Information Theory, Project 1

Sergey Ichtchenko

Mini-project, Michaelmas Term 2024

Contents

1 Abstract

In this report, we will be describing the construction of BCH codes along with their implementation. The sections below will all be referring to the source code contained in Appendix A. This report will consider a specific subset of BCH codes called *binary narrow-sense BCH codes*.

2 Construction of the code

BCH codes are a sub-class of cyclic codes. Traditionally, codes are first invented and constructed, after which their error correcting capability can be determined. However, in the case of BCH codes, we work in the other direction: we first specify the number of random errors we want our code to correct, after which we construct our parameters that allow us to encode and decode strings of a certain length to correct the specified number of errors. The parameters that we need to construct for a BCH code are the primitive polynomial, a primitive element, and the generator polynomial.

2.1 Mathematical background

To construct a BCH code, we require some theory about polynomials in finite fields. We assume knowledge of basic abstract algebra and field theory.

Definition 1: $\mathrm{GF}(q)$

Given any prime q, the Galois field $\mathrm{GF}(q)$ is a finite field of q elements.

From now on, we will focus our attention on the specific case of $q = 2$, as we are constructing binary BCH codes. In this case, $\mathrm{GF}(2)$ consists of the elements $0, 1$ with addition and multiplication defined modulo 2 as usual.

To encode and decode BCH codes, we want to utilise polynomials, which are defined as follows:

Definition 2: Polynomials over $\mathrm{GF}(2)$

- A polynomial $f(x) \in \mathrm{GF}(2)[x]$ is of the form $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_2x^2 + a_1x + a_0$, where $a_i \in \mathrm{GF}(2)$.
- Polynomials in $\mathrm{GF}(2)$ are added and multiplied like normal polynomials, but have their coefficients reduced modulo 2.
- We say that a polynomial $f \in \mathrm{GF}(2)[x]$ is *divisible* by a polynomial $g \in \mathrm{GF}(2)[x]$ if there exists a polynomial $h \in \mathrm{GF}(2)[x]$ such that $f(x) = g(x) \cdot h(x) + 0$, i.e. when the remainder of division of f by g is zero.
- We say that a polynomial $f \in \mathrm{GF}(2)[x]$ is *irreducible* if it is not divisible by a polynomial of degree m, where $\deg(f) < m < 0$
- We say that an irreducible polynomial $f \in \mathrm{GF}(2)[x]$ of degree m is *primitive* if the smallest integer n for which f divides $x^n + 1$ is $n = 2^m - 1$.

It is important to note that any irreducible polynomial of degree m divides $x^{2^m-1} + 1$, but primitive polynomials do not have any factors of the same form with a smaller exponent.

We now want to construct an extension field $\mathrm{GF}(2^m)$ of the base field $\mathrm{GF}(2)$ for any m. It can be shown that such a field exists and can be constructed as follows [3]:

Theorem 3: Construction of $\mathrm{GF}(2^m)$

Let $p(x)$ be a primitive polynomial of degree m. It can be shown that primitive polynomials exist for every such degree.
Let α be a root of the polynomial, $p(\alpha) = 0$. Then, the set

$$F = \{0, 1, \alpha, \alpha^2, ..., \alpha^{2^m-2}\}$$

forms a field of 2^m elements, where multiplication of elements is defined as addition of exponents.

Proof:

We note that since p divides $x^{2^m-1} + 1$, we have that

$$\begin{aligned} x^{2^m-1} + 1 &= h(x)p(x) \\ \alpha^{2^m-1} + 1 &= h(\alpha)p(\alpha) = h(\alpha) \cdot 0 = 0 \\ \alpha^{2^m-1} &= 1 \end{aligned}$$

Thus the set F is closed under multiplication.
To define addition, we note that for all exponents i we can divide x^i by p to get

$$x^i = h(x)p(x) + r(x)$$

where $r(x)$ is a polynomial of degree $m-1$ or less. Then, substituting α, we get

$$\alpha^i = h(\alpha)p(\alpha) + r(\alpha) = 0 + r(\alpha) = r(\alpha)$$

Thus, every power of α can be represented as a polynomial in α. It can be shown that all of these polynomials are distinct. Addition of elements in F is now defined as regular polynomial addition in $\mathrm{GF}(2)$.

Corollary 4: Isomorphism of extension field

The extension field $\mathrm{GF}(2^m)$ is isomorphic to the vector space $\mathrm{GF}(2)^m$ and to the field of polynomials $\mathrm{GF}(2)[\alpha]$ modulo some primitive polynomial of degree m.

With these prerequisites, we are ready to implement the BCH code in practice.

2.2 Primitive element and polynomial

Our first step is to generate a primitive element α for a primitive polynomial $p(x)$. We have seen above that primitive polynomials exist for every length m, but we must now define how to do this in practice. Our approach is to generate random irreducible polynomials and see whether they are primitive or not. The `sympy` function `sympy.polys.galoistools.gf_irreducible` is able to generate random irreducible polynomials of a specified degree. To test whether a polynomial is primitive, we must test whether it is divisible by $x^i + 1$ for all values $1 \leq i \leq n - 1$. If our random irreducible polynomial is a factor of any of these, then we regenerate the polynomial and try again. Otherwise, we know that the polynomial is primitive and use it for encoding and decoding. This logic is implemented in the `find_primitive` function on page 15. By definition, the symbol α is the primitive element which we represent as the `sympy` variable z in code.

If a user does not specify a primitive polynomial when running the encoding, the generated polynomial will be output and must be specified for the decoding (as the same polynomial has to be used for encoding and decoding the message).

2.3 Generator polynomial

The next step is to determine a generator polynomial for our BCH code. The generator polynomial determines how many errors we can correct and is used for encoding and decoding messages.

Definition 5: Generator polynomial

Let t be the number of errors our BCH code is able to correct.

The generator polynomial $g(x) \in \mathrm{GF}(2)[x]$ is defined as the lowest-degree polynomial of GF(2) that has $\alpha, \alpha^2, ..., \alpha^{2t}$ as its roots.

To find the generator polynomial, we must first find the minimal polynomials $\phi_i(x)$ of α^i for each $1 \leq i \leq 2t$. Then, the generator polynomial can be calculated by evaluating

$$g(x) = \mathrm{LCM}\{\phi_1(x), \phi_2(x), ..., \phi_{2t}(x)\}$$

From Theorem 2.14 in [3] we know that if $\phi(x)$ is the minimal polynomial of some element β in $\mathrm{GF}(2^m)$, and e is the smallest integer such that $\beta^{2^e} = \beta$, then

$$\phi(x) = \prod_{i=0}^{e-1}(x + \beta^{2^i})$$

Definition 6: BCH code

A binary narrow-sense BCH code consists of a primitive element α, primitive polynomial $p(x)$, and generator polynomial $g(x)$ constructed as above.

The BCH code has the following characteristics:

- m, the exponent of the extension Galois field
- $n = 2^m - 1$, the length of the codewords generated
- t, the number of errors that the code can correct
- $c = \deg(g)$, the degree of the generator polynomial, which indicates how many checksum bits the code will generate
- $k = n - c$, the number of data bits that can be encoded at one time

We say that a code is a $[n, k]$ BCH code if it has the parameters n and k as described above.

This logic is implemented in the `find_generator` and `find_minimal_polynomial` functions on page 16. The `find_generator` function takes the LCM of the minimal polynomials of α^i for all i from 1 to $2t$ using the `sympy.polys.galoistools.gf_lcm` function. The `find_minimal_polynomial` takes in an element β. It then multiplies together all polynomials $x - \beta^{2^i} \in \mathrm{GF}(2)[z][x]$, which are polynomials in x with coefficients in $\mathrm{GF}(2)[z]$, equivalent to having coefficients in $\mathrm{GF}(2^m)$. The loop stops when the algorithm finds a previous coefficient, which is guaranteed to happen due to the properties of our finite field. Afterwards, the result is expanded using the `expand_expression` function on page 17 and each coefficient is reduced modulo the primitive element. The resulting coefficient is guaranteed to be in $\mathrm{GF}(2)$ instead of $\mathrm{GF}(2)[z]$ by the aforementioned theorem.

3 Encoding using the code

There are two ways of encoding a message using a BCH code: either using a systematic encoding, where the message appears verbatim in the code, and a non-systematic encoding, where the codeword does not contain the message. For this project, we will be using the non-systematic encoding, as the systematic encoding is indistinguishable from a CRC checksum with binary BCH codes over GF(2) [1].

To do the encoding, we simply multiply the polynomial representation of the message $m(x)$ by the generator polynomial $g(x)$ to get the codeword $s(x)$:

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

This is implemented in the `encode` function on page 18, which does the multiplication, converts the polynomial back to an array and pads it with the right amount of zeros using the `fill_data` function on page 23.

4 Decoding using the code

We will now move on to decoding the code. While decoding a correct codeword is easy, as it just requires one to reverse the multiplication done by the encoding function, correcting possible errors requires some effort. The decoding logic is contained in the `decode` function on page 18, which chains together the functions we will see below.

4.1 Detecting errors

Errors in a BCH codeword are detected using *syndromes*. Notice that, if an error occurs in transmission, the received polynomial r can be represented as

$$r(x) = s(x) + e(x)$$

where e is the error polynomial, having a coefficient of 1 everywhere where an error occurred, $e(x) = x^{i_1} + ... + x^{i_\nu}$, where ν is the number of errors that have occurred. Recall that the generator polynomial, $g(x)$, is defined as a polynomial with zeroes at $\alpha^1, ...\alpha^{2t}$. Since the codeword s is a multiple of g, we also know that

$$r(\alpha^i) = s(\alpha^i) + e(\alpha^i) = 0 + e(\alpha^i) = e(\alpha^i)$$

We call these values $S_i := r(\alpha^i)$ the syndromes of the message. In particular, if no errors occurred, then we know that $e(x) = 0$ and all of the syndromes will be equal to zero also. On the other hand, if errors have occurred, then calculating the syndromes will let us isolate the error vector and find the error locations in the following steps.

The `find_syndromes` function on page 19 evaluates and returns all of these syndromes. It uses the `substitute` function on page 21 to substitute one polynomial into another.

4.2 Error locator polynomial

Suppose exactly ν errors have occurred in transmission. We will follow the method in [2] to construct and determine the coefficients of an error locator polynomial.

Recall from above that the syndromes were determined by evaluating the error polynomial at α^i. In particular, $S_1 = e(\alpha) = \alpha^{i_1} + ... + \alpha^{i_\nu}$ Define the error locations as $X_i = \alpha^{i_k}, i \in \{1, ..., \nu\}$. We can rewrite the syndromes determined above as follows:

$$
\begin{aligned}
S_1 &= X_1 + X_2 + ... + X_\nu \\
S_2 &= X_1^2 + X_2^2 + ... + X_\nu^2 \\
&\vdots \\
S_{2t} &= X_1^{2t} + X_2^{2t} + ... + X_\nu^{2t}
\end{aligned}
$$

The error locator polynomial is defined as a polynomial with coefficients in $\mathrm{GF}(2^m)$, or equivalently in $\mathrm{GF}(2)[z]$, such that its roots are the inverse error locations

$$
\begin{aligned}
\Lambda(x) &= \Lambda_\nu x^\nu + \Lambda_{\nu-1}x^{\nu-1} + ... + \Lambda_1 x^1 + 1 \\
&= (1 - xX_1)(1 - xX_2)...(1 - xX_\nu)
\end{aligned}
$$

Using this definition, we can determine the following relation for all $i \in \{1, 2, ..., \nu\}$:

$$\Lambda_1 S_{i+\nu-1} + \Lambda_2 S_{i+\nu-2} + ... + \Lambda_\nu S_i = -S_{i+\nu}$$

Rewriting this in matrix form results in the following:

$$
\begin{bmatrix}
S_1 & S_2 & S_3 & \cdots & S_{\nu-1} & S_\nu \\
S_2 & S_3 & S_4 & \cdots & S_\nu & S_{\nu+1} \\
S_3 & S_4 & S_5 & \cdots & S_{\nu+1} & S_{\nu+2} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
S_{\nu-1} & S_\nu & S_{\nu+1} & \cdots & S_{2\nu-3} & S_{2\nu-2} \\
S_\nu & S_{\nu+1} & S_{\nu+2} & \cdots & S_{2\nu-2} & S_{2\nu-1}
\end{bmatrix}
\begin{bmatrix}
\Lambda_\nu \\ \Lambda_{\nu-1} \\ \Lambda_{\nu-2} \\ \vdots \\ \Lambda_2 \\ \Lambda_1
\end{bmatrix}
=
\begin{bmatrix}
-S_{\nu+1} \\ -S_{\nu+2} \\ -S_{\nu+3} \\ \vdots \\ -S_{2\nu-1} \\ -S_{2\nu}
\end{bmatrix}
$$

It can be shown that this matrix is nonsingular if and only if there are ν errors [2]. Thus to find out how many errors have occurred given that there is at least one error, it suffices to try to establish the above syndrome matrix for every size starting from t, the number of errors our code can correct, down to 1. The largest size for which the determinant of this matrix is nonzero is the number of errors that actually happened.

The function `find_error_locator` on page 19 implements this calculation. It first calculates the determinants in turn modulo the primitive polynomial and continues to the next phase when the determinant is nonzero. Next, it augments the matrix by the syndrome vector shown above and finds the RREF of the matrix using SymPy's `rref` function. As SymPy cannot execute the calculations in our extension field itself, we must expand the output of the RREF calculation by evaluating powers, fractions, products, and sums. This is done by the `expand_expression` function on page 17. To calculate inverses in the extension field, the function uses the `find_inverse` function on page 21 which uses the `sympy.polys.galoistools.gf_gcdex` function to apply the extended Euclidean algorithm to find an inverse function modulo the primitive polynomial. For example, if we wanted to calculate the inverse of a polynomial $f(x)$ modulo the primitive polynomial $p(x)$, we use the extended Euclidean algorithm to calculate polynomials $a(x)$ and $b(x)$ such that

$$f(x)a(x) + p(x)b(x) = 1$$

This is always possible as p is a primitive polynomial and f is a nonzero polynomial modulo p. Then, $a(x)$ is returned as the inverse of $f(x)$.

Finally, after expanding, the function returns the coefficients of the error locator polynomial.

4.3 Finding the error locations

To find the error locations $X_1, ..., X_\nu$, we must construct the error locator polynomial, solve for its roots, and find the exponent of their inverse to get the locations of the errors. This is implemented in the `find_error_pos` function on page 20.

The function first constructs the locator polynomial as a polynomial in $\mathrm{GF}(2)[z][x]$ with the coefficients calculated above, and then finds all roots by enumerating powers of α, implemented in the `find_all_roots` function on page 22. It then finds the inverse of each root using the `find_inverse` function again, and looks up the power of α this inverse corresponds to using a lookup table. This lookup table is generated by the `find_all_powers` function on page 22.

4.4 Correcting and decoding

Finally, the `decode` function constructs the error polynomial $e(x) = x^{i_1} + ... + x^{i_\nu}$ using the values for $i_1, ..., i_\nu$ it found in the previous function, and adds this to the received polynomial $r(x)$ to get the corrected polynomial $m(x)$.

The `decode_correct_code` function on page 23 then simply divides this by the generator and padding the result with an adequate number of zeroes.

5 Demonstration

Let us demonstrate the functionality of this implementation by a few examples. The `main` function on page 25 is called when the script is run and provides a command-line interface for encoding, decoding, and testing of the error-correcting capabilities of the BCH code implementation.

```
$ python3 ./bch.py
usage: BCH [-h] -m EXPONENT -t ERRORS_CORRECTED [-p PRIMITIVE] [-e ENCODE | -d DECODE | -x]
```

5.1 Encoding and decoding ASCII messages

Let us start by encoding and decoding messages and introducing errors manually. First, we will encode a single 7-bit ASCII character using a $[15, 7]$ BCH code.

```
$ python3 ./bch.py -m 4 -t 2 -e "A"
111010110010001
primitive: 10011
```

Our output codeword is the 15-bit string 111010110010001 as expected, and the primitive polynomial is 10011. In reality, we would not have to also transmit the primitive polynomial when sending the message, as that would be agreed upon between parties beforehand. However, for these examples we will generate a new primitive polynomial every time we encode a message. Decoding the correct codeword works as expected:

```
$ python3 ./bch.py -m 4 -t 2 -p 10011 -d "111010110010001"
A
```

Decoding also works when there are up to two errors in the transmitted message:

```
$ python3 ./bch.py -m 4 -t 2 -p 10011 -d "111010110010001"
A

$ python3 ./bch.py -m 4 -t 2 -p 10011 -d "110010110010001"
A

$ python3 ./bch.py -m 4 -t 2 -p 10011 -d "111000110010101"
A

$ python3 ./bch.py -m 4 -t 2 -p 10011 -d "111010110010010"
A
```

```
$ python3 ./bch.py -m 4 -t 2 -p 10011 -d "001010110010001"
A
```

However, if more than two errors occur, the distance of the erroneous codeword will be closer to some other valid codeword and the wrong message will be returned. As such, this BCH code cannot detect or correct more than the two errors it was designed for.

```
$ python3 ./bch.py -m 4 -t 2 -p 10011 -d "111010110010110"
v
```

```
$ python3 ./bch.py -m 4 -t 2 -p 10011 -d "111011111011001"
X
```

We can also vary the length of the encoded string, the size of the Galois field, and the number of errors corrected. Note that in the previous example, the length of one ASCII character (7 bits) precisely matched the length of the message accepted by the BCH code. However, this is not the case in general. When the length of the encoded message is not equal to the length accepted by the BCH code, the `main` function will add zeroes to the end for padding and split the message into chunks of length $k = 2^m - 1 - \deg(g)$, where $\deg(g)$ is the degree of the generator polynomial constructed. Below are some examples of longer codes and messages being encoded and decoded correctly, with errors highlighted.

The first example is a $[15, 1]$ BCH code, which has 1 data bit and 14 checksum bits, but which can recover from 7 errors per block. This is also known as a trivial repetition code.

```
$ python3 ./bch.py -m 4 -t 7 -e "S"
111111111111111000000000000000111111111111111000000000000000000000000000000111111111111111
111111111111111
primitive: 10011
```

```
$ python3 ./bch.py -m 4 -t 7 -p 10011 -d
"111111111111111000000000000000111111111111111000000000000000000000000000000111111111111111
111111111111111"
S
```

```
$ python3 ./bch.py -m 4 -t 7 -p 10011 -d
"1111111111111111111111000000000000000111111110000000000000000000000000000000
1010101010101011111111111111111"
S
```

Here is another example with a longer piece of text being encoded with a $[31, 11]$ BCH code that can correct 4 errors:

```
$ python3 ./bch.py -m 5 -t 4 -e "The rain in Spain falls mainly on the plain!"
10010111100101110111111101101001000111000000011110100100110000101011001000100001001010010101
1001000101110011000001101110101011010101000000000010110001001101100101100101001000000001111
1111101010100000000000101100010011011001011001101000011010001000000100110110001000110101110
0101101011011011101000000100110001111001001100100111100110110010000011010000011100111001101
110010001111011101010101011101001111101010110111001101110010001111000000111001100001000011100
00001111010010000001111011100111110111101000111101101110100000010011000111100100110010010101
0111111111101001110111001001001000110010100001100100111011001001010101111111111101101110011
010111100111000110110010110010001000010010100101011010101110111010111000000010001011010001
111111011101001010001000001100010000110011000011011111101110011100110111001000111100001100
11110111100011111100001011011000000001000101101011101110101
primitive: 100101
```

```
$ python3 ./bch.py -m 5 -t 4 -p 100101 -d
"1001011110010111011111110110100100011100000001111010010011000010101100100010000100101001010
1100100010111001100000110111010101101010100000000001011000100110110010110010100100000000111
1111110101010000000000010110001001101100101100110100001101000100000010011011000100011010111
0010110101101101110100000010011000111100100110010011110011011001000001101000001110011100110
111001000111101110101010101110100111110101011011100110111001000111100000011100110000100001110
00000111101001000000111101110011111011110100011110110111010000001001100011110010011001001010
1011111111110100111011100100100100011001010000110010011101100100101010111111111110110111001
101011110011100011011001011001000100001001010010101101010111011101011100000001000101101000
111111101110100101000100000110001000011001100001101111110111001110011011100100011110000110
011110111100011111100001011011000000001000101101011101110101"
The rain in Spain falls mainly on the plain!
```

```
$ python3 ./bch.py -m 5 -t 4 -p 100101 -d
"1001011110100100011111110110100100011100000001111010010011000010101100100010000100101001010
01100100010111001100000110111010101101010100000000001011000100110110010110010100100000000011
1111111010101000000000001011000100110110010110011010000110100010000001001101100010001101011
1001011010110110111010000001001100011110010011001001111001101100100000110100000111001110011
011100100011110111010101010111010011111010101101110011011100100011110000001110011000010000111
00000011110100100000011110111001111101111010001111011011101000000100110001111001001100100101
0101111111111010011101110010010010001100101000011001001110110010010101011111111111011011100
```

```
11010111100111000110110010110010001000010010100101011010101110111010111000000010001011010001111111011101001010001000001100010000110011000011011111110111001110011011100100011110000110011110111100011111110000101101100001110000101101011101110101"
The rain in Spain falls mainly on the plain!
```

5.2 Correcting all possible errors

Finally, we will demonstrate the error-correcting capabilities of the BCH code by exhaustively checking all of the possible errors for the $[15, 7]$ BCH code. The `test` function on page 24 generates a random message, encodes it, and then tries to correct every possible 1-bit and 2-bit error in the message. If any of the erroneous codewords get decoded to a different message, then the function throws an error.

```
$ python3 ./bch.py -m 4 -t 2 --test
Message: [1, 1, 1, 1, 0, 0, 0]
All 1-bit errors corrected!
All 2-bit errors corrected!

$ python3 ./bch.py -m 4 -t 2 --test
Message: [0, 1, 1, 0, 1, 1, 1]
All 1-bit errors corrected!
All 2-bit errors corrected!

$ python3 ./bch.py -m 4 -t 2 --test
Message: [1, 0, 1, 1, 1, 1, 0]
All 1-bit errors corrected!
All 2-bit errors corrected!
```

In all of the generated test cases, all 1-bit and 2-bit errors were successfully corrected and no errors were thrown.

References

[1] *BCH code*. In: *Wikipedia*. Page Version ID: 1254730124. 1st Nov. 2024. URL: https://en.wikipedia.org/w/index.php?title=BCH_code&oldid=1254730124 (visited on 07/01/2025).

[2] Ranjan Bose. *Module #23: Bose-Chaudhuri Hocquenghem (BCH) Codes*. Information Theory, Coding and Cryptography. 28th Oct. 2018. URL: https://nptel.ac.in/courses/108102117 (visited on 25/12/2024).

[3] Shu Lin and Daniel Costello. *Error Control Coding*. Jan. 2004. ISBN: 978-1-4613-6787-1. DOI: 10.1007/978-1-4615-3998-8_3.

A Source Code

```python
"""BCH encoding and decoding project.

Sergey Ichtchenko
University of Oxford
Information Theory
MT24 mini-project
"""

import argparse
import random
import sys
import warnings
from sympy import GF, ZZ, Matrix, Poly, div, Symbol, Add, Mul, Pow, Integer, degree
from sympy.abc import x, z
from sympy.polys.galoistools import gf_gcdex, gf_lcm, gf_irreducible

warnings.filterwarnings("ignore", category=DeprecationWarning)


class BCH:
    """BCH code properties and functionality.

    On initialisation, constructs the required parameters for the BCH code.
    Encoding can be called using the `encode` function,
    and error-corrected decoding using the `decode` function.

    Attributes:
        m: The exponent of the Galois field size
        n: The codeword length
        t: The number of errors the code is able to correct
        c: The number of checksum bits in the code
        k: The number of data bits that can be encoded
        primitive: The primitive polynomial of the Galois field
        alpha: A primitive element of the Galois field
        generator: The generator polynomial used by the BCH code
    """
```

```python
def __init__(self, m, t, primitive=None):
    """Initialises the BCH code with the given parameters.

    Args:
        m: The exponent of the Galois field size. The codeword length is m**2-1
        t: The number of errors to correct
        primitive: An integer representing a primitive polynomial (optional)
    """
    self.m = m
    self.n = 2**m - 1
    self.t = t

    if not primitive:
        self.primitive = self.find_primitive()
    else:
        self.primitive = Poly([int(x) for x in primitive], z, domain=GF(2))
    self.alpha = Poly(z, z, domain=GF(2))
    self.generator = self.find_generator()

    self.c = degree(self.generator)
    self.k = self.n - self.c


def find_primitive(self):
    """Find a primitive polynomial for the Galois field

    Returns:
        A primitive polynomial for the Galois field
    """
    while True:
        irreducible = Poly(gf_irreducible(self.m, 2, ZZ), z, domain=GF(2))
        for i in range(1, self.n):
            test_poly = Poly(z**i - 1, z, domain=GF(2))
            _, remainder = div(test_poly, irreducible)
            if remainder == 0:
                break
        else:
            return irreducible
```

```python
def find_generator(self):
    """Find a generator polynomial for the Galois field

    Returns:
        A generator polynomial based on the primitive polynomial and alpha
    """
    generator = Poly(1, x, domain=GF(2))
    for i in range(1, 2*self.t):
        current = self.find_minimal_polynomial(self.alpha**i)
        generator = Poly(gf_lcm(
            generator.all_coeffs(), current.all_coeffs(), 2, ZZ
        ), x, domain=GF(2))
    return generator

def find_minimal_polynomial(self, element):
    """Find a minimal polynomial (mod the primitive polynomial) for a given element.
    Args:
        element: The element to find a minimal polynomial for. Usually a power of alpha.
    Returns:
        A minimal polynomial for the element
    """
    seen = set()
    i = 0
    result = Poly(1, x)
    while True:
        current = Poly(element**(2**i), z) % self.primitive
        if current in seen:
            break
        result *= Poly(x, x) - current.set_domain(ZZ)
        seen.add(current)
        i += 1

    result = Poly(result, x, domain=GF(2)[z])
    reduced = [
        self.expand_expression(coefficient)%self.primitive
        for coefficient in result.all_coeffs()
    ]
    assert(all(coefficient in (0,1) for coefficient in reduced))

    result = Poly(reduced, x, domain=GF(2))
    return result
```

```
def expand_expression(self, expression):
    """Expands an Expr type by evaluating all operations in the Galois field

    Args:
        expression: An Expr type to expand

    Returns:
        a Poly representing the evaluated expression
    """
    if isinstance(expression, Symbol):
        return Poly(expression, z, domain=GF(2)) % self.primitive

    if isinstance(expression, Integer):
        return Poly(expression, z, domain=GF(2)) % self.primitive

    if isinstance(expression, Mul):
        result = Poly(1, z, domain=GF(2))
        for term in expression.args:
            result *= self.expand_expression(term)
        return result % self.primitive

    if isinstance(expression, Add):
        result = Poly(0, z, domain=GF(2))
        for term in expression.args:
            result += self.expand_expression(term)
        return result % self.primitive

    if isinstance(expression, Pow):
        base, exponent = expression.args
        base = self.expand_expression(base)
        exponent = int(exponent)
        if exponent < 0:
            assert self.primitive is not None
            base = self.find_inverse(base)
            exponent = abs(exponent)

        return Poly(base**exponent, z, domain=GF(2)) % self.primitive
```

```python
def encode(self, bits):
    """Encode a message using the generated BCH code.

    Args:
        bits: A list containing the bits to encode

    Returns:
        A list containing the bits of the codeword
    """
    normalised = self.fill_data(bits, self.k)

    data = Poly(normalised, x, domain=GF(2))
    encoded = data * self.generator
    return self.fill_data(encoded.all_coeffs(), self.n)


def decode(self, bits):
    """Decode a codeword using the generated BCH code.

    Args:
        bits: A list containing the bits of the codeword

    Returns:
        A list containing the bits of the decoded message, corrected for errors
    """
    normalised = self.fill_data(bits, self.n)
    encoded = Poly(normalised, x, domain=GF(2))

    syndromes = self.find_syndromes(encoded)
    if all((syndrome == 0 for syndrome in syndromes)):
        return self.decode_correct_code(encoded)

    locator = self.find_error_locator(syndromes)
    errors = self.find_error_pos(locator)

    for error in errors:
        encoded += Poly(x**error, x, domain=GF(2))

    return self.decode_correct_code(encoded)
```

```python
def find_syndromes(self, encoded):
    """Find the syndromes of a codeword

    Args:
        A polynomial representing any codeword
    Returns:
        A list of syndromes of the polynomial.
        If no errors have occurred, all syndromes are zero.
    """
    syndromes = []
    for i in range(1,2*self.t+1):
        syndrome = self.substitute(encoded, self.alpha**i)
        syndrome %= self.primitive
        syndromes.append(syndrome)
    return syndromes


def find_error_locator(self, syndromes):
    """Find the error locator polynomial given syndromes.

    Args:
        syndromes: A list of syndromes obtained from `find_syndromes`
    Returns:
        An error locator vector,
        where the entries are coefficients of the error locator polynomial
    """
    for i in range(self.t):
        nu = self.t-i  # Number of errors
        syndrome_matrix = Matrix(nu, nu, lambda a,b: syndromes[a+b].as_expr())
        detection = Poly(syndrome_matrix.det(), z, domain=GF(2)) % self.primitive
        if detection == 0:
            continue

        syndrome_vector = Matrix(nu, 1, lambda a,_: -syndromes[nu+a].as_expr())
        augmented = syndrome_matrix.col_insert(nu, syndrome_vector)
        locator = augmented.rref(pivots=False).col(nu)
        result = []
        for row in range(nu):
            result.append(self.expand_expression(locator[row]))

        return result
```

```python
def find_error_pos(self, locator):
    """Find the error positions based on the error locator vector.

    Args:
        locator: The error locator vector obtained from `find_error_locator`

    Returns:
        A list of positions where errors have occurred in the codeword
    """
    locator_poly = Poly(1, x, domain=GF(2)[z])
    for i, lambda_i in enumerate(locator[::-1], start=1):
        locator_poly += Poly(lambda_i%self.primitive, x, domain=GF(2)[z]) *\
        Poly(x**i, x, domain=GF(2)[z])
    roots = self.find_all_roots(locator_poly)

    alpha_powers = self.find_all_powers(self.alpha)
    result = []
    for root in roots:
        inverse = self.find_inverse(root) % self.primitive
        inverse_coefficients = inverse.all_coeffs()
        result.append(alpha_powers[tuple(inverse_coefficients)])
    return result
```

```python
def find_inverse(self, polynomial):
    """Finds the inverse of a polynomial modulo the primitive polynomial

    Args:
        polynomial: The polynomial to find the inverse for

    Returns:
        The inverse of the polynomial
    """
    inv, _, gcd = gf_gcdex(polynomial.all_coeffs(), self.primitive.all_coeffs(), 2, ZZ)
    assert gcd == [1]
    return Poly(inv, z, domain=GF(2))


def substitute(self, polynomial, substitution):
    """Substitute a polynomial into the variables of another.

    Args:
        polynomial: The polynomial to substitute into. This will have its variables replaced.
        substitution: The polynomial to insert

    Returns:
        The evaluated expression `polynomial(substitution(z))`
    """
    result = Poly(0, z, domain=GF(2))
    for i, coeff in enumerate(polynomial.all_coeffs()[::-1]):
        result += Poly(coeff, z, domain=GF(2)) * Poly(substitution**i, z, domain=GF(2))
    return result
```

```python
def find_all_powers(self, element):
    """Find all powers of an element in the Galois field.

    This is useful for looking up powers based on an expression later on.

    Args:
        element: The element to find powers for, usually alpha

    Returns:
        A dict containing the coefficients of polynomials as keys
        and exponents of `element` as values
    """
    result = {}
    for i in range(0, self.n):
        power = Poly(element**i, z, domain=GF(2)) % self.primitive
        if tuple(power.all_coeffs()) in result:
            continue
        result[tuple(power.all_coeffs())] = i
    return result


def find_all_roots(self, polynomial):
    """Find all the roots of a polynomial in the Galois field.

    Args:
        polynomial: The desired polynomial

    Returns:
        A list of roots in the Galois field.
        These are composed of powers of `alpha` reduced modulo the primitive polynomial.
    """
    roots = []
    for i in range(1,self.n+1):
        root = self.substitute(polynomial, self.alpha**i) % self.primitive
        if root == 0:
            roots.append(self.alpha**i % self.primitive)
    return roots
```

```python
def fill_data(self, data, length):
    """Prepend a list of bits with zeroes if the length is too short

    Args:
        data: The list of bits to complete with zeroes
        length: The desired length of the list

    Returns:
        A list containing the original data prepended with zeroes.
        If the length of the list is longer than the desired length, an error is thrown.
    """
    assert len(data) <= length
    return [0]*(length-len(data)) + data


def decode_correct_code(self, encoded):
    """Given a codeword that is known to be correct, decode it.

    Args:
        encoded: A polynomial which has had its errors corrected

    Returns:
        A list containing the decoded codeword
    """
    decoded, _ = div(encoded, self.generator)
    result = decoded.all_coeffs()
    return self.fill_data(result, self.k)
```

```python
def test(bch):
    """Code to test BCH functionality.
    Generates a BCH code, sends a random message, and tries to correct every possible
    1- and 2-bit error.
    """

    correct = [random.choice((0,1)) for _ in range(bch.k)]
    print("Message:", correct)
    encoded = bch.encode(correct)

    # Test all 1-bit errors
    for i, bit in enumerate(encoded):
        error = encoded[:i] + [1-bit] + encoded[i+1:]
        corrected = bch.decode(error)
        assert corrected == correct

    print("All 1-bit errors corrected!")

    # Test all 2-bit errors
    for i, bit1 in enumerate(encoded[:-1]):
        for j, bit2 in enumerate(encoded[i+1:], start=i+1):
            error = encoded[:i] + [1-bit1] + encoded[i+1:j] + [1-bit2] + encoded[j+1:]
            corrected = bch.decode(error)
            assert corrected == correct

    print("All 2-bit errors corrected!")
```

```
def main():
    parser = argparse.ArgumentParser(prog="BCH")
    parser.add_argument("-m", "--exponent", required=True)
    parser.add_argument("-t", "--errors-corrected", required=True)
    parser.add_argument("-p", "--primitive")
    data = parser.add_mutually_exclusive_group()
    data.add_argument("-e", "--encode")
    data.add_argument("-d", "--decode")
    data.add_argument("-x", "--test", action="store_true")
    args = parser.parse_args(sys.argv[1:])
    bch = BCH(int(args.exponent), int(args.errors_corrected), primitive=args.primitive)

    if args.test:
        test(bch)
    elif args.encode:
        message = args.encode.encode("ascii")
        bitstring = [int(a) for a in "".join([bin(character)[2:].zfill(7) for character in message])]
        padding_length = -len(bitstring) % bch.k
        bitstring += [0]*padding_length
        chunks = [
            bitstring[bch.k*i:bch.k*(i+1)] for i in range((len(bitstring) + bch.k - 1) // bch.k)
        ]
        result = [bch.encode(chunk) for chunk in chunks]
        print("".join(["".join([str(bit) for bit in chunk]) for chunk in result]))
        if not args.primitive:
            print("primitive:", "".join(str(x) for x in bch.primitive.all_coeffs()))
    elif args.decode:
        message = args.decode
        bitstring = [int(a) for a in message]
        chunks = [
            bitstring[bch.n*i:bch.n*(i+1)] for i in range((len(bitstring) + bch.n - 1) // bch.n)
        ]
        decoded = [bch.decode(chunk) for chunk in chunks]
        decoded_string = "".join(["".join([str(bit) for bit in chunk]) for chunk in decoded])
        padding_length = len(decoded_string) % 7
        decoded_string = decoded_string[:-padding_length] if padding_length>0 else decoded_string
        decoded_chunks = [
            decoded_string[7*i:7*(i+1)] for i in range((len(decoded_string) + 7 - 1) // 7)
        ]
        result = bytes([int(chunk, 2) for chunk in decoded_chunks])
        print(result.decode("ascii"))
```

```python
if __name__ == "__main__":
    main()
```