USING THE MATLAB COMMUNICATIONS TOOLBOX[1] TO LOOK AT CYCLIC CODING

Wm. Hugh Blanton
East Tennessee State University

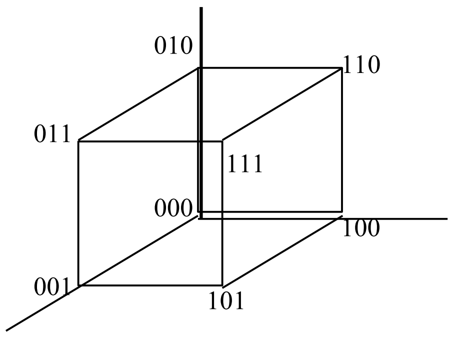
ABSTRACT

In wireless digital communications, the designer is constantly trying to minimize the probability of bit error rates within certain constraints, most notably signal power limits. One method of compensating for bit errors is the use of error control coding that provides sufficient structure to the signal to provide the location of the error. Error control coding requires circuits capable of performing matrix multiplication and comparing the result of various binary numbers. Although the concepts are relatively simple, the implementation becomes rapidly complex as the length of the code word and the uncoded message increase. As a result, most coding theory uses a *(7,4)* code in which the code word has seven bits of which four bits contain the information.[2] The code results in a manageable number of 128 code words of which only 16 form valid codes. The redundancy is used for error correction. Now suppose a *(15,7)* code is used allowing 32,768 possible code words for which only 128 are valid information codes. This complexity can be reduced by using several functions in the Matlab Communications Toolbox, providing a unique learning opportunity for the engineering technology student.

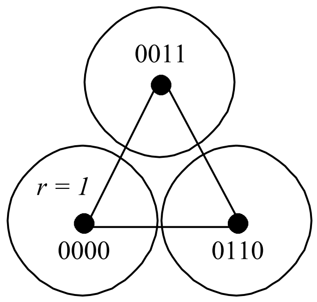
INTRODUCTION[2]

Channel coding refers to the class of signal transformations designed to improve communications performance by enabling the transmitted signals to better withstand the effects of various channel impairments, such as noise, interference, and fading. These signal processing techniques can be thought of as vehicles for accomplishing desirable system trade offs (e.g., error performance vs. bandwidth, power vs. bandwidth).

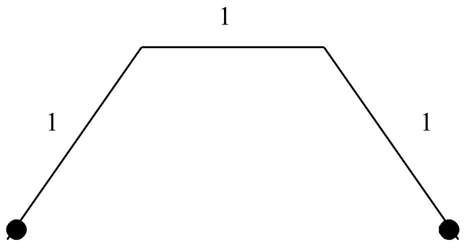
Much of the theory is based upon the *Hamming distance* which is defined as the number of bit positions in which two binary words differ. For example, consider the following figure.



The minimum distance between code words is 1. If a single bit error occurs in the three bit code, the error word will result from moving along one of the edges to an adjacent corner. If the minimum distance between code words is 2, the code words are separated by at least two edges. This is represented conceptually below where the radius of 1 implies all possible words within a distance of 1 from the center. Since the valid codes are separated by a *Hamming distance* of 2, a code word that differs by a *Hamming distance* of 1 will represent an erroneous code word.



If the minimum distance is 3, the code words are separated by at least three edges, shown conceptually below. Note that if a single error occurs, the erroneous code word will be closer to one of the correct code words. The error is corrected by assigning the received code word to the nearest (in *Hamming distance)* valid code word.



The important concept associated with *Hamming* distance is:

1. When the *Hamming distance* is 1, it is impossible to detect an error, much less correct the error.
2. When the *Hamming distance* is 2, the error can be detected, but not corrected.
3. When the *Hamming distance* is 3, a single error can be detected and corrected.

Clearly, *Hamming distance* is an important concept in the detection and correction of bit errors. But how do we expand this concept to realistic problems?

LINEAR BLOCK CODES[2]

A systematic *(n,k)* linear block code is a mapping from a *k*-dimensional message vector to an *n*-dimensional codeword in such a way that part of the sequence generated coincides with the *k* message digits. The difference $(n - k)$ represents the parity bits. A systematic linear block will have a $[k \times n]$ generator matrix $[G]$ of the form

$$G = \left[P \middle| I_k\right]$$

$$= \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} & 1 & 0 & \cdots & 0 \\ p_{21} & p_{21} & \cdots & p_{2m} & 0 & 1 & \cdots & 0 \\ & & \cdots & & & & \cdots & \\ p_{n1} & p_{n2} & \cdots & p_{nm} & 0 & 0 & \cdots & 1 \end{bmatrix}$$

Thus, a *(7,4)* code is generated by the matrix

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

There are four informational bits and three parity bits. The block code as shown in the table below is generated using the matrix equation

$$v = u[G]$$

where

u is a $[1 \times k]$ vector representing the information
v is a $[1 \times n]$ vector representing the code word
$[G]$ is $[k \times n]$ generating matrix

Note that block coding uses *modulo-2 addition* (*exclusive-OR* operation). If the distance between every pair of code words (120 pairs), the *Hamming distance* is found to be 3. This code can detect double bit errors and correct single bit errors.

Information	Code
0000	0000000
0001	1010001
0010	1110010
0011	0100011
0100	0110100
0101	1100101
0110	1000110
0111	0010111
1000	1101000
1001	0111001
1010	0011010
1011	1001011
1100	1011100
1101	0001101
1110	0101110
1111	1111111

Another important matrix associated with block codes is the *[(n − k) × n] parity check matrix*, *[H]*. The parity check matrix is formed by starting with the identity matrix and appending the transpose of the *nonidentity* portion of *[G]*:

$$H = \left\lfloor I_k \middle| P^T \right\rfloor$$

$$= \begin{bmatrix} 1 & 0 & \dots & 0 & p_{11} & p_{21} & \cdots & p_{n1} \\ 0 & 1 & \dots & 0 & p_{12} & p_{22} & \cdots & p_{n2} \\ & & \dots & & & & \cdots & \\ 0 & 0 & \dots & 1 & p_{1m} & p_{2m} & \cdots & p_{nm} \end{bmatrix}$$

For the *(7,4)* block code,

$$H = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

The parity check matrix has the property

$$v[H]^T = 0$$

That is, any errorless, received code word multiplied by the transpose of the parity check matrix, *[H]*, yields a zero vector, or *syndrome*. If the received code word contains an error, the resulting vector will match the corresponding bit that caused the error.

For example, consider one of the valid code words of the *(7,4)* block code, $v = 1100101$ and the transpose of the *(7,4)* parity check matrix above. Then,

$$v[H]^T = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

The product of the (1 × 7) row vector with a (7 × 3) yields a (1 × 3) *syndrome*. The matrix product of the row vector with the first column of the parity check matrix produces

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = 1\times 1 \oplus 1\times 0 \oplus 0\times 0 \oplus 0\times 1 \oplus 1\times 0 \oplus 0\times 1 \oplus 1\times 1$$

where the symbol $\oplus$ represents binary addition (*exclusive-OR* operation in Boolean algebra).

That is, binary addition produces the following variations:

$$
\begin{aligned}
0 \oplus 0 &= 0 \\
1 \oplus 0 &= 1 \\
0 \oplus 1 &= 1 \\
1 \oplus 1 &= 0
\end{aligned}
$$

The product of the row vector and the second column of the transpose for the parity check matrix is

$$
\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}
= 1\times 0 \oplus 1\times 1 \oplus 0\times 1 \oplus 0\times 1 \oplus 1\times 1 \oplus 0\times 1 \oplus 1\times 0
$$

$$
= 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 = 1 \oplus 1 = 0
$$

The product of the row vector and the third column of the transpose for the parity check matrix is

$$
\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}
= 1\times 0 \oplus 1\times 0 \oplus 0\times 0 \oplus 0\times 0 \oplus 1\times 1 \oplus 0\times 1 \oplus 1\times 1
$$

$$
= 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 1 = 1 \oplus 1 = 0
$$

With no errors, the *syndrome*, vH^T, is

$$vH^T = \begin{bmatrix}1 & 1 & 0 & 0 & 1 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 1 & 0\\ 1 & 1 & 0\\ 0 & 1 & 1\\ 1 & 1 & 1\\ 1 & 0 & 1\end{bmatrix} = \begin{bmatrix}0 & 0 & 0\end{bmatrix}$$

Suppose the least significant bit of the code, $v = 110010\underline{\mathbf{1}}$, becomes corrupted yielding $110010\underline{\mathbf{0}}$. The *syndrome* is now

$$\begin{bmatrix}1 & 1 & 0 & 0 & 1 & 0 & 0\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 1 & 0\\ 1 & 1 & 0\\ 0 & 1 & 1\\ 1 & 1 & 1\\ 1 & 0 & 1\end{bmatrix} = [(1\oplus 0\oplus 0\oplus 0\oplus 0\oplus 0\oplus 0)\quad (0\oplus 1\oplus 0\oplus 0\oplus 1\oplus 0\oplus 0)\quad (0\oplus 0\oplus 0\oplus 0\oplus 1\oplus 0\oplus 0)]$$

$$= \begin{bmatrix}1 & 0 & 1\end{bmatrix}$$

Since the syndrome has non-zero values, an error has been detected. Note that the value of the syndrome corresponds to the value in the last row of the transposed parity check matrix ([1 0 1]) = [1 0 1]). This is the seventh row that corresponds to seventh bit of the code word, indicating that bit 7 of the transmitted code word has been corrupted. Knowing where the error occurred, the error can be corrected by inverting bit 7 of the code word.

To reinforce the concept, suppose the code word, $v = 110\underline{\mathbf{0}}101$, becomes corrupted yielding $110\underline{\mathbf{1}}101$.

$$\begin{bmatrix}1 & 1 & 0 & 1 & 1 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 1 & 0\\ 1 & 1 & 0\\ 0 & 1 & 1\\ 1 & 1 & 1\\ 1 & 0 & 1\end{bmatrix} = [(1\oplus 0\oplus 0\oplus 1\oplus 0\oplus 0\oplus 1)\quad (0\oplus 1\oplus 0\oplus 1\oplus 1\oplus 0\oplus 0)\quad (0\oplus 0\oplus 0\oplus 0\oplus 1\oplus 0\oplus 1)]$$

$$= \begin{bmatrix}1 & 1 & 0\end{bmatrix}$$

Again the syndrome has non-zero values indicating an error. The syndrome [1 1 0] corresponds to the value [1 1 0] in the fourth row in the matrix. The fourth row corresponds to the fourth bit of the code word. By inverting the fourth bit, *110**1**101* will be converted to the correct value, *110**0**101.*

Each row in the transposed parity check matrix corresponds to a bit in the code word. By matching the non-zero syndrome with the contents contained in the rows of the transposed parity matrix, the corresponding corrupted bit can be detected and corrected.

CYCLIC CODES[2]

The implementation of linear block codes requires circuits capable of performing matrix multiplication and of comparing the result of various binary numbers[1]. Although integrated circuits have been developed to implement the most common codes, the circuitry can become quite complex for very long blocks of code. A special case of the block code, the c*yclic code* can be implemented relatively easily. For example, take the codeword $c=(c_1,c_2,...,c_n)$, then

$(c_2,c_3,...,c_n,c_1)$, and
$(c_3,c_4,...,c_n,c_1,c_2)$, etc.

are also code words. This structure enables cyclic codes to correct larger blocks of errors, than what non-cyclic block codes are capable of correcting, and specific rules for generating these codes may be set up.

The generating matrix can be derived from the generating polynomial using the following two theorems:

1. If $g(X)$ is a polynomial of degree $n - k$ and is a factor of $X^n + 1$, then $g(X)$ generates an (n,k) cyclic code.
2. Any *irreducible* polynomial of degree i is a factor of $X^{2i-1} + 1)$.

Given

$$X^7 + 1 = (1 + X)(1 + X + X^3)(1 + X^2 + X^3)$$ *--Remember we are using modulo-2 arithmetic*

Theorem 1 implies that $1 + X$ can generate a $(7,6)$ cyclic code, and either $1 + X + X^3$ or $1 + X^2 + X^3$ can generate a $(7,4)$ cyclic code. Note that $1 + X + X^3$ is irreducible. Thus, theorem 2 implies that $1 + X + X^3$ is a factor of $X^{2i-1} + 1$.

The generating matrix is derived from the coefficients of the generating polynomial by listing the coefficients in the first row and then shifting them one position to the right. For $g(X) = 1 + X + X^3$, the generating matrix is

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

Note that the generating matrix is *nonsystematic*, since there is no *identity* matrix existing as part of the generating matrix. Using row operations on the matrix, the *nonsystematic* generating matrix can be converted into the *systematic* generating matrix

$$G_{sys} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

MATLAB COMMUNICATIONS TOOLBOX DERIVATION

Although the previous discussion was not mathematically complicated, it was mathematically tortuous. Moreover, the discussion for the *(7,4)* cyclic code is relatively benign with a manageable number of calculations. For larger codes, the calculations become overwhelming, especially for the engineering technology student. In order to better manage larger codes, relieve the mathematical stress on engineering technology students, and focus on the application of block codes rather than the mathematics; the MATLAB Communications Toolbox provides a set of cyclic code functions.

Consider the *(7,4)* cyclic code. MATLAB Communications Toolbox can be used to find all the cyclic generating polynomials:

```
>> poly=cyclpoly(7,4,'all')

poly =

1   0   1   1
1   1   0   1
```

Thus, the generating polynomials are: $1 + x^2 + x^3$ and $1 + x + x^3$.

Suppose the second generating polynomial above is chosen for the calculations. The code that creates the nonsystematic generating matrix, *genmat*, and the parity matrix, *parmat*, is:

```
>> genpoly=[1 1 0 1];
>> [parmat,genmat]=cyclgen(7,genpoly,'nonsys')

parmat =

     1   0   1   1   1   0   0
     0   1   0   1   1   1   0
     0   0   1   0   1   1   1

genmat =

     1   1   0   1   0   0   0
     0   1   1   0   1   0   0
     0   0   1   1   0   1   0
     0   0   0   1   1   0   1
```

As one would suspect, the computer generated generating matrix (*genmat)* is equal to the calculated generating matrix *(G)*.

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

genmat =

The *systematic* generating matrix is found by

```
[parmatsys,genmatsys]=cyclgen(7,genpoly)

parmatsys =

     1   0   0   1   0   1   1
     0   1   0   1   1   1   0
     0   0   1   0   1   1   1
```

```
genmatsys =

     1     1     0     1     0     0     0
     0     1     1     0     1     0     0
     1     1     1     0     0     1     0
     1     0     1     0     0     0     1
```

$$\text{genmatsys} = G_{sys} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The MATLAB Communications Toolbox derives the same generating matrices that were derived by mathematical manipulation. The underlying result is that the electronic engineering technology student can apply more effort in understanding the theory rather than the mathematics, especially when considering more complex problems.

STUDENT PROBLEM

The *(7,4)* code is important, if only for historical insight. Practically every digital communications textbook refers to the *(7,4)* code. The code extends the engineering students to their matrix manipulation limits. Unfortunately, the *(7,4)* code surpasses many engineering technology students' matrix manipulation limits. The dilemma is to present a nontrivial, doable problem that reinforces cyclic code theory learned using the *(7,4)* code, but does not mathematically intimidate the student.

Such a problem is a *(15,7)* cyclic code with generator polynomial:

$$1 + x + x^3 + x^4 + x^5 + x^7 + x^8 .$$ [3]

The nonsystematic 7 × 15 generating matrix, G, can be derived as:

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

and the systematic generating matrix is derived by row, column matrix manipulation of the nonsystematic generating matrix until the identity matrix appears in the latter part of the generating matrix as shown below.

$$G_{sys} = \left[\begin{array}{cccccccc:ccccccc} 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

Although row,column matrix manipulation is a straightforward process for students that have taken a linear algebra course, many engineering technology students are not familiar with the process. By using the MATLAB Communications Toolbox, not only can the student do the transformation from the nonsystematic to the systematic generating matrix, but the students have access to several other functions that simplify the analysis and enhance the understanding of cyclic codes.

By using the function *cyclpoly*, the computer analysis yields a set of all generating polynomials

```
>> p=cyclpoly(15,7,'all')

p =

     1    0    0    0    1    0    1    1    1
     1    1    1    0    1    0    0    0    1
     1    1    0    1    1    1    0    1    1
```

which corresponds to:

1. $1+x^4+x^6+x^7+x^8$
2. $1+x^1+x^2+x^4+x^8$
3. $1+x+x^3+x^4+x^5+x^7+x^8$

Any one of the generating polynomials can be used to create a generating matrix. The third generating polynomial above ($1 + x + x^3 + x^4 + x^5 + x^7 + x^8$) is the generating polynomial used in this problem.

By using the function *cyclgen*, the nonsystematic generating matrix, *genmat*, is derived as follows:

```
>> genpoly=[1 1 0 1 1 1 0 1 1];
>> [parmat,genmat]=cyclgen(15,genpoly,'nonsys')


genmat =

    Columns 1 through 13

    1   1   0   1   1   1   0   1   1   0   0   0   0
    0   1   1   0   1   1   1   0   1   1   0   0   0
    0   0   1   1   0   1   1   1   0   1   1   0   0
    0   0   0   1   1   0   1   1   1   0   1   1   0
    0   0   0   0   1   1   0   1   1   1   0   1   1
    0   0   0   0   0   1   1   0   1   1   1   0   1
    0   0   0   0   0   0   1   1   0   1   1   1   0


    Columns 14 through 15

    0   0
    0   0
    0   0
    0   0
    0   0
    1   0
```

Cyclgen also generates the parity matrix.

Putting *genmat* into matrix form and comparing the result to a manually derived generating matrix, we get:

$$genmat = \begin{bmatrix} 1&1&0&1&1&1&0&1&1&0&0&0&0&0&0 \\ 0&1&1&0&1&1&1&0&1&1&0&0&0&0&0 \\ 0&0&1&1&0&1&1&1&0&1&1&0&0&0&0 \\ 0&0&0&1&1&0&1&1&1&0&1&1&0&0&0 \\ 0&0&0&0&1&1&0&1&1&1&0&1&1&0&0 \\ 0&0&0&0&0&1&1&0&1&1&1&0&1&1&0 \\ 0&0&0&0&0&0&1&1&0&1&1&1&0&1&1 \end{bmatrix} = G = \begin{bmatrix} 1&1&0&1&1&1&0&1&1&0&0&0&0&0&0 \\ 0&1&1&0&1&1&1&0&1&1&0&0&0&0&0 \\ 0&0&1&1&0&1&1&1&0&1&1&0&0&0&0 \\ 0&0&0&1&1&0&1&1&1&0&1&1&0&0&0 \\ 0&0&0&0&1&1&0&1&1&1&0&1&1&0&0 \\ 0&0&0&0&0&1&1&0&1&1&1&0&1&1&0 \\ 0&0&0&0&0&0&1&1&0&1&1&1&0&1&1 \end{bmatrix}$$

As expected, they are equal.

Likewise, the systematic generating matrix, *genmatsys*, can be computed using *cyclgen* without the conditional parameter (nonsys):

```
>> [parmat,genmatsys]=cyclgen(15,genpoly)

genmatsys =

 Columns 1 through 12

   1   1   0   1   1   1   0   1   1   0   0   0
   1   0   1   1   0   0   1   1   0   1   0   0
   1   0   0   0   0   1   0   0   0   0   1   0
   0   1   0   0   0   0   1   0   0   0   0   1
   0   0   1   0   0   0   0   1   0   0   0   0
   1   1   0   0   1   1   0   1   0   0   0   0
   1   0   1   1   1   0   1   1   0   0   0   0

 Columns 13 through 15

   0   0   0
   0   0   0
   0   0   0
   0   0   0
   1   0   0
   0   1   0
   0   0   1
```

Putting *genmatsys* into matrix form and comparing the result to the manipulated result yields:

$$genmatsys=\begin{bmatrix}1&1&0&1&1&1&0&1&1&0&0&0&0&0&0\\1&0&1&1&0&0&1&1&0&1&0&0&0&0&0\\1&0&0&0&0&1&0&0&0&0&1&0&0&0&0\\0&1&0&0&0&0&1&0&0&0&0&1&0&0&0\\0&0&1&0&0&0&0&1&0&0&0&0&1&0&0\\1&1&0&0&1&1&0&1&0&0&0&0&0&1&0\\1&0&1&1&1&0&1&1&0&0&0&0&0&0&1\end{bmatrix}=G_{sys}=\begin{bmatrix}1&1&0&1&1&1&0&1&1&0&0&0&0&0&0\\1&0&1&1&0&0&1&1&0&1&0&0&0&0&0\\1&0&0&0&0&1&0&0&0&0&1&0&0&0&0\\0&1&0&0&0&0&1&0&0&0&0&1&0&0&0\\0&0&1&0&0&0&0&1&0&0&0&0&1&0&0\\1&1&0&0&1&1&0&1&0&0&0&0&0&1&0\\1&0&1&1&1&0&1&1&0&0&0&0&0&0&1\end{bmatrix}$$

As expected, they are equal.

Another time-saving toolbox function for analyzing cyclic codes is *gfweight* that can be used to determine the minimum distance between code words. Knowing the minimum distance, D_{min}, between code words, $(D_{min} – 1)$ errors can be detected.[1] Likewise, we can correct $(D_{min} - 2)/2$ errors for D_{min} even and $(D_{min} - 1)/2$ errors for D_{min} odd.

With 128 (2^7) valid code words, trying to find the minimum weight is excessively laborious. But the Toolbox function, *gfweight* does it easily and quickly.

```
>> gfweight(genpoly,15)

ans =
        3
```

This implies that 2 errors can be detected, and 1 error can be corrected.

The *(15,7)* code produces 32,768 (2^{15}) possible code words of which 128 (2^7) are valid code words. Needless to say, picking the right 128 valid codes out of 32,768 possibilities is a daunting task. The toolbox provides an *encode* function that alleviates much of this mathematical anxiety.

```
>> data=(0:127);
>> c=de2bi(data,'left-msb');
>> valid_code=encode(c,15,7,'cyclic',genpoly)

valid_code =

0   0   0   0   0   0   0   0 ¦ 0   0   0   0   0   0   0
1   0   1   1   1   0   1   1 ¦ 0   0   0   0   0   0   1
1   1   0   0   1   1   0   1 ¦ 0   0   0   0   0   1   0
0   1   1   1   0   1   1   0 ¦ 0   0   0   0   0   1   1
0   0   1   0   0   0   0   1 ¦ 0   0   0   0   1   0   0
                                \_________________________/
                                       information
```

The preceding code generates 128 decimal characters called *data* using a standard MATLAB *colon* operator. The *de2bi* function converts the decimal numbers to binary numbers and aligns the binary numbers with the most significant bit to the left. The *encode* function converts the 128 binary numbers into a *(15,7)* cyclic code that has a generating polynomial *genpoly* that was previously defined (*110111011*). There are 128 valid code words (*valid_code)* of which the first five are shown above. Note that the original binary-coded information appears in the last seven bits of the valid code word.

CONCLUSION

Although the concepts covered in this presentation are relatively straightforward, the implementation becomes rapidly cumbersome as the length of the code word and the uncoded message increase. The discussion has shown that the functions contained in the MATLAB Communications Toolbox can handle the block coding functions regardless of the complexity of the system. Although the presentation looked at methods to convert from nonsystematic to systematic generating and parity matrices, there are many more applications (Reed Solomon codes, convolutional codes, Viterbi codes, etc.) where the Communication Toolbox reduces the complexity of matrix manipulations needed to perform common block coding functions such as: encoding or decoding a message, determining the error correcting capabilities of a code, finding the syndrome, and computing a decoding table. In addition, the inherent matrix manipulation capabilities of the standard MATLAB provide an invaluable tool to transpose and multiply matrices.

Although mathematical manipulation is important in the study of any engineering or engineering technology subject, the MATLAB Communications Toolbox provides one tool that can relieve the mathematical anxiety of the students, especially engineering technology students. By reducing the mathematical anxiety, students can concentrate more attentively on the actual nuances of block coding, and the instructor can move from the theoretical foundation of block codes to the circuit implementation of block codes.

REFERENCES

1. The MathWorks, *Communications Toolbox User's Guide, v.2*. The MathWorks, Inc. Natick, MA. 2000.
2. Roden, Martin S., *Analog Digital Communication Systems*, *4th Ed.*, Discovery Press. Los Angeles, CA. 2001.
3. Kieffer, John C., *Homework Set 3, Problem #4*. Online. Available: http://www.ece.umn.edu/users/kieffer/5581/5581hmk3sol.pdf. Fall, 2000.

Wm. Hugh Blanton

Wm. Hugh Blanton received the B.S. Technology degree in electronic engineering technology from the University of Houston in 1971, the M.S. in math/physics education from West Texas State University in 1979, the MBA from West Texas State University in 1986, and the Ed.D. in educational leadership and policy analysis from East Tennessee State University in 1992. He has taught electronic engineering technology at various colleges and universities since 1974 as well as worked as a biomedical technologist at Baylor College of Medicine, as a consultant in wind energy at the Alternative Energy Institute, and as a research engineer in instrumentation at Southwest Research Institute. He is currently an assistant professor of electronic engineering technology at East Tennessee State University and is interested in applications of DSP, neural networks, and fuzzy logic to telecommunications and control systems.