ECE300 Project 3: LBCs and Coding Theory

Jonathan Lam, Sam Shersher, Arthur Skok, Hadassah Yanofsky

June 3, 2021

Contents

1 Introduction

1.1 On Modulation Schemes and Coding

Practical communications theory has long since evolved from analog-dominated communication schemes to the binary coding schemes that are useful for digital systems. In the previous project, we explored several digital modulation schemes, such as PSK and QAM, and we compared differential and non-differential schemes.

We studied these methods in the context of one-shot communication (i.e., one signal corresponding to one message) and the effect of noise on each scheme. Each scheme is assumed to be transmitted over communication channels with additive white Gaussian noise (AWGN), and we analyzed their performances as measured by bit error rate (BER).

Having studied digital modulation schemes (which allow us to transmit digital signals), we then have to figure out how to encode our data into those digital signals. That brings us to information theory (in particular coding theory), which allows us to quantize information and develop codes that allow us to quantify and correct errors.

1.2 Coding Methods

Among the number of coding schemes to be explored in coding theory, this report will examine **Linear Block Codes**, through the manipulation and simulation of code-words in MATLAB.

A (n, k) linear block code is a coding scheme which contains a collection of $M = 2^k$ binary sequences where each are of length n. Each of these sequences are a **Codeword** belonging to this block's **Codebook**.

Hamming Codes are a special class of linear block codes that achieve the maximum possible rates for LBCs with a minimum Hamming distance between any two keywords of three and given their value of n. In a Hamming code, for some integer $m \geq 3$ then the code has $n = 2^m - 1$ and $k = 2^m - m - 1$.

For linear codes, one can create a **Generator Matrix**, such that all the codewords in the code are linear combinations of the rows in the generator matrix. The generator matrix for a (n, k) LBC has dimensions $k \times n$.

In special cases, one may have a **Systematic Code**, where the first k columns of a generator matrix are a $k \times k$ identity matrix, and thus each codeword would begin with an identifier for the respective word it represents, i.e., a block code that has $k = 2$ would have its first two columns be:

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

1.3 Context of the Project

In this project, we deal with systematic codes. Two generator matrices with dimensions $(4, 8)$ and $(4, 12)$ are provided.

$$G_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$G_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Parity Check Matrices were created for each code. A parity check matrix is created directly from systematic generator matrix, i.e., a generator matrix of form:

$$G = \begin{bmatrix} I_k \mid P \end{bmatrix}$$

The matrix P formed from the bits to the right of the identity matrix are also called the parity check bits.

Using these two components, the resulting parity check matrix H would be an $n-k$ identity matrix, appended to the negative of the transpose of the parity check bits:

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

It should be noted that the correctness of a parity check matrix can be verified via matrix multiplication, if the following matrix equation holds true:

$$GH^T = \mathbf{0}_{k \times (n-k)}$$

The resulting zero product is attributed to the orthogonality between rows of the generator matrix, (each row being a codeword of the original code) to the rows of the parity check matrix.

The code being used to generate and utilize these structures are described in §Methods.

2 Methods

2.1 Galois Field 2 Arithmetic

We implemented Galois field 2 (henceforth denoted $\mathbb{F}_2$) addition and multiplication in MATLAB by defining separate functions for each. To implement this arithmetic, we used the `mod` function since the $\mathbb{F}_2$ addition and multiplication operations are equivalent to ordinary matrix operations over the field $\mathbb{R}$ modulus 2. Note that these functions can be used for scalar values (i.e., degenerate matrices). Our implementations are shown in Figure 1.

```
function sum = galois2_add(A, B)
    sum = mod(A + B, 2);
end
```

(a) Addition

```
function prod = galois2_multiply(A, B)
    prod = mod(A * B, 2);
end
```

(b) Multiplication

Figure 1: Implementing $\mathbb{F}_2$ arithmetic on MATLAB arrays (or scalars).

2.1.1 Alternative Methods for $\mathbb{F}_2$ Arithmetic

MATLAB has a built-in $\mathbb{F}_2$ type that can be created with `gf2()`. It appears to be a special wrapper around uint8 matrices that overloads several matrix operations, (most notably the addition and multiplication field operations) and only allows values of unity and zero. We were able to implement this for most of the basic operations, but failed when we had to deal with `NaN` values. We assume that this would perform better than our implementation, since it uses smaller datatypes and fixed-point (integral) calculations.

Using the understanding that operations modulo two (and only dealing with values of unity and zero) act similarly to bitwise operations, we also wondered if these would improve the performance of our simulation. These functions are shown in Figure 2. While these produced the same results as the implementation shown above, this actually performed roughly 10% slower. Our conjecture is that floating-point modulus is implemented as a native instruction (e.g., `fmod`), while bitwise operations are only defined on integral types (and thus require an extra FP-to-int and int-to-FP operation) for each $\mathbb{F}_2$ operation, which could be more expensive than the benefits that bitwise operations provide.

2.2 Simulating the Binary Symmetric Channel

The next step was to simulate corruption of a bitstring in order to simulate a binary symmetric channel. We used the `rand` function to create an array of random bit values Bernoulli-distributed between 0 and 1 with probability P_{err}, and used this to corrupt a provided bitstring (a $\mathbb{F}_2$ array). The function implementation is shown in Figure 3.

```
function sum = galois2_add_alt(A, B)
    sum = bitxor(A, B);
end
```

(a) Addition

```
function prod = galois2_multiply(A, B)
    prod = bitand(A * B, 1);
end
```

(b) Multiplication

Figure 2: Bitwise implementation of $\mathbb{F}_2$ arithmetic.

```
function corrupted = corrupt_bitstring(str, p_err)
    corrupted = galois2_add(str, rand(size(str)) < p_err);
end
```

Figure 3: Bit corruption with error probability `p_err`

2.3 Generating the Parity Check Matrix H

We wrote a function to find H for a given systematic code using the definition of H:

```
function H = parity_check(G)
    [k,n] = size(G);
    P = G(:, (k+1):end);
    I = eye(n - k);
    H = [P.' I];
end
```

Figure 4: Generating the parity check matrix H for a LBC given G.

Then, we found the truncated syndrome array by multiplying the first word in each row - just the word of errors - of the truncated standard array by H in Galois field 2.

2.4 Correctable and Detectable Errors

To determine the maximum number of correctable errors, it was necessary to first determine the minimum Hamming distance between any two codewords in the code. This was simple to do given that $d_{min} = w_{min}$ in a LBC (proof deferred to class notes or the textbook), and the minimum hamming weight can be found by finding the codeword with the fewest number of set bits (ones).

Using this function we found that the d_{min} are 3 and 5 for codes 1 and 2, respectively. Using the formula for the maximum correctable errors:

$$e_c = \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor$$

```
function min_dist = min_hamming_distance(code)
    min_dist = min(sum(code(2:end,:), 2));
end
```

Figure 5: Finding the minimum Hamming distance between two codewords of a code using the property that $d_{min} = w_{min}$.

and found the maximum correctable errors to be 1 and 2, respectively. If the number of errors exceeds this number for any code, the codeword is not correctable because there is no unique closest codeword. However, as long as the error does not transform the original codeword into another codeword, the error is detectable. In this case, the receiver will still know that there were at least as many bit errors as the minimum Hamming distance between the received bitstring and any other codeword.

2.5 Generating the Truncated Syndrome Array

Normally, the standard array is a $2^{n-k} \times 2^k$ matrix that contains all of the possible 2^k possible n-bit strings, with each row representing all possible errors resulting in a unique n-bit string that is not a codeword (except for the first row, in which the "error" is the n-bit zero string). We were only concerned with finding correctable errors, so we constructed a truncated standard array with only e_c bit errors for each code. In general, with a maximum of e_c correctable errors, this truncated standard array would have r rows (i.e., number of correctable errors, plus one if we include the no-error case), where r can be found by the following equation:

$$r = \sum_{i=0}^{e_c} \binom{n}{i}$$

In other words, we count up the number of ways zero errors can be performed (this is always the first row of the standard array), the number of ways one error can be performed, the (distinct) number of ways two errors can be performed, and so on until the number of correctable errors is reached. In our case, the first truncated standard array has $r = 1 + 8 = 9$ rows and the second truncated standard array has $r = 1 + 12 + 66 = 79$ rows. The ful standard arrays of the first and second codes have 16 and 512 rows, respectively.

We can obtain the truncated syndrome array from the truncated standard array by applying the parity check matrix to any element in each row of the array. This holds because of a theorem discussed in class that states that every row of the standard array (and therefore the truncated standard array) is a coset, so that the parity check matrix H applied to any word in the row results in the coset leader, called the syndrome. This means that a matrix can be turned a single column, which leads to a large memory saving.

2.6 Error Correction

To correct errors efficiently for each code, we stored a hashtable (for $O(1)$ lookups) that maps each syndrome to its corresponding error. Since each syndrome and error are multiple-valued (matrices. which are not hashable by default), we first "hash" these by taking the decimal representation of both the syndromes and the errors. When the syndrome of a received word is found in this hashtable, then it be corrected by adding it to the received code (since addition is the same as subtraction in $\mathbb{F}_2$ arithmetic). Conversely, if it is not found, the error was not correctable since the truncated standard array contains all correctable errors. To signify this we set all the bits in the estimated received word to `NaN`. The implementation for the error correction is shown in Figure 6.

```
% correct_errors: returns corrected codeword if correctable,
% otherwise NaN
%
% params:
% SE_map= container.Map instance mapping syndromes to errors
% H     = parity check matrix for the code
% X     = received word ((# words) x N)
% returns:
% Xhat  = estimated codeword if correctable, else NaN
function Xhat = correct_errors(SE_map, H, X)
    % hashmap can only be accessed linearly (not vectorizable),
    % so this has to be implemented in a loop
    Xhat = zeros(size(X));
    for i = 1:size(X, 1)
        x = X(i, :);
        try
            Xhat(i,:) = galois2_add(x, ...
                de2bi(SE_map(bi2de(...
                    galois2_multiply(x, H.'))), length(x)));
        catch
            % if not found in SE_map
            Xhat(i,:) = NaN * ones(size(x));
        end
    end
end
```

Figure 6: Correcting correctable errors in the codeword. Note that we allow arbitrary matrices (in which each row is a codeword) as the input X, for the convenience of the caller.

3 Discussion

3.1 Bit Error Rate Comparison

This project explored two different error correction schemes. The first code is a $(8, 4)$ LBC and the second one is a $(12, 4)$ LBC. Both codes are systematic codes. For each code the bit error rate was calculated over a range of transmission bit error probabilities (ranging from 0.001 to 0.1).

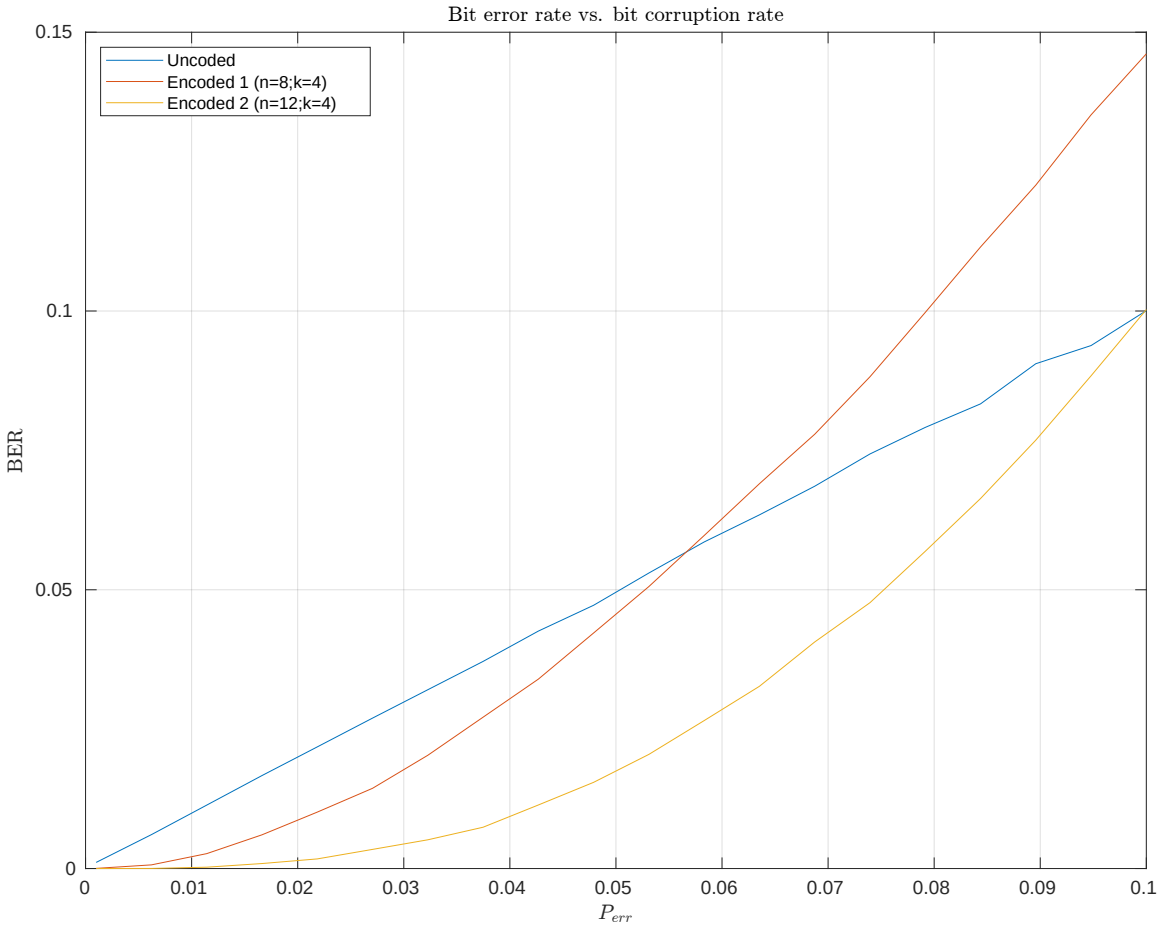

Figure 7: Error corrections schemes probability of error and corruption rates

We plot the different BERs over the range of transmission bit error probabilities for the two different schemes in Figure 7. Clearly the second scheme is better than the first scheme. The first scheme noticeably has larger bit rate error for the range of probabilities of error given, which makes sense given its ability to correct one more bit errors per codeword (the classic redundancy-error tradeoff). The probability of n errors occurring in the binary symmetric channel

is P_{err}^n, and thus even with the highest P_{err} tested, a non-correctable word is 10 times more likely to arise in the first code than in the second.

Note that the bit error rate (BER) using the first code actually overtakes the BER without an error correction scheme at around $P_{err} = 0.055$, meaning that in channels noisier than this, using no code seems be more reliable than using the first code. While this seems confusing, the reason for this is that our analysis between the encoded and uncoded schemes is different: when a detectable (but uncorrectable) error is found for the encoding scheme, then the entire word is thrown out (resulting in n errors); however, for the uncoded scheme, a single bit error is not regarded as the loss of an entire word; we only count the bits that are wrong.

The difference is that the encoded scheme (very likely) detects when a error has occurred in a codeword, and thus marks the entire word as wrong (and will most likely request re-transmission), while the uncoded scheme cannot tell when a code is incorrect without the use of some higher-level error checking scheme (e.g., checksumming). In other words, the "bit error rate" shown here for the encoded schemes is the number of words with a detected error divided by the total number of transmitted words, whereas the "bit error rate" shown here for the uncoded scheme is the total number of bit errors (which cannot be detected by the receiver) divided by the total number of transmitted bits. Clearly there is a discrepancy here in the interpretation (which comes from the phrasing of the question in the assignment). If we were to interpret the uncoded scheme in the former way, then no errors would ever be detected even when there are errors proportional to (roughly equal to) the transmission bit error; conversely, if we were to interpret the encoded scheme in the latter way, the bit error rate would be lower than the transmission bit error due to the fact that it has the ability to correct some errors. To summarize, the percentage of incorrect words should be lower for both encoded schemes than for the uncoded scheme.

3.2 The Truncated Syndrome Array

A truncated syndrome array was created in this project rather than the full syndrome array. This consists of the syndromes of all errors containing at most e_c bit errors. Errors with more than e_c map codewords to bitstrings that lie equidistant (in the Hamming sense) to at least two other codewords, or map codewords to other codewords; there is no way to choose the most reasonable correction, so we toss these words.

What we do with tossed words is application-dependent. For situations requiring high reliability, such as file transfer or any secure information transfer, it would make sense for a client (receiver) to re-request the corrupted words until it gets a word without corruption (and even then, it should check for non-detectable errors using other methods like checksums). For low-reliability situations, especially non-critical and low-latency (e.g., real-time in the consumer sense but not in the RTOS sense) applications like streaming, the tossed words may be discarded or re-requested depending on the communication channel bandwidth and rate, desired quality of communication, desired latency, etc.

4 Appendix: Source code

The code to generate the standard arrays and syndrome arrays for codes 1 and 2 is shown in Figure 8.

The code that drives the simulation and evaluation of the bit error rate after transmission for the uncoded and encoded schemes is shown in Figure 9. The standard output of this script, which comprises logging information about the probabilities of errors calculated and the elapsed time taken, is shown in Figure 10. The given output comes from running the simulation script on an i7-2600 CPU. It is evident from these slow output times ($\approx$ 400s for 100,000 codewords, a small number these days considering modern network speeds in the hundreds of megabits per second) that a CPU is not an optimal encoder and decoder for these schemes; it is probably much faster to use dedicated hardware with hardcoded logic and higher parallelization.

The source code for this project can also be found at `https://github.com/jlam55555/ece300-proj3`.

```
%% This file generates the syndrome arrays S1 and S2 (corresponding to
% codes c1, c2)
clc; clear; close all;

% sac = standard array correctable
load('code.mat');

%% code 1
corr1 = 1;
k1 = 4;
n1 = 8;
sac_1 = zeros(8+1, 2^k1, n1);

% first row of standard array is no errors
sac_1(1, :, :) = c1;

I = eye(n1);

for i = 1:n1
    sac_1(i+1, :, :) = galois2_add(c1, I(i, :));
end

% create
E1 = reshape(sac_1(:, 1, :), [], n1);
S1 = galois2_multiply(E1, parity_check(G1).');
SE_map1 = containers.Map(bi2de(S1), bi2de(E1));

%% code 2
corr2 = 2;
k2 = 4;
n2 = 12;
sac_2 = zeros(1+12+12*11/2, 2^k2, n2);

% 12
I = eye(n2);

% first row of standard array is no errors
sac_2(1, :, :) = c2;

counter = 2;

for i = 1:n2
    sac_2(counter, :, :) = galois2_add(c2, I(i, :));
    counter = counter + 1;
end

for i = 2:n2
    mat = I(i, :);
    for j = 1:i-1
        sac_2(counter, :, :) = galois2_add(c2, mat + I(j, :));
        counter = counter + 1;
    end
end

E2 = reshape(sac_2(:, 1, :), [], n2);
S2 = galois2_multiply(E2, parity_check(G2).');
SE_map2 = containers.Map(bi2de(S2), bi2de(E2));

%% save to .mat file
save('syndrome.mat', 'E1', 'S1', 'SE_map1', 'E2', 'S2', 'SE_map2');
```

Figure 8: Script to generate the standard and syndrome arrays

```
%% Q8: Evaluate the performance of the error correction
clc; clear; close all;
set(0, 'defaultTextInterpreter', 'latex');

load('code.mat');       % c1, c2, G1, G2
load('syndrome.mat');   % S1, S2, E1, E2, SE_map1, SE_map2

[K1, N1] = size(G1);
[K2, N2] = size(G2);

H1 = parity_check(G1);
H2 = parity_check(G2);

%% randomly generate 10^5 words for each scheme & encode
word_count = 1e5;

uncoded1 = de2bi(randi(2^K1-1, word_count, 1), K1);
uncoded2 = de2bi(randi(2^K2-1, word_count, 1), K2);

encoded1 = galois2_multiply(uncoded1, G1);
encoded2 = galois2_multiply(uncoded2, G2);

%% randomly corrupt bits with variable error probability
p_errs = linspace(1e-3, 1e-1, 20);

% results matrix
bers = zeros(3, length(p_errs));

tic();
for i = 1:length(p_errs)
    % note: this takes a while.....
    p_err = p_errs(i);
    fprintf('(%fs) Simulating p_err=%f\n', toc(), p_err);
    bers(:, i) = [
        ber(corrupt_bitstring(uncoded1, p_err), uncoded1); ...
        ber(correct_errors(SE_map1, H1, ...
            corrupt_bitstring(encoded1, p_err)), encoded1); ...
        ber(correct_errors(SE_map2, H2, ...
            corrupt_bitstring(encoded2, p_err)), encoded2)
    ];
end

%% plot results

uncoded_bers = bers(1, :);
encoded1_bers = bers(2, :);
encoded2_bers = bers(3, :);

figure('visible', 'off', 'Position', [0 0 1000 750]);
plot(p_errs, uncoded_bers(:), ...
    p_errs, encoded1_bers(:), ...
    p_errs, encoded2_bers(:));
grid on;
title('Bit error rate vs. bit corruption rate');
xlabel('$P_{err}$');
ylabel('BER');
legend(["Uncoded", "Encoded 1 (n=8;k=4)", "Encoded 2 (n=12;k=4)"], ...
    'Location', 'northwest');
exportgraphics(gca(), 'ber_vs_bcr.pdf');
```

Figure 9: Driver script for evaluating and plotting the BER

```
(0.000116s) Simulating p_err=0.001000
(19.434827s) Simulating p_err=0.006211
(38.710674s) Simulating p_err=0.011421
(57.417325s) Simulating p_err=0.016632
(76.253500s) Simulating p_err=0.021842
(96.177417s) Simulating p_err=0.027053
(115.950588s) Simulating p_err=0.032263
(135.571747s) Simulating p_err=0.037474
(155.492180s) Simulating p_err=0.042684
(175.412776s) Simulating p_err=0.047895
(195.985129s) Simulating p_err=0.053105
(216.431429s) Simulating p_err=0.058316
(236.948146s) Simulating p_err=0.063526
(257.570973s) Simulating p_err=0.068737
(278.481355s) Simulating p_err=0.073947
(299.737082s) Simulating p_err=0.079158
(322.724341s) Simulating p_err=0.084368
(345.474427s) Simulating p_err=0.089579
(368.047150s) Simulating p_err=0.094789
(391.268665s) Simulating p_err=0.1000
```

Figure 10: Simulation script standard output