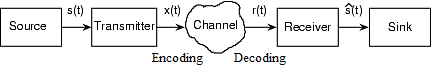
**Bose–Chaudhuri–Hocquenghem (BCH) Codes in MATLAB**

**Introduction**

Codes are used for data compression, cryptography, error-correction, and networking. This typically involves the removal or addition of redundancy and the correction or detection of errors in the transmitted data. Data compression attempts to compress the data from a source in order to transmit it more efficiently while Error Correction Codes (ECC) add extra data bits to make the transmission of data more robust to disturbances present in the transmission channel. The ordinary user may not be aware of the many applications using them. For example, a typical music CD uses the Reed-Solomon code to correct for scratches and dust.



**Figure 1 Structure of Communication System**

A noisy channel adds noise to the information. Any communication systems’ purpose is to convey the information with any degradation. For this efficient methods like Automatic Repeat Request (ARQ) and Forward Error Correction (FEC) are employed. The below explained coding and decoding process are implemented in the encoder and decoder as in the above Figure 1. In addition to these other techniques are also utilised to ensure data accuracy and integrity.

**Bose–Chaudhuri–Hocquenghem Codes**

The Bose–Chaudhuri–Hocquenghem (BCH) codes form a class of cyclic ECCs that are constructed using polynomials over a finite field. They were invented by Raj Bose and D. K. Ray-Chaudhuri, and independently by Alexis Hocquenghem. They are one of the most well-known binary multiple error detecting and correcting codes. It is possible to design binary BCH codes that can correct multiple bit errors.

**Introduction to Galois Fields (GF)**

A field is a set of elements on which the operations of addition and multiplication are defined. Well, known fields having an infinite number of elements include the real numbers *R*, the complex numbers *C*, and the rational numbers *Q*. Although the real, complex, and rational fields all have an infinite number of elements, finite fields also exist. If *p* is a prime number, then it is also possible to define a field with *pm* elements for any *m*. These fields are named after the French algebraist Evariste Galois. Representing data as a vector in a Galois Field (GF) allows mathematical operations to scramble data easily and effectively. Hence, they have many applications in coding theory. BCH Error Code is based on Galois Fields. BCH Codes can be found in electronic pagers and early versions of cell phones.

**Why Galois Fields in BCH?**

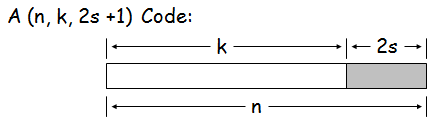
All computer systems operate on binary symbols. The binary symbols are strings of ones (1) and zeros (0) taken from Z2 (A field of binary numbers). However it only has 2 digits, so if each character is to be represented in the field, a bigger field is needed. Thus, Z2*m*is a good candidatewith ‘*m*’ to be determined. But it does not meet the criterion for inverses. Thus ordinary arithmetic fails during multiplication. The use of polynomial multiplication employing fields seems intuitive. This added along with the above-discussed advantages of Galois Fields make them suitable for BCH Codes.

**BCH Codes**

For any positive integer *m* ≥ 3 and *t* < 2*m*-1, there exists a binary BCH code with the following parameters. Figure 3 shows a diagrammatic representation.

Block length: *n* = 2*m* - 1  
Number of parity-check digits: *n* - *k* ≤ *m\*t*, where *k* is message length

Minimum distance: *dmin*≥ 2*t* + 1, where *t* is error correcting capacity



**Figure 3 BCH Codeword**

**The Encoding-Decoding-Error Correction Process**

**Generation of the Generator Polynomial**

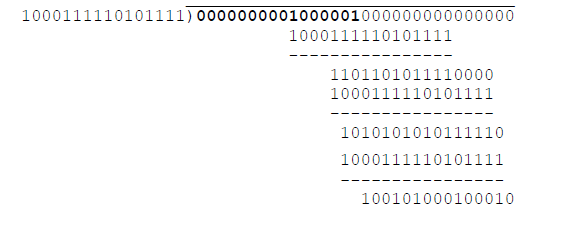
The generator polynomial is actually a combination of several polynomials corresponding to several powers of a primitive element of GF (2*m*). If α is a primitive element of GF (2*m*), the generator polynomial is the polynomial of a lowest degree over GF (2) with α, α*2*, α*3*, …, α*2t* as roots. The code word length is 2*m*-1 and ‘*t*’ is the number of correctable errors. The generator polynomial for few values of ‘*m*’ are shown in Table 1

**Table 1 Primitive polynomial for ‘*m*’**

|  |  |  |
| --- | --- | --- |
| ***m*** | **Default Primitive Polynomial** | **Integer Representation** |
| 1 | α + 1 | 3 |
| 2 | α2 + α + 1 | 7 |
| 3 | α3 + α + 1 | 11 |
| 4 | α4 + α + 1 | 19 |
| 5 | α5 + α2 +1 | 37 |
| 6 | α6 + α + 1 | 67 |

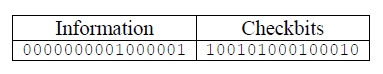
**Generation of the Code-Word**

The code word is formed by taking the remainder after dividing a polynomial representing the information by the generator polynomial, i.e. the information in binary (Bit Stream) is taken and then a number of zeros equal to the degree of generator polynomial are appended to the information bits. Then, it is divided by the generator polynomial using binary arithmetic as shown in Figure 4. The quotient generated here is of no use. Thus all code words are multiples of the generator polynomial. The remainder which has a degree less than the generator polynomial is represented in polynomial form.



**Figure 4 Generation of Check bits**

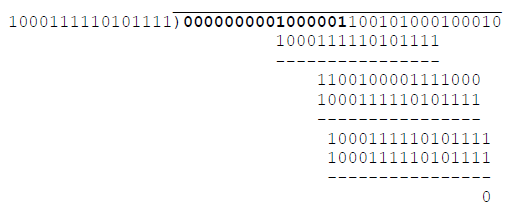
This remainder now acts as check bits, similar to parity bits in parity check codes. Now the information and check bits are arranged together so that they can be recognised as the resulting code word. This method is called systematic encoding as shown in Figure 5.



**Figure 5 Systematic Encoding**

**Error Detection:**

To test the code words for error, the code word is divided by the generator polynomial. The remainder is zero if there are no errors. If there were errors, then the remainder is non-zero. This error detection is illustrated in Figure 6. The remainder is called the syndrome and is used in further algorithms to actually locate the errant bits and correct them. As BCH is cyclic, and any cyclic shift in generated code word always results in a valid code word.



**Figure 6 Error Checking**

**Decoding Process:**

The decoding process where errors in the received codeword are located is a complicated process. Decoding involves,

1. Compute the syndrome from the received code word
2. Find the error location polynomial from a set of equations derived from the syndrome
3. Use the error location polynomial to identify errant bits and correct them

Let *v(x)* be the received code word, then *Si* = *v(x)* mod *m*i(*x*), where *mi(x)* is the minimal polynomial of α*i*. Thus

*S1* = *v(x)* mod *m*1(*x*)

*S2* = *v(x)* mod *m*2(*x*)

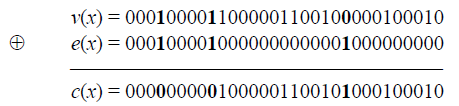
*S3* = *v(x)* mod *m*3(*x*)

*S4* = *v(x)* mod *m*4(*x*)

…

In selecting the minimal polynomial, the property of field elements whereby several powers of generating elements have the same minimal polynomial is used. Each of the syndrome equation generated is a function of the errors in the received codeword. If *Si* is represented in terms of α, then αi are unknowns. A solution to these equations yields information that can be used to construct an error locator polynomial. The solution that yields the minimum number of errors in the received code word is considered (best case scenario).

The solution mostly uses elementary symmetric functions and newton’s identities. If the error locator polynomial has a degree lower than ‘*t*’, then the errors in code word can be corrected. Using the error locator polynomial, roots are found by trial and error substitution. The bit positions of the error locations correspond to the inverses of these roots. Creating a bit stream from this polynomial and adding errors to received code word, results in the error corrected code word as shown in Figure 7.



**Figure 7 Error Correction**

But it is possible that the syndrome might result in zero and a false conclusion that the message is correct can be obtained. But by performing additional steps this can be overcome. The additional steps aren’t covered here. The error correcting capability of BCH codes depends on the redundant bits added which is illustrated in Table 2. As the codeword length increases with increase in Message word length, the Error correcting capability is improved.

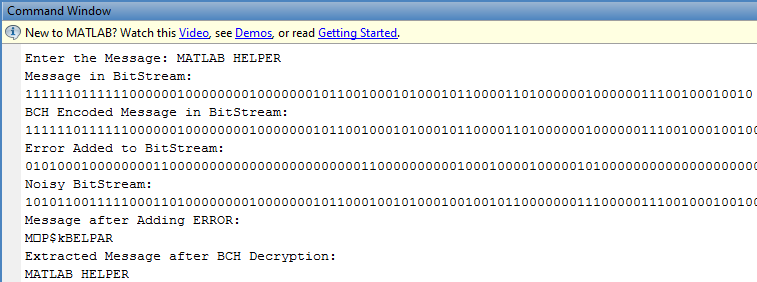
**Table 2 Error correcting capability of BCH**

|  |  |  |
| --- | --- | --- |
| **Code word**  **length (*n*)** | **Message word length (*k*)** | **Error correcting capability (*t*)** |
| 7 | 4 | 1 |
| 15 | 7 | 2 |
| 31 | 11 | 5 |
| 63 | 10 | 13 |
| 255 | 13 | 59 |
| 511 | 10 | 121 |

All these are ready to use functions that are available preinstalled in MATLAB. Functions include bchenc(), bchgenpoly(), bchdec(). All these functions use data (binary data). Hence the above mentioned BCH Process is to only understand the concept of BCH Encoding employed in MATLAB. Examples illustrating the concept as application are given below

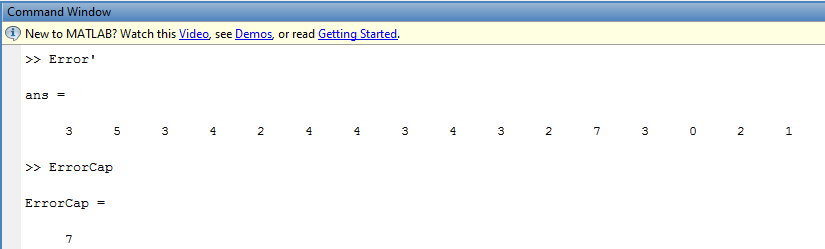
**Examples with MATLAB Outputs:**

**Example 1:**



**Figure 8 Encoding, Noise Addition and Recovery of Message**

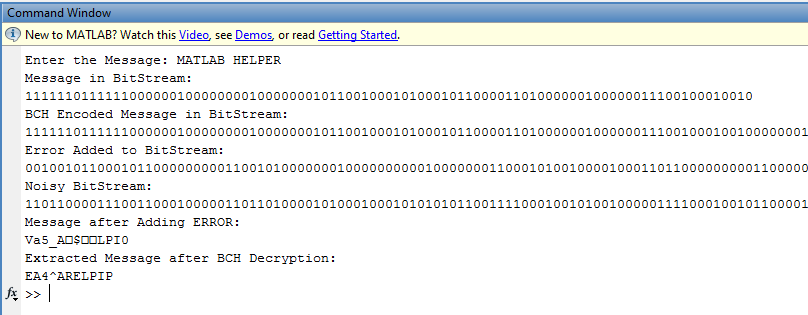
Here in Figure 8, The BCH Codes has been set using parameters with Code length of 32 and Message length of 6. This results in an error correction capability of 7-bit errors. Noise is added to the Encoded Bitstream intentionally. As the noise in binary data implies a change in the bit stream from ‘1’ to ‘0’ or vice-versa, *XOR* function is employed to implement this. The Noise affects the message and without any encoding, the noisy message is shown. But with BCH decoding, the message is fully recovered. The Figure 9, below gives the bit errors denoted by the positive values that had occurred and have been corrected in each code word by the algorithm.



**Figure 9 Bit Errors in each code word**

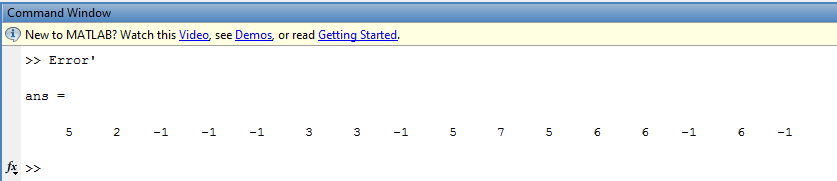
What happens when the error is too high? How does the algorithm perform?

**Example 2:**



**Figure 10 Output of Process when Noise is too high**

At times, the noise exceeds the error correcting capability of the BCH Code as in Figure 10. In such cases, the algorithm tries to correct as much error as possible and it tells the user how many errors are still uncorrected. The Figure 11, below shows values in negative. These negative values denote the number of uncorrectable bit errors present in a message whose location could not be found.

**Figure 11 Bit Errors when noise is too high**

Thus, BCH Error Correcting are efficient schemes that have every good potential for many applications. The key features of BCH codes is that there is a precise control over the number of symbol errors correctable by the code which makes the scheme dynamic enabling its use in highly varying noise environments. Another advantage of BCH codes is the ease with which they can be decoded.

**\*\*\*END\*\*\***