C++ Class Libraries for Error Control Codes

Jamshid Nazari
Purdue University School of Electrical Engineering

Okan K. Ersoy
Purdue University School of Electrical Engineering

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

http://docs.lib.purdue.edu/ecetr/195

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
C++ CLASS LIBRARIES FOR ERROR CONTROL CODES

JAMSHID NAZARI
OKAN K. ERSOY

TR-EE 94-26
AUGUST 1994

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285
C++ Class Libraries for Error Control Codes*

Jamshid Nazari†
Okan K. Ersoy
Purdue University
West Lafayette, IN 47907

Abstract

A tutorial on error control code is given and encoding and decoding using finite field transforms are outlined. Several C++ class libraries are then designed to implement some of the concepts in error control codes. A high level description of these libraries are given and several programs that uses these libraries are presented. RS and BCH codes can be generated by these libraries from primitive polynomials where the user specifies the code size and number of errors to be corrected. Using these libraries power programs in error control codes can be written in less than few pages.

†This research was sponsored by APPCOM Inc, West Lafayette, IN 47906. The programs and C++ libraries discussed in this paper can be obtained by an anonymous user from en.ecn.purdue.edu from pub directory. The name of files are C++ECC.README and C++ECC.tar.Z.

‡Received Ph.D. in Electrical Engineering, from Purdue University on May 1994.
1 Introduction

Error control codes are widely used in digital communication and storage systems. The purpose of error control coding is to add redundancy to transmitted or stored data. This provides the means for detecting and correcting errors that occur in a real system.

Section 2 gives a brief description of a digital communication system. Section 3 gives a brief description of algebra used in the design of encoders and decoders. In this section, the finite or Galois fields are explained and their properties examined.

Section 4 gives a brief description of several topics in error control coding [2, 9, 6]. The aim of this section is to familiarize ourself with the terminologies, which are used in later sections of this paper. Encoding and Decoding techniques, which use finite field transforms [1], are described in Section 5.

Section 6 describes C++ class libraries for error control codes. Using these classes we can generate RS and BCH codes where the user specifies code size and number of errors to be corrected. These classes are used to implementation several programs in error control codes. Section 7 gives the concluding remarks for this chapter.

2 Digital Communication System

A block diagram of a digital communication system is shown in Figure 1. As is shown in

![Block Diagram of a Digital Communication System](image)

Figure 1: Block Diagram of a Digital Communication System.

the figure, the digital source is producing streams of bits which need to be transmitted to the receiving end. The information bits are first grouped into $k$ bits. This is shown as the vector $u$ in the figure. The encoder takes these $k$ bits, adds several redundant bits and produces the vector $v$ of size $n$ bits. Here the vector $v$ is referred to as a code word. The redundant bits, which are added to each information vector $u$, are referred to as parity bits. The number of parity bits is $n = k$. The number of parity bits determines how many errors can be corrected. The higher the number of parity bits, the more error bits which can be corrected successfully. However the rate of actual information transmitted decreases because we have to pass more bits across the channel. Because of the 'noise', as data is passed through the channel, the received data vector $r$ will not be the same as the vector $u$. 


v. The decoder uses redundant information in r to detect errors and correct them. After stripping the parity bits the corrected data is passed to the receiving end.

3 Brief Introduction to Algebra

This section gives a brief description of algebra which provides the necessary tool to design encoders and decoders. We will define the mathematical concepts such as groups and fields. Our focus will be on fields which have finite number of elements. These fields are referred to as finite fields or Galois fields. We state several properties of Finite fields and show how Galois fields are constructed.

3.1 Definition of a Group

A set G on which a binary operation $\ast$ is defined is called a group if the following conditions are satisfied:

1. Closure property. If two elements are picked from a set and the binary operation $\ast$ is applied then the result will be in the set.

2. Associative property. Any three elements $a, b$ and $c$ in the set satisfy the condition

$$ (a \ast b) \ast c = a \ast (b \ast c) $$

3. Set has an identity element e. For any element a in the set, we have the condition

$$ a \ast e = e \ast a = a $$

4. Each element in the set has an inverse. i.e. $a$ and $a'$ are elements in the set and we have the condition

$$ a \ast a' = a' \ast a = e $$

A set G is a commutative group if G is a group and any two elements in G satisfies the condition

$$ a \ast b = b \ast a $$

For example set G={0,1,2} with modulo-3 arithmetic is a commutative group.

3.2 Definition of a Field

In an informal term, a set F is a field if we can do addition, multiplication, subtraction, and division. Formally a set F on which binary operations addition and multiplication are defined is called a field if the following conditions are satisfied:

1. F is a commutative group under addition. The additive identity element is referred to as the zero element and denoted by 0.

2. The set of non-zero elements of F is a commutative group under multiplication. The multiplicative identity element is referred to as the one and denoted by 1.
3. Multiplication is distributive over addition. For any elements $a$, $b$ and $c$ in the set $F$ we have the condition.

$$a(b + c) = ab + ac$$

From the definition, a field must contain at least two elements the additive identity $0$ and multiplicative identity $1$. The number of elements in the field is called the order of the field. A field which has a finite number of elements is called a finite field. Finite fields are also known as Galois fields. Finite field which has $q$ elements will be denoted by $GF(q)$. The additive inverse of a field element $a$ is denoted by $-a$. The multiplicative inverse of a field element $b$ is denoted by $b^{-1}$.

The set $F=\{0,1\}$ is a $GF(2)$ if the addition and the multiplication operations on field elements $0$ and $1$ are the Modulo-2 addition and Modulo-2 multiplication. Table 1 shows Modulo-2 addition and multiplication on the field elements. In $GF(2)$, $1 + 1 = 0$ implying that $1 = -1$. So subtraction is the same as addition.

The set $F=\{0,1,2,3,4\}$ is a $GF(5)$ if the addition and the multiplication operations on field elements are Modulo-5 addition and Modulo-5 multiplication. Table 2 shows Modulo-5 addition and multiplication on the field elements. From this table $3 + 2 = 0$. Therefore additive inverse of $3$ denoted by $-3$ is $2$. Also from the table $3 \times 2 = 1$. So multiplicative inverse of $3$ denoted by $3^{-1}$ is $2$. Additive and Multiplicative inverses are used to do subtraction and division. For example

$$2 - 3 = 2 + (-3) = 2 + 2 = 4$$

$$2 \div 3 = 2 \times 3^{-1} = 2 \times 2 = 4$$

### 3.3 Properties of Galois Field

In this section we will state several properties of Galois fields and proof some of these properties. Let $\lambda$ be a finite field of $q$ elements. The smallest positive integer $\lambda$ such that $\sum_{i=1}^{\lambda} 1 = 0$ is called the characteristic of the field $GF(q)$. Since the elements in the field are closed under addition

$$1, 1 + 1, 1 + 1 + 1, \ldots, 1 + 1 + 1 + \cdots$$

3
must be field elements and at some time one of the sums must be zero. Similarly the set of non-zero elements are closed under multiplication and the sequence
\[ a, an, aaa, \ldots, aaaa \cdots \]
must repeat at one point. The smallest positive integer \( n \) such that \( a^n = 1 \) is called the order of the field element \( a \). If order of a field element is \( q - 1 \) then this element is \( a \) primitive element of field \( GF(q) \). From the definition of order of a field element, the products of a primitive element will produce all non-zero field elements.

**Theorem 3.1** For any nonzero field element \( a \) in \( GF(q) \), we have \( a^{q-1} = 1 \).

**Proof:** Since field elements are closed under multiplication, we have
\[
\prod_{i=1}^{q-1} a_i b_i = \prod_{i=1}^{q-1} b_i
\]
where each \( a_i \) is a non-zero element of \( GF(q) \). But this is equal to
\[
a^{q-1} \prod_{i=1}^{q-1} b_i = \prod_{i=1}^{q-1} b_i
\]
so we have \( a^{q-1} = 1 \). \( \square \)

**Theorem 3.2** In \( GF(q) \) order of any nonzero field element divides \( q - 1 \).

**Proof:** Let us assume the order of a nonzero field element is \( n \) and it does not divide \( q - 1 \), i.e.
\[ q - 1 = nm + r \quad 0 < r < n \]
We have
\[
a^{q-1} = a^{nm} a^r = a^{nm} a^r = a^r
\]
\( a^{q-1} = 1 \) and \( a^n = 1 \), therefore \( r \) must be zero, a contradiction. So order of any nonzero field element divides \( q - 1 \). \( \square \)

**Theorem 3.3** The characteristic \( \lambda \) of Galois field is a prime number.

**Proof:** Let us assume it is not, i.e. \( \lambda = rs \) and \( r \) and \( s \) smaller than \( \lambda \). Since \( \sum_{i=1}^{r} 1 \) and \( \sum_{j=1}^{s} 1 \) are field elements, their product is a field element. Using distributive property we have
\[
\sum_{i=1}^{r} 1 \sum_{j=1}^{s} 1 = \sum_{k=1}^{rs} 1 = 0
\]
Therefore one of the sums \( \sum_{i=1}^{r} 1 \) or \( \sum_{j=1}^{s} 1 \) must be zero. But this is not possible because we chose \( \lambda \) to be the smallest positive number such that \( \sum_{i=1}^{\lambda} 1 = 0 \). So the characteristic of a Galois field is a prime number. \( \square \)
3.4 Polynomials Over GF(q)

A polynomial over GF(q) is a polynomial which has coefficients from field elements. For example in GF(2) the polynomial

\[ f(X) = a_0 + a_1 X + a_2 X^2 + \cdots + a_{n-1} X^{n-1} \]

will have coefficients which are 0 or 1. If \( a_{n-1} = 1 \) then the degree of the polynomial is \( n - 1 \). A field element \( b \) is a root of the polynomial \( f(X) \), if \( f(b) = 0 \). In this case \( (X - b) \) is said to divide \( f(X) \). A polynomial of degree \( m \) is said to be irreducible over GF(q) if it is not divisible by any polynomials over GF(q) of degree less than \( m \) but greater than 0.

We will be dealing with a special type of irreducible polynomial known as primitive polynomial in GF(2). Any irreducible polynomial over GF(2) divides the polynomial \( X^n - 1 \) where \( n = 2^m - 1 \). Any irreducible polynomial \( p(X) \) of degree \( m \) is said to be a primitive polynomial if the smallest positive integer \( n \) for which \( p(X) \) divides \( X^n - 1 \) is \( n = 2^m - 1 \).

3.5 Construction of GF(2^m)

For values of \( m > 1 \), Galois field GF(2^m) is an extension of Galois field GF(2). In this section we will show how we can construct GF(2^m) from GF(2). This is done by using a primitive polynomial \( p(x) \) of degree \( m \) and elements 0, 1 and a symbol \( a \) with the condition that \( p(a) = 0 \).

**Theorem 3.4** The set \( F = \{0, 1, a, a^2, \ldots, a^{2^{m-2}}\} \) forms a field.

**Proof:** The polynomial \( p(x) \) is a primitive polynomial therefore from the definition it divides \( X^{2^m - 1} + 1 \) i.e.,

\[
\begin{align*}
X^{2^m - 1} + 1 &= p(X)q(X) \\
a^{2^m - 1} + 1 &= p(a)q(a) \\
a^{2^m - 2} + 1 &= 0 \\
a^{2^m - 1} &= 1
\end{align*}
\]

\( a^{2^m - 2} \) is the last element in the field before it repeats.

Next we want to show that the elements of the set \( F \) are unique. To show this we use the primitive polynomial \( p(x) \) again. So

\[
\begin{align*}
X^i &= p(X)q_i(X) + r_i(X) \\
a^i &= p(a)q_i(a) + r_i(a) \\
a^i &= r_i(a)
\end{align*}
\]

Since \( X^i \) and \( p(X) \) are relatively prime then \( r_i(X) \) is not zero and its degree is \( m - 1 \) or less. Now, we will show that \( r_i(X) \) is not equal to \( r_j(X) \). Assume they are same and we draw a contradiction. Let us assume \( j > i \)

\[
\begin{align*}
X^i &= p(X)q_i(X) + r_i(X) \\
X^j &= p(X)q_j(X) + r_i(X) \\
X^i + X^j &= p(X)[q_i(X) + q_j(X)] \\
X^i(1 + X^{j-i}) &= p(X)[q_i(X) + q_j(X)]
\end{align*}
\]
Since \( p(X) \) and \( X^i \) are relatively prime then \( p(X) \) must divide \( 1 + X^{j-i} \). This is a contradiction, because \( p(X) \) is primitive polynomial and divides \( X^n - 1 \) where \( n = 2^m - 1 \). Here \( j - i < n \). Each \( \alpha^i \) is shown to be represented by a polynomial of degree \( m - 1 \) or less and since these polynomials are unique the \( \alpha^i \) elements are unique.

Addition between two elements of the set \( F \) is closed. Two polynomials when added will produce another polynomial of degree \( m - 1 \) or less. Therefore addition will produce an element which it is in the set. The multiplication between two elements is also closed.

\[
\alpha^i \cdot \alpha^j = \alpha^{i+j \mod n} \quad n = 2^m - 1
\]

Additive inverse of each element is itself. And multiplicative inverse of a nonzero element \( \alpha^i \) is \( \alpha^{2^m-1-i} \). So we have shown that \( GF(2^m) \) is a Galois field. In most cases we will be using this extended field.

### 3.6 An Example: Construction of \( GF(2^3) \)

In this section we want to show how each element of \( GF(2^3) \) is represented by a polynomial of degree 2 or less. We will use the primitive polynomial \( p(X) = X^3 + X + 1 \) with the condition \( p(\alpha) = 0 \). Since \( p(\alpha) = 0 \) we have \( \alpha^3 = \alpha + 1 \). \( \alpha^4, \alpha^5 \) and \( \alpha^6 \) are represented as follows:

\[
\begin{align*}
\alpha^4 &= \alpha \cdot \alpha^3 = \alpha(\alpha + 1) = \alpha^2 + \alpha \\
\alpha^5 &= \alpha \cdot \alpha^4 = \alpha(\alpha^2 + \alpha) = \alpha^3 + \alpha^2 = \alpha^2 + (\alpha + 1) \\
\alpha^6 &= \alpha \cdot \alpha^5 = \alpha(\alpha^2 + \alpha + 1) = \alpha^3 + \alpha^2 + \alpha = (\alpha + 1) + \alpha^2 + \alpha = \alpha^2 + 1
\end{align*}
\]

Each element of the field was shown to be represented by a polynomial of degree 2 or less. Table 3 shows each element is represented by a polynomial and equivalent 3-tuple representation.

<table>
<thead>
<tr>
<th>Element</th>
<th>Polynomial Representation</th>
<th>3-tuple Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>(000)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(001)</td>
</tr>
<tr>
<td>( \alpha^2 )</td>
<td>( \alpha )</td>
<td>(010)</td>
</tr>
<tr>
<td>( \alpha^2 )</td>
<td>( \alpha^2 )</td>
<td>(100)</td>
</tr>
<tr>
<td>( \alpha^3 )</td>
<td>( \alpha + 1 )</td>
<td>(011)</td>
</tr>
<tr>
<td>( \alpha^4 )</td>
<td>( \alpha^2 + \alpha )</td>
<td>(110)</td>
</tr>
<tr>
<td>( \alpha^5 )</td>
<td>( \alpha^2 + \alpha + 1 )</td>
<td>(111)</td>
</tr>
<tr>
<td>( \alpha^6 )</td>
<td>( \alpha^2 + 1 )</td>
<td>(101)</td>
</tr>
</tbody>
</table>

### 3.7 Minimal Polynomial of a Field element in \( GF(2^m) \)

Let us take any nonzero element \( \beta \) from \( GF(2^m) \) and \( q = 2^m \). From Theorem 3.1 we have that any nonzero element to the power of \( q - 1 \) is 1, i.e.

\[
\begin{align*}
\beta^{2^m-1} &= 1 \\
\beta^{2^m-1} + 1 &= 0
\end{align*}
\]

6
So \( \beta \) is a root of the polynomial \( X^{2^m-1} + 1 \). The smallest degree polynomial which has \( \beta \) as one of its roots is called the \textit{minimal polynomial} of the field element \( \beta \).

\textbf{Theorem 3.5} Minimal polynomial of a field element \( \beta \) is irreducible.

\textbf{Proof:} Let \( \phi(X) \) to be the minimal polynomial of \( \beta \). Assume it is not irreducible. So choose \( \phi(X) = a(X)b(X) \), with degree of \( a(X) \) and \( b(X) \) to be less than the degree of \( \phi(X) \). Since \( \phi(\beta) = 0 \), \( a(\beta) \) or \( b(\beta) \) must be zero. This is a contradiction because the minimal polynomial is chosen to be the smallest degree polynomial which has \( \beta \) as its roots. And here we find out that there is a lower degree polynomial \( a(X) \) or \( b(X) \) which has \( \beta \) as its roots. So the minimal polynomial of a field element \( \beta \) is irreducible.

The \textit{conjugates} of a field element \( \beta \) is the elements

\[ \beta^1, \beta^2, \ldots, \beta^{q^m-1} \]

where \( \beta^{q^m} \equiv \beta \). It can be shown that if a field element \( \beta \) is a root of a polynomial then the conjugates of \( \beta \) are also the roots of the same polynomial. In fact a minimal polynomial of field element \( \beta \) can be constructed using the following formula:

\[ \phi(X) = \prod_{i=0}^{q^m-1} (X + \beta^i) \]

\section{3.8 Roots of Polynomials in GF(q)}

Polynomials over real numbers sometimes have roots which belong to an \textit{extended} field of complex numbers. Similarly the polynomials over \( \text{GF}(q) \) can have roots which belong to the extended field \( \text{GF}(q^m) \). For example, we used the primitive polynomial \( p(X) = X^3 + X + 1 \) to construct the extended field \( \text{GF}(2^3) \). This polynomial has degree 3, therefore it has three roots. None of its roots belong to \( \text{GF}(2) \) and they all belong to the extended field \( \text{GF}(2^3) \). To determine its roots, each field element of \( \text{GF}(2^3) \) is substituted into \( p(X) \) and those which result in \( p(X) = 0 \) are the roots. The roots of this polynomial are \( \alpha, \alpha^2 \) and \( \alpha^4 \). For example:

\[ p(\alpha^4) = \alpha^{12} + \alpha^4 + 1 \]
\[ = \alpha^5 + \alpha^4 + 1 \]
\[ = (\alpha^2 + \alpha + 1) + (\alpha^2 + \alpha) + 1 \]
\[ = 0 \]

We used the reduction that \( d = 1 \). Note that the roots are conjugates.

\section{4 Brief Topics in Error Control Coding}

In this section we will give a brief description of several basic concepts in coding. This section contains the following:

- Definition of a block code and a linear block code.
- Encoding via a generator matrix.
• Parity check matrix
• Syndrome of a received vector.
• Undetectable error patterns.
• Definition of distance and weight in a block code.
• Syndrome decoding.
• Guarantees on error detection and correction.
• Definition of a perfect code.
• Binary symmetric channel (BSC).
• Nearest-neighbor decoding rule.
• Probability of undetected errors.
• Probability of erroneous decoding.
• A brief description of cyclic codes.
• A brief description of BCH codes over GF(q^m).
• A brief description of RS codes over GF(q).

4.1 Definition of a Block Code and a Linear Block Code
During encoding a vector of k q-ary bits is converted into a vector of n q-ary bits. These n bit vectors are referred to as code words. The number of code words is q^k. The set of q^k code words is called a block code. A block code is called a linear (n, k) block code if it can be generated from a k-dimensional subspace of an n-tuple vector space over GF(q).

4.2 Encoding via a Generator Matrix
We construct a generator matrix G by picking k n-tuple independent row vectors g_1, g_2, ..., g_k and arranging them as rows of the matrix G. Here matrix G is made of elements form the field GF(q).

Encoding can be accomplished by the following equation:

\[ v = u \cdot G \]

Note that the vector v is just a linear combination of rows of the matrix G. Vector v contains the k information bits and n - k parity bits. If these two segments are not mixed then the generator matrix G produces a systematic block code. The k information bits can easily be extracted from a code word generated from a systematic block code. Therefore systematic block codes are usually desired. To obtain a systematic block code, the generator matrix should contain a k x k identity matrix.
4.3 Parity Check Matrix

We can come up with another matrix $H$ which is made of $n-k$ independent row vectors such that each of the row vectors is orthogonal to the rows of the generator matrix $G$, i.e.,

$$G \cdot H^T = 0$$

The $H$ matrix constructed this way is referred to as the parity check matrix. Since a code word $v$ is made of a linear combination of the rows of the $G$ matrix then $v \cdot H^T = 0$.

4.4 Syndrome of a Received Vector

Due to "noise" a received vector $r$ might not be the same as the transmitted vector $v$. So we can assume

$$r = v + e$$

where $e$ is an $n$-tuple error vector. The calculation $s = r \cdot H^T$ is referred to as the syndrome of the received vector $r$. So error detection is accomplished by calculating the syndrome of the received vector $r$ and error has occurred if the syndrome is not 0. Also note that the syndrome of a vector $r$ is the same as the syndrome of the error vector $e$.

4.5 Undetectable Error Patterns

If the error vector is such that it changes a code word $v$ into another valid code word, then syndrome calculation will still be zero. So these types of errors are called undetectable errors patterns. The number of undetectable error patterns is $q^k - 1$ and the number of detectable errors is $q^n - q^k$.

4.6 Definition of Distance and Weight in a Block Code

The Hamming distance, $d(v,w)$, between two code words $v$ and $w$ is the number of places these code words differ. If we calculate the Hamming distance between any two code words and find the smallest number of these distances we obtain the minimum distance, $d_{\text{min}}$, of the block code. The Hamming weight (or simply weight) of a code word $v$, denoted by $w(v)$, is the number of nonzero elements in the code word $v$.

**Theorem 4.1** The minimum distance, $d_{\text{min}}$, of a linear block code is equal to minimum weight, $w_{\text{min}}$, of a nonzero code word.

Proof:

$$d_{\text{min}} = \min d(v,w) \quad v \neq w$$

$$= \min w(v+w)$$

$$= \min w(x) \quad x \neq 0$$

$$= w_{\text{min}}$$
4.7 Syndrome Decoding

This type of decoding first involves partitioning possible $q^n$ code words into $q^k$ disjoint sets. The partitioning also known as standard array construction is as follows. Arrange all the $q^k$ code words into a row with the zero code word as the first element. Pick an n-tuple which is not used before and place it as the first element in the second row, i.e. this element is now below the zero code word. Note that, this element is an error pattern and pick those error patterns with smallest error weights first. For the remaining columns of this row, add each code word with the error pattern and put the resulting n-tuple under the code word. Pick another n-tuple which is not seen before as the leader of the third row and fill the row as before. Continue this process until we run out of all the n-tuples. Each row is called a coset, and the first element of the coset is called the coset leader. It can be shown that all the elements in each coset has the same syndrome value. Each of the $q^k$ columns contains $q^{n-k}$ elements, and the columns are disjoint.

We can create a table with the coset leaders in the first column, and their syndrome values in the second column. The table-lookup decoding or syndrome-decoding is as follows:

1. Compute the syndrome of the received vector
2. Find the error pattern which corresponds to this syndrome in the table.
3. Add this error pattern to the received vector to obtain the code word which was transmitted.
4. Strip the parity bits.

If a decoder detects that there are more than $t$ errors and refuses to decode the received word then we have an incomplete decoder. If a decoder goes ahead and decodes any received word, regardless of the number of errors that the code word might have, it is called a complete decoder.

4.8 Guarantees on Error Detection and Correction

A block code with minimum distance $d_{\text{min}}$ guarantees can detect all error patterns of $d_{\text{min}} - 1$ or fewer errors. However it is capable of detecting some error patterns with $d_{\text{min}}$ or more errors. A block code which can correct $t$ errors is called a $t$-error correcting code. The number of errors which is guaranteed to be corrected by a block code is given by

$$t = \left\lfloor \frac{d_{\text{min}} - 1}{2} \right\rfloor$$

However the block code may correct $t + 1$ or more error patterns.

4.9 Definition of a Perfect Code

During the construction of the standard array we were picking the error patterns: with the smallest weights first, i.e., the error patterns which have weights 1 are selected first and then the error patterns which have weights 2 are selected next and so on. Finally error patterns which have weights $t$ are selected as the coset leaders. After this step there may not be any more n-tuples left to create the additional cosets in the standard array. If this happens then the block is referred to as a $t$-error correcting perfect code. Perfect codes exist and they are rare.
4.10 Binary Symmetric Channel (BSC)

Binary symmetric channel is an example of a discrete memoryless channel. A bit 0 sent through a channel might be received as a bit 0 or hit 1. The probability of sending 0 and receiving 1 is denoted by \( \text{Prob}(1|0) \). Similarly a bit 1 sent through a channel may be received as a bit 1 or bit 0. The probability of sending 1 and receiving 0 is denoted by \( \text{Prob}(0|1) \). If the transitional probabilities \( \text{Prob}(0|1) \) and \( \text{Prob}(1|0) \) are equal then the channel is referred to as a binary symmetric channel and the transitional probability value is referred to as \( p \). The probabilities of BSC is as follows:

\[
\begin{align*}
\text{Prob}(0|1) &= \text{Prob}(1|0) = p \\
\text{Prob}(1|1) &= \text{Prob}(0|0) = 1 - p
\end{align*}
\]

A bit 0 or 1 sent through a channel is erased if the receiver cannot make the decision whether it is 0 or 1. In this case The receiver produces erasure bits denoted by '?' A BSC is a binary symmetric erasure channel (BSEC) if the receiver produces erasures. The probabilities of BSEC is as follows:

\[
\begin{align*}
\text{Prob}(0|1) &= \text{Prob}(1|0) = p \\
\text{Prob}(?|1) &= \text{Prob}(?|0) = q \\
\text{Prob}(1|1) &= \text{Prob}(0|0) = 1 - p - q
\end{align*}
\]

4.11 Nearest-Neighbor Decoding Rule

We first give a geometric representation of a block code and then define the nearest-neighbor decoding rule. Each code word is put in the center of a sphere \( S \) of radius \( t \). We will have \( q^k \) of these spheres. Also let each sphere \( S \) contain the \( n \)-tuples which have distances \( t \) or less from the code word which is in the center of the sphere. These \( n \)-tuples will not be code words because the minimum distance between any two code words is assumed to be \( 2t + 1 \). If all the possible \( n \)-tuples don't lie outside any of the spheres then we have a perfect code. This definition is similar to the definition given before.

A received vector may fall inside any of the \( q^k \) spheres. In which case the code word in the center of the sphere is the transmitted vector. If the decoder calculates all the distances between the received vector and each of the code words and chooses the code word which results in the smallest distance, it is said to use the nearest-neighbor decoding rule. This is also known as the maximum-likelihood decoding.

**Theorem 4.2** Probability of decoding error is minimized when using the nearest-neighbor decoding rule.

**Proof:** Let us assume the received vector \( r \) is a code word \( v \), but the decoder assigns it to another code word \( w \), i.e., decoding error happens. We have

\[
\begin{align*}
\text{Prob}(E|r) &= \text{Prob}(w \neq v|r) \\
\text{Prob}(E) &= \sum \text{Prob}(E|r) \cdot \text{Prob}(r)
\end{align*}
\]
\( \text{Prob}(\tau) \) is independent of the decoding rule so to minimize \( \text{Prob}(E) \) is to minimize \( \text{Prob}(E|\tau) \).
This can be accomplished by maximizing the probability of correct decoding \( \text{Prob}(w = v|r) \) (or just \( \text{Prob}(v|r) \)). Using Bayes rule, we have

\[
\text{Prob}(v|r) = \frac{\text{Prob}(r|v) \cdot \text{Prob}(v)}{\text{Prob}(r)}
\]

Assuming all the code words are equally likely we need to maximize \( \text{Prob}(r|v) \). For discrete memoryless channel

\[
\text{Prob}(r|v) = \prod_{i=1}^{n} \text{Prob}(r_i|v_i)
\]

If we take log of both sides we have

\[
\log \text{Prob}(r|v) = \sum_{i=1}^{n} \log \text{Prob}(r_i|v_i)
\]

For binary symmetric channel,

\[
\text{Prob}(r_i \neq v_i|v_i) = p
\]

and

\[
\text{Prob}(r_i = v_i|v_i) = 1 - p
\]

therefore we have

\[
\log \text{Prob}(r|v) = d(r,v) \log p + (n - d(r,v)) \log (1 - p)
\]

\[
\log \text{Prob}(r|v) = d(r,v) \log \frac{p}{p - 1} + n \log (1 - p)
\]

The second term in the right hand side is independent of the decoding rule. So if \( p < 1/2 \) then \( \log(p/(p - 1)) \) is negative, therefore, to maximize \( \text{Prob}(r|v) \) is to minimize \( d(r,v) \). So we have shown that if we use the nearest-neighbor decoding rule, i.e., calculating the distance between the received word and all other code words and picking the code word which has the smallest distance, then the probability of decoding error is minimized. \( \Box \)

### 4.12 Probability of Undetected Errors

Let \( A_i \) to represent the number of code words which has weight equal to \( i \). Then the numbers \( A_0, A_1, \ldots, A_n \) are called the weight distribution of a block code. Errors are undetected when error patterns are identical to nonzero code words. For a binary symmetric channel with the transition probability \( p \) the probability of undetected error is given by

\[
P_u(E) = \sum_{i=1}^{n} A_i p^i (1 - p)^{n-i}
\]

### 4.13 Probability of Erroneous Decoding

The decoder is guaranteed to correct \( t \) or less errors but is capable of correcting more errors. For a binary symmetric channel with the transition probability \( p \), the probability that the decoder commits an erroneous decoding is upper bounded by

\[
P(E) \leq \sum_{i=t+1}^{n} \left( \begin{array}{c} n \\ i \\ \end{array} \right) p^i (1 - p)^{n-i}
\]

---

12
4.14 A Brief Description of Cyclic Codes

One cyclic shift of an n-tuple

\[ \mathbf{v} = (v_0, v_1, v_2, \ldots, v_{n-2}, v_{n-1}) \]

produces the n-tuple

\[ \mathbf{v}^i = (v_{n-1}, v_0, v_1, \ldots, v_{n-3}, v_{n-2}) \]

The ith cyclic shift of the vector \( \mathbf{v} \) produces the vector \( \mathbf{v}' \). In a linear block code, if every cyclic shift of a code vector produces another code vector then the block code is called a cyclic code.

The components of a vector \( \mathbf{v} \) may be used as coefficients of a polynomial \( v(X) \) of degree \( n - 1 \), i.e.,

\[ v(X) = v_0 + v_1X + v_2X^2 + \cdots + v_{n-1}X^{n-1} \]

Vector \( v(X) \) is referred to as a code polynomial. Similarly vector \( \mathbf{u} \) may be represented by a message polynomial \( u(X) \) of degree \( k - 1 \). It can be shown that

\[ \mathbf{v}^i = X^i \cdot v(X) \mod (X^n - 1) \]

and also for any \((n,k)\) cyclic code there exist a polynomial \( g(X) \) of degree \( n - k \),

\[ g(X) = 1 + g_1X + g_2X^2 + \cdots + g_{n-k-1}X^{n-k-1} + X^{n-k} \]

such that every code polynomial is a multiple of \( g(X) \). The generator polynomial \( g(X) \) can also be shown to be a factor of the polynomial \( X^n - 1 \).

The encoding of cyclic codes is accomplished by multiplying each message polynomial with the generator polynomial \( g(X) \). However this does not produce systematic code words. The systematic code words can be generated by the following equation,

\[ X^{n-k} \cdot u(X) + (X^{n-k} \cdot u(X) \mod g(X)) \]

Syndrome polynomial is the remainder of the division of the received code polynomial by \( g(X) \). If the syndrome polynomial is not zero then error must have occurred. Error correction can be done by using Meggitt decoder or variations of Meggitt decoding called error-trapping decoding.

4.15 A Brief Description of BCH Codes Over GF\( (q^n) \)

The Bose, Chaudhuri, and Hocquenghem (BCH) codes is an important class of random error-correcting cyclic codes. The generator polynomial \( g(X) \) of a BCH code contains \( 2t \) consecutive roots of \( \beta \), i.e., \( g(\beta^i) = 0 \) for \( 1 \leq i \leq 2t \). In addition \( g(X) \) must also be the smallest degree polynomial. To achieve these we find the least common multiple (LCM) of minimal polynomials of roots of \( g(X) \), i.e,

\[ g(X) = \text{LCM}\{\phi_1(X), \phi_2(X), \ldots, \phi_{2t}(X)\} \]

Where \( \phi_i(X) \) is the minimal polynomial of \( \beta^i \). Since the degree of each polynomial is \( m \) or less then \( n - k \leq mt \). We should mention that BCH codes are defined for \( m \geq 3 \).
If the field element $\beta$ is a primitive element then we obtain a \textit{primitive $t$-error-correcting} BCH code and each code word will have $n = q^m - 1$ bits. If the field element $\beta$ is not the primitive element then we obtain a \textit{non-primitive $t$-error-correcting} BCH code where each code word has $n \neq q^m - 1$ bits.

Roots of each code polynomial obtained from the generator polynomial are $\beta, \ldots, \beta^{2t}$ and their conjugates. Therefore the the minimum distance $d_{\text{min}}$ of the code is greater than or equal $2t + 1$.

Since the vector $g(X)$ is a factor of any valid code word. The code word evaluated at the roots of $g(X)$ should give value of 0. The syndrome $S_i$ of a received vector $r(X)$ is obtained by evaluating $r(\beta^i)$. If syndrome values are zero no errors have occurred. Berlekamp-Massey algorithm maybe used to find \textit{error location polynomial}. The roots of the error location polynomial is used to give the location of the errors. For the binary case there is no need to find the error values. However, for non-binary case Forney algorithm maybe used to find the error values.

4.16 A Brief Description of RS codes over $\text{GF}(q)$

Reed-Solomon (RS) codes are special type of BCH codes which have code polynomials over $\text{GF}(q)$. Where $q$ is a prime number, or power of a prime. Usually we select $q = 2^m$.

The minimal polynomial of field element is $(X - \beta)$. To get a $t$-error correcting q-ary code, we select $\beta, \ldots, \beta^{2t}$ as the roots of the generator matrix. The generator matrix is
\[
g(X) = (X - \beta)(X - \beta^2) \cdots (X - \beta^{2t})
\]

The degree of $g(X)$ is $n - k = 2t$. The size of each code word is the order of the element $\beta$. If $\beta$ is a primitive element then we obtain the \textit{primitive RS code} and each code word has length $n = q - 1$.

5 Encoding and Decoding using Finite Field Transform

This section explains the following topics:

- Definition of the finite field transform is given.
- Definition of the inverse finite field transform is given.
- Conjugacy constraints are explained?
- Transform encoding method is explained.
  - Two examples in transform encoding are given.
- Transform decoding is explained.
  - Recursive extension of the error spectrum is explained.
- A direct method is explained to find the error locator polynomial.
- Berlekamp-Massey algorithm is explained to find the error locator polynomial.
- An Example of transform decoding is given.
5.1 Definition of the Finite Field Transform

The finite field transform of the vector \( v \) over the field elements \( GF(q) \) is given by the equation

\[
V_j = \sum_{i=0}^{n-1} \beta^{ij} \cdot v_i \quad j = 0, \ldots, n - 1
\]

Where element \( \beta \) is an element of \( GF(q^m) \) and has order \( n \). From Theorem 3.2 \( \alpha \) divides \( q^m - 1 \). Therefore Finite field transform of any size vector is not possible. Indefinite \( i \) and \( j \) are called time and frequency respectively. The vectors \( v \) and \( V \) are called time-domain function and frequency-domain function. The vectors \( v \) and \( V \) are also known as signal and spectrum respectively.

5.2 Definition of the Inverse Finite Field Transform

Inverse finite field transform of the vector \( V \) is defined as follows:

\[
v_i = \frac{1}{n \mod p} \sum_{j=0}^{n-1} \beta^{-ij} \cdot V_j \quad i = 0, \ldots, n - 1
\]

Where \( p \) is the characteristic of the field \( GF(q) \).

If the code polynomial over \( v \) is \( v(X) \) and the code polynomial over \( V \) is \( V(X) \) then we have the following theorem.

Theorem 5.1 The polynomial \( v(X) \) has a zero at \( \beta^i \) if and only if the \( j \)th frequency component \( V_j \) equals zero.

Proof: We have \( v(\beta^i) = 0 \)

\[
v(X) = v_0 + v_1X + v_2X^2 + \cdots + v_{n-1}X^{n-1}
\]

\[
= \sum_{i=0}^{n-1} v_i \cdot X^i
\]

\[
v(\beta^i) = \sum_{i=0}^{n-1} v_i \cdot \beta^{ij}
\]

\[
= V_j
\]

Therefore, \( V_j = 0 \).

Similarly it can be shown that the polynomial \( V(X) \) has a zero at \( \alpha^{-i} \) if and only if the \( i \)th time component \( v_i \) equals zero.

5.3 Conjugacy Constraints

Equation 3 is used to convert an \( n \)-tuple vector \( v \) from \( GF(q) \) into an \( n \)-tuple transform vector \( V \) in \( GF(q^m) \), if we take an arbitrary spectrum vector \( V \) and take its inverse finite field transform using Equation 4 we may get a signal vector \( v \) which is not in \( GF(q) \). This problem can be avoided if the following equations known as conjugacy constraints are satisfied:

\[
V_j^q = V_{j_0 \mod n} \quad j = 0, \ldots, n - 1
\]
The components of the vector $V$ are $V_0, V_1, \ldots, V_{n-1}$. To meet the conjugacy constraint, a value selected for any frequency components fixes the values of other frequency components. For example for $q = 2$, $n = 15$, any valid value for $V_3$ gives

$$
\begin{align*}
V_3^2 &= V_6 \\
V_6^2 &= V_{12} \\
V_{12}^2 &= V_9 \\
V_9^2 &= V_3
\end{align*}
$$

The frequency components $V_6, V_9$ and $V_{12}$ are not arbitrary and can be obtained in terms of $V_3$.

To meet the conjugacy constraints, we may divide the indices $0, \ldots, n - 1$ into several classes where the frequency components in each class is related to each other. These classes are referred to as the conjugacy classes. The conjugacy class for index $j$ denoted by $A_j$ is

$$
A_j = \{j, jq, jq^2, \ldots, jq^{m_j - 1}\}
$$

where $m_j$ is the smallest positive integer which satisfies the equation

$$
j \cdot q^{m_j} \mod n = j
$$

In the example above $A_3 = \{3, 6, 9, 12\}$.

So far we have divided indices $0, \ldots, n - 1$ into several conjugacy classes. The conjugacy classes which contain any index in the range $1, \ldots, 2t$ will have $0$ in the spectrum vector $V$ in all of their indices. Now let us select a conjugacy class $A$ with $p$ elements, such that it does not have any indices in the range $1, \ldots, 2t$. Let us refer to these classes as free classes. Let us pick one of the indices in set $A$, say index $j$, as a free element, where we want to store a $\text{GF}(2^m)$ field element. The values stored in the other indices are based on $V_j$. Now, we need to determine which field elements of $\text{GF}(2^m)$ may be stored in $V_j$. It can be shown that the only valid field elements that can be stored in $V_j$ are zero element and those field elements that have order $2^p - 1$. Therefore, the bit content of $V_j$ is $p$ bits. The total bit content, $k$, is the summation of these $p$ values in the free sets.

### 5.4 Transform Encoding

We are interested in block codes which will have multiple-error correction capabilities. The BCH codes are linear cyclic codes which meet this criteria. In the encoding of the BCH codes, we need to find a generator polynomial $g(X)$. This polynomial is obtained by finding the least common multiple of minimal polynomials of $2t$ consecutive field elements. So the roots of $g(X)$ are $\beta^j$ for $j = 1, \ldots, 2t$ and their conjugates. The generator polynomial $g(X)$ multiplied with a message polynomial produces a code word.

The multiplication of two polynomials $g(X)$ and $u(X)$ in the time domain is equivalent to the cyclic convolution of their corresponding coefficients. The convolution in the time domain is equivalent to the inverse finite field transform of the product of finite field transforms of $g(X)$ and $u(X)$. From Theorem 5.1 we also have that if a polynomial has a root at $\beta^j$ then its spectrum component $V_j$ is $0$, so we define the transform encoding method as follows:
1. Set frequency components \( V_j = 0 \) for \( j = 1, \ldots, 2t \)

2. To meet conjugacy constraints, identify 'free' and "dependent" spectral components.

3. Use information symbols in \( GF(q^m) \) to specify the values for the free spectral component.

4. Use conjugacy constraint equations to calculate the values of other dependent spectral components.

5. Take the inverse finite field transform of the resulting frequency function to obtain an encoded time-domain code word.

In some codes, such as Reed-Solomon codes, the time and spectral domains are the same. In these codes, taking the inverse transform of any spectrum will result in a valid time-domain code word. Therefore, the conjugacy constraints on these codes are already met.

The above transform encoding steps are clarified further by giving several examples which are given next.

### 5.5 Examples of Transform Encoding

Here, we will give two examples in transform encoding. Both examples use the elements from \( GF(8) \). As a reminder, the field elements of \( GF(8) \) are

\[
\{0, 1, \alpha, \alpha^2, \ldots, \alpha^6\}
\]

And the polynomial representation of each field element is given in Table 3.

**Example 1**: In this example, the time and spectrum domains are both in \( GF(8) \). We are interested in obtaining a code which can correct up to 2 errors, so \( t = 2 \). We will use the primitive field element \( \alpha \) which has order 7, so \( n = 7 \).

The steps of the transform encoding method, which was given above, are explained for this example which are as follows:

1. Set frequency components \( V_1 = V_2 = V_3 = V_4 = 0 \)

2. Conjugacy constraints are already met, because the time and spectrum domains are the same. The spectral components \( V_0, V_5 \) and \( V_6 \) are "free" and there are no "dependent" spectral components.

3. Pick any field element from \( GF(8) \) for each of the three free variables.

4. We don't do anything in this step. There are no dependent spectral components.

5. We use Equation 4 to obtain the inverse finite field transform of the resulting frequency function. As an example, let us find the time component \( v_3 \) for specific values of \( V_0 = \alpha^2, V_3 = \alpha^6 \) and \( V_6 = \alpha \). We have

\[
v_i = \frac{1}{7 \mod 2} \sum_{j=0}^{6} \alpha^{-ij} \cdot V_j \quad i = 0, \ldots, 6
\]
The number of information symbols, \( k = 3 \). The encoding in this example produces a \((7,3)\) non-binary RS code. If we represent each element of \( GF(8) \) by 3 binary bits then we obtain a \((21,9)\) binary code.

**Example 2:** In this example, the time domain is in \( GF(2) \) and the spectrum domain is in \( GF(8) \). We are interested in obtaining a code which can correct 1 error, so \( t = 1 \). We also use the primitive field element \( a \), so our \( n = 7 \). The steps of the transform encoding is as follows:

1. \( V_1 = V_2 = 0 \)
2. To meet the conjugacy constraints, we determine the conjugacy classes which are:
   \[
   A_0 = \{0\} \\
   A_1 = \{1, 2, 4\} \\
   A_3 = \{3, 6, 5\}
   \]
   The "free" variables are \( V_0 \) and \( V_3 \). However we must meet the condition \( V_0^2 = V_0 \). The "dependent" variables are \( V_4, V_5, \) and \( V_6 \).
3. From \( GF(8) \) the elements 0, 1 are the only elements that satisfy the constraint \( V_0^2 = V_0 \). Pick 0 or 1 for \( V_0 \). The equivalent "bit content" of \( V_0 \) is 1 bit. Select \( V_3 \) to be any field element from \( GF(8) \). The equivalent "bit content" of \( V_3 \) is 3 binary bits. So we should expect to get a \((7,4)\) code.
4. The dependent values are \( V_4 = V_2^2, V_6 = V_5^2 \) and \( V_5 = V_6^2 \) and then since \( V_1 = V_2 = 0 \), we have \( V_4 = 0 \).
5. We use Equation 4 and Table 3 to obtain the inverse finite field transform of the resulting frequency function. Let us find the time domain component \( v_3 \) for specific values of \( V_0 = 1 \) and \( V_3 = \alpha^6 \). First we have \( V_6 = V_3^2 = \alpha^5 \) and \( V_5 = V_6^2 = \alpha^3 \) and then
   \[
   v_i = V_0 + \alpha^{3i} V_3 + \alpha^{5i} V_5 + \alpha^{6i} V_6 \\
   = V_0 + \alpha^{6-3i} + \alpha^{3-5i} + \alpha^{5-6i} \\
   i = 0, \ldots, 6
   \]
   \[
   v_3 = 1 + \alpha^4 + \alpha^2 + \alpha \\
   = 1 + (\alpha^2 + \alpha) + \alpha^2 + \alpha \\
   = 1
   \]
Note that each $v_i$ will be a value from $GF(2)$. This example produces a $(7, 4)$ binary code which is capable of correcting up to 1 error.

5.6 Transform Decoding

We have received a vector $r$ which is an $n$-tuple over $GF(q)$. Because of "noise" this vector may have been corrupted, therefore, we can think of the received vector $r$ as the sum of a valid code word $v$ and an error vector $e$, i.e.,

$$ r = v + e $$

First we want to detect to see if any errors have occurred, if no errors have occurred then we are done and the received vector is a valid code word. If errors have occurred then we want to find the location of the errors and the error values in those locations. So our goal is to find the error vector $e$ which when subtracted from the received vector $r$ will produce the desired code word $v$.

We are interested in decoding BCH codes. In BCH codes, the field elements $\beta^i$ for $i = 1, \ldots, 2t$ are the roots of a valid code word. Or from Theorem 5.1, the finite field transform of a valid code word will contain zeros at locations 1 through $2t$. Therefore, we define the transform decoding method as follows:

1. Find the finite field transform of the received vector $r$, and denote it by the vector $R$.

2. If $R_j = 0$ for $j = 1, \ldots, 2t$ then no errors have occurred and the received vector $r$ is the desired code word. So we don't do any of the following steps.

3. Set $E_j = R_j$ for $j = 1, \ldots, 2t$. These $2t$ values are referred to as the syndrome values. The remaining $n - 2t$ error spectrum components are determined recursively via a method known as the recursive extension of the error spectrum which is explained later in this section.

4. Take the inverse transform of the vector $E$, to obtain the time-domain vector $e$. Subtract the vector $e$ from the vector $r$ to obtain the desired code word $v$.

Another way to do the last step, is to subtract the error spectrum vector $E$ from the received spectrum vector $R$ to obtain the desired spectrum code vector $V$. The inverse finite field transform of the vector $V$ will produce the desired code word in the time-domain.

5.7 Recursive Extension of the Error Spectrum

We are given an error spectrum vector $E$ where the locations $E_j$ for $j = 1, \ldots, 2t$ are known and we want to determine the remaining $n - 2t$ spectrum components. We assume that the actual number of errors which have occurred, denoted by $v$, is less than or equal to $t$. It can be shown that

$$ E_k = - \sum_{j=1}^{v} \Lambda_j E_{k-j} \quad k = 0, \ldots, n - 1 \quad (7) $$

Where $\Lambda_j$ are the coefficients of the error locator polynomial $\Lambda(X)$, i.e.,

$$ \Lambda(X) = 1 + \Lambda_1 X + \Lambda_2 X^2 + \cdots + \Lambda_v X^v \quad (8) $$

19
This equation is defined so that if we find the roots of this equation and find the inverse of each root we obtain the location of each error.

We have a set of \( n \) equations in 7. The number of unknowns in the error spectrum is \( n - 2t \) and the number of unknowns in the error locator polynomial \( \Lambda(X) \) is \( u \). So there are a total of \( n - 2t + v \) unknowns. Of these \( n \) equations we have \( t \) equations which involves the known error spectrum components and the coefficients of the error locator polynomial. These equations are

\[
E_k = -\sum_{j=1}^{t} \Lambda_j E_{k-j} \quad k = t + 1, \ldots, 2t
\]

Since \( \Lambda_j = 0 \) for \( j > v \) we have

\[
E_k = -\sum_{j=1}^{v} \Lambda_j E_{k-j} \quad k = v + 1, \ldots, 2v
\]

This is a set of \( v \) equations and the \( v \) unknowns. The unknowns \( \Lambda_j \) for \( j = 1, \ldots, v \) can be found directly or iteratively. We will examine both a direct method, known as the Peterson method, and an iterative method, known as the Berlekamp-Massey algorithm, to solve for the unknowns.

After the \( \Lambda_j \) coefficients are found then \( n - 2t \) error spectra can be found from Equations 7. These calculations are shown in Table 4. The equations are evaluated from top to the down. In this way, all the variables in the right hand side of the equations are known. We now examine a direct method for finding the coefficients of the error locator polynomial.

### 5.8 Direct Method for Finding the Error Locator Polynomial

We are interested in solving the \( v \) equations in 9 to determine the \( \Lambda_j \) values. These equations can be written in matrix form as

\[
\begin{bmatrix}
E_v & E_{v-1} & E_{v-2} & \cdots & E_1 \\
E_{v+1} & E_v & E_{v-1} & \cdots & E_2 \\
E_{v+2} & E_{v+1} & E_v & \cdots & E_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
E_{2v-1} & E_{2v-2} & E_{2v-3} & \cdots & E_v
\end{bmatrix}
\begin{bmatrix}
\Lambda_1 \\
\Lambda_2 \\
\Lambda_3 \\
\vdots \\
\Lambda_v
\end{bmatrix}
= \begin{bmatrix}
-E_{v+1} \\
-E_{v+2} \\
-E_{v+3} \\
\vdots \\
-E_{2v}
\end{bmatrix}
\]

\[\text{(10)}\]
We can find the coefficients of the error locator polynomial by finding the inverse of the matrix. And multiplying the inverse with the right hand side produces the desired unknowns. But before we can find the inverse, we need to know if the inverse exists. We also need to find the number of errors, \( v \), which has actually occurred. To do this, set \( v = t \) and calculate the determinant. If the determinant is nonzero then inverse exists and the number of errors is \( \neq 0 \). If the determinant is zero, decrease \( v \) by 1, and repeat the process until the determinant becomes nonzero. Since the determinant is nonzero then the inverse exists. This direct method is also known as the Peterson method.

In this method, the number of computations for finding the inverse of the \( t \) by \( t \) matrix is proportional to \( t^3 \) operations. For large \( t \) this method is not feasible and we need to use another method which is more efficient. The Berlekamp-Massey Algorithm is examined next.

5.9 Berlekamp-Massey Algorithm

We are interested in solving the \( v \) equations in \( \beta \) iteratively. This method described here is the Berlekamp-Massey algorithm which is computationally more efficient than the direct method. During iterations, the algorithm builds a smallest length linear-feedback shift register which produces the sequence in \( \beta \). The algorithm is as follows:

1. Initialize

\[
\Lambda(X) = 1 \\
r = 0 \\
L = 0 \\
B(X) = 1
\]

Where \( L \) is the length of the current shift register, \( \Lambda(X) \) is the desired error locator polynomial and \( B(X) \) is an intermediate polynomial used to reduce a nonzero discrepancy. The discrepancy is explained in step 3.

2. Set

\[ r \leftarrow r + 1 \]

3. Compute the \( r \)th discrepancy, \( \Delta_r \). The discrepancy is nonzero when \( (r - 1) \)th shift register does not produce \( E \), correctly.

\[
\Delta_r = E_r + \sum_{j=1}^{L} A_j E_{r-j}
\]

4. If the \( r \)th discrepancy is zero, then \( (r - 1) \) shift register does not need to be modified, it already produces \( E_1, E_2, \ldots, E_r \). Go to step 8 to modify the intermediate polynomial \( B(X) \).

If the \( r \) discrepancy is is nonzero do the next step.
5. Compute new connection polynomial for which the \( r \) discrepancy is made to be zero.

\[
T(X) = \Lambda(X) - \Delta_r XB(X)
\]

So \( T(X) \) is the new shift register which produces \( E_1, E_2, \ldots, E_r \) correctly.

6. Find out if the new shift register needs to be lengthened. From the Berlekamp-Massey theorem we need to have \( L_1 \geq L_{r-1} \) and \( L_r \geq r - L_{r-1} \).

If \( 2L \geq r \) then shift register does not need to be lengthened, store the new shift register into, \( \Lambda(X) \), i.e.,

\[
\Lambda(X) \leftarrow T(X) \quad 2L \geq r
\]

Go to step 8 to modify the intermediate polynomial \( B(X) \).

If \( 2L < r \) then shift register needs to be lengthened, do the next step.

7. We want to store in the intermediate polynomial \( B(X) \) the last shift register which it had nonzero discrepancy and a length increase. This insures that the resulting shift register has the minimal length. The old shift register normalized with its discrepancy is stored in \( B(X) \), i.e.,

\[
B(X) \leftarrow \frac{\Lambda(X)}{\Delta_r}
\]

The new shift register \( T(X) \) for which the discrepancy \( \Delta_r \) was reduced to zero is put in \( \Lambda(X) \), i.e.,

\[
\Lambda(X) \leftarrow T(X)
\]

And the length of the shift register is updated by:

\[
L \leftarrow r - L
\]

Go to step 9 to see if we are done.

8. Modify intermediate polynomial \( B(X) \) as follows:

\[
B(X) \leftarrow XB(X)
\]

Go to next step.

9. Check to see if \( r = 2t \). If the number of iterations \( r \) is less than \( 2t \) then go to step 2 to continue.

If \( r = 2t \) then we are done, go to the next step.

10. If the degree \( \Lambda(X) \) is not equal to \( L \) then more than \( t \) errors have occurred, otherwise \( \Lambda(X) \) is the desired error locator polynomial.

The polynomial updates of \( \Lambda(X) \), \( T(X) \), and \( B(X) \) each require at most \( t \) multiplications. There are a total of \( 2t \) iterations, therefore, it takes \( 6t^2 \) multiplications to find the error locator polynomial. This method is more efficient than the direct method which requires \( t^3 \) operations. Next we give a decoding example which uses the algorithms mentioned above.
5.10 Example of a Transform Decoding

In the Example 1 of the transform encoding we showed an encoding method for a $(7,3)$ RS code. Here, we will give a specific decoding example and show the steps for decoding it.

**Example:** Let us assume a zero vector was sent, but because of the "noise" we received the following vector $r$

$$r = (0, 0, a^6, 0, a^2, 0, 0)$$

Our goal is to find the error vector $e$ which when subtracted from the received vector $r$, produces the desired transmitted vector $v$. Here we should expect to get the zero vector for $v$.

We will use the steps that we gave in the transform decoding subsection which are:

1. Find the finite field transform of the received vector $r$. From Equation 3 we have

   $$R_j = \sum_{i=0}^{6} \alpha^{ij} \cdot r_i, \quad j = 0, \ldots, 6$$

   $$= \alpha^{2j} \cdot r_2 + \alpha^{4j} \cdot r_4$$

   $$= \alpha^{6+2j} + \alpha^{2+4j}$$

   $$R_0 = \alpha^6 + \alpha^2 = (\alpha^2 + 1) + \alpha^2 = 1$$

   $$R_1 = \alpha^1 + \alpha^6 = \alpha + (\alpha^2 + 1) = \alpha^5$$

   $$R_2 = \alpha^3 + \alpha^3 = 0$$

   $$R_3 = \alpha^5 + 1 = (\alpha^2 + \alpha + 1) + 1 = \alpha^4$$

   $$R_4 = 1 + \alpha^4 = 1 + (\alpha^2 + \alpha) = \alpha^5$$

   $$R_5 = \alpha^2 + \alpha = \alpha^4$$

   $$R_6 = \alpha^4 + \alpha^5 = 1$$

**Therefore** the spectrum of the received vector $r$ is:

$$R = (1, \alpha^5, 0, \alpha^4, \alpha^5, \alpha^4, 1)$$

2. All $R_j$ for $j = 1, \ldots, 4$ are not zero therefore errors have occurred.

3. Set $E_j = R_j$ for $j = 1, \ldots, 4$. $E_1 = \alpha^5, E_2 = 0, E_3 = \alpha^4$, and $E_4 = \alpha^5$. The remaining error spectrum components are determined recursively via recursive extension method. To use the recursive extension we need to find the error locator polynomial. We will use both the direct and iterative methods to find the coefficients of the error locator polynomial.

**Direct Method:** Let us first to find out how many errors actually have occurred. To do this, we examine the determinant of the matrix in Equation 10. Assume $v = t = 2$.

$$\det \begin{vmatrix} E_3 & E_2 \\ E_2 & 0 \end{vmatrix} = \det \begin{vmatrix} \alpha^5 & \alpha^4 \\ 0 & 0 \end{vmatrix} = \alpha^2 \neq 0$$

23
Since the determinant is nonzero then 2 errors must have occurred From Equation 10 we have

\[ \begin{align*}
E_2 \Lambda_1 + E_1 \Lambda_2 &= -E_3 \\
E_3 \Lambda_1 + E_2 \Lambda_2 &= -E_4
\end{align*} \]

Substituting into above equations we get

\[ \begin{align*}
\alpha^5 \Lambda_2 &= \alpha^4 \\
\alpha^4 \Lambda_1 &= \alpha^5
\end{align*} \]

Therefore from direct method we obtain the following values for \( \Lambda_1 \) and \( \Lambda_2 \),

\[ \Lambda_1 = \alpha \quad \Lambda_2 = \alpha^6 \]

**Berlekamp-Massey Algorithm**: Here we use the Berlekamp-Massey algorithm to find the coefficients of the error locator polynomial.

For \( r = 0 \),

\[ \begin{align*}
\Lambda(X) &= 1 \\
B(X) &= 1 \\
L &= 0
\end{align*} \]

For \( r = 1 \),

\[ \begin{align*}
\Delta_1 &= E_1 = \alpha^5 \\
T(X) &= \Lambda(X) - \Delta_1 X B(X) = 1 - \alpha^5 X = 1 + \alpha^5 X \\
B(X) \xrightarrow{\Delta_1} \frac{\Lambda(X)}{\Delta_1} = \frac{1}{\alpha^5} = \alpha^2 \\
\Lambda(X) \xrightarrow{T(X)} = 1 + \alpha^5 X \\
L \xrightarrow{r - L} = 1
\end{align*} \]

For \( r = 2 \),

\[ \begin{align*}
\Delta_2 &= E_2 + \Lambda_1 E_1 = 0 + \alpha^5 \alpha^5 = \alpha^3 \\
T(X) &= \Lambda(X) - \Delta_2 X B(X) = (1 + \alpha^5 X) - \alpha^3 X \alpha^2 = 1 \\
\Lambda(X) \xrightarrow{T(X)} = 1 \\
B(X) \xrightarrow{X} = \alpha^2 X
\end{align*} \]

For \( r = 3 \),

\[ \begin{align*}
\Delta_3 &= E_3 + \Lambda_1 E_2 = \alpha^4 + 00 = \alpha^4 \\
T(X) &= \Lambda(X) - \Delta_3 X B(X) = 1 - \alpha^4 X (\alpha^2 X) = 1 + \alpha^6 X^2 \\
B(X) \xrightarrow{\Delta_3} \frac{\Lambda(X)}{\Delta_3} = \frac{1}{\alpha^4} = \alpha^3 \\
\Lambda(X) \xrightarrow{T(X)} = 1 + \alpha^6 X^2 \\
L \xrightarrow{r - L} = 2
\end{align*} \]
For \( r = 4 \)

\[
\Delta_4 = E_4 + \Lambda_1 E_3 + \Lambda_2 E_2 = \alpha^5 + 0\alpha^4 + \alpha^60 = \alpha^5
\]

\[
T(X) = \Lambda(X) - \Delta_4 XB(X) = (1 + \alpha^5X^2) - \alpha^5X \alpha^3 = 1 + \alpha X + \alpha^6 X^2
\]

\[
\Lambda(X) \leftarrow T(X) = 1 + \alpha X + \alpha^6 X^2
\]

\[
B(X) \leftarrow XB(X) = X \alpha^3
\]

Degree \( \Delta(X) = 2 \), therefore, more than 2 errors have not occurred. The results from Berleltamp-Massey Algorithm for \( \Lambda_1 \) and \( \Lambda_2 \) are

\[
\Lambda_1 = \alpha \quad \Lambda_2 = \alpha^6
\]

These were also the values we got from the direct method. Now we are ready to find the remaining \( n - 2t \) spectral components \( E_5, E_6, \) and \( E_0 \). From Table 4 we have,

\[
E_5 = -\Lambda_1 E_4 - \Lambda_2 E_3
\]

\[
= \alpha\alpha^5 + \alpha^6\alpha^4
\]

\[
= \alpha^6 + \alpha^3
\]

\[
= \alpha^4
\]

\[
E_6 = -\Lambda_1 E_5 - \Lambda_2 E_4
\]

\[
= \alpha\alpha^4 + \alpha^6\alpha^5
\]

\[
= \alpha^5 + \alpha^4
\]

\[
= 1
\]

\[
E_0 = -\Lambda_1 E_6 - \Lambda_2 E_5
\]

\[
= \alpha 1 + \alpha^6\alpha^4
\]

\[
= \alpha + \alpha^3
\]

\[
= 1
\]

The resulting error spectrum \( E \) is

\[
E(X) = (1, \alpha^5, 0, 4, \alpha^5, 4, 1)
\]

4. Take the inverse transform of the vector \( E \), to obtain the time-domain vector \( e \). From Equation 4, the inverse finite field transform is calculated as follows:

\[
e_i = \sum_{j=0}^{6} \alpha^{-ij} \cdot E_j \quad i = 0, \ldots, 6
\]

\[
e_0 = E_0 + E_1 + E_2 + E_3 + E_4 + E_5 + E_6
\]

\[
= \alpha^6 + \alpha^5 + 0 + \alpha^4 + \alpha^5 + \alpha + \alpha
\]

\[
= 1 + \alpha^5 + 0 + \alpha^4 + \alpha^5 + \alpha^4 + 1
\]

\[
= 0
\]
The error vector \( e \) then becomes
\[
\mathbf{e} = (0, 0, \alpha^6, 0, \alpha^2, 0, 0)
\]

Subtracting vector \( e \) from the vector \( r \) we obtain the desired code word \( v \) which is the zero code word.

Note that we did not use \( R_0, R_5, \) and \( R_6, \) i.e., Only \( 2t \) spectrum components were involved in the calculations.

As we mentioned above, another way to do the last step of the decoding, was to subtract the error spectrum \( E \) from the received spectrum vector \( R \) to obtain the desired code word spectrum \( V. \) For this example
\[
\mathbf{V} = (0, 0, 0, 0, 0, 0, 0)
\]

Taking the inverse finite field transform of \( V \) produces the code word in the time-domain. We again get the zero code word as before.
6 C++ Class Libraries for Error Control Codes

In this section we give description of several C++ class libraries. These C++ classes provide the following services:

- Arithmetic of polynomials over GF(2).
- Automatic generation of Galois field tables from primitive polynomials.
- Algebra over field elements in GF(2^m).
- Arithmetic of polynomials over GF(2^m).
- Automatic generation of conjugacy classes.
- Encoding and decoding of RS and BCH codes for a given code size n such that up to t errors can be corrected.

First we give description of each C++ class with sample code segments showing how to use each class. Sections 6.6, 6.7, 6.8, and 6.9 describe several programs where these classes are used. For topics in programming languages concepts and constructs and object oriented analysis and design see [13, 3]. For topics in how to do programming in C and C++ languages see references [5, 12, 7, 14, 8].

6.1 C++ Class polybin

This class deals with binary polynomials, i.e., the coefficients of each polynomial is 0 or 1. Table 5 gives sample code segments of how we can use this class. The class provides the following services:

- Sum several binary polynomials.

Table 5: Sample Code Segment for Class polybin.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>To define polynomial ( p_1(X) ) to be ( X^{10} + X^8 + X^4 + X^2 + X + 1 ) do</td>
<td></td>
</tr>
<tr>
<td>int expol[] = { 10, 8, 4, 2, 1, 0 };</td>
<td></td>
</tr>
<tr>
<td>polybin p1( expol, 6 );</td>
<td></td>
</tr>
<tr>
<td>To add polynomials ( p_1(X) ), ( p_2(X) ) and ( p_3(X) ) and print result do</td>
<td></td>
</tr>
<tr>
<td>polybin result = p1 + p2 + p3;</td>
<td></td>
</tr>
<tr>
<td>result.print();</td>
<td></td>
</tr>
<tr>
<td>To find remainder of two polynomials ( p_1(X) ) and ( p_2(X) ) do</td>
<td></td>
</tr>
<tr>
<td>result = p1 % p2;</td>
<td></td>
</tr>
<tr>
<td>To copy polynomial ( p_1(X) ) into polynomial ( p_2(X) ) do</td>
<td></td>
</tr>
<tr>
<td>polybin p2 = p1;</td>
<td></td>
</tr>
<tr>
<td>Result will be zero in the following expression</td>
<td></td>
</tr>
<tr>
<td>result = (p5 % p4) + (p5 % p4);</td>
<td></td>
</tr>
</tbody>
</table>
• Find remainder of two binary polynomials.
• Copy a polynomial to another polynomial.
• Print a binary polynomial.

6.2 C++ Class `gf2m`

This class provides automatic generation of Galois field tables from primitive polynomials. see Sections 3.5 and 3.6. Table 6 gives sample code segments of how we can use this class.

<table>
<thead>
<tr>
<th>Table 6: Sample Code Segment for Class <code>gf2m</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>To construct Galois field $GF(2^m)$, for example $m=3$</td>
</tr>
<tr>
<td>we can use primitive polynomial $X^3 + X + 1$ and do</td>
</tr>
<tr>
<td>int e[] = { 3, 1, 0 };</td>
</tr>
<tr>
<td><code>gf2m gf( e, 3 );</code></td>
</tr>
<tr>
<td>To print the constructed Galois field table do</td>
</tr>
<tr>
<td><code>gf.print();</code></td>
</tr>
<tr>
<td>To find order of field element $\alpha^3$ in $GF(2^3)$ do</td>
</tr>
<tr>
<td>int ans = <code>gf.ord( 6 );</code></td>
</tr>
<tr>
<td>($\alpha^4 = 6$ in vector notation)</td>
</tr>
<tr>
<td>To multiply two field elements $\alpha^2 \cdot 4$ and $\alpha^6 = 5$ do</td>
</tr>
<tr>
<td><code>ans = gf.mul( 4, 5 );</code></td>
</tr>
<tr>
<td>($\text{ans} = 2$)</td>
</tr>
<tr>
<td>To add or subtract two field elements $a$ and $b$ do</td>
</tr>
<tr>
<td><code>ans = gf.add( a, b );</code></td>
</tr>
<tr>
<td>To divide two field elements $a$ and $b$ do</td>
</tr>
<tr>
<td><code>ans = gf.div( a, b );</code></td>
</tr>
<tr>
<td>To find inverse of field element $a$ do</td>
</tr>
<tr>
<td><code>ans = gf.inv( a );</code></td>
</tr>
<tr>
<td>To do $a^c$ where $a$ is a field element and $c$ is a constant do</td>
</tr>
<tr>
<td><code>ans = gf.pow( a, c );</code></td>
</tr>
</tbody>
</table>

The class provides the following services:

• Find order of a field element.
• Multiply two field elements.
• Add or subtract two field elements.
• Divide two field elements.
• Find. inverse of a field element.
• Raise a field element to a power.
• Print generated Galois field table.

6.3 C++ Class polygf

This class deals with Galois Field (GF) polynomials. Here coefficients of each polynomial are elements of a Galois field. Table 7 gives sample code segments of how we can use this class. The class provides the following services:

• Sum two GF polynomials.
• Multiply a GF polynomial with a field element.
• Get the length of resulting polynomial.
• Get ith coefficient of a GF polynomial.
• Get ith exponent of a GF polynomial.
• Multiply a GF polynomial with X. This causes each exponent of the GF polynomial to be increased by 1.
• Multiply a GF polynomial with a field element and X.

Table 7: Sample Code Segment for Class polygf.

```
To define polynomial \( p_1(X) \) to be \( \alpha^6X^4 + \alpha X^2 + 1 \) in \( GF(2^3) \) do
\[
\begin{align*}
\text{int} & \; \text{primpoly}[] = \{ 3, 1, 0 \}; \quad \text{primitive polynomial is} \; X^3 + X + 1 \\
\text{int} & \; \text{expo}[] = \{ 4, 2, 0 \} ;
\text{int} & \; \text{coef}[] = \{ 5, 2, 1 \};
\text{int} & \; \text{lengthGfpoly} = 3;
\text{polygf} \; \text{pl} ( \&\text{gf}, \text{coef}, \text{expo}, \text{lengthGfpoly} );
\end{align*}
\]

To add two GF polynomials \( p_1(X) \) and \( p_2(X) \) and print result do
\[
\begin{align*}
\text{polygf} & \; \text{result} = \text{pl} + \text{p2};
\text{result.print}();
\end{align*}
\]

To find length of GF polynomial \( p_1(X) \) and value of it::second coefficient and value of its third exponent do
\[
\begin{align*}
\text{int} & \; \text{len} = \text{pl}.\text{length}();
\text{int} & \; \text{coefficient} = \text{pl}.\text{coef}(1);
\text{int} & \; \text{exponent} = \text{pl}.\text{expo}(2);
\end{align*}
\]

To multiply GF polynomial \( p_1(X) \) with a field element \( a \) do
\[
\begin{align*}
\text{result} & = \text{pl}.\text{multC}(a);
\end{align*}
\]

To multiply GF polynomial \( p_1(X) \) with \( X \) do
\[
\begin{align*}
\text{result} & = \text{pl}.\text{multX}();
\end{align*}
\]

To multiply GF polynomial \( p_1(X) \) with \( X \) and field element \( a \) do
\[
\begin{align*}
\text{result} & = \text{pl}.\text{multCX}(a);
\end{align*}
\]
```
• Print a GF polynomial.

6.4 C++ Class conjugacy

This class provides automatic generation of conjugacy classes for a given size code word and number of errors to be corrected. See Section 5.3. Table 8 gives sample code segments of how we can use this class. The class provides the following services:

• Generate conjugacy classes for given code size n such that we can correct t errors.

• Get the number of information bits in each code word.

• Fill array V with the contents of the array msg such that the conjugacy constraints are met.

• Extract information from V and fill msg.

• Print the conjugacy classes.

6.5 C++ Class ccode

This class provides encoding and decoding methods for binary and Reed-Solomon codes for arbitrary values of n and t where n is the size of the codeword and t is number of errors to
be corrected. The value \( n \) is based on any given field element. This means that the user has the option to generate primitive and non-primitive codewords. Table 8 gives sample code segments of how we can use this class. The class provides the following services:

- Define an RS code for user specified values of \( n \) and \( t \).
- Define a BCH code for user specified values of \( n \) and \( t \).
- Get number of information symbols in the resultant code
- Encode \( \text{msg} \) of \( k \) elements into a vector \( v \) of \( n \) elements. Encoding is based on filling the spectrum vector \( V \) with \( \text{msg} \) such that the conjugacy constraints are met and then doing inverse finite field transform of \( V \) produces \( v \) a code word in the time domain. See Sections 5.4, 5.5, 5.3, and 5.2.
- Decode a vector \( v \) of \( n \) elements having \( t \) or less errors into \( \text{msg} \) array of \( k \) elements. Decoding is based on taking finite field transform of vector \( v \) to produce spectrum vector \( V \). If an error has occurred Berlekamp Massey algorithm is used to generate

<table>
<thead>
<tr>
<th>Table 9: Sample Code Segment for Class ccode.</th>
</tr>
</thead>
<tbody>
<tr>
<td>To define a Reed-Solomon code of sire ( n = 7 ) capable of correcting ( t = 2 ) information symbols in ( GF(8) ) do</td>
</tr>
<tr>
<td>int ( pp[] = { 3, 1, 0 } ); Primitive polynomial of order in ( = 3 )</td>
</tr>
<tr>
<td>int ( ppLength = 3 );</td>
</tr>
<tr>
<td>int ( beta = 2 ); A field element from ( GF(2^m) )</td>
</tr>
<tr>
<td>int ( n = 7 ); Order of field element ( beta ).</td>
</tr>
<tr>
<td>int ( t = 2 ); Generate RS code</td>
</tr>
<tr>
<td>int ( GenRS = 1 );</td>
</tr>
<tr>
<td>ccode RS( ( pp, ppLength, beta, n, t, GenRS );)</td>
</tr>
</tbody>
</table>

| To get number of information symbols \( k \) in each code word of RS do |
| int num = RS.k(); num\(=\)n-2t |

| To define a BCH code of sire \( n = 63 \) capable of correcting \( t = 3 \) bits do |
| int \( pp[] = \{ 6, 1, 0 \} \); Primitive polynomial of order \( m = 6 \) |
| int \( ppLength = 3 \); |
| int \( beta = 2 \); A field element from \( GF(2^m) \) |
| int \( n = 63 \); Order of field element \( beta \). |
| int \( t = 3 \); ccode BCH( \( pp, ppLength, beta, n, t \);) |

| To get number of information bits \( k \) in each code word of BCH do |
| int num = BCH.k(); num\(=\)45 |

| To encode \( \text{msg} \) array of \( k \) elements and put result into \( v \) array of \( n \) elements do |
| X.encode( \( \text{msg}, v \);) \( X \) is RS or BCH |

| To decode \( v \) array and put resulting \( \text{msg} \) array do |
| X.decode( \( v, \text{msg} \);) \( X \) is RS or BCH |
error locator polynomial. Using Recursive extension method the remaining error spectrum values are obtained. After error spectrum is subtracted from V spectrum, we obtain msg by extracting information from V. See Sections 5.6, 5.10: 5.1, 5.9, and 5.7.

6.6 Error Control Code Program code-bin.cc

This program is written in C++ language and is based on the C++ class ccode of Section 6.5. The program listing is given in the Appendix A.

The input data to this program contains an arbitrary number of lines where each line contains k numbers. Each number is a 0 or 1 and is separated from the next number with one or more white spaces. The program produces (63,45) code and can correct 3 or less errors. As .t is seen from the program listing, the program can easily be modified to produce other code sizes.

The user provides two additional information when the program is run. The first piece of information is the noise level in the communication channel. The noise level is specified by the maximum number of errors maxErr that the channel induces on each encoded word. The second piece of information is a seed number seed for the random number generator. After user specifies maxErr and seed the program does the following:

1. Read an input line.
2. Encode the input line.
3. Pass encoded word through the communication channel. The channel produces a random number between 0,...,maxErr. This number specifies the number of errors to generate on this code word. The channel then generates errors in random locations of the encoded word.
4. Decode the received encoded message from channel, correcting any errors.
5. Print the decoded message.

6.7 Error Control Code Program code-rs.cc

This program is written in C++ language and is based on the C++ class ccode of Section 6.5. The program listing is given in the Appendix B.

The input data to this program contains an arbitrary number of lines where each line contains arbitrary number of ASCII characters. Code size is 25.5 and can correct 3 or less errors. The number of information symbols is 249. The number of errors to be corrected can easily be changed to lower and higher values.

The user tells the program to do encoding by the -e flag and to do decoding by -d flag. Given an ASCII file to the program as input and specifying -e flag, an ASCII output file can be generated. The encoded file will be sequences of numbers, instead of alphanumeric characters. The encoded output file, which is crypted, can be transmitted from point A to point B as if the communication channel is secure. Without the right decoder the contents of the file is not understandable. The encoded file can be corrupted by t errors or less in each code word and the original file can still be obtained by using the -d flag. One application of this program is to send secure e-mail to another party which has the same program. For topics in cryptography see for example [4].
6.8 Error Control Code Program query.cc

This program is written in C++ language and is based on the C++ class code of Section 6.5. The program listing is given in the Appendix C. The program prints \((n,k,t)\) values for several BCH codes from primitive polynomials. No input is required by the user.

6.9 Error Control Code Program nn-output.cc

This program is written in C++ language and is based on the C++ class code of Section 6.5. The program listing is given in the Appendix D.

The user needs to provide the following information when the program is run:

1. `-e` to do encoding or `-d` to do decoding.
2. `m` the order of a primitive polynomial. The valid values for \(m\) is set to be 3 or 5.
3. `t` is maximum number of errors to be corrected.
4. The number of lines in the input file.
5. An optional file name used only in decoding stage. If file `name` is given the program produces a summary report in this file. Each section of summary report starts with a line containing two numbers. The first number is the actual number of errors occurred and the second number is how many vectors had this many errors. Following this line is 0 or more lines stating which vectors had this many errors. Each vector number is printed one per line.

During encoding, the program takes an input line of size \(k\) binary numbers and produces an output line of size \(n\) binary numbers. It repeats this process for all input lines. During decoding, the program takes an input line of size \(n\) binary numbers, correcting any errors and producing an output line of \(k\) binary numbers. Again the process is repeated for all input lines.

7 Concluding Remarks

This paper was aimed to give a tutorial in error control code techniques and show several practical applications. It can also be shown that these techniques can be useful to do classification in neural networks [11, 10].

Another goal of the paper was to outline the design of several C++ class libraries. We also gave a higher level description of these libraries and presented several programs that uses these libraries. These libraries provides the capability to generate RS and BCH codes from primitive polynomials where the user specifies the code size and number of errors to be corrected. Using these libraries we showed power programs in error control codes can be written in less than few pages.
References


A Error Control Code Program code-bin.cc

Following is the program listing for BCH code of $n = 63, t = 3$, see Section 6.6. As it is seen, the program can easily be modified to generate other code sizes.
/*
 * Author: Jamshid Nazari Purdue University
 *
 * (c) COPYRIGHT 1993, by Jamshid Nazari. All rights reserved.
 */

// -< C++ -<-
// code-bin.cc

handleSubmit();

This is a sample program which shows how we encode binary data

//

// Modification History:

February 18, 1993 - (jn) Original version is written.

//

---

#include "ccode.h"

void binary-channel( int v[], int n, int max_err);

inline inl: getRnum( int M )
{
    return( rand() % M );
}

main( int argc, char **argv )
{
    if ( argc != 3 )
    {
        fprintf( stderr, "Usage: %s max-err seed\n", argv[ 0 ] );
        exit( 1 );
    }
    int max_err = atoi( argv[ 1 ] );
    int seed = atoi( argv[ 2 ] );
    srand( seed );

    int e6[] = { 6, 1, 0 }; // primitive polynomial, m=6
    int n = 63;
    int t = 3;
    ccode bch( e6, 3, 2, n, t ); // alpha=2

    int k = bch.k(); // number of information digits
    int *msg = new int[ k ]; assert( msg != 0 ); // information itself
```c
int *v = new int[n]; assert(v != 0); // encoded (received) msg

register int i;
int blknum = 1;
while (!cin.eof())
{
    for (i = 0; i < k; i++) // read a message.
        cin >> msg[i];
    if (!cin.good())
        break;

    bch.encode(msg, v);
    binary_channel(v, n, max_err);
    int rv = bch.decode(v, msg);
    if (rv > t)
        fprintf(stderr, "More than %d errors in block %d.\n", t, blknum);
    else
        if (rv > 0)
            fprintf(stderr, "%d errors have occurred in block %d.\n", rv, blknum);
    for (i = 0; i < k; i++)
        cout << msg[i] << " ";
    cout << "\n";
    blknum++;
}
delete [] msg;
delete [] v;

void binary_channel(int v[], int n, int max_err)
{
    int num_err = getRnum(max_err+1); // number of errors to generate
    cerr << "Channel generated " << num_err << " errors\n";

    register int i, j;
    for (i = 0; i < num_err; i++)
    {
        j = getRnum(n);
        if (v[j] == 0)
            v[j] = 1;
        else
            v[j] = 0;
    }
}
```
B  Error Control Code Program code-rs.cc

Following is the program listing for Reed-Solomon code of $n = 255$, $t = 3$, see Section 6.7. As it is seen in the program, the number of errors to be corrected can easily be changed to lower and higher values.

```c
/*
 *----------------------------------------------------------------------
 * This is a sample program which shows how RS code can be defined
 * and used.
 *----------------------------------------------------------------------
 */

// - * - C++ - * -
// code - ~-.
// ----------------------------------------------------------------------
// Modification History:
// February 18, 1993  - (jn) Original version is written.
//-----------------------------------------------------------------------

#include "ccode.h"

const int SYNC = 99999;

void encode( ccode *rs, int n, int km );
void decode( ccode *rs, int n, int km, int t );

main( int argc, char *argv[] )
{
    if ( argc != 2 )
    {
        fprintf( stderr, "Usage: %s [-ed]\n", argv[0] );
        exit(1);
    }

    int e8[] = {8, 4, 3, 2, 0}; // primitive polynominal, m=8
    int e8-len = 5;
    int n = 255;
    int t = 3;  // maximum # of error can correct
    ccode rs( e8, e8_len, 2, n, t, 1 ); // alpha=2

    int k = rs.k();              // number of information digits
    cerr << "k= " << k << "\n";
    int km = k - 1;              // use 1 byte as the length of block
```
if ( strcmp( argv[1], "-e" ) == 0 )
{
    encode( &rs, n, km );
    exit( 0 );
}

if ( strcmp( argv[1], "-d" ) == 0 )
{
    decode( &rs, n, km, t );
    exit( 0 );
}

printf( stderr, "Usage: %s [-ed][n], argv[0] ");
}

// main()

void encode( ccode *rs, int n, int km )
{
    int *msg = new int[ km+1 ]; assert( msg != 0 ); // information itself
    int *v = new int[ n ]; assert( v != 0 ); // encoded (received) msg
    unsigned char ch;
    register int i, count;
    while ( 1 )
    {
        count = 0;
        while ( cin.good() && count < km )
        {
            cin.get( ch );
            msg[ count++ ] = ch;
        }
        if ( !cin.good() )
            count--;
        msg[ km ] = count;
        rs->encode( msg, v );
        cout << SYNC << "\n";
        for ( i = 0; i < n; i++ )
        {
            cout << v[ i ] << "\n";
        }

        if ( count != km )
            break;
    }
    delete[] msg;
    delete[] v;
}

// encode()
void decode( ccode *rs, int n, int km, int t )
{
    int *msg = new int[ km+1 ]; assert( msg != 0 ); // information itself
    int *v = new int[ n ]; assert( v != 0 ); // encoded (received) msg
    int blknum = 1;
    int x
    register int i, rv, len;
    while ( cin >> x, cin.good() && x == SYNC )
    {
        for ( i = 0; i < n; i++)
            cin >> v[ i ];
        rv = rs->decode( v, msg );
        if ( rv > t )
            fprintf( stderr, "More than %d errors in Block %d.\n", t, blknum );
        else
            if ( rv > 0 )
                fprintf( stderr, "%d errors have occurred in Block %d.\n", rv, blknum );
        len = msg[ km ];
        for ( i = 0; i < len; i++)
            printf( "%c", msg[ i ] );
        blknum++;
    }
    delete [] msg;
    delete [] v;
    if ( cin.good() )
    {
        cerr << "\nSync error in block " << blknum << "\n";
        exit( 1 );
    }
} // decode()

C Error Control Code Program query.cc

Following is the program listing which prints \((n,k,t)\) values for several BCH codes, see Section 6.8.

/*****
// Author: Jamshid Nazari Purdue University
***/
(C) COPYRIGHT 1993, by Jamshid Nazari. All rights reserved.

39
This program prints \((n,k,t)\) for several BCH codes.

Modification History:

February 18, 1993 - (jn) Original version is written.

#include "codel.h"

void m3()
{
    int e3[] = { 3, 1, 0 }; // primitive polynomial. \(m=3\)
    int n = 7;
    int t = 1;
    ccode bchl(e3, 3, 2, n, t); // alpha=2
    printf( "n=%d, t=%d, k=%d\n", n, t, bchl.k() ); // # of info digits
}  // m3()

void m4()
{
    int e4[] = { 4, 1, 0 }; // primitive polynomial. \(m=4\)
    int n = 15;
    int t = 1;
    ccode bchl(e4, 3, 2, n, t); // alpha=2
    printf( "n=%d, t=%d, k=%d\n", n, t, bchl.k() ); // # of info digits

    t = 2;
    ccode bch2(e4, 3, 2, n, t); // alpha=2
    printf( "n=%d, t=%d, k=%d\n", n, t, bch2.k() );

    t = 3;
    ccode bch3(e4, 3, 2, n, t); // alpha=2
    printf( "n=%d, t=%d, k=%d\n", n, t, bch3.k() );
}  // m4()

void m5()
int e5[] = {5, 2, 0}; // primitive polynomial, m=5
int n = 31;

int t = 3;
ccode bch1(e5, 3, 2, n, t); // alpha=2
printf("n=%d t=%d k=%d\n", n, t, bch1.k()); // # of info digits

int t = 4;
ccode bch2(e5, 3, 2, n, t); // alpha=2
printf("n=%d t=%d k=%d\n", n, t, bch2.k());

int t = 5;
ccode bch3(e5, 3, 2, n, t); // alpha=2
printf("n=%d t=%d k=%d\n", n, t, bch3.k());

int t = 6;
ccode bch4(e5, 3, 2, n, t); // alpha=2
printf("n=%d t=%d k=%d\n", n, t, bch4.k());

int t = 7;
ccode bch5(e5, 3, 2, n, t); // alpha=2
printf("n=%d t=%d k=%d\n", n, t, bch5.k());
}

main(int argc, char **argv)
{
    m3();
    m4();
    m5();
}

D Error Control Code Program nn-output.cc

Following is the program listing which can do encoding and decoding for several BCH codes, see Section 6.9.

/*
   % Author: Jamshid Nazari Purdue University
   % (C) COPYRIGHT 1993, by Jamshid Nazari. All rights reserved.
*/

// -e- C++ -e-
// nn-output.cc
//
//                    41
This program performs encoding and decoding. The input is read from standard input and the output is written into standard output. see usage().

Format: of <summary-file> (when present) is as follows:
<#errors> <freq>
<line#>
...
<#errors> <freq>
<line#>
...

Example <summary-file>

0 3
12
...
15
27
1 0
...
3 2
...
13
22

In this example:
- Lines 12, 15, 27 had 0 errors. The line# are in increasing order.
- No lines had 1 error.
- No lines had 2 errors.
- Lines 13 and 22 had 3 errors. The line# are in increasing order.

Modification History:
February 18, 1993 - (ja) Original version is written.

#include "ccode.h"

void usage( char *str )
{
    fprintf( stderr, "Usage: %s -(e|d) <m> <t> <num_vects> [summary_file]\n", str );
    cerr << "<m>=(3|5) order of primitive polynomial\n";
    cerr << "<t>=number of bits to correct\n";
    exit( 1 );
}

int check_err_occurs( int rv, int t, int blknum )
{
    if ( rv == 0 )
        return 0;

    if ( rv <= t )
        fprintf( stderr, "%d errors in Block %d.\n", rv, blknum );
    else
        fprintf( stderr, "More than %d errors in Block %d.\n", t, blknum );

    return rv;
void encode( ccode *bch, int msg[], int v[], int n, int k )
{
    register int i;
    int last = n - 1;
    while ( !cin.eof() )
    {
        for ( i = 0; i < k; i++ )       // read a message.
            cin >> msg[i];
        if ( !cin.good() )
            break;

        bch->encode( msg, v );

        for ( i = 0; i < n; i++ )
            if ( i != last )
                cout << v[i] << " ";
            else
                cout << v[i];
        cout << "\n";
    }
}       // encode()

void decode( int argc, char **argv, ccode *bch, int msg[] )
{
    int **summary = new int*[t+2]; assert( summary != 0 );
    int **summarycnt = new int*[t+2]; assert( summarycnt != 0 );
    int num_vects = atoi( argv[4] );
    register int i, j;
    for ( i = 0; i < (t+2); i++ )
    {
        summary[i] = new int[num_vects]; assert( summary[i] != 0 );
        summarycnt[i] = 0;
        for ( j = 0; j < num_vects; j++ )
            summary[i][j] = 0;
    }

    int blknum = 1;
    int last = k - 1;
    while ( !cin.eof() )
    {
        for ( i = 0; i < n; i++ )       // read encoded data
            cin >> v[i];
        if ( !cin.good() )
            break;

    }
}       // decode()
main( int argc, char **argv )
{
  if ( argc != 5 || argc != 6 )
    usage( argv[ 0 ] );

  int m = atoi( argv[ 2 ] );
  if ( m != 3 || m != 5 )
    usage( argv[ 0 ] );

  j = check_err_occ( bch->decode( v, msg ), t, blknum );
  summary[ j ][ summarycnt[ j ]++ ] = blknum++;

  for ( i = 0; i < k; i++ )
    if ( i != last )
      cout << msg[ i ] << " ";
    else
      cout << msg[ i ];
  cout << "\n";

  if ( argc == 6 )
  {
    char summary_file[ 80 ];
    FILE *outfile;
    strcpy( summary_file, argv[ 5 ] );
    outfile = fopen( summary_file, "w" );
    if ( outfile == NULL )
    {
      fprintf(stderr, "File %s cannot be written.\n", summary_file);
      exit( 1 );
    }

    for ( i = 0; i < ( t+2 ); i++ )
    {
      fprintf( outfile, "%d %d\n", i, summarycnt[ i ] );
      for ( j = 0; j < summarycnt[ i ]; j++ )
        fprintf( outfile, "%d\n", summary[ i ][ j ] );
    }
    fclose( outfile );

    for ( i = 0; i < ( t+2 ); i++ )
      delete [] summary[ i ];
    delete [] summary;
    delete [] summarycnt;
  }  // decode()

  for ( i = 0; i < ( t+2 ); i++ )
    delete [] summary[ i ];
}
int t = atoi(argv[3]);
int e3[] = {3, 1, 0};     // primitive polynomial, m=3
int e5[] = {5, 2, 0};     // primitive polynomial, m=5
int n, len;
int *e;
switch (m) {
case 3:
    n = 7;
    e = e3;
    len = 3;
    break;

case 5:
    n = 31;
    e = e5;
    len = 3;
    break;
}
code bch(e, len, 2, n, t); // alpha=2

int k = bch.k();     // number of information digits

int *msg = new int[k]; assert(msg != 0); // information itself
int *v = new int[n]; assert(v != 0);     // encoded (received) msg

if (strcmp(argv[1], "-e") == 0)
    encode(&bch, msg, v, n, k);
else if (strcmp(argv[1], "-d") == 0)
    decode(argc, argv, &bch, msg, v, n, k, t);
else
    usage(argv[0]);

delete [] msg;
delete [] v;
} // main()