CEL 51, DCCN, Monsoon 2020
Lab 5: Error Detection and Correction

Objective:
Implement the main building blocks of error detection and correction to handle bit errors. Error correction using a linear block code (rectangular parity).

Theory:
Linear block codes are examples of algebraic block codes, which take the set of k-bit messages we wish to send (there are 2k of them) and produce a set of 2k code words, each n bits long (n ≥ k) using algebraic operations over the block. The word "block" refers to the fact that any long bit stream can be broken up into k-bit blocks, which are then expanded to produce n-bit code words that are sent.

Such codes are also called (n, k) codes, where k message bits are combined to produce n code bits (so each code word has n − k "redundancy" bits). Often, we use the notation (n, k, d), where d refers to the minimum Hamming distance of the block code. The rate of a block code is defined as k/n; the larger the rate, the less the overhead incurred by the code. A linear code (whether a block code or not) produces code words from message bits by restricting the algebraic operations to linear functions over the message bits. By linear, we mean that any given bit in a valid code word is computed as the weighted sum of one or more original message bits. Linear codes, as we will see, are both powerful and efficient to implement. They are widely used in practice. In fact, all the codes we will study—including convolutional codes—are linear, as are most of the codes widely used in practice.

To develop a little bit of intuition about the linear operations, let's start with a "hat" puzzle, which might at first seem unrelated to coding.

There are N people in a room, each wearing a hat colored red or blue, standing in a line in order of increasing height. Each person can see only the hats of the people in front, and does not know the color of his or her own hat. They play a game as a team, whose rules are simple. Each person gets to say one word: "red" or "blue". If the word they say correctly guesses the color of their hat, the team gets 1 point; if they guess wrong, 0 points. Before the game begins, they can get together to agree on a protocol (i.e., what word they will say under what conditions). Once they determine the protocol, they stop talking, form the line, and are given their hats at random. Can you think of a protocol that will maximize their score? What score does your protocol achieve?

A little bit of thought will show that there is a way to use the concept of parity to enable N − 1 of the people to correctly decode the colors of their hats. In general, the "parity" of a set of bits x1, x2,...,xn is simply equal to (x1 + x2 + ... + xn), where the addition is performed modulo 2 (it's the same as taking the exclusive OR of the bits). Even parity occurs when the sum is 0 (i.e., the number of 1's is even), while odd parity is when the sum is 1. Parity, or equivalently, arithmetic modulo 2, has a special name: algebra in a Galois Field of order 2, also denoted F2. A field must define rules for addition and

multiplication. Addition in F2 is as stated above: 0 + 0 = 1 + 1 = 0; 1 + 0 = 0 + 1 = 1. Multiplication is as usual: 0 · 0=0 · 1=1 · 0 = 0; 1 · 1=1. Our focus in 6.02 will be on linear codes over F2, but there are natural generalizations to fields of higher order (in particular, Reed Solomon codes, which are over Galois Fields of order 2q). A linear block code is characterized by the following rule (which is both a necessary and a sufficient condition for a code to be a linear block code):

A block code is said to be linear if, and only if, the sum of any two code words is another code word.

For example, the code defined by code words 000, 101, 011 is not a linear code, because 101 + 011 = 110 is not a code word. But if we add 110 to the set, we get a linear code because the sum of any two code words is another code word. The code 000, 101, 011, 110 has a minimum Hamming distance of 2 (that is, the smallest Hamming distance between any two code words in 2), and can be used to detect all single-bit errors that occur during the transmission of a code word. You can also verify that the minimum Hamming distance of this code is equal to the smallest number of 1's in a non-zero code word. In fact, that's a general property of all linear block codes, which we saw formally during the theory lecture. Refer such slides from EDMODO.

Rectangular Parity Single Error Correction (SEC) Code:
Let parity(w) equal the sum over F2 of all the bits in word w. We'll use · to indicate the concatenation (sequential joining) of two messages or a message and a bit. For any message (sequence of one or more bits), let w = M · parity(M). You should be able to confirm that parity(w)=0. Parity lets us detect single errors because the set of code words w (each defined as M · parity(M)) has a Hamming distance of 2. If we transmit w when we want to send some message M, then the receiver can take the received word, r, and compute parity(r) to determine if a single error has occurred. The receiver's parity calculation returns 1 if an odd number of the bits in the received message have been corrupted. When the receiver's parity calculation returns a 1, we say there has been a parity error.

If we transmit w when we want to send some message M, then the receiver can take the received word, r, and compute parity(r) to determine if a single error has occurred. The receiver's parity calculation returns 1 if an odd number of the bits in the received message have been corrupted. When the receiver's parity calculation returns a 1, we say there has been a parity error.

This section describes a simple approach to building a SEC code by constructing multiple parity bits, each over various subsets of the message bits, and then using the resulting parity errors (or non-errors) to help pinpoint which bit was corrupted.

Rectangular code construction: Suppose we want to send a k-bit message M. Shape the k bits into a rectangular array with r rows and c columns, i.e., k = rc. For example, if k = 8, the array could be 2 × 4 or 4 × 2 (or even 8 × 1 or 1 × 8, though those are a little less interesting). Label each data bit with subscript giving its row and column: the first

bit would be d_{11}, the last bit d_{rc}.

d_{11}	d_{12}	d_{13}	d_{14}	p_row$(M,1)$
d_{21}	d_{22}	d_{23}	d_{24}	p_row$(M,2)$
p_col$(M,1)$	p_col$(M,2)$	p_col$(M,3)$	p_col$(M,4)$	

A 2 × 4 **arrangement for an 8-bit message with row and column parity.**

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

(a)

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

(b)

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

(c)

Example received 8-bit messages. Which have an error?

Define p row(i) to be the parity of all the bits in row i of the array and let R be all the row parity bits collected into a sequence:

R = [p row(1),p row(2), . . . ,p row(r)]

Similarly, define p col(j) to be the parity of all the bits in column j of the array and let C be the all the column parity bits collected into a sequence:

C = [p col(1),p col(2), . . . ,p col(c)]

Above Figure shows what we have in mind when k = 8.

Let w = M · R · C, i.e., the transmitted code word consists of the original message M,followed by the row parity bits R in row order, followed by the column parity bits C in column order. The length of w is n = rc + r + c. This code is linear because all the parity bits are linear functions of the message bits. The rate of the code is rc/(rc + r + c).We now prove that the rectangular parity code can correct all single-bit errors.

Proof of single-error correction property: This rectangular code is an SEC code for all values of r and c. We will show that it can correct all single bit errors by showing that its minimum Hamming distance is 3 (i.e., the Hamming distance between any two codewords is at least 3). Consider two different uncoded messages,Mi andMj . There are three cases to discuss:

- If Mi and Mj differ by a single bit, then the row and column parity calculations involving that bit will result in different values. Thus, the corresponding code words, wi and wj , will differ by three bits: the different data bit, the different row parity bit, and the different column parity bit. So in this case HD(wi,wj) = 3.
- If Mi and Mj differ by two bits, then either (1) the differing bits are in the same row, in which case the row parity calculation is unchanged but two column parity calculations will differ, (2) the differing bits are in the same column, in which case the column parity calculation is unchanged but two row parity calculations

will differ,or (3) the differing bits are in different rows and columns, in which case there will be two row and two column parity calculations that differ. So in this case HD(wi,wj) ≥ 4

- If Mi and Mj differ by three or more bits, then in this case HD(wi,wj) ≥ 3 because wi and wj contain Mi and Mj respectively. Hence we can conclude that HD(wi,wj) ≥ 3 and our simple "rectangular" code will beable to correct all single-bit errors.

Decoding the rectangular code: How can the receiver's decoder correctly deduce M from the received w, which may or may not have a single bit error? (If w has more thanone error, then the decoder does not have to produce a correct answer.)

Upon receiving a possibly corrupted w, the receiver checks the parity for the rows and columns by computing the sum of the appropriate data bits and the corresponding parity
bit (all arithmetic in F2). This sum will be 1 if there is a parity error. Then:

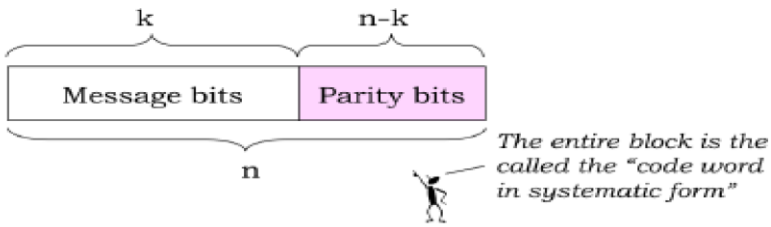
- If there are no parity errors, then there has not been a single error, so the receiver can use the data bits as-is for M. This situation is shown in above Figure (a).
- If there is single row or column parity error, then the corresponding parity bit is in error. But the data bits are okay and can be used as-is for M. This situation is shown in above Figure (c), which has a parity error only in the fourth column.
- If there is one row and one column parity error, then the data bit in that row and column has an error. The decoder repairs the error by flipping that data bit and then uses the repaired data bits for M. This situation is shown in Figure 6-4(b), where there are parity errors in the first row and fourth column indicating that d14 should be flipped to be a 0.
- Other combinations of row and column parity errors indicate that multiple errors have occurred. There's no "right" action the receiver can undertake because it doesn't have sufficient information to determine which bits are in error. A common approach is to use the data bits as-is for M. If they happen to be in error, that will be detected when validating by the error detection method.

This recipe will produce the most likely message, M, from the received code word if there has been at most a single transmission error.

In the rectangular code the number of parity bits grows at least as fast as √k (it should be easy to verify that the smallest number of parity bits occurs when the number of rows,
r, and the number of columns, c, are equal). Given a fixed amount of communication bandwidth, we're interested in devoting as much of it as possible to sending message bits, not parity bits. Are there other SEC codes that have better code rates than our simple rectangular code? A natural question to ask is: how little redundancy can we get away with and still manage to correct errors?

The Hamming code uses a clever construction that uses the intuition developed while answering the

How many parity bits are needed in a SEC code?
Let's think about what we're trying to accomplish with a SEC code: the goal is to correct transmissions with at most a single error. For a transmitted message of length n there are



A code word in systematic form for a block code. Any linear code can be transformed into an equivalent systematic code.

n+1 situations the receiver has to distinguish between: no errors and a single error in any of the n received bits. Then, depending on the detected situation, the receiver can make, if necessary, the appropriate correction.

Our first observation, which we will state here without proof, is that any linear code can be transformed into a **systematic** code. A systematic code is one where every n-bit code
word can be represented as the original k-bit message followed by the n − k parity bits (it actually doesn't matter how the original message bits and parity bits are interspersed).
Above Figure shows a code word in systematic form.

So, given a systematic code, how many parity bits do we absolutely need? We need to choose so that single error correction is possible. Since there are n − k parity bits, each combination of these bits must represent some error condition that we must be able to correct (or infer that there were no errors). There are 2^{n-k} possible distinct parity bit combinations, which means that we can distinguish at most that many error conditions. We therefore arrive at the constraint

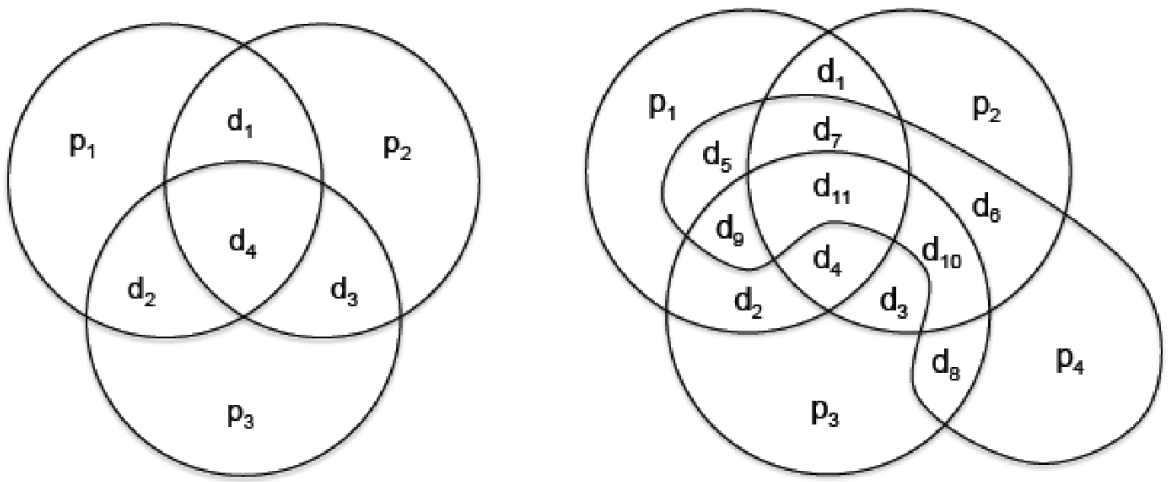
$$n+1 \leq 2^{n-k}$$

i.e., there have to be enough parity bits to distinguish all corrective actions that might need to be taken. Given k, we can determine the number of parity bits (n − k) needed to satisfy this constraint. Taking the log base 2 of both sides, we can see that the number of parity bits **must** grow at least logarithmically with the number of message bits. Not all codes achieve this minimum (e.g., the rectangular code doesn't), but the Hamming code, which we describe next, does.

Hamming Codes

Intuitively, it makes sense that for a code to be efficient, each parity bit should protect as many data bits as possible. By symmetry, we'd expect each parity bit to do the same amount of "work" in the sense that each parity bit would protect the same number of data bits. If some parity bit is shirking its duties, it's likely we'll need a larger number of parity bits in order to ensure that each possible single error will produce a unique combination of parity errors (it's the unique combinations that the receiver uses to deduce which bit, if any, had a single error).
The class of Hamming single error correcting codes is noteworthy because they are particularly efficient in the use of parity bits: the number of parity bits used by Hamming codes grows logarithmically with the size of the code word.

Below Figure shows two examples of the class: the (7,4) and (15,11) Hamming codes. The
(7,4) Hamming code uses 3 parity bits to protect 4 data bits; 3 of the 4 data bits are involved in each parity computation. The (15,11) Hamming code uses 4 parity bits to protect 11 data bits, and 7 of the 11 data bits are used in each parity computation (these properties will become apparent when we discuss the logic behind the construction of the Hamming code in Section)



(a) (7,4) code (b) (15,11) code

Venn diagrams of Hamming codes showing which data bits are protected by each parity bit.

.
Looking at the diagrams, which show the data bits involved in each parity computation,you should convince yourself that each possible single error (don't forget errors in one of the parity bits!) results in a unique combination of parity errors. Let's

work through the argument for the (7,4) Hamming code. Here are the parity-check computations performed by the receiver:

$$\begin{aligned} E_1 &= (d_1 + d_2 + d_4 + p_1) \mod 2 \\ E_2 &= (d_1 + d_3 + d_4 + p_2) \mod 2 \\ E_3 &= (d_2 + d_3 + d_4 + p_3) \mod 2 \end{aligned}$$

where each Ei is called a syndrome bit because it helps the receiver diagnose the "illness" (errors) in the received data. For each combination of syndrome bits, we can look for the bits in each code word that appear in all the Ei computations that produced 1; these bits are potential candidates for having an error since any of them could have caused the observed parity errors. Now eliminate from the candidates bits that appear in any Ei computations that produced 0 since those calculations prove those bits didn't have errors. We'll be left with either no bits (no errors occurred) or one bit (the bit with the single error). For example, if E1 = 1, E2 = 0 and E3 = 1, we notice that bits d2 and d4 both appear in the computations for E1 and E3. However, d4 appears in the computation for E2 and should be eliminated, leaving d2 as the sole candidate as the bit with the error.

Another example: suppose E1 = 1, E2 = 0 and E3 = 0. Any of the bits appearing in the computation for E1 could have caused the observed parity error. Eliminating those that appear in the computations for E2 and E3, we're left with p1, which must be the bit with the error.

Applying this reasoning to each possible combination of parity errors, we can make a table that shows the appropriate corrective action for each combination of the syndrome bits:

$E_3E_2E_1$	Corrective Action
000	no errors
001	p_1 has an error, flip to correct
010	p_2 has an error, flip to correct
011	d_1 has an error, flip to correct
100	p_3 has an error, flip to correct
101	d_2 has an error, flip to correct
110	d_3 has an error, flip to correct
111	d_4 has an error, flip to correct

Is There a Logic to the Hamming Code Construction?

So far so good, but the allocation of data bits to parity-bit computations may seem rather arbitrary and it's not clear how to build the corrective action table except by inspection.

The cleverness of Hamming codes is revealed if we order the data and parity bits in a certain way and assign each bit an index, starting with 1:

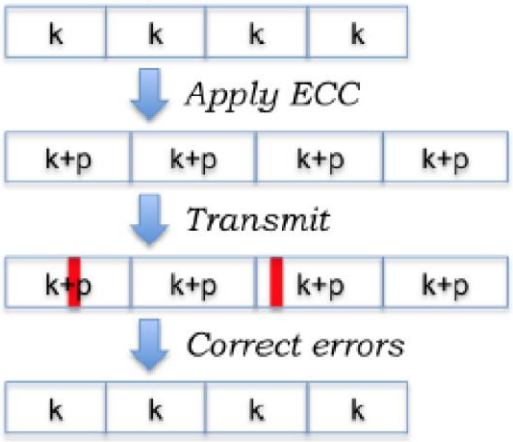
index	1	2	3	4	5	6	7
binary index	001	010	011	100	101	110	111
(7,4) code	p_1	p_2	d_1	p_3	d_2	d_3	d_4

This table was constructed by first allocating the parity bits to indices that are powers of two (e.g., 1, 2, 4, . . .). Then the data bits are allocated to the so-far unassigned indicies, starting with the smallest index. It's easy to see how to extend this construction to any number of data bits, remembering to add additional parity bits at indices that are a power of two.

Allocating the data bits to parity computations is accomplished by looking at their respective indices in the table above. Note that we're talking about the index in the table, not the subscript of the bit. Specifically, di is included in the computation of pj if (and only if) the logical AND of index(di) and index(pj) is non-zero. Put another way, di is included in the computation of pj if, and only if, index(pj) contributes to index(di) when writing the latter as sums of powers of 2.

So the computation of p1 (with an index of 1) includes all data bits with odd indices: d1, d2 and d4. And the computation of p2 (with an index of 2) includes d1, d3 and d4. Finally, the computation of p3 (with an index of 4) includes d2, d3 and d4. You should verify that these calculations match the Ei equations given above.

If the parity/syndrome computations are constructed this way, it turns out that E3E2E1, treated as a binary number, gives the index of the bit that should be corrected. For example, if E3E2E1 = 101, then we should correct the message bit with index 5, i.e., d2. This corrective action is exactly the one described in the earlier table we built by inspection.



Dividing a long message into multiple SEC-protected blocks of k **bits each, adding parity bits**
to each constituent block. The red vertical rectangles refer are bit errors.

The Hamming code's syndrome calculation and subsequent corrective action can be efficiently implemented using digital logic and so these codes are widely used in contexts, where single error correction needs to be fast, e.g., correction of memory errors when fetching data from DRAM.

Task #1: Rectangular parity SEC code

The intent in this task is to develop a decoder for the rectangular parity single error correction (SEC) code using the structure of the rectangular parity code to "triangulate" the location of the error, if any. Your job is to take a received codeword which consists of a data block organized into nrows rows and ncols columns, along with even parity bits for each row and column. The codeword is represented as a binary sequence (i.e., a list of 0's and 1's) in the following order:

```
[D(0,0), D(0,1), ..., D(0,ncols-1),        # data bits, row 0
 D(1,0), D(1,1), ...,                      # data bits, row 1
 ...
 D(nrows-1,0),   ..., D(nrows-1,ncols-1),  # data bits, last row
 R(0), ..., R(nrows-1),                    # row parity bits
 C(0), ..., C(ncols-1)]                    # column parity bits
```

In other words, all the data bits in row 0 (column 0 first), followed all the data bits in row 1, ..., followed by the row parity bits, followed by the column parity bits. The parity bits are chosen so that all the bits in any row or column (data and parity bits) will have an even number of 1's.

Define a (in Python/C/C++/java etc) function
rect_parity(*codeword*, *nrows*, *ncols*) as follows:
message_sequence = rect_parity(*codeword*,*nrows*,*ncols*) *codeword* is a binary sequence of length nrows*ncols + nrows + ncols whose elements are in the order described above.

The returned value *message_sequence* should have nrows*ncols binary elements consisting of the corrected data bits D(0,0), ..., D(nrows-1,ncols-1). If no correction is necessary, or if an uncorrectable error is detected, then return the raw data bits as they appeared in the codeword.

Do not use syndrome decoding in this task. (That will be the next programming task.) This goal of this task is to use the structure of the parity computations to pinpoint error locations. (This method is much faster than syndrome decoding, but is far less general and varies from code to code.)

PS2_tests.even_parity(*seq*) is a function that takes a binary sequence *seq* and returns True if the sequence contains an even number of 1's, otherwise it returns False. This parity check will be useful when performing the parity computations necessary to do error correction. PS2_rectparity.py is a template that you will extend by writing the function rect_parity.

The PS2_tests.test_correct_errors function will try a variety of test codewords and check for the correct results. If it finds an error, it'll tell you which codeword failed; if your code is working, it'll print out Testing all 2**n = 256 valid codewords...passed Testing all possible single-bit errors...passed(8,4) rectangular parity code successfully passed all 0,1 and 2 bit error tests

```
Testing all 2**n = 256 valid codewords
...passed
Testing all possible single-bit errors
...passed
(8,4) rectangular parity code successfully passed all 0,1 and 2 bit error tests
```

Code –

```
import numpy as np

def even_parity(seq):
    sum = None
    for d in seq:
        if sum == None:
            sum = d
        else:
            sum = sum ^ d

    return False if sum else True

def rect_parity(codeword, nrows, ncols):

    data_bits_len = nrows*ncols
    data_bits = codeword[:data_bits_len].reshape((nrows,ncols))

    R = codeword[data_bits_len:data_bits_len+nrows].tolist()
    C = codeword[data_bits_len+nrows:data_bits_len+nrows+ncols].tolist()

    print('Databits: ')
    print(data_bits)

    row_sum = []
```

```
    for row in data_bits:
        if even_parity(row):
            row_sum.append(0)
        else:
            row_sum.append(1)

    column_sum = []
    for row in data_bits.transpose():
        if even_parity(row):
            column_sum.append(0)
        else:
            column_sum.append(1)

    print('Row Sum: ',row_sum)
    print('Column Sum: ',column_sum)

    print('Row parity bits: ',R)
    print('Column parity bits: ',C)

    row_error = 0
    column_error = 0
    row_index = 0
    column_index = 0
    for i in range(nrows):
        if row_sum[i] != R[i]:
            print("Parity error in row {}".format(i+1))
            row_error += 1
            row_index = i

    for i in range(ncols):
        if column_sum[i] != C[i]:
            print("Parity error in column {}".format(i+1))
            column_error += 1
            column_index = i

    message_sequence = data_bits

    if row_error == 1 and column_error == 1:
          message_sequence[row_index,column_index] =
not(data_bits[row_index,column_index])
    elif row_error == 0 and column_error == 0:
          print('No correction is necessary')
    elif row_error == 1 or column_error == 1:
          print('Error detected in parity bits, hence no change in data bits')
    else:
          print('Uncorrectable error is detected')
```

```
    return message_sequence

def test_correct_errors():
    nrows = 2
    ncols = 4
    codeword1 = np.array( [0, 1, 1, 0, 1, 1, 0, 1,
                           0,1,
                           1, 0, 1, 1] )
    codeword2 = np.array( [1, 0, 0, 1, 0, 0, 1, 0,
                           1,1,
                           1, 0, 1, 0] )
    codeword3 = np.array( [0, 1, 1, 1, 1, 1, 1, 0,
                           1,1,
                           1, 0, 0, 0] )
    print('Codeword 1')
    message_sequence = rect_parity(codeword1,nrows,ncols)
    print('Message Sequence: ')
    print(message_sequence)
    print('\nCodeword 2')
    message_sequence = rect_parity(codeword2,nrows,ncols)
    print('Message Sequence: ')
    print(message_sequence)
    print('\nCodeword 3')
    message_sequence = rect_parity(codeword3,nrows,ncols)
    print('Message Sequence: ')
    print(message_sequence)


if __name__ == "__main__":
    test_correct_errors()
```

Output –

```
Command Prompt
Microsoft Windows [Version 10.0.18363.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\yashc>cd "OneDrive\Desktop\College Work\DCCN Lab\Experiment 5"

C:\Users\yashc\OneDrive\Desktop\College Work\DCCN Lab\Experiment 5>python task1.py
Codeword 1
Databits:
[[0 1 1 0]
 [1 1 0 1]]
Row Sum:  [0, 1]
Column Sum:  [1, 0, 1, 1]
Row parity bits:  [0, 1]
Column parity bits:  [1, 0, 1, 1]
No correction is necessary
Message Sequence:
[[0 1 1 0]
 [1 1 0 1]]

Codeword 2
Databits:
[[1 0 0 1]
 [0 0 1 0]]
Row Sum:  [0, 1]
Column Sum:  [1, 0, 1, 1]
Row parity bits:  [1, 1]
Column parity bits:  [1, 0, 1, 0]
Parity error in row 1
Parity error in column 4
Message Sequence:
[[1 0 0 0]
 [0 0 1 0]]
```

```
Command Prompt
 [1 1 0 1]]

Codeword 2
Databits:
[[1 0 0 1]
 [0 0 1 0]]
Row Sum:  [0, 1]
Column Sum:  [1, 0, 1, 1]
Row parity bits:  [1, 1]
Column parity bits:  [1, 0, 1, 0]
Parity error in row 1
Parity error in column 4
Message Sequence:
[[1 0 0 0]
 [0 0 1 0]]

Codeword 3
Databits:
[[0 1 1 1]
 [1 1 1 0]]
Row Sum:  [1, 1]
Column Sum:  [1, 0, 0, 1]
Row parity bits:  [1, 1]
Column parity bits:  [1, 0, 0, 0]
Parity error in column 4
Error detected in parity bits, hence no change in data bits
Message Sequence:
[[0 1 1 1]
 [1 1 1 0]]

C:\Users\yashc\OneDrive\Desktop\College Work\DCCN Lab\Experiment 5>
```

Task #2: Syndrome Decoding of Linear Block Codes

Write syndrome_decode, a function that takes the four arguments given below and returns an array of bits corresponding to the most likely transmitted message:

```
syndrome_decode(codeword, n, k, G)
```

codeword is an array of bits received at the decoder. The array has size n, which we explicitly specify as an argument to syndrome_decode for clarity. The original message is k bits long. G is the **generator matrix** of the code.

You may assume that G is a code that can correct all combinations of single bit errors. So your syndrome decoder needs to handle only that case. Of course, you may feel free to be ambitious and handle up to t bit errors, but our testing won't exercise those cases.

The following implementation notes may be useful:

1. Syndrome decoding uses matrix operations. You will very likely find it convenient to use matrix module (e.g Numpy in python),
2. All arithmetic should be modulo 2, in F_2. When you multiply two matrices whose elements are all 0 or 1, you may get numbers different from 0 or 1. Of course, one needs to replace each such number with its modulo-2 value. We have given you a function, mod2(A), which does that for an integer matrix, A.
3. We have also given you a simple function, equal(a, b), which returns True iff] two matrices a and b are equal element-by-element.
4. In this task, we are not expecting you to pre-compute the syndromes for the code, but to compute the syndromes in syndrome_decode just before decoding a codeword. It would be more efficient to pre-compute syndromes, but it would also mean that we need to specify the interface for a function like compute_syndromes, and for your software to adhere to that interface. In the interest of simplifying the interfaces and specifying only the behavior of syndrome_decode, we will take the small performance hit and just have you compute the syndromes every time you decode a codeword.

Code -

```
import numpy as np

#function to calculate mod to get binary matrix
def mod2(x):
    return np.mod(x,2)

#to compare matrix
def equal(a, b):
    return np.array_equal(a,b)

def syndrome_decode(codeword, n, k, G):
    #codeword is received message
```

```
    #calculate A
    A = G[0:k,k:n]
    print("Matrix A:")
    print(A)

    # calculate H
    At = A.transpose()
    I = np.identity(n-k,dtype=int)
    H = np.concatenate((At,I), axis=1)
    print("Matrix H:")
    print(H)

    #calculate c_d
    codewordT = codeword.transpose()
    c_d = mod2(H.dot(codewordT))   #c_d is syndrome for received message
    # print(c_d)

    for i in range(k):
        #generate syndromes
        P = np.array([0,0,0,0,0,0,0])
        P[i] = 1
        syndrome = H.dot(P.transpose())
        # print(syndrome)

        #check if syndrome is equal
        if equal(syndrome,c_d):
            codeword[i] = not(codeword[i])
            break
    return(codeword[:k])

if __name__ == "__main__":
    n=7
    k=4
    G = np.array([1,0,0,0,1,1,0,
                  0,1,0,0,1,0,1,
                  0,0,1,0,0,1,1,
                  0,0,0,1,1,1,1
                  ]).reshape(k,n)
    print("Generator Matrix:")
    print(G)
    codeword = np.array([1,1,1,0,1,0,1])   #error at pos 2
    message = syndrome_decode(codeword, n, k, G)
    # print(message)
    print("Received message:")
    print(''.join(map(str,message.tolist())))
```

Output -

```
Command Prompt
Microsoft Windows [Version 10.0.18362.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Yug Vajani>cd Desktop

C:\Users\Yug Vajani\Desktop>cd DCCN

C:\Users\Yug Vajani\Desktop\DCCN>python task2.py
Generator Matrix:
[[1 0 0 0 1 1 0]
 [0 1 0 0 1 0 1]
 [0 0 1 0 0 1 1]
 [0 0 0 1 1 1 1]]
Matrix A:
[[1 1 0]
 [1 0 1]
 [0 1 1]
 [1 1 1]]
Matrix H:
[[1 1 0 1 1 0 0]
 [1 0 1 1 0 1 0]
 [0 1 1 1 0 0 1]]
Received message:
1010
```

Task #3: Testing

We will test your code with a few different generator matrices. These tests will tell you whether your code works on that input; if not, it will print out the input on which the decoder failed (printing out what it produced and what was expected) and exit. We test all single-bit error patterns and the no-error case over the n-bit codeword. Hence you need to prepare two Test sequence of written error correction code systematically.

A successful test prints out lines similar to these:

```
Testing all 2**k = 16 valid codewords
...passed
Testing all n*2**k = 112 single-bit error codewords
...passed
All 0 and 1 error tests passed for (7,4,3) code with generator matrix G =
[[1 0 0 0 1 1 0]
 [0 1 0 0 1 0 1]
 [0 0 1 0 0 1 1]
 [0 0 0 1 1 1 1]]
```

A failed test might print out lines similar to these:

```
Testing all 2**k = 16 valid codewords
...passed
Testing all n*2**k = 112 single-bit error codewords
OOPS: Error decoding [1 0 0 0 0 0 0] ...expected [0 0 0 0] got [1 0 0 0]
```

When you're ready, please submit the file with your code.

Code –

```
from task2 import syndrome_decode, mod2, equal

import numpy as np


n = 7

k = 4

G = np.array([1,0,0,0,1,1,0,

              0,1,0,0,1,0,1,

              0,0,1,0,0,1,1,

              0,0,0,1,1,1,1

              ]).reshape(k,n)

# print(G)

print("Testing all 2**k = 16 valid codewords")


for number in range(2**k):

    message = np.array(list(bin(number)[2:].zfill(k)), dtype=int).reshape(1,k)

    # print(message.shape)

    codeword = mod2(message.dot(G))

    # print(codeword)

    decoded_message = syndrome_decode(codeword, n,k,G)

    # print(decoded_message)
```

```
    if not equal(codeword, decoded_message):
        print("Error decoding {} ... expected {} got
{}".format(codeword,message,decoded_message))
        break
else:
    print("...passed")


print("Testing all n*2**k = 112 single-bit error codewords")
for number in range(2**k):
    message = np.array(list(bin(number)[2:].zfill(k)),dtype=int).reshape(-1,k)
    # print(message)
    codeword = mod2(message.dot(G))
    # print(codeword)
    for i in range(len(codeword[0])):
        codeword[0][i] = not(codeword[0][i])
        # print(codeword)
        decoded_message = syndrome_decode(codeword, n,k,G)
        if not equal(codeword, decoded_message):
            print("Error decoding {} ... expected {} got
{}".format(codeword,message,decoded_message))
            break
        else:
            codeword[0][i] = not(codeword[0][i])
else:
    print("...passed")
print("All 0 and 1 error tests passed for (7,4,3) code with generator matrix G = \n",G)
```

Output –

```
Microsoft Windows [Version 10.0.18362.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

D:\Sarika\Sem 5\DCCN Lab\exp5>python task3.py
Testing all 2**k = 16 valid codewords
...passed
Testing all n*2**k = 112 single-bit error codewords
...passed
All 0 and 1 error tests passed for (7,4,3) code with generator matrix G =
 [[1 0 0 0 1 1 0]
 [0 1 0 0 1 0 1]
 [0 0 1 0 0 1 1]
 [0 0 0 1 1 1 1]]

D:\Sarika\Sem 5\DCCN Lab\exp5>
```

Deliverables:
Schematic view of encoder and decoder
Source code of Task #1, Task #2 and Task #3as per given function names
Test Sequence of successful test and failed test
Documentation.