The Standard Array

A Useful Tool for Understanding and Analyzing Linear Block Codes

by

Bernard Sklar

Introduction

The standard array can be thought of as an organizational tool, a filing cabinet that contains all of the possible 2^n binary n-tuples (called *vectors*)—nothing missing, and nothing replicated. The entire space of n-tuples is called the *vector space*, V_n. 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. However, even for large codes, the standard array allows visualization of important performance issues, such as bounds on error-correction capability, as well as possible tradeoffs between error correction and detection.

The Standard Array

For an (n, k) linear block code, all possible 2^n received vectors are arranged in an array, called the *standard array*, such that the first row contains the set of all the 2^k codewords, $\{\mathbf{U}\}$, starting with the all-zeros codeword (the all-zeros sequence must be a member of the codeword set [1]). The term *codeword* is exclusively used to indicate a valid codeword entry in the first row of the array. The term *vector* is used to indicate any ordered sequence (for example, any n-tuple in V_n). The first column of the standard array contains all the correctable error patterns.

The term *error pattern* refers to a binary n-tuple, $\mathbf{e}$, that when added to a transmitted codeword, $\mathbf{U}$, results in the reception of an n-tuple or vector, $\mathbf{r} = \mathbf{U} + \mathbf{e}$, which can be called a *corrupted codeword*. In the standard array, each row, called a *coset*, consists of a correctable error pattern in the leftmost position, called a *coset leader*, followed by corrupted codewords (corrupted by that error pattern). The structure of the standard array for an (n, k) code is shown below:

$$
\begin{array}{cccccc}
\mathbf{U}_1 & \mathbf{U}_2 & \cdots & \mathbf{U}_i & \cdots & \mathbf{U}_{2^k} \\
\mathbf{e}_2 & \mathbf{U}_2 + \mathbf{e}_2 & \cdots & \mathbf{U}_i + \mathbf{e}_2 & \cdots & \mathbf{U}_{2^k} + \mathbf{e}_2 \\
\mathbf{e}_3 & \mathbf{U}_2 + \mathbf{e}_3 & \cdots & \mathbf{U}_i + \mathbf{e}_3 & \cdots & \mathbf{U}_{2^k} + \mathbf{e}_3 \\
 & \vdots & \vdots & \vdots & & \\
\mathbf{e}_j & \mathbf{U}_2 + \mathbf{e}_j & \cdots & \mathbf{U}_i + \mathbf{e}_j & \cdots & \mathbf{U}_{2^k} + \mathbf{e}_j \\
 & \vdots & \vdots & \vdots & & \\
\mathbf{e}_{2^{n-k}} & \mathbf{U}_2 + \mathbf{e}_{2^{n-k}} & \cdots & \mathbf{U}_i + \mathbf{e}_{2^{n-k}} & \cdots & \mathbf{U}_{2^k} + \mathbf{e}_{2^{n-k}}
\end{array} \tag{1}
$$

Codeword $\mathbf{U}_1$, the all-zeros codeword, plays two roles. It is one of the codewords. Also, $\mathbf{U}_1$ can be thought of as the error pattern $\mathbf{e}_1$—the pattern that represents no error, such that $\mathbf{r} = \mathbf{U}$. The array contains all 2^n n-tuples in the space (each n-tuple appears only once). Each coset or row contains 2^k n-tuples. Therefore, there are $2^n/2^k = 2^{n-k}$ cosets (or rows).

The decoding algorithm calls for replacing a corrupted codeword, $\mathbf{U} + \mathbf{e}$, with the valid codeword $\mathbf{U}$, which is located at the top of the column where $\mathbf{U} + \mathbf{e}$ is located. Suppose that a codeword $\mathbf{U}_i$ is transmitted over a noisy channel. If the error pattern caused by the channel is a coset leader, the received vector will be decoded correctly into the transmitted codeword $\mathbf{U}_i$. If the error pattern is not a coset leader, an erroneous decoding will result. There are several bounds on the error-correcting capability of linear codes; any workable code system must meet all of these bounds. One such bound, called the *Hamming bound* [2], is described below.

Number of parity bits:

$$
n - k \geq \log_2 \left[1 + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{t} \right] \tag{2}
$$

or

Number of cosets:

$$
2^{n-k} \geq \left[1 + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{t} \right] \tag{3}
$$

where the binomial factor $\binom{n}{j}$ represents the number of ways in which j bits out of n may be in error. Note that the sum of the terms within the square brackets yields the minimum number of rows needed in the standard array to correct all combinations of errors through t-bit errors. The inequality gives a lower bound on $n - k$, the number of parity bits (or the number of 2^{n-k} cosets) as a function of the t-bit error-correction capability of the code. Similarly, the inequality can be described as giving an upper bound on the t-bit error-correction capability as a function of the number of $n - k$ parity bits (or 2^{n-k} cosets). For any (n, k) linear block code to provide a t-bit error-correcting capability, it is a necessary condition that the Hamming bound be met.

To demonstrate how the standard array provides a visualization of this bound, let's use the (127, 106) BCH code as an example. The array contains all $2^n = 2^{127} \approx 1.7 \times 10^{38}$ n-tuples in the space. The topmost row of the array contains the $2^k = 2^{106} \approx 8.1 \times 10^{31}$ codewords; hence, this is the number of columns in the array. The leftmost column contains the $2^{n-k} = 2^{21} = 2{,}097{,}152$ coset leaders (or correctable error patterns); hence, this is the number of rows in the array. Although the number of n-tuples and codewords is enormous, the concern is not with any individual entry; the primary interest is in the number of cosets. There are 2,097,152 cosets, and hence there are at most 2,097,151 error patterns that can be corrected by this code. Next, it is shown how this number of cosets dictates an upper bound on the t-bit error-correcting capability of the code.

Since each codeword contains 127 bits, there are 127 ways to make single errors. We next compute how many ways there are to make double errors, namely $\binom{127}{2} = 8{,}001$. We move on to triple errors because thus far only a small portion of the total 2,097,151 correctable error-patterns have been used. There are $\binom{127}{3} = 333{,}375$ ways to make triple errors. Table 1 lists these computations, indicating that the all-zeros error pattern requires the presence of the first coset. Also shown for single through quadruple error types are the number of cosets required for each error type and the cumulative number of cosets necessary through that error type. This table shows that a (127, 106) code can correct all single, double, and triple error patterns—and the unused rows are indicative of the fact that more error correction is possible. It might be tempting to try fitting all possible 4-bit error patterns into the array. However, Table 1 shows that that this is not possible, because the number of remaining cosets in the array is much smaller than the cumulative number of cosets required, as indicated by the last line of the table. Therefore, for this (127, 106) example, the code has a Hamming bound that guarantees the correction of up to and including all 3-bit errors.

Table 1

Error-Correction Bound for the (127, 106) Code

Number of Bit Errors	Number of Cosets Required	Cumulative Number of Cosets Required
0	1	1
1	127	128
2	8,001	8,129
3	333,375	341,504
4	10,334,625	10,676,129

Perfect Codes

The previously considered (127, 106) code with demonstrated single, double and triple error-correcting capability exemplifies what is true of many codes. That is, often there is residual error-correcting capability beyond the value t. A t-error correcting code is called a *perfect code* if its standard array has all the error patterns of t and fewer errors and no others as coset leaders (no residual error-correcting capability). Hamming codes are perfect codes that can correct single errors only; the structure of the standard array can be used to confirm this. Hamming codes are characterized by (n, k) dimensions as follows:

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

where $m = 3, 4,$ Thus the number of cosets is $2^{n-k} = 2^m$ since $n - k = m$. Because $n = 2^{m-1}$, there are 2^{m-1} ways of making single errors. Thus, the number of cosets, 2^m, equals exactly 1 (for the no-error case) plus the number of ways that one error in n bits can be made. Hence, all Hamming codes are indeed perfect codes that can correct single errors only.

A somewhat similar situation occurs for the triple-error correcting (23, 12) Golay code, which is a perfect code. Observe that for this code, the number of cosets in the standard array is $2^{n-k} = 2^{11} = 2048$. Following the format of Table 1, we develop Table 2 for the (23, 12) Golay code, as shown below:

Table 2

Error-Correction Bound for the (23, 11) Golay Code

Number of Bit Errors	Number of Cosets Required	Cumulative Number of Cosets Required
0	1	1
1	23	24
2	253	277
3	1,771	2,048

Table 2 demonstrates that the (23, 11) Golay code is indeed a perfect code since it has no residual error-correcting capability beyond $t = 3$.

An (*n, k*) Example

The standard array provides insight into the tradeoffs that are possible between error correction and detection. Consider a new (*n, k*) code example, and the factors that dictate what values of (*n, k*) should be chosen.

1. To perform a nontrivial tradeoff between error correction and error detection, it is desired that the code have an error-correcting capability of at least $t = 2$.

2. The Hamming distance between two codewords is the number of bit positions in which the two codewords differ. The smallest Hamming distance among all codewords comprising a code is called the *minimum distance*, $d_{\min}$, of the code. For error-correcting capability of $t = 2$, we use the following fundamental relationship [1] for finding the minimum distance:

$$d_{\min} = 2t + 1 = 5$$

3. For a nontrivial code system, it is desired that the number of data bits be at least $k = 2$. Thus, there will be $2^k = 4$ codewords. The code can now be designated as an (n, 2) code.

4. We look for the minimum value of n that will allow correcting all possible single and double errors. In this example, each of the 2^n n-tuples in the array will be tabulated. The minimum value of n is desired because whenever n is incremented by just a single integer, the number of n-tuples in the standard array doubles. Of course, it is desired that the list be of manageable size. For "real world" codes, we want the minimum n for different reasons—bandwidth efficiency and simplicity. If the Hamming bound is used in choosing n, then $n = 7$ could be selected. However, the dimensions of such a (7, 2) code will not meet our stated requirements of $t = 2$-bit error-correction capability and $d_{\min} = 5$. To see this, it is necessary to introduce another upper bound on the t-bit error correction capability (or $d_{\min}$). This bound, called the *Plotkin bound* [2], is described below:

$$d_{\min} \leq \frac{n \times 2^{k-1}}{2^k - 1} \tag{4}$$

In general, a linear (n, k) code must meet all upper bounds involving error-correction capability (or minimum distance). For high-rate codes, if the Hamming bound is met, the Plotkin bound will also be met; this was the case for the earlier (127, 106) code example. For low-rate codes, it is the other way around [2]. Since this example entails a low-rate code, it is important to test error-correction capability via the Plotkin bound. Because $d_{\min} = 5$, it should be clear from Equation (4) that n must be 8, and therefore, the minimum dimensions of the code are (8, 2) in order to meet the requirements for this example.

Designing the (8, 2) Code

A natural question to ask is, "For a linear code, how does one select codewords out of the space of 2^8 8-tuples?" There is no single solution, but there are constraints in how choices are made. Here are the elements that help point to a solution.

1. The number of codewords is $2^k = 2^2 = 4$.

2. The property of closure must apply. This property dictates that the sum of any two codewords in the space must yield a valid codeword in the space.

3. The all-zeros vector must be one of the codewords. This property is the result of the closure property, since any codeword that is added (modulo-2) to itself yields an all-zeros vector.

4. Each codeword is 8 bits long.

5. Since $d_{\min} = 5$, the weight of each codeword (except for the all-zeros codeword) must also be at least 5 (by virtue of the closure property). The weight of a vector is defined as the number of nonzero components in the vector.

6. Assume that the code is systematic, so the rightmost 2 bits of each codeword are the corresponding message bits.

Following is a candidate assignment of codewords to messages that meets all of the above conditions.

Messages	**Codewords**
0 0	0 0 0 0 0 0 0 0
0 1	1 1 1 1 0 0 0 1
1 0	0 0 1 1 1 1 1 0
1 1	1 1 0 0 1 1 1 1

(5)

The design of the codeword set can begin in a very arbitrary way; it is only necessary to adhere to the properties of weight and systematic form of the code. The selection of the first few codewords is often simple. However, as the process continues the selection routine becomes harder, and the choices become more constrained because of the need to adhere to the closure property.

Encoding, Decoding, and Error Correction

The generation of a codeword $\mathbf{U}_i$ in an (n, k) code involves forming the product of a k-bit message vector $\mathbf{m}_i$ and a $k \times n$ generator matrix $\mathbf{G}$ [1]. For the code system in Equation (5), $\mathbf{G}$ can be written as shown in Equation (6):

$$\mathbf{G} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

Forming the product $\mathbf{mG}$ for all the messages in Equation (5) will yield all the codewords shown in that equation.

Decoding starts with the computation of a syndrome, which can be thought of as learning the "symptom" of an error. For an (n, k) code, an $(n - k)$-bit syndrome, $\mathbf{s}$, is the product of an n-bit received vector, $\mathbf{r}$, and the transpose of an $(n - k) \times n$ parity-check matrix, $\mathbf{H}$, [1] where $\mathbf{H}$ is constructed so that the rows of $\mathbf{G}$ are orthogonal to the rows of $\mathbf{H}$; that is, $\mathbf{GH}^T = \mathbf{0}$. For this (8, 2) example, $\mathbf{s}$ is a 6-bit vector, and $\mathbf{H}$ is a 6 x 8 matrix, where $\mathbf{H}^T$ is written as shown in Equation (7):

$$\mathbf{H}^T = \begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{matrix} \tag{7}$$

The syndrome for each error pattern can be calculated as shown in Equation (8):

$$\mathbf{s}_i = \mathbf{e}_i\, \mathbf{H}^T \qquad i = 1, \ldots, 2^{n-k} \tag{8}$$

Figure 1 shows a tabulation of all $2^{n-k} = 64$ syndromes as well as the standard array for the (8, 2) code. Each row (except the first) of the standard array represents a *set* of corrupted codewords with something in *common*, hence the name *coset*. What do the entries in any one coset have in common? They have the same syndrome. After computing the syndrome, the correction of a corrupted codeword proceeds by locating the error pattern that corresponds to that syndrome. Finally, the error pattern is subtracted (modulo-2 added) from the corrupted codeword, yielding the corrected output.

SYNDROMES		STANDARD ARRAY			
000000	1.	00000000	11110001	00111110	11001111
111100	2.	00000001	11110000	00111111	11001110
001111	3.	00000010	11110011	00111100	11001101
000001	4.	00000100	11110101	00111010	11001011
000010	5.	00001000	11111001	00110110	11000111
000100	6.	00010000	11100001	00101110	11011111
001000	7.	00100000	11010001	00011110	11101111
010000	8.	01000000	10110001	01111110	10001111
100000	9.	10000000	01110001	10111110	01001111
110011	10.	00000011	11110010	00111101	11001100
111101	11.	00000101	11110100	00111011	11001010
111110	12.	00001001	11111000	00110111	11000110
111000	13.	00010001	11100000	00101111	11011110
110100	14.	00100001	11010000	00011111	11101110
101100	15.	01000001	10110000	01111111	10001110
011100	16.	10000001	01110000	10111111	01001110
001110	17.	00000110	11110111	00111000	11001001
001101	18.	00001010	11111011	00110100	11000101
001011	19.	00010010	11100011	00101100	11011101
000111	20.	00100010	11010011	00011100	11101101
011111	21.	01000010	10110011	01111100	10001101
101111	22.	10000010	01110011	10111100	01001101
000011	23.	00001100	11111101	00110010	11000011
000101	24.	00010100	11100101	00101010	11011011
001001	25.	00100100	11010101	00011010	11101011
010001	26.	01000100	10110101	01111010	10001011
100001	27.	10000100	01110101	10111010	01001011
000110	28.	00011000	11101111	00100110	11010111
001010	29.	00101000	11011001	00010110	11100111
010010	30.	01001000	10111001	01110110	10000111
100010	31.	10001000	01111001	10110110	01000111
001100	32.	00110000	11000001	00001110	11111111
010100	33.	01010000	10100001	01101110	10011111
100100	34.	10010000	01100001	10101110	01011111
011000	35.	01100000	10010001	01011110	10101111
101000	36.	10100000	01010001	10011110	01101111
110000	37.	11000000	00110001	11111110	00001111
110010	38.	00000111	11110110	00111001	11001000
110111	39.	00010011	11100010	00101101	11011100
111011	40.	00100011	11010010	00011101	11101100
100011	41.	01000011	10110010	01111101	10001100
010011	42.	10000011	01110010	10111101	01001100
111111	43.	00001101	11111100	00110011	11000010
111001	44.	00010101	11100100	00101011	11011010
110101	45.	00100101	11010100	00011011	11101010
101101	46.	01000101	10110100	01111011	10001010
011101	47.	10000101	01110100	10111011	01001010
011110	48.	01000110	10110111	01111000	10001001
101110	49.	10000110	01110111	10111000	01001001
100101	50.	10010100	01100101	10101010	01011011
011001	51.	01100100	10010101	01011010	10101011
110001	52.	11000100	00110101	11111010	00001011
011010	53.	01101000	10011001	01010110	10100111
010110	54.	01011000	10101001	01100110	10010111
100110	55.	10011000	01101001	10100110	01010111
101010	56.	10101000	01011001	10010110	01100111
101001	57.	10100100	01010101	10011010	01101011
100111	58.	10100010	01010011	10011100	01101101
010111	59.	01100010	10010011	01011100	10101101
010101	60.	01010100	10100101	01101010	10011011
011011	61.	01010010	10100011	01101100	10011101
110110	62.	00101001	11011000	00010111	11100110
111010	63.	00011001	11101000	00100111	11010110
101011	64.	10010010	01100011	10101100	01011101

Figure 1

The syndromes and the standard array for the (8, 2) code.

Since each coset has the same syndrome, Equation (8) can be written in terms of a received vector, **r**, as follows:

$$\mathbf{s}_i = \mathbf{r}_i \, \mathbf{H}^T \tag{9}$$

The vectors $\mathbf{U}_i$, $\mathbf{e}_i$, $\mathbf{r}_i$, and $\mathbf{s}_i$ can each be described as having the following general form:

$$\mathbf{x}_i = \{ x_1, x_2, \ldots, x_j, \ldots \}$$

For the codeword $\mathbf{U}_i$ in this example, the index $i = 1, \ldots, 2^k$ indicates that there are 4 distinct codewords, and the index $j = 1, \ldots, n$ indicates that there are 8 bits per codeword. For the received vector $\mathbf{r}_i$, the index $i = 1, \ldots, 2^n$ indicates that there are 256 distinct vectors, and the index $j = 1, \ldots, n$ indicates that there are 8 digits per vector. For the error pattern $\mathbf{e}_i$, the index $i = 1, \ldots, 2^{n-k}$ indicates that there are 64 distinct correctable error patterns, and the index $j = 1, \ldots, n$ indicates that there are 8 digits per error pattern. For the syndrome $\mathbf{s}_i$, the index $i = 1, \ldots, 2^{n-k}$ indicates that there are 64 distinct syndromes, and the index $j = 1, \ldots, n-k$ indicates that there are 6 digits per syndrome. For simplicity, the index i is dropped, and the vectors $\mathbf{U}_i$, $\mathbf{e}_i$, $\mathbf{r}_i$, and $\mathbf{s}_i$ will be denoted as **U**, **e**, **r**, and **s**, respectively, where in each case some ith vector is implied.

Error Detection Versus Error Correction Tradeoffs

Using the codeword set in Equation (5), the standard array is constructed (refer to Figure 1). Error-detection and error-correction capabilities can be traded, provided that the following distance relationship prevails [1]:

$$d_{\min} \geq \alpha + \beta + 1 \tag{10}$$

where α represents the number of bit errors to be corrected, β represents the number of bit errors to be detected, and $\beta \geq \alpha$. The tradeoff choices available for the (8, 2) code example are as follows:

Detection (β)	Correction (α)
2	2
3	1
4	0

This table shows that the (8, 2) code can be implemented to perform only error correction, which means that it first detects as many as $\beta = 2$ errors, and then corrects them. If some error correction is sacrificed so that the code will only correct single errors, then the detection capability is increased so that all $\beta = 3$ errors can be detected. And finally, if error correction is completely sacrificed, the decoder can be implemented so that all $\beta = 4$ errors can be detected. In the case of error detection only, the circuitry is very simple. The syndrome is computed and an error is detected whenever a nonzero syndrome occurs.

The decoder circuit for correcting single errors can be implemented with logic gates [3], as shown in Figure 2. The exclusive-OR (EX-OR) gate performs the same operation as modulo-2 arithmetic and hence uses the same symbol. The AND gates are shown as half-circles. A small circle at the termination of any line entering an AND gate indicates the logic-COMPLEMENT of the binary state. In this figure, entering the decoder at two places simultaneously is a received vector, **r**. In the upper part of the figure, the 8 digits of the received vector are loaded into a shift register whose stages are connected to 6 EX-OR gates, each of which yield a syndrome bit s_j, where $j = 1, \ldots ,6$. The circuit wiring between the received vector, **r**, and the EX-OR gates is dictated by Equation (9), as follows:

$$\mathbf{s} = [\, r_1\ r_2\ r_3\ r_4\ r_5\ r_6\ r_7\ r_8 \,] \begin{matrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{matrix} \tag{11}$$

Therefore each of the s_j digits comprising syndrome **s** can be described from Equation (11) as related to the r_j digits of the received vector in the following way:

$$s_1 = r_1 + r_8 \qquad s_2 = r_2 + r_8 \qquad s_3 = r_3 + r_7 + r_8$$

$$s_4 = r_4 + r_7 + r_8 \qquad s_5 = r_5 + r_7 \qquad s_6 = r_6 + r_7$$

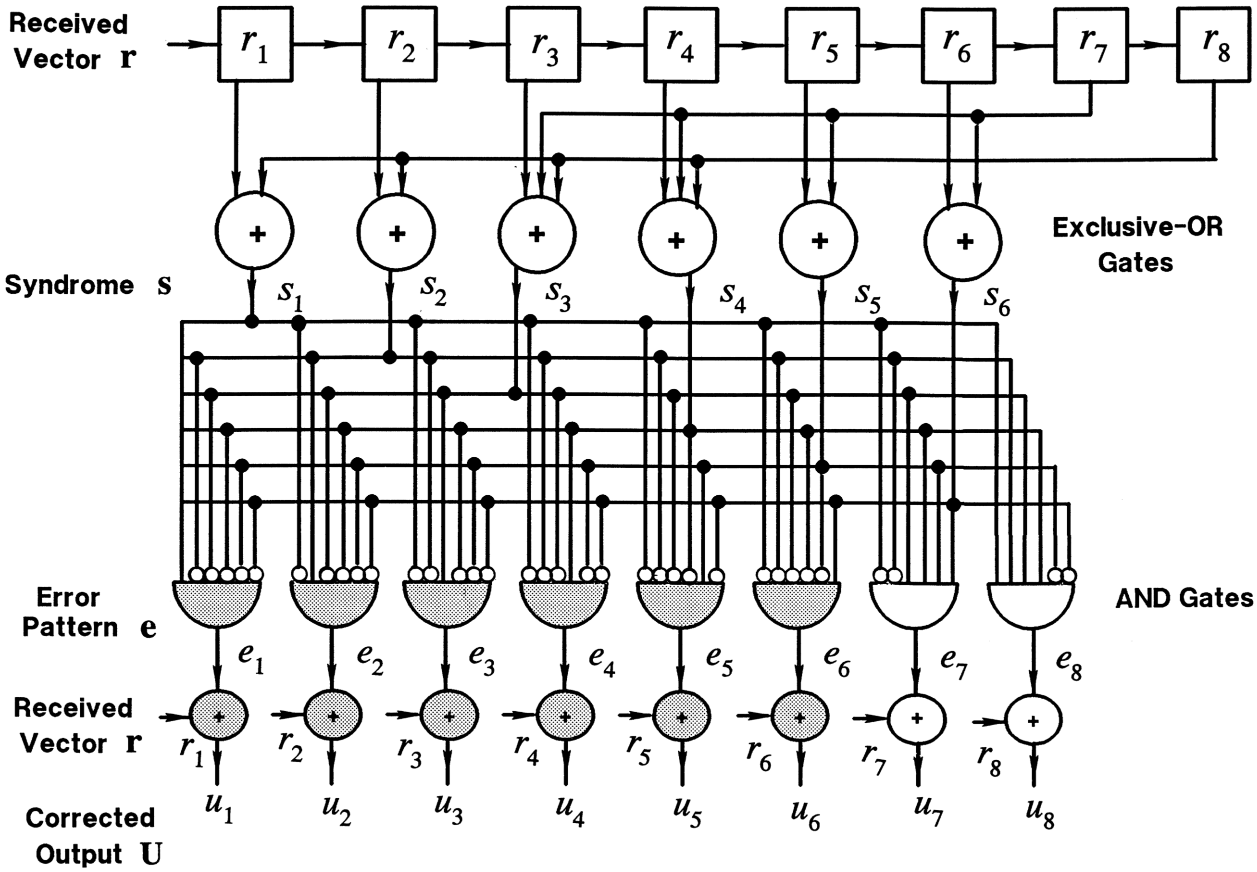

Figure 2

Decoding circuit for an (8, 2) code.

If the decoder is implemented to correct only single errors, that is $\alpha = 1$ and $\beta = 3$, then this is tantamount to drawing a line under coset 9 in Figure 1, and error correction takes place only when one of the 8 syndromes associated with a single error appears. It is easy to verify that the AND gates in Figure 2 convert any syndrome, numbered 1 through 9, to the corresponding error pattern $\mathbf{e}_i$. The error pattern is then subtracted (modulo-2 added) from the "potentially" corrupted received vector, yielding a corrected output, **U**. Additional gates are needed to test for the case when the syndrome is nonzero but the outputs of the AND gates are all zero—such an event happens for any of the syndromes numbered 10 through 64. This outcome is then used to indicate an error detection. Note that Figure 2, for tutorial reasons, has been drawn to emphasize the algebraic decoding steps—calculation of syndrome, error pattern, and finally corrected output. In the "real world," an (n, k) code is usually configured in systematic form, which means that the rightmost codeword digits are the k data bits, and the balance of the codeword consists of the $n - k$ parity bits. The decoder does not need to deliver the entire codeword; its output can consist of the data bits only. Hence, the Figure 2 circuitry becomes simplified by eliminating the gates that are shown with shading.

Notice that the process of decoding a corrupted codeword by first detecting and then correcting an error can be compared to a familiar medical analogy. A patient (a potentially corrupted codeword) enters a medical facility (a decoder). The examining physician performs diagnostic testing (multiplies by $\mathbf{H}^T$) in order to find a symptom (a syndrome). Imagine that the physician finds characteristic spots on the patient's x-rays. An experienced physician would immediately recognize the correspondence between the symptom and the disease (error pattern), say tuberculosis. A novice physician might have to refer to a medical handbook to associate the symptom with the disease (that is, syndrome versus error pattern as listed in Figure 1, or as formed by AND gates in Figure 2). The final step provides the proper medication to the patient, thereby removing the disease (in other words, adds the error pattern modulo-2 to the corrupted codeword, thereby correcting the flawed codeword). In the context of binary codes, an unusual type of medicine is being practiced here. The patient is cured by reapplying the original disease.

If the decoder is implemented to perform error correction only, then $\alpha = 2$ and $\beta = 2$. For this case, detection and correction of all single and double errors can be envisioned as drawing a line under coset 37 in the standard array of Figure 1. Even though the (8, 2) code is capable of correcting some combination of triple errors corresponding to the coset leaders 38 through 64, a decoder is most often implemented as a *bounded distance decoder*, which means that it corrects all combinations of errors up to and including t errors, but no combinations of errors greater than t. The decoder can again be realized with logic gates, using an implementation that is similar to the circuit in Figure 2.

Even though a small code was used to describe these tradeoffs, the example can be expanded (without entering details into the standard array) for any size code. The circuitry in Figure 2 performs decoding in a parallel manner, which means that all of the digits of the codeword are decoded simultaneously. Such decoders are useful only for relatively small codes. When the code is large, this parallel implementation becomes very complex, and one generally chooses a simpler sequential approach (which will require more processing time than the parallel circuitry) [3].

The Standard Array Provides Insight

In the context of Figure 1, the (8, 2) code satisfies the Hamming bound. That is, from the standard array it is recognizable that the (8, 2) code can correct all combinations of single and double errors. Consider the following question: “Suppose that transmission takes place over a channel that always introduces errors in the form of a burst of 3-bit errors, so that there is no interest in correcting single or double errors; wouldn’t it be possible to set up the coset leaders to correspond to only triple errors?” It is simple to see that in a sequence of 8 bits there are $\binom{8}{3} = 56$ ways to make triple errors. If we only want to correct all these 56 combinations of triple errors, there is sufficient room (sufficient number of cosets) in the standard array, since there are 64 rows. Won’t that work? No, it won’t. For any code, the overriding parameter for determining error-correcting capability is $d_{\min}$. For the (8, 2) code, $d_{\min} = 5$ dictates that only 2-bit error correction is possible.

How can the standard array provide some insight as to why this scheme won’t work? For a group of x-bit error patterns to enable x-bit error correction, the entire group of weight-x vectors must be coset leaders; that is, they must occupy only the leftmost column. In Figure 1, all weight-1 and weight-2 vectors appear in the leftmost column of the standard array, and nowhere else. Even if we forced all weight-3 vectors into row numbers 2 through 57, we would find that some of these vectors would have to reappear elsewhere in the array (which violates a basic property of the standard array). In Figure 1, a shaded box is drawn around each of the 56 vectors having a weight of 3. Look at the coset leaders representing 3-bit error patterns, in rows 38, 41–43, 46–49, and 52 of the standard array. Now look at the entries of the same row numbers in the rightmost column, where shaded boxes indicate other weight-3 vectors. Do you see the ambiguity that exists for each of the rows listed above, and why it is not possible to correct all 3-bit error patterns with this (8, 2) code? Suppose the decoder receives the weight-3 vector 1 1 0 0 1 0 0 0, located at row 38 in the rightmost column. This flawed codeword could have arisen in one of two ways. One way would be that codeword 1 1 0 0 1 1 1 1 was sent and the 3-bit error pattern 0 0 0 0 0 1 1 1 perturbed it. The other possibility would be that codeword 0 0 0 0 0 0 0 0 was sent and the 3-bit error pattern 1 1 0 0 1 0 0 0 perturbed it.

Conclusion

In this article, basic principles of block codes were reviewed, emphasizing the structure of the standard array. We used examples involving bounds, perfect codes, and implementation tradeoffs in order to gain some insight into the algebraic structure of linear block codes. Also, we showed how the standard array offers intuition as to why some desired error-correcting properties for a particular code might not be possible.

References

[1] Sklar, B., *Digital Communications: Fundamentals and Applications, Second Edition* (Upper Saddle River, NJ: Prentice-Hall, 2001).

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

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

About the Author

Bernard Sklar is the author of *Digital Communications: Fundamentals and Applications, Second Edition* (Prentice-Hall, 2001, ISBN 0-13-084788-7).