

Group 5 :

Jiten Sidhpura - 2018130051

Trusha Talati - 2018130054

Rudresh Veerkhare - 2018130061

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 2^k of them) and produce a set of 2^k code words, each n bits long ($n \geq k$) 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

x_1, x_2, \dots, x_n is simply equal to $(x_1 + x_2 + \dots + x_n)$, 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 F_2 . A field must define rules for addition and multiplication. Addition in F_2 is as stated above: $0 + 0 = 1 + 1 = 0$; $1 + 0 = 0 + 1 = 1$. Multiplication is as usual: $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0$; $1 \cdot 1 = 1$. Our focus in 6.02 will be on linear codes over F_2 , but there are natural generalizations to fields of higher order (in particular, Reed Solomon codes, which are over Galois Fields of order 2^q). 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 to such slides from EDMODO.

Rectangular Parity Single Error Correction (SEC) Code:

Let $\text{parity}(w)$ equal the sum over F_2 of all the bits in word w . We’ll use \cdot 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 $w = M \cdot \text{parity}(M)$. You should be able to confirm that $\text{parity}(w) = 0$. Parity lets us detect single errors because the set of code words w (each defined as $M \cdot \text{parity}(M)$) 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 $\text{parity}(r)$ 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 $\text{parity}(r)$ 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	0	0	1	1	0	1	1	1	1
1	1	0	1	1	0	0	1	0	1	1	1	1	0	1
1	0	1	1		1	0	1	0		1	0	0	0	
(a)	(b)	(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:

$$R = [p_row(1), p_row(2), \dots, p_row(r)]$$

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:

$$C = [p_col(1), p_col(2), \dots, p_col(c)]$$

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

Let $w = MRC$, 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, M_i and M_j . There are three cases to discuss:

- If M_i and M_j 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, w_i and w_j , will differ by three bits: the different data bit, the different row parity bit, and the different column parity bit. So in this case $HD(w_i, w_j) = 3$.

- If M_i and M_j 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 $HD(w_i, w_j) \geq 4$
- If M_i and M_j differ by three or more bits, then in this case $HD(w_i, w_j) \geq 3$ because w_i and w_j contain M_i and M_j respectively. Hence we can conclude that $HD(w_i, w_j) \geq 3$ and our simple “rectangular” code will be able 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 than one 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 F_2). 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 d_{14} 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.

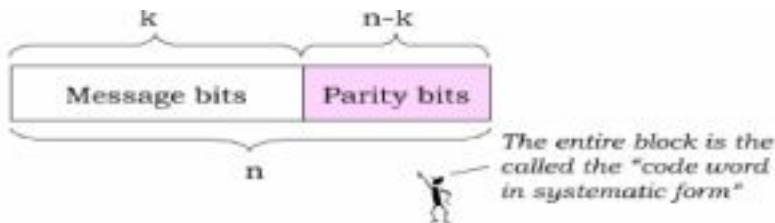
This recipe will produce the most likely message, M , from the received codeword if there has been at most a single transmission error.

In the rectangular code the number of parity bits grows at least as fast as \sqrt{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.

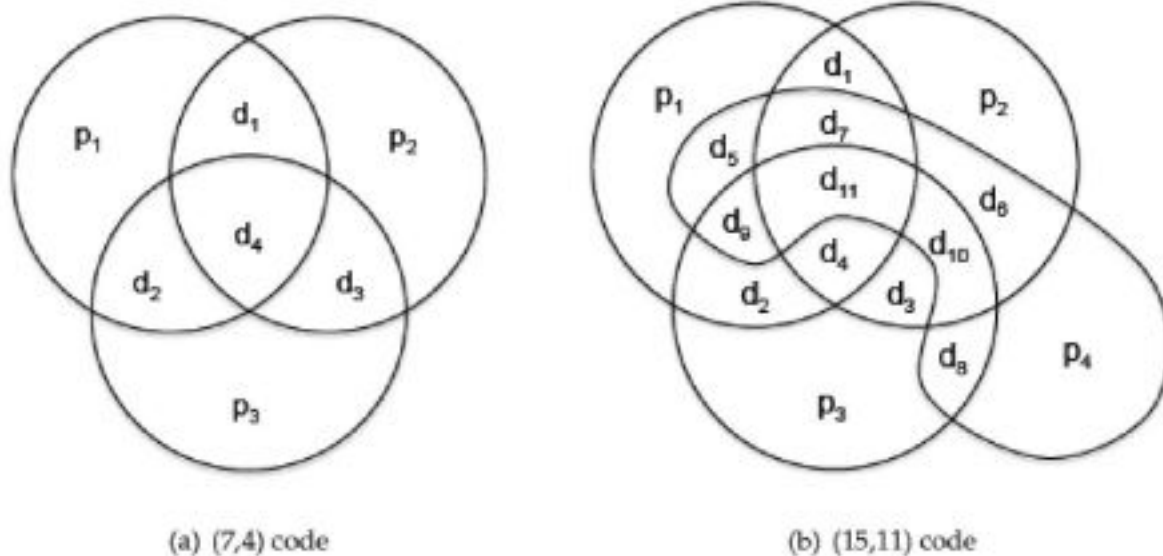
Hamming Codes

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)



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:

$$E_1 = (d_1 + d_2 + d_4 + p_1) \mod 2$$

$$E_2 = (d_1 + d_3 + d_4 + p_2) \mod 2$$

$$E_3 = (d_2 + d_3 + d_4 + p_3) \mod 2$$

where each E_i 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 E_i 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 E_i 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 $E_1 = 1$, $E_2 = 0$ and $E_3 = 1$, we notice that bits d_2 and d_4 both appear in the computations for E_1 and E_3 . However, d_4 appears in the computation for E_2 and should be eliminated, leaving d_2 as the sole candidate as the bit with the error.

Another example: suppose $E_1 = 1$, $E_2 = 0$ and $E_3 = 0$. Any of the bits appearing in the computation for E_1 could have caused the observed parity error. Eliminating those that appear in the computations for E_2 and E_3 , we're left with p_1 , 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:

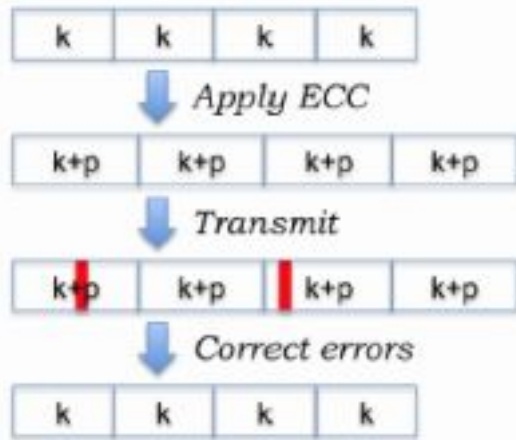
index	1	2	3	4	5	6	7
binary index	001	010	011	100	101	110	111
(7,4) code	p_1	p_2	d_1	p_3	d_2	d_3	d_4

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 indices, 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, d_i is included in the computation of p_j if (and only if) the logical AND of $\text{index}(d_i)$ and $\text{index}(p_j)$ is non-zero. Put another way, d_i is included in the computation of p_j if, and only if, $\text{index}(p_j)$ contributes to $\text{index}(d_i)$ when writing the latter as sums of powers of 2.

So the computation of p_1 (with an index of 1) includes all data bits with odd indices: d_1 , d_2 and d_4 . And the computation of p_2 (with an index of 2) includes d_1 , d_3 and d_4 . Finally, the computation of p_3 (with an index of 4) includes d_2 , d_3 and d_4 . You should verify that these calculations match the E_i equations given above.

If the parity/syndrome computations are constructed this way, it turns out that $E_3E_2E_1$, treated as a binary number, gives the index of the bit that should be corrected. For example, if $E_3E_2E_1 = 101$, then we should correct the message bit with index 5, i.e., d_2 . 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 to 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.

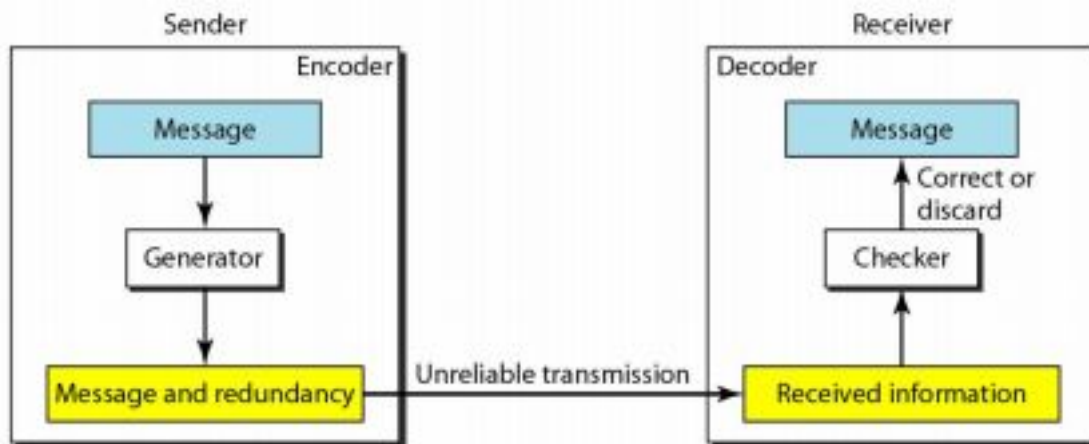
The returned value `message_sequence` should have `nrows*ncols` binary elements consisting of the corrected data bits $D(0,0), \dots, D(nrows-1,ncols-1)$. If no correction is necessary, or if an uncorrectable error is detected, then return the raw data bits as they appeared in the codeword.

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
```



Source Code for Task #1:

```
Exp 5 Task 1
Group 5
Jiten Sidhpura 57
Trusha Talati 60
Rudresh Veerkhare 66
```

```

import numpy as np
class DecodeRectParity:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        pass

    def check(self, codeword, verbose=False):
        """
        codeword - received message

        return type - tuple => dataBit, statusCode
        statusCode meaning
        0          No Error in bit detected
        -1         Error cannot be rectified
        1          Error detected and rectified
        """

        dataBits = codeword[:self.rows*self.cols]
        rowBits   =   codeword[self.rows*self.cols:
self.rows*self.cols+self.rows]
        colBits = codeword[self.rows*self.cols+self.rows: ]

        dataMatrix = dataBits.reshape((self.rows, self.cols))

        newRowBits = np.apply_along_axis(self.xor, 1, dataMatrix)
        newColBits = np.apply_along_axis(self.xor, 0, dataMatrix)

        if verbose:
            print(f>Data Matrix -\n{dataMatrix}")
            print(f>Row Xor received-\n{rowBits}")
            print(f>Row Xor calculated-\n{newRowBits}")
            print(f>Col Xor received-\n{colBits}")
            print(f>Col Xor calculated-\n{newColBits}")

        rowBitsCheck = np.where(rowBits != newRowBits)[0]

```

```

colBitsCheck = np.where(colBits != newColBits)[0]
statusCode = 0
if len(rowBitsCheck) and len(colBitsCheck):

    if len(rowBitsCheck) > 1 or len(colBitsCheck) > 1:
        # Error Cannot be rectified
        return None, -1
    statusCode = 1
    i, j = rowBitsCheck[0], colBitsCheck[0]
    if verbose:
        print(f"Bit flipped at row: {i+1}, col: {j+1}")
    dataBits[i*self.rows + j] = 0 if dataBits[i*self.rows + j] ==
1 else 0

    return dataBits, statusCode

@staticmethod
def xor(arr):
    ans = 0
    for i in arr:
        ans ^= i
    return ans

```

Function Return Type and Status Code:

```

return type - tuple => dataBit, statusCode
statusCode meaning

0          No Error in bit detected
-1         Error cannot be rectified
1          Error detected and rectified

```

Runner Code:

```
model.check(np.array([1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0] ),  
verbose=True)
```

Output:

Data Matrix -

```
[[1 0 0 1]
```

```
[0 0 1 0]]
```

Row Xor received-

```
[1 1]
```

Row Xor calculated-

```
[0 1]
```

Col Xor received-

```
[1 0 1 0]
```

Col Xor calculated-

```
[1 0 1 1]
```

Bit flipped at row: 1, col: 4

- Here, we have computed all possible permutations of length 8 and tested the rectangular SEC function against the codewords to detect/correct the code, and find the corrected 2x2 dataword.

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)
```

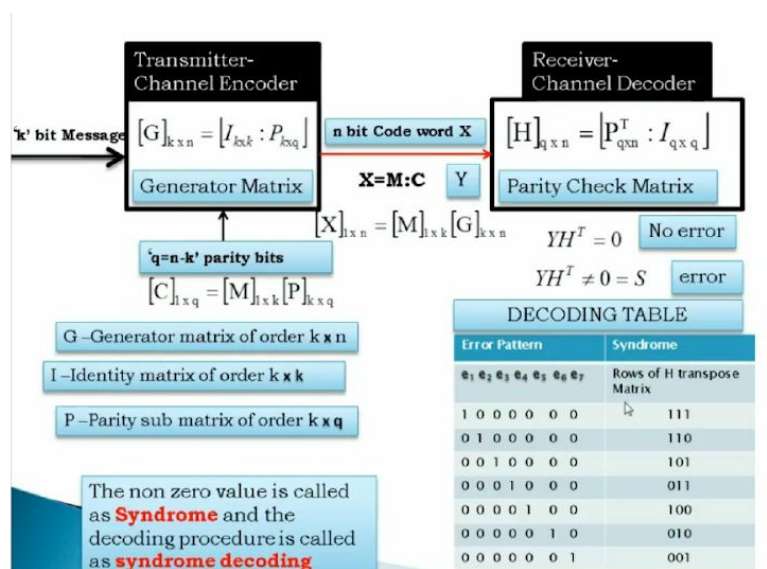
codeword is an array of bits received at the decoder. The array has size n , which we explicitly specify as an argument to `syndrome_decode` for clarity. The original message is k bits long. G is the **generator matrix** of the code.

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.

Encoder Decoder Schematic Diagram:



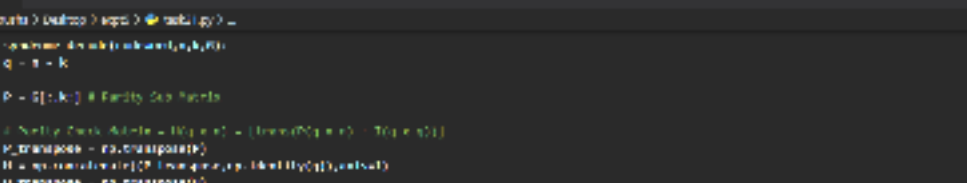
Source Code for Task #2:

The screenshot shows a Jupyter Notebook with the following code:

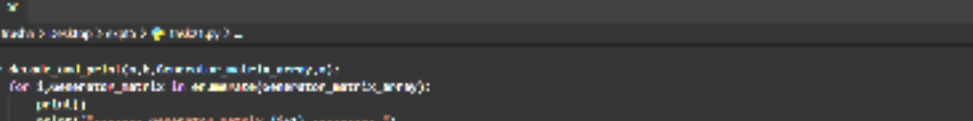
```

1  """
2  Page 4, Task 2
3  """
4  import numpy as np
5  from sklearn.metrics import mean_squared_error
6  """
7  """
8  import sys
9
10 def sigmoid(y):
11     return 1 / (1 + np.exp(-y))
12
13 """
14 """

```



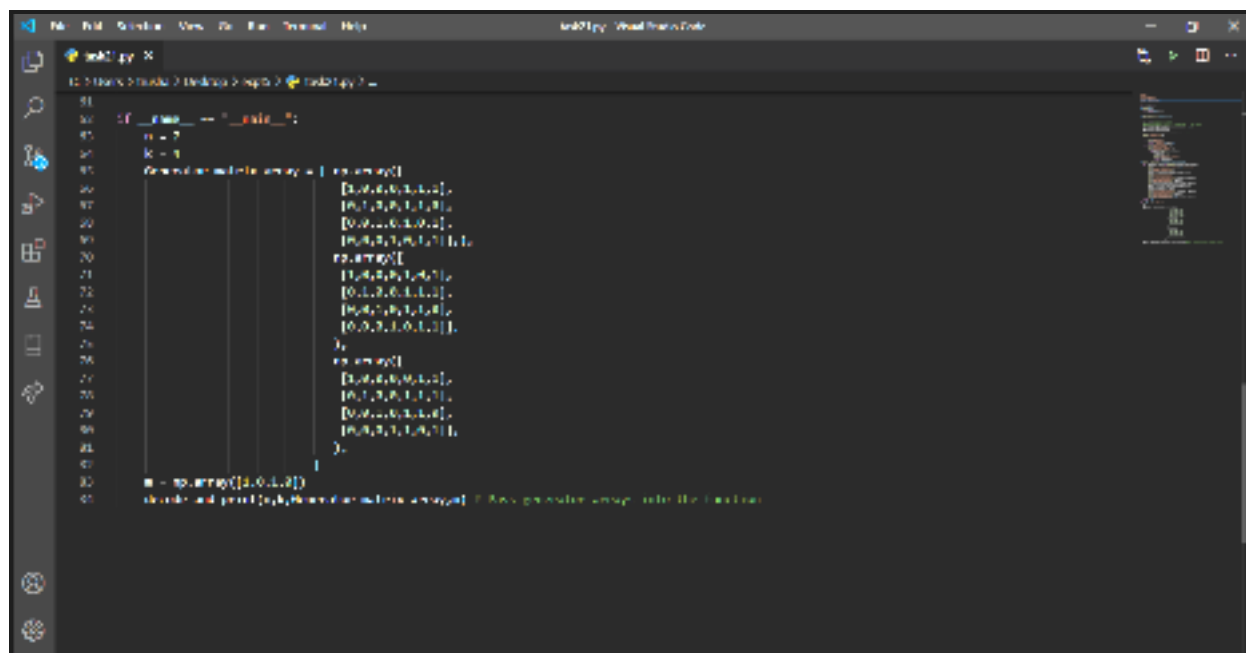
```
1 # code.py
2
3 # Import the necessary modules
4 from sys import argv
5 from string import ascii_lowercase as lowercase
6
7 # Get the message and keyword from the command line
8 message = argv[1]
9 keyword = argv[2]
10
11 # Convert the message and keyword to lists of integers
12 message = list(message)
13 keyword = list(keyword)
14
15 # Encrypt the message
16 def encrypt(message, keyword):
17     # Initialize the encrypted message
18     encrypted_message = []
19
20     # Loop through the message and keyword
21     for i in range(len(message)):
22         # Get the current message letter and keyword letter
23         message_letter = message[i]
24         keyword_letter = keyword[i % len(keyword)]
25
26         # Shift the message letter by the keyword letter
27         shifted_letter = (ord(message_letter) - ord('a') + ord(keyword_letter) - ord('a')) % 26
28
29         # Add the shifted letter to the encrypted message
30         encrypted_message.append(chr(shifted_letter + ord('a')))
31
32     # Return the encrypted message
33     return ''.join(encrypted_message)
34
35 # Decrypt the message
36 def decrypt(message, keyword):
37     # Initialize the decrypted message
38     decrypted_message = []
39
40     # Loop through the message and keyword
41     for i in range(len(message)):
42         # Get the current message letter and keyword letter
43         message_letter = message[i]
44         keyword_letter = keyword[i % len(keyword)]
45
46         # Shift the message letter by the keyword letter
47         shifted_letter = (ord(message_letter) - ord('a') - ord(keyword_letter) + ord('a')) % 26
48
49         # Add the shifted letter to the decrypted message
50         decrypted_message.append(chr(shifted_letter + ord('a')))
51
52     # Return the decrypted message
53     return ''.join(decrypted_message)
54
55 # Main function
56 def main():
57     # Get the message and keyword from the command line
58     message = argv[1]
59     keyword = argv[2]
60
61     # Encrypt the message
62     encrypted_message = encrypt(message, keyword)
63
64     # Decrypt the message
65     decrypted_message = decrypt(encrypted_message, keyword)
66
67     # Print the encrypted and decrypted messages
68     print(encrypted_message)
69     print(decrypted_message)
70
71 # Call the main function
72 if __name__ == '__main__':
73     main()
```



```

42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050

```



- Working of syndrome_decode function according to the formula given above in the encoder decoder diagram has been coded.

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 sequences 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  $n \cdot 2^k = 112$  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  $n \cdot 2^{**k} = 112$  single-bit error codewords
OOPS: Error decoding [1 0 0 0 0 0 0] ...expected [0 0 0 0] got [1 0 0 0]
```

Source Code for Task #3:

```

15
16 def decode_and_print(G, generator_matrix, array, n):
17     for i, row in enumerate(generator_matrix):
18         print(i, "Generator matrix: ", row)
19         print("Encoded dataword: ", row[n])
20         codeword = np.dot(r, np.transpose(generator_matrix[i]))
21         calculated_codeword = codeword
22         print("-----")
23         print("Transmitted codeword: ", (row[n] + calculated_codeword) % 2)
24         codeword[0] = 1
25         print("Received codeword: ", row[n])
26         print(syndrome_decode(codeword, n, generator_matrix))
27         print()
28         codeword = np.dot(r, np.transpose(generator_matrix[i]))
29         calculated_codeword = codeword
30         print("-----")
31         print("Transmitted codeword: ", (row[n] + calculated_codeword) % 2)
32         print("Received codeword: ", codeword)
33         print(syndrome_decode(codeword, n, generator_matrix))
34         print("Decoded dataword: ", row[n])
35     return None

```

Output showing both the cases (error and no error) :

[illegible]

```

----- generator matrix 2 -----
Encoded dataword :
[1 0 1 0]

Transmitted codeword:
[1 0 1 0 0 1 1]
Received codeword:
[0 0 1 0 0 1 1]
Syndrome:
[0 0 1 1]
Error detected :
Corrected codeword :
[1 0 1 0 0 1 1]
Identified dataword
[1 0 1 0]

Transmitted codeword:
[1 0 1 0 0 1 1]
Received codeword:
[1 0 1 0 0 1 1]
Syndrome:
[0 0 0 0]
No error..
Codeword: [1 0 1 0 0 1 1]
Dec (prev. dataword) [1 0 1 0]

```

```

TERMINAL  RUNNING  DISK USAGE
----- Generator matrix 2 -----
Encoded dataword :
[1 0 1 0]

Transmitted codeword:
[1 0 1 0 1 1 1]
Received codeword:
[0 0 1 0 1 1 1]
Syndrome:
[0 0 1 1]
Error detected :
Corrected codeword :
[1 0 1 0 1 1 1]
Identified dataword
[1 0 1 0]

Transmitted codeword:
[1 0 1 0 0 1 1]
Received codeword:
[1 0 1 0 1 1 1]
Syndrome:
[0 0 0 0]
No error..
Codeword: [1 0 1 0 1 1 1]
Dec (prev. dataword) [1 0 1 0]

```

- We have tested the function against 3 different generator matrices and displayed the corresponding output messages informing whether there was an error or not and if there was, then what was expected and what was found also has been outputted.

Conclusion:

- (1)The decoder for rectangular parity single error correction code in task 1 was implemented in Python which takes an $m \times n$ matrix as the input and locates the position of the error bit in the matrix using the principle of exclusive-OR for parity consistency.
- (2)For task 2, a syndrome decoder was implemented which took a codeword, the length of the actual message and a Generator matrix as the input parameters and returned the most likely transmitted message.
- (3)A testing function was implemented in task 3 which compares the input codeword and the derived codeword with respect to the syndrome and

calculates whether the derived codeword is the same as the input codeword.