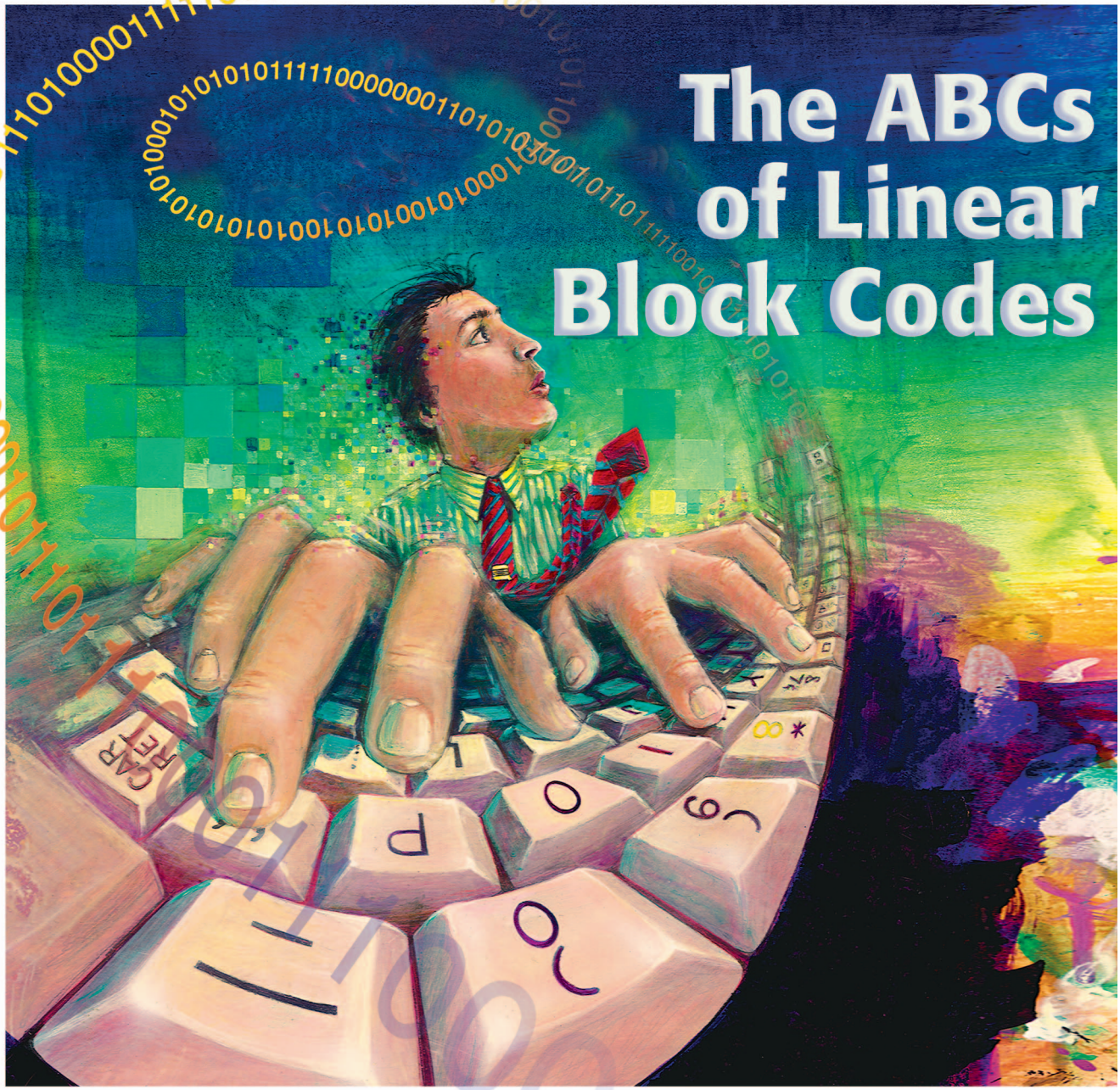


An intuitive treatment of error detection and correction

Bernard Sklar and Fredric J. Harris

Channel coding is an error-control technique used for providing robust data transmission through imperfect channels by adding redundancy to the data. There are two important classes of such coding methods: block and convolutional. For this tutorial, we focus on linear block codes because they provide much insight and allow for a simple visualization of the error detection/correction process. Forward error correction (FEC) is the name used when the receiving equipment does most of the work. In the case of block codes, the decoder looks for errors and, once detected, corrects them (according to the capability of the code). The technique has become an important signal-processing tool used in modern communication systems and in a wide variety of other digital applications such as high-density memory and recording media. Such coding provides system performance improvements at significantly lower cost than through the use of other methods that increase signal-to-noise ratio (SNR) such as increased power or antenna gain.

In this article we first develop the ideas behind simple binary codes. We then treat cyclic and nonbinary

1053-5888/04/$20.00©2004IEEE

codes. Finally, we review the astounding strides that have been made in this decade toward approaching the theoretical limitations of what is possible. This article can serve as a precursor for serious students who intend to further pursue the subject and a quick overview for experienced scientists or mathematicians who have not yet investigated this area.

Channel Coding

Channel coding involves data transformations that are used for improving a system's error performance by enabling a transmitted message to better withstand the effects of channel impairments such as noise, interference, and fading. For applications that use simplex channels (one-way channels such as compact disk recordings), the coding techniques must support FEC since the receiver must detect and correct errors without the use of a reverse channel (for retransmission requests). Such FEC techniques can be thought of as vehicles for accomplishing desirable tradeoffs that can reduce bit error rate (BER) at a fixed power level or allow a specified error rate at a reduced power level at the cost of increased bandwidth (or transmission delay) and a processing burden. Figure 1 illustrates such coding applied to a typical digital radio. A data or message vector $\mathbf{m} = m_1, m_2, \ldots, m_k$ containing k message elements from an alphabet is transformed by the block code into a longer code vector or code word $\mathbf{U} = u_1, u_2, \ldots, u_n$ containing n code elements constructed from the same alphabet. The elements in the alphabet have a one to one correspondence with elements drawn from a finite field. Finite fields are referred to as Galois fields, after the French mathematician Evariste Galois (1811–1832). A Galois field containing q elements is denoted GF(q), with the simplest such finite field being GF(2), the binary field with elements (1, 0), which have the obvious connection to the logical symbols (1, 0) called bits. When we deal with fields that contain more than two elements, these nonbinary elements are encoded as binary m-tuples (m-bit sequences). Then the elements are processed as binary words according to the rules of the field in much the same way that decimal integers were encoded as binary-coded decimal (BCD) symbols in early computers (such as ENIAC or the IBM 650) and in contemporary calculators. The number of output elements n (code bits) and input elements k (data bits) characterizing a block code are denoted by the ordered pair (n, k). Often, the designation (n, k, t) is used to indicate that the code is capable of correcting t-errors in the n-element code word.

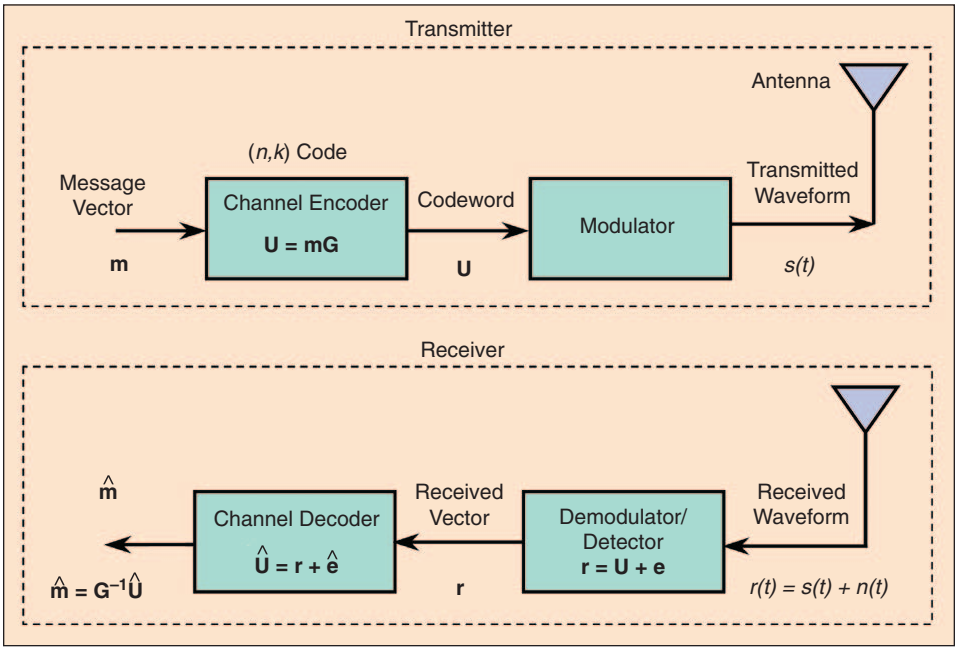
For transmitting the code bits (comprising $\mathbf{U}$) with waveforms, a common practice is to use bipolar pulses with values $(+1, -1)$ to represent the binary logic levels (1, 0), respectively. For a radio system, such pulses are modulated on to a carrier wave, typically denoted $s(t)$, as shown in Figure 1. Channel impairments are responsible for transforming a transmitted waveform $s(t)$ into a corrupted waveform $r(t) = s(t) + n(t)$, which is received and processed by a demodulator/detector. The demodulator recovers samples of the corrupted waveform, and the detector interprets the digital meaning of that waveform. A commonly used model for $n(t)$ is that of an additive white Gaussian noise (AWGN) process [1]. Noise, interference, and channel distortion mechanisms account for the detector making errors. Consequently, instead of accurately reproducing the bipolar pulse values or logic levels (representing $\mathbf{U}$), the detector might instead output a corrupted version $\mathbf{r}$, written as

$$\mathbf{r} = \mathbf{U} + \mathbf{e} \tag{1}$$

where $\mathbf{r} = r_1, r_2, \ldots, r_n$ represents a received block of n elements, and $\mathbf{e} = e_1, e_2, \ldots, e_n$ represents the corruption, referred to as the *error sequence* or *error pattern*.

Hard Decisions and Soft Decisions

In Figure 1, each detected element r_i of the received vector $\mathbf{r}$ can be described as a quantized-amplitude



▲ *1. Simplified diagram of channel coding applied to a typical digital radio.*

Table 1. Row and column parity for a two-dimensional data set.				
m_1	m_2	m_3	m_4	p_1
m_5	m_6	m_7	m_8	p_2
p_3	p_4	p_5	p_6	

decision. The decision may simply answer the question "Is the amplitude greater or less than zero?" yielding a binary decision of 1 or 0. Such a decision is called a hard decision because the detector firmly selects one of two levels. Sometimes the detector's decision may answer multiple questions such as "Is the amplitude greater or less than zero, *and* is it greater or less than some reference level?" For binary signaling, such multipart decisions are called *soft decisions*; they offer the decoder side information about the SNR of the corrupted analog waveform. A soft decision might tell the decoder, "this signal has a positive amplitude, but it is not very far from the zero amplitude" or "this signal has a positive amplitude, and it is quite far from the zero amplitude." The most popular soft-decision format entails eight-level signal quantization, which can be interpreted as a hard decision plus a measure of confidence. The figure-of-merit for the error performance of a digital communication system is usually expressed as a normalized SNR, known as the ratio of bit energy to noise power spectral density, E_b/N_0. The coding gain or benefit provided by an error-correcting code to a system can be defined as the "relief" or reduction in required E_b/N_0 that can be realized due to the code. For an AWGN channel, when the detector presents the decoder with such soft decisions, the system can typically manifest an improvement in coding gain of about 2 dB compared to hard-decision processing [1]. For the majority of block-code applications, hard-decision decoding is used. For most of this tutorial, a received vector $\mathbf{r}$ out of the detector is made up of hard-decision components, designated by pulses $(+1, -1)$ or by logic levels $(1, 0)$. However, later we show that soft decisions are of great value for systems using iterative decoding techniques that operate close to theoretical limitations. Examples of such techniques are turbo codes and low-density parity check (LDPC) codes.

Simple Parity Codes

At the transmitter, the encoder adds redundancy with a set of constraints that must be satisfied by the set of all code words. Error detection occurs when a received vector does not satisfy the constraints. The simplest approach to error detection modifies a binary data sequence into a code word by appending an extra bit called a *parity bit*. When using the constraint that a code word must contain an even number of ones, the scheme is referred to as *even parity* (the constraint of an odd number of ones is called *odd parity*). To establish the even-parity condition, the parity bit p is formed, as the modulo-2 sum of the message bits, as

$$p = m_1 \oplus m_2 \oplus m_3 \oplus \cdots \oplus m_k \tag{2}$$

where the symbol $\oplus$ indicates modulo-2 addition. The test (even-parity check) conducted by the receiver verifies that the modulo-2 sum of the parity plus message bits in the received sequence $\mathbf{r}$ is zero. If the sequence fails the test, an error has been detected. We refer to the test result as the syndrome S, written as

$$S = r_1 \oplus r_2 \oplus \cdots \oplus r_k \oplus r_{k+1}. \tag{3}$$

The syndrome in (3) can be modeled as the modulo-2 sum of the transmitted sequence and the error sequence, as

$$\begin{aligned} S &= (m_1 \oplus e_1) \oplus (m_2 \oplus e_2) \\ &\quad \oplus \cdots \oplus (m_k \oplus e_k) \oplus (p \oplus e_{k+1}) \qquad (4) \\ S &= (m_1 \oplus m_2 \oplus \cdots \oplus m_k \oplus p) \\ &\quad \oplus (e_1 \oplus e_2 \oplus \cdots \oplus e_k \oplus e_{k+1}) \\ &= 0 \oplus (e_1 \oplus e_2 \oplus \cdots \oplus e_k \oplus e_{k+1}). \qquad (5) \end{aligned}$$

When factored into separate message and error sequences as seen in (5), we recognize that the syndrome tests both the transmitted sequence and the error sequence, but since the syndrome of the transmitted sequence is zero, the syndrome is only responding to the error sequence. For the case of a single parity bit, as in (5), only an odd number of errors can be detected, since an even number of errors will yield the syndrome $S = 0$.

A single parity bit can only be used for error detection. To perform error correction, we require additional information to locate the error positions; the code word needs to be embedded with more than a single parity bit. A simple example of a code that appends additional parity bits to the message sequence is shown in Table 1. Here, a set of eight message elements is packed into a two-dimensional array from which we form parity for each row and parity for each column.

The appended array can be rearranged into a code word sequence $\mathbf{U}$ as

$$\mathbf{U} = m_1\ m_2\ m_3\ m_4\ m_5\ m_6\ m_7\ m_8\ p_1\ p_2\ p_3\ p_4\ p_5\ p_6. \tag{6}$$

When $\mathbf{U}$ is received, it can be mapped back to the same two-dimensional array, and a set of syndromes can be calculated corresponding to each row and each column. A single error located anywhere in the message positions will cause a nonzero syndrome in a row and in a column, and thus the intersection of the row and column corresponding to the parity failure contains the single error. One should conclude that a block code capable of detecting and correcting error sequences

needs to have multiple parity symbols appended to the data message and multiple syndromes generated during the parity checks at the receiver.

The Generator Matrix and Systematic Codes

The most general form of the parity generation process, in which each code element u_i of the code word $\mathbf{U}$ is a weighted sum of message elements, can be written in the form of a vector matrix equation as

$$\mathbf{U} = \mathbf{mG} \tag{7a}$$

$$[u_1\ u_2\ u_3\ \cdots\ u_n] = [m_1\ m_2\ m_3 \cdots m_k] \times \begin{bmatrix} \mathcal{G}_{1,1} & \mathcal{G}_{1,2} & \mathcal{G}_{1,3} & \cdots & \mathcal{G}_{1,n} \\ \mathcal{G}_{2,1} & \mathcal{G}_{2,2} & \mathcal{G}_{2,3} & \cdots & \mathcal{G}_{2,n} \\ \mathcal{G}_{3,1} & \mathcal{G}_{3,2} & \mathcal{G}_{3,3} & \cdots & \mathcal{G}_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathcal{G}_{k,1} & \mathcal{G}_{k,2} & \mathcal{G}_{k,3} & \cdots & \mathcal{G}_{k,n} \end{bmatrix} \tag{7b}$$

where the entries of the matrix $\mathbf{G}$, called the *generator matrix*, represent weights (field-element coefficients), and the multiplication operation follows the usual rules of matrix multiplication. The product of a message row-vector $\mathbf{m}$ with the ith column-vector of $\mathbf{G}$ forms u_i a weighted sum of message elements representing the ith element of the code word row-vector $\mathbf{U}$. For a binary code, the data elements as well as the matrix weights are 1s and 0s, but for a nonbinary code the data and weights are general field elements (of the nonbinary field) with arithmetic performed in accordance with the field structure [1], [2].

A useful variant of the code word $\mathbf{U}$ is one in which the vector of message elements is embedded, without change, in the code word along with an appended vector of parity elements. When the code word is constrained in this manner, the code is called a *systematic code*. To form a systematic code the generator matrix $\mathbf{G}$ can be modified in terms of submatrices $\mathbf{P}$ and $\mathbf{I}_k$ as follows:

$$\mathbf{U} = \mathbf{mG} = \mathbf{m}[\mathbf{P}|\mathbf{I}_k] \tag{8a}$$

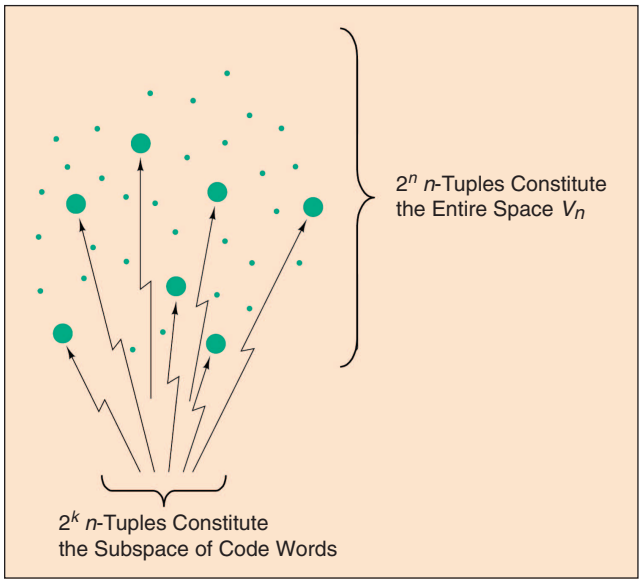
$$\mathbf{U} = [u_1\ u_2\ u_3\ \cdots\ u_n] = [m_1\ m_2\ m_3\ \cdots\ m_k] \times \begin{bmatrix} \underbrace{\begin{matrix} \mathcal{G}_{1,k+1} & \cdots & \mathcal{G}_{1,n} \\ \mathcal{G}_{2,k+1} & \cdots & \mathcal{G}_{2,n} \\ \mathcal{G}_{3,k+1} & \cdots & \mathcal{G}_{3,n} \\ \vdots & \ddots & \vdots \\ \mathcal{G}_{k,k+1} & \cdots & \mathcal{G}_{k,n} \end{matrix}}_{\mathbf{P}} & \underbrace{\begin{matrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{matrix}}_{\mathbf{I}_k} \end{bmatrix} \tag{8b}$$

where $\mathbf{P}$ is the parity portion of $\mathbf{G}$, and $\mathbf{I}_k$ is a k-by-k identity submatrix (ones on the main diagonal, and zeros elsewhere).

Vector Spaces and Subspaces

For now, we restrict our discussion to binary codes and n-bit sequences called n-tuples. The space of all binary n-tuples is called a vector space V_n over the GF(2). This field is characterized by two operations, modulo-2 addition and scalar multiplication [2], [3]. Within the space V_n, one can form as many as 2^n distinct sequences or n-tuples. A block of k message bits, referred to as a k-tuple, can form any one of 2^k possible message sequences. The encoding process involves a one-to-one assignment, whereby the 2^k message k-tuples are uniquely mapped into a new set of 2^k code words chosen from the set of 2^n n-tuples. When the code is in systematic form, one can view the procedure as the addition of $n-k$ parity bits (or redundant bits, since they carry no new information) to each k-bit message sequence. Even though such a code word is comprised of k data bits and $n-k$ parity bits, we typically refer to each element in a binary code word as a *code bit*. For linear codes, the mapping transformation is, of course, linear. Within a block, the ratio of data bits to total code bits, denoted k/n, is called the *code rate*; it represents the portion of a code bit that constitutes information. For example, in a rate $1/2$ code, each code bit carries $1/2$ b of information. Note that when the coding tradeoff involves expanded bandwidth to achieve improved performance, then use of a rate $1/2$ code will require twice as much transmission bandwidth as that of an uncoded system. Similarly, the use of a rate $3/4$ code will require a bandwidth expansion by the factor $4/3$.

To construct a valid linear block code, the 2^k n-tuples, making up the (n, k) code, must be confined to a subspace of the V_n space; all the code words of the system must satisfy this constraint. A subset of n-tuple vectors can be viewed as a subspace of V_n if the following two conditions are met: 1) the all-zeros vector is one of the vectors in the subset, and 2) the sum of any two vectors in the subset is also a member of the subset (known as the closure property). Note that condition 2 requires condition 1, because any vector added to itself modulo-2 is equal to an all-zeros vector. Figure 2 illustrates the framework of a linear block code using a simple pictorial. The entire vector space V_n comprises 2^n n-tuples (shown as points). Within this space there exists a subset of 2^k n-tuples (shown as darker points) comprising a subspace or code system. This subset of 2^k points, "sprinkled" among the more numerous 2^n points, represent the code words $\{\mathbf{U}\}$ assigned to messages. At the receiver, a corrupted version $\mathbf{r}$ of a transmitted $\mathbf{U}$ may be detected. If the corrupted vector $\mathbf{r}$ is not too unlike (not too distant from) the transmitted $\mathbf{U}$, the decoder can decode the message correctly. A correctable error is one that takes the transmitted code word outside the subspace, but doesn't take it "too far away." A noncorrectable error is one that takes the transmitted code word to the vicinity of another code



▲ *2. Linear block code framework.*

word. A nondetectable error is one that takes the transmitted code word to exactly another code word. To have a sense of what is "too far away" we next examine the concept of distance between vectors.

Weight and Distance Properties

The *Hamming weight* $w(\mathbf{U})$ of a code word $\mathbf{U}$ is defined as the number of nonzero elements in $\mathbf{U}$. For a binary vector (or a nonbinary vector with field elements represented in binary form), this is equivalent to the number of ones in the vector. For example, if $\mathbf{U} = 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1$, then $w(\mathbf{U}) = 5$. The *Hamming distance* $d(\mathbf{U}, \mathbf{V})$ between two binary code words $\mathbf{U}$ and $\mathbf{V}$ is defined as the number of bit positions in which they differ. For example

$$\begin{aligned} \text{if} \quad & \mathbf{U} = 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1 \\ \text{and} \quad & \mathbf{V} = 0\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0 \\ \text{then } & d(\mathbf{U}, \mathbf{V}) = 6. \end{aligned} \tag{9}$$

By the properties of subspaces, we note that the sum of two code words $\mathbf{U} + \mathbf{V}$ is another code word $\mathbf{W}$ in the subspace, with binary ones in those positions in which the code words $\mathbf{U}$ and $\mathbf{V}$ differ. For example

$$\mathbf{W} = \mathbf{U} + \mathbf{V} = 1\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1. \tag{10}$$

Thus, we observe that the Hamming distance between two code words is equal to the Hamming weight of the summed vectors: that is, $d(\mathbf{U}, \mathbf{V}) = w(\mathbf{U} + \mathbf{V})$. Also, note that the Hamming weight of a code word is equal to its Hamming distance from the all-zeros vector.

Decoding Task

The decoding task can be stated as follows: Having received the vector $\mathbf{r}$, find the best estimate of the particular code word $\mathbf{U}_i$ that was transmitted. The optimal decoder strategy is to minimize the decoder error probability, which is the same as maximizing the probability $P(\hat{\mathbf{U}} = \mathbf{U}_i | r)$. If all code words are equally likely and the channel is memoryless, this is equivalent to maximizing $P(\mathbf{r}|\mathbf{U}_i)$, the conditional probability density function (pdf) of $\mathbf{r}$, expressed as

$$\begin{aligned} p(\mathbf{r}|\mathbf{U}_i) &= \max p(\mathbf{r}|\mathbf{U}_j) \\ &\quad \text{over all } \mathbf{U}_j \end{aligned} \tag{11}$$

where the pdf, conditioned on having sent $\mathbf{U}_i$, is called the likelihood of $\mathbf{U}_i$. Equation (11), known as the maximum likelihood (ML) criterion [1], can be used for finding the "most likely" $\mathbf{U}_i$ that was transmitted. For algorithms using Hamming distances, the likelihood of $\mathbf{U}_i$ with respect to $\mathbf{r}$ is inversely proportional to the distance between $\mathbf{r}$ and $\mathbf{U}_i$, denoted $d(\mathbf{r}, \mathbf{U}_i)$. Therefore, we can express the decoder decision rule as: Decide in favor of $\mathbf{U}_i$ if

$$\begin{aligned} d(\mathbf{r},\ \mathbf{U}_i) &= \min\ d(\mathbf{r},\ \mathbf{U}_j) \\ &\quad \text{over all } \mathbf{U}_j. \end{aligned} \tag{12}$$

In the context of Figure 2, (12) means that for any n-tuple $\mathbf{r}$, received in the V_n space of 2^n possible values, the decoder shall choose the closest code word (heavy black point) as the best estimate $\hat{\mathbf{U}}$ of the transmitted code word.

Error-Detecting and Error-Correcting Capability

Consider the set of distances between all pairs of code words in the space V_n. The smallest member of the set is called the minimum distance of the code and is denoted $d_{\min}$. To find $d_{\min}$, we need not search the set of code words in a pairwise fashion. Because of the closure property, we need only find the nonzero code word having the minimum weight. The minimum distance, like the weakest link in a chain, gives us a measure of the code's capability (indicates the smallest number of channel errors that can lead to decoding errors). Figure 3 illustrates the distance between two code words $\mathbf{U}$ and $\mathbf{V}$ using a number line calibrated in Hamming distance, where each black dot represents a corrupted code word. In this example, let the distance $d(\mathbf{U}, \mathbf{V})$ be the minimum distance $d_{\min} = 5$. Figure 3(a) illustrates the reception of a vector $\mathbf{r}_1$, which is distance 1 from $\mathbf{U}$ and distance 4 from $\mathbf{V}$. An error-correcting decoder, following the ML strategy, will select $\mathbf{U}$ upon receiving $\mathbf{r}_1$. If $\mathbf{r}_1$ had been the result of a 1-b corruption to the transmitted code word $\mathbf{U}$, the decoder has successfully corrected the error. But if $\mathbf{r}_1$ had been the result of a 4-b corruption to the transmitted code word $\mathbf{V}$, the result is a decoding error. Similarly a double error in transmission of $\mathbf{U}$ might result in the received vector $\mathbf{r}_2$, which is distance 2 from $\mathbf{U}$ and distance 3

from **V**, as shown in Figure 3(b). Here too, the decoder will select **U** upon receiving $\mathbf{r}_2$. A triple error in transmission of **U** might result in a received vector $\mathbf{r}_3$ that is distance 3 from **U** and distance 2 from **V**, as shown in Figure 3(c). Here the decoder will select **V** upon receiving $\mathbf{r}_3$ and, given that **U** was transmitted, will have made a decoding error. From Figure 3, one can see that if the task is error detection (and not correction), then as many as 4-b errors can be detected. But, if the task is error-correction, the decision to choose **U** if **r** falls in region 1, and **V** if **r** falls in region 2, illustrates that this code (with $d_{\min} = 5$) can correct as many as 2-b errors. We can generalize a linear block code's error-detection capability ε and error-correction capability t as [1]

$$\varepsilon = d_{\min} - 1 \tag{13a}$$

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor \tag{13b}$$

where the notation $\lfloor x \rfloor$, called the floor of x, means the largest integer not to exceed x (in other words, round down if not an integer).

Table 2. Assignment of messages to code words for the (6, 3) code.

Message vector	Code Word
000	000000
100	110100
010	011010
110	101110
001	101001
101	011101
011	110011
111	000111

A (6, 3) Linear Block Code Example

Table 2 describes a code word-to-message assignment for a (6, 3) code, where the rightmost bit represents the earliest (and most-significant) bit. For each code word, the rightmost $k = 3$ bits represent the message (hence, the code is in systematic form). Since $k = 3$, there are $2^k = 2^3 = 8$ message vectors, and therefore there are eight code words. Since $n = 6$, then within the vector space $V_n = V_6$ there are a total of $2^n = 2^6 = 64$ 6-tuples.

It is easy to verify that the eight code words shown in Table 2 form a subspace of V_6 (the all-zeros vector is one of the code words, and the sum of any two code words is also a code word). Note that for a particular (n, k) code, a unique assignment does not exist; however, neither is there complete freedom of choice.

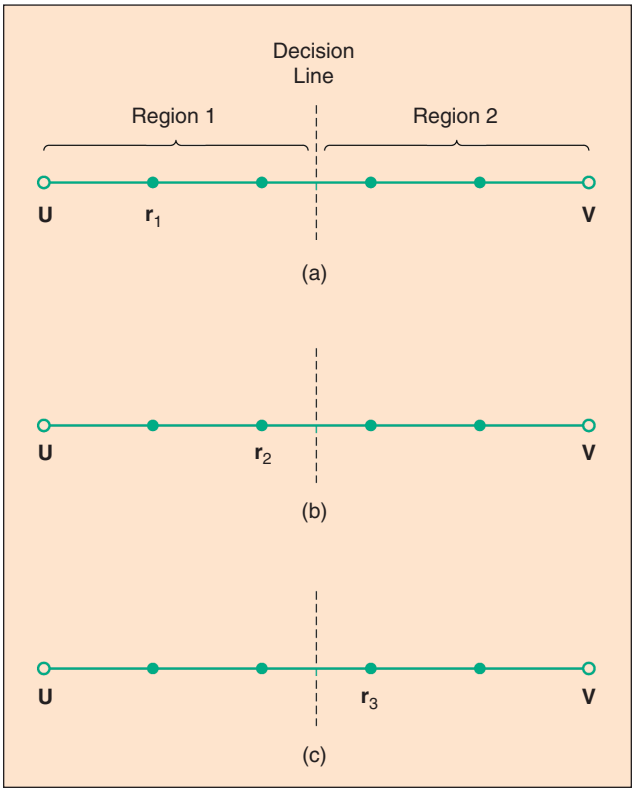
A Generator Matrix for the (6, 3) Code

For short codes, the message-to-code-word mapping in Table 2 can be accomplished via a lookup table, but if k is large, such an implementation would require a prohibitive amount of memory. Fortunately, by using a generator matrix **G**, described in (7) and (8), it is possible to reduce complexity by generating the required code words as needed instead of storing them. Since the set of code words is a k-dimensional subspace of the n-dimensional vector space, it is always possible to find a set of n-tuples (row-vectors of the matrix **G**), fewer than 2^k that can generate all the 2^k code words of the subspace. The generating set of vectors is said to span the subspace. The smallest linearly independent set that spans the subspace is called a basis of the subspace, and the number of vectors in this basis set is the dimension of the subspace. Any basis set of k linearly independent n-tuples $\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_k$ (that spans the subspace) can be used to form a generator matrix **G**. This matrix can then be used to generate the required code words, since each code word is a linear combination of $\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_k$. That is, each code word **U** within the set of 2^k code words can be described by

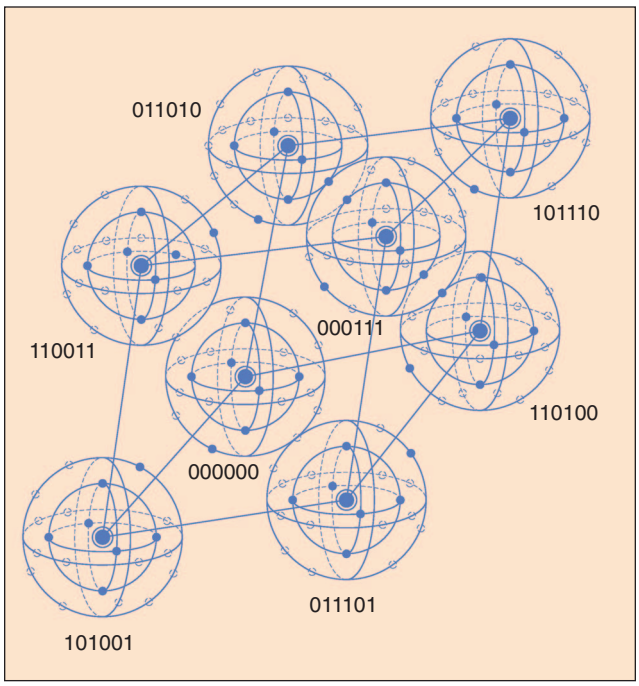
$$\mathbf{U} = m_1\mathbf{V}_1 + m_2\mathbf{V}_2 + \cdots + m_k\mathbf{V}_k \tag{14}$$

where each $m_i = (1 \text{ or } 0)$ is a message bit and the index $i = 1, \ldots, k$ represents its position. In general, we describe this code generation in terms of multiplying a message vector **m** by a generator matrix **G**. For the (6, 3) code introduced earlier, we can fashion a generator matrix **G** in systematic form, as

$$\mathbf{G} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \\ \mathbf{V}_3 \end{bmatrix} = \begin{bmatrix} \underbrace{\begin{matrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{matrix}}_{\mathbf{P}} & \underbrace{\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}}_{\mathbf{I}_k} \end{bmatrix} \tag{15}$$



▲ *3. Error detection and correction capability.*



▲ *4. Visualization of eight code words in a 6-tuple space.*

where $\mathbf{P}$ and $\mathbf{I}_k$ represent the parity and identity submatrices, respectively, and $\mathbf{V}_1, \mathbf{V}_2$, and $\mathbf{V}_3$ are three linearly independent vectors (a subset of the eight code vectors) that can generate all the code words, made up of the weights $\{g_{i,j}\}$ as described in (8). Note also that the sum of any two generating vectors does not yield any of the other generating vectors since linear independence is, in effect, the opposite of closure. The generator matrix $\mathbf{G}$ completely defines the code and represents a compact way of describing a block code. If the encoding operation utilizes storage, then the encoder only needs to store the k rows of $\mathbf{G}$ instead of all 2^k code words of the code. For systematic codes, the encoder only stores the $\mathbf{P}$ submatrix; it doesn't need to store the identity portion of $\mathbf{G}$.

Visualization of a 6-Tuple Space

Figure 4 is a visualization of the eight code words presented in the (6, 3) example of Table 2. Since $k = 3$ and the code words are generated from linear combinations of the $k = 3$ independent 6-tuples in (14), the code words form a three-dimensional subspace. Figure 4 shows such a subspace completely occupied by the eight code words (large black circles); the coordinates of the subspace are purposely drawn to emphasize their nonorthogonality. Figure 4 is an attempt to illustrate the entire space, containing 64 6-tuples, even though there is no precise way to draw or construct such a model. Spherical layers or shells are shown around each code word. Each of the nonintersecting inner layers is a Hamming distance of one from its associated code word; each outer layer is a Hamming distance of two from its code word. Larger distances are not useful in this example. The two shells surrounding each code word are occupied by corrupted code words. There are six such points on each inner sphere because, for a (6, 3) code word, there are six ways to make a 1-b error. Thus there are a total of 48 such 6-tuples on the inner spheres (eight code words × six ways for each to make an error). These 48 6-tuples are distinct in the sense that each one can best be associated with only one code word, and therefore for the case of a single bit being received in error, the code word can be corrected. Later, we show that there is also one 2-b error pattern that can be corrected for this (6, 3) code. There is a total of $\binom{6}{2} = 15$ different 2-b error patterns that can be inflicted on each code word, but only one of them, in our example the 010001 error pattern, can be corrected. The other 14 2-b error patterns yield vectors that cannot be uniquely identified with just one code word. In the figure, all correctable (56) 1- and 2-b error-corrupted code words are shown as small black circles. Corrupted code words that cannot be corrected are shown as small clear circles. Figure 4 is useful for understanding desirable goals for a code system. We would like for the space to be filled with as many code words as possible (yielding efficient utilization of the added redundancy), and we would also like these code words to be as far away from one another as possible (yielding reduced vulnerability to noise). Obviously, these goals conflict.

Error Detection and the Parity-Check Matrix

At the decoder, a method of verifying the correctness of a received vector is needed. Let us define a matrix $\mathbf{H}$, called the *parity-check matrix*, that will help us decode the received vectors. For each $(k \times n)$ generator matrix $\mathbf{G}$, one can construct an $(n - k) \times n$ matrix $\mathbf{H}$, such that the rows of $\mathbf{G}$ are orthogonal to the rows of $\mathbf{H}$. Another way to express this orthogonality is to say that $\mathbf{G}\mathbf{H}^T = \mathbf{0}$, where $\mathbf{H}^T$ is the transpose of $\mathbf{H}$, and $\mathbf{0}$ is a $k \times (n - k)$ all-zeros matrix [1]. $\mathbf{H}^T$ is an $n \times (n - k)$ matrix (whose rows are the columns of $\mathbf{H}$). To fulfill the orthogonality requirements of a systematic code, the $\mathbf{H}$ matrix can be written as $\mathbf{H} = [\mathbf{I}_{n-k} | \mathbf{P}^T]$, where $\mathbf{I}_{n-k}$ represents an $(n - k) \times (n - k)$ identity submatrix and $\mathbf{P}$ represents the parity submatrix defined in (8). Since by this definition of $\mathbf{H}$, we see that $\mathbf{G}\mathbf{H}^T = \mathbf{0}$, and since each $\mathbf{U}$ is a linear combination of the rows of $\mathbf{G}$, then any vector $\mathbf{r}$ is a code word generated by the matrix $\mathbf{G}$, if and only if

$$\mathbf{r}\mathbf{H}^T = 0. \tag{16}$$

Equation (16) is the basis for verifying whether a received vector $\mathbf{r}$ is a valid code word.

Towards Error Correction: Syndrome Testing

In (1), the received vector $\mathbf{r}$ was expressed as the addition of a transmitted code word $\mathbf{U}$ and an error pattern

e. Following (16), we define a *syndrome vector* **S** of **r** as

$$\mathbf{S} = \mathbf{r}\mathbf{H}^T. \tag{17}$$

The syndrome (like the symptom of an ailment) is the result of a parity check (like a diagnostic test) performed on **r** to determine whether **r** is a member of the code word set. If, in fact, **r** is a valid code word, then from (16), its syndrome **S** must be an all-zeros vector; in other words, $\mathbf{r} = \mathbf{U}$ must have been generated by the matrix **G**. If **r** contains detectable errors, its syndrome will have some nonzero value. If the detected errors are correctable, the syndrome will have a nonzero value that can uniquely earmark the particular error pattern. A forward error-correcting decoder will then take action to correct the errors.

The Standard Array and Error Correction

The syndrome test gives us the ability to detect errors and to correct some of them. Let us arrange the 2^n n-tuples that represent possible received vectors in an array, called the *standard array*. This array can be thought of as an organizational tool or a filing cabinet that contains all of the possible vectors in the space, nothing missing, and nothing replicated. The first row contains the set of all the 2^k code words $\mathbf{U}_1, \mathbf{U}_2, \ldots, \mathbf{U}_{2^k}$ starting with the all-zeros code word designated $\mathbf{U}_1$. In this array, each row, called a *coset*, consists of an error pattern in the leftmost position, called a *coset leader*, followed by corrupted code words (corrupted by that error pattern). Thus the first column, made up of coset leaders, displays all of the correctable error patterns. The structure of the standard array for an (n, k) code, is

$$\begin{array}{ccccccc}
\mathbf{U}_1 & \mathbf{U}_2 & \cdots & \mathbf{U}_i & \cdots & \mathbf{U}_{2^k} \\
\mathbf{e}_2 & \mathbf{U}_2+\mathbf{e}_2 & \cdots & \mathbf{U}_i+\mathbf{e}_2 & \cdots & \mathbf{U}_{2^k}+\mathbf{e}_2 \\
\mathbf{e}_3 & \mathbf{U}_2+\mathbf{e}_3 & \cdots & \mathbf{U}_i+\mathbf{e}_3 & \cdots & \mathbf{U}_{2^k}+\mathbf{e}_3 \\
\vdots & \vdots & & \vdots & & \vdots \\
\mathbf{e}_j & \mathbf{U}_2+\mathbf{e}_j & \cdots & \mathbf{U}_i+\mathbf{e}_j & \cdots & \mathbf{U}_{2^k}+\mathbf{e}_j \\
\vdots & \vdots & & \vdots & & \vdots \\
\mathbf{e}_{2^{n-k}} & \mathbf{U}_2+\mathbf{e}_{2^{n-k}} & \cdots & \mathbf{U}_i+\mathbf{e}_{2^{n-k}} & \cdots & \mathbf{U}_{2^k}+\mathbf{e}_{2^{n-k}}
\end{array} \tag{18}$$

Note that code word $\mathbf{U}_1$ plays two roles. It is one of the code words (the all-zeros code word), as well as the error pattern $\mathbf{e}_1$, that is the pattern that introduces no errors so that $\mathbf{r} = \mathbf{U} + \mathbf{e}_1 = \mathbf{U}$. Since the array contains all the 2^n n-tuples in the space, each n-tuple appearing only once, and each coset or row contains 2^k n-tuples, we can compute the number of rows in the array by dividing the total number of entries by the number of columns. Thus, in any standard array, there are $2^n/2^k = 2^{n-k}$ cosets. At first glance, the benefits of this tool seem limited to *small* block codes, because for code lengths beyond $n = 20$ there are millions of n-tuples in V_n. Even for large codes, however, the standard array concept allows visualization of important performance issues, such as bounds on error-correction capability, as well as possible tradeoffs between error correction and detection.

The process of decoding a corrupted code word by first detecting and then correcting an error can be compared to a familiar medical analogy.

In the sections that follow, we show how the decoding algorithm replaces a received corrupted code word $\mathbf{r} = \mathbf{U} + \mathbf{e}$ with an estimate $\widehat{\mathbf{U}}$ of the valid code word **U**. If code word $\mathbf{U}_i$ is transmitted over a noisy channel, and the corrupting error pattern is a coset leader, then the received vector will be decoded correctly into the transmitted code word $\mathbf{U}_i$. If the error pattern is not a coset leader, an erroneous decoding will result [4], [5].

The Syndrome of a Coset

The name coset is short for "a set of numbers having a common feature." What do the members of a coset have in common? Each member has the same syndrome. We confirm this as follows: If $\mathbf{e}_j$ is the coset leader or error pattern of the jth coset, then $\mathbf{U}_i + \mathbf{e}_j$ is an n-tuple in this coset. From (17), the syndrome of this n-tuple can be written as

$$\mathbf{S} = \mathbf{r}\mathbf{H}^T = (\mathbf{U}_i + \mathbf{e}_j)\mathbf{H}^T = \mathbf{U}_i\mathbf{H}^T + \mathbf{e}_j\mathbf{H}^T. \tag{19}$$

Since $\mathbf{U}_i$ is a valid transmitted code word, then $\mathbf{U}_i\mathbf{H}^T = \mathbf{0}$, since the parity check matrix **H** was constructed with this feature in mind. We can therefore express (19) as

$$\mathbf{S} = \mathbf{r}\mathbf{H}^T = \mathbf{e}_j\mathbf{H}^T. \tag{20}$$

Thus, the syndrome test, performed on either a corrupted code vector or on the error pattern that caused it, yields the same syndrome. Equation (20) establishes that the syndrome is in fact only responding to the error pattern, which was similarly shown in (5) for a simple parity code. An important property of linear block codes, fundamental to the decoding process, is that the mapping between correctable error patterns and syndromes is one to one. The syndrome for each coset is different from that of any other coset in the code; it is the syndrome that is used to estimate the error pattern, which then allows for the errors to be corrected.

Locating the Error Pattern

Returning to the (6, 3) code example, we arrange the

$2^6 = 64$ 6-tuples in a standard array as shown in Figure 5. The valid code words are the eight vectors in the first row, and the correctable error patterns are the seven nonzero coset leaders in the first column. Note that all 1-b error patterns are correctable. Also note that after exhausting all 1-b error patterns, there remains some error-correcting capability since we have not yet accounted for all 64 6-tuples. There is still one unassigned coset leader; therefore, there remains the capability of correcting one additional error pattern. We have the flexibility of choosing this error pattern to be any of the n-tuples in the remaining coset. In Figure 5, this final correctable error pattern was chosen, somewhat arbitrarily, to be the 2-b error pattern 010001. The error-correcting task performed by the decoder can be implemented to yield correct messages if, and only if, the error pattern caused by the channel is one of the coset leaders. For the (6, 3) code example, we now use (20) to determine the syndrome (symptom) corresponding to each correctable error pattern (ailment), by computing $\mathbf{e}_j \mathbf{H}^T$ for each coset leader, as follows:

$$\mathbf{S} = \mathbf{e}_j \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}. \tag{21}$$

The results are listed in Table 3. Since each syndrome in the table has a one-to-one relationship with the listed error patterns, solving for a syndrome earmarks the particular error pattern corresponding to that syndrome [6].

Table 3. Syndrome lookup table.

Error pattern e	Syndrome S
000000	0 0 0
000001	1 0 1
000010	0 1 1
000100	1 1 0
001000	0 0 1
010000	0 1 0
100000	1 0 0
010001	1 1 1

Error Correction Decoding

Given a received vector $\mathbf{r}$ at the input of the decoder in Figure 1, we summarize the procedure for deciding on $\hat{\mathbf{U}}$ and finally on $\hat{\mathbf{m}}$ as follows: 1) calculate the syndrome of $\mathbf{r}$ using $\mathbf{S} = \mathbf{r}\mathbf{H}^T$ and 2) use Table 3 to locate the coset leader (error pattern) $\mathbf{e}_j$, whose syndrome equals $\mathbf{r}\mathbf{H}^T$. This error pattern is assumed to be the corruption caused by the channel and will be our estimate $\hat{\mathbf{e}}$ of the error (3) An estimate of the code word $\hat{\mathbf{U}}$ is identified as $\hat{\mathbf{U}} = \mathbf{r} + \hat{\mathbf{e}}$. We can say that the decoder obtains an estimate of the transmitted code word by removing an estimate of the error $\hat{\mathbf{e}}$ (in modulo-2 arithmetic, the act of removal is effected via addition). This step can be written as

$$\hat{\mathbf{U}} = \mathbf{r} + \hat{\mathbf{e}} = (\mathbf{U} + \mathbf{e}) + \hat{\mathbf{e}} = \mathbf{U} + (\mathbf{e} + \hat{\mathbf{e}}). \tag{22}$$

If the estimated error pattern is the same as the actual error pattern, that is, if $\hat{\mathbf{e}} = \mathbf{e}$, then the estimate $\hat{\mathbf{U}}$ is equal to the transmitted code word $\mathbf{U}$. However, if the error estimate is incorrect, the decoder will choose a code word that was not transmitted, resulting in a decoding error.

As an example from the (6, 3) code, assume that code word $\mathbf{U} = 101110$ corresponding to $\mathbf{m} = 110$ (see Table 2) is transmitted and that the vector $\mathbf{r} = 001110$ is received. From (17) we compute the syndrome as

$$\mathbf{S} = [001110]\,\mathbf{H}^T = 100. \tag{23}$$

From Table 3, we can verify that the error pattern is $\mathbf{e} = 100000$. Then, using (22), the corrected vector is estimated as

$$\hat{\mathbf{U}} = \mathbf{r} + \hat{\mathbf{e}} = 001110 + 100000 = 101110. \tag{24}$$

Since in this example, the estimated error pattern is the actual error pattern, the error correction procedure yields $\hat{\mathbf{U}} = \mathbf{U}$, which means that the output $\hat{\mathbf{m}}$ will correspond to the actual message 110. Note that the process of decoding a corrupted code word by first detecting and then correcting an error can be compared to a familiar medical analogy. A patient $\mathbf{r}$ (potentially corrupted code word) enters a medical facility (decoder). The examining physician performs a diagnostic test (multiplies $\mathbf{r}$ by $\mathbf{H}^T$) to find a symptom (syndrome). Imagine that the physician finds characteristic spots on the patient's X rays. An experienced physician would immediately recognize the correspondence between the symp-

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

▲ *5. Example of a standard array for a (6, 3) code.*

tom and the ailment, for example tuberculosis. A novice physician might have to refer to a medical handbook (Table 3) to associate the symptom (syndrome) with the ailment (error pattern). The final step is to provide medication $\widehat{\mathbf{e}}$. If $\widehat{\mathbf{e}}$ is the proper medication (if $\widehat{\mathbf{e}} = \mathbf{e}$), then the ailment is removed, as seen in (22). In the context of binary codes and the medical analogy, (22) and (24) reveal an unusual type of medicine practiced here. The patient is cured by reapplying the original ailment, a process that works because in the binary field $1 + 1 = 0$.

Decoder Implementation

When the code is short as in the case of the (6, 3) code, the decoder can be implemented with simple circuitry. The steps that such a circuit must take are: 1) calculate the syndrome, 2) locate the error pattern corresponding to that syndrome, and 3) modulo-2 add the estimated error pattern to the received vector to yield an estimate of the corrected vector. Consider the circuit in Figure 6, made up of exclusive-OR gates and AND gates that can accomplish these decoder steps for any single-error pattern in the (6, 3) code. From Table 3 and (17), we can write an expression for the syndrome bits s_1, s_2, s_3 in terms of the received code word bits $r_1, \ldots, r_6$ as

$$\mathbf{S} = \mathbf{r}\mathbf{H}^T \tag{25a}$$

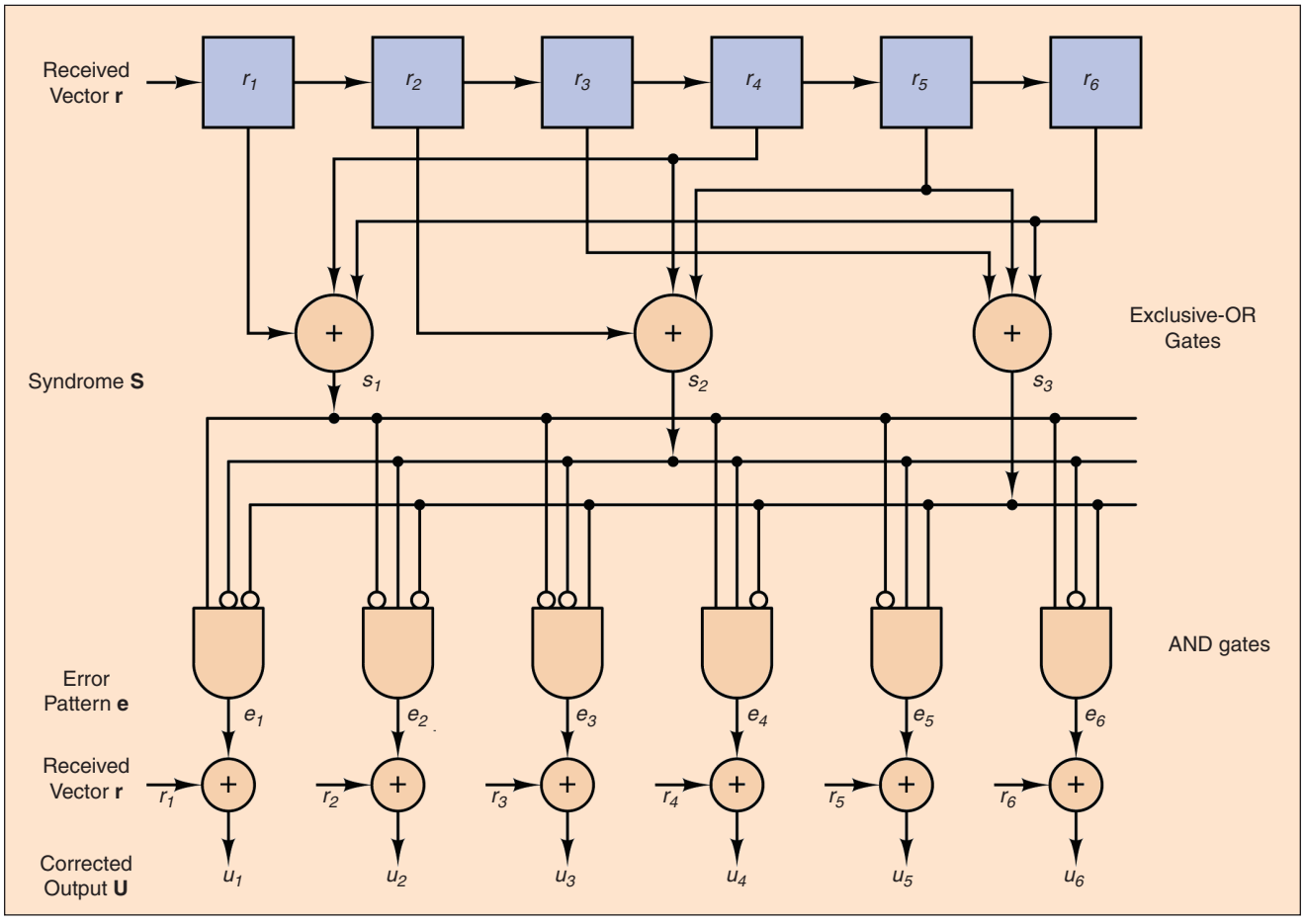
$$\mathbf{S} = [s_1\ s_2\ s_3] = [r_1\ r_2\ r_3\ r_4\ r_5\ r_6] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \tag{25b}$$

and

$$\begin{aligned} s_1 &= r_1 + r_4 + r_6 \\ s_2 &= r_2 + r_4 + r_5 \\ s_3 &= r_3 + r_5 + r_6. \end{aligned} \tag{26}$$

We use these syndrome expressions for wiring up the circuit in Figure 6. The exclusive-OR gate provides the same operation as modulo-2 arithmetic and hence uses the same symbol. A small circle at the termination of any line entering the AND gate indicates the logic complement of the signal.

The corrupted signal $\mathbf{r}$ enters the decoder at two places simultaneously. At the upper part of the circuit, the syndrome $\mathbf{S}$ is computed, and at the lower part that syndrome is transformed to its corresponding error pattern $\mathbf{e}$. The error is removed by adding it back



▲ *6. Implementation of the (6, 3) decoder.*

to the received vector yielding the corrected code word **U**. Note that, for tutorial reasons, Figure 6 has been drawn to emphasize the algebraic decoding steps, calculation of syndrome, error pattern, and corrected output. For real circuitry, the decoder would not need to deliver the entire code word; its output would consist of the message bits only. Hence, the Figure 6 circuitry becomes simplified by eliminating the gates that are shown with shading. For longer codes such an implementation is very complex, and the preferred decoding techniques conserve circuitry by using a sequential approach instead of this parallel method [2]. It is important to emphasize that Figure 6 has been configured to detect and correct only single-error patterns for the (6, 3) code. Error control for a double-error pattern would require additional circuitry.

Cyclic Codes

Cyclic codes are a subset of linear block codes based on an algebraic structure that leads to strong error correcting capabilities and to computationally efficient encoding and decoding algorithms. Important classes of cyclic codes are the binary Bose, Chaudhuri, and Hocquenghem (BCH) codes and the nonbinary Reed-Solomon (R-S) codes. The cyclic nature of such codes can be stated as follows: If **U** is a code word (which is made up of n elements) expressed as

$$\mathbf{U} = u_0, u_1, u_2, \ldots, u_{n-1} \tag{27}$$

then a single cyclic (or end around) shift of the code elements, written as

$$\mathbf{U}^{(1)} = u_{n-1}, u_0, u_1, \ldots, u_{n-2} \tag{28}$$

is also a code word. A rich set of properties and structure can be extracted from this simple relationship. The cyclic code properties are best understood by mapping data sequences and code words to polynomials in the indeterminate X, in a manner similar to the Z-transform (or delay-transform) [7] for sampled data sequences. For this article, the indexing scheme that was used for elements and vectors (data, code bits, parity, and errors) up until this section, started with the index 1, as seen in (6)–(8). Counting in this natural way generally facilitates the intuitive understanding of error-correction coding. When treating the subject of cyclic codes, where polynomials are used to represent sequences of elements, a different indexing scheme is preferred. Whenever polynomials are used, we typically start the counting of elements with the index 0, as seen in (27). With this scheme, a code word of the form shown in (27) can be more naturally mapped to its corresponding polynomial as

$$\mathbf{U}(X) = u_0 + u_1 X + u_2 X^2 + \cdots + u_{n-1} X^{n-1}. \tag{29}$$

The coefficients of the polynomial are the corresponding elements of the sequence. In this mapping, the indeterminate X uses positive-integer exponents (as opposed to the negative integers used in the Z-transform). In this notation, all code words can be represented by polynomials of degree $n-1$ or less, with coefficients drawn from the binary field (the degree indicates the value of the highest-order exponent). For nonbinary codes, the definition must include nonbinary coefficients. The polynomial corresponding to (28) is written as

$$\mathbf{U}^{(1)}(X) = u_{n-1} + u_0 X + u_1 X^2 + \cdots + u_{n-2} X^{n-1}. \tag{30}$$

We note that the cyclically shifted form shown in (30) can be obtained by using X as a delay operator followed by a set of trivial manipulations. We first form the polynomial $X\mathbf{U}(X)$ to obtain a linear shift of the elements and then add the terms $u_{n-1} + u_{n-1}$. In the binary field, the added terms sum to zero so this addition does not alter the shifted polynomial. We write this as

$$\begin{aligned} X\mathbf{U}(X) &= u_0 X + u_1 X^2 + \cdots + u_{n-2} X^{n-1} + u_{n-1} X^n \\ &= u_{n-1} + u_0 X + u_1 X^2 + \cdots + u_{n-2} X^{n-1} \\ &\quad + u_{n-1} X^n + u_{n-1} \\ &= u_{n-1} + u_0 X + u_1 X^2 + \cdots + u_{n-2} X^{n-1} \\ &\quad + u_{n-1}(X^n + 1). \end{aligned} \tag{31}$$

We now point out that we can form the cyclically shifted polynomial in (30) from the linearly shifted version in (31), as the remainder of $X\,\mathbf{U}(X)$ modulo-$(X^n + 1)$. We write this as

$$\begin{aligned} \mathbf{U}^{(1)}(X) &= X\mathbf{U}(X) \text{modulo-}(X^n + 1) \\ &= u_{n-1} + u_0 X + u_1 X^2 + \cdots + u_{n-2} X^{n-1}. \end{aligned} \tag{32}$$

Note that the modulo-$(X^n + 1)$ expression is sometimes seen in the literature as modulo-$(X^n - 1)$. In the binary field, either one is identical because $+1 = -1$. The expression in (32), often called the *residue reduction* operation, consists of dividing the right side of (31) by $X^n + 1$, then discarding the quotient polynomial and retaining the remainder polynomial. By repeated application of the cyclic shift using this residue reduction operation, we can similarly obtain the polynomial corresponding to all the cyclic shifts of any code word in the subspace.

The Generator Polynomial

In an (n, k) cyclic code there is one code word polynomial $\mathbf{g}(X)$ having minimum degree equal to $n-k$. Without proof, we state that $\mathbf{g}(X)$ must be a factor of the polynomial $X^n + 1$ [2], [8]. Because $\mathbf{g}(X)$ can be

used to generate all code words in the code subspace (in a manner similar to the operation of the generator matrix $\mathbf{G}$ described earlier), this polynomial is called the *generator polynomial*. The form of $\mathbf{g}(X)$ is

$$\mathbf{g}(X) = 1 + g_1 X + g_2 X^2 + \cdots + g_{n-k-1} X^{n-k-1} + X^{n-k}. \quad (33)$$

For this polynomial to be of minimum degree, the coefficient of X^0 and X^{n-k} must each be one while the coefficient of any term having a power higher than X^{n-k} must be zero. These zero-valued coefficients of the higher-powered terms represent the zeros in a code sequence needed to ensure that each code word is made up of exactly n elements.

There is a one-to-one correspondence between the basis vectors comprising a generator matrix $\mathbf{G}$ and the polynomials formed by cyclically shifted versions of $\mathbf{g}(X)$. Here is how we illustrate the correspondence. By the properties of cyclic codes, since $\mathbf{g}(X)$ is a proper code word polynomial, then any cyclic shift of $\mathbf{g}(X)$ is also a code word. In this discussion, we limit the number of shifts to $k-1$, the value that brings the degree of the shifted $\mathbf{g}(X)$ up to $n-1$. Each cyclic shift of $\mathbf{g}(X)$ is performed by first multiplying $\mathbf{g}(X)$ by X (linear shift) and then end-around shifting the zero coefficient of the X^n term so that it becomes the new coefficient of the X^0 term. Consequently we can say that for these $k-1$ shifts, the cyclic shift formed by the residue reduction in (32) yields the same result as the linear shift indicated as

$$X^i \mathbf{g}(X) \text{ modulo-}(X^n + 1) = X^i \mathbf{g}(X) \quad \text{where } i = 0, 1, 2, \ldots, k-1. \quad (34)$$

These i-shifted versions of the generator polynomial $\mathbf{g}(X)$ in (34) constitute a basis set of the code in the same manner as the row space of the generator matrix $\mathbf{G}$ provides a basis set in (7). In a linear code, the weighted sum of proper code words yields another code word. Hence, since any $X^i \mathbf{g}(X)$ of the form cited in (34), is a proper code word polynomial, then we can represent it in terms of a message sequence as

$$\begin{aligned}\mathbf{U}(X) &= (m_0 + m_1 X + m_2 X^2 + \cdots + m_{k-1} X^{k-1})\, \mathbf{g}(X) \\ &= m_0 \mathbf{g}(X) + m_1 X \mathbf{g}(X) + m_2 X^2 \mathbf{g}(X) \\ &\quad + \cdots + m_{k-1} X^{k-1} \mathbf{g}(X) \\ &= \mathbf{m}(X)\, \mathbf{g}(X). \end{aligned} \quad (35)$$

Note the similarity between the formation of a code word polynomial using the generator polynomial $\mathbf{g}(X)$ in (35) and the formation of a code word using the generator matrix $\mathbf{G}$ in (7). Since for a binary code, there are 2^k possible polynomials described by (35), and knowing that an (n, k) code contains precisely 2^k code words, we can conclude that (35) describes all proper code word polynomials in the cyclic code. That is, every code word polynomial can be formed as a product of a data polynomial $\mathbf{m}(X)$ and the generator polynomial $\mathbf{g}(X)$. One might recall from knowledge of linear systems, that the product of polynomials is equivalent to the convolution of their corresponding coefficients. Thus a tapped delay line convolver, with weights equal to the coefficients of the generator polynomial, can form the code words of a cyclic code [9].

Systematic Cyclic Code

The generator polynomial can also be used to form a systematic form of the code word polynomial. From the expression derived in (35), we know that a proper code word is a multiple of $\mathbf{g}(X)$. Thus, dividing a proper code word by $\mathbf{g}(X)$ yields a quotient (message polynomial) and a zero remainder. In general, if we divide any polynomial by $\mathbf{g}(X)$, the remainder must be a polynomial of degree less than the divisor $\mathbf{g}(X)$; that is, it must be less than degree $(n-k)$. We can encode a data polynomial $\mathbf{m}(X)$ into a systematic code word polynomial by first multiplying $\mathbf{m}(X)$ by X^{n-k} (representing $n-k$ shifts) and then performing polynomial division, written as

$$\frac{X^{n-k}\mathbf{m}(X)}{\mathbf{g}(X)} = \mathbf{q}(X) + \frac{\mathbf{p}(X)}{\mathbf{g}(X)} \quad (36)$$

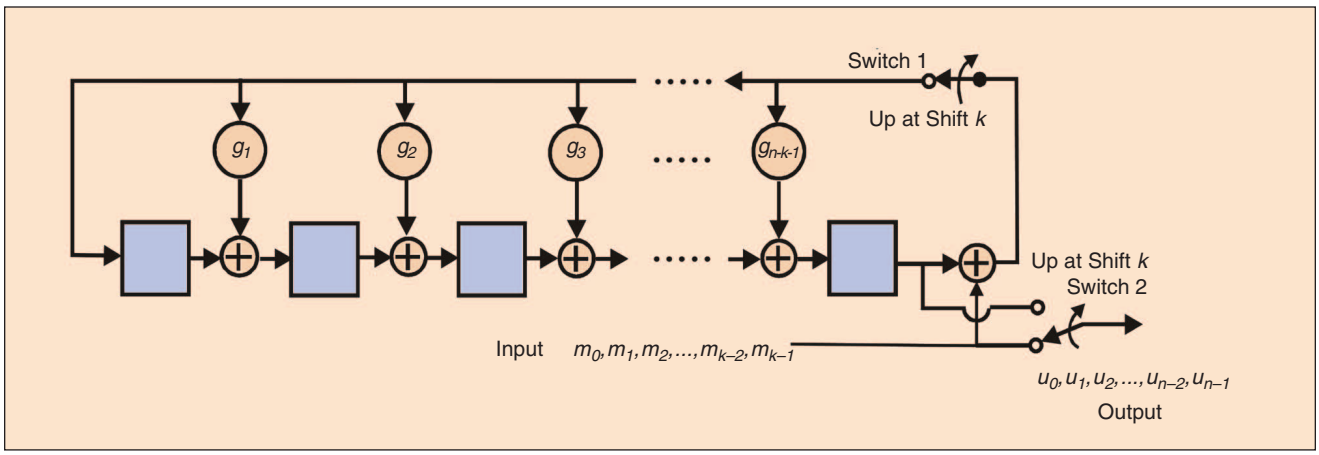
where $\mathbf{q}(X)$ and $\mathbf{p}(X)$ are the quotient and remainder polynomials, respectively. When each side of this expression is multiplied by the divisor polynomial we obtain an equivalent form known as the *Euclidian division algorithm*, written as

$$X^{n-k}\mathbf{m}(X) = \mathbf{q}(X)\mathbf{g}(X) + \mathbf{p}(X). \quad (37)$$

We next add the remainder polynomial $\mathbf{p}(X)$ to both sides of (37), yielding $\mathbf{U}(X)$ in systematic form, as follows:

$$\begin{aligned}\mathbf{p}(X) + X^{n-k}\,\mathbf{m}(X) &= \mathbf{q}(X)\,\mathbf{g}(X) + \mathbf{p}(X) + \mathbf{p}(X) \\ &= \mathbf{q}(X)\,\mathbf{g}(X) = \mathbf{U}(X). \end{aligned} \quad (38)$$

Since the $\mathbf{p}(X) + \mathbf{p}(X)$ sums to zero, then $\mathbf{U}(X)$ in (38) is seen to be a multiple of $\mathbf{g}(X)$; hence $\mathbf{U}(X)$ is a proper code word polynomial. We note that the polynomial $\mathbf{p}(X)$ contains powers of X from zero through $n-k-1$ (thus there are $n-k$ coefficients), while the shifted polynomial $X^{n-k}\,\mathbf{m}(X)$ contains the same coefficients as $\mathbf{m}(X)$, but positioned at powers of X from $n-k$ through $n-1$. When we add the prefix (parity) polynomial $\mathbf{p}(X)$ to the shifted message polynomial $X^{n-k}\,\mathbf{m}(X)$, there is no overlap in the powers of X. We can think of the k coefficients of the polynomial $X^{n-k}\,\mathbf{m}(X)$ as the message that has been right shifted $n-k$ positions to make room for parity bits, as the $n-k$ coefficients of the polynomial $\mathbf{p}(X)$.



▲ *7. Linear feedback (n−k stage) shift register.*

Encoding with a Linear Feedback Shift Register

The linear feedback shift register (LFSR) shown in Figure 7 performs the division operation as well as the $(n-k)$-bit shift presented in (36) and (37). This LFSR encoding process operates in the following manner. For the first k shifts of the input data sequence, the feedback path in the LFSR is enabled (switch 1 is closed); also, switch 2 is down. During this interval, the LFSR divides by $\mathbf{g}(X)$, while the output switch simultaneously passes the input data sequence to the output port. At the end of the kth input shift, the register contains the parity bits of the code word polynomial. For the next $n-k$ shifts, the feedback path is disabled (switch 1 is opened) during which time the $n-k$ parity bits from the register are transferred to the output port through switch 2, now moved to the up position. Note that the generator polynomial $\mathbf{g}(X)$ (analogous to the generator matrix $\mathbf{G}$) completely defines the cyclic code. $\mathbf{g}(X)$ is the minimum degree code word polynomial and thus represents the most compact description of a cyclic code.

One may recognize Figure 7 as the dual form of the recursive filter structure used in standard digital signal processing applications. The traditional recursive filter forms a feedback term as a weighted sum of states stored in a tapped delay line. This dual form is more desirable for high-speed implementation because of the simultaneous operation of the distributed summations between the registers, as opposed to the adder tree required in the tapped-delay feedback path.

Figure 8 demonstrates the use of an LFSR to encode the message $\mathbf{m} = 1\,0\,1\,1$ for a $(7, 4)$ cyclic code (in systematic form) using the generator polynomial $\mathbf{g}(X) = 1 + X + X^3$. The output code word is $\mathbf{U} = 1\,0\,0\,1\,0\,1\,1$, where the rightmost four bits represent the message. Table 4 tabulates the state of the LFSR, the input and output queues, and the switch settings, for seven successive operating cycles.

One should note that the wiring of the LFSR in Figures 7 and 8 correspond to the coefficients of the generator polynomial $\mathbf{g}(X)$, where a coefficient 1 or 0 corresponds to the presence or absence of a wire, respectively. Sometimes a schematic such as the one in Figure 8 is used to specify a particular cyclic code, though a more compact way is to represent the code with its generator polynomial.

Table 4. Description of LFSR encoding of the message (1 0 1 1).

Shift Number	S1	S2	Input Queue	Register Contents	Feedback	Output
0	D	D	1 0 1 1	0 0 0	0	1
1	D	D	1 0 1	1 1 0	1	1
2	D	D	1 0	1 0 1	1	0
3	D	D	1	1 0 0	1	1
4	U	U	—	1 0 0	—	0
5	U	U	—	0 1 0	—	0
6	U	U	—	0 0 1	—	1

S1 = Switch 1, S2 = Switch 2, U = Up, D = Down

Error Detection with an $(n-k)$ Stage LFSR

A code word that has been altered by AWGN can be described as a received polynomial $\mathbf{r}(X)$, which is the sum of the transmitted polynomial $\mathbf{U}(X)$ and an error polynomial $\mathbf{e}(X)$, written as

$$\mathbf{r}(X) = \mathbf{U}(X) + \mathbf{e}(X). \tag{39}$$

The decoder must perform an error-detection test on the received polynomial $\mathbf{r}(X)$ to determine if it is a proper code word. This proceeds as follows: First, the decoder forms a *syndrome polynomial* $\mathbf{S}(X)$ as the remainder of $\mathbf{r}(X)$ modulo-$\mathbf{g}(X)$. Note that $\mathbf{S}(X)$ serves the same function as the syndrome vector $\mathbf{S}$ in (19) and (20). Since the remainder of the transmitted polynomial $\mathbf{U}(X)$ modulo-$\mathbf{g}(X)$ is zero, then $\mathbf{S}(X)$ represents a test on the error polynomial $\mathbf{e}(X)$. In the context of the

medical analogy proposed earlier, this test is the cyclic-code "diagnostic procedure" for finding the symptom of an error. We represent this procedure as

$$\begin{aligned}\mathbf{S}(X) &= \mathbf{r}(X) \text{ modulo-}[\mathbf{g}(X)] \\ &= \mathbf{U}(X)\text{modulo-}[\mathbf{g}(X)] + \mathbf{e}(X) \text{ modulo-}[\mathbf{g}(X)] \\ &= \mathbf{e}(X) \text{ modulo-}[\mathbf{g}(X)]. \end{aligned} \tag{40}$$

Whenever a channel-induced error results in a received polynomial $\mathbf{r}(X)$ being a noncode word, then the syndrome polynomial $\mathbf{S}(X)$ will be nonzero and its nonzero Hamming weight is an indication of an error. The same LFSR that formed the parity sequence at the encoder can be used to generate the syndrome polynomial $\mathbf{S}(X)$ at the decoder. The n bits of the received $\mathbf{r}(X)$ are processed by such an $n-k$ stage LFSR (similar to the one in Figure 8), and after the n-shift operations, the contents of the register will be the coefficients of $\mathbf{S}(X)$. The syndrome along with the structure (embedded in the generator polynomial) can be used to locate the error pattern.

Hamming Codes and BCH Codes

Hamming codes are single-error correcting codes that exist for the $(n, k, 1)$ set described by

$$(n,\ k,\ 1) = (2^m - 1,\ 2^m - 1 - m,\ 1) \tag{41}$$

where m is the number of parity bits in the code. The encoder and decoder for a Hamming code can be implemented with an m-stage LFSR where m is the degree of the generator polynomial (a factor of the polynomial $X^n + 1$) and $n = 2^m - 1$.

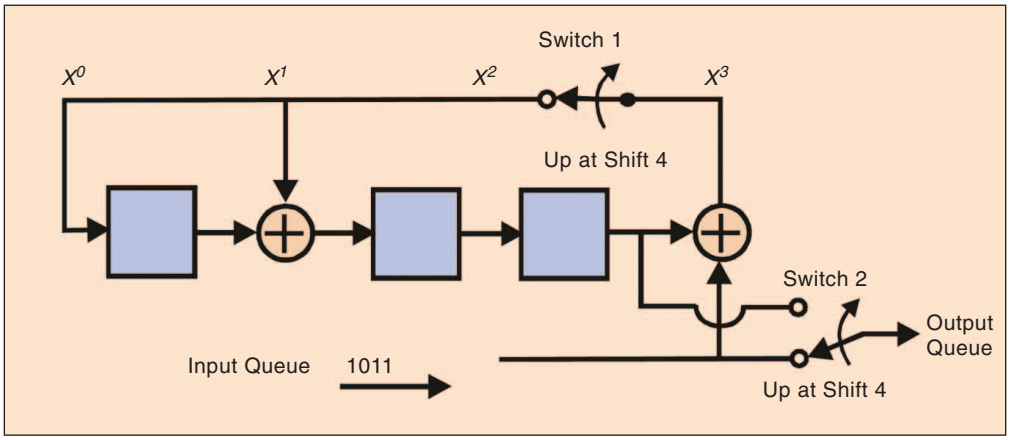
A generalization of the Hamming codes leads to the BCH codes [10]–[12], which represent a class of cyclic codes that offer a large range of block length, code rates, and error correcting strength. A t-error correcting BCH code exists for the (n, k, t) set described approximately by

For R-S codes, one can say that the codes "prefer" to see the noise occur in bursts. There is a kind of "economy of scale" at work.

$$(n,\ k,\ t) \approx (2^m - 1,\ 2^m - 1 - mt,\ t) \tag{42}$$

where the degree of the generating polynomial as well as number of parity bits in the code is less than or equal to mt. We use (42) as an upper bound for the number of parity bits required to correct t binary errors in a code word polynomial of length $2^m - 1$ bits. Since the BCH code is a cyclic code, the encoding process is performed by a standard LFSR in the same manner that we form any cyclic code.

The operational difference for a BCH code is in the decoding segment performed at the receiver. BCH decoders, except for the simplest codes, do not use parity check matrices and syndrome tables. These earlier-described techniques are only implemented for short simple codes. For each received code word, typical BCH decoders perform a sequence of algebraic operations that yield the number of errors and the error locations. Rather than deriving the underlying mathematics for BCH codes, we will simply state the properties of the polynomials and show how they enable the design of $\mathbf{g}(X)$ as well as the implementation of the error-detection and correction algorithms. Bear in mind that the processing mechanism used to select the generating polynomial and implement the decoding algorithms perform Galois field arithmetic, a detail that is not needed to understand the overall process. One concept that we do need to examine to understand the encoding and decoding process is the representation of



▲ 8. LFSR for $g(X) = 1 + X + X^3$.

the elements in a Galois field. The nonzero elements of GF(2^n) are the roots of the polynomial $X^n + 1$. In the complex field, the roots of $X^n - 1$ form the periodic sequence exp $(j2\pi/n)$, exp $(j4\pi/n)$, exp $(j6\pi/n)$, and so forth, which we write compactly as $w^1, w^2, w^3, \ldots$, where $w = \exp(j2\pi/n)$. Recall that in the binary field, $X^n - 1$ is equal to $X^n + 1$. We often refer to the roots of unity by their exponent, as we do for instance when referring to the kth bin of a fast-Fourier transform (FFT). In like manner the roots of $X^n + 1$ in a GF form a periodic sequence written as $\alpha, \alpha^2, \alpha^3, \ldots$ that are also referred to by their exponent. In the error location process described later, the exponents of a set of roots will be the indicator of the error locations.

Decoding of BCH Codes

The generator polynomial $\mathbf{g}(X)$ is selected as a product of factors of $X^n + 1$, with $n = 2^m - 1$, such that the composite polynomial has a set of $2t$ known roots, $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$ denoted here as $\beta_1, \beta_2, \ldots, \beta_{2t}$. The binary polynomials containing these roots are tabulated in many textbooks [1], [2]. We have already described in (40) how the remainder modulo-$\mathbf{g}(X)$ of the received code word polynomial is in fact the syndrome of the error polynomial. The relationship between the error polynomial $\mathbf{e}(X)$ and the syndrome polynomial $\mathbf{S}(X)$ is shown as

$$\mathbf{e}(X) = \mathbf{q}(X)\,\mathbf{g}(X) + \mathbf{S}(X). \tag{43}$$

Evaluating both sides of (43) at the known roots $X = \beta_i$ of the generator $\mathbf{g}(X)$, we get

$$\mathbf{e}(\beta_i) = \mathbf{q}(\beta_i)\,\mathbf{g}(\beta_i) + \mathbf{S}(\beta_i) = \mathbf{0} + \mathbf{S}(\beta_i). \tag{44}$$

Since $\mathbf{g}(\beta_i) = \mathbf{0}$, when we evaluate the syndrome polynomial at values of X equal to the roots β_i, we obtain the values that are the same as those we would obtain by evaluating the error polynomial. Hence, these values contain information (or clues) about the error polynomial. We compute the $2t$ scalar syndromes by first forming $\mathbf{S}(X)$ from the received vector and then evaluating it at the roots β_i as follows:

$$S_i = \mathbf{S}(X)|_{X=\beta_i} = \mathbf{S}(\beta_i): \quad i = 1, 2, \ldots, 2t. \tag{45}$$

In general, the error polynomial $\mathbf{e}(X)$ is of the form:

$$\mathbf{e}(X) = \{e_i\} = \sum_{j=1}^{\nu} X^{i(j)} \tag{46}$$

where the jth error is located at indicator-index $i(j)$ carried by the exponent of X, and ν represents the number of errors in the code word. Note that the decoder has no knowledge of either the number of errors ν or their location $i(j)$. A syndrome S_i, computed by evaluating the error polynomial at the ith power of α, is related to the error sequence by the relationship written as

$$\begin{aligned} S_i &= \sum_{j=1}^{\nu} (\alpha^i)^{i(j)} = \sum_{j=1}^{\nu} (\alpha^{i(j)})^i \\ &= \sum_{j=1}^{\nu} (\beta_j)^i, \quad i = 1, 2, \ldots, 2t. \end{aligned} \tag{47}$$

In (47) we have interchanged the ith power with the error location index $i(j)$ and simplified notation by replacing $\alpha^{i(j)}$ by β_j (similar to an earlier simplification). Suppose we have a double error correcting code, say a (15, 7, 2) code, and an error polynomial with nonzero errors at the two positions $i(1)$ and $i(2)$, written as

$$\mathbf{e}(X) = X^{i(1)} + X^{i(2)}. \tag{48}$$

We form the scalar values S_i, for $i = 1, 2, \ldots, 2t$, from the syndrome polynomial $\mathbf{S}(X)$ that for this example is an eighth-order polynomial. Then for this double-error correcting example ($t = 2$), $\mathbf{S} = S_1, S_2, S_3, S_4$. When we evaluate the eighth-order polynomial $\mathbf{S}(X)$ at the four known root values β_i we obtain a set of four simultaneous nonlinear equations in two unknowns written as

$$\begin{aligned} S_1 &= \beta_1^{i(1)} + \beta_1^{i(2)} \\ S_2 &= \beta_2^{i(1)} + \beta_2^{i(2)} \\ S_3 &= \beta_3^{i(1)} + \beta_3^{i(2)} \\ S_4 &= \beta_4^{i(1)} + \beta_4^{i(2)}. \end{aligned} \tag{49}$$

In the general case, the decoder does not know the number of errors or the number of exponents to be determined. Note that for this double-error example, (49) indicates that there are two unknowns, the exponents $i(1)$ and $i(2)$. Why are there two unknowns (errors) but four equations? The answer is that in a binary code two of the equations are dependent and can be discarded. But, in a nonbinary code with two errors, there are four unknowns—the two error positions and the two error values. A related question is, suppose we only had one error but four equations? The answer is the same; some of the equations are dependent and are discarded.

A decoding algorithm that solves for the unknown exponents is termed an error locating process. Once the number of exponents and their values has been determined, error correction with a binary code is a matter of adding a one to the identified location(s) in the received vector. One technique for solving the nonlinear set of relationships in (49) is to define an auxiliary polynomial known as the error locating polynomial $\boldsymbol{\sigma}(X)$ and use the syndrome values to solve for the coefficients $\boldsymbol{\sigma}_i$. The error locating polynomial is defined to have roots equal to the inverse of the error location numbers λ_i. The $\boldsymbol{\sigma}(X)$ polynomial is represented as

$$\boldsymbol{\sigma}(X) = (1-\lambda_1 X)(1-\lambda_2).....(1-\lambda_\nu X)$$
$$= \sigma_0 + \sigma_1 X + \cdots + \sigma_{\nu-1} X^{\nu-1} + \sigma_\nu X^\nu. \quad (50)$$

To determine the coefficients of the $\boldsymbol{\sigma}(X)$, we first form an infinite series $\mathbf{S}_\infty(X)$ from an extended list of syndrome coefficients, shown as

$$\mathbf{S}_\infty(X) = \sum_{i=1}^{\infty} S_i X^{(i-1)}. \quad (51)$$

Substituting (47), the expression for the syndromes S_i into (51), and rearranging the sum, we can write

$$\mathbf{S}_\infty(X) = \sum_{i=1}^{\infty} X^{(i-1)} \sum_{j=1}^{\nu} (\beta_j)^i$$
$$= \sum_{i=1}^{\infty} X^{(i-1)} \sum_{j=1}^{\nu} (\beta_j)^{(i-1)} \beta_j \quad (52)$$
$$= \sum_{j=1}^{\nu} \beta_j \sum_{i=0}^{\infty} (\beta_i X)^i = \sum_{j=1}^{\nu} \beta_j \frac{1}{1-\beta_j X}. \quad (53)$$

In (53) we have replaced each of the geometric series (connected to each error) with its closed form and recognize the resulting sum as the partial fraction expansion of the infinite series. If we now form the product of $\mathbf{S}_\infty(X)$ and $\boldsymbol{\sigma}(X)$, we find that the factors of $\boldsymbol{\sigma}(X)$ cancel the denominator terms of (53)

$$\mathbf{S}_\infty(X)\,\boldsymbol{\sigma}(X) = \sum_{j=1}^{\nu} \beta_j \frac{1}{1-\beta_j X} \prod_{\ell=1}^{\nu} (1-\beta_\ell X)$$
$$= \sum_{j=1}^{\nu} \beta_j \prod_{\substack{\ell=1 \\ \ell \neq j}}^{\nu} (1-\beta_\ell X). \quad (54)$$

From (54) we concluded thatz, due to the denominator canceling, the product of the infinite polynomial $\mathbf{S}_\infty(X)$ and the finite polynomial $\boldsymbol{\sigma}(X)$ is a finite polynomial of degree $\nu - 1$, where ν is the number of errors in the sequence. Consequently, the coefficients formed by the product in (54) for the powers of X greater than or equal to ν are precisely zero. The coefficients of any power of X formed by the polynomial product are seen to be a weighted sum of the syndrome coefficients. Setting the coefficients of the powers of X from ν to $2t$ to zero, leads to the collection of equations relating the error locating coefficients $\boldsymbol{\sigma}_i$ and the scalar syndromes S_i shown as

$$\begin{bmatrix} S_1 & S_2 & \cdots & S_t \\ S_2 & S_3 & \cdots & S_{t+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_t & S_{t+1} & \cdots & S_{2t-1} \end{bmatrix} \begin{bmatrix} \sigma_t \\ \sigma_{t-1} \\ \vdots \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_{t+1} \\ -S_{t+2} \\ \vdots \\ -S_{2t} \end{bmatrix}. \quad (55)$$

One may recognize (55) as the set of normal equations that describe a one-step linear predictor used in many signal-processing applications. The Berlekamp-Massey algorithm [2], [13], [14] iteratively forms the solution, but any standard technique to invert a matrix can be applied to the task. If (55) has more equations than unknowns its determinant will be zero and we can sequentially peel off one row and column and attempt to invert the reduced size matrix. Once the coefficients $\boldsymbol{\sigma}_i$ of the error location polynomial have been determined, the polynomial is tested to find its roots λ_i. The Chien algorithm [8], [15] is a method of exhaustively testing the polynomial to find these roots, which when inverted, identify the error locations. Then, adding a one to their identified locations easily repairs the errors.

Reed Solomon Codes

In 1960, Irving Reed and Gus Solomon described a new class of error correcting codes [16] that are now called R-S codes. The R-S codes are nonbinary cyclic codes in which the code symbols are binary m-tuples. They are used in many applications, from the compact disk to spacecraft traveling beyond the orbit of Pluto [17]. In 1964, Singleton showed that R-S codes have the best possible error-correction capability for any code of the same length and dimension [18]. Codes that achieve this "optimal" error correction capability are called maximum distance separable (MDS). As the name implies, their key property is that they provide maximum distances between code words [17], [18]. These codes are extensions of the BCH codes in much the same manner that the BCH codes are extensions of the Hamming codes. There exist t-error correcting R-S codes for the (n, k, t) set described as

$$(n, k, t) = (2^m - 1, 2^m - 1 - 2t, t) \quad (56)$$

where $2t$ is the degree of the generating polynomial as well as the number of parity symbols in the code [19], [20]. The coefficients of the generating polynomial $\mathbf{g}(X)$ are no longer binary but rather they are m-bit coefficients having the same meaning as the m-bit partition of the input data. The LFSR, which performs both encoding and decoding, stores m-bit words in its registers and performs products of the m-bit coefficients using arithmetic rules defined by an *extension field* referred to as GF(q), where the number of elements q in the field must be of the form p^m where p is a prime integer and m is positive integer [2], [21]. Elements from the extension field GF(2^m) are used in the construction of R-S codes. The rules for doing the arithmetic are not needed to understand the overall algorithms. Cyclic R-S codes with code words from GF(q) have length $m = q - 1$, as is seen in (56). Popular R-S code choices for many applications are chosen from the field GF(2^8) = GF(256), because each of the 256 field elements can be represented as an 8-b sequence, or byte [17].

Earlier we showed that simple block codes can be decoded by using a syndrome lookup table (or circuit). The syndrome was computed and its corresponding error pattern was subtracted from the received vector.

Unfortunately, this approach is out of the question for all but the most trivial R-S codes. For example, the (63, 53) five-error correcting R-S code has approximately 10^{20} syndromes. The construction of such a table is somewhat questionable. In [16], Reed and Solomon proposed a decoding algorithm based on solving sets of simultaneous equations. Though much more efficient than a lookup table, this algorithm is only useful for the smallest R-S codes. The decoding of longer R-S codes became viable in 1967 when Berlekamp demonstrated his efficient decoding algorithm for both R-S and nonbinary BCH codes [14], [22], [23].

Since the symbols in the R-S code are m-tuples, the error symbols introduced by the channel are also m-tuples. A typical two-symbol error polynomial may have the form shown as

$$\mathbf{e}(X) = e_1 X^{i(1)} + e_2 X^{i(2)} \tag{57}$$

where m-bit errors e_1 and e_2 have been added at positions $i(1)$ and $i(2)$. A channel error is detected at the decoder if the syndrome polynomial has nonzero coefficients. As with the BCH codes, the syndrome polynomial is evaluated at the roots of the generator polynomial to determine the received syndrome values S_i. These values are used in (55) to find the coefficients of the error location polynomial. The degree of the error location polynomial tells us the number of errors in the received code vector. The final task is to solve for the error values $\{e_i,\}$ by solving the following matrix (shown for the general case of ν errors)

$$\begin{bmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_\nu \\ \lambda_1^2 & \lambda_2^2 & \cdots & \lambda_\nu^2 \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_1^\nu & \lambda_2^\nu & \cdots & \lambda_\nu^\nu \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_\nu \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_\nu \end{bmatrix}. \tag{58}$$

Equation (58) expresses (57) evaluated at the roots $X = \lambda_i$ of the error-locating polynomial $\boldsymbol{\sigma}(X)$, where the indeterminate X is replaced with the known error locations. Once the error values are determined, the m-tuple error values are added to the received m-tuples at the previously determined error locations.

Channel Erasure Decisions

In one form of soft-decision making, the detector forms one of three decisions regarding the received processed waveform: binary 1, binary 0, or E, called an erasure. The 1 and 0 denote the sign of a processed signal if its magnitude is greater than some threshold (hard decision), and E denotes the condition that the signal magnitude is less than the threshold. The erasure reflects the information that the signal level is "too close to call" and might yield an unreliable decision.

A decoder can correct $d_{\min} - 1$ erasures (treated as unknown errors at known error locations). Normally half the information residing in the syndrome values is used to locate the error positions and half is used to correct the errors. Since erasure locations are known, only their values have to be determined by the decoding algorithm. Thus for the erasure correction task, the total syndrome information can be directed to repairing up to $d_{\min} - 1$ erasures (rather than about half that many $\lfloor d_{\min} - 1 \rfloor / 2$ for the error correction task). The decoding technique involves setting the erasured symbols to zero value and then proceeding with standard R-S decoding (omitting application of the error locating polynomial). The ability to recover from twice as many erasures as errors is one of the reasons that the compact disk player uses erasure detection and correction in its concatenated and interleaved R-S error-control procedures [24], [25].

Performance of R-S Codes in Bursty Noise

A key benefit of nonbinary codes, such as R-S codes, is their performance in the presence of bursty noise [26]. Even when a channel is characterized by random noise, R-S codes are often concatenated with convolutional codes to repair any convolutional decoding errors (which typically occur in bursts) [27], [28]. Consider as an example, the R-S (255, 247) code. In the context of (56), this code can be described using the (n, k) notation of $(2^8 - 1,\ 2^8 - 1 - 2t)$. Hence, $n - k = 2t$, or the code's error-correcting capability is $t = (n - k)/2$. In terms of its 8-b symbols (or bytes), $t = (255 - 247)/2 = 4$ B within each code word sequence of 255 code bytes. Imagine a burst of contiguous noise events, having a duration of 25 b. How many code bytes within the code word would be affected? No matter where these contiguous noise events might appear within the 255 code bytes, they must affect exactly 4 B. However, the R-S (255, 247) code is capable of correcting 4 B. The decoding is based on replacing an incorrect byte regardless how many bits within the byte have been corrupted. An incorrect byte is replaced with a correct one, whether the corruption was caused by a 1-b-duration noise event or an 8-b-duration noise event. Now compare this to the case where the noise does not appear as a contiguous burst, but as 25 random events. How many of the 255 R-S code bytes might be "hit" in such case? Twenty-five bytes might be hit, and the code would then be overwhelmed. For R-S codes, one can say that the codes "prefer" to see the noise occur in bursts. For nonbinary codes such as R-S, there is an kind of "economy of scale" at work whenever the degrading events occur in bursts rather than as random events.

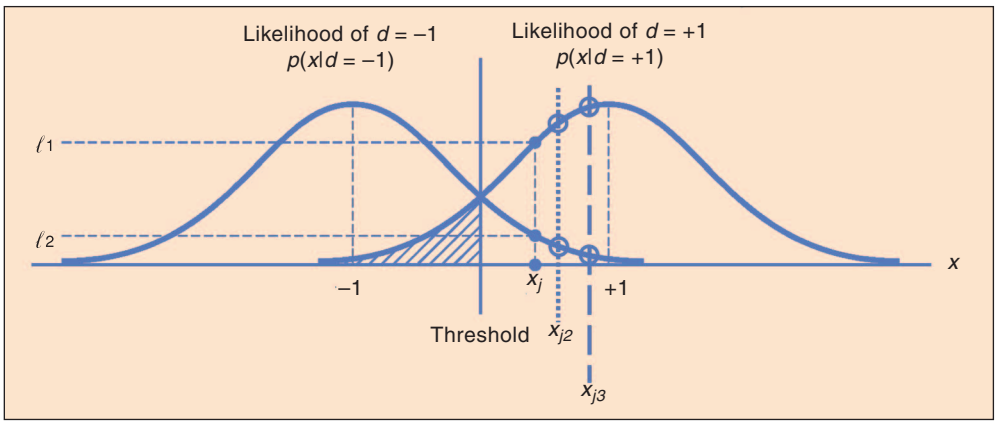
Approaching the Shannon Limit

Shannon [29], [30] developed many of the fundamental mathematical relationships for communicating information in the presence of noise. He showed that every channel has a channel capacity C, and for transmission rates less than C, there exists codes capable of achieving

arbitrarily small decoded-error probabilities. An outgrowth of his capacity theorem shows that with optimum coding, it is possible to approach an error probability of zero with an E_b/N_0 as low as -1.6 dB. This limiting value is known as the Shannon limit; it is often shown as a discontinuous curve (vertical line) on the bit-error probability P_B versus E_b/N_0 plot, extending from $P_B = 1/2$ to $P_B = 0$. Shannon's work proves the existence of codes that can approach this limiting performance but does not show how to construct them. It does show that to achieve very low error probabilities, long block lengths are needed. In effect, Shannon predicted that coding techniques would be found for achieving coding gains of about 11–12 dB. Ever since 1948, research in error-correction coding has focused on constructing good codes and finding easy-to-implement encoding/decoding methods toward fulfilling Shannon's prediction. The Shannon limit represents an interesting theoretical bound, but it is not a practical goal because it was derived for a rate 0 code. Several authors use an E_b/N_0 of 0.2 dB as a *pragmatic* Shannon limit for a rate $1/2$ code over a binary-input AWGN channel [31].

For about 30 years after Shannon's fundamental work, coding research produced a rich assortment of practical techniques (some of which are listed in the references), resulting in an achievable coding gain of about 6–7 dB at $P_B = 10^{-5}$. During the 1980s, there weren't any major developments to allow for larger gains. In 1993 Berrou et al. reported on a concatenated rate $1/2$ coding scheme that used iterative decoding to achieve a $P_B = 10^{-5}$ at an E_b/N_0 within about 0.5 dB away from the "pragmatic" Shannon limit [32]. That development, known as turbo coding, enabled the implementation of real systems with coding gains of about 8–10 dB.

The next revolution, known as LDPC codes, was actually a resurgence of a scheme that had been proposed by Gallager in his 1961 doctoral dissertation and in [33] and [34]. These codes were largely forgotten for several decades due to the lack of computational capability in those years. LDPC codes, which also use iterative decoding, were rediscovered by McKay and Neal [35] in 1996. Today the field of LDPC coding appears to represent the main competition to turbo coding. An LDPC rate $1/2$ code using a block length of 10^7 holds the distinction of achieving a BER of 10^{-6} within 0.04 dB of the "pragmatic" Shannon limit, as reported by Chung et al. [36] in 2001.



▲ *9. Likelihood functions.*

Likelihood Functions and the ML Criterion

Linear block codes, including cyclic codes and R-S codes, lend themselves to hard-decision decoding giving rise to the type of syndrome calculation and decision making described earlier. However, for iterative decoding (as exemplified by both turbo and LDPC codes), the improvements offered by soft-decision decoding (about 2 dB for eight-level quantization, in AWGN) are an essential part of the operation.

Figure 9 shows the conditional pdfs, referred to as likelihood functions, that depict the binary decision-making process associated with pulses $(+1, -1)$ disturbed by AWGN. Since AWGN is the most commonly used noise model in communication systems, we use it here and therefore show the likelihood functions in Figure 9 drawn with the familiar Gaussian shapes. The rightmost function $p(x|d = +1)$, called the likelihood of $d = +1$, shows the pdf of the random variable x (noisy signal sample) conditioned on the data $d = +1$ being sent. The leftmost function $p(x|d = -1)$, called the likelihood of $d = -1$, illustrates a similar pdf conditioned on $d = -1$ being sent. In Figure 9, one such arbitrary received sample value x_j is shown. A line subtended from x_j intercepts the two likelihood functions yielding the likelihood values $\ell_1 = p(x_j|d = +1)$ and $\ell_2 = p(x_j|d = -1)$. Stemming from Bayes' theorem [1], one can make a data decision by forming a ratio of likelihoods and deciding that $d = +1$ if

$$\frac{\ell_1}{\ell_2} = \frac{p(x_j \mid d = +1)}{p(x_j \mid d = -1)} > \frac{P(d = -1)}{P(d = +1)}$$

$$\text{or} \quad \frac{p(x_j \mid d = +1)P(d = +1)}{p(x_j \mid d = -1)P(d = -1)} > 1. \qquad (59)$$

Equation (59), known as the maximum a posteriori (MAP) criterion, corresponds to making a decision based on comparing ℓ_1/ℓ_2 to the threshold $P(d = -1)/P(d = +1)$, a ratio of a priori probabilities. That is, decide $d = +1$ if the likelihood ratio ℓ_1/ℓ_2 is greater than the threshold, and otherwise decide $d = -1$. When knowledge about the a

priori probabilities is not available, decisions are usually made by assuming equally likely a priori probabilities. When this is done, the MAP criterion in (59) is then known as the ML criterion.

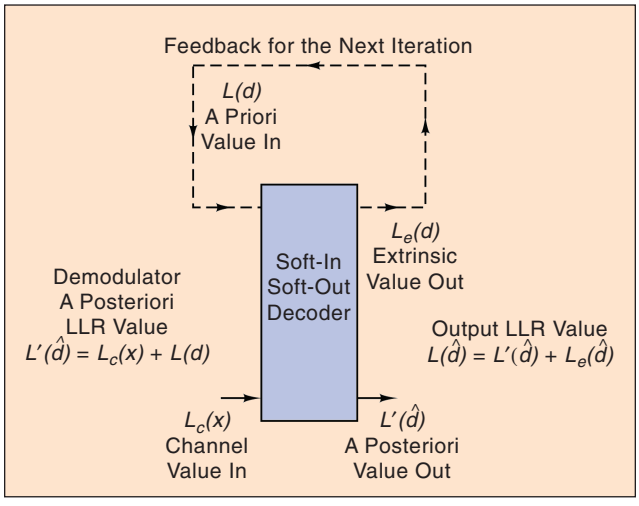
Turbo Codes

Initially, turbo codes were implemented with two or more convolutional encoders configured as concatenated component codes; but block encoders are equally applicable. By taking the logarithm of a likelihood ratio, we obtain a useful metric called the log-likelihood ratio (LLR). In the context of (59), we designate $L'(\hat{d})$ to represent the sum of two LLRs (one for the detector's soft-decision output and one for the ratio of a priori probabilities of the data). We show this as

$$L'(\hat{d}) = \log\left[\frac{P(x \mid d=+1)}{P(x \mid d=-1)}\right] + \log\left[\frac{P(d=+1)}{P(d=-1)}\right] \tag{60a}$$

$$= L(x \mid d) + L(d) = L_c(x) + L(d) \tag{60b}$$

where $L(x|d)$, designated $L_c(x)$, is the LLR of the random variable x representing the detector's measurement of the received signal x under the alternate conditions that $d=+1$ or $d=-1$ may have been sent, and $L(d)$ is the a priori LLR of the data bit d. The notation $L_c(x)$ emphasizes that this LLR term is the result of a channel measurement made at a receiver. Note that for equally likely a priori probabilities, the ratio of prior probabilities is equal to one; hence, its logarithm, seen as the last term in (60), is equal to zero. This corresponds to a threshold that is located at the crossover of the two likelihoods as seen in Figure 9. So far we have not considered the presence of any coding. The introduction of a code will typically yield decision-making benefits. For a systematic code, it can be shown [32] that the LLR (soft output) $L(\hat{d})$ out of the decoder can be expressed as



▲ *10. Soft-input/soft-output decoder.*

$$L(\hat{d}) = L'(\hat{d}) + L_e(\hat{d}) = L_c(x) + L(d) + L_e(\hat{d}) \tag{61}$$

where $L'(\hat{d})$ stems from (60) and $L_e(\hat{d})$, called the extrinsic LLR, represents extra knowledge that is gleaned from the decoding process. Thus, (61) expresses the decoder output in terms of three LLR elements—a channel measurement, a priori knowledge of the data, and an extrinsic LLR stemming solely from the decoder. Each of these individual LLRs can be added (for a systematic code) because these three terms are statistically independent [1], [33], [37]. This decoder output $L(\hat{d})$ is a real number that provides a hard decision plus a confidence metric regarding that decision. The sign of $L(\hat{d})$ denotes the hard decision; that is, for positive values of $L(\hat{d})$, we decide that $d=+1$, and for negative values decide that $d=-1$. The magnitude of $L(\hat{d})$ denotes the confidence of that decision. Often, the value of $L_e(\hat{d})$ has the same sign as $L_c(x)+L(d)$, and therefore acts to improve the reliability of $L(\hat{d})$.

Decoding of Turbo Codes

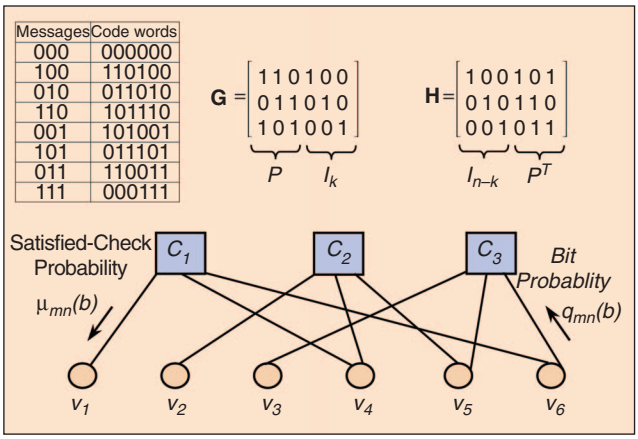
Turbo decoding involves feeding outputs from one decoder to the inputs of other decoders in an iterative fashion. The phenomenal performance of these techniques requires a novel type of decoder—one that accepts soft decisions at its input and delivers soft decisions at its output. For the first decoding iteration, illustrated in Figure 10, one generally starts with the assumption that the binary data is equally likely, yielding an initial a priori LLR value of $L(d)=0$ for the rightmost term in (60). The channel LLR value, $L_c(x)$, is measured by forming the logarithm of the ratio of the values of ℓ_1 and ℓ_2 for a particular observation of x (see Figure 9). As illustrated in Figure 10, where the decoder block represents two or more decoders, the extrinsic likelihood $L_e(\hat{d})$ out of the decoder is fed back to the decoder input, to serve as a refinement of the a priori probability of the data for the next iteration. Additional iterations yield values of $L(\hat{d})$ that asymptotically manifest further refinements and hence better error performance. A fundamental principle for feeding back information to another decoder is that a decoder should never be supplied with information that stems from its own input (because the input and output corruption will be highly correlated). Such a soft-input, soft-output decoder is sometimes referred to as a "SNR amplifier." Real systems are typically implemented with a fixed number of about four to eight such decoding iterations. Figure 9 offers some insight into the iterative process. As a result of a single pass through the decoder, the sample x_j yields a soft number, the LLR $L(\hat{d})$. This is illustrated by a vertical line passing through x_j and the likelihood intercepts ℓ_1 and ℓ_2. Imagine a second and third decoding iteration of the same sample, now labeled x_{j2} and x_{j3}, respectively, that may result in a more confident $L(\hat{d})$ output, as shown in Figure 9 by the rightward movement of the vertical line. Such

movement can come about (with increased iterations) because the initial a priori probabilities become "refined" during each round, resulting in an apparent movement of the sample point yielding a more confident $L(\hat{d})$. We can interpret this to mean that the turbo decoder iterations give us an apparent SNR enhancement. An extensive treatment of turbo encoding and decoding can be found in [32] and [38]–[51].

LDPC Codes

An LDPC code is traditionally defined by its $M \times N$ parity-check matrix $\mathbf{H}$. The M rows in $\mathbf{H}$ specify the M constraints in the code. For example, if one constraint dictates that bit numbers 1, 4, and 6 must sum to zero for each code word, then there must be one row in $\mathbf{H}$ that contains a one in positions 1, 4, and 6, and zeros elsewhere. The N columns in $\mathbf{H}$ correspond to the total number of code bits within a code word. A *regular* LDPC code has a sparse $\mathbf{H}$ matrix having exactly w_c ones per column and exactly $w_r = w_c(N/M)$ ones per row, where w_c and w_r are both small compared to N. Thus, each parity-check equation involves exactly w_r bits, and every bit in a code word is involved in exactly w_c parity check equations. The term "low density" indicates that these codes are characterized by sparse $\mathbf{H}$ matrices. The fraction of ones in a regular LDPC matrix is w_r/N, which can be quite small for long codes. In contrast, the average fraction of ones in a random binary matrix (with independent components equally likely to be one or zero) is $1/2$. If $\mathbf{H}$ is sparse, but the number of ones per column or row is not constant, the code is called an irregular LDPC code. Such irregular codes often outperform regular codes with similar dimensions [52], [53].

Any parity-check code (including an LDPC code) may be specified by a Tanner graph [54], as shown in Figure 11. The (6, 3) code example (described earlier) does not have a sparse $\mathbf{H}$ matrix; nevertheless, we use it here as a simple way to illustrate this graph. For LDPC codes, we shall use the index m to indicate a particular parity check location (row in $\mathbf{H}$), and index n to indicate a particular bit position (column in $\mathbf{H}$). The Tanner graph contains two kinds of nodes, check nodes (designated with squares) and bit nodes (designated with circles). There are M check nodes, one for each parity check $C_1, \ldots, C_m, \ldots, C_M$, and N bit nodes, one for each code bit $v_1, \ldots, v_n, \ldots, v_N$. The check nodes are connected to the bit nodes that they check. Specifically, a branch (also called an edge) connects check-node m to bit-node n if and only if the mth parity check involves the nth bit (i.e., only if $\mathbf{H}_{m,n} = 1$). Thus, the graph is analogous to the $\mathbf{H}$ matrix, since the entries in $\mathbf{H}$ are 1 if and only if the mth check node is connected to the nth bit. The branches between nodes can be thought of as information-flow pathways for the iterative computation of probabilistic quantities. During the decoding process, likelihoods obtained from soft-decision components of a received vector $\mathbf{r}$ initialize the bit nodes. Each bit node



▲ *11. Representing the (6, 3) code with a Tanner Graph.*

determines the probability of a bit having value one (also zero), given the parity check information from all the check nodes involving that bit. Each check node determines the probability that its parity check is satisfied, given that a particular bit is set to one (also zero) and the other participating bits have separable distributions.

In Figure 11, we have repeated properties of the (6, 3) code described earlier—code word assignments, matrix $\mathbf{G}$, and matrix $\mathbf{H}$. Only bits and checks that are related by having a one in the corresponding location of $\mathbf{H}_{m,n}$ need to be considered. For the (6, 3) code, the rows of $\mathbf{H}$, describing the parity checks that must be met by each proper code word, can be written as

$$\begin{aligned} v_1 \oplus v_4 \oplus v_6 &= 0 \\ v_2 \oplus v_4 \oplus v_5 &= 0 \\ v_3 \oplus v_5 \oplus v_6 &= 0. \end{aligned} \tag{62}$$

For the (6, 3) code, the set of relationships in (62) corroborates what was described in (26). That is, each relationship yields a syndrome, and the code constraints are met when each syndrome component has a zero value.

Decoding of LDPC Codes

For decoding an LDPC code, we want to find the probability that each bit v_n of a received vector $\mathbf{r}$ equals one (also zero), knowing that the estimated code word $\hat{\mathbf{U}}$ stemming from $\mathbf{r}$ must satisfy the constraint $\hat{\mathbf{U}}\mathbf{H}^T = 0$. Given a received vector $\mathbf{r}$, solving directly for the probability $P(v_n = b|\mathbf{r})$, that the nth bit equals either one or zero is very complex. Gallager [34] provided an iterative technique, known as the sum-product algorithm, where the probability $\mu_{mn}(b)$ that the mth check is satisfied by a received vector $\mathbf{r}$ (given $v_n = b$), is passed from check node C_m to bit node v_n. This satisfied-check probability $\mu_{mn}(b)$ is gleaned from all bits participating in the mth check (other than v_n). Likewise, the bit probability $q_{mn}(b)$ that the nth bit has value $v_n = b$, is passed from bit node v_n to check node C_m. This bit probability $q_{mn}(b)$ is gleaned from all the checks that the nth bit participates in (other than C_m).

The message-passing procedure is indicated on the Tanner graph of Figure 11 in the context of the (6, 3) code described earlier. Messages containing satisfied-check probabilities $\mu_{mn}(b)$ are shown moving from check nodes to bit nodes, and messages containing bit probabilities $q_{mn}(b)$ are shown moving from bit nodes to check nodes. The process is repeated until it converges to a code word solution or until a time limit is reached. With the aid of Figure 11, consider the following example of message-passing iterations, focusing particularly on bit-node v_4 and check-nodes C_1 and C_2. At the start, bit nodes are initialized with likelihood values stemming from a detector. Suppose that bit node v_4 passes the probability $q_{24}(b)$ that $v_4 = 1$ (also 0) to check-node C_2. (It passes similar information to check-node C_1.) Check-node C_2 collects the incoming probabilities from all other bits involved in check 2, namely v_2 and v_5, computes a probability $\mu_{24}(b)$ that parity-check C_2 is satisfied given that $v_4 = 1$ (also 0), and passes this message to bit-node v_4. Check-node C_2 passes similar information to v_2, given $v_2 = 1$ (also 0), and to v_5, given $v_5 = 1$ (also 0). When bit-node v_4 receives such satisfied-check information from all the check nodes involving v_4, namely C_1 and C_2, it recomputes the probability that $v_4 = 1$ (also 0) gleaned from the connected check nodes (apart from check-node C_2) and passes this message back to node C_2, and similar information to check-node C_1, and so forth. Note that when sending $\mu_{mn}(b)$ probabilities from C_m to v_n, this algorithm excludes information passed from v_n to C_m in the prior round. Similarly when sending $q_{mn}(b)$ probabilities from v_n to C_m, the algorithm excludes information passed from C_m to v_n in the prior round. These rules resemble those used in the iterative turbo-decoding process, where information is never fed back to a decoder that stems from itself.

After decoder initialization, we can think of the process of updating the probabilities between nodes as providing extrinsic information that yields improvements in coding performance. Each iteration typically provides more confident likelihood values of the code bits. The algorithm is typically implemented in the logarithmic domain using log-likelihood ratios similar to the turbo-decoding process. The iterative decoding techniques used in both turbo and LDPC codes are the tools that have brought about the astounding error performance that was promised by Shannon in 1948. Extensive treatments of LDPC coding can be found in [33], [34], [52]–[68]. An important coding task, yet to be accomplished, is to determine for implementing real systems, which codes and decoding techniques can yield optimum performance with minimum complexity.

Conclusion

Basic principles of block codes were presented with illustrations to visualize the concepts of vector spaces and subspaces. We offered intuitive explanations of goals, capabilities, and limitations of codes. We examined an important subclass of block codes called cyclic codes, described their algebraic structure, and looked at the very popular BCH and R-S cyclic codes. Also, we looked at the newest techniques, turbo codes and LDPC codes, that use iterative decoding to obtain performance exceedingly close to theoretical limitations.

Bernard Sklar has 50 years of electrical engineering experience at companies that include Hughes Aircraft, Litton Industries, and The Aerospace Corp. He is head of advanced systems at Communications Engineering Services. He received the 1984 Prize Paper Award from the IEEE Communications Society and is the author of *Digital Communications: Fundamentals and Applications*, 2nd edition (Prentice-Hall, 2001).

Fredric J. Harris is a professor of electrical and computer engineering at San Diego State University where he holds the CUBIC Signal Processing Chair. He holds several patents and is the author of *Multirate Signal Processing for Communication Systems* (Prentice Hall 2004). He was chair of the Asilomar Conference on Signals, Systems, and Computers, and the Software Defined Radio Conference.

References

[1] B. Sklar, *Digital Communications: Fundamentals and Applications*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall Inc., 2001.

[2] S. Lin and D.J. Costello, Jr., *Error Control Coding: Fundamentals and Applications.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

[3] G.C. Clark and J.B. Cain, *Error Correction Coding for Digital Communications.* New York: Plenum, 1988.

[4] J. Wolf, A. Michelson, and A. Levesque, "On the probability of undetected error for linear block codes," *IEEE Trans. Commun.*, vol. COM-30, pp. 317–325, Feb. 1982.

[5] T. Kasami, T. Klove, and S. Lin, "Linear block codes for error detection," *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 131–136, Jan. 1983.

[6] D. Hertz and Y. Azenkot, "Memory/speed tradeoffs for look-up table decoding of systematic linear block codes," *IEEE Trans. Commun.*, vol. 38, pp. 109–111, Jan. 1990.

[7] R.G. Gallager, *Information Theory and Reliable Communication.* New York: Wiley, 1968.

[8] W.W. Peterson and E.J. Weldon, *Error Correcting Codes.* Cambridge, MA: MIT Press, 1972.

[9] R.E. Blahut, *Fast Algorithms for Digital Signal Processing.* Reading, MA: Addison-Wesley, 1985.

[10] R.C. Bose and D.K. Ray-Chadhuri, "On a class of error correcting binary group codes," *Inform. Contr.*, vol. 3, pp. 68–79, Mar. 1960.

[11] R.C. Bose and D.K. Ray-Chadhuri, "Further results on error correcting binary group codes," *Inform. Contr.*, vol. 3, pp. 279–290, Sept. 1960.

[12] S.A. Vanstone and P.C. van Oorschot, *An Introduction to Error Correcting Codes with Applications.* Boston, MA: Kluwer, 1989.

[13] R.E. Blahut, *Theory and Practice of Error Control Codes.* Reading, MA: Addison-Wesley, 1983.

[14] R. Berlekamp, "On decoding binary Bose-Chaudhuri-Hocquenghem codes," *IEEE. Trans. Inform. Theory*, vol. IT-11, pp. 580–585, Oct. 1965.

[15] R.T. Chien, "Cyclic decoding procedure for the Bose-Chadhuri-Hocquenghem codes," *IEEE Trans. Inform. Theory*, vol. IT-10, pp. 357–363, Oct. 1964.

[16] I.S. Reed and G. Solomon, "Polynomial codes over certain finite fields,"

SIAM J. Appli. Math. vol. 8, pp. 300–304, 1960.

[17] S.B. Wicker and V.K. Bhargava, Ed., *Reed-Solomon Codes and Their Applications.* New York: IEEE Press, 1994.

[18] R.C. Singleton, "Maximum distance Q-nary codes," *IEEE Trans. Inform. Theory*, vol. IT-10, no. 2, pp. 116–118, Apr. 1964.

[19] S.B. Wicker, *Error Control Systems for Digital Communication and Storage.* Englewood Cliffs, NJ: Prentice Hall, 1994.

[20] B. Cipra, "The ubiquitous Reed-Solomon codes," *SIAM News*, vol. 26, no.1, Jan. 1993.

[21] R.J. McEliece, *Finite Fields for Computer Scientists and Engineers.* Boston, MA: Kluwer Academic, 1987.

[22] E.R. Berlekamp, "Nonbinary BCH decoding," presented at the 1967 Int. Symp. Information Theory, San Remo, Italy.

[23] E.R. Berlekamp, *Algebraic Coding Theory.* New York: McGraw Hill, 1968 (rev. ed., Aegean Park Press: Laguna Hills, CA, 1984.)

[24] K.C. Pohlmann, *The Compact Disk Handbook*, 2nd ed. Madison, WI: A-R Editions, 1992.

[25] M.J. Riley and I.E.G. Richardson, *Digital Video Communications.* Boston, MA: Artech House, 1997.

[26] G.D. Forney, "Burst-correcting codes for the classic bursty channel," *IEEE Trans. Commun.*, vol. COM-19, pp. 772–781, Oct. 1971.

[27] G.D. Forney, *Concatenated Codes.* Cambridge, MA: MIT Press, 1967.

[28] J.L Massey, "Deep space communications and coding: A match made in heaven," in *Advanced Methods for Satellite and Deep Space Communications*, J. Hagenauer, Ed. (Lecture Notes in Control and Information Sciences, vol. 182). Berlin: Springer-Verlag, 1992.

[29] C.E. Shannon, "A mathematical theory of communication," *BSTJ*, vol. 27, pp. 379–423, 623–657, 1948.

[30] C.E. Shannon, "Communication in the presence of noise," *Proc. IRE*, vol. 37, no. 1, pp. 10–21, Jan. 1949.

[31] S.A. Butman and R.J. McEliece, "The ultimate limits of binary coding for a wideband gaussian channel," JPL Deep Space Network Progress Rep. 42–22, May–June 1974, pp. 78–80, Aug. 15, 1974.

[32] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," in *IEEE Proc. Int. Conf. Communications*, Geneva, Switzerland, May 1993, pp. 1064–1070.

[33] R.G. Gallager, "Low-density parity-check codes," *IRE Trans. Inform. Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

[34] R.G. Gallager, *Low Density Parity-Check Codes.* Cambridge, MA: MIT Press, 1963.

[35] D.J.C. Mackay and R.M. Neal "Near Shannon-limit performance of low-density parity-check codes," *Electron. Lett.*, vol. 32, pp. 1645–1646, Aug. 1996.

[36] S.-Y. Chung, G.D. Forney, Jr., T.J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Commun. Letters*, vol. 5, pp. 58–60, Feb. 2001.

[37] J. Hagenauer, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. Inform. Theory*, vol. 42, no. 2, pp. 429–445, Mar. 1996.

[38] D. Divsalar and F. Pollara, "On the design of turbo codes," Jet Propulsion Laboratory, Pasadena, CA, TDA Progress Rep. 42–123, pp. 99–121, Nov. 15, 1995.

[39] D. Divsalar and R.J. McEliece, "Effective free distance of turbo codes," *Electron. Lett.,* vol. 32, no. 5, pp. 445–446, Feb. 29, 1996.

[40] S. Dolinar and D. Divsalar, "Weight distributions for turbo codes using random and nonrandom permutations," Jet Propulsion Laboratory, Pasadena, CA, TDA Progress Rep. 42–122, pp. 56–65. Aug. 15, 1995.

[41] D. Divsalar and F. Pollara, "Turbo codes for deep-space sommunications," Jet Propulsion Lab, Pasadena, CA, TDA Progress Rep. 42–120, pp. 29–39, Feb. 15, 1995.

[42] D. Divsalar and F. Pollara, "Multiple turbo codes for deep-space communications," JPL, Pasadena, CA, TDA Progress Rep. 42–121, pp. 66–77, May 15, 1995.

[43] D. Divsalar and F. Pollara, "Turbo codes for PCS applications," in *Proc. ICC '95*, Seattle, Washington, June 18–22, 1995, vol. 1, pp. 54–59.

[44] L.R. Bahl, J. Cocke, F. Jeinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 248–287, Mar. 1974.

[45] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Soft output decoding algorithm in iterative decoding of turbo codes," Jet Propulsion Lab, Pasadena, CA, TDA Progress Rep. 42-124, pp. 63–87, Feb. 15, 1996.

[46] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "A soft-input soft-output maximum a posteriori (MAP) module to decode parallel and serial concatenated codes," Jet Propulsion Lab, Pasadena, CA, TDA Progress Rep. 42-127, pp. 63–87, Nov. 15, 1996.

[47] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "A soft-input soft-output APP module for iterative decoding of concatenated codes," *IEEE Commun. Lett.*, vol. 1, no. 1, pp. 22–24, Jan. 1997.

[48] S. Pietrobon, "Implementation and performance of a turbo/MAP decoder," *Int. J. Satellite Commun.*, vol. 16, pp. 23–46, Jan.–Feb. 1998.

[49] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," in *Proc. ICC'95*, Seattle, WA, June 1995, pp. 1009–1013.

[50] F. Burkert and J. Hagenauer, "A serial concatenated coding scheme with iterative turbo and feedback decoding," in *Proc. Int. Symp. Turbo Codes*, Sept. 1997, pp. 227–230.

[51] R. Pyndiah, "Near optimum decoding of product codes: Block turbo codes," *IEEE Trans. Commun.*, vol. 46, no. 8, pp. 1003–1010, Aug. 1998.

[52] T.J. Richardson, M.A. Shokrollahi, and R.L. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Inform. Theory*, vol. 47, pp. 619–637, Feb. 2001.

[53] Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inform. Theory*, vol. IT-27, pp. 533–547, Sept. 1981.

[54] X.-Y. Hu, E. Eleftheriou, D.-M. Arnold, and A. Dholakia, "Efficient implementations of the sum product algorithm for decoding LDPC codes," in *Proc. Globecom*, 2001, vol. 2, pp. 25–29.

[55] D. MacKay, "Good error correcting codes based on very sparse matrices," *IEEE Trans. Inform. Theory*, vol. 45, pp. 399–431, Mar. 1999.

[56] M.P.C. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity-check codes based on belief propagation," *IEEE Trans. Commun.*, vol. 47, pp. 673–680, May 1999.

[57] T.J. Richardson and R.L. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inform. Theory*, vol. 47, pp. 638–656, Feb. 2001.

[58] T.J. Richardson and R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 599–618, Feb. 2001.

[59] D.J.C. MacKay, S.T. Wilson, and M.C. Davey, "Comparison of constructions of irregular Gallager codes," *IEEE Trans. Commun.*, vol. 47, pp. 1449–1454, Oct. 1999.

[60] M.G. Luby, M. Mitzenmacher, M.A Shokrollahi, and D.A. Spielman, "Improved low-density parity check codes using irregular graphs," *IEEE Trans. Inform. Theory*, vol. 47, pp. 585–598, Feb. 2001.

[61] J. Fan, E. Kurtas, A. Friedman, and S. McLaughlin, "Low-density parity-check codes for magnetic recording," in *Proc. 1999 Allerton Conf,* pp. 1314–1323.

[62] R. Lucas, M. Fossorier, et al., "Iterative decoding of one-step majority-logic decodable codes based on belief propagation," *IEEE Trans. Commun.*, vol. 48, pp. 931–937, June 2000.

[63] J.R. Barry, "Low-density parity-check codes," course notes, Georgia Institute of Technology, Oct. 5, 2001 [Online]. Available: http://users.ece.gatech.edu/~barry/6606/handouts/ldpc.pdf

[64] B. Levine, R. Taylor, and H. Schmit, "Implementation of near Shannon limit error-correcting codes using reconfigurable hardware," in *Proc. 2000 IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, Apr., 2000, pp. 217–226.

[65] A. Shokrollahi, "LDPC codes: An introduction," Digital Fountain, Inc., Fremont, CA, Tech. Rep., Apr. 2, 2003.

[66] F.R. Kschischang, "Codes defined on graphs," *IEEE Commun. Mag.,* vol. 41, no. 8, pp. 118–125, Aug. 2003.

[67] T. Richardson and R. Urbanke, "The renaissance of Gallager's low-density parity-check codes," *IEEE Commun. Mag.*, vol. 41, no. 8, pp. 126–131, Aug. 2003.

[68] E. Yeo and V. Anantharam, "Iterative decoder architectures," *IEEE Commun. Mag.*, vol. 41, no. 8, pp. 132–140, Aug. 2003.