SCHOOL OF ELECTRICAL ENGINEERING, UNIVERSITY OF THE WITSWATERSRAND

ELEN3015 - Data and Information Management

Lecturer: Ling Cheng

Lab 5 Linear Block Codes

30 April 2018

Tyson Cross 1239448

INTRODUCTION

Hamming introduced the first linear block codes[1] shortly after Shannon's seminal 1948 paper established the Shannnon limit in the newly-created field of Information Theory. Hamming codes are linear, forward error-correcting block codes[2]. Binary Hamming codes are generated with the use of primitive polynomials. Under a Galois field, there is a direct and single mapping between elements of the field and polynomials over the field. Entries for a Hamming Code can be considered as either polynomials over the field, or as members of the null space of a matrix generated by the primitive polynomial[3].

Hamming(k,n) encodes a k-bit message as a n-bit codeword, adding m parity bits (where $n = 2^m - 1$, and $k = n - m$). A codeword v is the matrix product of the input message u and a generator matrix G. After receiving a codeword (v'), error-detection is achieved by examining the *syndrome* of the received codeword, which is the matrix product of the recieved codeword and the transpose of the parity-check matrix H.

1 HAMMING(K.N) CODE

Binary row vectors are used to represent message sequences and parity bits, but message sequences can also be interpreted as coefficients to entries in a polynomial. Hamming(k,n) encoding takes place over the Galois field GF(2^m), which is constructed using an irreducible (primitive) polynomial $g(x)$. In the Hamming(7,4) code $g(x) = 1 + x + x^3$ is represented as `[1 1 0 1]`.

1.1 Encoding

Error-encoding is achieved by multiplying the message u with a generator matrix G to generate codeword v, as shown in equation 1:

$$\mathbf{v} = \mathbf{u}G \tag{1}$$

The generator matrix G wcan be created as shown in equation 2:

$$G := \begin{bmatrix} P \ I_k \end{bmatrix} \tag{2}$$

Each row of the parity matrix P is constructed m parity bits:

$$P = \begin{bmatrix} p_1 \ p_2 \ p_3 \\ \vdots \\ p_{1_k} \ p_{2_k} \ p_{3_k} \\ \end{bmatrix} \tag{3}$$

For example, the entries of the first row of the Hamming(7,4) parity matrix P can be constructed for using the 4 bits of the first row in identity matrix I_4 as follows:

$$p_1 = b_1 + b_2 + b_3$$
$$p_2 = b_2 + b_3 + b_4$$
$$p_3 = b_2 + b_3 + b_4$$

The parity-check matrix H can be constructed as shown in equation 4

$$H := \begin{bmatrix} I_{n-k} \ P^T \end{bmatrix} \tag{4}$$

This is not the exclusive method creating these matrices, and either generator matrix G or parity-check matrix H can be constructed first, with each matrix able to be created from the other. If G is constructed from H as in [4], the column ordering of H can vary.

1.2 Decoding

The encoded sequence (each codeword) is passed over a channel and then on the receiving side error-checked by the matrix multiplication of the received codeword, v' with the inverse of the parity-check matrix H. The error checking is performed as shown in equation 5:

$$syndrome = \mathbf{v}'H^T; \tag{5}$$

Because the codewords of a Hamming code can be considered as members of the null space of the generating matrix, mutiplying the message by the transpose of the parity-check matrix H should result in all zeros. If the resulting syndrome is not the zero vector $\mathbf{0}$, then a transmission error has occurred. Any single-bit error can be detected and corrected by comparing the syndrome to a corresponding row-entry in H. The row number is the bit position of the error.

2 Assignment 1: Encoding

2.1 *Question 1*

A code C_1 is generated using the specified generator matrix G from the handout[4]. Matrix G is shown in equation 6:

$$G = \left[\begin{array}{ccc|cccc} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \tag{6}$$

The input codewords (u) are constructed as all possible 4-bit binary values, shown in Table I.

TABLE I: Input messages u

1:	0	0	0	0
2:	0	0	0	1
3:	0	0	1	0
4:	0	0	1	1
5:	0	1	0	0
6:	0	1	0	1
7:	0	1	1	0
8:	0	1	1	1
9:	1	0	0	0
10:	1	0	0	1
11:	1	0	1	0
12:	1	0	1	1
13:	1	1	0	0
14:	1	1	0	1
15:	1	1	1	0
16:	1	1	1	1

Each row of u is multiplied by G, as shown in equation 7:

$$C_1 = \mathbf{u}G \tag{7}$$

The codewords comprising C_1 are shown in Table II, as a result of this operation.

TABLE II: Code C_1

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

Note that the first 3 bits in each row are the parity bits, and the last 4 bits are the original message. The MATLAB code to generate this code is shown in lines 1-39 in Code Listing 1 in Appendix I.

2.2 *Question 2*

Using primitive polynomial $g(x) = 1 + x + x^3$, code C_2 is constructed by multiplying each row of the same input u (all possible 4-bit binary sequences) with the binary representation of $g(x)$. Each input binary sequence of the message rows is considered as the representation of a polynomial. i.e. the message `9` in binary is `[1 0 0 1]` which is interpreted at $1 + x^3$. Using this example,

$$\begin{aligned}
&[1\ 0\ 0\ 1] \Rightarrow (1 + x^3) \\
&(1 + x^3)g(x) = (1 + x^3)(1 + x + x^3) \\
&= 1 + x + \cancel{2x^3} + x^4 + x^6 \\
&= 1 + x + x^4 + x^6 \\
&\Rightarrow [1\ 1\ 0\ 0\ 1\ 0\ 1]
\end{aligned}$$

If the result is less than 7-bits, then it is zero padded (with the absent higher polynomial powers added as zeros). This operation performed on all inputs of u results in code C_2 shown in Table III. The MATLAB code to generate C_2 is shown in lines 40-74 in Code Listing 1 in Appendix I.

TABLE III: Code C_2

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

2.3 *Question 3*

The comparison of C_1 and C_2 at first reveals limited matching of rows and entries. However, if the rows of C_2 are reordered, according to an ascending sorting of the first 4-bits of each row, then C_2 and C_1 can be usefully compared as follows: As can be seen in Table IV, C_1 and C_2 are essentially the same, with all input messages from u represented, and the structural placement of the parity bits either prepended or appended. The MATLAB code reordering and comparing the two codes is shown in lines 79-96 in Code Listing 1 in Appendix I.

2.4 *Question 4*

The systematic encoding of Hamming(7,4) is performed in MATLAB in a function

TABLE IV: Comparison of C_1 and row-reordered C_2

C_1	C_2
0 0 0 0 0 0 0	0 0 0 0 0 0 0
1 0 1 0 0 0 1	0 0 0 1 1 0 1
1 1 1 0 0 1 0	0 0 1 0 1 1 1
0 1 0 0 0 1 1	0 0 1 1 0 1 0
0 1 1 0 1 0 0	0 1 0 0 0 1 1
1 1 0 0 1 0 1	0 1 0 1 1 1 0
1 0 0 0 1 1 0	0 1 1 0 1 0 0
0 0 1 0 1 1 1	0 1 1 1 0 0 1
1 1 0 1 0 0 0	1 0 0 0 1 1 0
0 1 1 1 0 0 1	1 0 0 1 0 1 1
0 0 1 1 0 1 0	1 0 1 0 0 0 1
1 0 0 1 0 1 1	1 0 1 1 1 0 0
1 0 1 1 1 0 0	1 1 0 0 1 0 1
0 0 0 1 1 0 1	1 1 0 1 0 0 0
0 1 0 1 1 1 0	1 1 1 0 0 1 0
1 1 1 1 1 1 1	1 1 1 1 1 1 1

`systematicHamming()`, shown in Code Listing 5 in Appendix I. Encoding proceeds as follows:

First the input message (the rows of u) are pre-padded with zeros to be n-bits long. For Hamming(7,4) this is 7 bits long. Each row represents the binary value represented as a polynomial, $m(x)$ as discussed before. To create the parity bits, $p(x)$, each row polynomial is multiplied by x^{n-k}, and then divided by $g(x)$, as shown in equation 8. Note that $x^{n-k} = x^m$. All multiplication and division is closed under the primitive polynomial $g(x)$.

$$p(x) = \frac{m(x)x^m}{g(x)} \tag{8}$$

The resulting polynomial is of maximum order 3. This value is now post-padded with zeros to be n-bits long. The pre-padded message and post-padded parity bits are then added (XORed) to produce code C_3, as shown in Table V.

TABLE V: Code C_3

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

Note that the resulting code C_3 is identical to C_3, and uses the same convention of parity bits prepending the message bits. The MATLAB code calling the `systematicHamming()` function is shown in lines 97-118 in Code Listing 1 in Appendix I.

2.5 *Question 5*

Hamming distance measures the number of symbols (or bits) that differ, for each position in two compared messages. The minimum Hamming distance measures the minimum number of differences between each non-zero codeword, for each bit position. A shorter method to calculate the minimum Hamming distance is to XOR every combination of the rows in G and count the weight of each non-zero row. This code is implemented as a funtion `minHammingDistance()`, shown in Code Listing 7 in Appendix I.

There are specialised functions for most computing architectures to perform optimised population counts on non-zero bit values, but for code clarity and for the purposes of this laboratory, a general function capable of operating row- or column-wise is implemented, using a single 'for'-loop. The algorithm constructs an double index array from the binomial expansion of each row (or column) number of C_1 and 2. This index provides a listing all possible row combinations of the input array. Each of these row or column combinations are then XORed, and the sum of the number of 1s of the result is stored. This value is compared for all combinations, and the lowest sum is the resulting minimum Hamming distance. The calculated minimum Hamming distance of C_1 is calculated as $d_m in = 3$.

2.6 *Question 6*

The minimum Hamming weight of C_1 is calculated with the implemented function `minHammingWeight()`, shown in Code Listing 6 in Appendix I. This is implemented as the lowest sum of all non-zero bits for each codeword. This is the equivalent of the operation performed in Question 4. The resulting minimum Hamming weight is calculated as $w_m in = 3$.

2.7 *Question 7*

According to [5], the minimum Hamming distance is equivalent to the weight of the sum of any two codewords (which is equal to another possible codeword.) If all the weights of non-zero codewords are all least m, then the minimum Hamming distance between codewords will be $\geq m$. For Hamming(7,4), $d_m in = 3$ and $w_m in = 3$.

2.8 *Question 8*

The last three digits of number `1239448` were converted to 4-bit binary values and encoded using the generator matrix G shown in equation 2. The input message is shown in Table VI:

TABLE VI: Student Number input message

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

TABLE VII: Codewords c

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

When encoded, the resulting codewords c is shown in Table VII: The MATLAB code to encode the student number digits is shown in lines 155-171 in Code Listing 1 in Appendix I.

3 Assignment 2: Decoding

3.1 Question 1

Decoding for error-checking and single-bit error-correction is implemented as shown in lines 35-78 in Code Listing 2 in Appendix I.

A known set of 7 codewords are generated, each with a single bit error in each of the 7 bit positions, shown in Table VIII: This message with errors was previously

TABLE VIII: Received codewords, each with a single bit error

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

written to disk with `dlmwrite()`, and is read into the implemented code solution using `dlmread()`. Each row of this received message is called v', shown in equation 10.

$$\mathbf{v} = \mathbf{u}G \tag{9}$$

$$\mathbf{v} + \mathbf{e} = \mathbf{v}' \tag{10}$$

Each row of this received message, called v', is multiplied by the transpose of the parity-check matrix H to calculate the syndrome for each input message, as shown in equation 5. This operation is sufficient to decode the message as can be seen in equation 11:

$$\begin{aligned}(\mathbf{u}G + \mathbf{e})H^T &= \mathbf{v}H^T + \mathbf{e}H^T \\ &= \mathbf{0} + \mathbf{e}H^T \\ &= \mathbf{e}H^T\end{aligned} \tag{11}$$

If $v' \neq \mathbf{0}$ then the message has an error. To determine which bit in the message has an error, the three-bit syndrome is compared to the row entries in H. Because $\mathbf{e}$ is a row of zeros, except for a single error bit entry of 1, $\mathbf{e}H^T = h_i$, where i is the row entry of H^T. In other words, the matching row in H^T is the bit position of the error in the received message. Because the message is encoded in binary, correcting the error consists of flipping the bit to the opposite value ($0 \to 1$, and $1 \to 0$). The resulting corrected messages are written to disk using `dlmwrite()`. The corrected message values is shown in Table IX:

TABLE IX: Error-corrected recovered codewords

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

3.2 Question 2

Another method of detecting an single error in the received message is with division by the irreducible primitive polynomial $g(x)$. The received message is read into memory with `dlmread()`, and then the residue of the division under modulo $g(x)$ is found using MATLAB's `gfdeconv` function. The resulting remainder polynomial is post-padded with zeros to make it 3-bits long. This polynomial is now considered as a binary row vector, and if it it not the zero vector $\mathbf{0}$ there is an error. The parity bit value is now looked up in H^T as before to determine the error position, and the relevant bit is flipped to correct the message. Hamming(7,4) is only able to correct a single bit error in a message of 4-bits, so if more than one bit has an error, the original message is not recoverable. The calculation of the syndrome for the received sequences was implemented as shown in lines 81-125 in Code Listing 2 in Appendix I.

Conclusion

Hamming(7,4) encoding and decoding was implemented in MATLAB, for all assignment equations and exercises. The use of linear block codes for forward error-correction was demonstrated, and analysed with practical examples. The successful encoding and decoding of a sequence with a known single bit error in all possible positions was demonstrated. Hamming codes use mathematical matrix operations under a closed algebraic field and the linear properties of the null space of a matrix for the simple and elegant detection and correction of single bit errors in linear block decoding.

REFERENCES

[1] R. W. Hamming, “Error detecting and error correcting codes,” *Bell Labs Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[2] I. S. Reed and X. Chen, *Error-control coding for data networks*. Springer Science & Business Media, 2012, vol. 508.

[3] A. S. Barashko, “Polynomials generating hamming codes,” *Ukrainian Mathematical Journal*, vol. 45, no. 7, pp. 987–992, Jul 1993. [Online]. Available: https://doi.org/10.1007/BF01057445

[4] L. Cheng, “A short course on error control coding,” ELEN3015 Course Handout 2018, University of the Witswatersrand, October 2010.

[5] D. Kleitman, “Matrix Hamming Codes,” http://www-math.mit.edu/ djk/18.310/18.310F04/matrix_hamming_codes.html, 18.310 Discreet Mathematics Course Notes, 2007.

APPENDIX I

ASSIGNMENTS CODE

Code 1: Assignment_3_1.m

```
%% Question 1:
disp('Question 1:'); disp(' ');

% The matrices from the Handout equation (2.6)
H1 = [  1     0     0     1     0     1     1 ;...
        0     1     0     1     1     1     0 ;...
        0     0     1     0     1     1     1 ];

G1 = [  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 ];

Code = {G1,H1};

% input
u = binaryArray(4); % all possible 4-bit values
usize = size(u);

% C1
C1 = mod(u*G1,2);

% output
disp('C1:');
disp('---');
disp('H ='); disp(H1);
disp('G ='); disp(G1);
disp('Input (u) = ')
disp(u);  disp(' ');
disp('C1 = uG'); disp(' ');
disp('C1 = '); disp(C1); disp(' ');
disp('--------------------------------------------------------');disp(' ');
%{
Comment:
 In C1, the first three bits are the added parity bits,
 and the last four bits are the data message (u)
%}

%% Question 2:
disp('Question 2:'); disp(' ');

p = [1 1 0 1];       % irreducible polynomial: 1 + x + x^3

m = 3;               % number of parity bits
n = 2^m - 1;         % length of codeword
k = n - m;           % length of message
assert(k==usize(2)); % confirm that k is the message length

% parity check and generator matrices
disp([' g(x) = ', num2str(p) ]); disp(' ');
fprintf(' g(x) = '); gfpretty(p); disp(' ');

for i=1:length(u)
   C2(i,:) = zeropad(gfconv(u(i,:),p), n, 'after');
end

% output
disp('C2:');
disp('---');
disp('Input (u) = ');
disp(u);  disp(' ');
disp('C2 = u*g(x)'); disp(' ');
disp('C2 = '); disp(C2); disp(' ');
disp('C2 (rows sorted by message value) :');
% reorder the rows to match the input message order
```

```
[¬,sort_index] = sortrows(C2(:,1:k));
C2 = C2(sort_index,:);
disp(C2); disp (' ');
%{
Comment:
 In C2, the first four bits are the data message (u),
 and the last three bits are the added parity bits.
%}

%% Question 3:
disp('Question 3:');  disp(' ');

% output
disp('C1 = '); disp(C1); disp(' ');
disp('C2 (sorted) = '); disp(C2); disp(' ');
assert(isequal(C1(:,k:end),C2(:,1:k)) && isequal(C1(:,1:m),C2(:,k+1:end)));
if isequal(C1(:,k:end),C2(:,1:k)) && isequal(C1(:,1:m),C2(:,k+1:end))
    disp([  'C1 and C2 are essentially the same, containing the same information in a ...
        different structure.']);
    disp([  'Using the irreducible polynomial 1+x+x^3 to create the parity and generator ...
        matrices (G, H) ']);
    disp([  'results in the matrices shown in Handout equation (2.6). Encoding a 4-bit ...
        message through ']);
    disp([  'matrix multiplication with G produces a codeword with 3 parity bits, ...
        followed by the message.']);
    disp(' ')
    disp([  'In the generated codewords for C2, this order is reversed: the four bits ...
        are the original ']);
    disp([  'message (u) and the three bits are the parity bits. The row ordering of the ...
        codewords in C2 is ']);
    disp([  'not the same as C1, but each individual row of a message and corresponding ...
        parity-bits match.']);
else
    disp('C1 and C2 do not match')
end
disp(' ');
disp('--------------------------------------------------------');disp(' ');

%% Question 4:
disp('Question 4:');  disp(' ');

[C3,P_x] = systematicHamming(m,u,p);
assert(isequal(C1,C3));

% output
fprintf('g(x) = '); gfpretty (p); disp(' ');
disp(['g(x) as a binary row vector: ', num2str(p)]); disp(' '); disp(' ');
disp('Input m(x) = ')
disp(u);  disp(' ');
disp('P(x) = m(x)(X^(n-k))/g(x) = ');
disp(P_x);
disp(' *Note: P(x) and m(x) are zero-padded to the length of codeword (n)');
disp(' ');
disp('C3 =  m(x) + P(x)');
disp(' ');
disp('C3 = ');
disp(C3); disp(' ');
disp('--------------------------------------------------------');disp(' ');

%% Question 5:
disp('Question 5:');  disp(' ');

% all possible combinations of the basis vectors in G
d_min = minHammingDistance(Code{1});

% output
disp(['d_min of C1 = ', num2str(d_min)]); disp(' ')
disp('--------------------------------------------------------');disp(' ');

%% Question 6:
disp('Question 6:');  disp(' ');

u_nozero = nonZeroBinaryArray(k);
```

```
v = mod(u_nozero*G1,2);
% Calculate the weight of every nonzero codeword
w_min = minHammingWeight(v);

% output
disp('All possible non-zero codewords of C1 = ')
disp(v);
disp(['w_min of C1 = ', num2str(w_min)]); disp(' ')
disp('--------------------------------------------------------');disp(' ');

%% Question 7:
disp('Question 7:');  disp(' ');

% output
disp(' "It is sufficient to arrange it that the minimum weight of a code');
disp('word is 3 (or k)... because the Hamming distance between two code words, ');
disp('A and B, say, is the weight of their sum, which sum is another code word.');
disp('This means that if all weights of non-zero code words are at least');
disp('3 (or k) the minimum Hamming distance between code words will be ');
disp('at least 3 (or k)."');
disp('Source: www-math.mit.edu/¬djk/18.310/18.310F04/matrix_hamming_codes.html');
disp('--------------------------------------------------------');disp(' ');

%% Question 8:
disp('Question 8:');  disp(' ');

student_no = [1 2 3 9 4 4 8];
for i=1:3
    student_no_bin(i,:) = dec2binary(student_no(end-3+i),4);
end

c = mod(student_no_bin*G1,2);

% output
disp(['Student Number = ', num2str(student_no)]); disp(' ');
disp('Binary row vectors of the last three digits =  ');
disp(student_no_bin);
disp('Codewords C = ');
disp(c); disp(' ');
disp('--------------------------------------------------------');disp(' ');
```

Code 2: Assignment_3_2.m

```
% setup
filename1 = 'input_codewords_error.txt';
filename2 = 'input_codewords_corrected_mod.txt';
filename3 = 'input_codewords_corrected_remainder.txt';

p = [1 1 0 1];       % irreducible polynomial: 1 + x + x^3
m = 3;               % number of parity bits
n = 2^m - 1;         % length of codeword
k = n - m;           % length of message

H = parityMatrix(m,p);
G = generatorMatrix(H);
C = {G,H};

% encode
u = nonZeroBinaryArray(4);
u = u(1:n,:);                % only need n rows
v = mod(u*G,2);

% % create errors in all 7 bit positions
% v_err = v;
% for i=1:7
%     val = v_err(i,i);
%     if val==1
%         v_err(i,i) = 0;
%     elseif val==0
%         v_err(i,i) = 1;
%     else
```

```
%         error('not a valid binary array')
%     end
% end
% dlmwrite(filename1,v_err,' ');

%% Question 1:
disp('Question 1:'); disp(' ');

receieved_message = dlmread(filename1);
disp('Message recieved:');
disp(receieved_message);

disp('Checking for errors using transpose of the parity matrix'); disp(' ');

% check for errors
for i=1:n
    codeword = receieved_message(i,:);
    syndrome = mod(codeword*H',2);      % get the syndrome from the matrix product with H'
    if sum(syndrome)≠0
        fprintf(['Error detected in recieved message at row ', num2str(i)])
        [correction_possible,row_loc] = ismember(syndrome,H','rows');
        if ¬correction_possible
            disp('Error correction not possible')
        else
            fprintf(' in position %d \n', row_loc);
            val = codeword(row_loc);
            if val==1
                codeword(row_loc) = 0;
            elseif val==0
                codeword(row_loc) = 1;
            else
                error('Not a valid binary array')
            end
            receieved_message(i,:) = codeword;
        end
    end
end
disp(' ')

if isequal(v,receieved_message)
    disp('All errors corrected'); disp(' ');
    dlmwrite(filename2,receieved_message,' ');
    output_message = receieved_message;
    disp('Corrected message is:')
    disp(output_message)
else
    disp('Uncorrected errors!')
end

disp('--------------------------------------------------------');disp(' ');

%% Question 2:
disp('Question 2:'); disp(' ');

clear receieved_message;

receieved_message = dlmread(filename1);
disp('Message recieved:');
disp(receieved_message);

disp('Checking for errors using modulo polynomial division'); disp(' ');

for i=1:n
    codeword = receieved_message(i,:);
    [¬, rem] = gfdeconv(codeword,p);     % get the syndrome from the residue
    syndrome = zeropad(rem,m,'after');
    if sum(syndrome)≠0
        fprintf(['Error detected in recieved message at row ', num2str(i)])
        [correction_possible,row_loc] = ismember(syndrome,H','rows');
        if ¬correction_possible
            disp('Error correction not possible')
        else
            fprintf(' in position %d \n', row_loc);
```

```
            val = codeword(row_loc);
            if val==1
                codeword(row_loc) = 0;
            elseif val==0
                codeword(row_loc) = 1;
            else
                error('Not a valid binary array')
            end
            receieved_message(i,:) = codeword;
        end
    end
end

disp(' ')

if isequal(v,receieved_message)
    disp('All errors corrected'); disp(' ');
    dlmwrite(filename3,receieved_message,' ');
    output_message = receieved_message;
    disp('Corrected message is:')
    disp(output_message)
else
    disp('Uncorrected errors!')
end
```

Additional function code

Code 3: parityMatrix.m

```
function [H] = parityMatrix(m,p)
%parityMatrix() creates a parity matrix H of size (n,k) from m & polynomial p

if length(p)≠(m+1)
    error('Irreducible Polynomial incorrect length')
end

n = 2^m - 1;
val = (0:n-1)';
H = gftuple(val,p)';                % parity-check matrix from Galois field

end
```

Code 4: generatorMatrix.m

```
function [G] = generatorMatrix(H)
%generatorMatrix() creates a generator matrix G from the parity matrix H

[m,n] = size(H);
j = n-m;
I = eye(m);
I_j = eye(j);

if H(:, (j+1):n) == I
    G = [I_j H(:,1:j)'];
elseif H(:, 1:m) == I
    G = [H(:,m+1:n)' I_j];
end
end
```

Code 5: systematicHamming.m

```
function [C,P_x] = systematicHamming(m,u,p)
%systematicHamming() performs systematic encoding of the Hamming(7,4) code

usize = size(u);
x_m = order2bin(m);                 % x^(n-k) = x^m
n = 2^m - 1;
k = n - m;

% initialise arrays
M_x = zeros(usize(1),n);        % m(x)
P_x = zeros(usize(1),n);        % parity bits
C   = zeros(usize(1),n);        % Hamming(7,4)

for i=1:usize(1)
    M_x(i,:)   = zeropad(gfconv(u(i,:),x_m),n,'after');
    [¬,p_x] = gfdeconv(M_x(i,:),p);        % m(x)*x^3 / g(x)
    P_x(i,:) = zeropad(p_x,n,'after');
    C(i,:) = bitxor(P_x(i,:), M_x(i,:));
end

P_x = P_x(:,1:m);

end
```

Code 6: minHammingWeight.m

```
function [w_min] = minHammingWeight(C)
%minHammingWeight measures the minimum Hamming weight (sum of 1s in non-zero rows)

csize = size(C);
w_min = csize(2);
```

```
len = csize(1);

for i=1:len
    val = sum(C(i,:));
    if val<w_min
        w_min = val;
    end
end
end
```

Code 7: minHammingDistance.m

```
function [d_min] = minHammingDistance(C,dir)
%minHammingDistance measures the minimum Hamming distance in a code by
%comparing all combinations of each row (or column) XORed with each other
%row (or column)

if nargin < 2
    dir = 'r';
end

csize = size(C);
if strcmp(dir,'row') || strcmp(dir,'r')
    combinations = nchoosek([1:csize(1)],2);
    do_row_wise = true;
    d_min = csize(2);
elseif strcmp(dir,'column') || strcmp(dir,'col') || strcmp(dir,'c')
    combinations = nchoosek([1:csize(2)],2);
    do_row_wise = false;
    d_min = csize(1);
else
    error('Please specify row or column')
end

if do_row_wise
    for i=1:length(combinations)
        a = C(combinations(i,1),:);
        b = C(combinations(i,2),:);
        val = sum(bitxor(a,b));
        if val<d_min
            d_min = val;
        end
    end
else
    for i=1:length(combinations)
        a = C(:,combinations(i,1));
        b = C(:,combinations(i,2));
        val = sum(bitxor(a,b));
        if val<d_min
            d_min = val;
        end
    end
end
end
```

Code 8: dec2binary.m

```
function [ bin ] = dec2binary( decimal, num_of_bits )
%dec2binary() converts an unsigned decimal value to a fixed length binary value
% if the bitdepth is not specified, then an appropriate fixed length is
% chosen, or 64,32,16 or 8 bits. Accurate conversion from 8 to
% 64-bits is supported, for decimal values from 0 to intmax('uint64').

if decimal > intmax('uint64')
   error('Input cannot be greater than intmax(''utint64'')')
end

if nargin<2
    if decimal > intmax('uint32')
        num_of_bits = 64;
    elseif decimal > intmax('uint16')
        num_of_bits = 32;
    elseif decimal > intmax('uint8')
        num_of_bits = 16;
    else
        num_of_bits = 8;
    end
end

decimal = uint64(decimal);
value = uint64(num_of_bits-1:-1:0);                    % Array of exponents for binary ...
    entries
base = uint64(2).^value;                               % Decimal values for each bit

if decimal > sum(uint64(base), 'native')
   error('Not enough bits specified to represent decimal value')
end

bin = false(1,num_of_bits);                            % Initialise logical array
for i=1:num_of_bits
    if decimal ≥ base(i)                              % For each applicable column of 2^i
        decimal = decimal - base(i);                   % Reduce the value of decimal
        bin(i) = true;                                 % Set the binary bit
    end
end
end
```

Code 9: binaryArray.m

```
function [P] = binaryArray(m)
%binaryArray() creates an array of all integers of m-bits in binary

P = zeros(2^m-1,m);
for i=1:(2^m)
    P(i,:) = dec2binary(i-1,m);
end
end
```

Code 10: nonZeroBinaryArray.m

```
function [P] = nonZeroBinaryArray(m)
%nonZeroBinaryArray() creates a list of all non-zero integer of m-bits

P = zeros(2^m-1,m);
for i=1:(2^m-1)
    P(i,:) = dec2binary(i,m);
end
end
```

Code 11: order2bin.m

```
function [bin] = order2bin(m)
%order2bin returns a binary vector representation of x^m

bin = zeros(1,m+1);
bin(m+1) = 1;

end
```

Code 12: zeropad.m

```
function [padded_array] = zeropad(array_to_be_padded, len, pos)
%zeropad() pads an numeric array with zeros to be specific length

if nargin < 3
    pos = 'front';
end

if len<length(array_to_be_padded)
    warning('Padding not possible')
end

padded_array = [array_to_be_padded];

for i=1:len-length(array_to_be_padded)
    if strcmp(pos,'front') || strcmp(pos,'prefix') || strcmp(pos,'before')
        padded_array = [0 padded_array];
    elseif strcmp(pos,'behind') || strcmp(pos,'suffix') || strcmp(pos,'after')
        padded_array = [padded_array 0];
    else
        error('Padding position not recognised')
    end
end
end
```