

EEC269A - Error Correcting Codes I

Project Report

Chenyue Yang, Pranav Kharche, Parisa Oftadeh

June 11, 2023

Contents

1 Workload	2
2 Source & Destination	2
2.1 Source	2
2.1.1 Text string	2
2.1.2 Image	2
2.1.3 Audio	4
2.2 Destination	6
3 Channel	6
3.1 Binary Symmetric Channel (BSC)	6
3.2 Additive White Gaussian Noise (AWGN) Channel	7
4 (7,4) Systematic Linear Block (Hamming) Code	7
4.1 Encoder	8
4.2 Syndrome Decoder	8
4.2.1 Results: text string	8
4.2.2 Results: image	8
4.2.3 Results: audio	8
5 (n, k) Systematic Cyclic Code	8
5.1 Adjustable (n, k) for correcting t errors	10
5.2 Encoder	10
5.3 Syndrome Decoder	12
5.3.1 Text string	12
5.3.2 Image	12
5.3.3 Audio	12
5.4 Error Trapping Decoder	12
6 Appendix: Python source code	13
6.1 Source	13
6.2 Channel	18
6.3 Destination	24
6.4 Utilities	28
6.5 Linear code	36
6.6 Cyclic code	41

1 Workload

Table 1: Workload

Function	Workload	Contributor
Source	Text string (TXT), Image (PNG), Audio (WAV)	Chenye
Encoder	(7, 4) Systematic Linear Block (Hamming) Code (n, k) Systematic Cyclic Code	Chenye
Channel	Binary Symmetric Channel (BSC), adjustable error prob p	Chenye
Error Corrector	Syndrome Lookup Table for (7, 4) Linear Code Syndrome Lookup Table for (n, k) Cyclic Code, $t = 1$ Error Trapping for (n, k) Cyclic Code, adjustable t	Chenye Chenye Pranav
Decoder	(7, 4) Systematic Linear Block (Hamming) Code (n, k) Systematic Cyclic Code	Chenye
Destination	Text string (TXT), Image (PNG), Audio (WAV)	Chenye
Advanced features	Create generator matrix for (n, k) cyclic code Adjustable: (n, k) and correctable errors t	Pranav
Presentation	Slides	Parisa
Report	Written by contributors	-

2 Source & Destination

2.1 Source

2.1.1 Text string

The very basic function of the information source is to read a hard-coded text file into a bit stream. In our text file, the following string is stored in ASCII format:

```
Hello World!
EEC269A Error Correcting Code Demo
```

In ASCII format, each character is represented by 8 bits, shown in Table 2. Then, after transformation, the bit stream is of size 376 bits:

```
0 1 0 0 1 0 0 0 0 1 1 0 0 1 0 1 ...
```

2.1.2 Image

PNG (Portable Network Graphics) is a raster graphics file format that supports lossless data compression. PNG supports a large number of colors (up to 16 million), as well as variable transparency, which makes it useful for images with varying degrees of transparency or opacity. Additionally, because PNG is a lossless format, it preserves all the detail in the original image, which is not the case with lossy formats like JPEG. It is used for the storage and display of images on the internet, and is also used in graphic design and editing applications due to its lossless compression.

PNG files consist of a header followed by a series of data chunks, e.g.:

Table 2: ASCII Table example

Character	Hexadecimal	Decimal	Binary
...
A	41	65	0 1 0 0 0 0 0 1
B	42	66	0 1 0 0 0 0 1 0
C	43	67	0 1 0 0 0 0 1 1
D	44	68	0 1 0 0 0 1 0 0
E	45	69	0 1 0 0 0 1 0 1
F	46	70	0 1 0 0 0 1 1 0
G	47	71	0 1 0 0 0 1 1 1
H	48	72	0 1 0 0 1 0 0 0
...

1. Signature: The first eight bytes of a PNG file always contain the following decimal numbers: 137, 80, 78, 71, 13, 10, 26, 10. This signature indicates that the file is a PNG.
2. Header Chunk (IHDR): The first chunk after the signature is the IHDR chunk, which contains basic information about the image, such as width, height, bit depth, color type, compression method, filter method, and interlace method.
3. Palette Chunk (PLTE): This chunk is optional and only present for color type 3 (indexed color). It contains the color palette for the image.
4. Data Chunks (IDAT): These chunks contain the actual image data, which is formed by pixels. This data is compressed to reduce the size of the file.
5. End Chunk (IEND): This is the final chunk in a PNG file. It does not contain any data and its purpose is to indicate the end of the file.

Each chunk contains three standard fields: 4-byte length, 4-byte type code, 4-byte CRC and various internal fields that depend on the chunk type, shown in Figure 1 ¹. For example, the image file we are using has more than one image data (IDAT) chunks, each of which contains a portion of the image, shown in Table 3.

Ideally, the entire image file should be read into a bit stream and transmitted through the channel. However, if there are uncorrectable errors in the chunks other than the image data (IDAT), the received image will not be able to display. Therefore, we shall only work with the image data (IDAT) chunks for the purpose of visualization of a corrupted image. Also, this trick will not affect the statistical analysis of the system.

The information source is able to only extract the color information from a PNG file and convert it into a bit stream to be passed through the channel. This is done by using the Python library *NumPy*.

The library *NumPy* provides a method to only read out the color information of an image. The shape of the result array is typically (*height*, *width*, *channels*), where:

1. *height* is the number of pixels in the vertical direction (i.e. the number of rows of pixels);
2. *width* is the number of pixels in the horizontal direction (i.e. the number of columns of pixels);
3. *channels* is the number of color channels per pixel. This value depends on what type of image it is:
 - In an RGB image, the three channels correspond to Red, Green, and Blue, respectively. Each channel value usually ranges from 0 to 255, where 0 indicates none of that color is present and 255 indicates that color is fully present.
 - In a grayscale image, there is typically only one channel. The value in this single channel indicates the level of gray, where 0 is black and 255 is white.

¹Figure 1 is from webpage PNG file chunk inspector - Project Nayuki

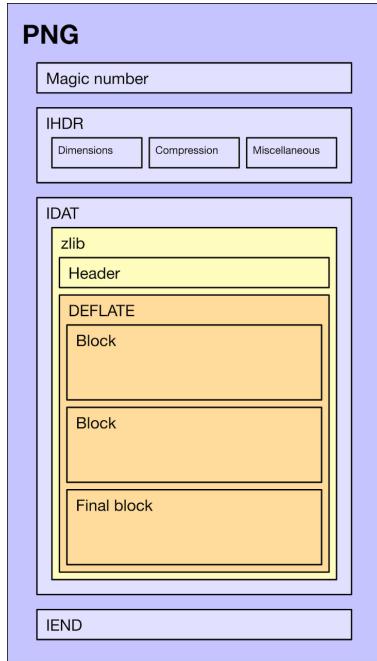


Figure 1: PNG file structure



Figure 2: Our testing PNG image

- There are many other color spaces that have different meanings for their channels.

The data type for each channel of each pixel is *uint8*, which is an unsigned integer that takes 8 bits. Then, the array is flattened into a bit stream by converting each channel of each pixel into an 8-bits binary number and appending them together.

For example, as for our testing image², shown in Figure 2, the shape of the result array is $(1280, 854, 3)$, which means that the image has 1280 rows of pixels, 854 columns of pixels, and 3 color channels per pixel (RGB). Then, the array is flattened into a bit stream of 26,234,880 bits.

2.1.3 Audio

WAV (Waveform Audio File Format) is a digital audio standard for storing audio bitstream on PCs. WAV is an application of the Resource Interchange File Format (RIFF) method for storing data in chunks, and it is primarily used on Windows systems. WAV files are typically used for raw and uncompressed audio, though they can also contain compressed audio. A WAV file is divided into several sections or chunks, shown in Figure 3³. Each chunk serves a different purpose and holds different types of data. The basic structure of a WAV file includes the following chunks:

1. RIFF Chunk: The RIFF chunk is the first chunk in a WAV file and identifies the file as a WAV file. It includes a header with the "RIFF" identifier and an integer indicating the remaining length of the entire file.
2. Format Chunk: Also known as the "fmt " chunk (with a space after 'fmt'), this contains important information about the audio data. This includes the audio format (e.g., PCM), the number of channels (mono, stereo, etc.), the sample rate, the byte rate, the block alignment, and the bit depth (bits per sample).
3. Data Chunk: This is where the actual audio data is stored. The "data" header is followed by an integer representing the length of the data, and then by the raw audio data itself.

²The image file used in this project is photographed by *Chenyue Yang*. The material is free from any copyright restrictions and can be used without any potential legal implications.

³Figure 3 is from webpage WAV Files: File Structure, Case Analysis and PCM Explained

Table 3: Our PNG file structure

Start offset	Chunk outside
0	Special: File signature; Length: 8 bytes
8	Data length: 13 bytes; Type: IHDR; Name: Image header; CRC-32: CB3954EC
33	Data length: 1 bytes; Type: sRGB; Name: Standard RGB color space; CRC-32: AECE1CE9
46	Data length: 976 bytes; Type: eXIf; Name: Exchangeable Image File (Exif) Profile; CRC-32: 47FCFA4D
1,034	Data length: 9 bytes; Type: pHYs; Name: Physical pixel dimensions; CRC-32: 5024E7F8
1,055	Data length: 4 514 bytes; Type: iTxt; Name: International textual data; CRC-32: C9C76B16
5,581	Data length: 16 384 bytes; Type: IDAT; Name: Image data; CRC-32: 8462CABD
21,977	Data length: 16 384 bytes; Type: IDAT; Name: Image data; CRC-32: 2C9D007C
...	...
1,727,161	Data length: 12 170 bytes; Type: IDAT; Name: Image data; CRC-32: 68067F52
1,739,343	Data length: 0 bytes; Type: IEND; Name: Image trailer; CRC-32: AE426082

Similar to the image file, the information source is able to only extract the audio Data Chunk from a WAV file and convert it into a bit stream to be passed through the channel. This trick ensures both the visualization of the results and the statistical analysis of the system. The extraction is done by using the Python library *soundfile*.

The library *soundfile* provides a method to only read out the audio data from a WAV file. The result contains two parts:

1. audio array: This is a *NumPy* array that contains the audio data from the file. The shape of the array depends on the number of channels in the audio file. If the audio is mono, the array will be one-dimensional. If the audio is stereo, the array will be two-dimensional, with one sub-array for each channel. The values in the array represent the amplitude of the audio signal at each sample point, and are of the data type specified.
2. sample rate: This is an integer that represents the number of samples per second in the audio file, measured in Hertz (Hz). Common sample rates include 44100 Hz (standard for audio CDs), 48000 Hz (standard for video production and DVDs), and 96000 Hz (used in high-definition formats).

Then, the *int16* array is "viewed" as *uint8* array by *numpy.view()* and flattened into a bit stream by converting each sample into an 8-bits binary number and appending them together. Note that the *numpy.view()* operation simply reinterprets the binary data, and does not convert or scale the data, which originally could be negative or positive.

For example, as for our testing audio file⁴, shown in Figure 4 and Figure 5, the shape of the result audio array is *(268237, 2)* and the sample rate is 8000 Hz. It means that the 33.5-second audio file has 268237 samples per channel, and there are two channels (stereo). Then, the array is flattened into a bit stream of 8,583,584 bits.

⁴The audio file used in this project is free downloaded from file-examples.com. The material is free from any copyright restrictions and can be used without any potential legal implications.

The Canonical WAVE file format

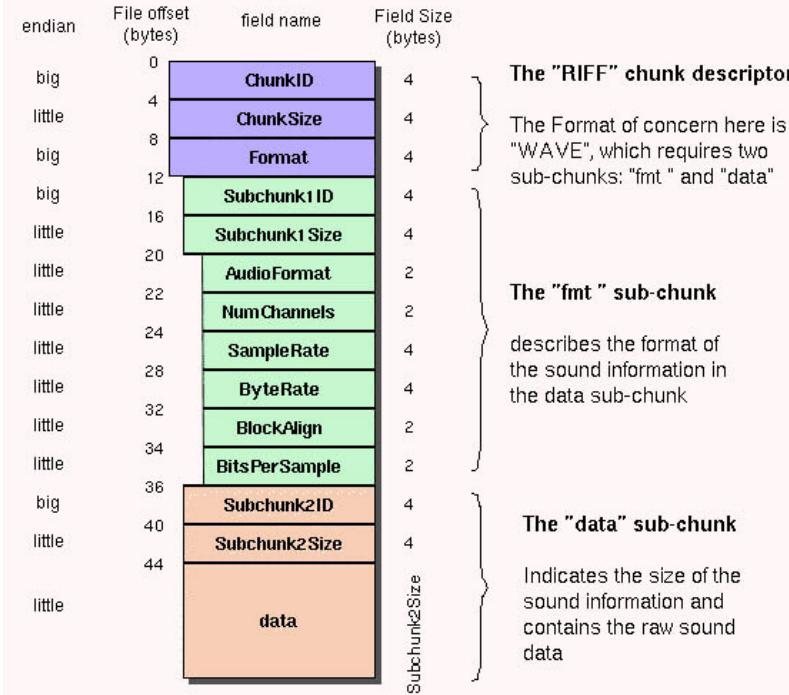


Figure 3: Canonical WAV file structure

2.2 Destination

At the destination, the bit stream is re-construed into the original format, and stored. It is inevitable that some metadata is lost when processed by the system, since they are not transmitted through the channel. For example, the re-constructed image file will not have the camera and lens information. However, this is the trade-off for a better visualization, and will not affect the statistical analysis of the system.

3 Channel

3.1 Binary Symmetric Channel (BSC)

The Binary Symmetric Channel (BSC) is a fundamental concept in information theory and telecommunications, specifically in the field of error detection and correction. It is a model used to represent a communication channel, where the information is transmitted in the form of binary digits, or bits: 0s and 1s.

The "symmetric" aspect of the BSC refers to the fact that it has the same probability of an error occurring whether the transmitted bit is a 0 or a 1. For instance, if the error probability is 0.01, then 1% of 0s are received as 1s, and 1% of 1s are received as 0s.

A BSC is characterized by two parameters: the input bit and the crossover probability / error probability, shown in Figure 6. The input bit is either 0 or 1, which represents the binary information being transmitted. The crossover probability, often denoted as p , represents the likelihood of the transmitted bit being received incorrectly.

Although the model is simple, the BSC is a building block for understanding more complex communication systems. By studying the BSC, researchers can develop algorithms and systems that minimize errors in binary data transmission, improving the reliability of digital communications.

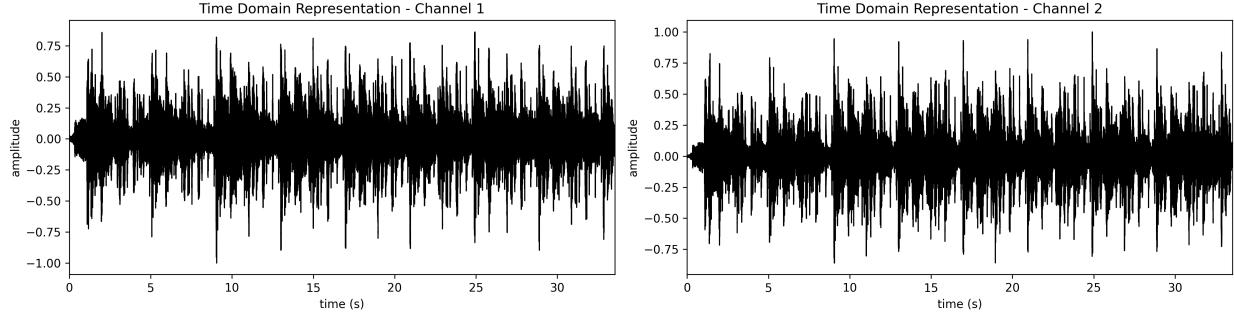


Figure 4: Time domain waveform of our testing audio file

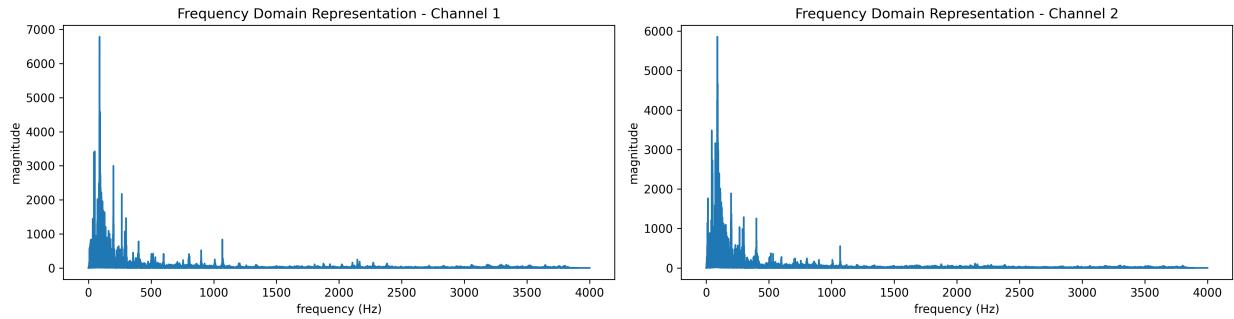


Figure 5: Frequency domain waveform of our testing audio file

3.2 Additive White Gaussian Noise (AWGN) Channel

4 (7,4) Systematic Linear Block (Hamming) Code

The (7,4) Systematic Linear Block (Hamming) Code, often referred to simply as the Hamming (7,4) Code, is a fundamental concept in the realm of error detection and correction, and plays a crucial role in the field of digital communications and information theory. This coding scheme was introduced by Richard Hamming in the late 1940s to address the problem of error detection and correction in data transmission over noisy channels.

The Hamming (7,4) Code is a block code that maps 4-bit messages to 7-bit codewords. It gets its name from the fact that it takes in 4 data bits and outputs 7 bits (including 3 parity bits). The parity bits are extra bits added to the data to enable the detection and correction of errors that might occur during transmission.

A key characteristic of the Hamming (7,4) Code is its systematic form. This means that the bits of the original message appear unaltered in the encoded output, making it easy to extract the original data from the received message, even if errors are detected.

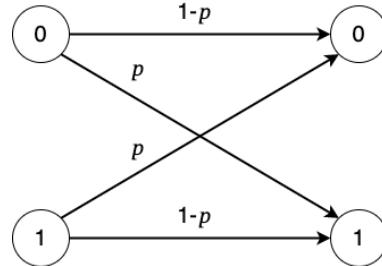


Figure 6: Binary Symmetric Channel (BSC)

The Hamming (7,4) Code is designed to detect and correct single-bit errors, meaning it can identify and fix any situation where only one bit in the 7-bit codeword has been flipped due to noise or interference in the communication channel. This makes it a powerful tool in improving the reliability and robustness of data communication systems.

4.1 Encoder

The encoding process in a Hamming (7,4) code involves the multiplication of a 4-bit message vector with a generator matrix. The generator matrix for a (7,4) Hamming Code, in its systematic form, is given as:

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

If we denote the message vector as

$$u = [u_1 \ u_2 \ u_3 \ u_4],$$

then the encoded codeword v is computed by:

$$v = u \cdot G.$$

This multiplication produces a 7-bit codeword, which is then ready for transmission.

4.2 Syndrome Decoder

For the Hamming (7, 4) Code, the parity check matrix H is a 3 by 7 matrix that is used to detect errors in received code words. In this context, the matrix is designed such that the product of H and any valid code word (expressed as a column vector) is a zero vector. This property is used to detect errors in received messages: if the product of H and the received vector is not zero, then the received vector is not a valid code word, and an error has occurred. The decoding process of the (7,4) Hamming code involves utilizing a syndrome lookup table. The syndrome, a binary vector, is computed from the received bits and is used to identify the error bit position. Each syndrome corresponds to a potential error location, and its lookup table is a key tool for efficient error correction.

4.2.1 Results: text string

Table 4: Text string encoded with Linear Hamming passed through BSC

Original	Without correction	Corrected
Hello World! EEC269A Error Correcting Code Demo	Hello Wopld! EEC269A Arror Correcting Kode Demo	Hello World! EEC269A Error Correcting Code Demo

4.2.2 Results: image

4.2.3 Results: audio

5 (n, k) Systematic Cyclic Code

The (n, k) Systematic Cyclic Code is a powerful and widely used error correction code (ECC) in the field of digital communications and information theory. The notation (n, k) denotes that each block of the code consists of ' n ' bits, where ' k ' bits are the actual information or data, and the remaining ' $n - k$ ' bits are the parity or check bits used for error detection and correction.

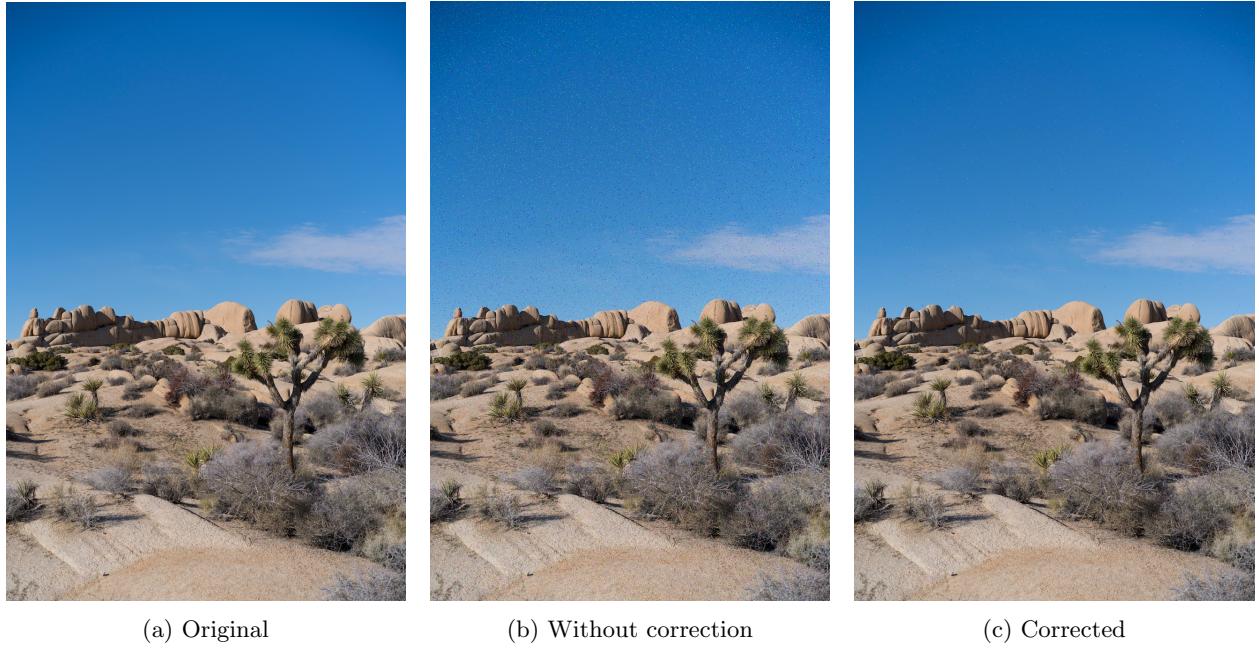


Figure 7: Image encoded with Linear Hamming passed through BSC (entire)

The term "systematic" implies that the data bits remain in their original form in the encoded output, while the parity bits are appended to form the full code word. This is in contrast to non-systematic coding schemes, where the data and parity bits are intermixed.

"Cyclic" codes, meanwhile, have a key property where a cyclic shift of a code word results in another valid code word. This cyclic nature leads to efficient implementation, especially in hardware, as simple shift registers can be used.

When the linear error-correcting codes is able to correct only single-bit errors in codewords, they are also called "Hamming" code. They accomplish this by introducing additional parity bits that allow the identification of single-bit errors.

In the context of (n, k) Systematic Cyclic Codes, the codes are designed such that any t -bit error can be detected and corrected, making them a reliable choice for environments where such errors are common and the integrity of information is paramount.

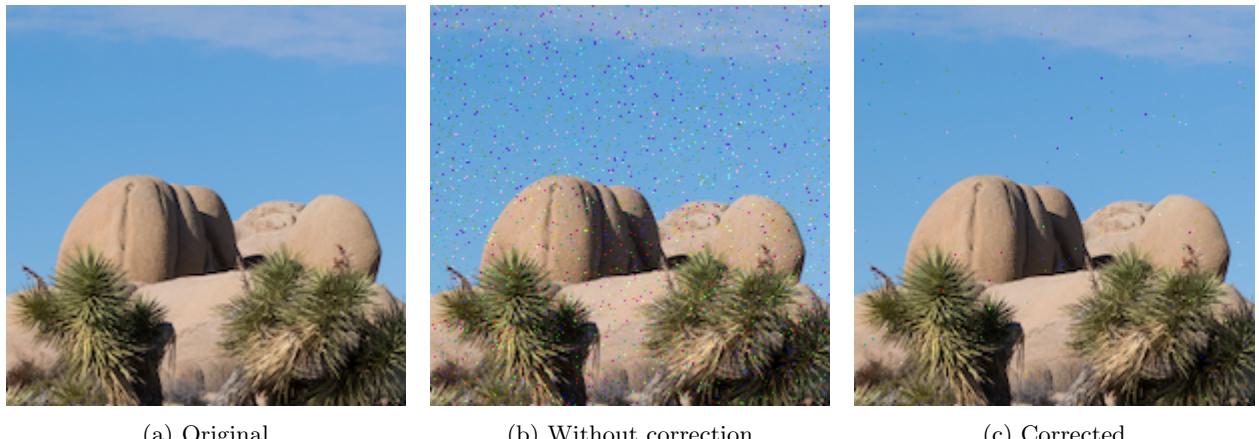


Figure 8: Image encoded with Linear Hamming passed through BSC (details)

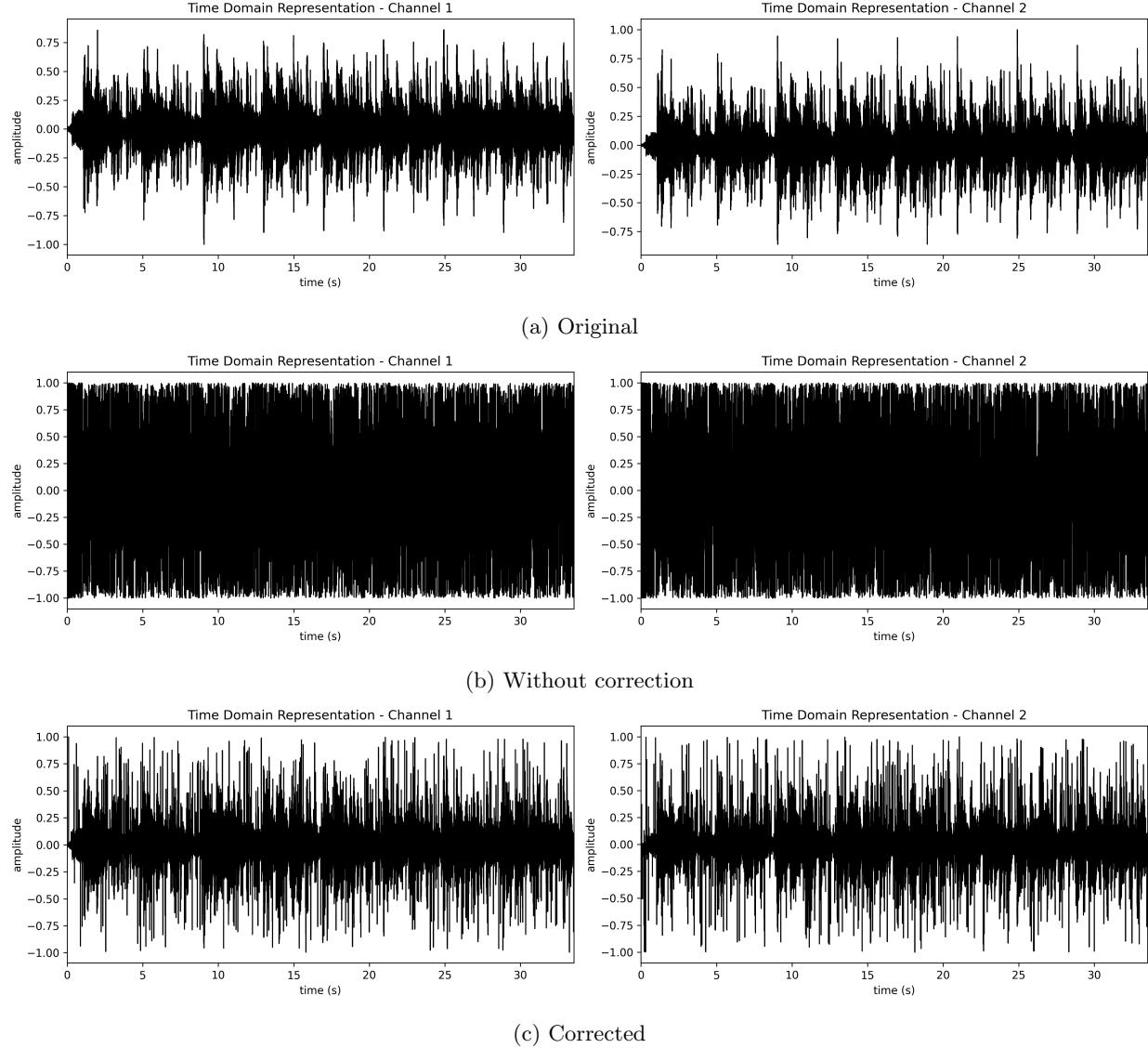


Figure 9: Audio encoded with Linear Hamming passed through BSC

5.1 Adjustable (n, k) for correcting t errors

TODO

5.2 Encoder

The encoding process in a (n, k) Systematic Cyclic Code involves the multiplication of a k -bit message vector with a generator matrix. The generator matrix for a (n, k) Cyclic Hamming Code, in its systematic form, is given as:

$$G = [P \quad I_k]$$

where I_k represents a k by k identity matrix, and P represents a k by $(n - k)$ matrix for parity-check bits. Following is an example of the generator matrix for a $(15, 11)$ Cyclic Hamming Code, which is generated

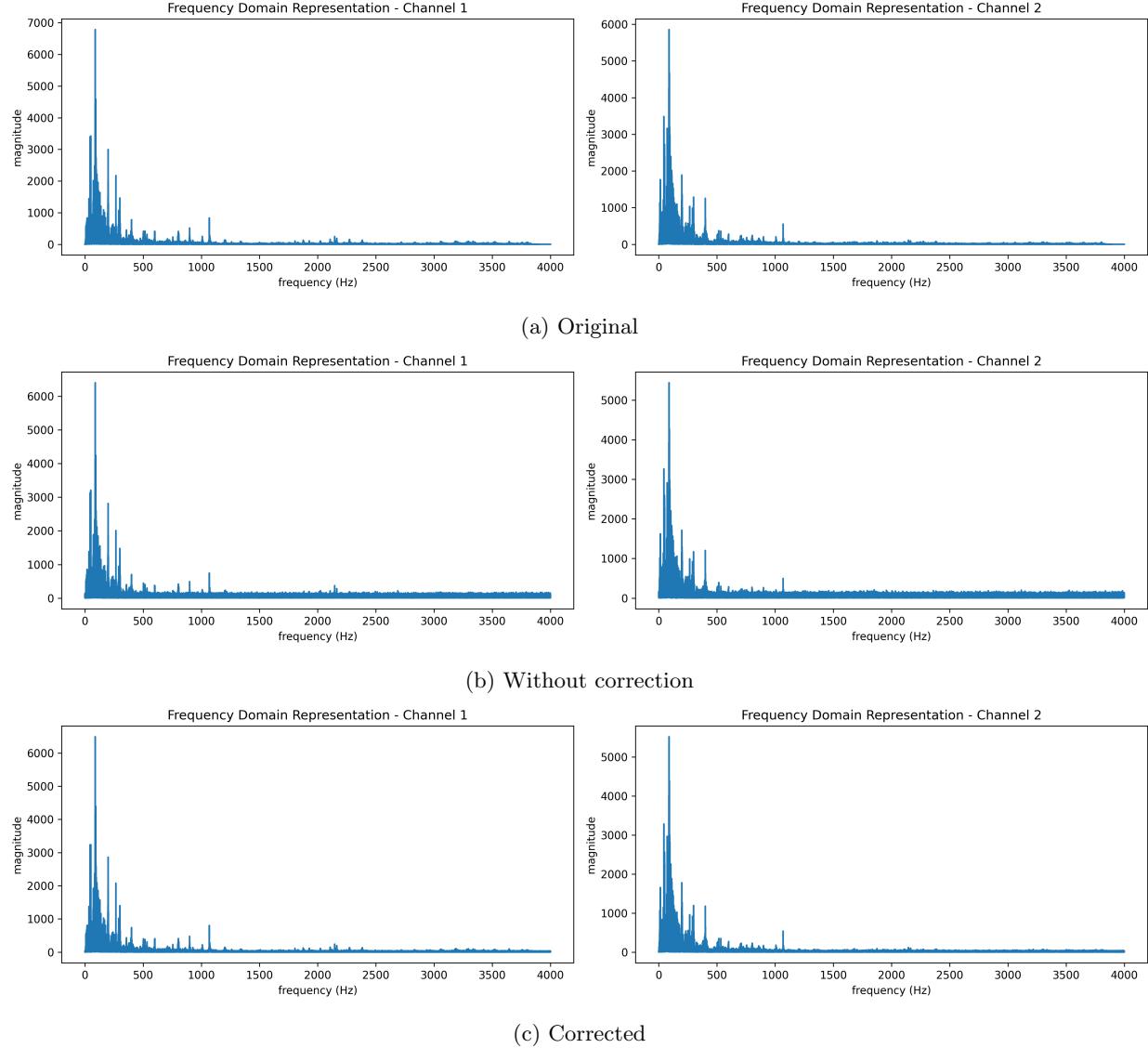


Figure 10: Audio encoded with Linear Hamming passed through BSC

with procedures in Section 5.1:

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

If we denote the message vector as

$$u = [u_1 \ u_2 \ \dots \ u_k],$$

then the encoded codeword v is computed by:

$$v = u \cdot G.$$

This multiplication produces a n -bit codeword, which is then ready for transmission.

5.3 Syndrome Decoder

When there are only 1-bit errors in the received codewords, the error bit can be corrected by using the syndrome lookup table.

5.3.1 Text string

Table 5: Text string encoded with Cyclic Hamming passed through BSC

Original	Without correction	Corrected
Hello World! EEC269A Error Correcting Code Demo	H <u>u</u> lo World! EEC269A Error Correctifg Code Demo	Hello World! EEC269A Error Correcting Code Demo

5.3.2 Image

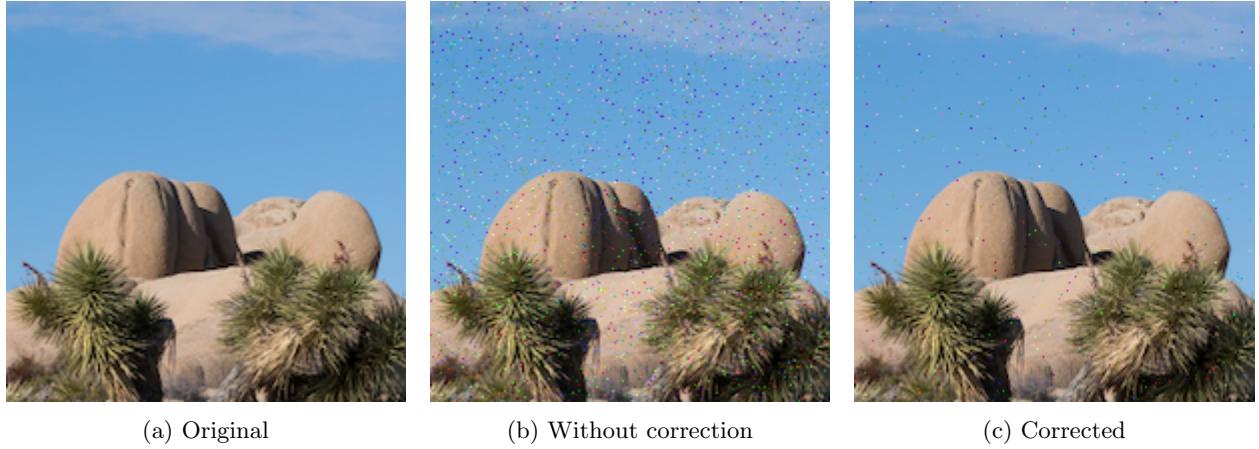


Figure 11: Image encoded with Cyclic Hamming passed through BSC (entire)

5.3.3 Audio

5.4 Error Trapping Decoder

TODO



(a) Original

(b) Without correction

(c) Corrected

Figure 12: Image encoded with Cyclic Hamming passed through BSC (details)

6 Appendix: Python source code

6.1 Source

```

1 # Copyright (c) 2023 Chenye Yang
2
3 import numpy as np
4 from PIL import Image
5 import soundfile as sf
6 # from scipy.io import wavfile
7 import logging
8
9 # Create a logger in this module
10 logger = logging.getLogger(__name__)
11
12
13 class Source:
14     """
15         The data source
16     """
17
18     def read_txt(self, src_path):
19         """
20             Read the text file as binary bits
21
22             @type src_path: string
23             @param src_path: source file path, with extension
24         """
25         with open(src_path, 'rb') as file:
26             byte_array = bytearray(file.read())
27             self._digital_data = np.unpackbits(np.frombuffer(byte_array, dtype=np.uint8))
28
29
30     def read_png(self, src_path):
31         """

```

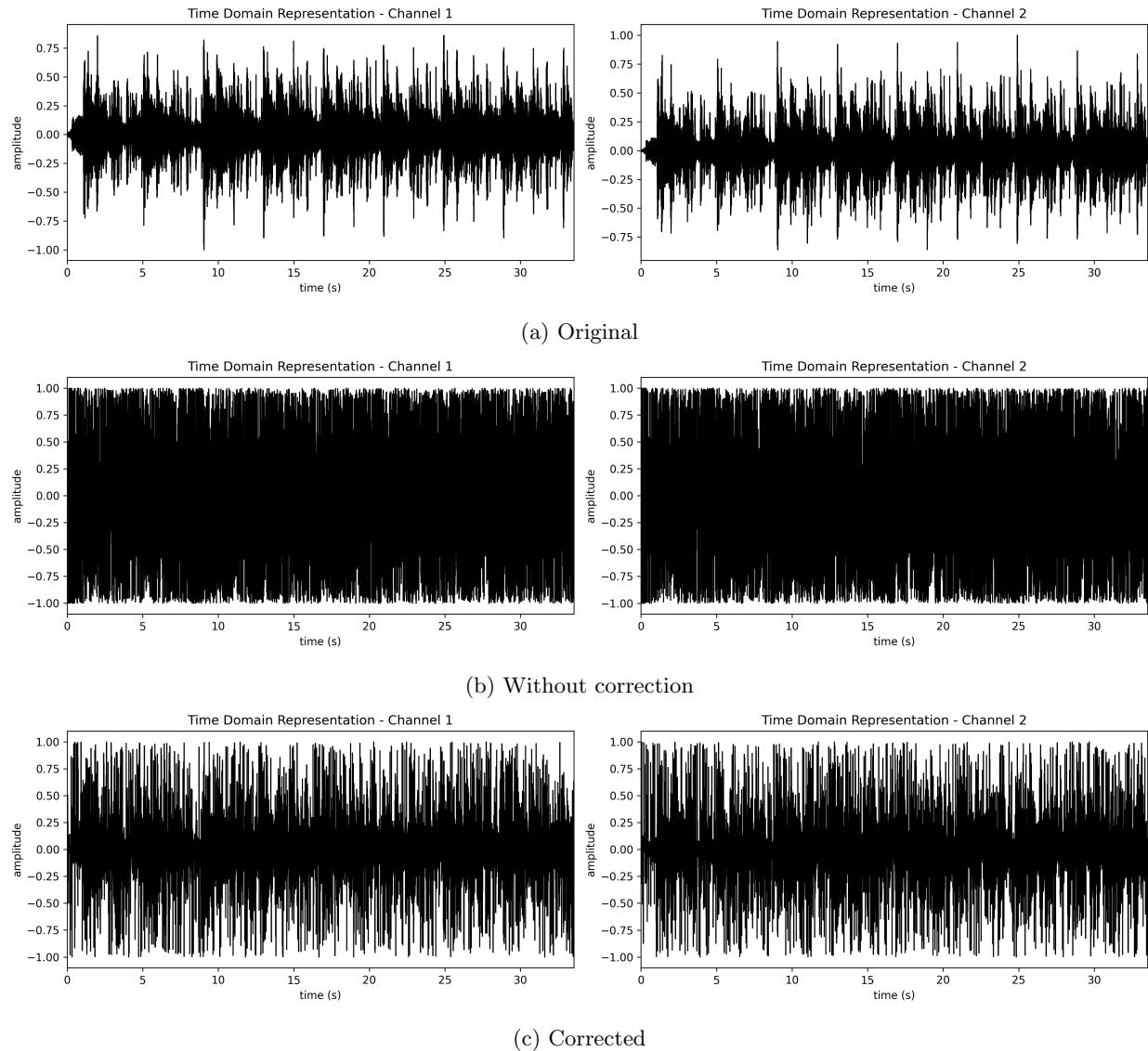


Figure 13: Audio encoded with Cyclic Hamming passed through BSC

```

32     Read the png file ,
33     store the color information as binary bits in _digital_data ,
34     store the color information as ndarray in _analogue_data .
35     Return the height , width , channels of the image .
36
37     @type  src_path: string
38     @param src_path: source file path , with extension
39
40     @rtype:    tuple
41     @return:   height , width , channels
42     """
43     # Open the image file
44     image = Image.open(src_path)
45
46     # Convert the image to a NumPy array

```

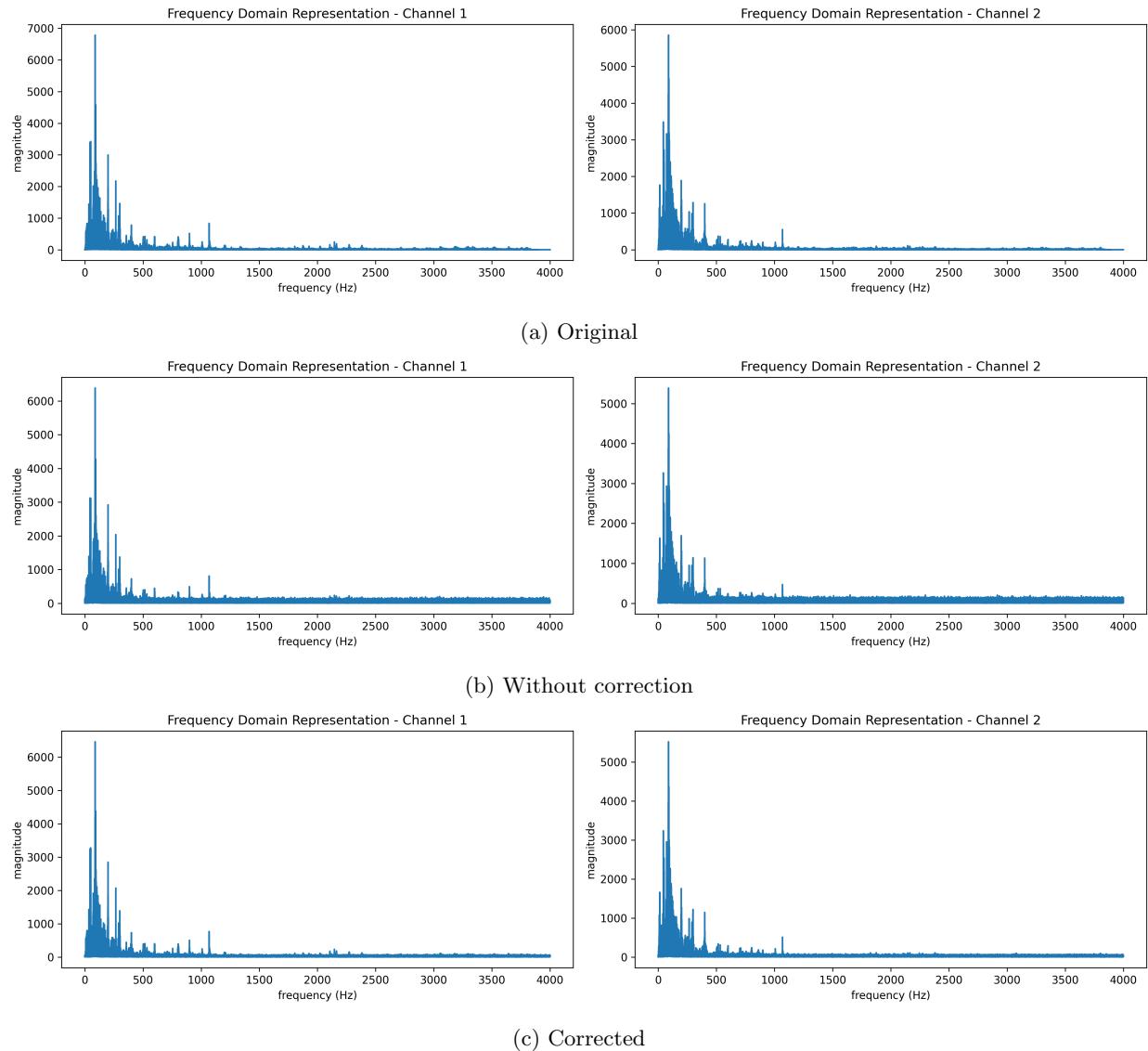


Figure 14: Audio encoded with Cyclic Hamming passed through BSC

```

47     img_array = np.array(image)
48
49     # Get the shape of the array (height, width, channels)
50     height, width, channels = img_array.shape
51
52     # Convert the img_array to binary format and flatten the array
53     bits = np.unpackbits(img_array.astype(np.uint8))
54
55     # Store the binary bits
56     self._digital_data = bits
57     # Store the analogue data
58     self._analogue_data = img_array
59
60     return height, width, channels
61

```

```

62
63     def read_wav(self, src_path):
64         """
65             Read the wav file,
66             store the audio information as binary bits in _digital_data,
67             store the audio information as ndarray in _analogue_data.
68             Return the frame_rate, sample_width, channels of the audio.
69
70             @type src_path: string
71             @param src_path: source file path, with extension
72
73             @rtype: tuple, int
74             @return: shape, sample_rate
75         """
76
77         # Open the audio file
78         # sample_rate, audio_array = wavfile.read(src_path)
79         audio_array, sample_rate = sf.read(src_path, dtype='int16')
80         shape = audio_array.shape
81
82         # Convert the audio_array to binary format and flatten the array
83         bits = np.unpackbits(audio_array.astype(np.int16).view(np.uint8))
84
85         # Store the binary bits
86         self._digital_data = bits
87         # Store the analogue data
88         self._analogue_data = audio_array
89
90         return shape, sample_rate
91
92     # def read_mp3(self, src_path):
93     """
94         # Read the mp3 file,
95         # store the audio information as binary bits in _digital_data,
96         # store the audio information as ndarray in _analogue_data.
97         # Return the frame_rate, sample_width, channels of the audio.
98
99         #             @type src_path: string
100        #             @param src_path: source file path, with extension
101
102        #             @rtype: tuple
103        #             @return: frame rate, sample width, channels
104        #
105        #             # Open the audio file
106        #             audio = AudioSegment.from_file(src_path, format="mp3")
107        #             frame_rate, sample_width, channels = audio.frame_rate, audio.
108        #             sample_width, audio.channels
109
110        #             # Convert the audio to a NumPy array
111        #             audio_array = np.array(audio.get_array_of_samples())
112        #             # print(audio_array.dtype)
113
114        #             # Get the shape of the array: e.g. (2392270,)
115        #             # shape = audio_array.shape

```

```

115
116     #     # Convert the audio_array to binary format and flatten the array
117     #     bits = np.unpackbits(audio_array.astype(np.uint8))
118
119     #     # Store the binary bits
120     #     self._digital_data = bits
121     #     # Store the analogue data
122     #     self._analogue_data = audio_array
123
124     #     return frame_rate, sample_width, channels
125
126
127
128     def get_digital_data(self):
129         """
130             Get the bits to be transmitted
131
132             @rtype: ndarray
133             @return: data bits
134         """
135
136         return self._digital_data
137
138     def get_analogue_data(self):
139         """
140             Get the analogue data to be transmitted
141
142             @rtype: ndarray
143             @return: analogue data
144         """
145
146         return self._analogue_data
147
148 if __name__ == '__main__':
149     # test
150     source = Source()
151     source.read_txt("Resource/hardcoded.txt")
152     text_bits = source.get_digital_data()
153     print(text_bits)
154     print(type(text_bits))
155
156     source.read_png("Resource/image.png")
157     image_bits = source.get_digital_data()
158     print(image_bits[:16])
159     image_pixels = source.get_analogue_data()
160     print(image_pixels[0][0][:2])
161
162     # frame_rate, sample_width, channels = source.read_mp3("Resource/
163     # file_example_MP3_1MG.mp3")
164     # print(frame_rate, sample_width, channels)
165     # audio_bits = source.get_digital_data()
166     # print(audio_bits[:16])
167     # audio_array = source.get_analogue_data()
168     # print(audio_array[:2])

```

```

168     shape, sample_rate = source.read_wav("Resource/file_example_WAV_1MG.wav")
169     print('`')
170     print(shape, sample_rate)
171     audio_bits = source.get_digital_data()
172     print(audio_bits[:16])
173     audio_array = source.get_analogue_data()
174     print(audio_array[0])
175

```

6.2 Channel

```

# Copyright (c) 2023 Chenye Yang, Pranav Kharche

import numpy as np
import logging

from Utils import polyTools as pt

# Create a logger in this module
logger = logging.getLogger(__name__)

def create_parity_check_matrix(G):
    """
    Create the parity-check matrix, H = [I_{n-k} | P.T]

    @type G: ndarray
    @param G: generator matrix in systematic form, G = [P | I_k]

    @rtype: ndarray
    @return: parity-check matrix
    """
    # Get the size of the generator matrix
    k, n = G.shape

    # Create the parity-check matrix
    # G = [P | I_k], Extract P
    P = G[:, :n-k]

    # H = [I_{n-k} | P.T]
    H = np.hstack((np.eye(n-k, dtype=np.uint8), P.T))

    return H

def create_syndrome_table(H):
    """
    Create a table of all possible syndromes and their corresponding error
    vectors (syndrome look-up table)

    @type H: ndarray
    @param H: parity-check matrix
    """

```

```

42     """
43     # Get the size of the parity-check matrix
44     _, n = H.shape
45
46     # Create a table of all possible syndromes and their corresponding error
47     # vectors (syndrome look-up table)
48     coset_leader = np.vstack((np.zeros(n, dtype=np.uint8), np.eye(n, dtype=np.
49                               uint8)))
50     possible_syndromes = (coset_leader @ H.T) % 2
51     syndrome_table = {tuple(possible_syndromes[i]) : coset_leader[i] for i in
52                         range(n+1)}
53
54     return syndrome_table
55
56
57 def pad_bits(bits, k):
58     """
59     Pad the bits array with zeroes so its length is divisible by k.
60
61     @type bits: ndarray
62     @param bits: TX message
63
64     @type k: int
65     @param k: number of bits per codeword
66
67     """
68     padded_bits = bits.copy()
69     remainder = len(bits) % k
70     if remainder != 0:
71         padding_length = k - remainder
72         padded_bits = np.pad(padded_bits, (0, padding_length), mode='constant',
73                             )
74     return padded_bits
75
76
77 def remove_padding(bits, padding_length):
78     """
79     Remove the padding from the array.
80
81     @type bits: ndarray
82     @param bits: RX message
83
84     @type padding_length: int
85     @param padding_length: length of the original TX message
86     """
87
88     return bits[:-padding_length] if padding_length != 0 else bits
89
90
91 class Linear_Code:
92     """
93     (7, 4) Systematic Linear Block (Hamming) Code
94     """
95     def __init__(self):

```

```

92     self.n, self.k = 7, 4
93     self.G = np.array([[1, 1, 0, 1, 0, 0, 0],
94                         [0, 1, 1, 0, 1, 0, 0],
95                         [1, 1, 1, 0, 0, 1, 0],
96                         [1, 0, 1, 0, 0, 0, 1]], dtype=np.uint8)
97     self.H = np.array([[1, 0, 0, 1, 0, 1, 1],
98                         [0, 1, 0, 1, 1, 1, 0],
99                         [0, 0, 1, 0, 1, 1, 1]], dtype=np.uint8)
100    self.syndrome_table = {
101        (0, 0, 0): np.array([0, 0, 0, 0, 0, 0, 0], dtype=np.uint8),
102        (1, 0, 0): np.array([1, 0, 0, 0, 0, 0, 0], dtype=np.uint8),
103        (0, 1, 0): np.array([0, 1, 0, 0, 0, 0, 0], dtype=np.uint8),
104        (0, 0, 1): np.array([0, 0, 1, 0, 0, 0, 0], dtype=np.uint8),
105        (1, 1, 0): np.array([0, 0, 0, 1, 0, 0, 0], dtype=np.uint8),
106        (0, 1, 1): np.array([0, 0, 0, 0, 1, 0, 0], dtype=np.uint8),
107        (1, 1, 1): np.array([0, 0, 0, 0, 0, 1, 0], dtype=np.uint8),
108        (1, 0, 1): np.array([0, 0, 0, 0, 0, 0, 1], dtype=np.uint8),
109    }
110
111
112 def encoder_systematic(self, bits):
113     """
114         Systematic – Encode the to-be-transmitted binary bits message with (n, k) systematic encoder, pad with zero if not divisible, return the to-be-transmitted codewords
115
116     @type bits: ndarray
117     @param bits: TX message
118
119     @rtype: ndarray
120     @return: TX codewords
121     """
122     # Pad the bits array with zeroes so its length is divisible by self.k
123     padded_bits = pad_bits(bits, self.k)
124
125     # Reshape the bits array to have one row per message
126     messages = padded_bits.reshape(-1, self.k)
127
128     # Perform the matrix multiplication operation in one go, and flatten
129     # the result to 1D array
130     encoded_array = np.dot(messages, self.G) % 2
131
132     # Flatten the array
133     encoded_array = encoded_array.flatten()
134
135     return encoded_array
136
137 def decoder_systematic(self, encoded_array, padding_length=0):
138     """
139         Systematic – Decode the received binary bits codeword with (n, k) systematic decoder, remove padding, return the received message
140
141     @type encoded_array: ndarray

```

```

142     @param encoded_array: RX codewords
143
144     @type padding_length: int
145     @param padding_length: length of the padding (default: 0, means no
146     padding)
147
148     @rtype: ndarray
149     @return: RX message
150     """
151
152     # Reshape the array so each row is a codeword
153     reshaped_array = encoded_array.reshape(-1, self.n)
154
155     # Compute the syndrome for each codeword
156     syndromes = np.dot(reshaped_array, self.H.T) % 2
157
158     # Count the number of non-zero syndromes (errors)
159     err_count = np.count_nonzero(np.any(syndromes, axis=1))
160
161     # Decode by taking the last self.k elements from each codeword
162     decoded_array = reshaped_array[:, self.n - self.k:self.n].flatten()
163
164     logger.debug(f"Error codeword rate: {err_count}/{len(encoded_array)} // {self.n}")
165
166     # Remove the padding from the array
167     if padding_length != 0:
168         decoded_array = remove_padding(decoded_array, padding_length)
169
170     return decoded_array
171
172
173 def corrector_syndrome(self, received_array):
174     """
175
176     Systematic – Correct the received binary bits codeword (up to 1 error
177     bit) with (n, k) Hamming syndrome look-up table corrector,
178     return the estimated TX codeword = (RX codeword + error pattern)
179
180     @type received_array: ndarray
181     @param received_array: RX codewords
182
183     @rtype: ndarray
184     @return: estimated TX codewords
185     """
186
187     # Reshape the received_array so each row is a codeword
188     reshaped_array = received_array.reshape(-1, self.n)
189
190     # Compute the syndrome for each codeword
191     syndromes = np.dot(reshaped_array, self.H.T) % 2
192
193     # Copy reshaped_array to corrected_array for correction
194     corrected_array = reshaped_array.copy()
195
196     # Loop over the syndromes
197     for i, syndrome in enumerate(syndromes):

```

```

193     if np.any(syndrome):
194         # Convert syndrome to tuple for lookup
195         tuple_syndrome = tuple(syndrome)
196         corrected_array[i] = (reshaped_array[i] + self.syndrome_table[
197             tuple_syndrome]) % 2
198
199         # Flatten corrected_array to match the shape of the input
200         received_array
201         corrected_array = corrected_array.flatten()
202
203
204
205     class Cyclic_Code(Linear_Code):
206         """
207         (n, k) Systematic Cyclic Code
208         """
209
210         def __init__(self, n, k, nECC = 1):
211             self.n = n
212             self.k = k
213             self.G_dec = pt.findMatrix(self.n, self.k, nECC)
214             self.G = pt.genMatrixDecmial2Ndarray(self.G_dec, self.n)
215
216             # Following used in trapping corrector
217             self.nECC = pt.correctableErrors(self.n, self.k, self.G_dec)
218             self.HT_dec = pt.buildParityMatrix(self.n, self.k, self.G_dec)
219             self.HT = pt.genMatrixDecmial2Ndarray(self.HT_dec, self.n-self.k)
220
221             # Following used in sydrome look-up table corrector (inherit from
222             # Linear_Code)
223             self.H = create_parity_check_matrix(self.G)
224             self.syndrome_table = create_syndrome_table(self.H)
225
226             logger.info("Generated_a_(%d,_%d)_cyclic_code", self.n, self.k)
227             logger.info("%d_correctable_errors", self.nECC)
228             logger.info("Generator_matrix:\n%s", self.G)
229
230
231         def corrector_trapping(self, received_array):
232             """
233                 Systematic - Correct the received binary bits codeword (up to nECC
234                 error bits) with (n, k) Error trapping corrector,
235                 return the estimated TX codeword = (RX codeword + error pattern)
236
237             @type received_array: ndarray
238             @param received_array: RX codewords
239
240             @rtype:    ndarray
241             @return:   estimated TX codewords
242             """
243
244             reshaped_array = received_array.reshape(-1, self.n)
245             corrected_array = np.empty(reshaped_array.shape, dtype=np.uint8)
246             logger.debug('size=%s', reshaped_array.shape)

```

```

243     # print('correcte\n', corrected_array, corrected_array.shape)
244     syndromes = np.dot(reshaped_array, self.HT) % 2
245
246     for word_count, received_word in enumerate(reshaped_array):
247         if word_count % 100000 == 0:
248             logger.debug('count: %s', word_count)
249         if syndromes[word_count].sum() == 0:
250             corrected_array[word_count] = received_word
251             continue
252         syndrome = syndromes[word_count]
253         for shift in range(self.n):
254             # print(shift, syndrome)
255             if syndrome.sum() <= self.nECC:
256                 padding = np.zeros(self.k, dtype=np.uint8)
257                 error = np.concatenate((syndrome, padding))
258                 corrected_word = np.bitwise_xor(received_word, error)
259                 corrected_word = np.roll(corrected_word, shift)
260                 # print('word', np.roll(received_word, shift))
261                 # print('fix', corrected_word)
262                 corrected_array[word_count] = corrected_word
263                 break
264             received_word = np.roll(received_word, -1)
265             if shift == self.n-1:
266                 logger.debug('Uncorrectable_Error')
267                 corrected_array[word_count] = received_word
268                 syndrome = np.dot(received_word, self.HT) % 2
269             # print(syndromes)
270             corrected_array = corrected_array.flatten()
271     return corrected_array
272
273
274 class Channel:
275     """
276     Channel
277     """
278     def binary_symmetric_channel(self, input_bits, p):
279         """
280             BSC – binary symmetric channel with adjustable error probability
281
282             @type input_bits: ndarray
283             @param input_bits: TX codewords
284
285             @type p: float
286             @param p: error_probability
287
288             @rtype: ndarray
289             @return: RX codewords
290         """
291         # Generate a uniform random array of the same shape as input_bits
292         random_numbers = np.random.rand(*input_bits.shape)
293
294         # Identify where the random array is less than p
295         mask = random_numbers < p
296

```

```

297     # Use this mask to flip the bits at those indices
298     output_bits = np.where(mask, 1 - input_bits, input_bits)
299
300     return output_bits
301
302
303
304 if __name__ == '__main__':
305     # test
306     tx_msg = np.array([0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1], dtype=np.uint8)
307     n = 7
308     k = 4
309     print("Original_bits:", tx_msg)
310
311     # Channel
312     channel = Channel()
313
314     # Linear Code
315     linear_code = Linear_Code()
316
317     # Encoding
318     tx_codewords = linear_code.encoder_systematic(tx_msg)
319     print("Encoded_bits:::::", tx_codewords)
320
321     # Passing through the channel
322     rx_codewords = channel.binary_symmetric_channel(tx_codewords, 0.1)
323     print("Bits_after_channel:", rx_codewords)
324
325     # Correction with syndrome look-up table
326     estimated_tx_codewords = linear_code.corrector_syndrome(rx_codewords)
327     print("Estimated_TX_bits:", estimated_tx_codewords)
328
329     # Decoding
330     padding_length = k - (len(tx_msg) % k)
331     rx_msg = linear_code.decoder_systematic(estimated_tx_codewords,
332                                             padding_length)
332     print("Decoded_bits:", rx_msg)

```

6.3 Destination

```

1 # Copyright (c) 2023 Chenye Yang
2
3 import numpy as np
4 from PIL import Image
5 import soundfile as sf
6 # from scipy.io import wavfile
7 import logging
8
9 # Create a logger in this module
10 logger = logging.getLogger(__name__)
11
12
13 class Destination:

```

```

14     """
15     The destination
16     """
17
18     def set_digital_data(self, bits):
19         """
20             Record the received digital data
21
22                 @type bits: ndarray
23                 @param bits: data bits
24             """
25         self._digital_data = bits
26
27
28     def set_analogue_data(self, array):
29         """
30             Record the received analogue data
31
32                 @type array: ndarray
33                 @param array: analogue data array
34             """
35         self._analogue_data = array
36
37
38     def get_digital_data(self):
39         """
40             Get the received digital data
41
42                 @rtype: ndarray
43                 @return: data bits
44             """
45         return self._digital_data
46
47
48     def get_analogue_data(self):
49         """
50             Get the received analogue data
51
52                 @rtype: ndarray
53                 @return: analogue data
54             """
55         return self._analogue_data
56
57
58     def write_txt(self, dest_path):
59         """
60             Write data to a text file
61
62                 @type dest_path: string
63                 @param dest_path: destination file path, with extension
64             """
65         byte_array = bytearray(np.packbits(self._digital_data).tobytes())
66         with open(dest_path, 'wb') as file:
67             file.write(byte_array)

```

```

68
69
70 def write_png_from_digital(self , dest_path , height , width , channels):
71     """
72         Write color information in bits array to a png file , at the same time
73             store the color information as ndarray in _analogue_data
74
75             @type dest_path: string
76             @param dest_path: destination file path , with extension
77
78             @type height: int
79             @param height: height of the image
80
81             @type width: int
82             @param width: width of the image
83
84             @type channels: int
85             @param channels: channels of the image
86
87         # Convert the bit array to uint8 array
88         uint8_array = np.packbits(self._digital_data)
89
90         # Reshape the array to a 3D array of pixels (height , width , channels)
91         pixels = uint8_array.reshape(height , width , channels)
92
93         # Store the analogue data
94         self._analogue_data = pixels
95
96         # Create a new image from the pixel values
97         new_image = Image.fromarray(pixels)
98
99         # Save the new image to a file
100        new_image.save(dest_path)
101
102    def write_png_from_analogue(self , dest_path):
103        """
104            Write color information in (height , width , channels) array to a png
105                file , at the same time store the color information as bit array in
106                    _digital_data
107
108            @type dest_path: string
109            @param dest_path: destination file path , with extension
110
111            # Create a new image from the pixel values
112            new_image = Image.fromarray(self._analogue_data)
113
114            # Save the new image to a file
115            new_image.save(dest_path)
116
117            # Convert the img_array to binary format and flatten the array
118            bits = np.unpackbits(self._analogue_data.astype(np.uint8))

```

```

119     self._digital_data = bits
120
121
122     def write_wav_from_digital(self, shape, sample_rate, dest_path):
123         """
124             Write audio information in bits array to a wav file, at the same time
125             store the audio information as ndarray in _analogue_data
126
127                 @type dest_path: string
128                 @param dest_path: destination file path, with extension
129             """
130
131             # Convert the bit array to int16 array
132             int16_array = np.packbits(self._digital_data).view(np.int16)
133
134             # Reshape the array to a 2D array of samples (channels, samples)
135             audio_array = int16_array.reshape(shape)
136
137             # Store the analogue data
138             self._analogue_data = audio_array
139
140             # Write the array to a wav file
141             # wavfile.write(dest_path, sample_rate, audio_array)
142             sf.write(dest_path, audio_array, sample_rate)
143
144
145     def write_wav_from_analogue(self, sample_rate, dest_path):
146         """
147             Write audio information in audio array to a wav file, at the same time
148             store the audio information as bit array in _digital_data
149
150                 @type dest_path: string
151                 @param dest_path: destination file path, with extension
152             """
153
154             # Write the array to a wav file
155             # wavfile.write(dest_path, sample_rate, self._analogue_data)
156             sf.write(dest_path, self._analogue_data, sample_rate)
157
158             # Convert the audio_array to binary format and flatten the array
159             bits = np.unpackbits(self._analogue_data.astype(np.int16).view(np.
160                 uint8))
161
162             # Store the binary bits
163             self._digital_data = bits
164
165
166     # def write_mp3_from_digital(self, frame_rate, sample_width, channels,
167     #     dest_path):
168     #
169     #         """
170     #             Write audio information in bits array to a mp3 file
171     #
172     #                 @type dest_path: string
173     #                 @param dest_path: destination file path, with extension
174     #
175     #             """
176     #             # Convert the bit array to uint16 array

```

```

169     #     uint16_array = np.packbits(self._digital_data).astype(np.int16)
170
171     #     # Create a new AudioSegment object from the modified audio array
172     #     modified_audio = AudioSegment(uint16_array.tobytes(), frame_rate,
173     #                                     sample_width, channels)
174
175     #     # Export the modified audio to a new MP3 file
176     #     modified_audio.export(dest_path, format="mp3")
177
178     # def write_mp3_from_analogue(self, frame_rate, sample_width, channels,
179     #                             dest_path):
180     #     """
181     #         Write audio information in audio array to a mp3 file
182     #
183     #             @type dest_path: string
184     #             @param dest_path: destination file path, with extension
185     #
186     #             # Create a new AudioSegment object from the modified audio array
187     #             modified_audio = AudioSegment(self._analogue_data.tobytes(),
188     #                                           frame_rate, sample_width, channels)
189
190     #             # Export the modified audio to a new MP3 file
191     #             modified_audio.export(dest_path, format="mp3")
192
193 if __name__ == '__main__':
194     # test
195     bits = np.array([0, 1, 0, 0, 1, 0, 0, 0]) # H
196     destination = Destination()
197     destination.set_data(bits)
198     destination.write_file("output.txt")

```

6.4 Utilities

```

1  # Copyright (c) 2023 Pranav Kharche, Chenye Yang
2  # Toolbox for polynomials and CRC
3
4  import numpy as np
5  import logging
6
7  # Create a logger in this module
8  logger = logging.getLogger(__name__)
9
10
11 # find the order of a polynomial in integer form
12 def order(p):
13     p = p >> 1
14     order = 0
15     while p:
16         p = p >> 1
17         order += 1

```

```

18     return order
19
20 # find the hamming weight of a polynomial in integer form
21 def weight(p):
22     weight = 0
23     while p:
24         if p & 1:
25             weight += 1
26             p = p >> 1
27     return weight
28
29 # reverse the order of the bits of the polynomial
30 # this is used because some tasks work better with the highest order
31 # coefficient as the least significant bit
32 # if no order is specified, it is found using order(p) from above
33 # returns the reversed polynomial
34 def bitRev(p, pOrder = None):
35     if p == 0:
36         return 0
37     if pOrder == None:
38         pOrder = order(p)
39
40     retVal = 0
41     for i in range(pOrder+1):
42         retVal = (retVal << 1) + (p & 1)
43         p = p >> 1
44     return retVal
45
46 # divide two polynomials
47 # each polynomial must be with the lowest order coeff as the LSB and the
48 # highest order as the MSB
49 # if order is not provided it is found automatically
50 # returns a tuple of the result and remainder of the division.
51 def polyDiv(dividend, divisor, dividendOrder = None, divisorOrder = None):
52     if dividendOrder == None or divisorOrder == None:
53         dividendOrder = order(dividend)
54         divisorOrder = order(divisor)
55
56     sizeDiff = dividendOrder - divisorOrder
57     rem = dividend
58     remOrder = dividendOrder
59     result = 0
60     for i in range(sizeDiff, -1, -1):
61         if(rem >> remOrder):
62             rem = rem ^ (divisor << i)
63             result = result + (1 << i)
64     remOrder == 1
65
66     # if rem:
67     #     print(f'{dividend:b}', ':', f'{divisor:b}', '=', f'{result:b}', 'R', f'{rem:0{divisorOrder}b}', sep=' ')
68     # else:
69     #     print(f'{dividend:b}', ':', f'{divisor:b}', '=', f'{result:b}')

```

```

68         return result , rem
69
70 # find a generator polynomial that can create a cyclic code
71 # this uses the mathematical rules of a cyclic code and brute force searches
72     for a compatible polynomial
73 # returns None if polynomial cannot be found.
74 # n, k: code length and information length of the code
75 # findN: find the nth solution. function normally stops upon finding the first
76     solution that matches n and k. However, this is not always optimal.
77 # Callers of this function may find that the 2nd or 3rd
78     solution works better.
79 def findGen(n,k, initialGen = None):
80     target = (1 << n) + 1
81     if initialGen:
82         gen = initialGen
83     else:
84         gen = (1 << (n-k)) + 1
85     maxGen = (1 << (n-k+1))
86     rem = 1
87     while rem:
88         gen += 2
89         if gen >= maxGen:
90             return None
91         parity , rem = polyDiv(target , gen , n , n-k)
92     logger.debug('generator ==%s' , f'{gen:b}')
93     logger.debug('parity ==%s' , f'{parity:b}')
94     return gen
95
96 # build a generator matrix for a systematic cyclic code
97 # n, k: code length and information length of the code
98 # genPoly: the generator polynomial to be used
99 def buildGenMatrix(n, k, genPoly):
100    genMatrix = [genPoly]
101    # print(f'{genPoly:0{n}b}')
102    for i in range(k-1):
103        nextLine = genMatrix[i] << 1
104        if (nextLine >> (n-k))%2:
105            nextLine = nextLine ^ genPoly
106        genMatrix.append(nextLine)
107        # print(f'{nextLine:0{n}b}')
108    logger.debug('Generator matrix')
109    for i in range(k):
110        genMatrix[i] = bitRev(genMatrix[i], n-1)
111        logger.debug(f'{genMatrix[i]:0{n}b}')
112    return genMatrix
113
114 # build the transpose of the parity matrix using the generator matrix
115 # n, k: code length and information length of the code
116 # genMatrix: the generator matrix to be used
117 def buildParityMatrix(n, k, genMatrix):
118     Ht = []
119     m = n-k
120     logger.debug('Parity Matrix Transpose')
121     for i in range(m-1, -1, -1):

```

```

119             Ht.append(1 << i)
120             logger.debug(f'{(Ht[-1]):0{m}b}')
121         for row in genMatrix:
122             Ht.append(row >> k)
123             logger.debug(f'{(Ht[-1]):0{m}b}')
124     return Ht
125
126
127 # create a generator matrix for a systematic cyclic code that matches the
128 # requirements provided by the input
129 # returns None if requirements are impossible to meet.
130 # n, k: code length and information length of the code
131 # nECC: number of correctable errors the code must have
132 #           input of nECC = None means it will search for the best
133 #           possible code.
134 def findMatrix(n,k, nECC = 1):
135     numErrors = 0
136     bestPoly = 0
137     genPoly = 0
138     bestGen = None
139     maxErrors = 0
140     while nECC == None or numErrors < nECC:
141         genPoly = findGen(n,k,genPoly)
142         if genPoly == None:
143             genMatrix = None
144             break
145         genMatrix = buildGenMatrix(n, k, genPoly)
146         numErrors = correctableErrors(n, k, genMatrix)
147         if numErrors > maxErrors:
148             bestGen = genMatrix
149             maxErrors = numErrors
150             bestPoly = genPoly
151         if (nECC == None and bestGen == None) or (nECC and genMatrix == None):
152             logger.debug('no_viable_matrix_found')
153             # print('no viable matrix found')
154             return None
155             logger.debug('used_polynomial_%d', bestPoly)
156             logger.debug('%d_correctable_errors', maxErrors)
157             # print('used polynomial', bestPoly)
158             # print(maxErrors, 'correctable errors')
159             return bestGen
160
161 # encode a data word with the provided generator matrix
162 def encode(data, genMatrix):
163     if order(data)+1 > len(genMatrix):
164         return None
165     result = 0
166     data = bitRev(data, len(genMatrix)-1)
167     for row in genMatrix:
168         result = result ^ ((data & 1)*row)
169         data = data >> 1
170     return result
171
172 # Find the number of errors that a specific code can fix.

```

```

171 # This is based on the minimum distance between code words which
172 # is eq to the minimum weight of the non-zero codewords
173 def correctableErrors_old(n, k, genMatrix):
174     weightTable = [0] * (n+1)
175     minWeight = n
176     for i in range(1, 1<<k, 1):
177         wordWeight = weight(encode(i, genMatrix))
178         if wordWeight < minWeight:
179             minWeight = wordWeight
180             # print(minWeight)
181         weightTable[wordWeight] += 1
182     # print(weightTable)
183     return ((minWeight-1) // 2)
184
185
186 def nCr_Seq(n, r):
187     if r > n:
188         return []
189     if r == 0:
190         return [0]
191     if n == r:
192         return [(1<<n) - 1]
193
194     seq = []
195     for inc in nCr_Seq(n-1, r-1):
196         seq.append((inc<<1) + 1)
197     for exc in nCr_Seq(n-1, r):
198         seq.append(exc<<1)
199     return seq
200
201 def correctableErrors(n, k, genMatrix):
202     minWeight = n
203     for minPossible in range(1, n-k+1, 1):
204         if minWeight <= minPossible:
205             break
206             # print(k, 'choose', minPossible)
207         for word in nCr_Seq(k, minPossible):
208             if minWeight <= minPossible:
209                 break
210             wordWeight = weight(encode(word, genMatrix))
211             if wordWeight < minWeight:
212                 minWeight = wordWeight
213                 # print(minWeight, word)
214     return (minWeight-1) // 2
215
216 def genMatrixDecmial2Ndarray(genMatrix_decimal, n):
217     """
218     Convert the generator matrix in decimal form to numpy array form
219
220     @type genMatrix_decimal: list
221     @param genMatrix_decimal: generator matrix in decimal form
222
223     @type n: int
224     @param n: number of bits in a codeword

```

```

225
226         @rtype:    ndarray
227         @return:   generator matrix in numpy array form
228     """
229     # Convert the generator matrix in decimal form to binary form
230     genMatrix_binary = [[int(bit) for bit in f'{num:0{n}b}'] for num in
231                         genMatrix_decimal]
232
233     # Convert the binary generator matrix to a numpy array
234     G = np.array(genMatrix_binary, dtype=np.uint8)
235
236     return G

```

```

1  # Copyright (c) 2023 Chenye Yang
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from scipy.fft import fft
6
7
8  def plot_wav_time_domain(audio_array, sample_rate, dest_path):
9      """
10         Plot the audio signal in the time domain.
11
12         @type audio_array: ndarray
13         @param audio_array: audio data array
14
15         @type sample_rate: int
16         @param sample_rate: sample rate of the audio
17
18         @type dest_path: string
19         @param dest_path: destination file path, with extension
20     """
21
22     # Copy the audio array
23     data = audio_array.copy()
24     # Normalize to [-1, 1]
25     data = data / np.max(np.abs(data), axis=0)
26
27     times = np.arange(len(data))/float(sample_rate)
28
29     # Prepare the subplots
30     fig, axs = plt.subplots(1, 2, figsize=(15, 4))
31
32     # Time domain representation for channel 1
33     axs[0].fill_between(times, data[:, 0], color='k')
34     axs[0].set_xlim(times[0], times[-1])
35     axs[0].set_xlabel('time(s)')
36     axs[0].set_ylabel('amplitude')
37     axs[0].set_title('Time-Domain-Representation--Channel-1')
38
39     # Time domain representation for channel 2
40     axs[1].fill_between(times, data[:, 1], color='k')
41     axs[1].set_xlim(times[0], times[-1])

```

```

41     axs[1].set_xlabel('time_(s)')
42     axs[1].set_ylabel('amplitude')
43     axs[1].set_title('Time-Domain_Representation--Channel_2')
44
45     # Display the plot
46     plt.tight_layout()
47     plt.savefig(dest_path, dpi=300)
48     plt.close()
49
50
51 def plot_wav_frequency_domain(audio_array, sample_rate, dest_path):
52     """
53         Plot the audio signal in the frequency domain.
54
55         @type audio_array: ndarray
56         @param audio_array: audio data array
57
58         @type sample_rate: int
59         @param sample_rate: sample rate of the audio
60
61         @type dest_path: string
62         @param dest_path: destination file path, with extension
63     """
64
65     # Copy the audio array
66     data = audio_array.copy()
67     # Normalize to [-1, 1]
68     data = data / np.max(np.abs(data), axis=0)
69
70     # Prepare the subplots
71     fig, axs = plt.subplots(1, 2, figsize=(15, 4))
72
73     # Frequency domain representation for channel 1
74     fft_out_channel_1 = fft(data[:, 0])
75     magnitude_spectrum_channel_1 = np.abs(fft_out_channel_1)
76     frequencies_channel_1 = np.linspace(0, sample_rate, len(
77         magnitude_spectrum_channel_1))
78
79     axs[0].plot(frequencies_channel_1[:int(len(frequencies_channel_1)/2)],
80                 magnitude_spectrum_channel_1[:int(len(magnitude_spectrum_channel_1)/2)])
81     # plot only first half of frequencies
82     axs[0].set_xlabel('frequency_(Hz)')
83     axs[0].set_ylabel('magnitude')
84     axs[0].set_title('Frequency-Domain_Representation--Channel_1')
85
86     # Frequency domain representation for channel 2
87     fft_out_channel_2 = fft(data[:, 1])
88     magnitude_spectrum_channel_2 = np.abs(fft_out_channel_2)
89     frequencies_channel_2 = np.linspace(0, sample_rate, len(
90         magnitude_spectrum_channel_2))
91
92     axs[1].plot(frequencies_channel_2[:int(len(frequencies_channel_2)/2)],
93                 magnitude_spectrum_channel_2[:int(len(magnitude_spectrum_channel_2)/2)])
94     # plot only first half of frequencies
95     axs[1].set_xlabel('frequency_(Hz)')

```

```

89     axs[1].set_ylabel('magnitude')
90     axs[1].set_title('Frequency-Domain-Representation--Channel-2')
91
92     # Display the plot
93     plt.tight_layout()
94     plt.savefig(dest_path, dpi=300)
95     plt.close()

```

```

1  # Copyright (c) 2023 Chenye Yang
2
3 import numpy as np
4
5
6
7 def num_codeword_with_t_errors(original_bits, corrupted_bits, t, n):
8     """
9         Calculate how many codewords in the bit stream contains t error bits.
10
11     @type original_bits: ndarray
12     @param original_bits: original bit stream
13
14     @type corrupted_bits: ndarray
15     @param corrupted_bits: corrupted bit stream
16
17     @type t: int
18     @param t: number of errors
19
20     @type n: int
21     @param n: length of the codeword
22
23     @rtype: int
24     @return: number of codewords that contains exactly t errors
25     """
26
27     # Reshape the array so each row is a codeword
28     original_bits = original_bits.reshape(-1, n)
29     corrupted_bits = corrupted_bits.reshape(-1, n)
30
31     # Modulo 2 addition
32     diff = (original_bits + corrupted_bits) % 2
33
34     # Count the number of 1s in each row
35     result = np.count_nonzero(diff, axis=1)
36
37     # Count the number of rows that contains exactly t 1s
38     return np.count_nonzero(result == t)
39
40
41 def num_correct_messages(original_bits, corrupted_bits, k):
42     """
43         Calculate how many messages in the bit stream are correct.
44
45     @type original_bits: ndarray
46     @param original_bits: original bit stream

```

```

46
47     @type  corrupted_bits: ndarray
48     @param corrupted_bits: corrupted bit stream
49
50     @type  k: int
51     @param k: length of the message
52
53     @rtype:   int
54     @return:  number of messages that are correct
55 """
56 # Reshape the array so each row is a message
57 original_bits = original_bits.reshape(-1, k)
58 corrupted_bits = corrupted_bits.reshape(-1, k)
59
60 # Count the number of rows that are exactly the same
61 return np.count_nonzero(np.all(original_bits == corrupted_bits, axis=1))
62
63
64 def num_correct_bits(original_bits, corrupted_bits):
65 """
66 Calculate how many bits in the bit stream are correct.
67
68     @type  original_bits: ndarray
69     @param original_bits: original bit stream
70
71     @type  corrupted_bits: ndarray
72     @param corrupted_bits: corrupted bit stream
73
74     @rtype:   int
75     @return:  number of bits that are correct
76 """
77 # Count the number of bits that are exactly the same
78 return np.count_nonzero(original_bits == corrupted_bits)

```

6.5 Linear code

```

1 # Copyright (c) 2023 Chenye Yang
2
3 import logging
4 import os
5
6 import source
7 import channel
8 import destination
9 from Utils import plot_wav, stat_analysis
10
11
12 # Check if the directory exists
13 if not os.path.exists('Result/Linear/'):
14     os.makedirs('Result/Linear/')
15
16 # Configure the logging
17 logging.basicConfig(filename='Result/Linear/logfile-linear.log',

```

```

18     filemode='w', # Overwrite the file
19     level=logging.INFO,
20     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
21
22 # Create a logger in the main module
23 logger = logging.getLogger(__name__)
24
25
26
27 def linear_txt():
28     """
29         Source - Channel encoder - Channel - Channel decoder - Destination
30
31         txt      - (7,4) linear      - bsc      - (7,4) linear      - txt
32     """
33     logger.info("/**/TXT**/")
34     src = source.Source()
35     chl = channel.Channel()
36     linear_code = channel.Linear_Code()
37     dest = destination.Destination()
38
39
40     src.read_txt("Resource/hardcoded.txt")
41     tx_msg = src.get_digital_data()
42
43     tx_codeword = linear_code.encoder_systematic(tx_msg)
44
45     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
46
47     # Statistic analysis
48     correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
49                     rx_codeword, 0, 7)
50     one_error_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
51                     rx_codeword, 1, 7)
52     total_codewords = len(tx_codeword) // 7
53     uncorrectable_codewords = total_codewords - correct_codewords -
54         one_error_codewords
55     logger.info("Channel_statistic_analysis:")
56     logger.info("Total codewords: %d", total_codewords)
57     logger.info("Correct codewords: %d", correct_codewords)
58     logger.info("Codeword_error_rate: %f", (total_codewords -
59         correct_codewords) / total_codewords)
60     logger.info("One_error_codewords: %d", one_error_codewords)
61     logger.info("Uncorrectable_codewords: %d", uncorrectable_codewords)
62
63     # without error correction
64     rx_msg = linear_code.decoder_systematic(rx_codeword)
65     dest.set_digital_data(rx_msg)
66     dest.write_txt("Result/Linear/linear-bsc-output.txt")
67
68     # Statistic analysis
69     correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
70     logger.info("Before_correction:")

```

```

67     logger.info("Number_of_correct_bits: %d", correct_bits)
68     logger.info("Number_of_incorrect_bits: %d", len(tx_msg) - correct_bits)
69     logger.info("Bit_error_rate: %f", (len(tx_msg) - correct_bits) / len(
70         tx_msg))
71
72     # with error correction
73     estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
74     rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
75     dest.set_digital_data(rx_msg)
76     dest.write_txt("Result/Linear/linear-bsc-output-syndrome-corrected.txt")
77
78     # Statistic analysis
79     correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
80     logger.info("After_correction:")
81     logger.info("Number_of_correct_bits: %d", correct_bits)
82     logger.info("Number_of_incorrect_bits: %d", len(tx_msg) - correct_bits)
83     logger.info("Bit_error_rate: %f", (len(tx_msg) - correct_bits) / len(
84         tx_msg))
85
86 def linear_png():
87     """
88     Source - Channel encoder - Channel - Channel decoder - Destination
89
90     png      - (7,4) linear      - bsc      - (7,4) linear      - png
91     """
92     logger.info("***PNG***")
93     src = source.Source()
94     chl = channel.Channel()
95     linear_code = channel.Linear_Code()
96     dest = destination.Destination()
97
98     height, width, channels = src.read_png("Resource/image.png")
99     tx_msg = src.get_digital_data()
100
101    tx_codeword = linear_code.encoder_systematic(tx_msg)
102
103    rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
104
105    # Statistic analysis
106    correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
107        rx_codeword, 0, 7)
108    one_error_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
109        rx_codeword, 1, 7)
110    total_codewords = len(tx_codeword) // 7
111    uncorrectable_codewords = total_codewords - correct_codewords -
112        one_error_codewords
113    logger.info("Channel_statistic_analysis:")
114    logger.info("Total_codewords: %d", total_codewords)
115    logger.info("Correct_codewords: %d", correct_codewords)
116    logger.info("Codeword_error_rate: %f", (total_codewords -
117        correct_codewords) / total_codewords)
118    logger.info("One_error_codewords: %d", one_error_codewords)

```

```

115 logger.info("----Uncorrectable_codewords:%d", uncorrectable_codewords)
116
117 # without error correction
118 rx_msg = linear_code.decoder_systematic(rx_codeword)
119 dest.set_digital_data(rx_msg)
120 dest.write_png_from_digital("Result/Linear/linear-bsc-output.png", height,
121 width, channels)
122
123 # Statistic analysis
124 correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
125 logger.info("Before_correction:")
126 logger.info("Number_of_correct_bits:%d", correct_bits)
127 logger.info("Number_of_incorrect_bits:%d", len(tx_msg) - correct_bits)
128 logger.info("Bit_error_rate:%f", (len(tx_msg) - correct_bits) / len(
129 tx_msg))
130
131 # with error correction
132 estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
133 rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
134 dest.set_digital_data(rx_msg)
135 dest.write_png_from_digital("Result/Linear/linear-bsc-output-syndrome-
136 corrected.png", height, width, channels)
137
138 # Statistic analysis
139 correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
140 logger.info("After_correction:")
141 logger.info("Number_of_correct_bits:%d", correct_bits)
142 logger.info("Number_of_incorrect_bits:%d", len(tx_msg) - correct_bits)
143 logger.info("Bit_error_rate:%f", (len(tx_msg) - correct_bits) / len(
144 tx_msg))
145
146 def linear_wav():
147     """
148         Source - Channel encoder - Channel - Channel decoder - Destination
149
150         wav - (7,4) linear - bsc - (7,4) linear - wav
151     """
152     logger.info("***WAV***")
153     src = source.Source()
154     chl = channel.Channel()
155     linear_code = channel.Linear_Code()
156     dest = destination.Destination()
157
158     shape, sample_rate = src.read_wav("Resource/file_example_WAV_1MG.wav")
159     plot_wav.plot_wav_time_domain(src.get_analogue_data(), sample_rate, "
160                                     Result/Linear/wav-time-domain-TX.png")
161     plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, "
162                                     Result/Linear/wav-frequency-domain-TX.png")
163
164     # tx_msg = src.get_analogue_data()
165     # rx_msg = tx_msg
166     # dest.set_analogue_data(rx_msg)

```

```

163 # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
164     "Result/Linear/wav-time-domain-RX.png")
165 # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
166     , "Result/Linear/wav-frequency-domain-RX.png")
167 # dest.write_wav_from_analogue(sample_rate, "Result/Linear/hamming-bsc-
168     output.wav")
169
170 tx_msg = src.get_digital_data()
171
172 tx_codeword = linear_code.encoder_systematic(tx_msg)
173
174 rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
175
176 # Statistic analysis
177 correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
178     rx_codeword, 0, 7)
179 one_error_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
180     rx_codeword, 1, 7)
181 total_codewords = len(tx_codeword) // 7
182 uncorrectable_codewords = total_codewords - correct_codewords -
183     one_error_codewords
184 logger.info("Channel statistic analysis:")
185 logger.info("Total codewords: %d", total_codewords)
186 logger.info("Correct codewords: %d", correct_codewords)
187 logger.info("Codeword error rate: %f", (total_codewords -
188     correct_codewords) / total_codewords)
189 logger.info("One error codewords: %d", one_error_codewords)
190 logger.info("Uncorrectable codewords: %d", uncorrectable_codewords)
191
192 # without error correction
193 rx_msg = linear_code.decoder_systematic(rx_codeword)
194
195 dest.set_digital_data(rx_msg)
196 dest.write_wav_from_digital(shape, sample_rate, "Result/Linear/linear-bsc-
197     output.wav")
198
199 plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
200     Result/Linear/linear-bsc-wav-time-domain-RX.png")
201 plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
202     "Result/Linear/linear-bsc-wav-frequency-domain-RX.png")
203
204 # Statistic analysis
205 correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
206 logger.info("Before correction:")
207 logger.info("Number of correct bits: %d", correct_bits)
208 logger.info("Number of incorrect bits: %d", len(tx_msg) - correct_bits)
209 logger.info("Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
210     tx_msg))
211
212 # with error correction
213 estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
214 rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
215

```

```

206 dest.set_digital_data(rx_msg)
207 dest.write_wav_from_digital(shape, sample_rate, "Result/Linear/linear-bsc-
208 output-syndrome-corrected.wav")
209 plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
210     Result/Linear/linear-bsc-wav-time-domain-RX-syndrome-corrected.png")
211 plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
212     "Result/Linear/linear-bsc-wav-frequency-domain-RX-syndrome-corrected.
213     png")
214
215 # Statistic analysis
216 correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
217 logger.info("After correction:")
218 logger.info("Number of correct bits: %d", correct_bits)
219 logger.info("Number of incorrect bits: %d", len(tx_msg) - correct_bits)
220 logger.info("Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
221     tx_msg))
222
223 if __name__ == '__main__':
224     linear_txt()
225     linear_png()
226     linear_wav()

```

6.6 Cyclic code

```

1 # Copyright (c) 2023 Chenye Yang
2
3 import logging
4 import os
5
6 import source
7 import channel
8 import destination
9 from Utils import plot_wav, stat_analysis
10
11
12 # Work with (3, 1) (7, 4) (15, 11) (31, 26) (63, 57) (127, 120) (255, 247)
13 # (511, 502) cyclic hamming code (t=1)
14 # Syndrome look-up table corrector | trapping corrector
15 # N, K = 127, 120
16 # FLAG_SYNDROME = True
17 # FLAG_TRAPPING = False
18
19 # Work with (15, 11, 1) (15, 7, 2) (15, 5, 3) (31, 26, 1) (31, 21, 2) (31, 16,
20 # 3) (31, 11, 5) (31, 6, 7) cyclic code
21 # Trapping corrector
22 N, K = 31, 6
23 FLAG_SYNDROME = False
24 FLAG_TRAPPING = True

```

```

25
26 # Check if the directory exists
27 if not os.path.exists(f'Result/Cyclic/{N}-{K}/'):
28     os.makedirs(f'Result/Cyclic/{N}-{K}/')
29
30 # Configure the logging
31 logging.basicConfig(filename=f'Result/Cyclic/{N}-{K}/logfile-cyclic.log',
32                     filemode='a', # Append the file
33                     level=logging.INFO,
34                     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
35
36 # Create a logger in the main module
37 logger = logging.getLogger(__name__)
38
39
40
41 def cyclic_txt():
42     """
43         Source - Channel encoder - Channel - Channel decoder - Destination
44
45         txt      - (n, k) cyclic      - bsc      - (n, k) cyclic      - txt
46     """
47     logger.info("***TXT***")
48     src = source.Source()
49     chl = channel.Channel()
50     cyclic_code = channel.Cyclic_Code(N, K, None)
51     dest = destination.Destination()
52
53
54     src.read_txt("Resource/hardcoded.txt")
55     tx_msg = src.get_digital_data()
56     padding_length = (-len(tx_msg)) % K
57
58     tx_codeword = cyclic_code.encoder_systematic(tx_msg)
59
60     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
61
62 # Statistic analysis
63     correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
64         rx_codeword, 0, N)
65     nECC = cyclic_code.nECC
66     multiple_error_codewords = []
67     for i in range(1, nECC+1):
68         multiple_error_codewords.append(stat_analysis.
69             num_codeword_with_t_errors(tx_codeword, rx_codeword, i, N))
70     total_codewords = len(tx_codeword) // N
71     uncorrectable_codewords = total_codewords - correct_codewords - sum(
72         multiple_error_codewords)
73     logger.info("Channel statistic analysis:")
74     logger.info("Total codewords: %d", total_codewords)
75     logger.info("Correct codewords: %d", correct_codewords)
76     logger.info("Codeword error rate: %f", (total_codewords -
77         correct_codewords) / total_codewords)

```

```

74     for i in range(1, nECC+1):
75         logger.info("----%d_error_codewords:%d", i, multiple_error_codewords[
76             i-1])
77         logger.info("----Uncorrectable_codewords:%d", uncorrectable_codewords)
78
79     # without error correction
80     rx_msg = cyclic_code.decoder_systematic(rx_codeword, padding_length)
81     dest.set_digital_data(rx_msg)
82     dest.write_txt(f"Result/Cyclic/{N}-{K}/cyclic-bsc-output.txt")
83
84     # Statistic analysis
85     correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
86     logger.info("Before_correction:")
87     logger.info("Number_of_correct_bits:%d", correct_bits)
88     logger.info("Number_of_incorrect_bits:%d", len(tx_msg) - correct_bits)
89     logger.info("Bit_error_rate:%f", (len(tx_msg) - correct_bits) / len(
90         tx_msg))
91
92     # with error correction
93     if FLAG_SYNDROME:
94         estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
95     elif FLAG_TRAPPING:
96         estimated_tx_codeword = cyclic_code.corrector_trapping(rx_codeword)
97     rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword,
98                                             padding_length)
99     dest.set_digital_data(rx_msg)
100    if FLAG_SYNDROME:
101        dest.write_txt(f"Result/Cyclic/{N}-{K}/cyclic-bsc-output-syndrome-
102                     corrected.txt")
103    elif FLAG_TRAPPING:
104        dest.write_txt(f"Result/Cyclic/{N}-{K}/cyclic-bsc-output-trapping-
105                     corrected.txt")
106
107    # Statistic analysis
108    correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
109    if FLAG_SYNDROME:
110        logger.info("After_correction_(syndrome):")
111    elif FLAG_TRAPPING:
112        logger.info("After_correction_(trapping):")
113    logger.info("Number_of_correct_bits:%d", correct_bits)
114    logger.info("Number_of_incorrect_bits:%d", len(tx_msg) - correct_bits)
115    logger.info("Bit_error_rate:%f", (len(tx_msg) - correct_bits) / len(
116        tx_msg))
117
118
119    def cyclic_png():
120        """
121            Source - Channel encoder - Channel - Channel decoder - Destination
122
123            png      - (n,k) cyclic      - bsc      - (n,k) cyclic      - png
124        """
125        logger.info("***PNG***")
126        src = source.Source()
127        chl = channel.Channel()

```

```

122 cyclic_code = channel.Cyclic_Code(N, K, None)
123 dest = destination.Destination()
124
125
126 height, width, channels = src.read_png("Resource/image.png")
127 tx_msg = src.get_digital_data()
128 padding_length = (- len(tx_msg)) % K
129
130 tx_codeword = cyclic_code.encoder_systematic(tx_msg)
131
132 rx_codeword = ch1.binary_symmetric_channel(tx_codeword, 0.01)
133
134 # Statistic analysis
135 correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
136     rx_codeword, 0, N)
137 nECC = cyclic_code.nECC
138 multiple_error_codewords = []
139 for i in range(1, nECC+1):
140     multiple_error_codewords.append(stat_analysis.
141         num_codeword_with_t_errors(tx_codeword, rx_codeword, i, N))
142 total_codewords = len(tx_codeword) // N
143 uncorrectable_codewords = total_codewords - correct_codewords - sum(
144     multiple_error_codewords)
145 logger.info("Channel statistic analysis:")
146 logger.info("Total codewords: %d", total_codewords)
147 logger.info("Correct codewords: %d", correct_codewords)
148 logger.info("Codeword error rate: %f", (total_codewords -
149     correct_codewords) / total_codewords)
150 for i in range(1, nECC+1):
151     logger.info("%%d error codewords: %d", i, multiple_error_codewords[
152         i-1])
153 logger.info("Uncorrectable codewords: %d", uncorrectable_codewords)
154
155 # without error correction
156 rx_msg = cyclic_code.decoder_systematic(rx_codeword, padding_length)
157 dest.set_digital_data(rx_msg)
158 dest.write_png_from_digital(f"Result/Cyclic/{N}-{K}/cyclic-bsc-output.png"
159     , height, width, channels)
160
161 # Statistic analysis
162 correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
163 logger.info("Before correction:")
164 logger.info("Number of correct bits: %d", correct_bits)
165 logger.info("Number of incorrect bits: %d", len(tx_msg) - correct_bits)
166 logger.info("Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
167     tx_msg))
168
169 # with error correction
170 if FLAG.SYNDROME:
171     estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
172 elif FLAG.TRAPPING:
173     estimated_tx_codeword = cyclic_code.corrector_trapping(rx_codeword)
174 rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword,
175     padding_length)

```

```

168 dest.set_digital_data(rx_msg)
169 if FLAG_SYNDROME:
170     dest.write_png_from_digital(f"Result/Cyclic/{N}-{K}/cyclic-bsc-output-
171         syndrome-corrected.png", height, width, channels)
172 elif FLAG_TRAPPING:
173     dest.write_png_from_digital(f"Result/Cyclic/{N}-{K}/cyclic-bsc-output-
174         trapping-corrected.png", height, width, channels)
175
176 # Statistic analysis
177 correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
178 if FLAG_SYNDROME:
179     logger.info("After_correction_(syndrome):")
180 elif FLAG_TRAPPING:
181     logger.info("After_correction_(trapping):")
182 logger.info("Number_of_correct_bits: %d", correct_bits)
183 logger.info("Number_of_incorrect_bits: %d", len(tx_msg) - correct_bits)
184 logger.info("Bit_error_rate: %f", (len(tx_msg) - correct_bits) / len(
185     tx_msg))
186
187 def cyclic_wav():
188     """
189     Source - Channel encoder - Channel - Channel decoder - Destination
190
191     wav      - (n, k) cyclic      - bsc      - (n, k) cyclic      - wav
192     """
193     logger.info("***WAV***")
194     src = source.Source()
195     chl = channel.Channel()
196     cyclic_code = channel.Cyclic_Code(N, K, None)
197     dest = destination.Destination()
198
199     shape, sample_rate = src.read_wav("Resource/file-example_WAV_1MG.wav")
200     plot_wav.plot_wav_time_domain(src.get_analogue_data(), sample_rate, f"
201         Result/Cyclic/{N}-{K}/wav-time-domain-TX.png")
202     plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, f"
203         Result/Cyclic/{N}-{K}/wav-frequency-domain-TX.png")
204
205     # tx_msg = src.get_analogue_data()
206     # rx_msg = tx_msg
207     # dest.set_analogue_data(rx_msg)
208     # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
209         Result/wav-time-domain-RX.png")
210     # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
211         , "Result/wav-frequency-domain-RX.png")
212     # dest.write_wav_from_analogue(sample_rate, "Result/hamming-bsc-output.wav
213         ")
214
215     tx_msg = src.get_digital_data()
216     padding_length = (-len(tx_msg)) % K
217
218     tx_codeword = cyclic_code.encoder_systematic(tx_msg)

```

```

214 rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
215
216
217 # Statistic analysis
218 correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
219     rx_codeword, 0, N)
220 nECC = cyclic_code.nECC
221 multiple_error_codewords = []
222 for i in range(1, nECC+1):
223     multiple_error_codewords.append(stat_analysis.
224         num_codeword_with_t_errors(tx_codeword, rx_codeword, i, N))
225 total_codewords = len(tx_codeword) // N
226 uncorrectable_codewords = total_codewords - correct_codewords - sum(
227     multiple_error_codewords)
228 logger.info("Channel statistic analysis:")
229 logger.info("Total codewords: %d", total_codewords)
230 logger.info("Correct codewords: %d", correct_codewords)
231 logger.info("Codeword error rate: %f", (total_codewords -
232     correct_codewords) / total_codewords)
233 for i in range(1, nECC+1):
234     logger.info("%d error codewords: %d", i, multiple_error_codewords[
235         i-1])
236 logger.info("Uncorrectable codewords: %d", uncorrectable_codewords)

# without error correction
237 rx_msg = cyclic_code.decoder_systematic(rx_codeword, padding_length)
238
239 dest.set_digital_data(rx_msg)
240 dest.write_wav_from_digital(shape, sample_rate, f"Result/Cyclic/{N}-{K}/
241     cyclic-bsc-output.wav")
242
243 plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, f"
244     Result/Cyclic/{N}-{K}/cyclic-bsc-wav-time-domain-RX.png")
245 plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
246     f"Result/Cyclic/{N}-{K}/cyclic-bsc-wav-frequency-domain-RX.png")

# Statistic analysis
247 correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
248 logger.info("Before correction:")
249 logger.info("Number of correct bits: %d", correct_bits)
250 logger.info("Number of incorrect bits: %d", len(tx_msg) - correct_bits)
251 logger.info("Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
252     tx_msg))

# with error correction
253 if FLAG_SYNDROME:
254     estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
255 elif FLAG_TRAPPING:
256     estimated_tx_codeword = cyclic_code.corrector_trapping(rx_codeword)
257 rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword,
258     padding_length)

dest.set_digital_data(rx_msg)
259 if FLAG_SYNDROME:

```

```

258     dest.write_wav_from_digital(shape, sample_rate, f"Result/Cyclic/{N}-{K}"
259         }/cyclic-bsc-output-syndrome-corrected.wav")
260     plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, f
261         "Result/Cyclic/{N}-{K}/cyclic-bsc-wav-time-domain-RX-syndrome-
262         corrected.png")
263     plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(),
264         sample_rate, f"Result/Cyclic/{N}-{K}/cyclic-bsc-wav-frequency-
265         domain-RX-syndrome-corrected.png")
266
267     # Statistic analysis
268     correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
269     if FLAG_SYNDROME:
270         logger.info("After_correction_(syndrome):")
271     elif FLAG_TRAPPING:
272         logger.info("After_correction_(trapping):")
273         logger.info("Number_of_correct_bits: %d", correct_bits)
274         logger.info("Number_of_incorrect_bits: %d", len(tx_msg) - correct_bits)
275         logger.info("Bit_error_rate: %f", (len(tx_msg) - correct_bits) / len(
276             tx_msg))
277
278     if __name__ == '__main__':
279         cyclic_txt()
280         cyclic_png()
281         cyclic_wav()

```