
      Error found at position:  79
      Error found at position:  86
      Error found at position:  90
      Error found at position:  93
      Error found at position:  98
      Error found at position:  112
      Error found at position:  119
      Error found at position:  130
      Error found at position:  141
      Error found at position:  158
      }
      Decoded message:  [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 3: Examples using $q = 5$ and $q = 13$ correcting different numbers of errors.

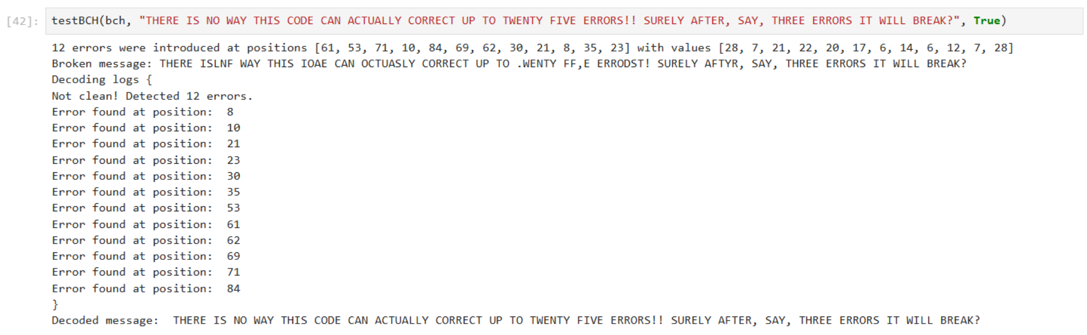
with a complete implementation of **general BCH codes**, along with some experiments to test the full error correction potential of this approach.

By covering both the theoretical framework and practical implementation, this report aims to equip the attentive reader to fully understand and reproduce general BCH codes. Special emphasis was placed on motivating the constructions needed, while presenting the content in a clear and concise way. The appendix includes the full implementation code, providing a reference to those interested in experimenting with or implement BCH codes themselves.



```
[36]: bch = BCHCode(31,960, 51, 1)

[37]: testBCH(bch, "WHAT SHOULD MY FIRST MESSAGE BE, THEN? I HAVE NO IDEA ... HOW ABOUT HELLO WORLD!", True)

      21 errors were introduced at positions [23, 69, 57, 40, 36, 53, 33, 72, 44, 2, 19, 55, 76, 74, 70, 21, 34, 61, 10, 7, 58] with values [12, 2, 6, 11, 5,
      9, 1, 28, 13, 5, 13, 20, 30, 22, 4, 10, 5, 24, 23, 11, 11]
      Broken message: WHFT SHZUL. MY FIRSB WE SAGE BE, UMES? IKHAVR NO IDEAI.P.FSOWXABOUT HGPLL NOQLD!
      Decoding logs {
      Not clean! Detected 21 errors.
      Error found at position:  2
      Error found at position:  7
      Error found at position:  10
      Error found at position:  19
      Error found at position:  21
      Error found at position:  23
      Error found at position:  33
      Error found at position:  34
      Error found at position:  36
      Error found at position:  40
      Error found at position:  44
      Error found at position:  53
      Error found at position:  55
      Error found at position:  57
      Error found at position:  58
      Error found at position:  61
      Error found at position:  69
      Error found at position:  70
      Error found at position:  72
      Error found at position:  74
      Error found at position:  76
      }
      Decoded message:  WHAT SHOULD MY FIRST MESSAGE BE, THEN? I HAVE NO IDEA ... HOW ABOUT HELLO WORLD!
```

Figure 4: BCH code with $q = 31, n = 960, d = 51$ and $c = 1$ encoding English text and correcting 21 errors. There are 863 data symbols and 97 check-symbols, which were not printed.



```
[42]: testBCH(bch, "THERE IS NO WAY THIS CODE CAN ACTUALLY CORRECT UP TO TWENTY FIVE ERRORS!! SURELY AFTER, SAY, THREE ERRORS IT WILL BREAK?", True)

      12 errors were introduced at positions [61, 53, 71, 10, 84, 69, 62, 30, 21, 8, 35, 23] with values [28, 7, 21, 22, 20, 17, 6, 14, 6, 12, 7, 28]
      Broken message: THERE ISLNF WAY THIS IOAE CAN OCTUASLY CORRECT UP TO .WENTY FF,E ERRODST! SURELY AFTYR, SAY, THREE ERRORS IT WILL BREAK?
      Decoding logs {
      Not clean! Detected 12 errors.
      Error found at position:  8
      Error found at position:  10
      Error found at position:  21
      Error found at position:  23
      Error found at position:  30
      Error found at position:  35
      Error found at position:  53
      Error found at position:  61
      Error found at position:  62
      Error found at position:  69
      Error found at position:  71
      Error found at position:  84
      }
      Decoded message:  THERE IS NO WAY THIS CODE CAN ACTUALLY CORRECT UP TO TWENTY FIVE ERRORS!! SURELY AFTER, SAY, THREE ERRORS IT WILL BREAK?
```

Figure 5: The same BCH code encoding another English text and correcting 12 errors.

References

[1] David S. Dummit and Richard M. Foote. *Abstract Algebra*. John Wiley & Sons, 3rd edition, 2004.

[2] John Gill. Ee387 notes #7, handout #28. `https://web.archive.org/web/20140630172526/http://web.stanford.edu/class/ee387/handouts/notes7.pdf`. Accessed: 2024-12-29.

[3] Yunghsiang S. Han. Bch codes. `https://web.ntpu.edu.tw/~yshan/BCH_code.pdf`. Slides 22 to 30. Accessed: 2024-12-29.

[4] Pedro Martins. Coding theory and cryptography notes, chapter 7: Finite fields. `https://www.math.tecnico.ulisboa.pt/~pmartins/CTC/CTC21Notes7.pdf`, 2021. Propositions 33 and 36. Accessed: 2024-12-28.

[5] W. Peterson. Encoding and error-correction procedures for the bose-chaudhuri codes. *IRE Transactions on Information Theory*, 6(4):459–470, 1960.

[6] Wikipedia contributors. Bch code — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/BCH_code`, 2024. Accessed: 2024-12-28.

A. Appendix: BCH Code Implementation

For a detailed walk-through of each part of the code, as well as testing results, refer to Section 5.

A.1. Dependencies and subroutines

```python
from sympy import *
from sympy.abc import x, alpha
from sympy import lcm
import numpy as np
import random
import copy
from itertools import *
import galois

def order(a,p):
    cur = a%p
    counter =0
    while True:
        counter +=1
        if (cur==1):
            return counter
        cur = (cur*a)%p
    raise Exception("Couldn't determine order of poly")

def fastExp(poly, exp, irr_poly):
    if exp == 0:
        return Poly(1, x, domain = poly.domain)
    if exp == 1:
        return poly%irr_poly
    res = fastExp(poly, exp//2, irr_poly)
    res = (res*res)%irr_poly
    if exp%2==1:
        res = (res * poly)%irr_poly
    return res

def findNthRoot(q, m, irr_poly, n):
    assert((q**m -1)%n==0)
    for i in range(2*(q**m)):
        coeffs = [0] + [random.randint(0,q-1) for _ in range(m)]
        if sum(coeffs)==0:
            continue
        poly = Poly(coeffs, x, domain = GF(q))
        r = order(poly, irr_poly)
        if r%n==0:
            return fastExp(poly, (r//n), irr_poly)
    raise Exception("Couldn't determine nth root")
```

```python
def evaluatePoly(poly, min_poly, q, m, irr_poly):#poly has variable x, min_poly has alpha
    res = Poly(0, x, domain = GF(q))
    for i in min_poly.all_terms():
        if i[1]==0:
            continue
        res = (res + i[1]*fastExp(poly, i[0][0], irr_poly))%irr_poly
    return res

# return minimum polynomial of poly in GF(q**m) with reducing polynomial irr_poly
def minPoly(poly, q, m, irr_poly):#poly and irr_poly are both in variable x
    for i in range(1,m+1):
        if m%i!=0:
            continue
        for coeffs in product(range(q), repeat = i):
            candidate_poly = Poly([1]+list(coeffs), alpha, domain = GF(q))
            if evaluatePoly(poly, candidate_poly, q, m, irr_poly).is_zero:
                return candidate_poly
    raise Exception("Couldn't find minimal polynomial")

def polyToInt(poly, q):
    return sum(int((c%q)*(q**i)) for i,c in enumerate(reversed(poly.all_coeffs())))

def intToPoly(k, q):
    deg = 0
    res = Poly(0,x,domain = GF(q))
    while(k>0):
        res = res + Poly((k%q)*x**deg, x, domain = GF(q))
        deg = deg+1
        k = k//q
    return res
```

A.2. Main BCH Code Implementation

```python
class BCHCode:
    def __init__(self, q, n, d, c):
        assert(isprime(q))
        assert (np.gcd(q, n)==1)
        assert(d<n), "The value of d is too large."
        self.q = q
        self.n = n
        self.d = d
        self.c = c
        self.m = order(q,n)
        self.t = (d-1)//2
        self.elem = []
        self.gf = galois.GF(self.q,self.m)
        (self.g_poly,self.irr_poly, self.nth_root) = self.__gen()

    def __gen(self):
        coeffs = [int(c) for c in self.gf.irreducible_poly.coeffs]
        irr_poly = Poly(coeffs, x, domain = GF(self.q))
        nth_root = findNthRoot(self.q, self.m, irr_poly, self.n)
        self.__calcElem(nth_root, irr_poly)
        g_poly = Poly(1, alpha, domain = GF(self.q))
        current_root =  fastExp(nth_root, self.c, irr_poly)
        for i in range(self.c, self.c + self.d -1):
            g_poly = lcm(g_poly, minPoly(current_root, self.q, self.m, irr_poly))
            current_root = (current_root*nth_root)%irr_poly
        return(g_poly, irr_poly, nth_root)

    def __calcElem(self, nth_root, irr_poly):#called once during __gen
        cur = Poly(1, x, domain = GF(self.q))
        for i in range(self.n):
            self.elem.append(cur)
            cur = (cur * nth_root)%irr_poly

    #message is a q-ary list of any size at most n - g_poly.degree. Automatically padded with zeros to the right
    def encode(self, message):
        assert(len(message)+self.g_poly.degree(alpha)<=self.n), "Message too long!"
        m_poly = Poly(message, alpha, domain = GF(self.q))*Poly(alpha**(self.n - len(message)), alpha, domain = GF(self.q))
        r_poly = m_poly % self.g_poly
        m_poly = m_poly-r_poly
        code = [0]*(self.n - m_poly.degree(alpha)-1)
        for i in m_poly.all_coeffs():
            code.append(i%self.q)
        return code

    #code is a q-ary list of size n
    def decode(self, code):
        code1 = copy.deepcopy(code)
        code_poly = Poly(code1, alpha, domain = GF(self.q))
        syndromes = []
        clean = True
```

```python
        for i in range(self.c, self.c + self.d - 1):
            s = evaluatePoly( self.elem[i%self.n], code_poly, self.q, self.m, self.irr_poly)
            syndromes.append(s)#poly in x variable
            if not s.is_zero:
                clean = False
        if clean:
            print("No errors detected!")
            return code1[:self.n - self.g_poly.degree(alpha)]

        S = self.gf([[polyToInt(syndromes[i+j], self.q) for j in range(self.t)] for i in range(self.t)])
        while np.linalg.det(S)==0 and S.size>1:
            S = S[:-1, :-1]

        v = S.shape[0]
        C = self.gf([[polyToInt(syndromes[i+v], self.q)] for i in range(v)])
        print(f"Not clean! Detected {v} errors.")
        Inv = np.linalg.inv(S)
        lambdas = Inv @ C

        error_locator_poly = Poly(-1, x, domain = GF(self.q))#-1 since we solved for -lambda instead
        for i in range(len(lambdas)):
            error_locator_poly = error_locator_poly + intToPoly(int(lambdas[i][0]),self.q)*(Poly(alpha**(v-i), alpha, domain = GF(self.q)))

        #test each power of nth_root to find the solutions
        error_pos = []
        for j in range(self.n):#try nth_root**j
            res = Poly(0, x, domain = GF(self.q))
            for i in error_locator_poly.terms():
                if i[1]==0:
                    continue
                res = (res + i[1]*fastExp(self.elem[j], i[0][1], self.irr_poly)*Poly(x**i[0][0], x, domain = GF(self.q)))%self.irr_poly
            if res.is_zero:
                print("Error found at position: ", (j-1)%self.n)
                if self.q ==2:
                    code1[(j-1)%self.n] = 1 if code1[(j-1)%self.n] ==0 else 0
                else:
                    error_pos.append((self.n-j)%self.n)
        if self.q==2:
            return code1[:self.n - self.g_poly.degree(alpha)]

        # Solve final linear system
        S = self.gf([[polyToInt( fastExp(self.elem[error_pos[j]],
        ↪  self.c+i, self.irr_poly), self.q) for j in range(v)] for i in
        ↪  range(v)])
        C = self.gf([[polyToInt(syndromes[i], self.q)] for i in
        ↪  range(v)])
        Inv = np.linalg.inv(S)
        errors = Inv @ C

        for j in range(v):
            code1[(self.n-error_pos[j]-1)%self.n] =
            ↪  (code1[(self.n-error_pos[j]-1)%self.n] -
            ↪  int(errors[j][0]))%self.q
        return code1[:self.n - self.g_poly.degree(alpha)]
```

A.3. Auxiliary testing functions

```
def f(c):
    if c==' ':
        return 0
    elif c=='.':
        return 27
    elif c==',':
        return 28
    elif c=='?':
        return 29
    elif c=='!':
        return 30
    else:
        return ord(c)-64

def fInv(i):
    if i==0:
        return ' '
    elif i==27:
        return '.'
    elif i==28:
        return ','
    elif i==29:
        return '?'
    elif i==30:
        return '!'
    else:
        return chr(i+64)

def englishToCode(message):
```

```python
    return [f(c)for c in message]

def codeToEnglish(code):
    return ''.join([fInv(i) for i in code])

def testBCH(bch, message, english= False):
    if english:
        assert(bch.q==31), "For english text to work, we need to be working over q=31."
        length = len(message)
        message = englishToCode(message)
    code = bch.encode(message)
    number_of_errors = random.randint(1, bch.t)
    if english:
        assert(number_of_errors<=length), "There are more errors than characters in the message, please choose a longer message."
        error_pos = random.sample(range(length), number_of_errors)
    else:
        error_pos = random.sample(range(bch.n), number_of_errors)
    error_value = [random.randint(1,bch.q-1) for _ in range(number_of_errors)]
    if number_of_errors==1:
        print(number_of_errors, "error was introduced at position", error_pos, "with value", error_value)
    else:
        print(number_of_errors, "errors were introduced at positions", error_pos, "with values", error_value)

    bad_code = copy.deepcopy(code)
    for i in range(number_of_errors):
        bad_code[error_pos[i]]= (bad_code[error_pos[i]]+error_value[i])%bch.q
    if english:
        print("Broken message:", codeToEnglish(bad_code)[:length])
    print("Decoding logs {")
    decoded = bch.decode(bad_code)
    if english:
        print("}\nDecoded message: ",codeToEnglish(decoded)[:length])
    else:
        print("}\nDecoded message: ",decoded)

```