5

Block Codes

Coding Theorist's Pledge: I swear by Galois that I will be true to the noble traditions of coding theory; that I will speak of it in the secret language known only to my fellow initiates; and that I will vigilantly guard the sacred theory from those who would profane it by practical applications.

J. L. Massey

In Chapter 4, we established the possibility of reliable communication over various memoryless channels, provided the attempted information rate is less than the channel capacity. In this chapter, we take up the constructive aspect of the coding problem, designing and building encoding and decoding schemes that approach the promise of Chapter 4. Here we shall be concerned with block codes; trellis codes are described in Chapter 6.

Block encoders are presented with a message block of k symbols from an alphabet of size D, and the encoder produces a codeword of length n, the code symbols taken from an alphabet of size q. These n code symbols depend only on the k input symbols of the current block. For uniqueness of the encoding, we must have $D^k \leq q^n$. Usually, but not always, the input and output alphabets for the encoder are of the same size. We shall assume this is the case here, and let this common alphabet size be q. Most often

this is a binary ($q = 2$) alphabet relation, but there are several interesting cases where nonbinary codes offer performance advantages. We refer to such an encoding relation as an ***(n, k) code over the q-ary alphabet.*** The ***rate*** R of the code is defined as $R = k/n$ message symbols per code symbol, or equivalently $(k/n) \log_2 q$ message bits per code symbol. Assuming that codewords are selected for transmission with equal probability, the entropy of the codeword selection process is $k \log_2 q$ bits per codeword, so in this case the code's designated rate is the amount of information passed per symbol, assuming a perfect channel.

This description of a block code is simple enough and very general, requiring only more details of the mapping from message blocks to codewords to be complete. We will need to restrict our attention to more special codes, however, to make implementation of the encoder, and especially the decoder, feasible. To see why implementation issues are of concern, consider the encoding task. In general, we must resort to use of a *codebook* in the form of a read-only memory having q^k entries of length n symbols, as shown in Figure 5.0.1. For the simplest case of $q = 2$, long (and powerful) codes of modern interest, for example, with $k = 64$, imply an enormous codebook, even for the microelectronic age. The problem is simply exponential growth of the code size with message length. The decoder's job is even more challenging, for a brute-force decoder would evaluate all q^k codewords against the received vector, $\mathbf{r}$, also of length n, and find that codeword with largest likelihood. A faster, but more memory-intensive approach, would be to precompute a table of size Q^n, where Q is the (finite) alphabet size for the demodulator output, with each table entry storing the most likely message k-tuple. Again, we are faced with exponential complexity as a function of blocklength. What we need instead are codes with encoding rules that are algorithmic; that is, they allow fast computation of the codeword and possess short-cut routes to maximum likelihood decoding, or at least nearly maximum likelihood decoding.

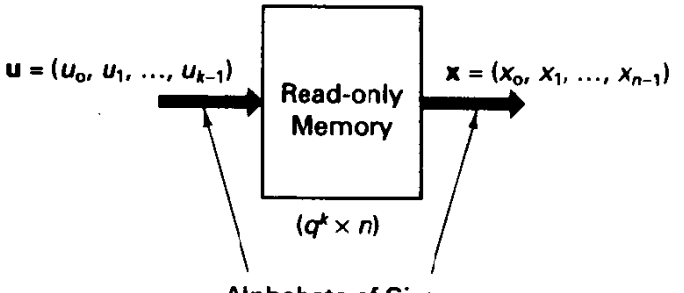

Figure 5.0.1 Table lookup encoder for general (n, k) block code.

To understand the essence and basic terminology of block codes, we first consider perhaps the archetypal block code, the (7, 4) binary code due to Hamming, which we will see later is a member of an infinite family of Hamming codes. This particular code provides a convenient working example for the chapter; it's small enough in size to be manageable, yet possesses most of the interesting ingredients of good codes. This code is also the first nontrivial code to be introduced in coding history.

5.0 THE (7, 4) BINARY HAMMING CODE

The code is described with the aid of the Venn diagram shown in Figure 5.0.2.[1] There are seven distinct regions labeled $i_1, i_2, i_3, i_4, p_1, p_2, p_3$, in each of which we may place a red or green token. We encode as follows: in each slot i_j, place a message (or information) token. There are 16 such color patterns, or messages. In each of slots p_j, commonly called parity positions, place another token such that the number of red tokens found in each of the three circles is even, that is, 0, 2, or 4. To each of the $2^4 = 16$ messages there corresponds a unique seven-token (binary) color pattern.

With this encoding, we have installed the memory and redundancy cited in Chapter 4 as the essential features in coding for error control. The influence of the information bits is diffused throughout the codeword, three information bits affecting three positions each, while the information bit i_2 affects four positions of the codeword. Also, the codewords convey four bits of information, not seven, so the seven tokens are in some sense mutually redundant. To see what can be gained from this encoding process, we assume a color-changing channel that occasionally changes red to green, or vice versa, with probability ϵ and that each token is processed independently.

Without coding, the message error probability for a four-bit message would be

$$P_{e_{\text{no coding}}} = 1 - (1 - \epsilon)^4 \approx 4\epsilon \tag{5.0.1}$$

for small ϵ. With the (7, 4) code, however, we are capable of correcting any single error, whether it occurs in the information slots or in a parity position, as is easy to show. First, note that between any pair of seven-token color patterns there are at least three color disparities, one of which is a message position. From the discussion in Chapter 4, we know that maximum likelihood decoding corresponds to finding that color pattern among the 16 code patterns that has the most color matches with the received pattern, at least

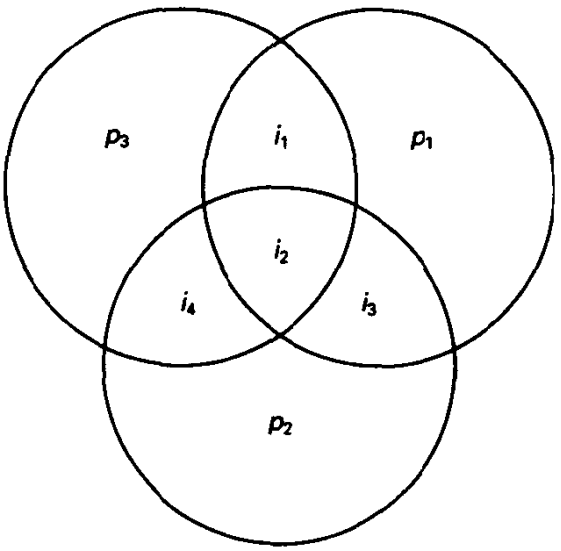

Figure 5.0.2 Venn diagram illustration of (7, 4) binary Hamming code.

[1] This presentation is borrowed from R. McEliece, shown at the 1982 Information Theory Symposium.

provided $\epsilon \leq \frac{1}{2}$. Since valid patterns differ in at least three slots, meaning the minimum Hamming distance between codewords is 3, a single error will still leave the transmitted message with a higher likelihood than any other, thus allowing correction of the error. On the other hand, if two or more color errors happen, we can be assured that the decoder will produce an incorrect message estimate, so the coding system is not foolproof.

The postdecoding error probability is

$$\begin{aligned} P_{e_{\text{coding}}} &= P(2 \text{ or more errors in 7 positions}) \\ &= \sum_{i=2}^{7} C_i^7 \epsilon^i (1-\epsilon)^{n-i}, \end{aligned} \tag{5.0.2}$$

which, for small ϵ, is about $21\epsilon^2$, a clear improvement over (5.0.1). In fact, (5.0.2) is smaller than (5.0.1) for any ϵ.

The cost of this improvement is twofold:

1. Seventy-five percent more transmissions, or storage locations, are required for the coded scheme; equivalently, the code rate is only $R = 4/7$.
2. Additional encoding and decoding complexity occurs.

Regarding the complexity issue, we have given an algorithm for producing codewords that avoids the need for a codebook, or table, although in this simple example table encoding would be no problem. Furthermore, we may shortcut the "examine all codewords" approach in decoding by the following approach: let the decoder perform ***parity checks*** on the received pattern, that is, test whether each circle possesses an even number of red tokens. If an error occurs in only position i_2, the decoder will report failures on each circle (all will have an odd number of red tokens, irrespective of the message sent). If a single error occurs in any of i_1, i_3, or i_4, precisely two parity checks fail. Finally, if a single error occurs in one of the p_j slots, a single check fails. This set of three tests produces what is called the ***syndrome*** of the error sequence, in analogy with the syndrome of a disease, a set of symptoms a physician may use to characterize a certain condition, but that in general does not uniquely identify the cause of the illness. Notice also that this syndrome depends only on the error pattern and not the message. Since all the zero-error or one-error cases produce unique syndromes, we may correctly identify the location of color errors. On the other hand, we may be assured that if two or more errors occur the decoder will certainly fail, since all error patterns with two or more errors produce syndrome patterns identical to the zero- or one-error syndromes. (The sharp demarcation between guaranteed success and failure is a rather special property of this code and a few others, called perfect codes.)

You have undoubtedly realized that there is a logical, that is, Boolean, description of this system and based on it, a high-speed electronic encoder/decoder. It is formed by letting *red* $\rightarrow$ 1 and *green* $\rightarrow$ 0. Then the encoding rules are

$$\begin{aligned} p_1 + i_1 + i_2 + i_3 &= 0, \\ p_2 + i_2 + i_3 + i_4 &= 0, \\ p_3 + i_1 + i_2 + i_4 &= 0, \end{aligned} \tag{5.0.3}$$

where addition of 0 and 1 is modulo 2. The zero-sum constraints require that the number of 1's (red tokens) in a circle be even. Given $\mathbf{i} = (i_1, i_2, i_3, i_4)$, the p_j may be computed by (5.0.3), and the complete codeword is $\mathbf{x} = (\mathbf{i}, \mathbf{p}) = (i_1, i_2, i_3, i_4, p_1, p_2, p_3)$.

Decoding consists of recomputing the left-hand side for each equation in (5.0.3) and testing for zero. Depending on how many of the three tests are nonzero, our procedure gives us the most likely, but possibly wrong, message vector.

Despite its almost trivial nature, this example raises several ideas and questions fundamental to code design, which we shall extend and formalize in the remainder of this chapter.

1. The role of distance between codewords is clear; two codewords are less likely to become confused in the presence of errors if they have a large distance, in this case measured by the number of color disparities, equivalent to Hamming distance. (For other channels other measures of distance are more appropriate.)
2. For binary messages with $k = 4$, we may ask whether $n = 7$ is the smallest value that guarantees the correction of single errors. Or with $k = 4$, what is the smallest n ensuring double-error correction? (The answers, as we shall see, are "yes" and "11"). More generally, the relationship between k, n, and the error-processing power is of interest.
3. What happens if the Venn diagram positional assignments of i_1 and p_1 are reversed, with the same circle rules still in effect, thus changing the code?
4. In the scheme shown, the message to be encoded also appears explicitly in the codeword, referred to as ***systematic encoding***. Is this good, bad, or indifferent?

Aside from the perfect nature of the example code, perhaps the most remarkable aspect of the code is that it is a ***linear*** code. This implies that there is a linear set of equations, (5.0.3), defined on the binary alphabet, that form the code words, and this allows a relatively simple electronic implementation. Just as important for analysis purposes is the fact that linearity gives the code a symmetry property, wherein the performance of the system depends only on the error pattern that occurred, and not on which message was selected. Furthermore, there is a linear relation between the error pattern and the syndrome vector, which is a sufficient statistic for decoding when the demodulator performs hard (binary) decisions on each code symbol. Virtually all block codes in practice are linear codes for these and other reasons. In fact, most all important codes possess additional algebraic structure, which admits even more reductions in complexity. These are known as ***cyclic*** codes. We shall take up these topics shortly after a brief review of the theory of finite fields, which is necessary to understanding the greater potential of block codes.

5.1 ALGEBRA OF FINITE FIELDS

We have just seen an example of a code defined on the alphabet of numbers 0 and 1 and invoked familiar rules for performing arithmetic with these elements. For implementation reasons, more general codes will be restricted to alphabets that are algebraic objects

known as ***fields***, of which the binary field is the simplest example. Extending our scope to larger fields is necessary for two reasons: we will be interested in nonbinary codes and must know how to perform arithmetic for such codes. Second, the theory of finite fields is central to the description and operation of powerful encoders and decoders, even for binary codes. Another contemporary application of this material is in cryptography, but we shall not delve into this.

The topic of finite fields has an immense and elegant literature, and detailed treatments oriented toward digital communications applications can be found in texts by Peterson and Weldon [1], Berlekamp [2], Lin and Costello [3], Blahut [4], MacWilliams and Sloane [5], and Michelson and Levesque [6], to name a few. Our coverage will be rather descriptive, aiming toward the essential tools for the practitioner.

A ***field*** is an algebraic system formed by a collection of elements, F, together with dyadic (two-operand) operations $+$ and $*$ called addition and multiplication, which are defined for all pairs of field elements in F and which behave in an arithmetically consistent manner. Specifically, we require that for every α, β, γ contained in F

$$\begin{aligned} \alpha + \beta &\in F \\ \alpha + (\beta + \gamma) &= (\alpha + \beta) + \gamma \\ \alpha + \beta &= \beta + \alpha \\ \alpha * \beta &\in F \\ \alpha * \beta &= \beta * \alpha \\ \alpha * (\beta * \gamma) &= (\alpha * \beta) * \gamma \\ \alpha * (\beta + \gamma) &= \alpha * \beta + \alpha * \gamma. \end{aligned} \tag{5.1.1}$$

Thus, a field is said to be closed under addition and multiplication, and the usual associative, distributive, and commutative rules hold. Furthermore, we can identify an element in F (call it 0) that is the additive identity element ($\alpha + 0 = \alpha$), and for every α there must exist a unique additive inverse element β such that $\alpha + \beta = 0$. Likewise, there is an element denoted by 1, called the multiplicative identity, such that $\alpha * 1 = \alpha$, and every nonzero α has a unique multiplicative inverse, or reciprocal, $\beta = \alpha^{-1}$ such that $\alpha * \beta = 1$. Subtraction in the field F is performed by adding the additive inverse element, and division (by an element other than 0) is accomplished by multiplying by the multiplicative inverse element.

The real numbers, denoted R, together with addition and multiplication as taught in primary school, form a field. However, the set of integers together with normal arithmetic rules is not a field, there being no integer-valued multiplicative inverse for every nonzero integer. Also, the complex numbers $a + jb$, where a and b are real, form a field denoted C, with addition and multiplication performed as usual for complex numbers.

A ***finite field***, or Galois field (after E. Galois, noted French algebraist of the early 19th century), is a field with a finite number, q, of elements, and is denoted here by GF(q). (In various texts the finite field is also denoted GF_q or F_q). The simplest field is the binary field, GF(2) = {0, 1}, with addition performed modulo 2 and multiplication according to the logical "and" function. Addition and multiplication tables are shown next.

Addition		
+	0	1
0	0	1
1	1	0

Multiplication		
*	0	1
0	0	0
1	0	0

Of practical importance is the fact that addition and subtraction are equivalent in the binary field, and multiplication is trivial—multiplication of any field element by 0 or 1 produces a product 0 or the field element, respectively, so multiplication is actually not required.

A field of size three, GF(3), although of little practical interest in digital communications, is formed by the set of elements $\{0, 1, 2\}$ with addition and multiplication performed as usual for integers, except the results are represented modulo 3. Thus, $2 * 2 = 1$, and therefore 2 is its own reciprocal. This tempting construction does not supply a field, however, unless q is prime [you might try it for $q = 4$ and discover which of the field requirements of (5.1.1) fails].

A result of number theory is that finite fields exist only for q equaling a prime number, p, or power of a prime, $q = p^m$. Thus, we are simply unable to construct a field of size 6 or, more disappointing, 10, satisfying the requirements of (5.1.1). We have seen how to construct fields and perform arithmetic when q is prime; for prime-power fields of size $q = p^m$, called ***extension fields***, a construction based on polynomials over the prime field GF(p) is standard.

5.1.1 Polynomials over Fields and Extension Fields

A polynomial $a(D)$ of degree m, over a finite field of size p,[2] is written as

$$a(D) = a_0 + a_1 D + \cdots + a_m D^m, \tag{5.1.2}$$

where the coefficients a_i are elements in GF(p), with $a_m \neq 0$, and D is an indeterminate. The ***degree*** of the polynomial is the largest exponent. We refer to the polynomial as ***monic*** if the coefficient of the highest-degree term is the field element 1. For example, $1 + D^2$ is a second-degree polynomial over the binary field, and $D + 2D^3$ is a third-degree polynomial over GF(3), although not monic. Such polynomials are like polynomials over the real-number field; they can be added, multiplied, and possibly be factored into products of smaller-degree polynomials over the same field. The fundamental theorem of algebra still pertains and holds that polynomials of degree m also have m roots, x_i,

[2] p does not need to be prime to define such polynomials, although our present interest is in this case.

such that $a(x_i) = 0$, although the roots may not be in the field of the coefficients, called the ***ground field***, but may lie in some extension field. A familiar example is that the polynomial $1 + D^2$ over the real numbers has no roots in the field of real numbers, but does in the field of complex numbers, specifically $x_i = j$ and $-j$, where j is implicitly defined as a root of this polynomial. This is why j is denoted an imaginary number.

We define the addition of two polynomials $a(D)$ and $b(D)$ to be another polynomial $c(D)$, where the coefficients $c_i = a_i + b_i$ are obtained by addition rules for the field of the coefficients. Thus, $(1+D^2)+(1+D+D^2) = D$ over the binary field, since the coefficients for the degree-0 terms add to 0, and likewise for the degree-2 coefficients. Similarly, multiplication of polynomials is performed using the long hand method, coefficients again being handled according to the rules of the field. For example, over GF(2)

$$(1 + D) * (D + D^2) = D + D^2 + D^2 + D^3 = D + D^3. \tag{5.1.3}$$

(Notice that it is imperative to be clear on the coefficient field of a polynomial to unambiguously perform arithmetic.)

A polynomial $a(D)$ over a field GF(q) is said to be ***irreducible*** if it cannot be factored into products of lesser-degree polynomials over the *same* field. The polynomial $1 + D^2$ over GF(2) is not irreducible since $1 + D^2 = (1 + D)^2$ in binary arithmetic, but $1 + D + D^4$ is irreducible, as may be verified by testing all binary polynomials of degree 2 or less as divisors. An irreducible polynomial has none of its roots in the ground field, but this is not a sufficient condition for irreducibility; the polynomial $a(D) = D^4+D^2+1$ has no roots in GF(2) since $a(0) \neq 0$ and $a(1) \neq 0$, but $a(D)$ may be factored into the product of two GF(2) polynomials as $a(D) = (D^2 + D + 1)^2$.

To form an ***extension field*** of size $q = p^m$, we take the field elements to be all polynomials[3] of degree $m - 1$ over GF(p), of which there are p^m. Addition and multiplication of elements are performed by the usual rules for polynomial arithmetic, except we define the results of multiplication (which may produce a polynomial of degree m or larger) as the remainder upon division by an irreducible polynomial of degree m. We will indicate this as $c(D) = a(D) * b(D) \bmod f(D)$, where $f(D)$ is irreducible. (Notice that the addition of two polynomials of degree $m - 1$ or less produces a polynomial of degree $m - 1$ or less, hence there is no formal need for reduction.) This reduction, modulo an irreducible polynomial, is analogous to reducing results modulo a prime number for prime fields such as GF(3). The additive identity element of the field is clearly the polynomial $a(D) = 0$, and the multiplicative identity element is the polynomial $a(D) = 1$.

A more detailed treatment of finite fields would verify at this point that all the field axioms (5.1.1) are satisfied by this construction. Some are verified easily, such as the closure and commutative properties. Proving that a unique reciprocal exists for every nonzero element is slightly more involved; there the requirement for reducing products of polynomials modulo an irreducible polynomial will be seen as crucial. (See Exercise 5.1.5.)

We now illustrate the construction of GF(8), an extension field of GF(2).

---

[3]Some confusion frequently results in identifying field elements with polynomials; keep in mind that these are only labels for the elements, as in any numbering scheme, and this polynomial labeling provides a convenient means of manipulating field elements.

Example 5.1 Description of $GF(2^3) = GF(8)$

The eight elements of the field can be represented by the binary polynomials 0, 1, D, $1 + D$, D^2, $1 + D^2$, $D + D^2$, and $1 + D + D^2$. Polynomial form is one representation of the elements, convenient for manipulation. We could also label each element with a 3-bit vector (the vector of coefficients of each polynomial), or we could label the elements as 0 through 7 by taking the integer equivalent of each binary representation. Figure 5.1.1 provides such a listing of these possibilities.

To perform arithmetic in this field, we select $f(D) = 1 + D + D^3$ as a third-degree irreducible polynomial over GF(2). [We can quickly verify that $f(D)$ is irreducible by testing the first-degree polynomials D and $D + 1$ as divisors.] Now, suppose we wish to add 2 and 7. By adding polynomials or, more easily, the binary vectors modulo 2, we find that the sum is 5. Likewise, the product of 2 and 7 through polynomial multiplication gives

$$D * (D^2 + D + 1) \bmod (D^3 + D + 1) = D^2 + 1, \tag{5.1.4}$$

Polynomial Form	Integer Form	m-tuple Form
0	0	000
1	1	001
D	2	010
$D + 1$	3	011
D^2	4	100
$D^2 + 1$	5	101
$D^2 + D$	6	110
$D^2 + D + 1$	7	111

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

Addition Table

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

Multiplication Table

Figure 5.1.1 Construction of GF(8) using $f(D) = D^3 + D + 1$ as irreducible polynomial.

so the product of 2 and 7 also equals 5. A complete addition and multiplication table can be constructed in this manner and is shown in Figure 5.1.1. We also note that if a different irreducible polynomial were employed to perform arithmetic, for example, $f(D) = 1 + D^2 + D^3$, a different multiplication table would result.

Before moving to a study of electronic calculation in such fields and some important signal-processing results, we develop some fundamental notions associated with finite fields. First, a field of size q may contain a ***subfield*** of size r, a smaller field that obeys the field axioms. In Example 5.1, the set $\{0, 1\}$ forms a subfield of GF(8). It is known that a field of size $q = p^m$ contains a subfield of size $r = p^s$ if and only if s divides m [1, 2]. Obviously then, there is always a GF(p) subfield of GF(p^m). There is, however, no subfield of size 4 in GF(8). Exercise 5.1.6 considers the subfields of GF(16) and GF(256).

Now consider the field element 1 in a finite field and form the sequence $1, 1 + 1, 1 + 1 + 1, \ldots$. We must eventually find a sum equaling 0, and the smallest number of 1's that can be added to produce 0 is called the ***characteristic*** of the field. For GF(2), the characteristic is 2, and for GF(3), the characteristic is 3. For any field whose size is a prime number p, the characteristic is p. In general, for extension fields of size $q = p^m$, the characteristic is p, since two polynomials over GF(p), when added p times, will always produce the zero polynomial by virtue of addition properties in GF(p).

In a similar vein, consider the sequence of powers of a nonzero field element, β, that is, $\beta^1, \beta^2, \beta^3, \ldots$, which is a sequence of nonzero field elements. This sequence will eventually produce the field element 1 since the field is finite and thereafter be periodic. The ***order*** of an element β is the smallest nonzero n such that $\beta^n = 1$. (We also say that β is an nth root of unity when $\beta^n = 1$.) It is clear that the largest value for the order of an element is $q - 1$, since there are only $q - 1$ nonzero elements, but another result of number theory, due to Fermat, is that the order of every field element must divide $q - 1$. Thus, in GF(16) the nonzero field elements may have orders 1, 3, 5, or 15. In GF(256), the nonzero elements may have orders 1, 3, 5, 15, 17, 51, 85, and 255.

Every finite field with q elements has at least one ***primitive element***, an element α whose order is $q - 1$. Thus, the power sequence $\alpha^1, \alpha^2, \alpha^3, \ldots, \alpha^{q-1} = 1 = \alpha^0$ produces all nonzero field elements. Any nonprimitive element will have a power sequence that has shorter period and whose sequence does not contain the full set of nonzero elements in the field. If $q - 1$ is prime, it is clear from the previous paragraph that nonzero elements have order 1 or $q - 1$ and that all elements, other than 0 or 1, are primitive, which could be verified for GF(8).

Since each nonzero field element in a finite field can be expressed as a power of a primitive element α, we may associate with every element a logarithm to the base α. (By convention we take $-\infty$ as the logarithm of the element 0.) As with the field of real numbers, multiplication and division of elements can be performed by adding and subtracting logarithms, except that exponents must be combined modulo $q - 1$.

In the sequel we will need to identify these primitive field elements. The task of finding primitive elements is finessed if we use a ***primitive polynomial*** of degree m as a generating polynomial for the field. By definition, a primitive polynomial over GF(p) is one whose m roots are primitive elements in GF(p^m). We remark that being primitive is a stronger condition on a polynomial than being irreducible, since all the roots for

primitive polynomials must lie in the extension field GF(p^m). Primitive polynomials over GF(2) are tabulated in [1], for example, and are listed in Figure 5.1.2 for $m = 2$ through 10. These in turn supply fields of size 4, 8, 16, . . . , 1024. Enumeration of these fields is found in [3].

m	
2	$1 + D + D^2$
3	$1 + D + D^3$
4	$1 + D + D^4$
5	$1 + D^2 + D^5$
6	$1 + D + D^6$
7	$1 + D^3 + D^7$
8	$1 + D^2 + D^3 + D^4 + D^8$
9	$1 + D^4 + D^9$
10	$1 + D^3 + D^{10}$

Figure 5.1.2 Primitive polynomials over GF(2).

The existence of finite fields of size $q = p^m$ for any prime p and integer m is guaranteed by the fact that there is at least one irreducible polynomial over GF(p) of degree m [2]. The use of two different irreducible (or even primitive) polynomials would seemingly produce different fields. However, it can be shown that these fields are all equivalent to within a relabeling of elements, or are *isomorphic*, and there is really just one unique field for every $q = p^m$. In communication applications, the fields of primary interest are those extension fields of size $q = 2^m$.

An illuminating example of fields and the properties just discussed is the case of GF(16).

Example 5.2 Construction of GF(2^4) = GF(16)

We adopt as our polynomial for generating the field the primitive polynomial $f(D) = D^4 + D + 1$ and could proceed as before to build arithmetic tables. However, a more convenient construction is provided by letting 0 and 1 be the first two elements (the additive and multiplicative identities, respectively) and the next element be α, defined only as a root of the primitive polynomial; thus α is primitive in GF(16). We associate with this element the polynomial D. The remaining field elements are taken to be successive powers of α, for these certainly generate the remaining nonzero elements. Thus, α^2 is associated with the polynomial D^2, and α^3 is associated with D^3. Next, we encounter α^4 as a field element and use the fact that $\alpha^4 + \alpha + 1 = 0$, but since α and the element 1 are really just polynomials over GF(2), where subtraction is the same as addition, $\alpha^4 = \alpha + 1$. Thus, α^4 is associated with the polynomial $D + 1$. Continuing this procedure enumerates all the field elements as powers of α, or as polynomials. Figure 5.1.3 provides a listing for GF(16)

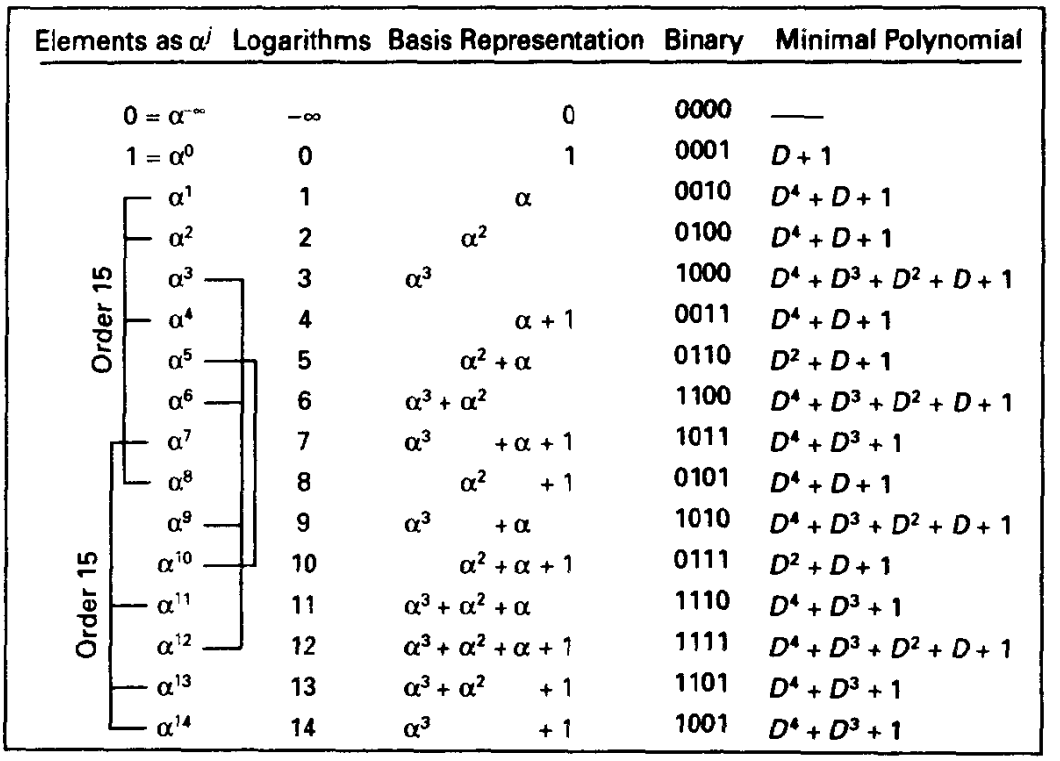

Figure 5.1.3 Representations of GF(16), where α is root of $f(D) = D^4 + D + 1$.

elements ordered as increasing powers of α, along with the corresponding polynomial forms. The nonzero elements in this field have orders 1, 3, 5, and 15, as could be determined by multiplication. There happen to be eight primitive elements in the field, α being one of them.

A final important topic in our survey of finite fields is that of ***minimal polynomials***. A minimal polynomial $m_\beta(D)$ for an element β in GF(p^m) is the minimum-degree monic polynomial over GF(p) having β as a root:

$$m_\beta(D)\big|_{D=\beta} = 0. \tag{5.1.5}$$

This minimum polynomial will always have degree m or less. In a finite field of size $q = p^m$, if β^r is a root of a polynomial, then so is β^{rp} [1]. Thus, the minimal polynomials for $\beta, \beta^2, \beta^4, \ldots$, in GF($2^m$) are identical, as are those for the elements $\beta^3, \beta^6, \beta^{12}, \ldots$. Elements sharing the same minimal polynomial are said to be ***conjugates***, just as j and $-j$ are conjugates in the complex number field, and for which $D^2 + 1$ is the minimal polynomial over the real numbers. A set of conjugate elements is known as a ***cyclotomic coset***, and every finite field can be decomposed into disjoint cosets. This will become relevant in our study of cyclic codes.

In Figure 5.1.3, we have listed the cyclotomic cosets for GF(16) according to exponents of α. Note that elements that are related by squaring, or exponent doubling, modulo 15, form a coset, since the field characteristic is $p = 2$.

The minimal polynomial for a given field element may be shown to be [2]

$$m_\beta(D) = \prod_{i \in S_\beta} (D - \alpha^i), \tag{5.1.6}$$

where α is primitive and S_β denotes the set of exponents in the cyclotomic coset for the element β. Thus, the minimal polynomial for the field element α^5 in GF(16) is $(D - \alpha^5)(D - \alpha^{10})$, which eventually simplifies to $D^2 + D + 1$ over the *binary* field. In Figure 5.1.3, the minimal polynomials so obtained are listed adjacent to the respective elements. Note, for example, that the elements designated $\alpha, \alpha^2, \alpha^4$, and α^8 have the same minimal polynomial, as do α^3, α^6, and so on. We also note that some field elements in Figure 5.1.3 have minimal polynomials of degree less than $m = 4$. From (5.1.6) this will occur when the size of that element's cyclotomic coset is less than m.

As a further example, Figure 5.1.4 lists the cyclotomic cosets of GF(64). It is worth noting that these cosets can be formulated without dependence on a particular irreducible polynomial, further testimony to the fact that there is really only one field of a given size.

Exponents of α	Minimal Polynomial
0	$D + 1$
1 2 4 8 16 32	$D^6 + D + 1$
3 6 12 24 48 33	$D^6 + D^4 + D^2 + D + 1$
5 10 20 40 17 34	$D^6 + D^5 + D^2 + D + 1$
7 14 28 56 49 35	$D^6 + D^3 + 1$
9 18 36	$D^3 + D^2 + 1$
11 22 44 25 50 37	$D^6 + D^5 + D^3 + D^2 + 1$
13 26 52 41 19 38	$D^6 + D^4 + D^3 + D + 1$
15 30 60 57 51 39	$D^6 + D^5 + D^4 + D^2 + 1$
21 42	$D^2 + D + 1$
23 46 29 58 53 43	$D^6 + D^5 + D^4 + D + 1$
31 62 61 59 55 47	$D^6 + D^5 + 1$
45 27 54	$D^3 + D + 1$

Figure 5.1.4 Conjugate sets and minimal polynomials for GF(64); primitive polynomial $D^6 + D + 1$ used to define α.

5.1.2 Computation in Finite Fields

For implementation of GF(q) arithmetic, table lookup for the sum and product of field elements is a possibility if q is not too large. When $q = 2^m$, the operands can be identified with m-bit representations, and a table could be constructed with $2m$-bit addresses and m-bit outputs. Logarithm tables could be used for multiplication as well. Equivalently,

combinational logic circuits could implement the truth table defined by the addition and multiplication tables.

These approaches rapidly become unmanageable for larger fields. We may perform the required computations easily, however, with electronic circuits that implement polynomial operations over the ground field. Addition of two field elements is easily handled by adding their corresponding coefficient vectors, which for fields of size $q = 2^m$ requires m exclusive-OR gates [GF(2) adders].[4]

To perform multiplication of a general field element β by another element, say γ, we represent field elements using $\alpha^0, \alpha^1, \ldots, \alpha^{m-1}$ as a basis; that is, $\beta = \beta_0 + \beta_1\alpha^1 + \beta_2\alpha^2 + \cdots + \beta_{m-1}\alpha^{m-1}$, where the coefficients β_i are in the ground field. Thus, in the previous example the element $\alpha^{11} = \alpha^3 + \alpha^2 + \alpha$. Expressing both multiplicands in this manner, we have

$$\begin{aligned} \beta &= \beta_0 + \beta_1\alpha^1 + \beta_2\alpha^2 + \cdots + \beta_{m-1}\alpha^{m-1}, \\ \gamma &= \gamma_0 + \gamma_1\alpha^1 + \gamma_2\alpha^2 + \cdots + \gamma_{m-1}\alpha^{m-1}. \end{aligned} \tag{5.1.7}$$

If we perform longhand multiplication of these two elements and simplify, we find that we can determine a Boolean relation giving the m output bits in terms of the $2m$ input operands. This process is illustrated in Figure 5.1.5 for multiplying two elements in

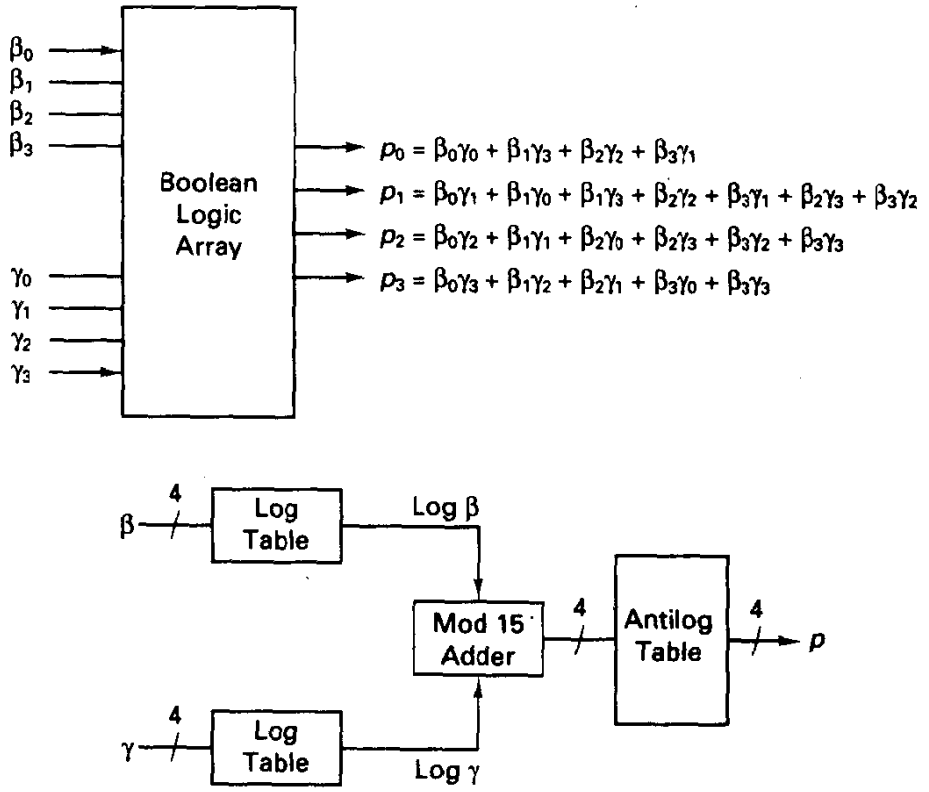

Figure 5.1.5 Galois field multipliers for GF(16).

[4]Those with a computer engineering background may be tempted to "carry" in doing this addition, but it is not allowed.

GF(16). If we need a multiplier to continually multiply by a *fixed* operand, the circuit can be simplied even further. Modern gate-array technology is well suited for performing such operations. Numerous design tricks exist for minimizing the complexity of such operations, but the point is that arithmetic in GF(q) is not difficult; in fact, it is simpler than the usual integer-field arithmetic, and there is no concern about carries or loss of precision as occurs in real-number arithmetic on a digital machine.

Another operation of frequent need is the "multiply and accumulate with previous sum" operation. Such a circuit has one q-ary field element as an input and one field element as output and may be easily realized as an extension of Figure 5.1.5. (See Exercise 5.1.7.)

5.1.3 Discrete Fourier Transforms over Finite Fields

In signal processing of real (or complex) discrete-time sequences, the discrete Fourier transform (DFT) plays a central role, familiar to present-day engineers. This situation derives from the well-known analytical and operational appeal of Fourier transforms in linear system analysis, as well as the existence of a fast algorithm for computing the DFT. It is less well known that this transform calculus can pertain to other fields, just as the standard operations of linear algebra remain valid over fields other than the real or complex numbers. These generalized Fourier transforms play an important descriptive role in the study of cyclic codes later in this chapter and to an increasing degree in the implementation of these codes.

We first recall the familiar: the usual definition of the DFT for a sequence of complex numbers $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$ is

$$X_i = \sum_{k=0}^{N-1} x_k e^{-j2\pi ki/N}, \qquad i = 0, 1, \ldots, N-1, \tag{5.1.8}$$

and the inverse DFT is

$$x_k = \frac{1}{N} \sum_{i=0}^{N-1} X_i e^{j2\pi ik/N}, \qquad k = 0, 1, 2, \ldots, N-1. \tag{5.1.9}$$

The kernel $e^{-j2\pi/N}$ is a primitive Nth root of unity in the complex number field, that is, the smallest k for which $e^{-(j2\pi/N)k} = 1$ is $k = N$, and the existence of such an element is all that is required for the existence of a more general transform pair and its associated operational properties, as inspection of the development of these properties will reveal. The DFT of a complex sequence and its inverse transform are useful for any N precisely because in the complex field we always have a primitive Nth root of unity, that is, $e^{-j2\pi/N}$. (The use of so-called fast Fourier transforms only applies if N is a power of a prime or highly composite, however.)

To generalize, consider a sequence of length N, $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$, from GF($q$). Suppose there exists a primitive Nth root of unity, α, in an extension field $GF(q^m)$; that is, N is the smallest integer for which $\alpha^N = 1$. Then we define the ***discrete Fourier transform*** of the sequence $\mathbf{x}$ to be the sequence

$$X_i = \sum_{k=0}^{N-1} x_k \alpha^{ik}, \qquad i = 0, 1, \ldots, N-1. \tag{5.1.10}$$

The operation (5.1.10) transforms a sequence $\mathbf{x}$ into another sequence $\mathbf{X}$ of length N whose elements are contained in the extension field GF(2^m). In analogy with the usual application of transforming a sequence of time-domain samples into the frequency-domain representation, we will refer to the sequences $\mathbf{x}$ and $\mathbf{X}$ as being time- and frequency-domain descriptions, respectively, although this interpretation is less motivated in the present case.

Note that the addition and multiplication in (5.1.10) are well defined since the original sequence elements are in a subfield of GF(q^m). If the desired block length N equals $q - 1$ or is a factor of $q - 1$, then an extension field is not invoked and the transform takes the original sequence into a sequence from the same field. Normally, however, we will be interested in a longer block length N, which is a factor of $q^m - 1$ so that the requisite primitive Nth root of unity exists. [Recall again that the order of all nonzero field elements in GF(q^m) is a divisor of $q^m - 1$.]

The generalized ***inverse DFT*** is defined as

$$x_k = \frac{1}{N'} \sum_{i=0}^{N-1} X_i \alpha^{-ik}, \tag{5.1.11a}$$

where N' is a normalizing factor representing the N-fold sum of 1 in the field F:

$$N' = \sum_{i=0}^{N-1} 1. \tag{5.1.11b}$$

Of course, for real or complex field transforms, $N' = N$, hence the scaling in (5.1.9). For transforms over GF(2) or an extension field with characteristic 2, $N' = 1$ since N' will always be the addition of an odd number of 1's. Thus, the normalization may typically be omitted.

The fact that (5.1.10) and (5.1.11) form a transform pair follows from substitution of (5.1.10) into (5.1.11):

$$x_k = \frac{1}{N'} \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N-1} x_j \alpha^{ij} \right) \alpha^{-ik} = \frac{1}{N'} \sum_{j=0}^{N-1} x_j \sum_{i=0}^{N-1} \alpha^{(j-k)i}. \tag{5.1.12}$$

Now we invoke an important property associated with Nth roots of unity:

$$\sum_{i=0}^{N-1} \alpha^{mi} = \begin{cases} N', & \text{if } N \text{ divides } m, \\ 0, & \text{else.} \end{cases} \tag{5.1.13}$$

(Verification of this result is left to the exercises.) We apply this result to the last sum in (5.1.12), noting that $0 \le i \le N - 1$ and $0 \le j \le N - 1$. Thus, $j - k$ is divisible by N; that is, $(j - k) \bmod N = 0$ only if $k = j$. The right-hand side of (5.1.12) thus becomes $(1/N')(x_k N')$, yielding the desired equality.

Just as the inverse transform of an arbitrary complex sequence may not produce a real sequence with the standard DFT,[5] we will find that an inverse transform of a sequence in GF(q^m) may not produce a sequence in the ground field GF(q). This will be encountered in our study of decoding of BCH codes.

[5] It will only if $X_{N-i} = X_i^*, i = 0, 1, \ldots, N - 1$, where * denotes conjugation.

Before developing some of the key properties of this general Fourier transform, which will in hindsight be familiar to anyone exposed to the usual DFT, we consider two examples.

Example 5.3 DFT over GF(5)

In GF(5), the field elements are isomorphic to the set of integers $\{0, 1, 2, 3, 4\}$ and have order 1, 2, or 4. Thus, the only transform lengths that take GF(5) sequences into GF(5) sequences are 2 and 4. Consider $N = 4$, and suppose that $\mathbf{x} = (1, 0, 3, 4)$. Take $\alpha = 2$, a fourth root of unity, to define the transform. Then, by substitution in (5.1.10), we can readily show with modulo-5 arithmetic that $\mathbf{X} = (3, 0, 0, 1)$ is the DFT of this sequence. We could also verify that (5.1.11) produces the original time-domain sequence. In this field, N' is 4.

If we desired longer transform block lengths, we could invoke use of a kernel that is a primitive Nth root of unity in an extension field, say GF(25). In this case we could perform transforms of length 2, 3, 4, 6, 8, 12, and 24, all of which factor $5^2 - 1$. Zero padding can be used to handle transforms of other lengths.

Example 5.4 DFT over GF(2)

By appealing to the extension field GF(16), the possible DFT sizes for binary sequences are $N = 3$, 5, and 15, since these are the possible orders (other than 1) of the field elements in GF(16). Consider a transform of the length-15 binary (subfield) sequence (100001000000000), with the lowest-order element written on the left. If we take as our primitive 15th root of unity the field element designated α in Figure 5.1.3, then simple computation will show that the DFT is the periodic sequence $\mathbf{X} = (0, \alpha^{10}, \alpha^5, 0, \alpha^{10}, \alpha^5, \ldots, 0, \alpha^{10}, \alpha^5)$. To perform the inverse transform, note that N' is 1, as will be the case whenever we work in an extension field of GF(2), since N' is obtained by adding an odd number of 1's. Thus, for example, $x_0 = \sum X_i = 1$, which is the desired element in the subfield GF(2).

In our development of cyclic codes it will be convenient to utilize a polynomial representation of symbol sequences, and this proves useful in the description of DFT properties as well. We associate with a sequence $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$ over GF(q) the polynomial $x(D) = x_0 + x_1 D + x_2 D^2 + \cdots + x_{N-1} D^{N-1}$, which has degree at most $N - 1$. For the present the indeterminate D is merely a place keeper. Letting $\mathbf{X}$ denote the transform sequence for $\mathbf{x}$, we have

$$X_i = \sum_{k=0}^{N-1} x_k \alpha^{ik} = \sum_{k=0}^{N-1} x_k D^k \Big|_{D=\alpha^i}, \qquad i = 0, 1, \ldots, N-1. \tag{5.1.14}$$

Thus the spectral coefficients X_i are merely the values of the polynomial $x(D)$ evaluated at α^i, $i = 0, 1, \ldots, N - 1$, where α is a primitive Nth root of unity in GF(q^m). This is true for the usual DFT as well.

Similarly, it follows from the definition of the inverse transform (5.1.11) that

$$x_k = \frac{1}{N'} X(D)|_{D=\alpha^{-k}}. \tag{5.1.15}$$

In other words, evaluation of transform coefficients in either domain is equivalent to polynomial evaluation at designated powers of the Nth root of unity.

Properties of the DFT

The foremost property is ***linearity*** of the transform operators, as may be shown by applying the superposition test to both transforms. Thus, if $\mathbf{x}_1$ corresponds with $\mathbf{X}_1$, and likewise for $\mathbf{x}_2$ and $\mathbf{X}_2$, then the transform of $a_1\mathbf{x}_1 + a_2\mathbf{x}_2$ is $a_1\mathbf{X}_1 + a_2\mathbf{X}_2$, where a_1 and a_2 are any constants in the field of the original sequence.

Another important property is the ***cyclic-shift property***, soon to be of special interest in our discussion of cyclic codes. Again, let $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$, and let $x^{(p)}$ denote a p-place left-cyclic shift of $\mathbf{x}$:

$$\mathbf{x}^{(p)} = (x_p, x_{p+1}, \ldots, x_{N-1}, x_0, \ldots, x_{p-1}). \tag{5.1.16}$$

The discrete Fourier transform of this rotated sequence is, from the definition (5.1.10),

$$\begin{aligned} X_i^{(p)} &= \sum_{k=0}^{N-1} x_k^{(p)} \alpha^{ik}, \qquad i = 0, 1, \ldots, N-1 \\ &= x_p\alpha^{0\cdot i} + x_{p+1}\alpha^{1\cdot i} + \cdots + x_{p-1}\alpha^{(N-1)\cdot i} \\ &= x^{(p)}(D)|_{D=\alpha^i}, \end{aligned} \tag{5.1.17}$$

as before. These transform coefficients may be easily related to the transform coefficients of the original sequence. By factoring D^{-p} from each term of the polynomial form of the $x^{(p)}$ sequence, we have that

$$\begin{aligned} x^{(p)}(D) = D^{-p}(&x_pD^p + x_{p+1}D^{p+1} + \cdots + x_{N-1}D^{N-1} \\ &+ x_0D^N + \cdots + x_{p-1}D^{N-1+p}). \end{aligned} \tag{5.1.18}$$

By adding and subtracting to the right-hand side $\tilde{x}(D) = x_0 + x_1D + \cdots + x_{p-1}D^{p-1}$ and then regrouping, we obtain

$$x^{(p)}(D) = D^{-p}x(D) + (D^N - 1)\tilde{x}(D). \tag{5.1.19}$$

Now we recall that the transform coefficients $X_i^{(p)}$ are merely the values of the polynomial $x^{(p)}(D)$ evaluated at α^i, and since $(\alpha^i)^N = 1$, we find

$$X_i^{(p)} = \alpha^{-ip}X_i. \tag{5.1.20}$$

Thus, to obtain the ith "frequency" term in the DFT of a *left-cyclic rotation* of a sequence, we merely multiply the original transform coefficients X_i by α^{-ip}, where p is the number of places shifted. This is clearly a generalization of the Fourier transform principle that a cyclic shift of a sequence corresponds to a phase shift in the frequency domain. If we were to repeat the preceding by shifting to the right by p places, the former transform coefficients would be modified by α^{ip}.

The transform sequence, if extended beyond its usual range of definition, is ***periodic*** with period N. That is,

$$X_{i+N} = X_i, \qquad i = 0, 1, \ldots, N-1. \tag{5.1.21}$$

This is easily proved by noting that

$$X_{i+N} = \sum_k x_k\alpha^{(i+N)k} = \sum_k x_k\alpha^{ik}(\alpha^N)^k = \sum x_k\alpha^{ik} = X_i \tag{5.1.22}$$

since $\alpha^N = 1$ by construction. Likewise, given frequency-domain coefficients λ inverse transform sequence is periodic: $x_{k+N} = x_k, k = 0, 1, \ldots, N-1$.

Finally, the ***convolution theorem*** for the generalized DFT holds that if a sequence $\mathbf{x}$ is obtained as the *cyclic* (or circular) convolution of sequences $\mathbf{a}$ and $\mathbf{b}$, that is,

$$x_k = \sum_{i=0}^{N-1} a_i b_{k-i}, \qquad k = 0, 1, \ldots, N-1, \tag{5.1.23}$$

where the summand subscripts are interpreted modulo N, then the transform of $\mathbf{x}$ is obtained by multiplying the transforms of $\mathbf{a}$ and $\mathbf{b}$:

$$X_i = A_i B_i, \qquad i = 0, 1, \ldots, N-1. \tag{5.1.24}$$

All other properties that hold for the standard DFT, such as Parseval's relation and the effect of scaling in either domain, can be proved easily from the definition. Again, all that is crucial is using a transform kernel that is a primitive Nth root of unity in an appropriate extension field.

5.2 LINEAR BLOCK CODES

We are now ready to formalize the description of linear codes over general finite field alphabets. The linear algebraic structure provides significant reduction in encoding and decoding complexity, relative to that for arbitrary block codes. It might be asked whether restricting our focus to linear codes is ultimately harmful in the information-theoretic sense. The earlier random coding proof of Chapter 4 pertained to general block codes, but it has been shown that the ensemble of linear codes is strong in the same sense [7]; that is, there is a sequence of linear codes with increasing block length and fixed rate just smaller than capacity whose error probability approaches zero exponentially as block length increases.

5.2.1 Structure of Linear Codes over GF(q)

It is fairly traditional, especially in engineering texts, to first introduce binary coding procedures and then (perhaps) generalize to the case of codes over nonbinary fields. There is no significant conceptual problem in handling the general case at the outset, and, given our study of finite fields in the previous section, the tools are now in hand.

Let $\mathbf{u} = (u_0, u_1, \ldots, u_{k-1})$ denote an arbitrary message k-tuple from GF(q). A ***linear*** (n, k) ***code*** C ***over*** GF(q) is a set of q^k codewords $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$, $x_j \in$ GF(q), defined by the linear transformation

$$\mathbf{x} = \mathbf{u}\mathbf{G}, \tag{5.2.1}$$

where $\mathbf{G}$ is a $k \times n$ matrix comprised of elements from GF(q). Having just seen how to add and multiply in finite fields in the previous section, we perform matrix multiplication along conventional lines. Figure 5.2.1 illustrates the concept of the encoding process and the process of building each codeword element in terms of adders and multipliers over GF(q).

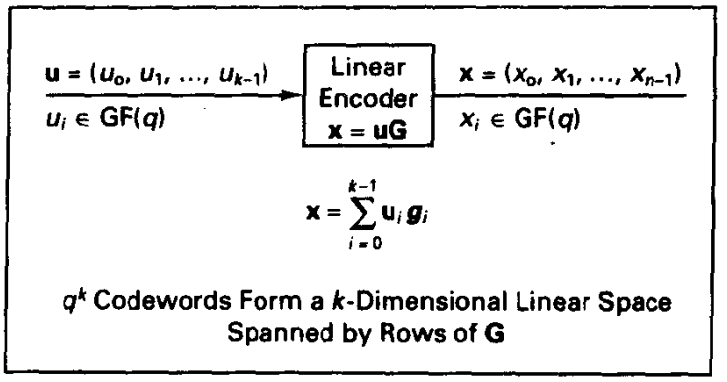

Figure 5.2.1 Linear (n, k) encoder over GF(q).

G is called the ***generator matrix*** of the code, for (5.2.1) implies that the code C is comprised of all linear combinations of rows of G:

$$\mathbf{x} = u_0\mathbf{g}_0 + u_1\mathbf{g}_1 + \cdots + u_{k-1}\mathbf{g}_{k-1}, \tag{5.2.2}$$

where $\mathbf{g}_j$ is the jth row vector of G. Provided these rows are linearly independent, that is, **G** has rank k, the code has q^k distinct codewords, which, of course, is desired for signaling of q^k messages. In vector space terminology, we say that the code C is a k-dimensional subspace of the set of all n-tuples, this subspace being spanned by k linearly independent basis vectors $\mathbf{g}_j$.[6] The choice of basis vectors is not unique, and thus the same set of codewords can be formed with different generator matrices **G**, changing only the association between messages and codewords. We will return to this notion shortly.

Let us reconsider our earlier example, the (7, 4) code over the binary field. By studying the encoding equations (5.0.3) and by agreeing to place the information bits in the leading positions of the code vector, followed by the three parity bits, we have the encoding equations

$$\begin{aligned}
x_0 &= u_0 = i_1, \\
x_1 &= u_1 = i_2, \\
x_2 &= u_2 = i_3, \\
x_3 &= u_3 = i_4, \\
x_4 &= u_0 + u_1 + u_2 = p_1 \quad \text{or} \quad x_4 + x_0 + x_1 + x_2 = 0, \\
x_5 &= u_1 + u_2 + u_3 = p_2 \quad \text{or} \quad x_5 + x_1 + x_2 + x_3 = 0, \\
x_6 &= u_0 + u_1 + u_3 = p_3 \quad \text{or} \quad x_6 + x_0 + x_1 + x_3 = 0.
\end{aligned} \tag{5.2.3}$$

Thus, **G** in (5.2.1) becomes

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{5.2.4}$$

[6] This provides an alternative definition of an (n, k) linear code.

so for $\mathbf{u} = (1010)$, $\mathbf{x} = (1010011)$. There are 16 distinct linear combinations of the rows of G, as listed in Figure 5.2.2.

u	x
0000	0000000
0001	0001011
0010	0010110
0011	0011101
0100	0100111
0101	0101100
0110	0110001
0111	0111010
1000	1000101
1001	1001110
1010	1010011
1011	1011000
1100	1100010
1101	1101001
1110	1110100
1111	1111111

$$\mathbf{G} = \begin{bmatrix} 1000101 \\ 0100111 \\ 0010110 \\ 0001011 \end{bmatrix}$$

Figure 5.2.2 Listing of codewords for (7, 4) code.

To generalize to a nonbinary example, we define a simple (3, 2) code over GF(4), with symbols labeled 0, 1, α, and $\beta = \alpha^2$. Following the earlier rules for the construction of a field, we have arithmetic tables in Table 5.1.

TABLE 5.1
ARITHMETIC
TABLES FOR GF(4)

Addition				
+	0	1	α	β
0	0	1	α	β
1	1	0	β	α
α	α	β	0	1
β	β	α	1	0

Multiplication				
*	0	1	α	β
0	0	0	0	0
1	0	1	α	β
α	0	α	β	1
β	0	β	1	α

Now, if we adopt

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & \alpha \\ \alpha & \beta & 1 \end{bmatrix}, \tag{5.2.5}$$

then the codeword for the message $\mathbf{u} = (0, \alpha)$ is $\mathbf{x} = (\beta, 1, \alpha)$, which is just α times the second row of $\mathbf{G}$, in GF(4). This code contains $4^2 = 16$ codewords of length 3.

Inspection of the generator matrix in (5.2.4) shows it has the canonical form

$$\mathbf{G} = [\mathbf{I}_{k,k}\ \mathbf{P}_{k,n-k}], \tag{5.2.6}$$

where $\mathbf{I}_{k,k}$ is the k by k identity matrix and $\mathbf{P}_{k,n-k}$ is a k by $(n-k)$ matrix reflecting how the $n-k$ additional code symbols are formed from the components of $\mathbf{u}$. Equation (5.2.6) implies that the first k components of any codeword are precisely the information symbols in original order. This form of linear encoding is called ***systematic*** encoding, and several implementation advantages accrue from the use of systematic-form codes, not the least of which is a quick-look feature: information can be extracted trivially from the codeword, if desired, without any decoding. Naturally, these decisions will be of lesser reliability than when the full code vector is used to infer the message content.

For linear codes, any code is equivalent to a code in systematic form, equivalent in the sense of having the same block error performance on a memoryless channel. This follows from noting that a linear code is the row space, or set of linear combinations, of the rows of the $\mathbf{G}$ matrix. If we interchange two rows of $\mathbf{G}$, or multiply any row by a scalar in GF(q), or add one row to another, we do not change the row space, hence the set of codewords. All we have done is change the basis for the code, which amounts to changing the association between messages and code vectors. By a succession of these elementary row operations on a given $\mathbf{G}$, we may bring it into the canonical form of (5.2.6). (An additional operation that may be required in this reduction is the interchanging of columns of $\mathbf{G}$, or merely reordering the coordinates. Clearly, this also leaves the fundamental properties of the code unchanged, but alters the row space.)

Example 5.5 Converting the (3, 2) Code over GF(4) to Systematic Form

Let's recall the previous code over GF(4), with

$$G = \begin{bmatrix} 1 & 0 & \alpha \\ \alpha & \beta & 1 \end{bmatrix}. \tag{5.2.7a}$$

Multiplying the first row by α and adding it to the second, we have

$$G_1 = \begin{bmatrix} 1 & 0 & \alpha \\ 0 & \beta & \alpha \end{bmatrix}. \tag{5.2.7b}$$

Then multiplying the second row by the reciprocal of β we obtain

$$G_1 = \begin{bmatrix} 1 & 0 & \alpha \\ 0 & 1 & \beta \end{bmatrix}, \tag{5.2.7c}$$

which is in the desired systematic form. We could verify that all three corresponding sets of codewords are identical, although the association with messages changes.

Since every linear code is equivalent to a systematic code, it is sufficient to focus on systematic form generator matrices in our study. (This is oddly not true of linear trellis codes, which we examine in Chapter 6, traceable to the fact that in a trellis code

it is useful to decode with a memory length longer than the encoder's memory length.) In systematic form, the codeword $\mathbf{x}$ is comprised of an information segment and a set of $n-k$ symbols that are linear combinations of certain information symbols, as determined by the $\mathbf{P}$ matrix. These additional symbols are called ***parity check symbols***, in keeping with the terminology for familiar single-parity-bit codes that append a single parity bit to a binary k-tuple to make the sum of symbols in the codeword be zero. (This is referred to as forcing even parity.) For this reason, linear codes are sometimes referred to as ***parity check codes***.

We have seen that the $\mathbf{G}$ matrix applies linear constraints to the code symbols. To gain another important perspective on this same set of constraints, we invoke another concept of linear algebra. Any k-dimensional linear subspace of an n-dimensional linear space, for example, a code C, has associated with it a ***null space***, which we shall designate C^+, such that every vector in the null space is orthogonal to every vector in the original space. Denoting $\mathbf{x}_i$ and $\mathbf{y}_j$ as members of a linear subspace and its null space, respectively, this means the vector inner product

$$\mathbf{x}_i \mathbf{y}_j^T = 0 \tag{5.2.8}$$

for all $\mathbf{x}_i \in C$ and $\mathbf{y}_j \in C^+$. [In (5.2.8) the superscript T denotes the transpose operation.] This null space has dimension $n-k$ and is spanned by a set of $n-k$ linearly independent vectors of length n. In matrix form we then have that, for any element $\mathbf{y}$ in the null space,

$$\mathbf{y} = \mathbf{vH} \tag{5.2.9}$$

for some $\mathbf{v}$, where $\mathbf{v}$ is an $(n-k)$ row vector and $\mathbf{H}$ has size $(n-k)$ by n, both with entries in GF(q). Thus, for any codeword $\mathbf{x}_i$ in the original space C, we require

$$\mathbf{x}_i(\mathbf{vH})^T = \mathbf{x}_i \mathbf{H}^T \mathbf{v}^T = 0, \qquad \text{all } \mathbf{v}. \tag{5.2.10a}$$

This can only be possible if

$$\mathbf{x}_i \mathbf{H}^T = \mathbf{0}, \qquad \text{all } \mathbf{x}_i, \tag{5.2.10b}$$

where $\mathbf{0}$ represents a vector with zero elements.

Equation (5.2.10b) constitutes a set of $n-k$ linear equations that the codeword must satisfy, which are called the parity check equations of the code. $\mathbf{H}$ is therefore denoted the ***parity check matrix***, because it describes the total set of parity check constraints. If we write out the $n-k$ equations produced by (5.2.10b), we obtain

$$\sum_{j=0}^{n-1} x_j h_{ij} = 0, \qquad i = 0, 1, \ldots, n-k-1. \tag{5.2.10c}$$

For example, the last three equations in (5.2.3) provide the parity check equations for the binary Hamming (7, 4) code.

Since (5.2.10b) must hold for any codeword in the space spanned by the rows of $\mathbf{G}$, we have that $\mathbf{G}$ and $\mathbf{H}$ must satisfy

$$\mathbf{G}\mathbf{H}^T = [0]_{k,n-k}, \tag{5.2.11}$$

where the right-hand side is a k by $n-k$ matrix of zeros. For any given $\mathbf{G}$ matrix, many solutions for $\mathbf{H}$ are possible.

If the generator matrix $\mathbf{G}$ *is in systematic form, then finding a parity check matrix* $\mathbf{H}$ is simple. Taking

$$\mathbf{H} = [-\mathbf{P}^T_{k,n-k}\ \mathbf{I}_{n-k,n-k}] \tag{5.2.12}$$

satisfies (5.2.11), as substitution will verify.[7]

Since $\mathbf{G}$ specifies $\mathbf{H}$ (even in the nonsystematic case where $\mathbf{H}$ is not so directly found), a completely equivalent definition of a linear code is that C is the set of all codewords in the null space of an $n-k$ by n matrix $\mathbf{H}$, whose rank is $n-k$, that is, the set of vectors $\mathbf{x}_i$ for which

$$\mathbf{x}_i\mathbf{H}^T = \mathbf{0}. \tag{5.2.13}$$

Furthermore, if $\mathbf{G}$ generates an (n, k) linear code C with a null space generated by $\mathbf{H}$, then $\mathbf{H}$ generates an $(n, n-k)$ linear code C^+, with null space generated by $\mathbf{G}$. These two codes are called ***dual codes***, with the generator matrix of one being the parity check matrix of the other. The structure of one code tells us a great deal about the other. If the code and its dual code are equivalent, we say that the code is ***self-dual***. Such codes have $n-k=k$ or $k=n/2$ and therefore have rate $R=\frac{1}{2}$.

Example 5.6 (7, 4) Binary Code Revisited

Recalling the generator matrix for the (7, 4) code, (5.2.4), we can immediately write the parity check matrix for the code, using (5.2.12), as

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}. \tag{5.2.14}$$

[Recall again that in GF(2) addition and subtraction are equivalent operations.] This $\mathbf{H}$ matrix also generates a (7, 3) linear code that is a dual code of the (7, 4) code, and every codeword in this code is orthogonal to every codeword in the Hamming code.

Example 5.7 $(n, 1)$ Repetition Code

A repetition code is a rather trivial code produced by $\mathbf{G} = [1, 1, 1, \ldots, 1]$, so the code is comprised of q vectors whose components are n transmissions of the same symbol, and the code rate is $R = 1/n$. The code has minimum Hamming distance n, since vectors differ in all n positions. The parity check matrix for this code is

$$\mathbf{H} = [\mathbf{1}, \mathbf{I}], \tag{5.2.15}$$

where $\mathbf{1}$ is an $(n-1)$ by 1 column vector of 1's and $\mathbf{I}$ is an $(n-1)$ by $(n-1)$ identity matrix. This $\mathbf{H}$ matrix, in turn, generates an $(n, n-1)$ code that appends a single parity symbol to $n-1$ information symbols, and this dual code has minimum Hamming distance of 2.

5.2.2 Distance Properties of Linear Codes and Error Protection Properties

We have seen that linear codes have algebraic structure especially useful to their description, as well as implementation of encoding. We now turn toward the performance

[7] Use the matrix identity $[\mathbf{A}\ \ \mathbf{B}] \cdot \begin{bmatrix} \mathbf{C} \\ \mathbf{D} \end{bmatrix} = \mathbf{AC} + \mathbf{BD}$, where dimensional consistency is assumed.

analysis, focusing on the Hamming distance structure of a linear code and guarantees we can make for error control.

Especially important to the analysis is the fact that the sum of any two codewords in a linear code is itself a codeword. (This, after all, is the definition of linearity for the encoding operator). Thus, the codeword $\mathbf{x}_3$ formed by adding $\mathbf{x}_1$ and $\mathbf{x}_2$ is

$$\mathbf{x}_3 = \mathbf{x}_1 + \mathbf{x}_2 = \mathbf{u}_1\mathbf{G} + \mathbf{u}_2\mathbf{G} = (\mathbf{u}_1 + \mathbf{u}_2)\mathbf{G} = \mathbf{u}_3\mathbf{G}, \tag{5.2.16}$$

so $\mathbf{x}_3$ is the codeword for the message $\mathbf{u}_1 + \mathbf{u}_2$, which is just another message k-tuple $\mathbf{u}_3$. Also, if $\mathbf{x}_2$ is a codeword, so is $-\mathbf{x}_2$, corresponding to the message $-\mathbf{u}_2$, the vector formed by the additive inverses of the elements in $\mathbf{u}_2$. Thus, $\mathbf{x}_1 - \mathbf{x}_2$ is also a valid codeword. Furthermore, the all-zeros vector is a member of every linear code, corresponding to use of the all-zeros message vector in (5.2.1).

The fundamental importance of the intercodeword Hamming distances has been seen in Chapter 4 and in the study of the (7, 4) code. For general block codes, the set of Hamming distances $\{d_{ij}\}$ between a codeword $\mathbf{x}_i$ and all other words $\mathbf{x}_j$ depends on choice of the reference vector $\mathbf{x}_i$. For linear codes, however, every codeword has an *identical* set of distances to other codewords. To see why, we consider the Hamming distance between any two codewords:

$$\begin{aligned} d(\mathbf{x}_i, \mathbf{x}_j) &= \text{number of places where } x_{i_m} \neq x_{j_m}, \qquad m = 0, 1, \ldots, n-1 \\ &= wt(\mathbf{x}_i - \mathbf{x}_j) = wt(\mathbf{x}_h), \end{aligned} \tag{5.2.17}$$

where $wt(\mathbf{z})$ denotes the ***Hamming weight*** of $\mathbf{z}$, or the number of nonzero positions, and $\mathbf{x}_h = \mathbf{x}_i - \mathbf{x}_j$ is the difference between the original code vectors, which is also a codeword as just demonstrated.

The important result is that the set of Hamming distances between distinct codewords is the same as the set of weights of the nonzero codewords in the code and is invariant to choice of reference vector. [The reader may want to verify this invariance for the (7, 4) code already tabulated.] This invariance is quite profound, meaning, first, that to find the distance structure of the code we only need examine the weight structure of the q^k codewords, rather than the distance between all q^{2k} pairs and, second, that any performance analysis for channels that are uniform from the input (UFI) can be performed by focusing on transmission of any one codeword. A convenient choice is the all-zeros vector, always a vector in a linear code.

By listing the various Hamming weights, w, of the codewords, and the number of codewords, A_w, at each weight, we obtain the ***weight spectrum*** of the code. This information may be compactly summarized in the form of a polynomial, called the ***weight enumerator polynomial***:

$$A(z) = \sum_{w=0}^{n} A_w z^w. \tag{5.2.18}$$

Clearly, $A_0 = 1$, and $A_w = 0$ for $0 < w < d_{\min}$.

For general linear codes, tabulating the complete weight spectrum requires evaluating all sums of the rows of $\mathbf{G}$, certainly tedious business best left to a computer; even this soon becomes impractical as code size grows. However, for some important codes the weight enumerator is known in analytic form. We will say more about this as we

develop specific codes, but for the (7, 4) code it is easy to count in Figure 5.2.2 that there is one word of weight 0 (the all-zeros vector), seven words of weight 3, seven of weight 4, and one of weight 7. The weight enumerator polynomial for this code is then

$$A(z) = \sum_{w=0}^{7} A_w z^w = 1 + 7z^3 + 7z^4 + z^7. \tag{5.2.19}$$

To illustrate another simple binary code with not so special properties as the (7, 4) code, we consider the (6, 3) binary code obtained by deleting the first row and column from the generator matrix of (5.2.4). [This is called shortening the code, as described in Section 5.6, and amounts to adoption of a subset of codewords in the (7, 4) code, having 0 in the first message position, and then not transmitting this fixed code coordinate.] The generator matrix for this code is

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{5.2.20}$$

This rate $\frac{1}{2}$ code has $2^3 = 8$ codewords, of which one has weight 0, four have weight 3, and three have weight 4.

The determination of the weight spectrum is often simplified by analyzing the dual code and then applying relations due to MacWilliams [8], which relate the weight spectrum of a code to that of its dual code. Specifically, let A_w and B_w denote the numbers of codewords of weight w in an (n, k) code C and its $(n, n-k)$ dual code C^+, respectively. The MacWilliams relations, which we merely state, are a set of n linear equations:

$$\sum_{i=0}^{n-m} B_i C_m^{n-i} = q^{n-k-m} \sum_{j=0}^{n} A_j C_{n-m}^{n-j}, \qquad m = n, n-1, \ldots, 0. \tag{5.2.21}$$

Of course, for linear codes $A_0 = B_0 = 1$. The MacWilliams identity is particularly useful in analyzing very large codes, whose dual codes are, however, small enough to be enumerated directly. Short of this, a simple binomial approximation [6] to the weight spectrum is discussed in Exercises 5.2.5 and 5.2.6.

Certainly, the most important aspect of the distance structure of a code is the ***minimum Hamming distance*** between any two codewords, denoted $d_{\min}$. From our recent discussion, this quantity is exactly the minimum nonzero Hamming weight of all code vectors in a linear code. To grasp the principal importance of $d_{\min}$, consider the use of q-ary codes on a q-ary uniform channel, described in Chapter 2. The action of the channel can be expressed as the addition of an error sequence $\mathbf{e}$ to the transmitted sequence $\mathbf{x}_i$. Here, maximum likelihood decoding corresponds to finding the codeword $\mathbf{x}_i$ that exhibits fewest discrepancies with the channel output sequence $\mathbf{r}$, or which is closest in Hamming distance to $\mathbf{r}$. Suppose that $\mathbf{x}_i$ is selected for transmission and the closest codeword is $d_{\min}$ Hamming units distant, as shown schematically in Figure 5.2.3a. If the channel error pattern $\mathbf{e}$ has

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor \tag{5.2.22}$$

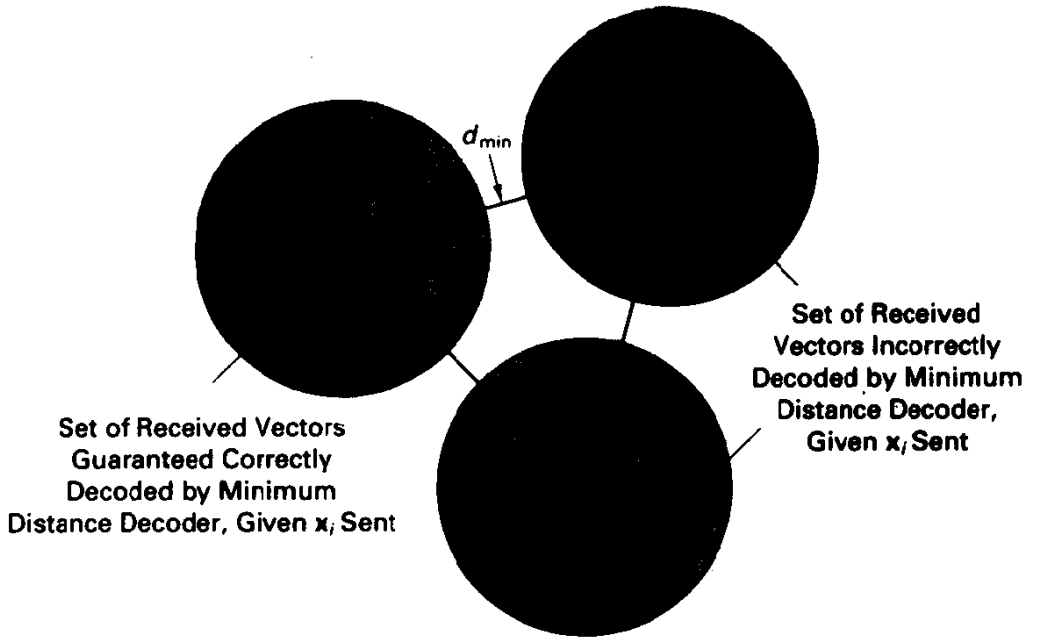

Figure 5.2.3a Zones of error correction and incorrect decoding.

or fewer errors, we are guaranteed that $\mathbf{r} = \mathbf{x}_i + \mathbf{e}$ will remain closer in Hamming distance to $\mathbf{x}_i$ than to any other codeword and will thus be correctly decoded.[8] Thus, we say that

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor \text{ is the guaranteed error-correcting capability of the code.}$$

If $t + 1$ or more errors occur in a codeword, we may be fortunate to find that $\mathbf{r}$ is still closer to $\mathbf{x}_i$ than any other codeword, but *some* such error patterns will lead to decoding error. This geometric argument applies for nonlinear codes as well.

As a decoding alternative, suppose the decoder's task is only to detect the presence or absence of errors and, if errors are detected, to label the codeword as unreliable. This detector can be fooled only if $\mathbf{e}$ takes the transmitted vector $\mathbf{x}_i$ to $\mathbf{x}_j$, another codeword; that is, $\mathbf{x}_i + \mathbf{e} = \mathbf{x}_j$. This cannot occur if there are $d_{\min} - 1$ or fewer errors in the n positions of the code (Figure 5.2.3b). Consequently,

$$d_{\min} - 1 \text{ is the guaranteed error detection capability of the code.}$$

Again, if $d_{\min}$ or more errors occur, the error pattern may still be detectable. Observe that the guaranteed number of detectable errors is roughly twice the number of correctable errors.

Hybrid modes of operation employing concurrent error correction and detection are possible and will be described shortly under performance analysis. It is not difficult

[8] The notation $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x, sometimes known as the *floor* function.

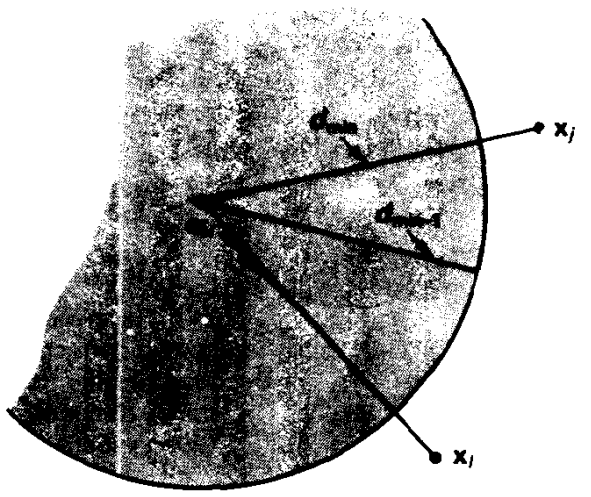

Figure 5.2.3b Guaranteed error *detection region, given* $\mathbf{x}_i$ *sent.*

to show by geometric arguments that we can guarantee correction of t_1 errors and still detect up to $t_2 > t_1$ errors provided that $t_1 + t_2 < d_{\min}$.

Still another decoder alternative is provided if the channel occasionally erases code symbols, as described in Chapter 2, in addition to causing errors.[9] Using similar geometric arguments, we may be assured of simultaneously correcting any combination of t errors and s erasures, provided that $2t + s < d_{\min}$. (See Exercise 5.2.13.)

Other channel models are certainly important, and we must reiterate that this kind of hard-decision decoding is often inferior to true maximum likelihood decoding for the actual physical channel. However, we will see later in this chapter that $d_{\min}$ plays a key role in more general decoding situations, motivating the search for linear codes with large $d_{\min}$ for a given (n, k, q).

Another important perspective on $d_{\min}$ can be gained by the parity check matrix. We recall that $\mathbf{x} = \mathbf{0}$ is a codeword of every linear code, and study of the error-correcting potential can be performed by assuming that this code vector is transmitted. Clearly, this vector, when postmultiplied by $\mathbf{H}$, will produce the zero vector of parity checks. Other valid codewords must also do the same. A relevant question is, "What is the *minimum weight* of other vectors $\mathbf{x}_i$ that have the requisite parity check result?", for this will reveal the smallest number of changes in the $\mathbf{0}$ vector that is required to allow it to masquerade as another codeword. Multiplying $\mathbf{x}$ by $\mathbf{H}^T$ amounts to forming linear combinations of columns of $\mathbf{H}$, or rows of $\mathbf{H}^T$; that is, $\mathbf{x}\mathbf{H}^T = \sum x_j \mathbf{h}_j^T$, where $\mathbf{h}_j$ is the jth column of $\mathbf{H}$. In effect, we are asking, "What is the smallest number of columns of $\mathbf{H}$ that, when multiplied by a nonzero scalar in GF(q) and added together, can give a zero vector?" In the language of linear algebra, "What is the smallest number of columns that are linearly dependent?" Thus, $d_{\min}$ is the smallest number of columns of $\mathbf{H}$ that are linearly dependent; alternatively, if we can show that all sets of $d-1$ columns of the parity check matrix are linearly independent, then we know that the code's minimum distance, $d_{\min}$, is at least d.

[9] Such erasures are neutral with respect to code symbols; the decoder proceeds as if this position of the codeword had never been transmitted.

5.2.3 Decoding of Linear Block Codes (Maximum Likelihood and Algebraic)

Suppose that we encode a message $\mathbf{u}$ into $\mathbf{x}$ by $\mathbf{x} = \mathbf{uG}$ as discussed. Let each code symbol x_j be transmitted through a channel, producing a *vector* observation $\mathbf{r}_j$, $j = 0, 1, \ldots, n-1$. For example, when PSK modulation is employed with binary codes, each code symbol produces a real demodulator output $r_j = \pm E_s^{1/2} + n_j$, where n_j is Gaussian noise. If the symbols are in GF(8), then use of 8-ary orthogonal modulation with noncoherent detection would produce eight numbers for each code position, they being the magnitudes (or squared magnitudes) of matched filter outputs.[10] The ML decoder will find the codeword $\mathbf{x}_i$ that maximizes the likelihood $f(\tilde{\mathbf{r}}|\mathbf{x}_i)$, where $\tilde{\mathbf{r}} = (\mathbf{r}_0, \ldots, \mathbf{r}_{n-1})$ is a supervector combining all the available data. If the channel is memoryless from symbol to symbol, the p.d.f. factors into product form, and if we instead maximize the logarithm of the p.d.f, we find that the optimal choice for the codeword is

$$\hat{\mathbf{x}} = \arg_{x_i} \max \sum_{j=0}^{n-1} \log f(\mathbf{r}_j|x_{i_j}) = \arg_{x_i} \max \sum_{j=0}^{n-1} \lambda(\mathbf{r}_j, x_{i_j}), \tag{5.2.23}$$

where $\lambda(\mathbf{r}_j, x_{i_j})$ is a metric of goodness (the log likelihood) for symbol x_{i_j} when the vector $\mathbf{r}_j$ is received. On the binary antipodal channel, the ML symbol metric is simply $\lambda = r_j x_{i_j}$, while on the M-ary orthogonal noncoherent channel, the ML metric is $\lambda(\mathbf{r}_j, x_j = m) = \log I_0(\mu r_{j_m}/\sigma^2)$; that is, we utilize only the analog demodulator channel output corresponding to the code symbol under test in a given codeword. In both cases, we have removed unnecessary constants from the log-likelihood expressions to simplify the metric.

The general ML decoder must apparently compute q^k sums as in (5.2.23) and choose the codeword index with the largest metric. For general linear codes there appears to be no way of surmounting this exponential complexity problem, because the ML decoding problem has been shown to be *NP-complete*, meaning that it is equivalent to a class of problems known to be computationally difficult in the sense that no polynomial-time solution has been found and is not likely to be. (Actually, we will see that it is possible to decode with complexity proportional to q^{n-k}, which may be much smaller than q^k.) We should not be unduly deterred, however, because the problem is certainly feasible for modest-sized codes, and even for certain long codes, efficient suboptimal algorithms can at least approximate ML decoding.

We will return to the general ML decoding problem later in the chapter, following a development of cyclic codes, and investigate procedures for efficiently organizing or perhaps approximating ML decoding. For the present, however, we develop what are referred to as ***algebraic decoding*** algorithms, which operate with an error-correction viewpoint. That is, the channel is viewed as a q-ary input, q-ary output discrete memoryless channel. We further assume a uniform channel such that when an error is made in transmission of a code symbol, with probability P_s, it is equally likely to be one of $q-1$ possibilities. (A possible extension is to let the demodulator produce one of q symbols

[10] We should realize that these form sufficient statistics for the message decoding problem.

or an *erasure* when the demodulator holds low confidence in its ability to decide which symbol occurred.)

Now consider the log-likelihood function for such a channel. The metrics $\lambda(r_j, x_{i_j})$ are two-valued:

$$\lambda(r_j, x_{i_j}) = \begin{cases} \log(1 - P_s), & r_j = x_{i_j}, \\ \log\left(\dfrac{P_s}{q-1}\right), & r_j \neq x_{i_j}. \end{cases} \tag{5.2.24}$$

That is, when the observed demodulator output agrees with the symbol of a test codeword, we assign a slightly negative metric, whereas when a discrepancy is found, we attach a more negative metric.[11] Actually, this may be simplified by noting that we may translate the two metrics so that the largest is zero and then scale so that the smallest is -1. Then ML decoding for this channel is equivalent to using a Hamming distance (0/1) metric and *minimizing* the vector Hamming distance over all choices of codewords. To emphasize, for the M-ary uniform channel, ML decoding is equivalent to finding the $\mathbf{x}_i$ that is closest in Hamming distance to $(r_0, r_1, \ldots, r_{n-1})$. This ***minimum Hamming distance decoding*** is so plausible that we may expect it is always valid, but all we need to do otherwise is to change the channel symmetry slightly, or let the channel output be Q-ary, where $Q > q$.

The task then is to find the closest (in the Hamming distance sense) codeword to $\mathbf{r}$ among the q^k codewords. Clearly, exhaustive search of the codebook is possible, but we can do much better. The received vector may be written as

$$\mathbf{r} = \mathbf{x}_i + \mathbf{e}, \tag{5.2.25}$$

where $\mathbf{e}$ is an n-tuple of channel error symbols from GF(q) and addition is over GF(q). On a binary channel, $\mathbf{e} = (000110)$ would mean that errors occurred in positions designated 3 and 4 in a vector of length 6; on an 8-ary channel, the error vector might be $\mathbf{e} = (0, \alpha, 0, \alpha^4, 0, 0)$, where α is a primitive element in GF(8). Both error vectors have Hamming weight 2.

Since $d(\mathbf{r}, \mathbf{x}_i) = wt(\mathbf{r} - \mathbf{x}_i) = wt(\mathbf{e})$, we need to find the minimum weight (or most probable) error pattern that could have produced the given $\mathbf{r}$ and then subtract this from the received vector. If our estimate, $\hat{\mathbf{e}}$, is correct, the proper codeword condition is restored and the errors in the message have been corrected. A conceptual way to proceed, outlined in Figure 5.2.4a, is to begin subtracting low-weight $\mathbf{e}$ vectors from $\mathbf{r}$, each time checking whether $(\mathbf{r} - \mathbf{e})\mathbf{H}^T = \mathbf{0}$. As soon as the parity check equation is satisfied, we may stop. The problem is that many error vectors may need to be tested and, for each, a vector subtraction and vector–matrix multiplication must be performed. There is an easier way for *linear* codes.

Consider first forming the $(n-k)$-vector $\mathbf{s} = \mathbf{r}\mathbf{H}^T$. Note that

$$\mathbf{s} = \mathbf{r}\mathbf{H}^T = (\mathbf{x}_i + \mathbf{e})\mathbf{H}^T = \mathbf{x}_i\mathbf{H}^T + \mathbf{e}\mathbf{H}^T = \mathbf{e}\mathbf{H}^T, \tag{5.2.26}$$

where the last step follows from the parity check property of any code vector. Therefore, the elements of $\mathbf{s}$ are linear combinations of the errors, e_i, and $\mathbf{s}$ is referred to as the ***syndrome***, or symptom, of the error pattern. The remaining difficulty is that there

[11] We assume without penalty that $P_s \leq (q-1)/q$.

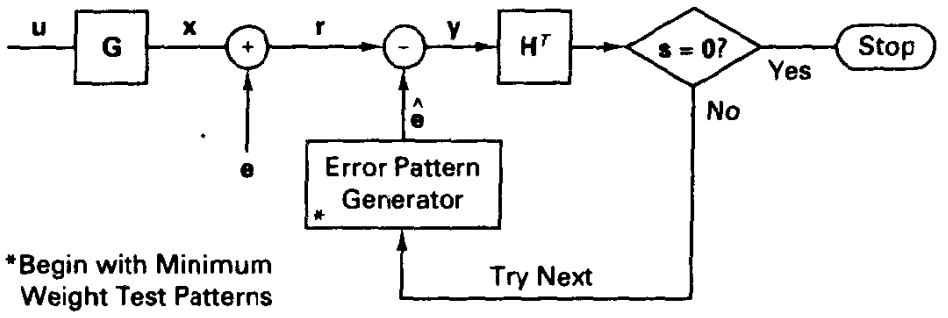

Figure 5.2.4a Naive maximum likelihood decoder.

are many (in fact q^k) error patterns capable of producing a given syndrome, one error pattern being associated with transmission of each valid codeword; ML decoding requires determination of the *minimum weight* error pattern producing s. In Figure 5.2.4b, we illustrate conceptually this signal processing. Forming the syndrome is operationally equivalent to encoding, that is, multiplying a vector by a matrix, but the syndrome-to-error pattern conversion is potentially more difficult.

A reasonable concern is whether in condensing **r** down to **s** by a matrix multiplication we have changed the nature of the solution obtained for the error pattern. (Equivalently, is **s** a sufficient statistic for the problem?) It turns out that we have not altered the problem—the solution sets are identical. That is, the set of error patterns that when added to codewords produces a given **r** is exactly the same set of error patterns that can produce the computed syndrome. (See Exercise 5.2.8.) Finding the minimum-weight member of either set obviously yields the same answer.

Now, let's resume the hunt for the minimum-weight **e** consistent with **s**. For cases where the number of distinct syndromes, q^{n-k}, is reasonably sized, the following approach is a possible implementation of the final step. (Even for cases where it is not feasible, it illuminates the general decoding problem.) We can precompute the relation $\mathbf{s} = \mathbf{e}\mathbf{H}^T$ for all error patterns and form a table, called the ***standard array***, of error cosets producing the same syndrome. There will be q^{n-k} such cosets, each with q^k entries. The most likely (minimum weight) error pattern, or any one of several equally likely patterns, is designated the ***coset leader***. (It may be helpful to think of cosets, indexed

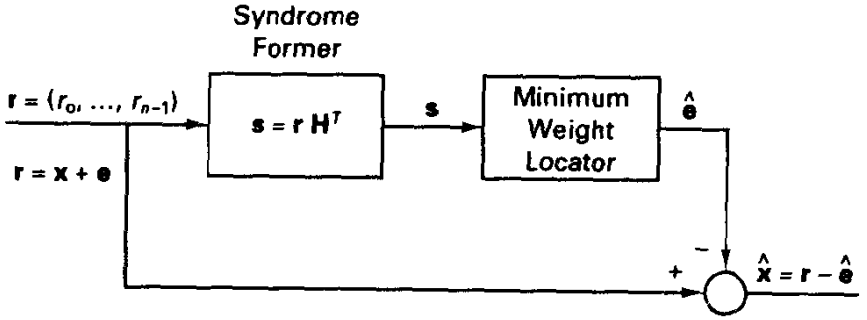

Figure 5.2.4b Generic decoder for q-ary symmetric channel.

by syndromes, as sets of diseases that produce a set of observable conditions and the coset leader as the most likely diagnosis.) In our decoding table, we only need to store the coset leaders for each syndrome address and, in fact, for systematic codes we must only store the k information positions of the coset leaders, since we normally only need to deliver information estimates. Thus, the table size is q^{n-k} by k. The "correction" of errors is performed by subtracting this coset leader from the receiver vector.

Example 5.8 Standard Array for (6, 3) Code

In Figure 5.2.5, we show the standard array for the (6, 3) code described in (5.2.20). This table can be constructed by enumerating all possible error patterns and storing them in the row corresponding to the associated syndrome. However, to find the essential coset leaders, we begin with low-weight (most likely) error patterns, form $\mathbf{s} = \mathbf{e}\mathbf{H}^T$, and store $\mathbf{e}$ at the corresponding location in the syndrome table. We proceed through higher-weight error patterns until all syndromes have a coset leader assigned. For the (6, 3) code, the zero-error pattern and all six single-error patterns consume seven distinct syndromes. The remaining syndrome, $\mathbf{s} = (101)$, can only be produced by an error pattern with two or more errors. Although we have shown the complete array here, a syndrome decoding table need only associate coset leaders with syndromes.

It should be observed in Figure 5.2.5 that the error coset corresponding to the syndrome $\mathbf{s} = (000)$ is exactly the set of codewords, since $\mathbf{x}\mathbf{H}^T = \mathbf{0}$ for valid codewords. Thus, the top row of the array in Figure 5.2.5 is the set of eight valid codewords. Also, each remaining coset is formed by adding the coset leader vector to each vector in the code. This is true more generally.

s	e							
000	000000	100111	010110	001011	110001	011101	101100	111010
001	000001	100110	010111	001010	110000	011100	101101	111011
010	000010	100101	010100	001001	110011	011111	101110	111000
011	001000	101111	011110	000011	111001	010101	100100	110010
100	000100	100011	010010	001111	110101	011001	101000	111110
101	000101	100010	010011	001110	110100	011000	101001	111111
110	010000	110111	000110	011011	100001	001101	111100	101010
111	100000	000111	110110	101011	010001	111101	001100	011010

Coset Leader — Coset with q^k Vectors

$$\mathbf{G} = \begin{bmatrix} 100111 \\ 010110 \\ 001011 \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} 110100 \\ 111010 \\ 101001 \end{bmatrix}.$$

Figure 5.2.5 Standard array for (6, 3) binary code.

Notice that the only error patterns that are correctable by such a table lookup decoder are the coset leaders, for if an error pattern occurs that is not a coset leader, the decoder retrieves an erroneous error pattern and typically leaves the "corrected" vector with more errors than it contained originally. In Example 5.8, the zero-error event, all six one-error patterns, and a single two-error pattern are corrected. Notice, however, several two-error patterns are equally likely to produce the syndrome $\mathbf{s} = (101)$, and

attempting to correct this condition is less probable of success than attempting to correct when observing other less ambiguous syndromes.

5.2.4 Performance Measures for Algebraic Decoding

We now describe three possible modes of operation for a decoder, two of which allow the decoder to avoid decision in some cases, but instead report an error condition it cannot correct. In the case when the decoder does not produce a decision, we presume there is a higher-level decoding that can fill the erasure of a codeword or that we may request that the same codeword be repeated again.[12]

Mode 1: Error detection only

Here we simply test for a zero syndrome vector, since $\mathbf{s} = \mathbf{0}$ only for valid codewords. If the syndrome is not zero, we report a detected error. This decision can be wrong only if the error pattern is very special, that is, $\mathbf{e}$ itself is a code vector, for then $\mathbf{r} = \mathbf{x}_i + \mathbf{e}$ will be some other valid code vector, and its syndrome will be the zero vector.

For this mode of operation, the performance measure of interest is the ***probability of undetected error***, denoted P_{UE}. We then have for the *q-ary uniform channel*

$$\begin{aligned} P_{\text{UE}} &= P(\mathbf{e} = \mathbf{x}_j : \mathbf{x}_j \neq \mathbf{0}) \\ &= \sum_{w=d_{\min}}^{n} A_w P(\mathbf{e} \text{ is a specific weight } w \text{ pattern}) \\ &= \sum_{w=d_{\min}}^{n} A_w \left(\frac{P_s}{q-1}\right)^w (1-P_s)^{n-w}. \end{aligned} \tag{5.2.27a}$$

Thus, knowledge of the complete weight spectrum of the code allows exact calculation of P_{UE}. For the (6, 3) binary code on the BSC, $P_{\text{UE}} = 4P_s^3(1-P_s)^3 + 3P_s^4(1-P_s)^2$, where P_s is the code symbol error probability.

If only the minimum distance of the code is known, an upper bound on P_{UE} is

$$P_{\text{UE}} \leq \sum_{w=d_{\min}}^{n} C_j^n \left(\frac{P_s}{q-1}\right)^w (1-P_s)^{n-w}. \tag{5.2.27b}$$

Mode 2: Complete decoding

Here the decoder is always required to deliver its best estimate of the transmitted codeword based on syndrome measurement. This too can fail, but the probability of *correct* decision is given by

$$P_{\text{CD}} = P(\mathbf{e} \text{ is a coset leader}). \tag{5.2.28}$$

For the (6, 3) code, $P_{\text{CD}} = (1-P_s)^6 + 6P_s(1-P_s)^5 + P_s^2(1-P_s)^4$. The probability of incorrect decoding in this mode is $P_{\text{ICD}} = 1 - P_{\text{CD}} = P(\mathbf{e} \text{ is not a coset leader})$. Since exact calculation of P_{CD} requires knowledge of the set of coset leaders, for reasons of

[12]This is called ARQ for automatic request of retransmission.

tractability we often upper-bound P_{ICD} by the probability of error types that are not *guaranteed* correctable. Thus, for the binary (6, 3) code, we could upper-bound the probability of incorrect decoding by

$$\begin{aligned} P_{ICD} &\leq P(\mathbf{e} \text{ has } 2, 3, \ldots, 6 \text{ errors}) \\ &= \sum_{j=2}^{6} C_j^6 P_s^j (1 - P_s)^{6-j}, \end{aligned} \tag{5.2.29a}$$

since all error patterns with two or more errors are not guaranteed correctable. In general, we have the bound

$$P_{ICD} \leq \sum_{j=t+1}^{n} C_j^n P_s^j (1 - P_s)^{n-j}. \tag{5.2.29b}$$

Knowing t alone is sufficient to compute this bound.

Mode 3: Incomplete decoding

Here we allow the decoder to attempt to correct situations that it believes correspond to as many as $t_1 \leq t$ errors, where t is the guaranteed error-correcting limit for the code, and to report detected errors for other syndromes. Thus, we adopt a hybrid mode of operation combining the first two modes. In essence, the standard array is divided into two sections: a set of cosets for which error correction is attempted and a set for which transmission error will be reported. In mode 3, three possibilities exist: we can decode correctly; we may decode incorrectly when $\mathbf{r}$ falls within t_1 units of an incorrect codeword; or we may have a detected error, when $\mathbf{r}$ lies in the interstitial space between spheres of radius t_1 drawn about all codewords. (See Figure 5.2.6.) The probabilities of these three events are obviously related by

$$P_{CD} + P_{ICD} + P_{DE} = 1. \tag{5.2.30}$$

(DE denotes the detected error event.)

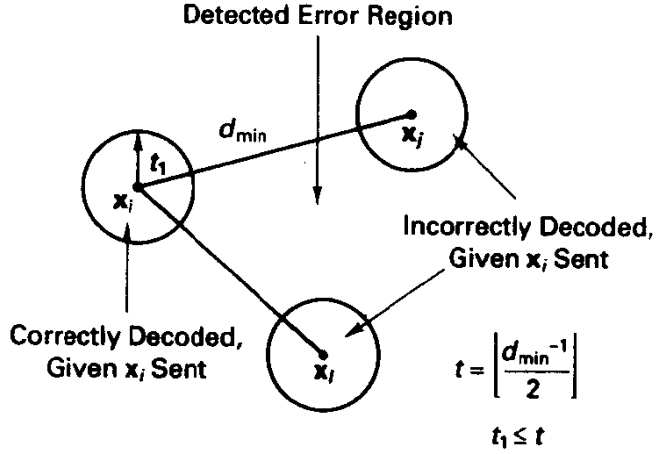

Figure 5.2.6 Geometry for mode 3, incomplete decoding.

For example, with the (6, 3) code, we could take $t_1 = t = 1$ and decide to report detected errors when the ambiguous syndrome $\mathbf{s} = (101)$ is observed; otherwise, we decode as usual. In this case the probability of correct decoding is slightly less than in mode 2:

$$P_{CD} = (1 - P_s)^6 + 6P_s(1 - P_s)^5, \tag{5.2.31}$$

but we have incurred a lesser probability of incorrect decoding, at the expense of failing to decode occasionally. In general, P_{ICD} is the probability of an error pattern that is not a coset leader, but that lies in the cosets for which correction is attempted. Similarly, P_{DE} is the probability that the error pattern is in the detected-error cosets.

Incomplete decoding is particularly common when the minimum distance of the code is even. In such a case we may perform the decoding so that $t_1 = \lfloor (d_{min} - 1)/2 \rfloor$ errors are corrected, while $d_{min}/2$ errors are detected. (Both are guaranteed.) In the case when $d_{min} = 4$, the code is referred to as single error correcting, double error detecting (SEC/DED). There may be confusion at this point with the claim of mode 1 decoding: that up to $d_{min} - 1$ errors are detectable. The guarantees are simply different if simultaneous error correction and error detection are attempted.

Many algebraic decoding algorithms are incomplete decoding procedures, producing the single valid codeword within t Hamming units of the received vector, if one exists. If no such codeword exists because more than t errors occur in transmission, the algorithms may simply fail to decode, declaring detected errors.

For the Hamming (7, 4) code (and in fact for all Hamming codes), the standard array is segmented in a special way, a corollary to its perfectness. All single-error patterns, plus the zero-error pattern, are coset leaders. No other error pattern is a coset leader. Thus, in mode 1 (error detection only), $P_{UE} = P(3 \text{ or more errors})$, and in mode 2 (complete decoding), $P_{ICD} = P(2 \text{ or more errors})$. Mode 3 is not applicable for the Hamming codes.

The general syndrome decoder, when combined with table lookup, is not practical for long codes, since the syndrome table requires q^{n-k} words. This rapidly becomes unmanageable, especially for $q > 2$. It is true, however, that table lookup decoders that were infeasible 10 years ago are perfectly reasonable today due to dramatic increase in the sizes and speeds of semiconductor memory.

In addition to this complexity issue surrounding general linear codes, we note that encoding ($\mathbf{x} = \mathbf{uG}$) and syndrome computation ($\mathbf{s} = \mathbf{rH}^T$) are matrix operations over GF(q), and, in general, no shortcuts are possible to avoid brute-force matrix multiplication, meaning that up to nk and $n(n-k)$ multiplications and additions are necessary for these respective operations. In Section 5.4, we take up cyclic codes, which are linear codes allowing still simpler encoding and syndrome formation. More importantly, if we wish to avoid table lookup for the error pattern, algebraic procedures are available for the most important of these codes to solve for the minimum-weight error pattern, or to do so with high probability.

We will now turn to some constructive procedures for building simple codes, the Hamming codes and Reed–Muller codes. More general constructions will be presented later in the context of cyclic codes.

5.2.5 Hamming Codes over GF(q)

The description of Hamming codes, both longer binary codes and nonbinary codes, is straightforward, involving specification of the parity check matrix. In Section 5.4, we will show that these codes are in fact equivalent to cyclic codes. The codes are the oldest nontrivial codes, discovered in the binary case about the same time by Hamming and Golay.

We consider single-error correcting (n, k) codes over GF(q) and specify as the coset leaders the zero vector and all n-vectors with a single nonzero entry, and no others. There are $n(q-1)$ such vectors of weight 1 and one vector with weight 0. Since the number of distinct syndromes is q^{n-k}, it is *necessary* that

$$q^{n-k} = 1 + n(q-1) \tag{5.2.32a}$$

or

$$n = \frac{q^{n-k}-1}{q-1} = q^{n-k-1} + q^{n-k-2} + \cdots + 1. \tag{5.2.32b}$$

Now we show how to produce such codes by specifying their parity check matrices. We begin by picking a value of $n-k$, that is, choosing the number of parity symbols. The required code length then follows from (5.2.32b). We select as columns of the $\mathbf{H}$ matrix all the distinct nonzero $(n-k)$-tuples over GF(q), with the proviso that the first nonzero entry in each column be 1. All single-error patterns and the zero-error pattern will produce a unique syndrome, since $\mathbf{s} = \mathbf{e}\mathbf{H}^T$, (5.2.26). These error patterns will thus be correctable. Furthermore, no other error patterns are correctable since the coset leaders have been completely assigned.

Hamming codes are particularly easily decoded by studying the dependence of the syndrome vector on the error pattern, assumed to contain at most one error. The error pattern can then be determined with minimal algebraic effort. In the binary case, if the parity check matrix columns are ordered in natural binary fashion (producing nonsystematic code), a nonzero binary syndrome vector points to the location of the single error.

For $q = 2$, the first few Hamming codes are $(3, 1)$,[13] $(7, 4)$, $(15, 11)$, $(31, 26)$, and $(63, 57)$. For $q = 16$, the shortest Hamming code is $(17, 15)$, and two others are $(273, 270)$ and $(4369, 4365)$. Notice that the rate $R = k/n$ is steadily increasing as the block length n grows, yet $d_{\min} \equiv 3$.

The weight distribution for q-ary Hamming codes is known in closed form [2] and is given by

$$\begin{aligned} A(z) = \frac{1}{n(q-1)+1}\Big[&[1+(q-1)z]^n \\ &+ n(q-1)[1+(q-1)z]^{(n-1)/q}(1-z)^{(n(q-1)+1)/q}\Big]. \end{aligned} \tag{5.2.33}$$

[13] This is a degenerate Hamming code, a repetition code.

Example 5.9 (5, 3) Hamming Code over GF(4)

Suppose we pick $q = 4$ and $n - k = 2$. Then (5.2.32b) gives $n = 5$. Thus, the code will be a (5, 3) single-error-correcting code. The parity check matrix has nonzero columns that are distinct and for which the leading nonzero element is 1. This produces

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & \alpha & \beta & 0 & 1 \end{bmatrix}. \tag{5.2.34}$$

Note that we have put the $\mathbf{H}$ matrix in systematic form, $\mathbf{H} = [-\mathbf{P}^T, \mathbf{I}]$.

Substitution in the weight enumerator polynomial expression (5.2.33) yields, after routine algebra, the polynomial

$$A(z) = 1 + 30z^3 + 15z^4 + 18z^5. \tag{5.2.35}$$

This reveals that the code has 1 weight 0 vector (as we already knew), 30 codewords of weight 3, 15 vectors of weight 4, and 18 vectors of weight 5. The weight spectrum totals 64 vectors, as it must.

5.2.6 Reed–Muller Codes

Reed–Muller (RM) codes [9] are binary linear codes whose lengths are a power of 2, $n = 2^m$, and for which there is a very simple decoding algorithm, capable of inferring information symbols directly from majority voting on the results of symbol estimator equations. For modest block lengths, the codes compare well against the best-known codes with equivalent rate. Lately, they have reemerged as fundamental to the description of dense multidimensional lattices.

The generator matrix is defined as follows. For any m, let $n = 2^m$, and pick the first row of the generator matrix as the n-tuple of 1's:

$$\mathbf{G}_0 = \mathbf{g}_0 = (1, 1, 1, 1, \ldots, 1, 1, 1). \tag{5.2.36}$$

Used alone, this would form a 0th-order RM code, which is just a repetition code, having $d_{\min} = n = 2^m$. Next, define $\mathbf{G}_1$ to be the $m \times n$ matrix whose columns correspond to all distinct binary m-tuples. (We order these left to right in natural binary order.) Appended underneath $\mathbf{G}_0$, we form a generator matrix for a first-order RM code.

Similarly, we define $\mathbf{G}_2$ to be the matrix formed by taking the logical AND of all pairs of rows in $\mathbf{G}_1$. (This is equivalent to forming the product of vectors, bit by bit.) There are C_2^m ways of forming such pairs. Appending this matrix to the former, we have a matrix with

$$k = 1 + C_1^m + C_2^m \tag{5.2.37}$$

rows, which can be shown to be linearly independent. This constitutes the second-order RM code.

If desired, we can keep adding rows by taking rows from $\mathbf{G}_1$ three at a time, forming the logical AND, then four at a time, and so on. In general, the rth-order RM code will have

$$k = 1 + C_1^m + C_2^m + \cdots + C_r^m \tag{5.2.38}$$

rows and will be of the form

$$\mathbf{G} = \begin{bmatrix} \mathbf{G}_0 \\ \mathbf{G}_1 \\ \mathbf{G}_2 \\ \vdots \\ \mathbf{G}_r \end{bmatrix} \tag{5.2.39}$$

Although we will not demonstrate it, the matrix has rank k, and the minimum distance of the RM code with order r is

$$d_{\min} = 2^{m-r}, \qquad r = 0, 1, \ldots, m-1. \tag{5.2.40}$$

It is readily seen that the rth-order RM code is a subcode of the $(r+1)$st-order code (in Section 5.6 we will refer to this as an expurgated code). Alternatively, an $(r+1)$st-order RM code is comprised of cosets of the rth-order RM code.

Reed [9] proposed a simple voting algorithm that recovers information bits directly from sums of the received binary symbols; no syndromes are computed. The procedure is iterative. We think of the message as broken into blocks, $\mathbf{u} = (\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_r)$, of length corresponding to the row dimension of the various component matrices in (5.2.39). The first subblock has one information bit, the second m, and the last C_r^m. We decode the last block bit by bit by forming 2^{m-r} sums of 2^r received bits each. The sums are chosen so that the position to be decoded appears once in each sum, and all other positions appear an even number of times in each sum. This means that a majority of the sums will yield a result equaling the message bit in question, provided that

$$t_r = \left\lfloor \frac{2^{m-r}-1}{2} \right\rfloor = \left\lfloor \frac{d_{\min}-1}{2} \right\rfloor \tag{5.2.41}$$

or fewer errors occur. To decode the other bits of this block, we permute the arrangement of sums, obeying the preceding rule.

When the last subblock is decoded, we remove the contribution of these information bits from the transmitted (and received) words by

$$\mathbf{r}' = \mathbf{r} - \hat{\mathbf{u}}_r \mathbf{G}_r \tag{5.2.42}$$

If no errors occurred, this vector would be a codeword in the $(r-1)$st-order RM code. The bits of the next subblock are decoded in similar fashion. Now we form 2^{m-r-1} sums each involving 2^{r+1} positions and claim that if fewer than

$$t_{r-1} = \left\lfloor \frac{2^{m-r-1}-1}{2} \right\rfloor \tag{5.2.43}$$

errors occur, then the next subblock is correct, and so on. The last block of one bit is decoded by summing all the remaining code positions, as modified, which is equivalent to a majority vote. The complete process is successful if t_r or fewer errors exist.

Example 5.10 (16, 11) Second-order RM Code

Following the preceding recipe, the generator matrix for this code can be constructed as

$$\mathbf{G} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2.44}$$

Notice that the submatrices have weights 16, 8, and 4 in the various rows.

We view the 11-bit message as

$$\mathbf{u} = (\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2) = (u_0 \mid u_1, u_2, u_3, u_4 \mid u_5, u_6, u_7, u_8, u_9, u_{10}). \tag{5.2.45}$$

To decode u_{10}, we form the four estimates

$$\begin{aligned} \hat{u}_{10} &= r_0 + r_1 + r_2 + r_3, \\ \hat{u}_{10} &= r_4 + r_5 + r_6 + r_7, \\ \hat{u}_{10} &= r_8 + r_9 + r_{10} + r_{11}, \\ \hat{u}_{10} &= r_{12} + r_{13} + r_{14} + r_{15}. \end{aligned} \tag{5.2.46}$$

By studying the generator matrix, it is seen that each estimate equation includes position 15 once, and all other positions appear an *even* number of times. Thus, the value of the other message bits is self-canceling in each equation, leaving

$$\begin{aligned} \hat{u}_{10} &= u_{10} + e_0 + e_1 + e_2 + e_3, \\ \hat{u}_{10} &= u_{10} + e_4 + e_5 + e_6 + e_7, \\ \hat{u}_{10} &= u_{10} + e_8 + e_9 + e_{10} + e_{11}, \\ \hat{u}_{10} &= u_{10} + e_{12} + e_{13} + e_{14} + e_{15}. \end{aligned} \tag{5.2.47}$$

Observe that if there is 0 or 1 error in the 16 received bits a *majority* of the preceding estimates will yield the proper value, $\hat{u}_{10} = u_{10}$. To decode u_9, we point out that positions (0, 1, 4, 5), (2, 3, 6, 7), (8, 9, 12, 13), and (10, 11, 14, 15) provide the appropriate set of estimator equations.

After the bits $u_5 \ldots u_{10}$ are decoded in corresponding manner, we subtract $\hat{\mathbf{u}}_2\mathbf{G}_2$ from $\mathbf{r}$ and begin to work on the next four bits, which are a fragment of message bits in a first-order RM code. To decode u_4, for example, we form eight sums (estimates) each involving two bits. The appropriate check sets for this bit are (0, 1), (2, 3), (4, 5).... Provided three or fewer errors exist in the original symbols, this bit and others in its subblock are correctly decoded. Once this subblock's influence is finally removed, we decide u_0 by summing all the bits, or majority voting.

Inspection of the procedure will show correction of up to $\lfloor (d_{\min} - 1)/2 \rfloor$ errors in an rth-order RM code, which is the guaranteed error-correcting capacity based on $d_{\min}$.