Chapter 13

Error-Correcting Codes

Contents

As introduced in §1.1.4, the subject of linear error-correcting codes (ECCs), a.k.a. linear codes (LCs), is at once elegant, intricate, and practical. Efficient LCs may be devised for vectors in $\mathbf{F}_q^n$, with each of their n elements defined over a set of symbols in a *finite field* $\mathbf{F}_q$ of order q, where $q = p^a$ with p prime; cases of particular interest include the *binary field* $\mathbf{F}_2 = \{0, 1\}$, the *ternary field* $\mathbf{F}_3 = \{0, 1, 2\}$, and the *quaternary field* $\mathbf{F}_4 = \{0, 1, \omega, \bar{\omega}\}$, where $\omega = (-1 + \mathrm{i}\sqrt{3})/2$. Codes with $q = 2$, 3, and 4 are called, respectively, linear binary codes (LBCs), linear ternary codes (LTCs), and linear quaternary codes (LQCs). An LC is defined by two matrices, a *basis matrix* V and a *parity-check matrix* H. The use of an LC to communicate data over a noisy channel is straightforward:

- **group** the data into *vectors* (blocks) of length k, with each element defined over an *alphabet* of q symbols;
- **code** each resulting data vector $\mathbf{a} \in \mathbf{F}_\mathbf{q}^\mathbf{k}$ into a longer codeword $\mathbf{w} \in \mathbf{F}_q^n$, with $n > k$, via $\mathbf{w} = V_{[n,k]_q}\mathbf{a}$;
- **transmit** the corresponding codeword $\mathbf{w}$ over the noisy channel;
- **receive** the (possibly corrupted) message $\hat{\mathbf{w}} \in \mathbf{F}_q^n$ on the other end, and
- **decode** the received message $\hat{\mathbf{w}}$ leveraging $H_{[n,k]_q}$; that is, find the most likely codeword $\mathbf{w}$ corresponding to the received message $\hat{\mathbf{w}}$, and the data vector $\mathbf{a}$ that generated it.

The identification of matrices $\{V_{[n,k]_q}, H_{[n,k]_q}\}$ that define efficient LCs, and streamlined algorithms that can quickly code and decode messages using such LCs, have a rich history and many remarkable solutions.

On a finite field $\mathbf{F}_q$, addition $(+)$ and multiplication $(\cdot)$ are closed (that is, they map to elements within the field) and satisfy the usual rules: they are associative, commutative, and distributive, there is a 0 element such that $a + 0 = a$, there is a 1 element such that $a \cdot 1 = a$, for each a there is an element $(-a)$ such that $a + (-a) = 0$, and for each $a \neq 0$ there is an element a^{-1} such that $a \cdot a^{-1} = 1$. For example, addition and multiplication on $\mathbf{F}_2$, $\mathbf{F}_2$, and $\mathbf{F}_4$ are defined as follows:

$\mathbf{F}_2$:

$+$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

$\mathbf{F}_3$:

$+$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

$\cdot$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

$\mathbf{F}_4$:

$+$	0	1	ω	$\bar\omega$
0	0	1	ω	$\bar\omega$
1	1	0	$\bar\omega$	ω
ω	ω	$\bar\omega$	0	1
$\bar\omega$	$\bar\omega$	ω	1	0

$\cdot$	0	1	ω	$\bar\omega$
0	0	0	0	0
1	0	1	ω	$\bar\omega$
ω	0	ω	$\bar\omega$	1
$\bar\omega$	0	$\bar\omega$	1	ω

(13.1)

The **Hamming distance** between two vectors in $\mathbf{F}_q^n$ is simply the number of elements that differ between them. An $[n,k]_q$ LC is defined via a set of $k < n$ independent **basis vectors** $\mathbf{v}^i \in \mathbf{F}_q^n$; the q^k distinct valid **codewords** $\mathbf{w}^j \in \mathbf{F}_q^n$ of this LC are given by all *q-ary linear combinations* of the basis vectors $\mathbf{v}^i$ (that is, by all linear combinations with coefficients selected from $\mathbf{F}_q$, with addition and multiplication defined elementwise on $\mathbf{F}_q$). The basis vectors $\mathbf{v}^i$ are generally selected such the **minimum Hamming distance** d of the resulting LC (that is, the minimum distance between any two resulting codewords) is maximized. An LC is often denoted $[n,k,d]_q$, with the minimum distance d of the code specified explicitly.

We denote by $V_{[n,k]_q}$ the $n \times k$ **basis matrix** with the k basis vectors $\mathbf{v}^i$ as columns, and by $W_{[n,k]_q}$ the $n \times q^k$ **codeword matrix** with the q^k codewords $\mathbf{w}^j$ as columns. Without loss of generality, we write $V_{[n,k]_q}$ and a companion $(n-k) \times n$ **parity-check matrix** $H_{[n,k]_q}$ in the **systematic form**[1]

$$H_{[n,k]_q} = \begin{bmatrix} -P_{(n-k)\times k} & I_{(n-k)\times(n-k)} \end{bmatrix}, \quad V_{[n,k]_q} = \begin{bmatrix} I_{k\times k} \\ P_{(n-k)\times k} \end{bmatrix}, \quad \mathbf{w}^j = \begin{bmatrix} \mathbf{a^j} \\ \mathbf{b}^j \end{bmatrix} = V_{[n,k]_q}\mathbf{a^j}. \tag{13.2}$$

When written in systematic form, each of the codewords $\mathbf{w}^j$ block decomposes into its k **data symbols**[2] $\mathbf{a^j}$ and its $r = n-k$ **parity symbols** $\mathbf{b^j}$; r is sometimes called the *redundancy* of the code. Note that $H_{[n,k]_q} V_{[n,k]_q} = 0$ (on $\mathbf{F}_q$)[3], which establishes that each of the basis vectors $\mathbf{v}^i$ so constructed satisfies the **parity-check equation** $H_{[n,k]_q}\mathbf{v}^i = 0$ (on $\mathbf{F}_q$), and thus each of the resulting codewords $\mathbf{w}^j$ also satisfies the parity-check equation $H_{[n,k]_q}\mathbf{w}^j = 0$ (on $\mathbf{F}_q$). Note further that, for LBCs and LQCs, $P = -P$.

Any $[n,k]_q$ LC C has associated with it an $[n, n-k]_q$ **dual code** $C^\perp$ defined such that[4]

$$C^\perp = \{\mathbf{w} \in \mathbf{F}_q^n \;:\; \mathbf{w} \cdot \bar{\mathbf{u}} = 0 \text{ for all } \mathbf{u} \in C\}; \tag{13.3a}$$

the $k \times n$ parity-check matrix $H^\perp$ and $n \times (n-k)$ basis matrix $V^\perp$ of $C^\perp$ may be written in systematic form as

$$H^\perp_{[n,n-k]_q} = \begin{bmatrix} \bar{P}^T & I_{k\times k} \end{bmatrix}, \quad V^\perp_{[n,n-k]_q} = \begin{bmatrix} I_{(n-k)\times(n-k)} \\ -\bar{P}^T \end{bmatrix}. \tag{13.3b}$$

[1]In the literature on this subject, it is more common to use a "generator matrix" G to describe the construction of linear codes; the "basis matrix" convention V used here is related simply to the corresponding generator matrix such that $V = G^T$. We find the basis matrix convention to be a bit more natural in terms of its linear algebraic interpretation.

[2]The word "bit", a portmanteau of "binary digit", is reserved for the case with $q = 2$; for $q \geq 2$, we use the word "symbol" instead.

[3]The qualifier "on $\mathbf{F}_q$" is used to reinforce the fact that each individual multiplication and addition specified is performed on the finite field $\mathbf{F}_q$, following the rules specified explicitly in (13.1).

[4]Overbar simply denotes (elementwise) complex conjugate; for LBCs and LTCs, $\bar{\mathbf{u}} = \mathbf{u}$ and $\bar{P} = P$.

Note that $\bar{P}^T$ is of order $k \times (n-k)$, and, of course, that $H^{\perp}_{[n,n-k]_q} V^{\perp}_{[n,n-k]_q} = 0$ (on $\mathbf{F}_q$).

If H and V are the parity-check and basis matrices of an $[n,k,d]_q$ LC, with $HV = 0$ (on $\mathbf{F}_q$), then an **equivalent** LC (with the same n, k, and d) may be generated[5] by taking

$$\tilde{H} = R^T H Q, \quad \tilde{V} = Q^T V S, \quad \tilde{\mathbf{w}} = Q^T \mathbf{w} \qquad (\text{on } \mathbf{F}_q), \tag{13.4}$$

where $Q_{n\times n}$ is a permutation matrix, and each column of $R_{(n-k)\times(n-k)}$ and $S_{k\times k}$ is nonzero and independent (on $\mathbf{F}_q$) from the other columns of R and S, respectively[6]. The permutation matrix Q reorders the rows of V and the corresponding columns of H (i.e., it reorders the data symbols and parity symbols in the LC). Each column of S performs a linear combination of the columns of $(Q^T V)$ to form the corresponding column of $\tilde{V}$, while each column of R (that is, each row of R^T) performs a linear combination of the rows of (HQ) to form the corresponding row of $\tilde{H}$; these modifications by S and R leave the set of valid codewords in the LC unchanged.

A **self-dual** code C is an LC for which the dual code $C^{\perp}$ [see (13.3)] is equivalent to C [see (13.4)].

13.1 Characterizing the minimum Hamming distance d of linear codes

Graphically, the codewords of an $[n,k,d]_2$ LBC amount to a carefully chosen subset of 2^k of the 2^n corners on a single n-dimensional unit hypercube, as illustrated for $n = \{3,\ 4,\ 7,\ 8\}$ in Figures $\{$13.1, 13.2, 13.4, 13.5$\}$, where d quantifies the minimum number of edges (i.e., bits) that differ between any two valid codewords. Further:

- An LC with $d = 2$ is **single error detecting** (**SED**) [see, e.g., Figures 13.1a and 13.2a]. In this case, the sum (on $\mathbf{F}_q$) of the symbols in each transmitted codeword is zero, so if it is assumed that at most one symbol error occurred and this sum is nonzero, then a symbol error in transmission occurred, whereas if this sum is zero, then a symbol error did not occur. However, if a symbol error in transmission occurred, the received (invalid) message is generally equidistant from multiple codewords, so it is not possible to correct the error. Two or more symbol errors generally cause the codeword to be misinterpreted.
- An LC with $d = 3$ is **single error correcting** (**SEC**) [see, e.g., Figures 13.1b and 13.4]. In this case, if it is again assumed that at most one symbol error in transmission occured, then if the received codeword is not a codeword, there is only one codeword that is unit Hamming distance away, so the single symbol error may in fact be *corrected*. Again, two or more symbol errors generally cause the codeword to be misinterpreted.
- An LC with $d = 4$ is **single error correcting and double error detecting** (**SECDED**) [see, e.g., Figures 13.2b and 13.5]. In this case, if a single symbol error occurs, the received codeword will be unit Hamming distance away from a single codeword, and thus single symbol errors can be corrected. On the other hand, if two symbol errors occur, the received codeword is generally Hamming distance 2 away from multiple codewords, so double symbol errors can be detected but *not* corrected. In this case, three or more symbol errors cause the codeword to be misinterpreted.

Such labels and their natural extensions (DEC for $d = 5$, DECTED for $d = 6$, TEC for $d = 7$, TECQED for $d = 8$, 4EC for $d = 9$, 4EC5ED for $d = 10$, 5EC for $d = 11$, 5EC6ED for $d = 12$, etc) may be used to quantify the error correction capability of an LC. Alternatively, if error correction is *not* attempted, then:

- an LC with $d = 2$ is single error detecting, with 2 or more symbol errors generally causing misinterpretation,
- an LC with $d = 3$ is double error detecting, with 3 or more symbol errors generally causing misinterpretation,
- an LC with $d = 4$ is triple error detecting, with 4 or more symbol errors generally causing misinterpretation,

etc. Error correcting codes are useful for a broad range of data transmission and data storage applications in which it is difficult or impossible to request that a corrupted codeword be retransmitted; algorithms which

[5] Though many equivalent codes can be transformed between their various representations via (13.4) [see, e.g., conversions between the systematic, cyclic, and syndrome forms of the $[2^m-1, 2^m-1-m, 3]$ binary Hamming codes discussed in §13.5], not all $[n,k,d]_q$ codes that are equivalent may be transformed between their various forms via (13.4), as their parity bits may be defined differently.

[6] For a permutation matrix, $QQ^T = 0$. (In general, matrix inversion on a finite field $\mathbf{F}_q$ can be performed using Cramer's rule.)

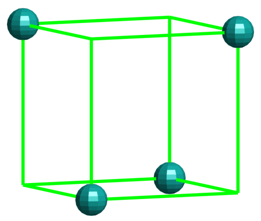
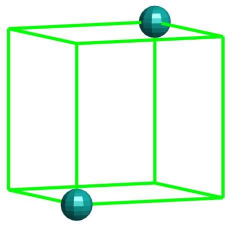

Figure 13.1: Codewords of (left) the (SED) $[3, 2, 2]$ LBC; (right) its dual, the (perfect, SEC) $[3, 1, 3]$ LBC.

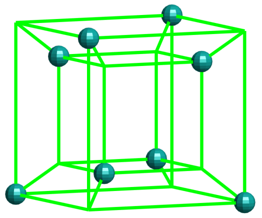
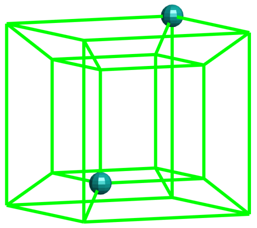

Figure 13.2: Codewords of (left) the (SED) $[4, 3, 2]$ LBC; (right) its dual, the (quasi-perfect, SECDED) $[4, 1, 4]$ LBC.

use such codes for error detection only may be useful when efficient handshaking is incorporated in a manner which makes it easy to request and resend any messages that might be corrupted during transmission.

An $[n, k, d]_q$ LC is said to be **perfect** if, for some integer $t > 0$, each possible n-dimensional q-ary codeword is of Hamming distance t or less from a single codeword (that is, there are no "wasted" vectors that are Hamming distance $t + 1$ or more from the valid codewords, and thus may not be corrected under the assumption that at most t symbol errors have occured). A perfect code has odd $d = 2t + 1 > 1$. A remarkable proof by Tietäväinen (1973) establishes that the *only* nontrivial perfect LCs are the $[(q^r - 1)/(q - 1), (q^r - 1)/(q - 1) - r, 3]_q$ q-ary Hamming codes [§13.5], the $[23, 12, 7]_2$ binary Golay code [§13.9] and the $[11, 6, 5]_3$ ternary Golay code.

An $[n, k, d]_q$ LC is said to be **quasi-perfect** if, for some integer $t > 1$, each possible n-dimensional q-ary codeword is either (a) of Hamming distance $t - 1$ or less from a single codeword, and thus up to $t - 1$ symbol errors may be corrected, or (b) of Hamming distance t from at least one codeword, and thus codewords with t symbol errors may be detected but not necessarily corrected (that is, there are no "wasted" vectors that are Hamming distance $t + 1$ or more from a valid codeword, and thus may not be reconciled under the assumption that at most t symbol errors have occured). A quasi-perfect code has even $d = 2t > 2$; examples include the $[(q^r - 1)/(q - 1) + 1, (q^r - 1)/(q - 1) - r, 4]_q$ extended q-ary Hamming codes [§13.6], the $[24, 12, 8]_2$ extended binary Golay code [§13.9], and the $[12, 6, 6]_3$ extended ternary Golay code.

For a revealing high-level view of this general subject, the "best" (largest d) available $[n, k, d]_2$ LBCs, for a range of k data bits and $r = n - k$ parity bits, are illustrated in Figure 13.3, which plots the best available **data rate** k/n versus k for various values of d, from $d = 2$ (SED) to $d = 12$ (5EC6ED), as highlighted by the broken diagonal lines and highlighted by colors. The $[3, 1, 3]$, $[4, 1, 4]$, $[3, 2, 2]$, and $[4, 3, 2]$ LBCs depicted visually in Figures 13.1 and 13.2, and the $[7, 4, 3]$ and $[8, 4, 4]$ LBCs depicted visually in Figures 13.4 and 13.5, are indicated near the left of Figure 13.3 (in the $k = \{1,\ 2,\ 3,\ 4\}$ columns). It is seen that, as the number of data bits in the packet, k, is increased, codes with improved data rates k/n generally become available for a given degree of error correction capability d. Many of the most notable codes indicted in this figure (before each break in the diagonal lines) are discussed in §13.3-13.9; codes connected by vertical lines are related by **puncturing** the code below; codes connected by diagonal lines are related by **shortening** the code to the upper-right (see §13.10).

The selection of the "best" code for a given purpose actually involves a tradeoff. In addition to maximizing the data rate k/n for a given number of data bits k and a given degree of error correction capability d, the existence of fast algorithms to **code** the message (determining its parity bits), and to **decode** the message (checking for errors, and determining the nearest valid codeword) is also quite important. In this regard, LCs with **cyclic forms** (see §13.2) and **syndrome decoding** strategies (see §13.5.1) are especially valuable.

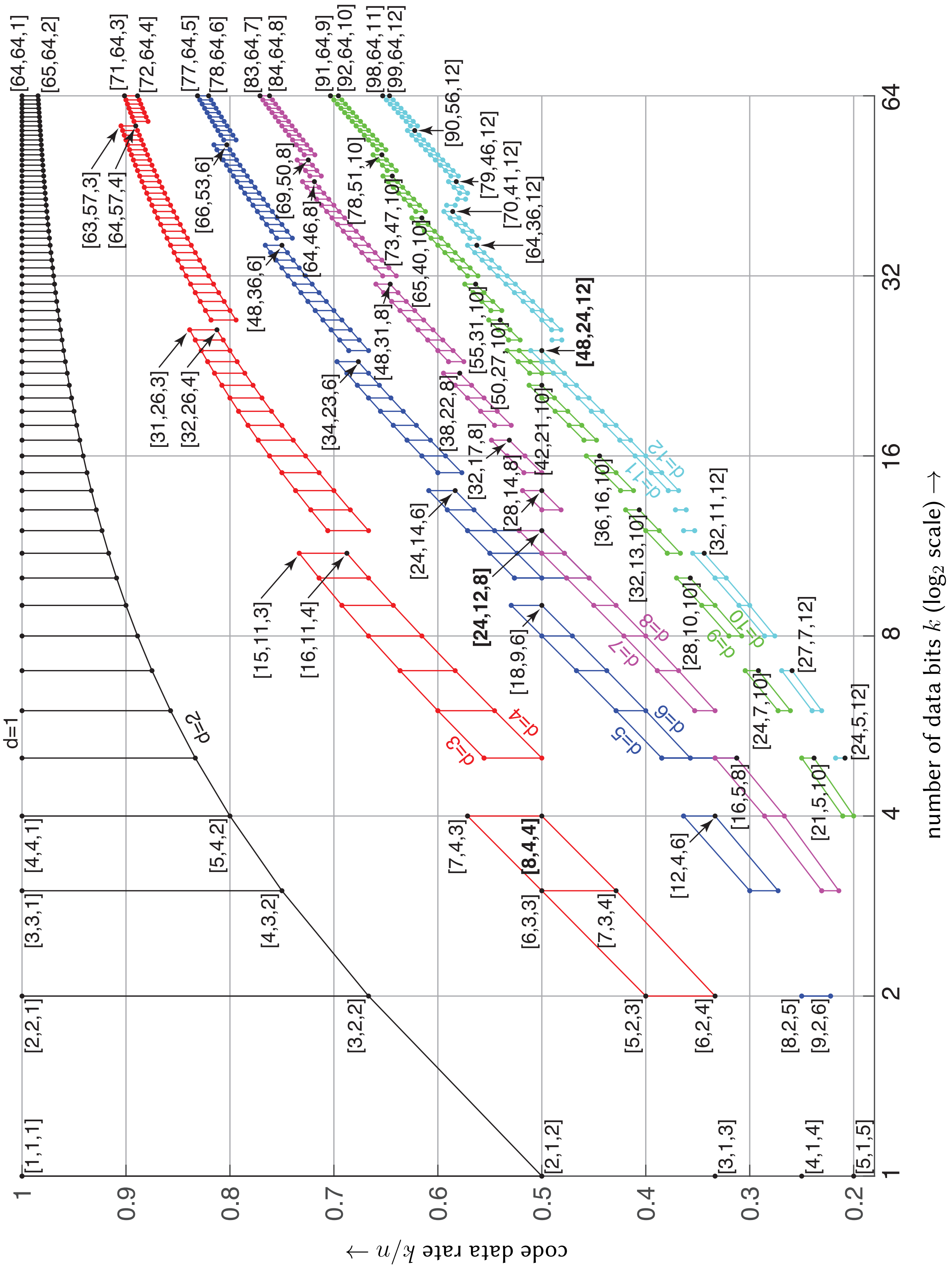

Figure 13.3: The densest known linear binary codes $[n, k, d]_2$ with Hamming distance $1 \leq d \leq 12$.

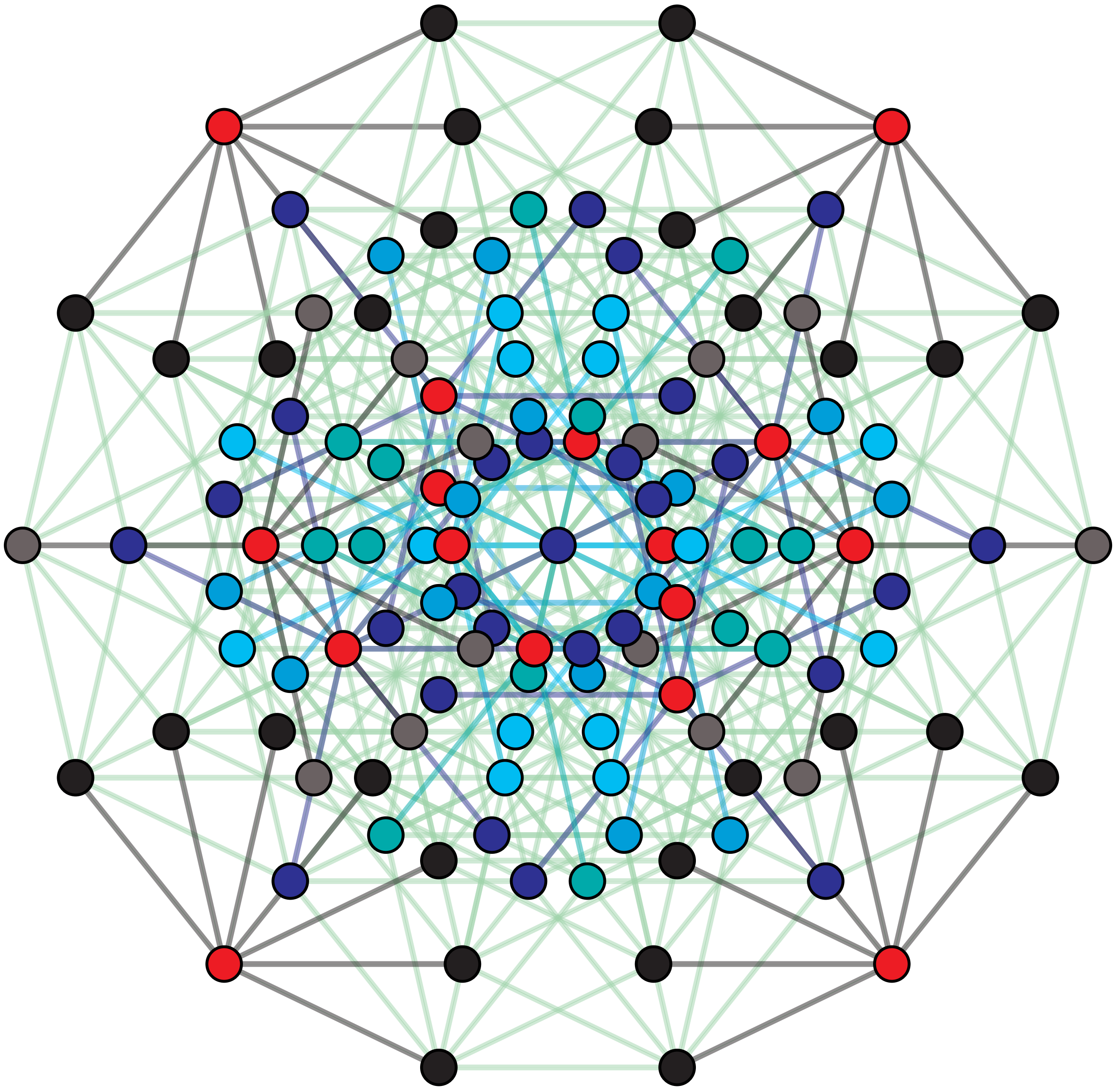

Figure 13.4: The $2^k = 16$ valid codewords (in red) of the (perfect, SEC) $[7, 4, 3]$ LBC (see §13.5 and Figure 1.2a), amongst the $2^n = 128$ possible received messages. Each of the $16 \cdot 7 = 112$ invalid messages is unit Hamming distance from a single valid codeword, thus facilitating single error correction. Note that each valid codeword in the above figure is indeed 3 edges from the nearest valid codeword; in this 2-dimensional orthogonal projection (called a hepteract) of the corners of a 7-dimensional hypercube, some edges overlap. The **data rate** is $k/n = 4/7 = 0.571$ (that is, 4 data bits out of every 7 message bits), which compares favorably to the data rate of 1/3 for the (perfect, SEC) $[3, 1, 3]$ LBC in Figure 13.1b (see Figure 13.3).

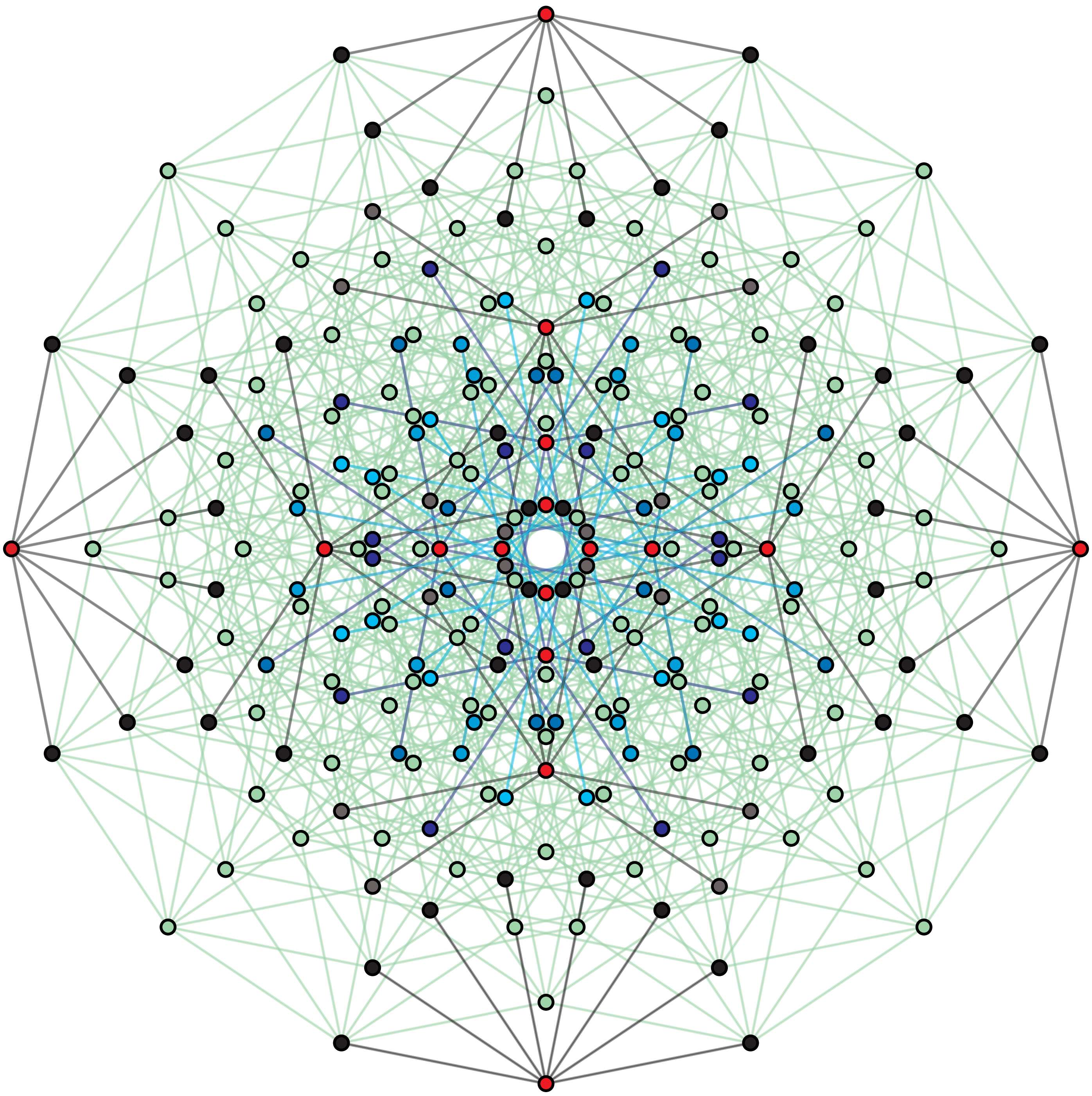

Figure 13.5: The $2^k = 16$ valid codewords (in red) of the (quasi-perfect, SECDED, self-dual) $[8, 4, 4]$ LBC (see §13.6), amongst the $2^n = 256$ possible received messages. Of the 240 invalid messages, $16 \cdot 8 = 128$ messages are each unit Hamming distance from a single valid codeword, thus facilitating single error correction, and the remaining $240 - 128 = 112$ messages (in green) are each Hamming distance 2 from multiple valid codewords, thus facilitating double error detection. Each valid codeword in the above figure is 4 edges from the nearest valid codeword; in this 2-dimensional orthogonal projection (called a octeract) of the corners of a 8-dimensional hypercube, some edges overlap. The data rate is $k/n = 4/8 = 1/2$, which compares favorably to the data rate of $1/4$ for the (quasi-perfect, SECDED) $[4, 1, 4]$ LBC in Figure 13.2b (see Figure 13.3).

13.2 Cyclic form

A cyclic code is an LC that may be written in a **cyclic form** in which the $n \times k$ basis matrix $V^c_{[n,k]_q}$ and the $r \times n$ parity-check matrix $H^c_{[n,k]_q}$, with $n = r + k$, have the form

$$H^c_{[n,k]_q} = \begin{pmatrix} h_k & h_{k-1} & \dots & h_0 & & & 0 \\ & h_k & h_{k-1} & \dots & h_0 & & \\ & & \ddots & \ddots & \ddots & \ddots & \\ 0 & & & h_k & h_{k-1} & \dots & h_0 \end{pmatrix}, \quad V^c_{[n,k]_q} = \begin{pmatrix} v_0 & & & 0 \\ v_1 & v_0 & & \\ \vdots & v_1 & \ddots & \\ v_r & \vdots & \ddots & v_0 \\ & v_r & \ddots & v_1 \\ & & \ddots & \vdots \\ 0 & & & v_r \end{pmatrix}, \tag{13.5a}$$

where $\{h_k, h_0, v_r, v_0\}$ are nonzero with

$$h_k\, v_r + h_0\, v_0 = 0 \quad (\text{on } \mathbf{F}_q), \tag{13.5b}$$

and where [as with the systematic form (13.2)], of course,

$$H^c_{[n,k]_q}\, V^c_{[n,k]_q} = 0 \quad (\text{on } \mathbf{F}_q). \tag{13.5c}$$

A convenient construction which simplifies the math of cyclic LCs is the *cyclic shift* operator z, which is used in a manner akin to the Z-transform analysis of discrete-time linear systems (see §9), with the major difference being that polynomials in z are manipulated here in a cyclic context on $\mathbf{F}^n_q$. That is, arithmetic with polynomials in z and coefficients in $\mathbf{F}_q$ is performed as usual, except that the coefficients of each power of z are combined via the arithmetic rules on $\mathbf{F}_q$ [see (13.1)], and higher powers of z^k are simplified via the *cyclic condition*

$$\boxed{z^n = 1.} \tag{13.6}$$

In the analysis and use of an $[n, k]_q$ cyclic LC, the operator z appears in

the **data polynomials**	$a(z) = a_{k-1}z^{k-1} + a_{k-2}z^{k-2} + \dots + a_1 z + a_0$	(13.7a)
the **basis polynomial**	$v(z) = v_r z^r + v_{r-1}z^{r-1} + \dots + v_1 z + v_0$	(13.7b)
the **codeword polynomials**	$w(z) = w_{n-1}z^{n-1} + w_{n-2}z^{n-2} + \dots + w_1 z + w_0$	(13.7c)
the **received-message polynomials**	$\hat{w}(z) = \hat{w}_{n-1}z^{n-1} + \hat{w}_{n-2}z^{n-2} + \dots + \hat{w}_1 z + \hat{w}_0,$ and	(13.7d)
the **parity-check polynomial**	$h(z) = h_k z^k + h_{k-1}z^{k-1} + \dots + h_1 z + h_0.$	(13.7e)

The use of a cyclic LC to communicate data over a noisy channel is again straightforward:

- **group** the data into vectors $\mathbf{a} \in \mathbf{F}^{\mathbf{k}}_{\mathbf{q}}$, and express as polynomials $a(z)$ with the k elements of $\mathbf{a}$ as coefficients;
- **code** each $a(z)$ as a polynomial $w(z)$ with $h(z)\,w(z) = 0$[7], and express $w(z)$ as a vector $\mathbf{w} \in \mathbf{F}^n_q$;
- **transmit** the corresponding codeword vector $\mathbf{w}$ over the noisy channel;
- **receive** the (possibly corrupted) message $\hat{\mathbf{w}} \in \mathbf{F}^n_q$ on the other end, and express as a polynomial $\hat{w}(z)$;
- **decode** $\hat{w}(z)$ leveraging the parity-check polynomial $h(z)$; i.e., find the most likely codeword polynomial $w(z)$ corresponding to $\hat{w}(z)$, and the data polynomial $a(z)$ [and, thus, the data vector $\mathbf{a}$] that generated it.

[7]One way of accomplishing this is simply by defining $w(z) = a(z)\,v(z)$. In practical implementations, a different convention (which is faster to decode), called the **systematic cyclic form**, is commonly used. With this approach, the codeword polynomial $w(z)$ is defined by taking the first k coefficients of $w(z)$ as the coefficients the data polynomial $a(z)$ [see (13.11)]. The remaining coefficients of $w(z)$, which make up the **parity polynomial** $b(z)$, are then defined such that $h(z)\,w(z) = 0$; (13.12) and (13.13) present two equivalent constructions to compute the required $b(z) = b_{r-1}z^{r-1} + \dots + b_1 z + b_0$ when using this form.

The basis and parity-check polynomials, $v(z)$ and $h(z)$, are constructed by factorization of $(z^n - 1)$ as follows:

$$\boxed{h(z)\,v(z) = [h_k z^k + \ldots + h_1 z + h_0]\,[v_r z^r + \ldots + v_1 z + v_0] = z^n - 1 = 0 \quad (\text{on } \mathbf{F}_q).} \tag{13.8}$$

Note that:

(i) the scalar product of the first row of H^c and the first column of V^c in (13.5a) is precisely the coefficient of z^k in (13.8), once multiplied out, which by construction [see the RHS of (13.8)] is zero;
(ii) the scalar product of the first row of H^c and the last column of V^c is the coefficient of z^1 in (13.8) [that is, $h_1\,v_0 + h_0\,v_1$ (on $\mathbf{F}_q$)], which by construction is also zero;
(iii) ...analogously, the scalar product of any row of H^c with any column of V^c is simply the coefficient of z^j in (13.8), for $1 \le j \le k$, which by construction is zero.

Thus, the condition (13.8) enforces (13.5c) [that is, $H^c\,V^c = 0$]. Applying the cyclic condition (13.6), the coefficient of z^0 in (13.8), $h_k\,v_r + h_0\,v_0$, is also zero, and thus (13.5b) is enforced. Indeed, it follows from (13.8) and (13.6) that

$$[z^i\,h(z)]\,[z^j\,v(z)] = z^{i+j}\,[h(z)\,v(z)] = z^{i+j}\,[z^n - 1] = 0 \quad (\text{on } \mathbf{F}_q) \tag{13.9}$$

for any integer i and j and thus, performing arithmetic on $\mathbf{F}_q$ and computing in the cyclic setting with $z^n = 1$, interpreting z as a shift operator, it follows that *any cyclic shift of the parity check polynomial $h(z)$ is orthogonal (on $\mathbf{F}_q^n$) to any cyclic shift of the basis polynomial $v(z)$*. Since it follows that all cyclic shifts of the basis vectors are themselves valid codewords, and noting that all codewords are formed by linear combinations of the basis vectors, it follows that *any cyclic shift of a valid codeword is itself also a valid codeword*, which is indeed how the class of cyclic codes gets its name.

13.2.1 Constructing cyclic codes

Cyclic codes are thus constructed via factorizations of the form (13.8). One factorization of $(z^n - 1)$ on $\mathbf{F}_q$, which exists for any n and q, is

$$z^n - 1 = (z-1)(z^{n-1} + z^{n-2} + \ldots + z + 1); \tag{13.10}$$

this leads to the single parity check code $[n, n-1, 2]_q$ (see §13.3) if one takes $v(z) = (z-1)$ and $h(z)$ equal to the rest, and to the repetition code $[n, 1, n]_q$ (see §13.4) if one takes $h(z) = (z-1)$ and $v(z)$ equal to the rest.

If q is not prime, developing other factorizations of $(z^n - 1)$ over $\mathbf{F}_q$ is a bit delicate, as $(z^n - 1)$ does not, in general, factor into the product of unique irreducible forms in this case. As an example, two irreducible factorizations of $(z^5 - 1)$ over $\mathbf{F}_4$ are listed in Table 13.1.

For prime q, however, the development of other factorizations of $(z^n - 1)$ over $\mathbf{F}_q$ is significantly more straightforward, as $(z^n - 1)$ may be factored into the product of unique irreducible forms in this case. A few such factorizations for various values of n are listed in Table 13.2 for $q = 3$ (in which "-1" $= 2$), and in Table 13.3 for $q = 2$ (in which "-1" $= 1$); others are easily found using Matlab or Mathematica. Noting (13.8), these unique irreducible factors of $(z^n - 1)$ may be grouped in different ways to form $v(z)$ and $h(z)$. Many cyclic codes may be constructed via such an approach, only some of which have a favorable minimum distance d and an available simple error correction scheme. A few such codes, with $q = 2$, are listed in Table 13.4.

$z^5 - 1 = (z+1)(z^4 + z^3 + z^2 + z + 1)$
$z^5 - 1 = (z^2 + \omega z + 1)(z^3 + \omega z^2 + \omega z + 1)$

Table 13.1: Two (nonunique!) irreducible factorizations of $(z^5 - 1)$ over $\mathbf{F}_4$ [see (13.1)], with $\omega = (-1 + \mathrm{i}\sqrt{3})/2$.

$$\boxed{\begin{aligned} z^4-1&=(z+2)(z+1)(z^2+1)\\ z^{11}-1&=(z+2)(z^5+2z^3+z^2+2z+2)(z^5+z^4+2z^3+z^2+2)\\ z^{13}-1&=(z+2)(z^3+2z+2)(z^3+z^2+2)(z^3+z^2+z+2)(z^3+2z^2+2z+2)\end{aligned}}$$

Table 13.2: Some unique irreducible factors of (z^n-1) over $\mathbf{F}_3$ [see (13.1)] for various values of n.

$$\boxed{\begin{aligned} z^5-1&=(z+1)(z^4+z^3+z^2+z+1)\\ z^7-1&=(z+1)(z^3+z+1)(z^3+z^2+1)\\ z^{15}-1&=(z+1)(z^2+z+1)(z^4+z+1)(z^4+z^3+1)(z^4+z^3+z^2+z+1)\\ z^{23}-1&=(z+1)(z^{11}+z^9+z^7+z^6+z^5+z+1)(z^{11}+z^{10}+z^6+z^5+z^4+z^2+1)\end{aligned}}$$

Table 13.3: Some unique irreducible factors of (z^n-1) over $\mathbf{F}_2$ [see (13.1)] for various values of n.

$[n,k,d]_q$ code	r	$v(z)$	$h(z)$
$[n,n-1,2]_2$	1	$z+1$	$z^{n-1}+z^{n-2}+\ldots+z+1$
$[7,4,3]_2$	3	z^3+z+1	z^4+z^2+z+1
$[15,11,3]_2$	4	z^4+z+1	$z^{11}+z^8+z^7+z^5+z^3+z^2+z+1$
$[31,26,3]_2$	5	z^5+z^2+1	Matlab: `mod(deconv([1 zeros(1,30) 1],[1 0 0 1 0 1]),2)`
$[63,57,3]_2$	6	z^6+z+1	`mod(deconv([1 zeros(1,62) 1],[1 0 0 0 0 1 1]),2)`
$[127,120,3]_2$	7	z^7+z^3+1	`mod(deconv([1 zeros(1,126) 1],[1 0 0 0 1 0 0 1]),2)`

Table 13.4: Some small binary cyclic codes, defined such that $v(z)\,h(z)=(z^n-1)$ over $\mathbf{F}_2$, with $r=n-k$.

13.2.2 Cyclic coding

As mentioned in Footnote 7 a couple of pages back, by convention, practical implementations of cyclic codes usually shift the k data symbols of $a(z)$ to one end of the codeword $w(z)$, for example

$$\begin{aligned} w(z)&=z^r a(z)+b(z)\\ &=a_{k-1}z^{n-1}+\ldots+a_1z^{r+1}+a_0z^r+b_{r-1}z^{r-1}+\ldots+b_1z+b_0\\ &=w_{n-1}z^{n-1}+\ldots+w_1z^{n-k+1}+w_{n-k}z^{n-k}+w_{n-k-1}z^{n-k-1}+\ldots+b_1z+b_0,\end{aligned}\tag{13.11}$$

and then determine the parity polynomial $b(z)$ within $w(z)$ in a manner such that $h(z)\,w(z)=0$.

For $k\lesssim r$ [that is, short $h(z)$ and long $v(z)$], a recursive approach may be used to determine the parity symbols $b(z)$ from $h(z)$ and $a(z)$. By (13.8), (13.9), and (13.6), and the fact that each valid codeword polynomial $w(z)$ is itself a linear combination of the basis polynomials $v(z)$, we have

$$h(z)\,w(z)\triangleq u_{n-1}z^{n-1}+u_{n-2}z^{n-2}+\ldots+u_1z+u_0=0.$$

Initializing the first k symbols of $w(z)$ as a_{k-1} through a_0 as in (13.11), noting that $h_k\neq 0$, the remaining symbols of $w(z)$, in $b(z)$, may be determined from the resulting convolution formulae for u_{n-1} through u_k as follows:

$$\begin{aligned} u_{n-1}&=h_0w_{n-1}+\ldots+h_kw_{n-k-1}=0 &\Rightarrow\quad & b_{r-1}=w_{n-k-1}=-[h_0w_{n-1}+\ldots+h_{k-1}w_{n-k}]/h_k,\\ u_{n-2}&=h_0w_{n-2}+\ldots+h_kw_{n-k-2}=0 &\Rightarrow\quad & b_{r-2}=w_{n-k-2}=-[h_0w_{n-2}+\ldots+h_{k-1}w_{n-k-1}]/h_k,\\ &\vdots & & \qquad\vdots\\ u_k&=h_0w_k+\ldots+h_kw_0=0 &\Rightarrow\quad & b_0=w_0=-[h_0w_k+\ldots+h_{k-1}w_1]/h_k.\end{aligned}\tag{13.12}$$

For $r < k$ [that is, short $v(z)$ and long $h(z)$], a direct polynomial division approach to determine the parity symbols from $v(z)$ and $a(z)$ is more efficient. This may by achieved by writing the shift of the data symbols as some multiple of the basis polynomial $v(z)$ plus a remainder $t(z)$:

$$z^r a(z) = q(z)v(z) + t(z) \quad \Leftrightarrow \quad [z^r a(z)] \bmod v(z) = t(z).$$

Note that $v_r \neq 0$. Calculating $t(z)$ as shown above, taking $b(z) = -t(z)$ and moving this term to the LHS in the above expression, it is seen [cf. (13.11)] that

$$z^r a(z) + b(z) = q(z)v(z) \triangleq w(z),$$

verifying that the $w(z)$ so generated is a valid codeword polynomial, as it is a multiple of $v(z)$, and thus $h(z)\,w(z) = [h(z)\,v(z)]\,q(z) = 0$. Computation of $t(z)$ via polynomial division is straightforward[8]:

```
$t(z) = z^r a(z)$   % Initialize $t(z)$ as the $(n-1)$'th order polynomial $z^r a(z)$
for $i = n : -1 : r+1$         % Zero the $i$'th coefficient in $t(z)$ by subtracting multiples of $v(z)$
    $t(z) = t(z) - z^{i-r-1}\, v(z)\, [t_i / v_r]$       % Note: arithmetic must be performed on $\mathbf{F}_q$!
end                      % Returns remainder $t(z)$ when $v(z)$ is divided into $z^r a(z)$
```
(13.13)

A binary ($q = 2$) implementation of (13.13) is given in Algorithm 13.1. Defining the $(r-1)$'th order parity polynomial $b(z) = -t(z)$ computed via this approach, the remaining symbols of $w(z)$ in (13.11) are determined.

13.2.3 Cyclic decoding

Using cyclic codes (13.5) constructed in the form (13.11) [leveraging (13.12) or (13.13) to compute $b(z)$], received codewords are easy to check for errors, and trivial to decode. To check for errors, two approaches are possible:

a) multiply each of the $r = n - k$ rows of H [which by (13.5a) are each simple shifts of the k symbols of the parity check polynomial $h(z)$, padded with zeros] times each received message $\hat{\mathbf{w}}$ [corresponding to the n symbols of the polynomial $\hat{w}(z)$], noting that the received message is error free [i.e., $\hat{w}(z) = w(z)$] if each of these scalar products is zero; or

b) recompute the check bits from the data bits in the received message $\hat{w}(z)$ using (13.13), and compare with the check bits in $\hat{w}(z)$; the received message is error-free if they match.

To decode, note by (13.11) that the k symbols of the data polynomial $a(z)$ are simply the first k symbols of $w(z)$.

The polynomial multiplications and divisions involved in the cyclic coding and decoding algorithms described above may be calculated efficiently in either an application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA), as discussed in §1.5.3, in which repeated computations with shifted data may be performed, in parallel, remarkably quickly. The reduced storage associated with the vector representation of the basis matrix and the parity-check matrix in cyclic form help to facilitate such implementations.

An extended summary of the subject of linear binary ($q = 2$), ternary ($q = 3$), and quaternary ($q = 4$) codes with a broad range of k and d (including a summary of the longer turbo codes), and the connection between these codes and n-dimensional lattice packings, is provided in *RP*, and the references therein. The remainder of this chapter focuses on a few families of **linear binary codes** (LBCs) with $q = 2$ and rather small k and d (see Figure 13.3), usually denoted simply as $[n, k]$ or $[n, k, d]$ (dropping the q subscript), which are useful in embedded applications. We also, in §13.11, discuss the use of binary **cyclic redundancy check** (CRC) codes, defined again by a basis polynomial $v(z)$ as laid out above, but for which, in application, the number of data bits k to be transmitted is *variable*, based on the communication needs of the system and the noise on the transmission line, and is determined on the fly.

[8]Note that the quotient $q(z)$ of this polynomial division is not actually needed, just the remainder $t(z)$.

13.3 Binary single parity check codes

The $[n, n-1, 2]$ *binary*[9] *single parity-check codes* are SED, and include $[2, 1, 2]$ (self-dual), $[3, 2, 2]$, $[4, 3, 2]$, etc. Using such a code, for each $k = n-1$ data bits to be sent, a single ($r = 1$) *parity bit* is generated such that the sum (on $\mathbf{F}_2$) of the data bits and parity bit is 0; when decoding, an error is flagged if this sum (on $\mathbf{F}_2$) is 1. The $[3, 2, 2]$ code illustrated in Figure 13.1a, with $P = \begin{pmatrix} 1 & 1 \end{pmatrix}$ [see (13.2)] and $2^k = 4$ valid codewords, is given by

$$H_{[3,2,2]} = \begin{pmatrix} \underbrace{1 \quad 1}_{P} & 1 \end{pmatrix}, \quad V_{[3,2,2]} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad W_{[3,2,2]} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}. \tag{13.14}$$

The $[4, 3, 2]$ code illustrated in Figure 13.2a, with $P = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$ and $2^k = 8$ valid codewords, is given by

$$H_{[4,3,2]} = \begin{pmatrix} \underbrace{1 \quad 1 \quad 1}_{P} & 1 \end{pmatrix}, \quad V_{[4,3,2]} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \quad W_{[4,3,2]} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}. \tag{13.15}$$

Other binary single parity-check codes have a partity submatrix P [see (13.2)] of similar form (a row of 1's).

In cyclic form [see (13.5)], binary single parity-check codes are generated by $h(z) = z^{n-1} + z^{n-2} + \ldots + z + 1$ and $v(z) = z + 1$. For example, for $n = 4$, the H and V matrices for a binary single parity check code in cyclic form is given below left; these two matrices may be transformed back to the systematic form in (13.15) via (13.4) by taking $R = I$, $Q = I$, and (for any n) S as defined below right:

$$H^c_{[4,3,2]} = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}, \quad V^c_{[4,3,2]} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}; \qquad S = \begin{pmatrix} 1 & & 0 \\ \vdots & \ddots & \\ 1 & \ldots & 1 \end{pmatrix}. \tag{13.16}$$

A single parity-check code (binary or otherwise), with $d = 2$, can detect but not correct an error in an unknown position. However, it can correct an **erasure** (i.e., the loss of data from a known position). A common application of this capability is in a **RAID 5** system (i.e., *Redundant Array of Independent Disks*) for data storage. In such a system, data is striped across n hard disk drives using a single parity check code; if any single drive fails (which is unfortunately fairly common, relative to the expected shelf life of certain data), the failed drive can be swapped out, and data on it can be recovered simply by achieving parity with the other disks.

13.4 Binary repetition codes

The dual [see (13.3b)] of the binary single parity-check codes are the $[n, 1, n]$ *binary repetition codes*, which include $[2, 1, 2]$ (SED, self-dual), $[3, 1, 3]$ (SEC, perfect), $[4, 1, 4]$ (SECDED, quasi-perfect), $[5, 1, 5]$ (DEC, perfect), etc. This family of codes repeats a single ($k = 1$) data bit n times (i.e., $r = n - 1$); when decoding, if an error is detected (i.e., if $H\hat{\mathbf{w}} \neq 0$), one may find which of the two valid codewords that the received message is closest to simply by majority vote. The $[3, 1, 3]$ code illustrated in Figure 13.1b, with $P = \begin{pmatrix} 1 & 1 \end{pmatrix}^T$, is given by

$$H_{[3,1,3]} = \begin{pmatrix} 1 & 1 & 0 \\ \underbrace{1}_{P} & 0 & 1 \end{pmatrix}, \quad V_{[3,1,3]} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad W_{[3,1,3]} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}. \tag{13.17}$$

The $[4, 1, 4]$ code illustrated in Figure 13.2b, with $P = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T$, is given by

$$H_{[4,1,4]} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \underbrace{1}_{P} & 0 & 0 & 1 \end{pmatrix}, \quad V_{[4,1,4]} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad W_{[4,1,4]} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}. \tag{13.18}$$

[9]As suggested previously, for the remainder of §13, the q subscript is suppressed for notational clarity when $q = 2$.

Other repetition codes have a partity submatrix P of similar form (a column of 1's); note that $P_{[n,1,n]} = P^T_{[n,n-1,2]}$.

In cyclic form, binary repetition codes are generated by $h(z) = z + 1$ and $v(z) = z^{n-1} + z^{n-2} + \ldots + z + 1$. For example, for $n = 4$, the H and V matrices for a binary single parity check code in cyclic form is given below left; these two matrices may be transformed back to the systematic form in (13.18) via (13.4) by taking $Q = I$, $S = I$, and (for any n) R as defined below right:

$$H^c_{[4,1,4]} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}, \quad V^c_{[4,1,4]} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}; \qquad R = \begin{pmatrix} 1 & \ldots & 1 \\ & \ddots & \vdots \\ 0 & & 1 \end{pmatrix}. \tag{13.19}$$

13.5 Binary Hamming codes

The $[2^r - 1, 2^r - (r+1), 3]$ *binary Hamming codes* are perfect and SEC, and include $[3,1,3]$, $[7,4,3]$, $[15,11,3]$, $[31,26,3]$, $[63,57,3]$, $[127,120,3]$, etc. For a given $k = 2^r - (r+1)$ data bits to be transmitted, each of the r parity bits is generated such that the sum (on $\mathbf{F}_2$) of a particular subset of the data bits plus that parity bit is 0. The parity-check matrix H of a binary Hamming code has as columns all nonzero binary vectors of length $r = n - k$; when expressed in systematic form, the r columns of H corresponding to the identity matrix are shifted to the end, and the remaining k columns of H, in arbitrary order (often, binary order is used[10]), make up the parity submatrix P. For example, the $[7,4,3]$ code (see Figure 13.4), with four data bits $\{a_1, a_2, a_3, a_4\}$, three parity bits $\{a_1, a_2, a_3\}$, and $2^k = 16$ valid codewords $\mathbf{w}^i$, is given in systematic form by

$$H_{[7,4,3]} = \begin{pmatrix} \underbrace{\begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{matrix}}_{P} & \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix} \end{pmatrix}, \quad V_{[7,4,3]} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}, \quad \mathbf{w} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}. \tag{13.20a}$$

$$W_{[7,4,3]} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

Other binary Hamming codes are built in the same manner.

In cyclic form, $[2^r - 1, 2^r - (r+1), 3]$ binary Hamming codes are generated by selecting $v(z)$ as an r'th-order polynomial that is a root of $(z^n - 1)$, where $n = 2^r - 1$, and by taking $h(z) = (z^n - 1)/v(z)$ on $\mathbf{F}_2$; for $r = 3$ through 7, $v(z)$ and $h(z)$ are listed in Table 13.4. For example, the $[7,4,3]$ code may be written in cyclic form (13.5) by taking $v(z) = z^3 + z + 1$ and $h(z) = z^4 + z^2 + z + 1$, and thus H and V take the form given below:

$$H^c_{[7,4,3]} = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}, \quad V^c_{[7,4,3]} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{13.20b}$$

[10] In (13.20a), binary order using big endian convention on the individual bits is used on the columns of the parity submatrix P, with the msb in the top row and the lsb in the bottom row; little endian convention may also be used [see §1.1.2 for further discussion].

By defining Q, R, and S in (13.4) such that

$$Q^c = \begin{pmatrix} 0&0&1&0&0&0&0\\ 1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&1&0&0&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1 \end{pmatrix}, \qquad R^c = \begin{pmatrix} 1&1&0\\ 0&1&1\\ 0&0&1 \end{pmatrix}, \qquad S^c = \begin{pmatrix} 0&0&1&0\\ 1&0&1&0\\ 1&0&1&1\\ 1&1&0&1 \end{pmatrix},$$

it is seen that $H_{[7,4,3]} = (R^c)^T H^c_{[7,4,3]} Q^c$ and $V_{[7,4,3]} = (Q^c)^T V^c_{[7,4,3]} S^c$, thus demonstrating the equivalence of these two LBCs. In fact, it may be shown (see [Hill, 1986]) that Q, R, and S may always be found such that the cyclic and systematic forms of any $[2^r-1, 2^r-(r+1), 3]$ LBC may be related by (13.4).

A binary Hamming code, with $d=3$, can correct a single error in an unknown position (see §13.5.1). However, it can correct up to *two* erasures (cf. §13.3). A common application of this capability is in a **RAID 6** system for storage of large amounts of critical data. In such a system, data may be striped across n hard disk drives using a binary Hamming code; if any single drive fails, the data on it can be recovered using an appropriate parity check equation (that is, one of the parity check equations that takes that bit into account). If (while rebuilding the information on that disk, which might take a while) a *second* drive fails, then two useful equations may be derived (by linear combination on $\mathbf{F}_q^n$) from the r parity check equations: one that takes failed disk A into account but not failed disk B, and one that takes failed disk B into account but not failed disk A. By restoring parity in these two derived equations, the information on *both* drives may be rebuilt.

13.5.1 Syndrome based error correction of binary Hamming codes

The $r = n-k$ parity bits of a $[2^r-1, 2^r-(r+1), 3]$ binary Hamming code may be used in a remarkably simple fashion to determine not only *whether* or not a received message $\hat{\mathbf{w}}$ has a single bit error (which is true, as usual, if $H\hat{\mathbf{w}} \neq 0$) but if it does, *which* bit contains the error. To see this, consider a code equivalent to a binary Hamming code in systematic form, but permuted such that the columns of the modified parity check matrix H^s appear in binary order. For example, the $[7,4,3]$ binary Hamming code (13.20a) may be permuted into this non-systematic order (where superscript s denotes syndrome form) as

$$H^s_{[7,4,3]} = \begin{pmatrix} 0&0&0&1&1&1&1\\ 0&1&1&0&0&1&1\\ 1&0&1&0&1&0&1 \end{pmatrix}, \quad V^s_{[7,4,3]} = \begin{pmatrix} 1&1&0&1\\ 1&0&1&1\\ 1&0&0&0\\ 0&1&1&1\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&1 \end{pmatrix}, \quad \mathbf{w}^s = \begin{pmatrix} b_3\\ b_2\\ d_1\\ b_1\\ d_2\\ d_3\\ d_4 \end{pmatrix}. \tag{13.20c}$$

By examining the relation $\mathbf{w} = Q^s\mathbf{w}^s$, it is seen immediately that the $[7,4,3]$ code defined in systematic form in (13.20a) may be transformed into syndrome form in (13.20c) using (13.4) with a permutation matrix of

$$Q^s = \begin{pmatrix} 0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1\\ 0&0&0&1&0&0&0\\ 0&1&0&0&0&0&0\\ 1&0&0&0&0&0&0 \end{pmatrix} \quad \Rightarrow \quad H^s_{[7,4,3]} = H_{[7,4,3]}Q^s, \quad V^s_{[7,4,3]} = (Q^s)^T V_{[7,4,3]}, \quad \mathbf{w}^s = (Q^s)^T\mathbf{w}, \tag{13.21}$$

thus demonstrating its equivalence. Note that each row of H^s so constructed adds to 0 (on $\mathbf{F}_2$), and of course that each row of H^s is orthogonal (on $\mathbf{F}_2$) to each column of V^s. Now define the product (on $\mathbf{F}_2$) of the matrix

H^s times any (possibly corrupted) received message $\hat{\mathbf{w}}^s$, arranged in the corresponding order (see above), as the *syndrome vector*

$$\mathbf{s} = H^s \hat{\mathbf{w}}^s = H^s (Q^s)^T \hat{\mathbf{w}} = H\hat{\mathbf{w}}. \tag{13.22}$$

If $\mathbf{s} = 0$, the message $\hat{\mathbf{w}}^s$ is interpreted as being error free (i.e., $\hat{\mathbf{w}}^s = \mathbf{w}^s$). If $\mathbf{s} \neq 0$, we may interpret $\mathbf{s}$ as an r-bit binary representation of a number called the *syndrome*, denoted s, of the received message $\hat{\mathbf{w}}^s$. Remarkably, as a direct result of the structure of H^s used in this construction [see, e.g., (13.20c)], the syndrome s identifies precisely which bit of the n-bit received message, $\hat{\mathbf{w}}^s$, must be flipped in order to determine the nearest codeword, thereby performing single error correction (SEC). This correction may be written as a correction vector $\mathbf{e}^s$, which is taken as 1 in the s element and zero in all others.

In practice, the systematic form [see, e.g., (13.20a)] of a binary Hamming code may still be preferred to code and check each data vector $\mathbf{d}$, due to its simplicity in decoding, thus transmitting a message $\mathbf{w} = V\mathbf{d}$ and receiving a corresponding (possibly corrupted) message $\hat{\mathbf{w}}$. In this case, by (13.22), the syndrome s is calculated from the received message $\hat{\mathbf{w}}$ using the original systematic form [in (13.2)] of the parity check matrix, H. The interpretation of the corresponding syndrome s, however, remains the same: that is, as identifying, with $\mathbf{e}^s$, which bit of received message $\hat{\mathbf{w}}^s$, written in syndrome form [see, e.g., (13.20c)], must be flipped to perform single error correction. This interpretation must thus be permuted (via $\mathbf{e} = P\mathbf{e}^s$) to determine a correction vector $\mathbf{e}$ for the corresponding received message $\hat{\mathbf{w}}$ written in systematic form.

Alternatively, (and, in many cases, faster, as discussed in the second paragraph of §13.2.3) the cyclic form (13.5a) of a binary Hamming code may be preferred to code and check each data vector $\mathbf{d}$. If the received message is found to have an error (that is, if $H^c\hat{\mathbf{w}}^c \neq 0$), one may simply permute the received message (in cyclic form) to syndrome form form via $\hat{\mathbf{w}}^s = Q^{sc}\hat{\mathbf{w}}^c$, where $Q^{sc} = (Q^s)^T(Q^c)^T$ is built from the permutation matrices Q^s and Q^c discussed above, compute the syndrome $\mathbf{s} = H^s\hat{\mathbf{w}}^s$ and the corresponding correction vector $\mathbf{e}^s$ in syndrome form as discussed above, then permute this correction vector back to cyclic form via $\mathbf{e}^c = (Q^{sc})^T\mathbf{e}^s$. In fact, these vector permutations can be hard wired into the implementation of the error correction code itself, and thus may be performed at essentially zero computational cost.

13.6 Extended binary Hamming codes

The $[2^{r-1}, 2^{r-1} - r, 4]$ *extended binary Hamming codes* are quasi-perfect and SECDED, and include $[4,1,4]$, $[8,4,4]$ (self-dual), $[16,11,4]$, $[32,26,4]$, $[64,57,4]$, $[128,120,4]$, etc. These codes may be formed simply as binary Hamming codes (see §13.5) with an additional overall parity bit b_p; that is, with an extra row in V such that each column of V adds to zero, and an extra row in H that checks the overall parity of the received message. To illustrate, the $[8,4,4]$ code (see Figure 13.5) is given in syndrome form by

$$H^s_{[8,4,4]} = \begin{pmatrix} 0&0&0&1&1&1&1&0\\ 0&1&1&0&0&1&1&0\\ 1&0&1&0&1&0&1&0\\ 1&1&1&1&1&1&1&1 \end{pmatrix}, \quad V^s_{[8,4,4]} = \begin{pmatrix} 1&1&0&1\\ 1&0&1&1\\ 1&0&0&0\\ 0&1&1&1\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&1\\ 1&1&1&0 \end{pmatrix}, \quad \mathbf{w}^s = \begin{pmatrix} b_3\\ b_2\\ d_1\\ b_1\\ d_2\\ d_3\\ d_4\\ b_p \end{pmatrix}. \tag{13.23a}$$

Other extended binary Hamming codes may be constructed similarly, by extending a $[2^r - 1, 2^r - (r+1), 3]$ binary Hamming code in syndrome form with an extra row in V such that each column of V adds to zero, and an extra row in H that checks the overall parity of the received message.

The parity check of each received message is block decomposed into the syndrome $\mathbf{s}$, defined exactly as in (13.22) [neglecting the new parity bit b_p], and the overall parity calculation p such that

$$\begin{bmatrix} \mathbf{s} \\ p \end{bmatrix} = H^s \hat{\mathbf{w}}^s \quad (\text{on } \mathbf{F}_q).$$

Assuming no more than two bit errors, the syndrome $\mathbf{s}$ and overall parity p are interpreted as follows:

- If $\mathbf{s} = 0$ and $p = 0$, the received message is error free.
- If $\mathbf{s} \neq 0$ and $p = 0$, the received message has two bit errors, which can not be uniquely corrected.
- If $p = 1$, the received message has a single bit error, which if $\mathbf{s} = 0$ is simply in the parity bit, and if $\mathbf{s} \neq 0$ can be located within the rest of the message using $\mathbf{s}$, precisely as laid out in §13.5.1.

This algorithm thus efficiently performs single error correction and double error detection (SECDED).

Note that the syndrome form (with additional overall parity) of an extended binary Hamming code, as in (13.23a), can be converted back to systematic form by replacing the last row of H^s with the sum (on $\mathbf{F}_q$) of all of the rows of H^s, then permuting the first $(n-1)$ columns of H^s and the first $(n-1)$ rows of V^s consistent with §13.5.1. For example, using Q^s as defined in (13.21), the $[8,4,4]$ extended binary Hamming code given in (13.23a) may be transformed back to systematic form via (13.4) as follows

$$\bar{Q} = \begin{pmatrix} (Q^s)^T & 0 \\ 0 & 1 \end{pmatrix}, \quad \bar{R} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \Rightarrow \quad H_{[8,4,4]} = \bar{R}^T H^s_{[8,4,4]} \bar{Q}, \quad V_{[8,4,4]} = \bar{Q}^T V^s_{[8,4,4]}, \quad \mathbf{w} = \bar{Q}^T \mathbf{w}^s,$$

which gives

$$H_{[8,4,4]} = \begin{pmatrix} \underbrace{\begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}}_{P} & \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \end{pmatrix}, \quad V_{[8,4,4]} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}. \tag{13.23b}$$

13.7 Binary simplex codes

The dual of the binary Hamming codes are the $[2^k - 1, k, 2^{k-1}]$ *binary simplex codes*, which include $[3,2,2]$ (SED), $[7,3,4]$ (SECDED), $[15,4,8]$ (TECQED), $[31,5,16]$ (5EC6ED), etc. The codewords of these LCs form a simplex with 2^k vertices in $2^k - 1$ dimensions; this simplex is regular in the $[3,2,2]$ case, but irregular in the other cases. The $[3,2,2]$ code is illustrated in Figure 13.1a; the $[7,3,4]$ code is given by

$$H_{[7,3,4]} = \begin{pmatrix} \underbrace{\begin{matrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{matrix}}_{P} & \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \end{pmatrix}, \quad V_{[7,3,4]} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}. \tag{13.24}$$

Other binary simplex codes have a partity submatrix given similarly by the transpose of the corresponding binary Hamming code.

13.8 Binary biorthogonal codes

The dual of the extended binary Hamming codes are the $[2^m, m+1, 2^{m-1}]$ *binary biorthogonal codes* (a.k.a. *Hadamard codes*), and include $[4,3,2]$ (SED; see §13.3), $[8,4,4]$ (SECDED, quasi-perfect, self-dual; see §13.6), $[16,5,8]$ (TECQED), $[32,6,16]$ (5EC6ED), etc. The $[32,6,16]$ code was used on the Mariner 9 spacecraft. The codewords of these LCs are mutually orthogonal [that is, $\mathbf{w}^i \cdot \mathbf{w}^j = 0$ (on $\mathbf{F}_2$) for $i \neq j$]. The binary biorthogonal codes each have a partity submatrix that is simply the transpose of the parity submatrix of the corresponding extended binary Hamming code.

13.9 Binary quadratic residue codes

The $[n, (n+1)/2, d]$ *binary quadratic residue codes* are defined for all prime n for which there exists an integer $1 < x < n$ such that $x^2 = 2$ (mod n) [equivalently, for all prime n of the form $n = 8m \pm 1$ where m is an integer], and include $[7,4,3]$ (SEC, perfect; see §13.5), $[17,9,5]$ (DEC), $[23,12,7]$ (TEC, perfect, a.k.a. the *binary Golay code*), $[31,16,7]$ (TEC), $[41,21,9]$ (4EC), $[47,24,11]$, etc. Adding an overall parity bit to these codes, the $[n_0+1, (n_0+1)/2, d+1]$ *extended binary quadratic residue codes* include $[8,4,4]$ (SECDED, quasi-perfect, self-dual; see §13.6), $[18,9,6]$ (DECTED), $[24,12,8]$ (TECQED, quasi-perfect, self-dual, a.k.a. the *extended binary Golay code*), $[32,16,8]$ (TECQED), $[42,21,10]$ (4EC5ED), $[48,24,12]$ (5EC6ED, self-dual), etc. The $[24,12,8]$ extended binary Golay code, used by the Voyager 1 & 2 spacecraft, is given by

$$H_{[24,12,8]} = \begin{bmatrix} P_{12\times 12} & I_{12\times 12} \end{bmatrix}, \quad V_{[24,12,8]} = \begin{bmatrix} I_{12\times 12} \\ P_{12\times 12} \end{bmatrix}, \quad P_{12\times 12} = \begin{pmatrix} 0&1&1&1&1&1&1&1&1&1&1&1 \\ 1&1&1&0&1&1&1&0&0&0&1&0 \\ 1&1&0&1&1&1&0&0&0&1&0&1 \\ 1&0&1&1&1&0&0&0&1&0&1&1 \\ 1&1&1&1&0&0&0&1&0&1&1&0 \\ 1&1&1&0&0&0&1&0&1&1&0&1 \\ 1&1&0&0&0&1&0&1&1&0&1&1 \\ 1&0&0&0&1&0&1&1&0&1&1&1 \\ 1&0&0&1&0&1&1&0&1&1&1&0 \\ 1&0&1&0&1&1&0&1&1&1&0&0 \\ 1&1&0&1&1&0&1&1&1&0&0&0 \\ 1&0&1&1&0&1&1&1&0&0&0&1 \end{pmatrix}. \tag{13.25}$$

Note that P is symmetric. The $[23,12,7]$ binary Golay code may be obtained by *puncturing* the $[24,12,8]$ code listed above; that is, by eliminating any row of P (typically, the last).

13.10 Puncturing/extending, augmenting/expurgating, and shortening/lengthening

For a given alphabet q, there are six essential operations that may be applied to any $[n,k,d]_q$ LC, with $r = n - k$:

Puncturing:	fix k, reduce r,	reduce n	(eliminate check symbols)
Extending:	fix k, increase r,	increase n	(add check symbols)
Augmenting:	fix n, reduce r,	increase k	(eliminate check symbols and add data symbols)
Expurgating:	fix n, reduce k,	increase r	(eliminate data symbols and add check symbols)
Shortening:	fix r, reduce k,	reduce n	(eliminate data symbols)
Lengthening:	fix r, increase k,	increase n	(add data symbols)

A code obtained by eliminating a check symbol, and thus reducing both r and n by 1, is said to be a **punctured** code, and (conversely) a code obtained by adding a check symbol is said to be an **extended** (a.k.a. **expanded**) code. If starting from a perfect (or, nearly perfect) LC, puncturing it by 1 symbol will typically also reduce d by 1. For example, as seen in §13.5-13.6 and §13.9, the (perfect) binary Hamming and binary Golay codes may be extended to quasi-perfect codes by adding an overall parity bit, thereby increasing n by 1 (and, in the case of these specific codes, increasing d by 1 as well); conversely, the corresponding quasi-perfect codes may be punctured to create the corresponding perfect codes by removing the overall parity bit.

In contrast, a code obtained by simultaneously eliminating a check symbol and adding a data symbol is said to be an **augmented** code, and (conversely) a code obtained by simultaneously eliminating a data symbol and adding a check symbol is said to be an **expurgated** code. An example of this is the [7,3,4] code, which may be augmented to form the [7,4,3] code; conversely, the [7,4,3] code may be expurgated to form the [7,3,4] code.

Finally, a code obtained by eliminating a data symbol, and thus reducing both k and n by 1, is said to be a **shortened** code, and (conversely) a code obtained by adding a data symbol, and thus increasing k and n by 1, is said to be a **lengthened** code. Shortening an LC leaves both $r = n - k$ and d unchanged, but reduces the data rate k/n. Shortening is useful for developing LBCs for error-correcting memory systems, in which the data comes naturally in blocks of 8, 16, 32, 64, 128, or 256 bits (see §1.2). In particular,

- starting from $[15,11,3]$ or $[16,11,4]$, eliminating 3 data bits creates the shortened $[12,8,3]$ or $[13,8,4]$ LBCs,
- starting from $[31,26,3]$ or $[32,26,4]$, eliminating 10 data bits creates the $[21,16,3]$ or $[22,16,4]$ LBCs,
- starting from $[63,57,3]$ or $[64,57,4]$, eliminating 25 data bits creates the $[38,32,3]$ or $[39,32,4]$ LBCs,
- starting from $[127,120,3]$ or $[128,120,4]$, eliminating 56 data bits creates the $[71,64,3]$ or $[72,64,4]$ LBCs,
- starting from $[255,247,3]$ or $[256,247,4]$, eliminating 119 data bits creates the $[136,128,3]$ or $[137,128,4]$ LBCs,
- starting from $[511,502,3]$ or $[512,502,4]$, eliminating 246 data bits creates the $[265,256,3]$ or $[266,256,4]$ LBCs.

Most ECC memory and RAID 6 storage systems are based on one of these shortened LBCs[11], which are SEC (if $d = 3$) or SECDED (if $d = 4$), and are both simple and fast to use.

The operations of shortening and puncturing are perhaps the most important of the six operations discussed above; as highlighted in Figure 13.3 for $1 \le d \le 12$ and $1 \le k \le 64$, the densest LBCs available may be obtained simply by shortening and/or puncturing a few exemplary LBCs, many of which are defined in §13.3-13.9.

A shortened LC can be coded, decoded, and (if necessary) corrected using essentially the same algorithms available to code, decode, and correct the LC before it was shortened, with the unused data symbols simply set to zero (and, thus, not actually transmitted over the channel), and the corresponding symbols on the other end of the channel asserted to be zero (and, error-free).

[11]Note that the perhaps peculiar name "shortened extended binary Hamming code" means precisely a binary Hamming code that has been *extended* by adding an overall parity bit, and then *shortened* by removing some data bits.

Algorithm 13.1: Main loop of a binary CRC code implementing (13.13); full code available at RR_CRC_encode.m.

```
t=bitshift(a,r);   % initialize t corresponding to t(z) = z^r * a(z)
for  i=n:-1:r+1    % zero coefficient i in t(z) by subtracting shift of v(z)
  if bitget(t,i), t=bitxor(t,bitshift(v,i-r-1)); end
end
b=dec2bin(t,r); whos
```

13.11 Binary Cyclic Redundancy Check (CRC) codes

As laid out in §13.2, LCs in systematic cyclic form are defined by basis polynomials $v(z) = v_r z^r + \ldots + v_1 z + v_0$, and/or corresponding parity check polynomials $h(z) = h_k z^k + \ldots + h_1 z + h_0$, defined mutually such that $h(z)\, v(z) = z^n - 1 = 0$, for a given r, k, and n such that $r + k = n$.

Binary cyclic codes are also commonly implemented in a cyclic redundancy check (CRC) setting, in which the number of data bits k to be transmitted is actually *variable*. In standardized applications (USB, bluetooth, ethernet, etc), r and $v(z)$ are predefined by the particular version of the standard being used (see §3), and k is restricted to particular values within a certain range (e.g., from 1 B to 16 KiB). Binary cyclic redundancy check codes are commonly denoted CRC-rA, where $r = n - k$ is the number of parity bits used (typically, $r = 4$ to 32), and A is a set of letters and numbers used to identify that CRC code.

A CRC code is defined by its basis polynomial $v(z)$. For example, the CRC-32 code used by ethernet is

$$v(z) = z^{32} + z^{26} + z^{23} + z^{22} + z^{16} + z^{12} + z^{11} + z^{10} + z^8 + z^7 + z^5 + z^4 + z^2 + z + 1.$$

It is helpful to introduce a shorthand[12] to define the polynomial $v(z)$. Treating the coefficients of all but the constant $[z^0]$ term as a binary number, we can write these coefficients as $1000\,0010\,0110\,0000\,1000\,1110\,1101\,1011_2$ in this case, or in hexadecimal as 0x82608edb, as listed in the corresponding (802.3) entry in Table 13.5. Additional shorthand examples, and their corresponding basis polynomials, are given in the first 7 rows of Table 13.5.

The CRC approach takes a block of data $a(z)$ [with k bits], and applies a binary version of (13.13), leveraging the basis polynomial $v(z)$ implemented, to determine the $b(z)$ [with r bits] of the corresponding codeword $w(z)$ [with $n = k + r$ bits] to be transmitted [see (13.11)]. The r bits of $b(z)$ may be determined by the (inefficient, but easy-to-read) code illustrated in Algorithm 13.1; in practice, this algorithm should be implemented either in an ASIC, or using a carefully-tuned (but generally much more difficult to read) low-level (C or assembly) code. As an example, applying this approach to the data vector $1101\,1010\,0011\,1101_2 =$ 0xda3d using the CRC-5-USB code, each nontrivial step of the binary xor performed in the corresponding calculation is as follows:

```
t = 110110100011110100000
v = 100101000000000000000
t = 010011100011110100000
v = 010010100000000000000
t = 000001000011110100000
v = 000001001010000000000
t = 000000001001110100000
v = 000000001001010000000
t = 000000000000100100000
v = 000000000000100101000
t = 000000000000000001000
```

The $r = 5$ parity bits to be transmitted in this case are thus 01000.

[12]The version of the shorthand notation used in this text is referred as **implicit+1** or **reversed reciprocal** notation.

				maximum k for a given d				
r	nickname	shorthand	basis polynomial $v(z)$	$d=3$,	4,	5,	6,	7
4	HAM(15, 11, 3)	0x9	z^4+z+1	11				
5	HAM(31, 26, 3)	0x12	z^5+z^2+1	26				
	CRC-5-ITU	0x15	z^5+z^3+z+1	→	10	→	→	1
8	HAM(255, 247, 3)	0x8e	$z^8+z^4+z^3+z^2+1$	247	13	6		
	CRC-8F/3	0xe7	$z^8+z^7+z^6+z^3+z^2+z+1$	247	19			
	CRC-8P	0x83	z^8+z^2+z+1	→	119			
	CRC-8F/5	0xeb	$z^8+z^7+z^6+z^4+z^2+z+1$	→	→	9	2	1
	BCH(15, 7, 5)	0xe8	$z^8+z^7+z^6+z^4+1$	→	→	7		

			maximum k for a given Hamming distance d							
r	nickname	shorthand	$d=3$ (SEC),	4,	5 (DEC),	6,	7,	8,	9,	10
16	HAM(65535, 65519, 3)	0x8016	65519	551	40					
	CRC-16F/3	0x8d95	65519	1149	62	19	9	→	5	
	CRC-16K/5	0x9627	65519	390	119	15	11	→	4	
	CRC-16K/6	0x86f2	65519	83	53	40	→	6	5	
	CRC-16K/7.2	0xb82d	65519	221	22	→	18	→	2	
	C5	0xd175	→	32751	→	93	→	11	→	2
	C3	0xac9a	→	→	241	35	10	8	3	
	BCH(255, 239, 5)	0xb7b1	→	→	239	14	13	→	→	2
	C1	0x9eb2	→	→	→	135	→	6	→	4
	C4	0x808d	→	28642	→	99				
24	HAM($2^{24}-1, 2^{24}-25, 3$)	0x80000d	16777191	5815	509					
	CRC-24K/3.2	0x8f90e3	16777191	22868	599	47	37	33	12	10
	CRC-24K/3.3	0x93cb4f	16777191	8880	1060	52	39	19	→	9
	CRC-24K/3.4	0xa2e4ce	16777191	2562	457	248	70	30	11	9
	CRC-24K/4	0x9945b1	→	8388583	→	822	→	37	→	12
	CRC-24K/5.1	0x98ff8c	→	→	4073	228	13	→	→	9
	BCH(4095, 4071, 5)	0xa0efce	→	→	4071	121	54	→	24	9
	CRC-24K/6.2	0xbd80de	→	4074	→	2026	→	59	→	12
	BCH(255, 231, 7)	0xddd0da	→	→	→	→	231	9	8	2
	CRC-24K/6sub8	0x80009a	→	8388583	→	667				
32	HAM($2^{32}-1, 2^{32}-33, 3$)	0x8C000001	4294967263	76947	2210					
	CRC-32 (802.3)	0x82608edb	4294967263	91607	2974	268	171	91	57	34
	CRC-32K/3.1	0xad0424f3	4294967263	427053	817	522	149	63	27	19
	CRC-32K/3.2	0x9d9947fd	4294967263	146826	8162	361	145	60	56	33
	CRC-32K/3.3	0x8e2371ef	4294967263	118103	2438	1199	201	66	45	38
	CRC-32K/4.2	0xc9d204f5	→	2147483615	→	6167	→	148	→	44
	CRC-32/5.1	0xd419cc15	→	→	65505	1060	81	58	→	27
	BCH(65535, 65503, 5)	0x80af10a3	→	→	65503	501	157	62	55	26
	CRC-32K/6.4	0x9960034c	→	65506	→	32738	→	193	→	31
	CRC-32K/6sub8	0x80000072	→	1761607438	→	4113				
36	HAM($2^{36}-1, 2^{36}-37, 3$)	0x800000400	68719476699							
	BCH(262143, 262107, 5)	0x820843820	→	→	262107	?	?	?	?	?
	BCH(4095, 4059, 7)	0xa21e33520	→	→	→	→	4059	?	?	?
	BCH(511, 475, 9)	0xe615cc4d0	→	→	→	→	→	→	475	46

Table 13.5: Performance of various cyclic codes. Entries indicated with '→' represent no improvement over the entry for next larger value of d. Additional columns, for $d > 10$ and reduced k, are (to save space) not shown. Tabulated data and related software (used for calculating d for HAM and BCH codes) courtesy of Koopman.

Algorithm 13.2: Binary code for computing $h(z)$ from $v(z)$; full code available at RR_CRC_h_from_v.m.

```
t=bitshift(1,n)+0b1u64; h=0b0u64; % initialize t=100...001 corresponding to t(z)=z^n-1
for i=n+1:-1:r+1                  % calculate h=t/v on F2
  if bitget(t,i)
    h=h+bitshift(1,i-r-1); t=bitxor(t,bitshift(v,i-r-1));
  end
end
```

Table 13.5 quantifies the performance (that is, the maximum number of data bits k for a given Hamming distance d, up to[13] $d = 10$) of various cyclic codes. Maximum values of k highlighted in red in the Table are optimal amongst all CRC codes for that value of r and d; amongst all CRC codes with the indicated (in red) optimal k values for that r and d, tuned secondary k values (for different d) are highlighted in blue. Note that k values highlighted in green are optimal, for $d = 6$ and that value of r, amongst all 6sub8 CRC codes (that is, cyclic codes with a basis polynomial $v(z)$ that is zero except for its leading bit and its last 8 bits, which significantly streamlines certain numerical implementations).

Noting Table 13.5 and Koopman, for a given $r \leq 32$, the highest data-rate (largest k) CRC codes are given,

in the $d = 3$ (SEC) case for $r \geq 4$, by $[2^r - 1, 2^r - (r+1), 3]$ binary Hamming codes [see §13.5],
in the $d = 4$ (SECDED) case for $r \geq 4$, by $[2^{r-1} - 1, 2^{r-1} - (r+1), 4]$ LBCs [cf. §13.6],
in the $d = 5$ (DEC) case for even $r \geq 8$, by $[2^{r/2} + 1, 2^{r/2} - (r-1), 5]$ LBCs, and
in the $d = 6$ (DECTED) case for $r \geq 17$, by $[2^{r_2} - 2\,r_2 + r, 2^{r_2} - 2\,r_2, 6]$ LBCs where $r_2 = \lfloor (r-1)/2 \rfloor$.

Remarkably, as seen in Table 13.5, there is sufficient freedom in the ordering of a CRC LBC design in cases that are optimal in the SEC setting (as highlighted in red in the $d = 3$ column), which are equivalent to binary Hamming codes[14], to simultaneously tune its performance at some higher d (as highlighted in blue), for situations in which data packets with substantially shorter k, but using the same $v(z)$, are also to be encountered in practice.

Often, CRC codes are used for error *detection* only; the same ASIC or low-level code may be used on the receiving end of the transmission to recalculate the r parity bits from the k data bits received, and an error is flagged (and, retransmission of that particular packet requested) if the received and recalculated parity bits don't match[15]; this approach can detect up to $d-1$ random bit errors. Alternatively, error *correction* may be performed by determining the closest valid codeword to the received transmission. As discussed in §13.1, for odd d, an error correcting code can correct up to $(d-1)/2$ bit errors, and, for even d, an error correcting code can both correct up to $d/2 - 1$ bit errors and (simultaneously) detect but not correct $d/2$ bit errors.

Error correction in the CRC setting can be performed using the cyclic form of the parity check matrix H [see (13.5a)] as before, which is built on the coefficients of $h(z)$, which is determined from $v(z)$ such that $h(z)\,v(z) = z^n - 1$, as shown in Algorithm 13.2.

Bursts.

[13] Additional columns are less important when selecting a CRC scheme for practical use with $r \geq 16$, which typically use $k > r$.

[14] Note that binary Hamming codes are perfect, which implies that they really can't be tweaked very much without reducing the value of d (from $d = 3$) for a given r and a given optimal value of k. However, they can still be modified according to (13.4), which can have a significant effect on the maximum k for a given $d > 3$ when a reduced number of bits are to be transmitted.

[15] Equivalently, the (possibly corrupted) received message $\hat{w}(z)$, with the parity bits $b(z)$ (instead of r zeros) appended [see (13.11)] may be fed directly into this ASIC or low-level code, with an error being flagged if a zero is not returned.

13.12 Binary BCH Codes

The Bose, Chaudhuri, Hocquenghem (BCH) family of cyclic LCs have $n = 2^m - 1$, $r \leq mt$, and $d \geq 2t + 1$, where $m \geq 3$ and $1 \leq t < 2^{m-1}$. We focus on binary ($q = 2$) BCH codes in the discussion that follows, though the BCH family generalizes to other q. The SEC ($t = 1$, $d = 3$, $r = m$) binary BCH codes are equivalent to the cyclic $[2^r - 1, 2^r - (r + 1), 3]$ binary Hamming codes discussed previously.

Binary BCH codes generalize binary Hamming codes to higher d [see (13.27)] while maintaining a cyclic form. Their construction is presented in §13.12.1. The 31 smallest redundancy [$r \leq 36$, perhaps the "most useful"] (n, k, d) binary BCH codes with $t > 1$, which do not correspond to codes discussed previously in this chapter, are listed below in the same implicit+1 shorthand format introduced in §13.11:

- the DEC ($t = 2$) BCH codes, $(2^m - 1, 2^m - 1 - 2m, 5)$ for $m = \{4, 5, \ldots, 18\}$, $r = \{8, 10, \ldots, 36\}$:
  (15,7,5)=0xe8, (31,21,5)=0x3b4, (63,51,5)=0xa9c, (127,113,5)=0x2a3e, (255,239,5)=0xb7b1, (511,493,5)=0x24ae4, (1023,1003,5)=0x80c3b, (2047,2025,5)=0x2482d8, (4095,4071,5)=0xa0efce, (8191,8165,5)=0x26a8aa5, (16383,16355,5)=0x92dfcf5, (32767,32737,5)=0x21080632, (65535,65503,5)=0x80af10a3, (131071,131037,5)=0x20006003b, (262143,262107,5)=0x820843820;
- the TEC ($t = 3$) BCH codes for $m = \{4, 5, 6, 7, 8, 9, 10, 11, 12\}$, $r = \{10, 15, 18, 21, 24, 27, 30, 33, 36\}$:
  (15,5,7)=0x29b, (31,16,7)=0x47d7, (63,45,7)=0x3c167, (127,106,7)=0x14980d, (255,231,7)=0xddd0da, (511,484,7)=0x6b095bc, (1023,993,7)=0x28548889, (2047,2014,7)=0x137c5373e, (4095,4059,7)=0xa21e33520;
- the 4EC ($t = 4$) BCH codes for $m = \{6, 7, 8, 9\}$, $r = \{24, 28, 32, 36\}$:
  (63,39,9)=0xed93bb, (127,99,9)=0xc52bc9f, (255,223,9)=0xf72da17e, (511,475,9)=0xe615cc4d0;
- the 5EC ($t = 5$) BCH codes for $m = \{5, 6, 7\}$, $r = \{20, 27, 35\}$:
  (31,11,11)=0xb136a, (63,36,11)=0x4374089, (127,92,11)=0x708e54dab;
- the 6EC ($t = 6$) BCH code for $m = \{6\}$, $r = \{33\}$:
  (63,30,13)=0x1be6875b3 (??).

Note that the maximum data rate k/n of these codes *increases* quickly as m (and thus both r and $k = n - r$) are increased for a given t, but the maximum data rate k/n *decreases* as t (and thus the Hamming distance d) is increased for a given m. Following the constructive process in §13.12.1, efficient binary BCH codes with large n and t may be generated; for example,

- the $m = 10$ BCH codes for $t = \{3, 4, 5, 6, 12, 26, 57, 106\}$, $r = \{30, 40, 50, 60, 120, 255, 510, 765\}$:
  (1023,993,7)=0x28548889, (1023,983,9), (1023,973,11), (1023,963,13),
(1023,903,25), (1023,768,53), (1023,513,115), (1023,258,213).

Note also that the performance (that is, the maximuum k for a given Hamming distance d) for the 8 codes highlighted above in magenta is provided in full in Table 13.5. Many other BCH codes with large n and t may also be generated (see, e.g., Table 6.1 of Lin & Costello 1983).

The reason binary BCH codes are interesting is that, as generalizations of Hamming codes, fast error correction algorithms may be constructed for them, as discussed briefly in §??.

As seen in Table 13.5 and comparing with the corresponding entries highlighted in red, the performance of BCH codes are seen to be nearly optimal at the Hamming distances for which they were designed. This fact is quite remarkable, given the convenient structure of BCH codes from the perspective of error correction.

In summary, at $d = 3$, cyclic binary Hamming codes are perfect, and error correction may be being performed easily if desired (see §13.5.1). For higher d using a cyclic LBC, if performing error *detection* only, one of the optimal entries highlighted in red in Table 13.5 should be selected, whereas if performing error *correction*, a BCH code, such as one of those listed above, should generally be selected instead.

13.12.1 Construction of binary BCH codes†

As described in §13.12, a binary BCH code has $n = 2^m - 1$, $r \leq mt$, and $d \geq 2t+1$, where m and t are small positive integers. In this section, we describe the framework used to identify the binary BCH codes.

13.12.1.1 Primitive polynomials

To begin, we first define an **irreducible** polynomial as a polynomial that can not be factored as the product of other (lower-order) polynomials with coefficients of the same type (in this case, binary).

We then define a **primitive** polynomial $f(z)$ as an irreducible polynomial of degree m such that, for any root α of $f(z)$, the first $n = 2^m - 1$ powers of α (namely, $\beta_0 = \alpha^0$ to $\beta_{n-1} = \alpha^{n-1}$) generate all n of the n'th roots of unity (that is, this process generates all of the distinct β such that $\beta^n - 1 = 0$).

The test for whether or not an irreducible polynomial is primitive is perhaps best illustrated by two examples. In each example, we denote α as a zero of the polynomial considered, and simplify the expressions of the first n powers of α using the polynomial itself. For primitive polynomials like $f(z) = z^4 + z + 1$ (restricting our attention here, for simplicity, to the case with binary coefficients, with $m = 4$ and thus $n = 15$), all n'th roots of unity are reached via this process; for non-primitive polynomials like $g(z) = z^4 + z^2 + 1$, this fails to be true:

primitive example: $f(z) = z^4 + z + 1$		**non-primitive example**: $g(z) = z^4 + z^2 + 1$	
α^p mod $f(\alpha)$	binary representation	α^p mod $g(\alpha)$	binary representation
$\alpha^0 \to 1$	0001	$\alpha^0 \to 1$	0001
$\alpha^1 \to \alpha$	0010	$\alpha^1 \to \alpha$	0010
$\alpha^2 \to \alpha^2$	0100	$\alpha^2 \to \alpha^2$	0100
$\alpha^3 \to \alpha^3$	1000	$\alpha^3 \to \alpha^3$	1000
$\alpha^4 \to \alpha + 1$	0011	$\alpha^4 \to \alpha^2 + 1$	0101
$\alpha^5 \to \alpha(\alpha^4) = \alpha^2 + \alpha$	0110	$\alpha^5 \to \alpha(\alpha^4) = \alpha^3 + \alpha$	1010
$\alpha^6 \to \alpha(\alpha^5) = \alpha^3 + \alpha^2$	1100	$\alpha^6 \to \alpha^4 + \alpha^2 = 1$	0001
$\alpha^7 \to \alpha^4 + \alpha^3 = \alpha^3 + \alpha + 1$	1011	$\alpha^7 \to \alpha \cdot \alpha^6 = \alpha$	0010
$\alpha^8 \to \alpha^4 + \alpha^2 + \alpha = \alpha^2 + 1$	0101	$\alpha^8 \to \alpha^2$	0100
$\alpha^9 \to \alpha^3 + \alpha$	1010	$\alpha^9 \to \alpha^3$	1000
$\alpha^{10} \to \alpha^4 + \alpha^2 = \alpha^2 + \alpha + 1$	0111	$\alpha^{10} \to \alpha^4 = \alpha^2 + 1$	0101
$\alpha^{11} \to \alpha^3 + \alpha^2 + \alpha$	1110	$\alpha^{11} \to \alpha^5 = \alpha^3 + \alpha$	1010
$\alpha^{12} \to \alpha^4 + \alpha^3 + \alpha^2 = \alpha^3 + \alpha^2 + \alpha + 1$	1111	$\alpha^{12} \to \alpha^6 = 1$	0001
$\alpha^{13} \to \alpha^4 + \alpha^3 + \alpha^2 + \alpha = \alpha^3 + \alpha^2 + 1$	1101	$\alpha^{13} \to \alpha \cdot \alpha^{12} = \alpha$	0010
$\alpha^{14} \to \alpha^4 + \alpha^3 + \alpha = \alpha^3 + 1$	1001	$\alpha^{14} \to \alpha^2$	0100
$\alpha^{15} \to \alpha^4 + \alpha = 1$	0001	$\alpha^{15} \to \alpha^3$	...

(13.26)

The process of computing α^p mod $f(\alpha)$ is automated in RR_Binary_Field_Mod.m.

Note next that, for example, the polynomial $z^n + 1$ (recall that "-1" is 1 in binary) for $m = 6$ and thus $n = 63$ may be expressed as the product [on GF(2^m)] of the following 13 irreducible polynomial factors (calculated using the Cantor-Zassenhaus algorithm):

$$z^{63} + 1 = (z+1)(z^2+z+1)(z^3+z+1)(z^3+z^2+1)(z^6+z^3+1)(z^6+z^4+z^2+z+1)(z^6+z^5+z^4+z^2+1)\cdot$$
$$(z^6+z+1)(z^6+z^4+z^3+z+1)(z^6+z^5+1)(z^6+z^5+z^2+z+1)(z^6+z^5+z^3+z^2+1)(z^6+z^5+z^4+z+1).$$

The last 6 terms in this factorization, highlighted in red and blue, are the $m = 6$ primitive polynomials. The other 3 polynomials of order 6 in this factorization are *not* primitive, as they fail the test illustrated in (13.26). For $m = \{1, 2, 3, 4, 5, 6, 7, \ldots\}$, the number of primitive polynomials follows the OEIS A011260 sequence: $\{1, 1, 2, 2, 6, 6, 18, 16, 48, 60, 176, 144, 630, 756, 1800, 2048, \ldots\}$. For BCH code generation, at any m, it is customary to use a primitive polynomial with the smallest number of terms and, amongst these, that polynomial with the smallest powers in the terms after the first, as highlighted in red above. Such a primitive polynomial is said to be **minimal**, and may be found by exhaustive search. The first 50 minimal primitive polynomials for binary fields GF(2^m) are listed in Table 13.6; for practical BCH code development, these should be all you need.

$z+1$	z^2+z+1	z^3+z+1	z^4+z+1	z^5+z^2+1
z^6+z+1	z^7+z+1	$z^8+z^4+z^3+z^2+1$	z^9+z^4+1	$z^{10}+z^3+1$
$z^{11}+z^2+1$	$z^{12}+z^6+z^4+z+1$	$z^{13}+z^4+z^3+z+1$	$z^{14}+z^5+z^3+z+1$	$z^{15}+z+1$
$z^{16}+z^5+z^3+z^2+1$	$z^{17}+z^3+1$	$z^{18}+z^7+1$	$z^{19}+z^6+z^5+z+1$	$z^{20}+z^3+1$
$z^{21}+z^2+1$	$z^{22}+z+1$	$z^{23}+z^5+1$	$z^{24}+z^4+z^3+z+1$	$z^{25}+z^3+1$
$z^{26}+z^8+z^7+z+1$	$z^{27}+z^8+z^7+z+1$	$z^{28}+z^3+1$	$z^{29}+z^2+1$	$z^{30}+z^{16}+z^{15}+z+1$
$z^{31}+z^3+1$	$z^{32}+z^{28}+z^{27}+z+1$	$z^{33}+z^{13}+1$	$z^{34}+z^{15}+z^{14}+z+1$	$z^{35}+z^2+1$
$z^{36}+z^{11}+1$	$z^{37}+z^{12}+z^{10}+z^2+1$	$z^{38}+z^6+z^5+z+1$	$z^{39}+z^4+1$	$z^{40}+z^{21}+z^{19}+z^2+1$
$z^{41}+z^3+1$	$z^{42}+z^{23}+z^{22}+z+1$	$z^{43}+z^6+z^5+z+1$	$z^{44}+z^{27}+z^{26}+z+1$	$z^{45}+z^4+z^3+z+1$
$z^{46}+z^{21}+z^{20}+z+1$	$z^{47}+z^5+1$	$z^{48}+z^{28}+z^{27}+z+1$	$z^{49}+z^9+1$	$z^{50}+z^{27}+z^{26}+z+1$

Table 13.6: Minimal primitive polynomials for binary fields GF(2^m) for orders $m = 1$ through 50.

13.12.1.2 Minimal polynomials with $\beta_p = \alpha^p$ as a root

Given a primitive polynomial $f(z)$ of order m, and denoting α as a root of this polynomial, we next seek to find the polynomial of lowest order with $\beta_p = \alpha^p$ as a root, dubbed the **minimal polynomial** $\phi_p(z)$, for various powers p between 0 and $n-1$. Of course, since $f(z)$ is irreducible, the minimal polynomial for $p = 1$ is just the primitive polynomial itself, $\phi_1(z) = f(z)$. For larger p, the corresponding minimal polynomial may be found by expressing β_p^q, for $q = 1$ to m, in terms of the α^i for $i = 0$ to $m-1$, following the general approach demonstrated in (13.26) [using, e.g., RR_Binary_Field_Mod.m], then using the results to rewrite the equation

$$c_0 + c_1\beta_p + c_2\beta_p^2 + \ldots + c_{m-1}\beta_p^{m-1} + c_m\beta_p^m = 0$$

in terms of the α_i (for $i = 0$ to $m-1$). Setting the coefficients of like powers of α equal to zero in the resulting equation leads to a homogeneous set of m linear equations in the $m+1$ unknowns $\mathbf{x} = \{c_0, c_1, \ldots, c_m\}$, which may be written in the form $A\mathbf{x} = 0$. This underdetermined binary system has one or more nontrivial solutions, the smallest of which may be found by transforming A to reduced echelon form R [see *NR* §2.6] using binary arithmetic, then setting the coefficient c_i corresponding to the first non-pivot column of equal to 1, and the coefficients corresponding to all other non-pivot columns (if any) equal to 0, and solving the remaining nonsingular system for the remaining coefficients c_i. The resulting minimal polynomial may then be written as

$$\phi_p(z) = c_0 + c_1 z + c_2 z^2 + \ldots + c_{m-1}z^{m-1} + c_m z^m,$$

and has β_p as a root by construction. This process is automated in RR_Binary_Field_Mod.m.

13.12.1.3 Putting it all together

Given a primitive polynomial $f(z)$ of order m [see §13.12.1.1; often, the minimal primitive polynomial from Table 13.6 is used], denoting α as a root of $f(z)$, and given the minimal polynomials $\phi_p(z)$ with $\beta_p = \alpha^p$ as roots [see §13.12.1.2] for $p = 3, 5, 7, \ldots$, denoting $LCM[\cdot, \cdot, \ldots]$ as the least common multiple of the polynomials indicated, a binary BCH code with $n = 2^m - 1$ and $r = n - k \le mt$ may then be constructed as follows:

$$\begin{aligned}
&\text{for } t = 1,\ d \ge 3 \text{ (i.e., binary Hamming)}, && \text{take } v(z) = f(z),\\
&\text{for } t = 2,\ d \ge 5 \text{ (for } m \ge 4), && \text{take } v(z) = LCM[f(z), \phi_3(z)],\\
&\text{for } t = 3,\ d \ge 7 \text{ (for } m \ge 4), && \text{take } v(z) = LCM[f(z), \phi_3(z), \phi_5(z)],\\
&\text{for } t = 4,\ d \ge 9 \text{ (for } m \ge 6), && \text{take } v(z) = LCM[f(z), \phi_3(z), \phi_5(z), \phi_7(z)],\\
&\text{for } t = 5,\ d \ge 11 \text{ (for } m \ge 5), && \text{take } v(z) = LCM[f(z), \phi_3(z), \phi_5(z), \phi_7(z), \phi_9(z)],\\
&\text{for } t = 6,\ d \ge 13 \text{ (for } m \ge 6), && \text{take } v(z) = LCM[f(z), \phi_3(z), \phi_5(z), \phi_7(z), \phi_9(z), \phi_{11}(z)],
\end{aligned} \tag{13.27}$$

etc. All of the BCH codes summarized in the introduction to §13.12 were generated in this manner; the process is automated in RR_BCH_Constructor.m.

13.13 Soft decision decoding

The type of decoding discussed thus far, in which the received vector $\hat{\mathbf{w}}$ is assumed to be in $\mathbf{F}_q^n$ [see (13.1)], is known as **hard decision decoding**.

Another formulation of the decoding problem, which we describe here in the binary case $q = 2$, assumes again that $\mathbf{w} \in \mathbf{F}_2^n$ (that is, that the symbols being transmitted are binary, and in this formulation are usually rescaled to be ± 1), but that $\hat{\mathbf{w}} \in \Re^n$ (that is, that the received data is real). In this setting, missing bits in known positions (aka erasures) are assigned a value of zero, and bits received only "weakly" are assigned numeric values closer to zero than to ± 1. The decoding problem in this case, called **soft decision decoding**, is similar to that considered before (again, to find the most likely valid codeword $\mathbf{w}$ corresponding to the received vector $\hat{\mathbf{w}}$, and the original data vector $\mathbf{a}$ that generated it), but is now based on finding the valid codeword $\mathbf{w}$ that minimizes the Euclidian distance to $\hat{\mathbf{w}}$, rather than that which minimizes the Hamming distance.

Implementation of soft decision decoding is somewhat difficult in many applications, as it requires some sort of ADC (analog-to-digital converter) in order to quantify the (real) value associated with the "certainty" of each received bit. This approach is thus not often used in embedded computing today. However, when the data is difficult to obtain but easy to process (e.g., signals received from an interplanetary space probe by NASA's Deep Space Network, or signals received by a wall-powered server from an ultra-low-power embedded IoT device), a soft decision decoding approach may indeed be warranted.

References

Lin, S, Costello, D.J. (1983) *Error Control Coding: Fundamentals and Applications*. Prentice-Hall.

Part III

System Design, Development, and Integration