EEC269A - Error Correcting Codes I
Project Report

Chenye Yang, Pranav Kharche, Parisa Oftadeh

June 10, 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 **7**
3.1 Binary Symmetric Channel (BSC) . . . 7
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 (Hamming) Code **8**
5.1 Adjustable (n, k) . . . 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 LFSR Decoder . . . 12

6 Appendix: Python source code **13**
6.1 Source . . . 13
6.2 Channel . . . 18
6.3 Destination . . . 23
6.4 Utilities . . . 27
6.5 Linear code . . . 35
6.6 Cyclic code . . . 40

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	Chenye
	(n,k) Systematic Cyclic (Hamming) Code	Pranav, Chenye
Channel	Binary Symmetric Channel (BSC), error probability p adjustable	Chenye
	AWGN	Parisa
Error Corrector	Syndrome Lookup Table for $(7,4)$ Linear Code	Chenye
	Syndrome Lookup Table for (n,k) Cyclic Code	Chenye
	LFSR for (n,k) Cyclic Code	Pranav
Decoder	$(7,4)$ Systematic Linear Block (Hamming) Code	Chenye
	(n,k) Systematic Cyclic (Hamming) Code	Chenye
Destination	Text string (TXT), Image (PNG), Audio (WAV)	Chenye
Advanced features	**Create generator matrix for** (n,k) **cyclic code**	Pranav
	Adjustable (n,k)	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

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
...	...	...	...

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.:

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:

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

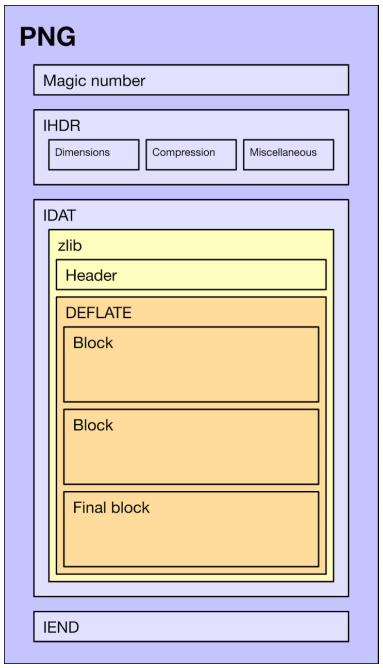


Figure 1: PNG file structure

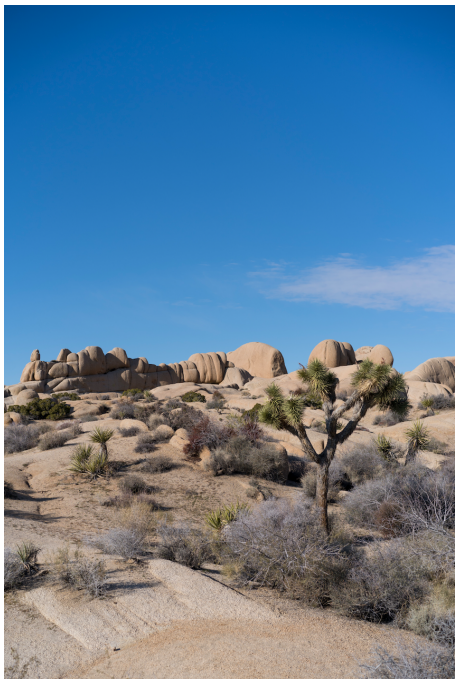


Figure 2: Our testing PNG image

- 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.
- 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 image[2], 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 3 [3]. 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]The image file used in this project is photographed by *Chenye Yang*. The material is free from any copyright restrictions and can be used without any potential legal implications.

[3]Figure 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

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.

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.

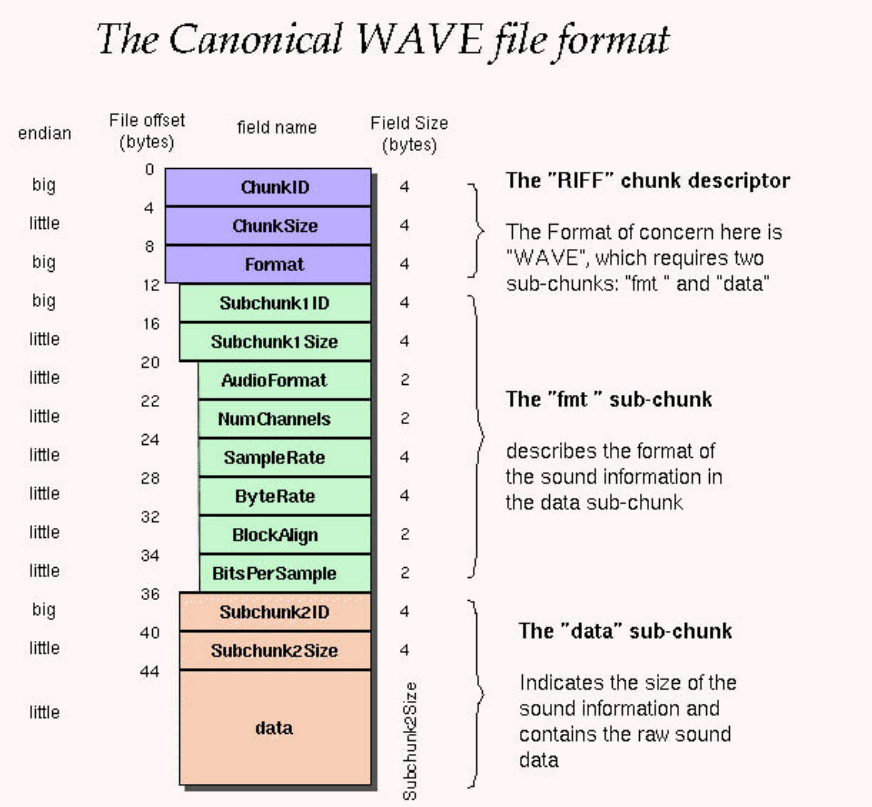


Figure 3: Canonical WAV file structure

For example, as for our testing audio file[4], 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.

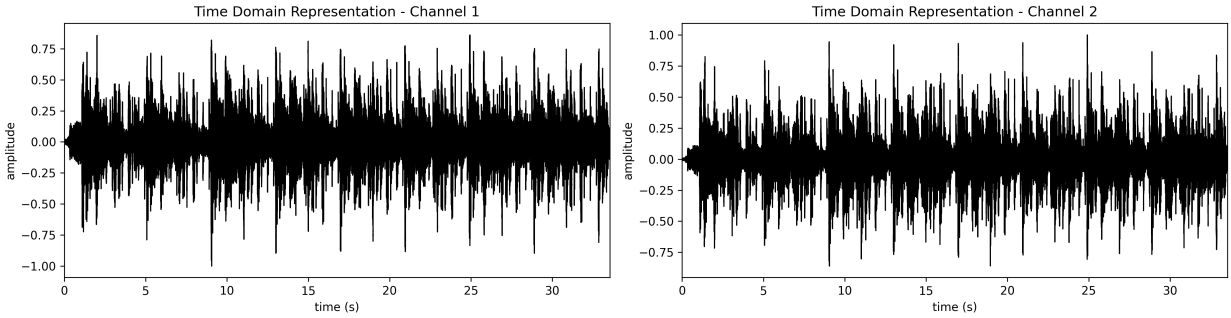


Figure 4: Time domain waveform of our testing audio file

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.

[4]The 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.

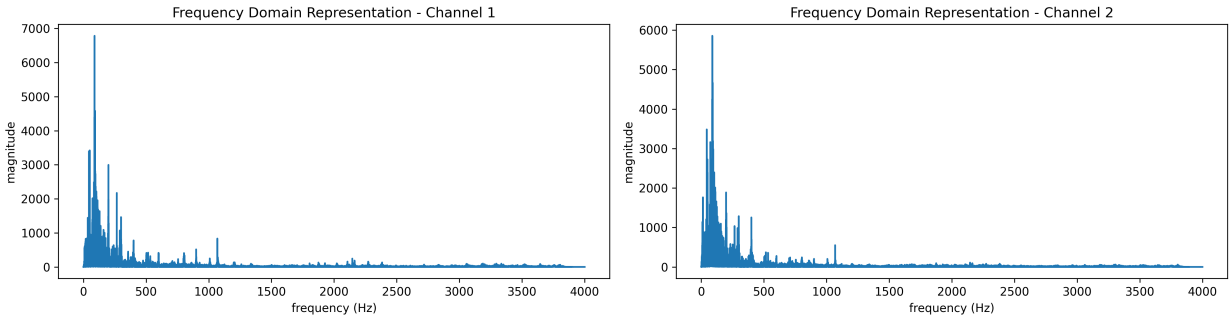


Figure 5: Frequency domain waveform of our testing audio file

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.

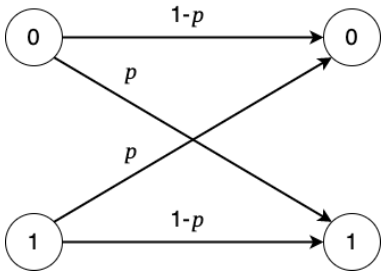


Figure 6: Binary Symmetric Channel (BSC)

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.

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

$$u = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 \end{bmatrix},$$

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

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 (Hamming) Code

The (n, k) Systematic Cyclic Hamming 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

(a) Original (b) Without correction (c) Corrected

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

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.

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.

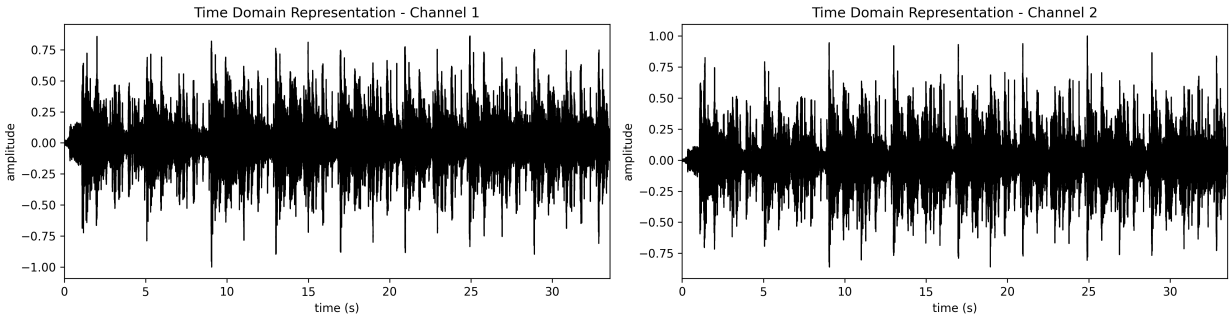
"Cyclic" codes, meanwhile, have a key property where a cyclic shift of a code word results in another valid code word. This cyclic nature leads to efficient implementation, especially in hardware, as simple shift registers can be used.

Hamming codes, named after Richard Hamming, are a specific class of linear error-correcting codes that are renowned for their ability to correct single-bit errors in code words. They accomplish this by introducing additional parity bits that allow the identification of single-bit errors.

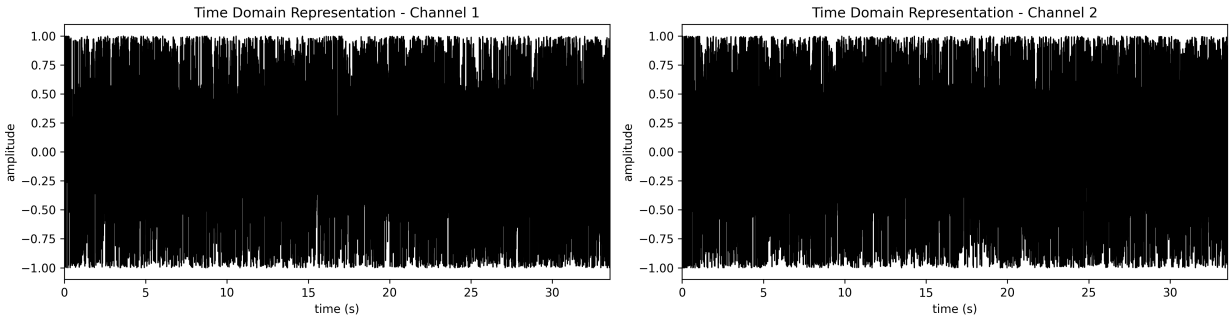
In the context of (n, k) Systematic Cyclic Hamming Codes, the codes are designed such that any single-bit error can be detected and corrected, making them a reliable choice for environments where such errors

(a) Original (b) Without correction (c) Corrected

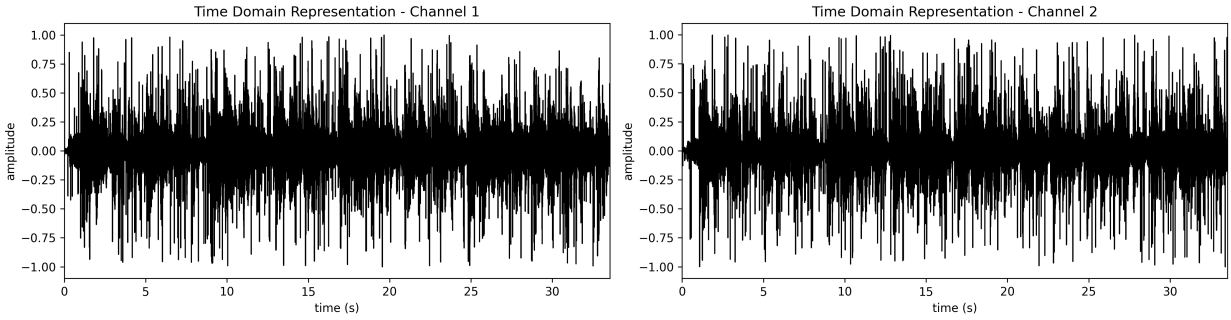
Figure 8: Image encoded with Linear Hamming passed through BSC (details)



(a) Original



(b) Without correction



(c) Corrected

Figure 9: Audio encoded with Linear Hamming passed through BSC

are common and the integrity of information is paramount.

5.1 Adjustable (n, k)

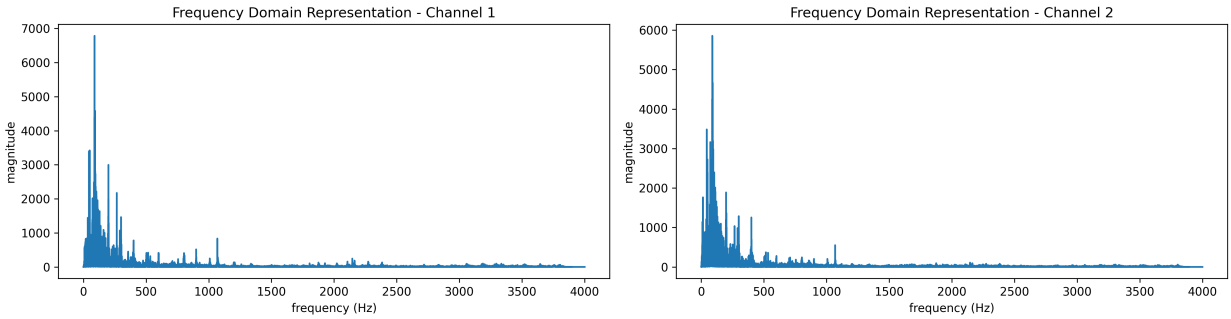
TODO

5.2 Encoder

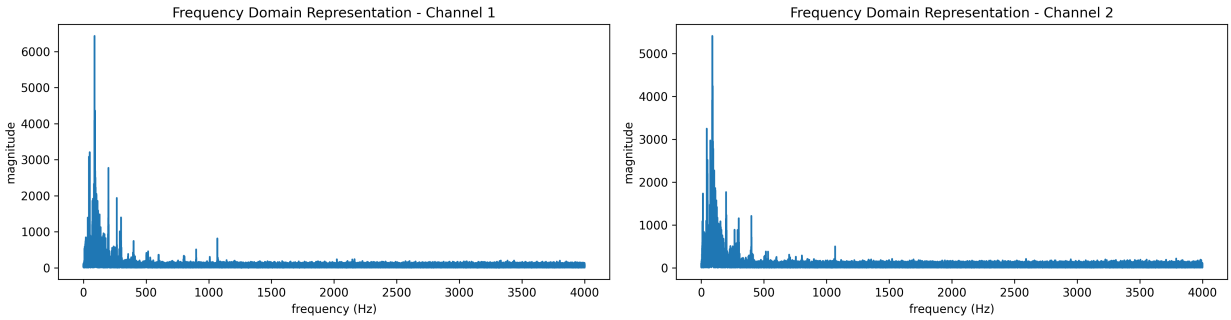
The encoding process in a (n, k) Systematic Cyclic Code involves the multiplication of a k-bit message vector with a generator matrix. The generator matrix for a (n, k) Cyclic Hamming Code, in its systematic form, is given as:

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

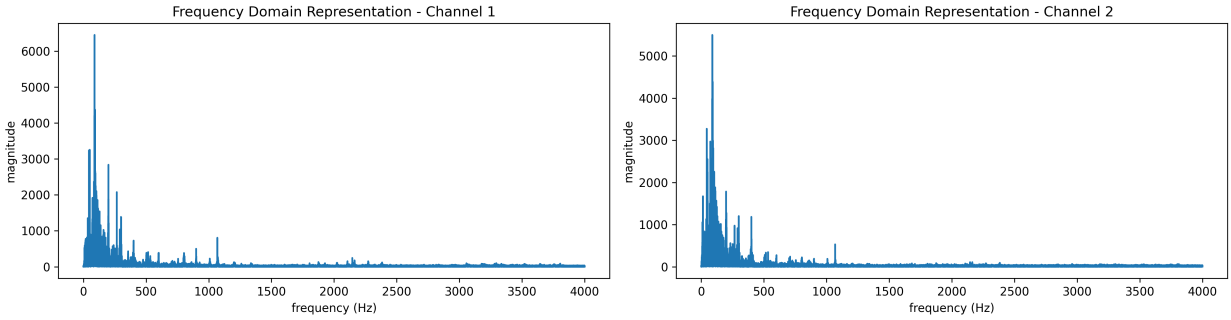
where I_k represents a k by k identity matrix, and P represents a k by $(n - k)$ matrix for parity-check bits. Following is an example of the generator matrix for a $(15, 11)$ Cyclic Hamming Code, which is generated



(a) Original



(b) Without correction



(c) Corrected

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

$$u = \begin{bmatrix} u_1 & u_2 & \dots & u_k \end{bmatrix},$$

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

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	Hu,lo World! EEC269A Error Correctifg Code Demo	Hello World! EEC269A Error Correcting Code Demo

5.3.2 Image

(a) Original (b) Without correction (c) Corrected

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

5.3.3 Audio

5.4 LFSR Decoder

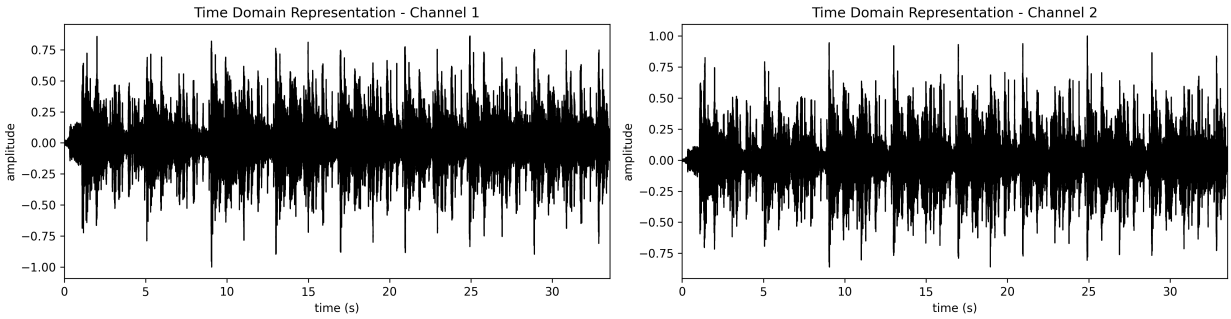
(a) Original (b) Without correction (c) Corrected

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

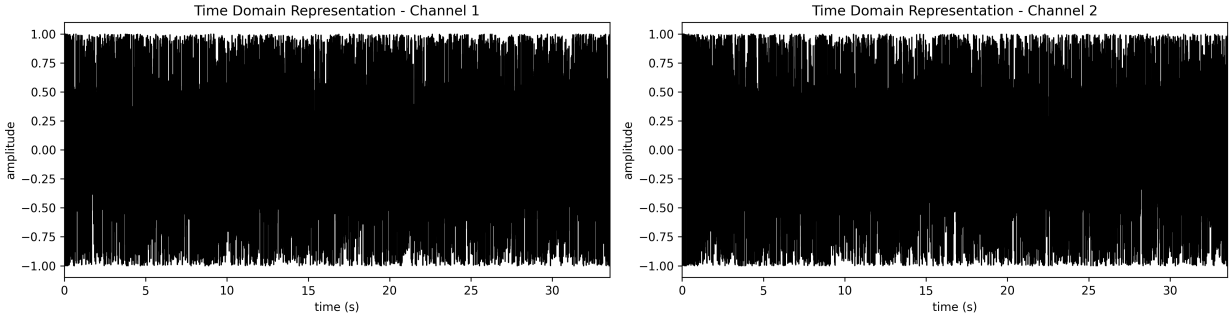
6 Appendix: Python source code

6.1 Source

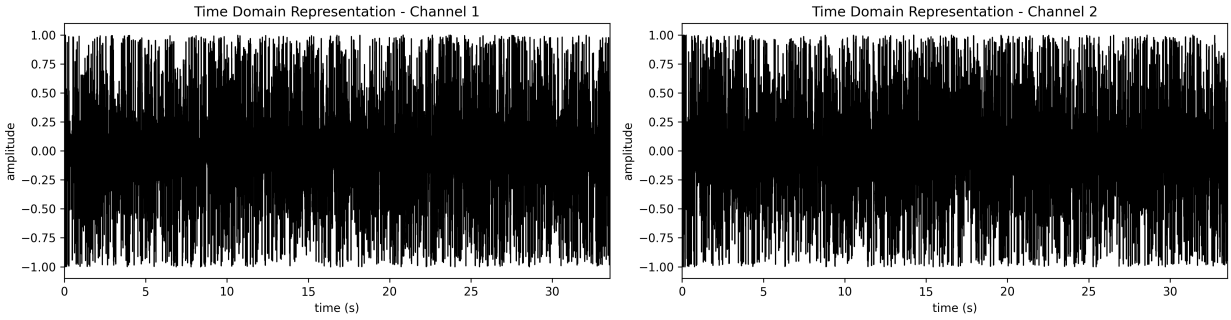
```
# Copyright (c) 2023 Chenye Yang

import numpy as np
from PIL import Image
import soundfile as sf
# from scipy.io import wavfile
import logging

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


class Source:
    """
    The data source
    """

    def read_txt(self, src_path):
        """
        Read the text file as binary bits

            @type  src_path: string
            @param src_path: source file path, with extension
        """
        with open(src_path, 'rb') as file:
            byte_array = bytearray(file.read())
            self._digital_data = np.unpackbits(np.frombuffer(byte_array, dtype
                =np.uint8))


    def read_png(self, src_path):
        """
```



(a) Original



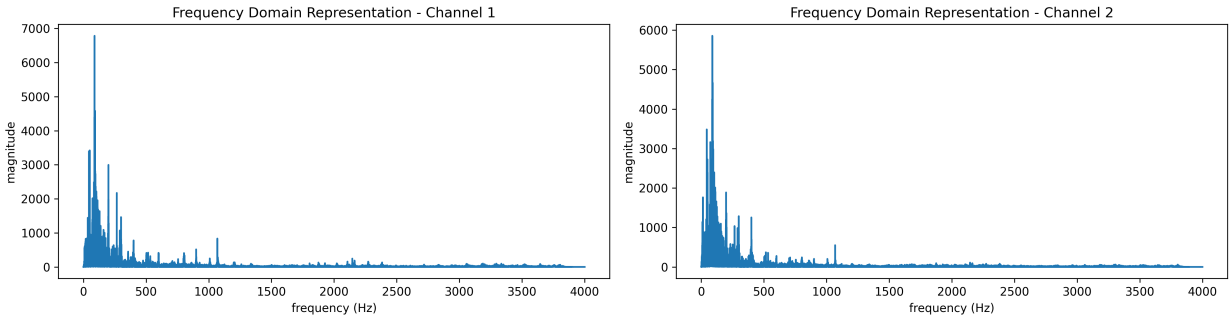
(b) Without correction



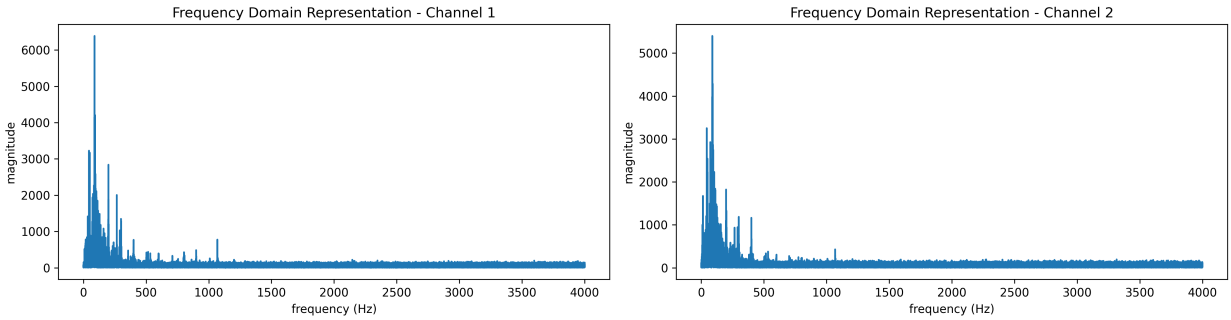
(c) Corrected

Figure 13: Audio encoded with Cyclic Hamming passed through BSC

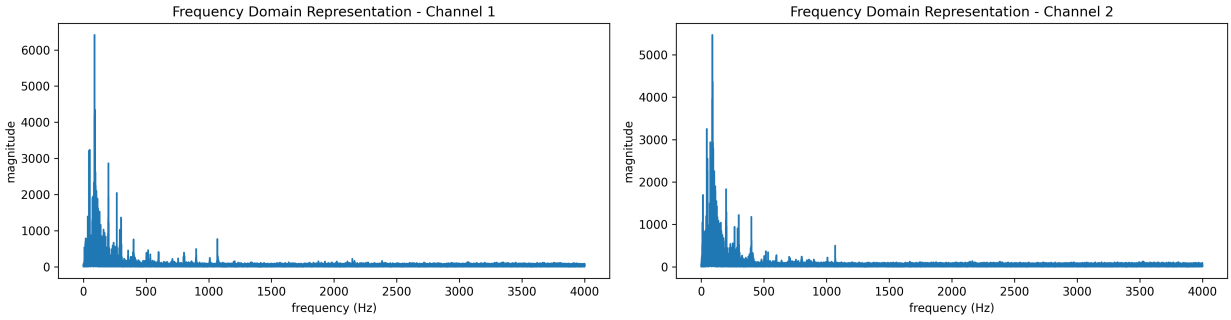
```
        Read the png file,
        store the color information as binary bits in _digital_data,
        store the color information as ndarray in _analogue_data.
        Return the height, width, channels of the image.

            @type   src_path: string
            @param src_path: source file path, with extension

            @rtype:    tuple
            @return:   height, width, channels
        """
        # Open the image file
        image = Image.open(src_path)

        # Convert the image to a NumPy array
```



(a) Original



(b) Without correction



(c) Corrected

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          # Open the audio file
77          # sample_rate, audio_array = wavfile.read(src_path)
78          audio_array, sample_rate = sf.read(src_path, dtype='int16')
79          shape = audio_array.shape
80
81          # Convert the audio_array to binary format and flatten the array
82          bits = np.unpackbits(audio_array.astype(np.int16).view(np.uint8))
83
84          # Store the binary bits
85          self._digital_data = bits
86          # Store the analogue data
87          self._analogue_data = audio_array
88
89          return shape, sample_rate
90
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.
        sample_width, audio.channels
108     #
109     #     # Convert the audio to a NumPy array
110     #     audio_array = np.array(audio.get_array_of_samples())
111     #     # print(audio_array.dtype)
112     #
113     #     # Get the shape of the array: e.g. (2392270,)
114     #     # 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         return self._digital_data
136
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         return self._analogue_data
146
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/
        file_example_MP3_1MG.mp3")
163     # print(frame_rate, sample_width, channels)
164     # audio_bits = source.get_digital_data()
165     # print(audio_bits[:16])
166     # audio_array = source.get_analogue_data()
167     # print(audio_array[:2])
```

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

6.2 Channel

```
1   # Copyright (c) 2023 Chenye Yang
2
3   import numpy as np
4   import logging
5
6   from Utils import polyTools
7
8   # Create a logger in this module
9   logger = logging.getLogger(__name__)
10
11
12
13  def create_parity_check_matrix(G):
14      """
15      Create the parity-check matrix, H = [I_{n-k} | P.T]
16
17          @type  G: ndarray
18          @param G: generator matrix in systematic form, G = [P | I_k]
19
20          @rtype:   ndarray
21          @return:  parity-check matrix
22      """
23      # Get the size of the generator matrix
24      k, n = G.shape
25
26      # Create the parity-check matrix
27      # G = [P | I_k], Extract P
28      P = G[:, :n-k]
29
30      # H = [I_{n-k} | P.T]
31      H = np.hstack((np.eye(n-k, dtype=np.uint8), P.T))
32
33      return H
34
35
36  def create_syndrome_table(H):
37      """
38      Create a table of all possible syndromes and their corresponding error
            vectors (syndrome look-up table)
39
40          @type  H: ndarray
41          @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
                vectors (syndrome look-up table)
47          coset_leader = np.vstack((np.zeros(n, dtype=np.uint8), np.eye(n, dtype=np.
                uint8)))
48          possible_syndromes = (coset_leader @ H.T) % 2
49          syndrome_table = {tuple(possible_syndromes[i]) : coset_leader[i] for i in
                range(n+1)}
50
51          return syndrome_table
52
53
54      def pad_bits(bits, k):
55          """
56          Pad the bits array with zeroes so its length is divisible by k.
57
58              @type   bits: ndarray
59              @param bits: TX message
60
61              @type   k: int
62              @param k: number of bits per codeword
63
64          """
65          padded_bits = bits.copy()
66          remainder = len(bits) % k
67          if remainder != 0:
68              padding_length = k - remainder
69              padded_bits = np.pad(padded_bits, (0, padding_length), mode='constant'
                    )
70          return padded_bits
71
72
73      def remove_padding(bits, padding_length):
74          """
75          Remove the padding from the array.
76
77              @type   bits: ndarray
78              @param bits: RX message
79
80              @type   padding_length: int
81              @param padding_length: length of the original TX message
82          """
83          return bits[:-padding_length] if padding_length != 0 else bits
84
85
86
87      class Linear_Code:
88          """
89          (7, 4) Systematic Linear Block (Hamming) Code
90          """
91          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
                the result to 1D array
129         encoded_array = np.dot(messages, self.G) % 2
130
131         # Flatten the array
132         encoded_array = encoded_array.flatten()
133
134         return encoded_array
135
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
                    padding)
146
147             @rtype:    ndarray
148             @return:  RX message
149         """
150         # Reshape the array so each row is a codeword
151         reshaped_array = encoded_array.reshape(-1, self.n)
152
153         # Compute the syndrome for each codeword
154         syndromes = np.dot(reshaped_array, self.H.T) % 2
155
156         # Count the number of non-zero syndromes (errors)
157         err_count = np.count_nonzero(np.any(syndromes, axis=1))
158
159         # Decode by taking the last self.k elements from each codeword
160         decoded_array = reshaped_array[:, self.n - self.k:self.n].flatten()
161
162         logger.debug(f"Error codeword rate: {err_count / (len(encoded_array)
                // self.n)}")
163
164         # Remove the padding from the array
165         if padding_length != 0:
166             decoded_array = remove_padding(decoded_array, padding_length)
167
168         return decoded_array
169
170
171     def corrector_syndrome(self, received_array):
172         """
173         Systematic - Correct the received binary bits codeword with (n,k)
                Hamming syndrome look-up table corrector,
174         return the estimated TX codeword = (RX codeword + error pattern)
175
176             @type  received_array: ndarray
177             @param received_array: RX codewords
178
179             @rtype:    ndarray
180             @return:  estimated TX codewords
181         """
182         # Reshape the received_array so each row is a codeword
183         reshaped_array = received_array.reshape(-1, self.n)
184
185         # Compute the syndrome for each codeword
186         syndromes = np.dot(reshaped_array, self.H.T) % 2
187
188         # Copy reshaped_array to corrected_array for correction
189         corrected_array = reshaped_array.copy()
190
191         # Loop over the syndromes
192         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[
                            tuple_syndrome]) % 2
197 
198             # Flatten corrected_array to match the shape of the input
                    received_array
199             corrected_array = corrected_array.flatten()
200 
201             return corrected_array
202 
203 
204 
205 class Cyclic_Code(Linear_Code):
206     """
207     (n, k) Systematic Cyclic (Hamming) Code
208     """
209     def __init__(self, G):
210         self.G = G
211         self.k, self.n = self.G.shape
212         self.H = create_parity_check_matrix(self.G)
213 
214         self.syndrome_table = create_syndrome_table(self.H)
215 
216 
217     def corrector_lfsr(self, received_array):
218         """
219         Systematic - Correct the received binary bits codeword with (n, k)
                Hamming LFSR corrector,
220         return the estimated TX codeword = (RX codeword + error pattern)
221 
222             @type  received_array: ndarray
223             @param received_array: RX codewords
224 
225             @rtype:   ndarray
226             @return:  estimated TX codewords
227         """
228         corrected_array = received_array.copy()
229         pass
230 
231 
232 class Channel:
233     """
234     Channel
235     """
236     def binary_symmetric_channel(self, input_bits, p):
237         """
238         BSC - binary symmetric channel with adjustable error probability
239 
240             @type  input_bits: ndarray
241             @param input_bits: TX codewords
242 
243             @type  p: float
```

```
            @param p: error_probability

            @rtype:   ndarray
            @return:  RX codewords
        """
        # Generate a uniform random array of the same shape as input_bits
        random_numbers = np.random.rand(*input_bits.shape)

        # Identify where the random array is less than p
        mask = random_numbers < p

        # Use this mask to flip the bits at those indices
        output_bits = np.where(mask, 1 - input_bits, input_bits)

        return output_bits



if __name__ == '__main__':
    # test
    tx_msg = np.array([0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1], dtype=np.uint8)
    n = 7
    k = 4
    print("Original bits:", tx_msg)

    # Channel
    channel = Channel()

    # Linear Code
    linear_code = Linear_Code()

    # Encoding
    tx_codewords = linear_code.encoder_systematic(tx_msg)
    print("Encoded bits       :", tx_codewords)

    # Passing through the channel
    rx_codewords = channel.binary_symmetric_channel(tx_codewords, 0.1)
    print("Bits after channel:", rx_codewords)

    # Correction with syndrome look-up table
    estimated_tx_codewords = linear_code.corrector_syndrome(rx_codewords)
    print("Estimated TX bits :", estimated_tx_codewords)

    # Decoding
    padding_length = k - (len(tx_msg) % k)
    rx_msg = linear_code.decoder_systematic(estimated_tx_codewords,
        padding_length)
    print("Decoded bits :", rx_msg)
```

6.3 Destination

```
# Copyright (c) 2023 Chenye Yang

```

```
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
                store the color information as ndarray in _analogue_data
73
74              @type  dest_path: string
75              @param dest_path: destination file path, with extension
76
77              @type  height: int
78              @param height: height of the image
79
80              @type  width: int
81              @param width: width of the image
82
83              @type  channels: int
84              @param channels: channels of the image
85          """
86          # Convert the bit array to uint8 array
87          uint8_array = np.packbits(self._digital_data)
88
89          # Reshape the array to a 3D array of pixels (height, width, channels)
90          pixels = uint8_array.reshape(height, width, channels)
91
92          # Store the analogue data
93          self._analogue_data = pixels
94
95          # Create a new image from the pixel values
96          new_image = Image.fromarray(pixels)
97
98          # Save the new image to a file
99          new_image.save(dest_path)
100
101
102     def write_png_from_analogue(self, dest_path):
103         """
104         Write color information in (height, width, channels) array to a png
                file, at the same time store the color information as bit array in
                 _digital_data
105
106             @type  dest_path: string
107             @param dest_path: destination file path, with extension
```

```
108             """
109             # Create a new image from the pixel values
110             new_image = Image.fromarray(self._analogue_data)
111 
112             # Save the new image to a file
113             new_image.save(dest_path)
114 
115             # Convert the img_array to binary format and flatten the array
116             bits = np.unpackbits(self._analogue_data.astype(np.uint8))
117 
118             # Store the binary bits
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
                    store the audio information as ndarray in _analogue_data
125 
126                 @type  dest_path: string
127                 @param dest_path: destination file path, with extension
128             """
129             # Convert the bit array to int16 array
130             int16_array = np.packbits(self._digital_data).view(np.int16)
131 
132             # Reshape the array to a 2D array of samples (channels, samples)
133             audio_array = int16_array.reshape(shape)
134 
135             # Store the analogue data
136             self._analogue_data = audio_array
137 
138             # Write the array to a wav file
139             # wavfile.write(dest_path, sample_rate, audio_array)
140             sf.write(dest_path, audio_array, sample_rate)
141 
142 
143         def write_wav_from_analogue(self, sample_rate, dest_path):
144             """
145             Write audio information in audio array to a wav file, at the same time
                    store the audio information as bit array in _digital_data
146 
147                 @type  dest_path: string
148                 @param dest_path: destination file path, with extension
149             """
150             # Write the array to a wav file
151             # wavfile.write(dest_path, sample_rate, self._analogue_data)
152             sf.write(dest_path, self._analogue_data, sample_rate)
153 
154             # Convert the audio_array to binary format and flatten the array
155             bits = np.unpackbits(self._analogue_data.astype(np.int16).view(np.
                    uint8))
156 
157             # Store the binary bits
158             self._digital_data = bits
```

```
159
160
161     # def write_mp3_from_digital(self, frame_rate, sample_width, channels,
            dest_path):
162     #     """
163     #     Write audio information in bits array to a mp3 file
164
165     #         @type  dest_path: string
166     #         @param dest_path: destination file path, with extension
167     #     """
168     #     # 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,
            sample_width, channels)
173
174     #     # Export the modified audio to a new MP3 file
175     #     modified_audio.export(dest_path, format="mp3")
176
177
178     # def write_mp3_from_analogue(self, frame_rate, sample_width, channels,
            dest_path):
179     #     """
180     #     Write audio information in audio array to a mp3 file
181
182     #         @type  dest_path: string
183     #         @param dest_path: destination file path, with extension
184     #     """
185     #     # Create a new AudioSegment object from the modified audio array
186     #     modified_audio = AudioSegment(self._analogue_data.tobytes(),
            frame_rate, sample_width, channels)
187
188     #     # Export the modified audio to a new MP3 file
189     #     modified_audio.export(dest_path, format="mp3")
190
191
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 becase some tasks work better with the highest order
       coeffecient as the least significant bit
31 # if no order is specified, it is found using order(p) from above
32 # returns the reversed polynomial
33 def bitRev(p, pOrder = None):
34         if p == 0:
35                 return 0
36         if pOrder == None:
37                 pOrder = order(p)
38
39         retVal = 0
40         for i in range(pOrder+1):
41                 retVal = (retVal << 1) + (p & 1)
42                 p = p >>1
43         return retVal
44
45 # divide two polynomials
46 # each polynomial must be with the lowest order coeff as the LSB and the
       highest order as the MSB
47 # if order is not provided it is found automatically
48 # returns a tuple of the result and remainder of the division.
49 def polyDiv(dividend, divisor, dividendOrder = None, divisorOrder = None):
50         if dividendOrder == None or divisorOrder == None:
51                 dividendOrder = order(dividend)
52                 divisorOrder = order(divisor)
53
54         sizeDiff = dividendOrder - divisorOrder
55         rem = dividend
56         remOrder = dividendOrder
57         result = 0
58         for i in range(sizeDiff, -1, -1):
```

```
59                  if(rem >> remOrder):
60                          rem = rem ^ (divisor << i)
61                          result = result + (1 << i)
62                  remOrder -= 1
63
64          # if rem:
65          #       print(f'{dividend:b}',' : ',  f'{divisor:b}', ' = ', f'{result
                :b}',' R', f'{rem:0{divisorOrder}b}', sep='')
66          # else:
67          #       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
        for a compatible polynomial
72  # returns None if polynomial cannot be found.
73  # n, k: code length and information length of the code
74  # findN: find the nth solution. function normally stops upon finding the first
        solution that matches n and k. However, this is not always optimal.
75  #                 Callers of this function may find that the 2nd or 3rd
        solution works better.
76  def findGen(n,k, initialGen = None):
77          target = (1 << n) + 1
78          if initialGen:
79                  gen = initialGen
80          else:
81                  gen = (1 << (n-k)) + 1
82          maxGen = (1 << (n-k+1))
83          rem = 1
84          while rem:
85                  gen += 2
86                  if gen >= maxGen:
87                          return None
88                  parity, rem = polyDiv(target, gen, n, n-k)
89          logger.debug('generator = %s', f'{gen:b}')
90          logger.debug('parity    = %s', f'{parity:b}')
91          return gen
92
93  # build a generator matrix for a systematic cyclic code
94  # n, k: code length and information length of the code
95  # genPoly: the generator polynomial to be used
96  def buildGenMatrix(n, k, genPoly):
97          genMatrix = [genPoly]
98          # print(f'{genPoly:0{n}b}')
99          for i in range(k-1):
100                 nextLine = genMatrix[i] << 1
101                 if (nextLine >> (n-k))%2:
102                         nextLine = nextLine ^ genPoly
103                 genMatrix.append(nextLine)
104                 # print(f'{nextLine:0{n}b}')
105         logger.debug('Generator matrix')
106         for i in range(k):
107                 genMatrix[i] = bitRev(genMatrix[i], n-1)
```

```
108                     logger.debug(f'{genMatrix[i]:0{n}b}')
109             return genMatrix
110
111     # build the transpose of the parity matrix using the generator matrix
112     # n, k: code length and information length of the code
113     # genMatrix: the generator matrix to be used
114     def buildParityMatrix(n, k, genMatrix):
115             Ht = []
116             m = n-k
117             logger.debug('Parity Matrix Transpose')
118             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
            requirements provided by the input
128     # returns None if requirements are impossible to meet.
129     # n, k: code length and information length of the code
130     # nECC: number of correctable errors the code must have
131     #               input of nECC = None means it will search for the best
            possible code.
132     def findMatrix(n,k, nECC = 1):
133             numErrors = 0
134             bestPoly = 0
135             genPoly = 0
136             bestGen = None
137             maxErrors = 0
138             while nECC == None or numErrors < nECC:
139                     genPoly = findGen(n,k,genPoly)
140                     if genPoly == None:
141                             genMatrix = None
142                             break
143                     genMatrix = buildGenMatrix(n, k, genPoly)
144                     numErrors = correctableErrors(n, k, genMatrix)
145                     if numErrors > maxErrors:
146                             bestGen = genMatrix
147                             maxErrors = numErrors
148                             bestPoly = genPoly
149             if (nECC == None and bestGen == None) or (nECC and genMatrix == None):
150                     logger.debug('no viable matrix found')
151                     # print('no viable matrix found')
152                     return None
153             logger.debug('used polynomial %d', bestPoly)
154             logger.debug('%d correctable errors', maxErrors)
155             # print('used polynomial', bestPoly)
156             # print(maxErrors,'correctable errors')
157             return bestGen
158
159     # encode a data word with the provided generator matrix
```

```
160 def encode(data, genMatrix):
161         if order(data)+1 > len(genMatrix):
162                 return None
163         result = 0
164         data = bitRev(data, len(genMatrix)-1)
165         for row in genMatrix:
166                 result = result ^ ((data & 1)*row)
167                 data = data >> 1
168         return result
169
170 # 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
                genMatrix_decimal]
231
232         # Convert the binary generator matrix to a numpy array
233         G = np.array(genMatrix_binary, dtype=np.uint8)
234
235         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     # Copy the audio array
22     data = audio_array.copy()
23     # Normalize to [-1, 1]
24     data = data / np.max(np.abs(data),axis=0)
25
26     times = np.arange(len(data))/float(sample_rate)
27
28     # Prepare the subplots
29     fig, axs = plt.subplots(1, 2, figsize=(15, 4))
```

```
30
31      # Time domain representation for channel 1
32      axs[0].fill_between(times, data[:, 0], color='k')
33      axs[0].set_xlim(times[0], times[-1])
34      axs[0].set_xlabel('time (s)')
35      axs[0].set_ylabel('amplitude')
36      axs[0].set_title('Time Domain Representation - Channel 1')
37
38      # Time domain representation for channel 2
39      axs[1].fill_between(times, data[:, 1], color='k')
40      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      # Copy the audio array
65      data = audio_array.copy()
66      # Normalize to [-1, 1]
67      data = data / np.max(np.abs(data),axis=0)
68
69      # Prepare the subplots
70      fig, axs = plt.subplots(1, 2, figsize=(15, 4))
71
72      # Frequency domain representation for channel 1
73      fft_out_channel_1 = fft(data[:, 0])
74      magnitude_spectrum_channel_1 = np.abs(fft_out_channel_1)
75      frequencies_channel_1 = np.linspace(0, sample_rate, len(
            magnitude_spectrum_channel_1))
76
77      axs[0].plot(frequencies_channel_1[:int(len(frequencies_channel_1)/2)],
            magnitude_spectrum_channel_1[:int(len(magnitude_spectrum_channel_1)/2)
            ]) # plot only first half of frequencies
78      axs[0].set_xlabel('frequency (Hz)')
79      axs[0].set_ylabel('magnitude')
80      axs[0].set_title('Frequency Domain Representation - Channel 1')
```

```
81
82      # Frequency domain representation for channel 2
83      fft_out_channel_2 = fft(data[:, 1])
84      magnitude_spectrum_channel_2 = np.abs(fft_out_channel_2)
85      frequencies_channel_2 = np.linspace(0, sample_rate, len(
            magnitude_spectrum_channel_2))
86
87      axs[1].plot(frequencies_channel_2[:int(len(frequencies_channel_2)/2)],
            magnitude_spectrum_channel_2[:int(len(magnitude_spectrum_channel_2)/2)
            ]) # plot only first half of frequencies
88      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      # Reshape the array so each row is a codeword
27      original_bits = original_bits.reshape(-1, n)
28      corrupted_bits = corrupted_bits.reshape(-1, n)
29
30      # Modulo 2 addition
31      diff = (original_bits + corrupted_bits) % 2
32
33      # Count the number of 1s in each row
34      result = np.count_nonzero(diff, axis=1)
```

```
35
36      # Count the number of rows that contains exactly t 1s
37      return np.count_nonzero(result == t)
38
39
40  def num_correct_messages(original_bits, corrupted_bits, k):
41      """
42      Calculate how many messages in the bit stream are correct.
43
44          @type  original_bits: ndarray
45          @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
5   import source
6   import channel
```

```
7  import destination
8  from Utils import plot_wav, stat_analysis
9
10
11 # Configure the logging
12 logging.basicConfig(filename='Result/logfile-linear.log',
13                     filemode='w', # Overwrite the file
14                     level=logging.INFO,
15                     format='%(asctime)s - %(name)s - %(levelname)s - %(message
                           )s')
16
17 # Create a logger in the main module
18 logger = logging.getLogger(__name__)
19
20
21
22 def linear_txt():
23     """
24     Source - Channel encoder - Channel - Channel decoder - Destination
25
26     txt    - (7,4) linear    - bsc     - (7,4) linear    - txt
27     """
28     logger.info("***TXT***")
29     src = source.Source()
30     chl = channel.Channel()
31     linear_code = channel.Linear_Code()
32     dest = destination.Destination()
33
34
35     src.read_txt("Resource/hardcoded.txt")
36     tx_msg = src.get_digital_data()
37
38     tx_codeword = linear_code.encoder_systematic(tx_msg)
39
40     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
41
42     # Statistic analysis
43     correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
           rx_codeword, 0, 7)
44     one_error_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword
           , rx_codeword, 1, 7)
45     total_codewords = len(tx_codeword) // 7
46     uncorrectable_codewords = total_codewords - correct_codewords -
           one_error_codewords
47     logger.info("Channel statistic analysis:")
48     logger.info("  Total codewords: %d", total_codewords)
49     logger.info("  Correct codewords: %d", correct_codewords)
50     logger.info("  Codeword error rate: %f", (total_codewords -
           correct_codewords) / total_codewords)
51     logger.info("    One error codewords: %d", one_error_codewords)
52     logger.info("    Uncorrectable codewords: %d", uncorrectable_codewords)
53
54     # without error correction
55     rx_msg = linear_code.decoder_systematic(rx_codeword)
```

```
56      dest.set_digital_data(rx_msg)
57      dest.write_txt("Result/linear-bsc-output.txt")
58
59      # Statistic analysis
60      correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
61      logger.info("Before correction:")
62      logger.info("  Number of correct bits: %d", correct_bits)
63      logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
64      logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
            tx_msg))
65
66      # with error correction
67      estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
68      rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
69      dest.set_digital_data(rx_msg)
70      dest.write_txt("Result/linear-bsc-output-syndrome-corrected.txt")
71
72      # Statistic analysis
73      correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
74      logger.info("After correction:")
75      logger.info("  Number of correct bits: %d", correct_bits)
76      logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
77      logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
            tx_msg))
78
79
80  def linear_png():
81      """
82      Source - Channel encoder - Channel - Channel decoder - Destination
83
84      png    - (7,4) linear    - bsc     - (7,4) linear    - png
85      """
86      logger.info("***PNG***")
87      src = source.Source()
88      chl = channel.Channel()
89      linear_code = channel.Linear_Code()
90      dest = destination.Destination()
91
92
93      height, width, channels = src.read_png("Resource/image.png")
94      tx_msg = src.get_digital_data()
95
96      tx_codeword = linear_code.encoder_systematic(tx_msg)
97
98      rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
99
100     # Statistic analysis
101     correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
            rx_codeword, 0, 7)
102     one_error_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword
            , rx_codeword, 1, 7)
103     total_codewords = len(tx_codeword) // 7
104     uncorrectable_codewords = total_codewords - correct_codewords -
            one_error_codewords
```

```
105     logger.info("Channel statistic analysis:")
106     logger.info("  Total codewords: %d", total_codewords)
107     logger.info("  Correct codewords: %d", correct_codewords)
108     logger.info("  Codeword error rate: %f", (total_codewords -
            correct_codewords) / total_codewords)
109     logger.info("    One error codewords: %d", one_error_codewords)
110     logger.info("    Uncorrectable codewords: %d", uncorrectable_codewords)
111
112     # without error correction
113     rx_msg = linear_code.decoder_systematic(rx_codeword)
114     dest.set_digital_data(rx_msg)
115     dest.write_png_from_digital("Result/linear-bsc-output.png", height, width,
            channels)
116
117     # Statistic analysis
118     correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
119     logger.info("Before correction:")
120     logger.info("  Number of correct bits: %d", correct_bits)
121     logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
122     logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
            tx_msg))
123
124     # with error correction
125     estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
126     rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
127     dest.set_digital_data(rx_msg)
128     dest.write_png_from_digital("Result/linear-bsc-output-syndrome-corrected.
            png", height, width, channels)
129
130     # Statistic analysis
131     correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
132     logger.info("After correction:")
133     logger.info("  Number of correct bits: %d", correct_bits)
134     logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
135     logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
            tx_msg))
136
137
138 def linear_wav():
139     """
140     Source - Channel encoder - Channel - Channel decoder - Destination
141
142     wav    - (7,4) linear    - bsc     - (7,4) linear    - wav
143     """
144     logger.info("***WAV***")
145     src = source.Source()
146     chl = channel.Channel()
147     linear_code = channel.Linear_Code()
148     dest = destination.Destination()
149
150
151     shape, sample_rate = src.read_wav("Resource/file_example_WAV_1MG.wav")
152     plot_wav.plot_wav_time_domain(src.get_analogue_data(), sample_rate, "
            Result/wav-time-domain-TX.png")
```

```
153     plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, "
            Result/wav-frequency-domain-TX.png")
154
155     # tx_msg = src.get_analogue_data()
156     # rx_msg = tx_msg
157     # dest.set_analogue_data(rx_msg)
158     # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
            Result/wav-time-domain-RX.png")
159     # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate
            , "Result/wav-frequency-domain-RX.png")
160     # dest.write_wav_from_analogue(sample_rate, "Result/hamming-bsc-output.wav
            ")
161
162
163     tx_msg = src.get_digital_data()
164
165     tx_codeword = linear_code.encoder_systematic(tx_msg)
166
167     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
168
169     # Statistic analysis
170     correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
            rx_codeword, 0, 7)
171     one_error_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword
            , rx_codeword, 1, 7)
172     total_codewords = len(tx_codeword) // 7
173     uncorrectable_codewords = total_codewords - correct_codewords -
            one_error_codewords
174     logger.info("Channel statistic analysis:")
175     logger.info("  Total codewords: %d", total_codewords)
176     logger.info("  Correct codewords: %d", correct_codewords)
177     logger.info("  Codeword error rate: %f", (total_codewords -
            correct_codewords) / total_codewords)
178     logger.info("    One error codewords: %d", one_error_codewords)
179     logger.info("    Uncorrectable codewords: %d", uncorrectable_codewords)
180
181     # without error correction
182     rx_msg = linear_code.decoder_systematic(rx_codeword)
183
184     dest.set_digital_data(rx_msg)
185     dest.write_wav_from_digital(shape, sample_rate, "Result/linear-bsc-output.
            wav")
186
187     plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
            Result/linear-bsc-wav-time-domain-RX.png")
188     plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
            "Result/linear-bsc-wav-frequency-domain-RX.png")
189
190     # Statistic analysis
191     correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
192     logger.info("Before correction:")
193     logger.info("  Number of correct bits: %d", correct_bits)
194     logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
```

```
195     logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
            tx_msg))
196
197     # with error correction
198     estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
199     rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
200
201     dest.set_digital_data(rx_msg)
202     dest.write_wav_from_digital(shape, sample_rate, "Result/linear-bsc-output-
            syndrome-corrected.wav")
203
204     plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
            Result/linear-bsc-wav-time-domain-RX-syndrome-corrected.png")
205     plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
            "Result/linear-bsc-wav-frequency-domain-RX-syndrome-corrected.png")
206
207     # Statistic analysis
208     correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
209     logger.info("After correction:")
210     logger.info("  Number of correct bits: %d", correct_bits)
211     logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
212     logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
            tx_msg))
213
214
215
216 if __name__ == '__main__':
217     linear_txt()
218     linear_png()
219     linear_wav()
```

6.6 Cyclic code

```
1  # Copyright (c) 2023 Chenye Yang
2
3  import logging
4
5  import source
6  import channel
7  import destination
8  from Utils import plot_wav, polyTools, stat_analysis
9
10
11 # Configure the logging
12 logging.basicConfig(filename='Result/logfile-cyclic.log',
13                     filemode='w', # Overwrite the file
14                     level=logging.INFO,
15                     format='%(asctime)s - %(name)s - %(levelname)s - %(message
                           )s')
16
17 # Create a logger in the main module
18 logger = logging.getLogger(__name__)
19
```

```
20
21
22  # Work with (3, 1) (7, 4) (15, 11) (31, 26) (63, 57) (127, 120) (255, 247)
        (511, 502) cyclic hamming code
23  N, K = 31, 26
24
25  # Create the generator matrix
26  genMatrix_decimal = polyTools.findMatrix(N, K)
27  G = polyTools.genMatrixDecmial2Ndarray(genMatrix_decimal, N)
28
29
30  def cyclic_txt():
31      """
32      Source - Channel encoder - Channel - Channel decoder - Destination
33
34      txt    - (n,k) cyclic    - bsc     - (n,k) cyclic    - txt
35      """
36      logger.info("***TXT***")
37      src = source.Source()
38      chl = channel.Channel()
39      cyclic_code = channel.Cyclic_Code(G)
40      dest = destination.Destination()
41
42
43      src.read_txt("Resource/hardcoded.txt")
44      tx_msg = src.get_digital_data()
45      padding_length = K - (len(tx_msg) % K)
46
47      tx_codeword = cyclic_code.encoder_systematic(tx_msg)
48
49      rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
50
51      # Statistic analysis
52      correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
            rx_codeword, 0, N)
53      one_error_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword
            , rx_codeword, 1, N)
54      total_codewords = len(tx_codeword) // N
55      uncorrectable_codewords = total_codewords - correct_codewords -
            one_error_codewords
56      logger.info("Channel statistic analysis:")
57      logger.info("  Total codewords: %d", total_codewords)
58      logger.info("  Correct codewords: %d", correct_codewords)
59      logger.info("  Codeword error rate: %f", (total_codewords -
            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 = cyclic_code.decoder_systematic(rx_codeword, padding_length)
65      dest.set_digital_data(rx_msg)
66      dest.write_txt("Result/cyclic-bsc-output.txt")
67
68      # Statistic analysis
```

```
69      correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
70      logger.info("Before correction:")
71      logger.info("  Number of correct bits: %d", correct_bits)
72      logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
73      logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
            tx_msg))
74
75      # with error correction
76      estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
77      rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword,
            padding_length)
78      dest.set_digital_data(rx_msg)
79      dest.write_txt("Result/cyclic-bsc-output-syndrome-corrected.txt")
80
81      # Statistic analysis
82      correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
83      logger.info("After correction:")
84      logger.info("  Number of correct bits: %d", correct_bits)
85      logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
86      logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
            tx_msg))
87
88
89  def cyclic_png():
90      """
91      Source - Channel encoder - Channel - Channel decoder - Destination
92
93      png    - (n,k) cyclic    - bsc     - (n,k) cyclic    - png
94      """
95      logger.info("***PNG***")
96      src = source.Source()
97      chl = channel.Channel()
98      cyclic_code = channel.Cyclic_Code(G)
99      dest = destination.Destination()
100
101
102     height, width, channels = src.read_png("Resource/image.png")
103     tx_msg = src.get_digital_data()
104     padding_length = K - (len(tx_msg) % K)
105
106     tx_codeword = cyclic_code.encoder_systematic(tx_msg)
107
108     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
109
110     # Statistic analysis
111     correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
            rx_codeword, 0, N)
112     one_error_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword
            , rx_codeword, 1, N)
113     total_codewords = len(tx_codeword) // N
114     uncorrectable_codewords = total_codewords - correct_codewords -
            one_error_codewords
115     logger.info("Channel statistic analysis:")
116     logger.info("  Total codewords: %d", total_codewords)
```

```
117         logger.info("  Correct codewords: %d", correct_codewords)
118         logger.info("  Codeword error rate: %f", (total_codewords -
                correct_codewords) / total_codewords)
119         logger.info("    One error codewords: %d", one_error_codewords)
120         logger.info("    Uncorrectable codewords: %d", uncorrectable_codewords)
121 
122         # without error correction
123         rx_msg = cyclic_code.decoder_systematic(rx_codeword, padding_length)
124         dest.set_digital_data(rx_msg)
125         dest.write_png_from_digital("Result/cyclic-bsc-output.png", height, width,
                channels)
126 
127         # Statistic analysis
128         correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
129         logger.info("Before correction:")
130         logger.info("  Number of correct bits: %d", correct_bits)
131         logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
132         logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
                tx_msg))
133 
134         # with error correction
135         estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
136         rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword,
                padding_length)
137         dest.set_digital_data(rx_msg)
138         dest.write_png_from_digital("Result/cyclic-bsc-output-syndrome-corrected.
                png", height, width, channels)
139 
140         # Statistic analysis
141         correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
142         logger.info("After correction:")
143         logger.info("  Number of correct bits: %d", correct_bits)
144         logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
145         logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
                tx_msg))
146 
147 
148 def cyclic_wav():
149     """
150     Source - Channel encoder - Channel - Channel decoder - Destination
151 
152     wav    - (n,k) cyclic    - bsc     - (n,k) cyclic    - wav
153     """
154     logger.info("***WAV***")
155     src = source.Source()
156     chl = channel.Channel()
157     cyclic_code = channel.Cyclic_Code(G)
158     dest = destination.Destination()
159 
160 
161     shape, sample_rate = src.read_wav("Resource/file_example_WAV_1MG.wav")
162     plot_wav.plot_wav_time_domain(src.get_analogue_data(), sample_rate, "
            Result/wav-time-domain-TX.png")
```

```
163     plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, "
            Result/wav-frequency-domain-TX.png")
164
165     # tx_msg = src.get_analogue_data()
166     # rx_msg = tx_msg
167     # dest.set_analogue_data(rx_msg)
168     # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
            Result/wav-time-domain-RX.png")
169     # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate
            , "Result/wav-frequency-domain-RX.png")
170     # dest.write_wav_from_analogue(sample_rate, "Result/hamming-bsc-output.wav
            ")
171
172
173     tx_msg = src.get_digital_data()
174     padding_length = K - (len(tx_msg) % K)
175
176     tx_codeword = cyclic_code.encoder_systematic(tx_msg)
177
178     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
179
180     # Statistic analysis
181     correct_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword,
            rx_codeword, 0, N)
182     one_error_codewords = stat_analysis.num_codeword_with_t_errors(tx_codeword
            , rx_codeword, 1, N)
183     total_codewords = len(tx_codeword) // N
184     uncorrectable_codewords = total_codewords - correct_codewords -
            one_error_codewords
185     logger.info("Channel statistic analysis:")
186     logger.info("  Total codewords: %d", total_codewords)
187     logger.info("  Correct codewords: %d", correct_codewords)
188     logger.info("  Codeword error rate: %f", (total_codewords -
            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 = cyclic_code.decoder_systematic(rx_codeword, padding_length)
194
195     dest.set_digital_data(rx_msg)
196     dest.write_wav_from_digital(shape, sample_rate, "Result/cyclic-bsc-output.
            wav")
197
198     plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
            Result/cyclic-bsc-wav-time-domain-RX.png")
199     plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
            "Result/cyclic-bsc-wav-frequency-domain-RX.png")
200
201     # Statistic analysis
202     correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
203     logger.info("Before correction:")
204     logger.info("  Number of correct bits: %d", correct_bits)
205     logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
```

```
206         logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
                tx_msg))
207
208         # with error correction
209         estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
210         rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword,
                padding_length)
211
212         dest.set_digital_data(rx_msg)
213         dest.write_wav_from_digital(shape, sample_rate, "Result/cyclic-bsc-output-
                syndrome-corrected.wav")
214
215         plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
                Result/cyclic-bsc-wav-time-domain-RX-syndrome-corrected.png")
216         plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
                "Result/cyclic-bsc-wav-frequency-domain-RX-syndrome-corrected.png")
217
218         # Statistic analysis
219         correct_bits = stat_analysis.num_correct_bits(tx_msg, rx_msg)
220         logger.info("After correction:")
221         logger.info("  Number of correct bits: %d", correct_bits)
222         logger.info("  Number of incorrect bits: %d", len(tx_msg) - correct_bits)
223         logger.info("  Bit error rate: %f", (len(tx_msg) - correct_bits) / len(
                tx_msg))
224
225
226 if __name__ == '__main__':
227     cyclic_txt()
228     cyclic_png()
229     cyclic_wav()
```