

# EEC269A - Error Correcting Codes I

## Project Report

Chenyue Yang, Pranav Kharche, Parisa Oftadeh

June 5, 2023

## Contents

<b>1 Workload</b>	<b>2</b>
<b>2 Source &amp; Destination</b>	<b>2</b>
2.1 Source . . . . .	2
2.1.1 Text string . . . . .	2
2.1.2 Image . . . . .	2
2.1.3 Audio . . . . .	5
2.2 Destination . . . . .	6
<b>3 Channel</b>	<b>7</b>
3.1 Binary Symmetric Channel (BSC) . . . . .	7
3.2 Additive White Gaussian Noise (AWGN) Channel . . . . .	7
<b>4 (7,4) Systematic Linear Block (Hamming) Code</b>	<b>7</b>
4.1 Syndrome Decoder . . . . .	7
4.1.1 Text string . . . . .	7
4.1.2 Image . . . . .	7
4.1.3 Audio . . . . .	8
<b>5 (<math>n, k</math>) Systematic Cyclic (Hamming) Code</b>	<b>8</b>
5.1 Syndrome Decoder . . . . .	8
5.1.1 Text string . . . . .	8
5.1.2 Image . . . . .	8
5.1.3 Audio . . . . .	8
5.2 LFSR Decoder . . . . .	8
<b>6 Appendix: Python source code</b>	<b>9</b>
6.1 Source . . . . .	9
6.2 Channel . . . . .	16
6.3 Destination . . . . .	20
6.4 Utilities . . . . .	24
6.5 Linear code . . . . .	28
6.6 Cyclic code . . . . .	31

# 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
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	TODO
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	<b>Create generator matrix for (<math>n, k</math>) cyclic code</b>	Pranav
	<b>Adjustable (<math>n, k</math>)</b>	Pranav

## 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.:

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.

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

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. 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.

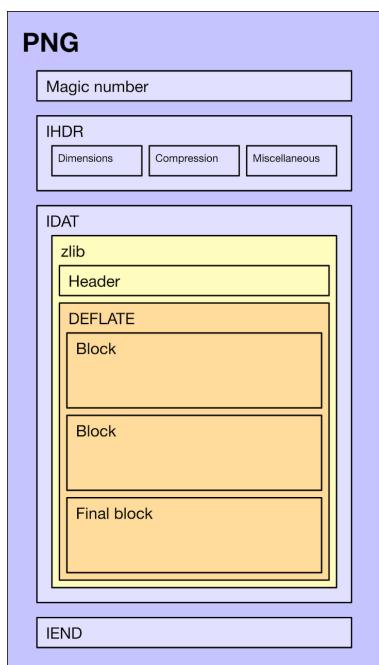


Figure 1: PNG file structure

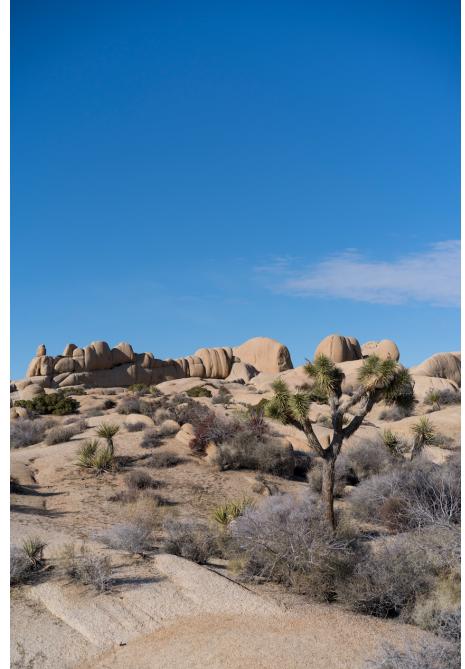


Figure 2: Our testing PNG image

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

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.
  - 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<sup>1</sup>, 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.

---

<sup>1</sup>The image file ‘image.png’ used in this project is photographed by Chenye Yang at Joshua Tree National Park.

### 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. 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.

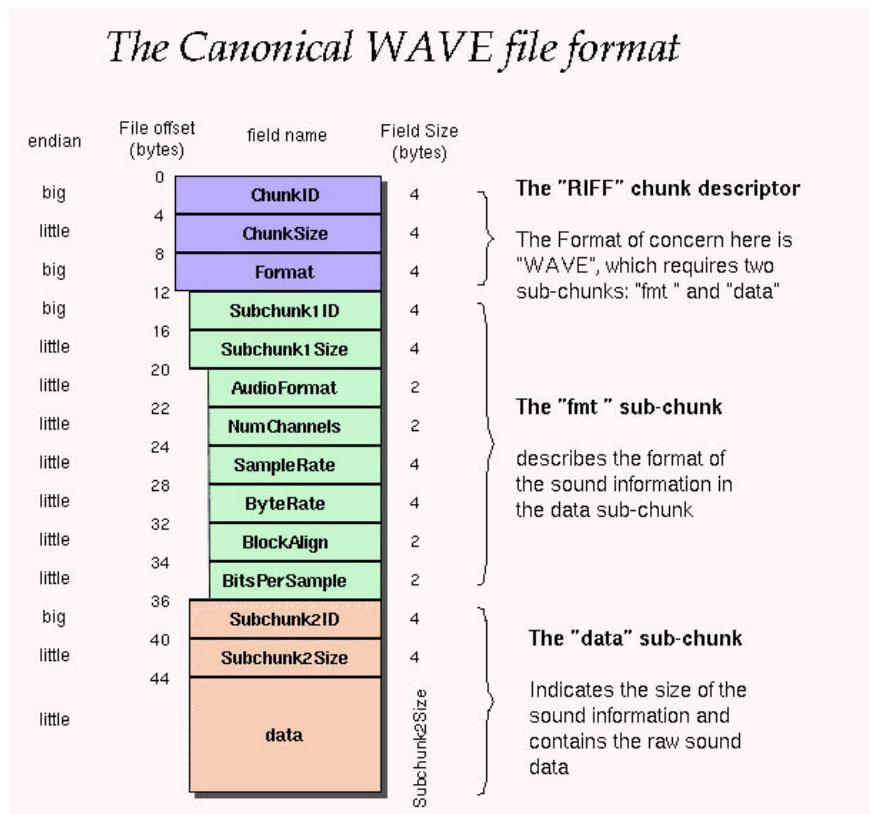


Figure 3: Canonical WAV file structure

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 file<sup>2</sup>, 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.

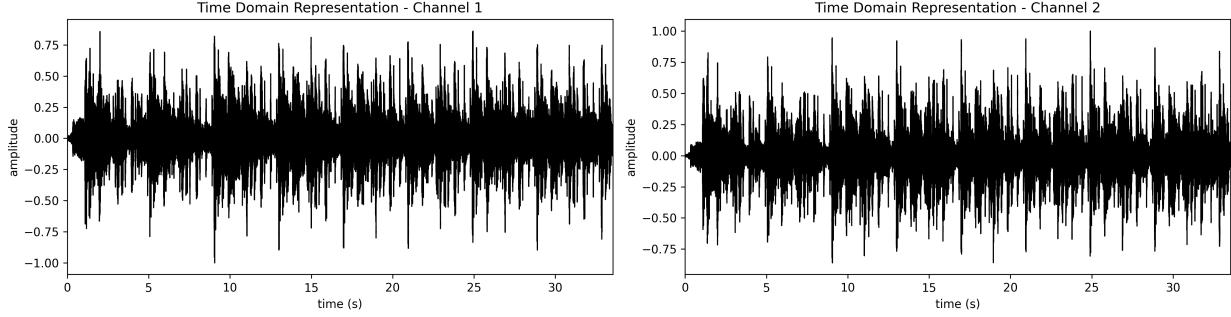


Figure 4: Time domain waveform of our testing audio file

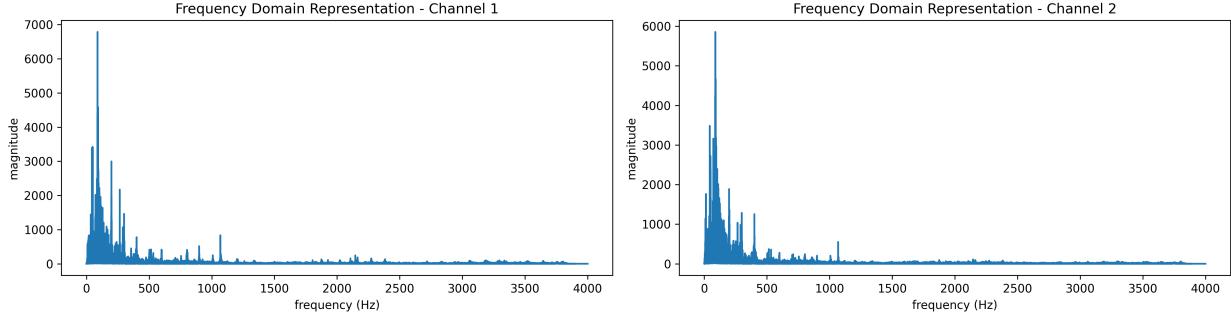


Figure 5: Frequency domain waveform of our testing audio file

## 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. 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.

---

<sup>2</sup>The audio file '*file\_example\_WAV\_1MG.wav*' used in this project is free downloaded from [file-examples.com](http://file-examples.com).

### 3 Channel

#### 3.1 Binary Symmetric Channel (BSC)

#### 3.2 Additive White Gaussian Noise (AWGN) Channel

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

#### 4.1 Syndrome Decoder

##### 4.1.1 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

##### 4.1.2 Image



(a) Original

(b) Without correction

(c) Corrected

Figure 6: Image encoded with Linear Hamming passed through BSC (entire)

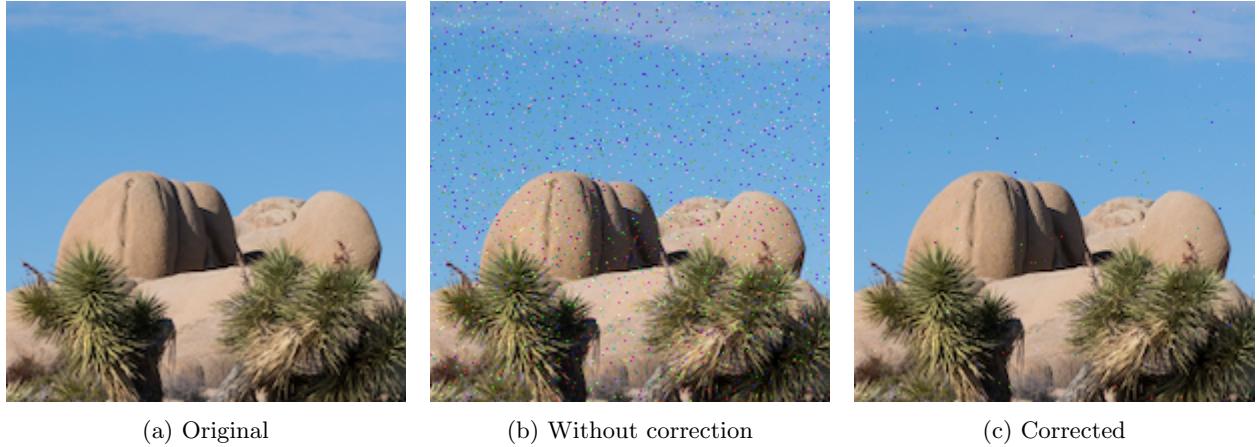


Figure 7: Image encoded with Linear Hamming passed through BSC (details)

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

#### 4.1.3 Audio

## 5 $(n, k)$ Systematic Cyclic (Hamming) Code

### 5.1 Syndrome Decoder

#### 5.1.1 Text string

#### 5.1.2 Image

#### 5.1.3 Audio

### 5.2 LFSR Decoder

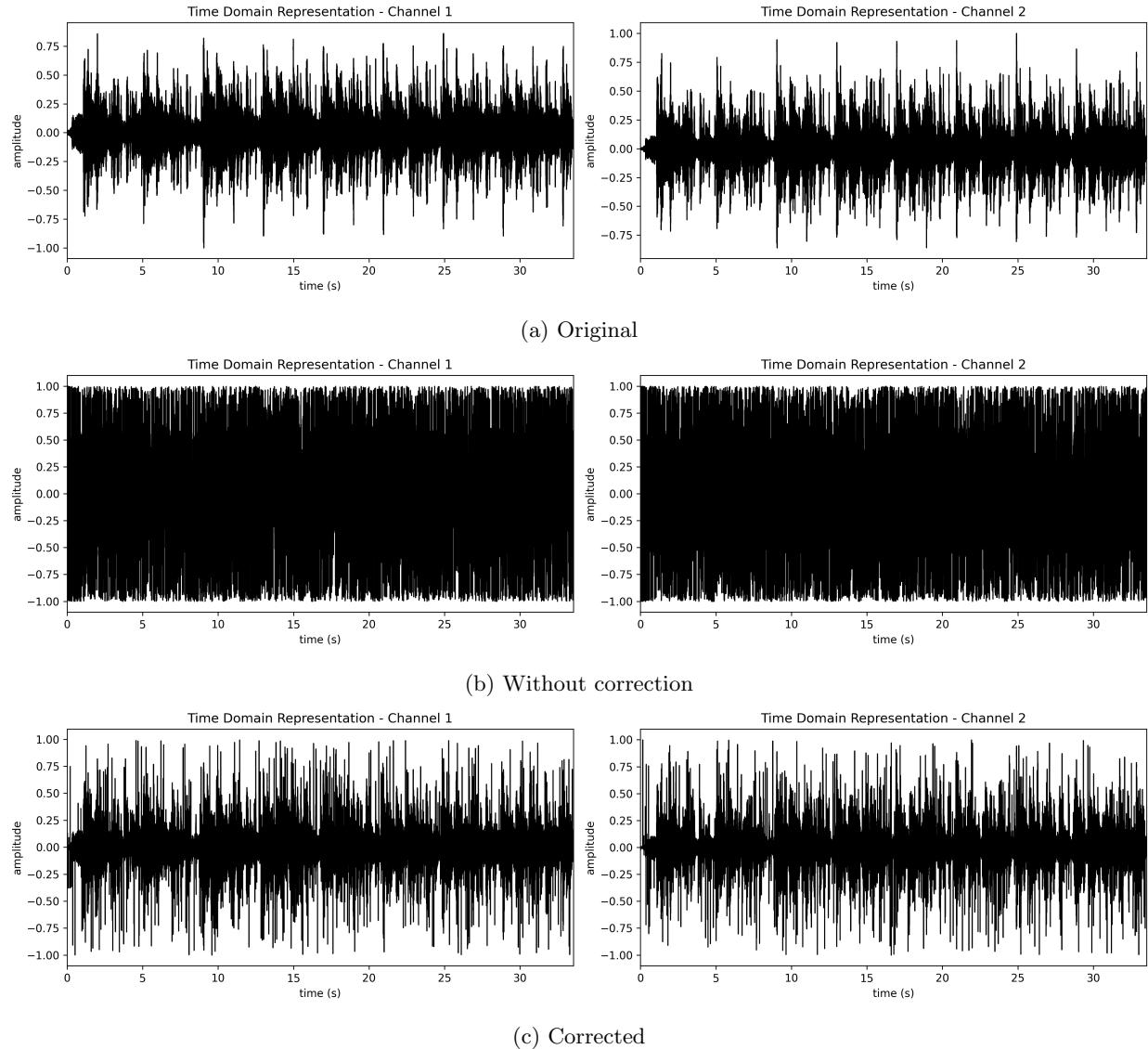


Figure 8: Audio encoded with Linear Hamming passed through BSC

## 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__)

```

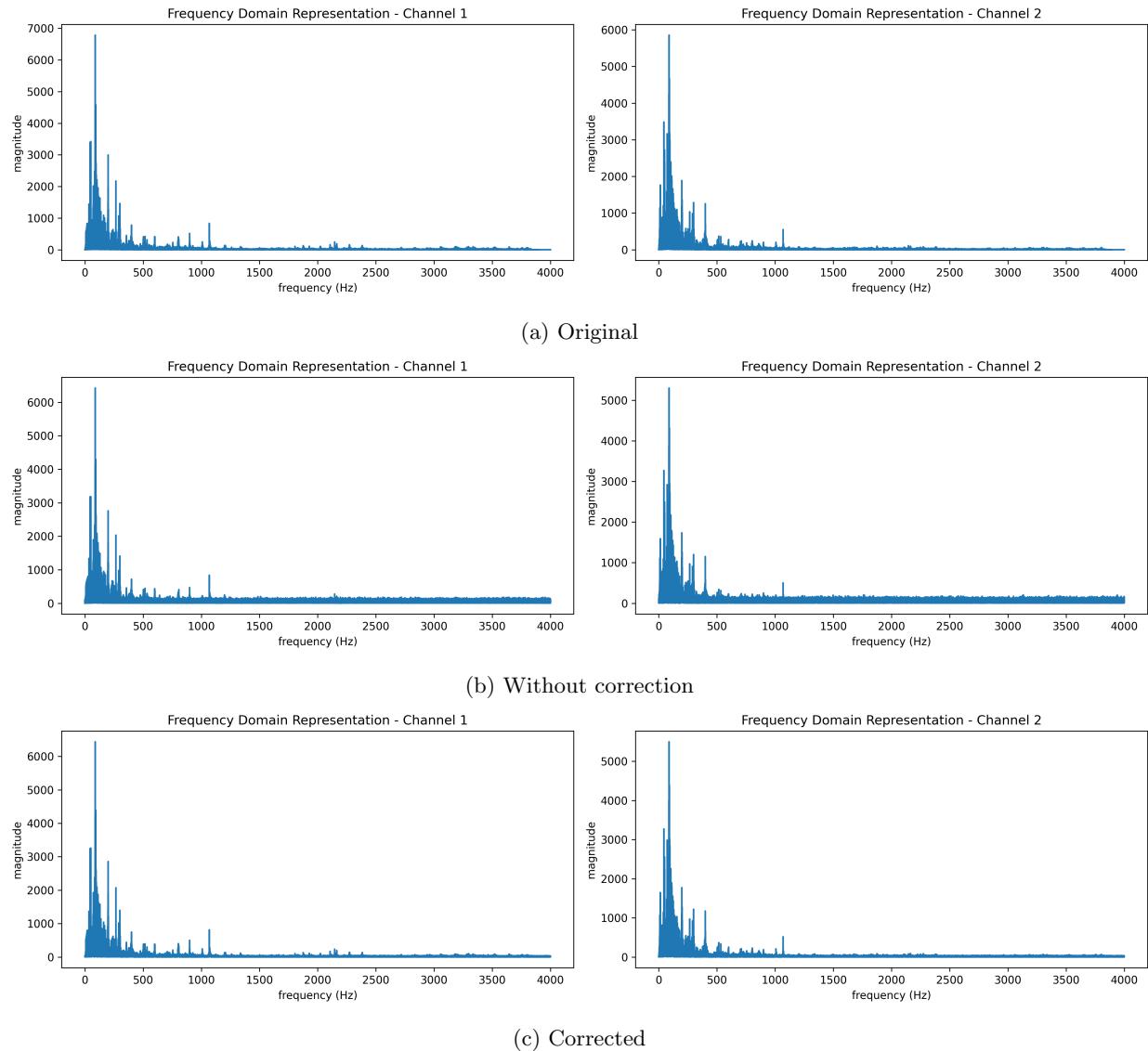
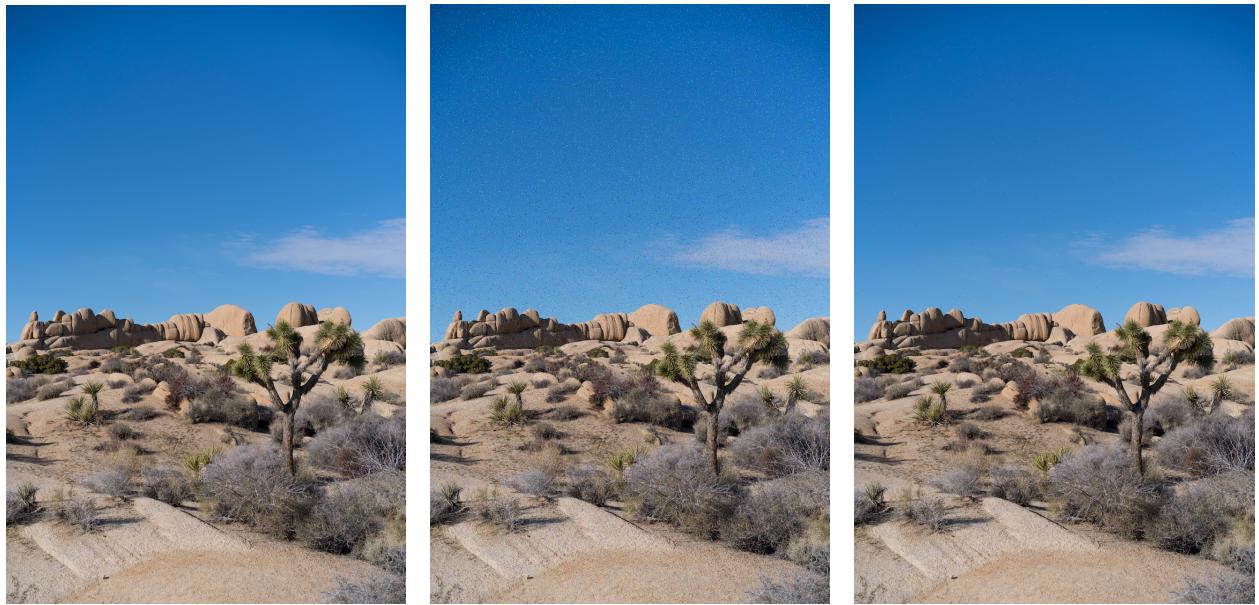


Figure 9: Audio encoded with Linear Hamming passed through BSC

```

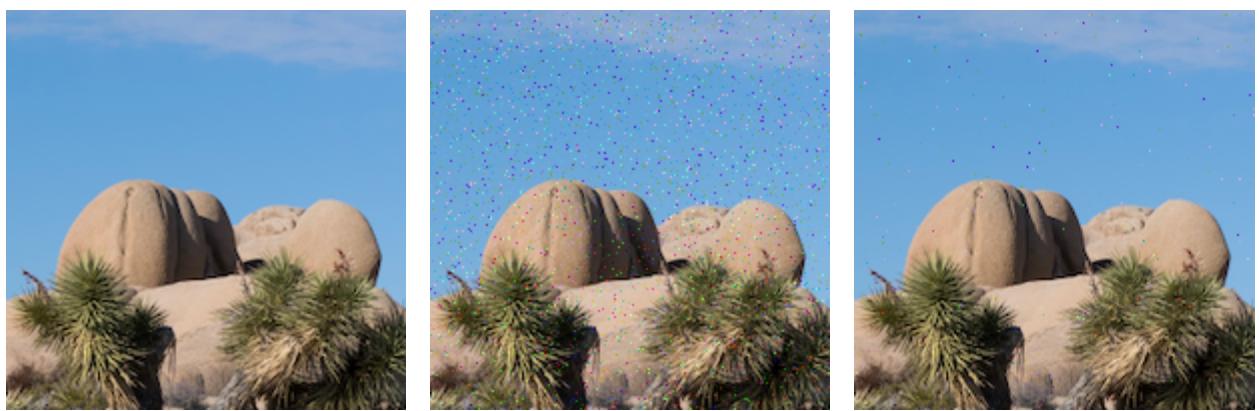
11
12
13 class Source:
14     """
15     The data source
16     """
17
18 def read_txt(self, src_path):
19     """
20         Read the text file as binary bits
21
22         @type src_path: string
23         @param src_path: source file path, with extension
24     """
25     with open(src_path, 'rb') as file:

```



) Original (b) Without correction (c) Corrected

```
26     byte_array = bytearray(file.read())
27     self._digital_data = np.unpackbits(np.frombuffer(byte_array, dtype
28                                         =np.uint8))
29
30     def read_png(self, src_path):
31         """
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
```



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

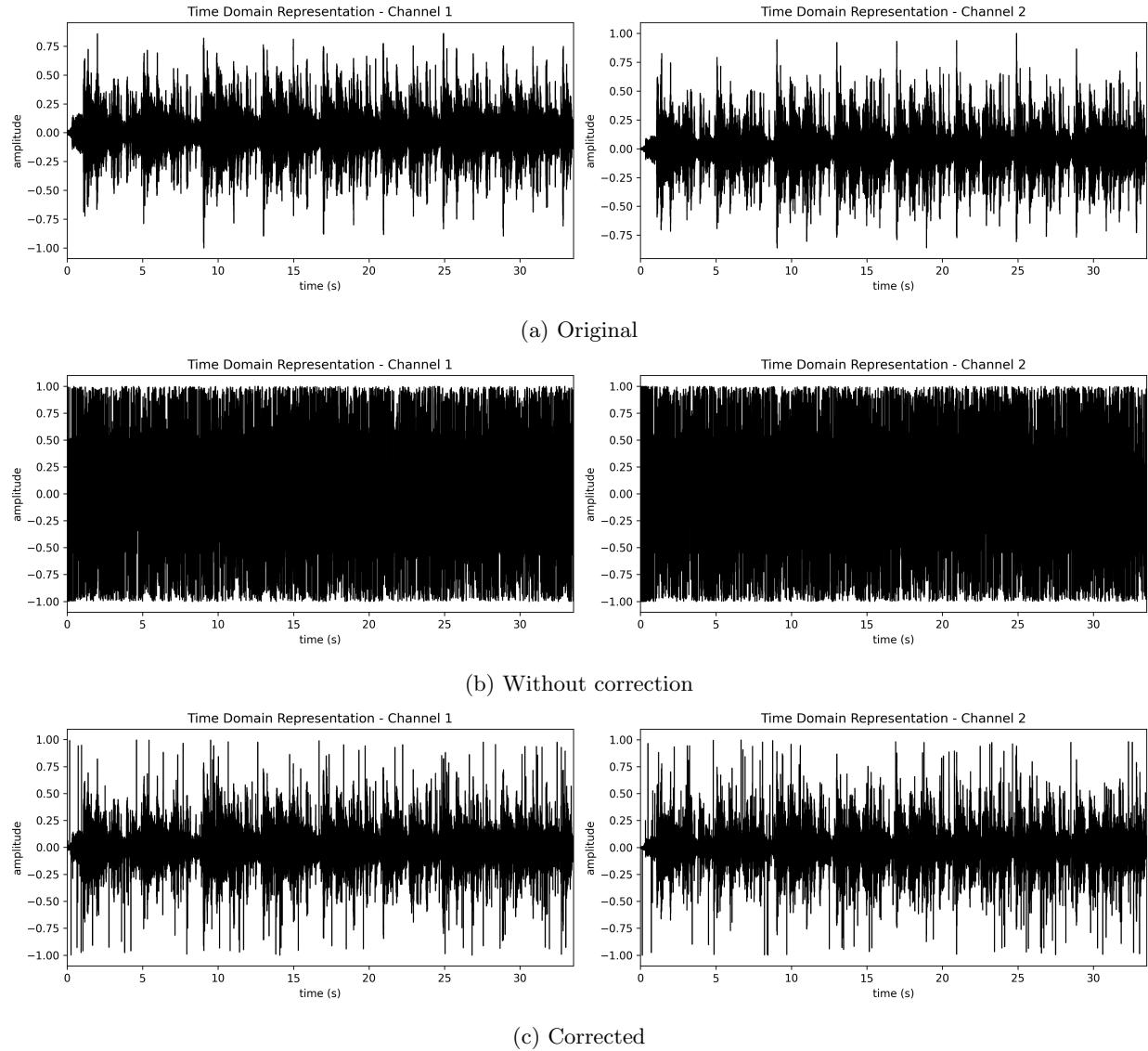


Figure 12: Audio encoded with Cyclic Hamming passed through BSC

```

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
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

```

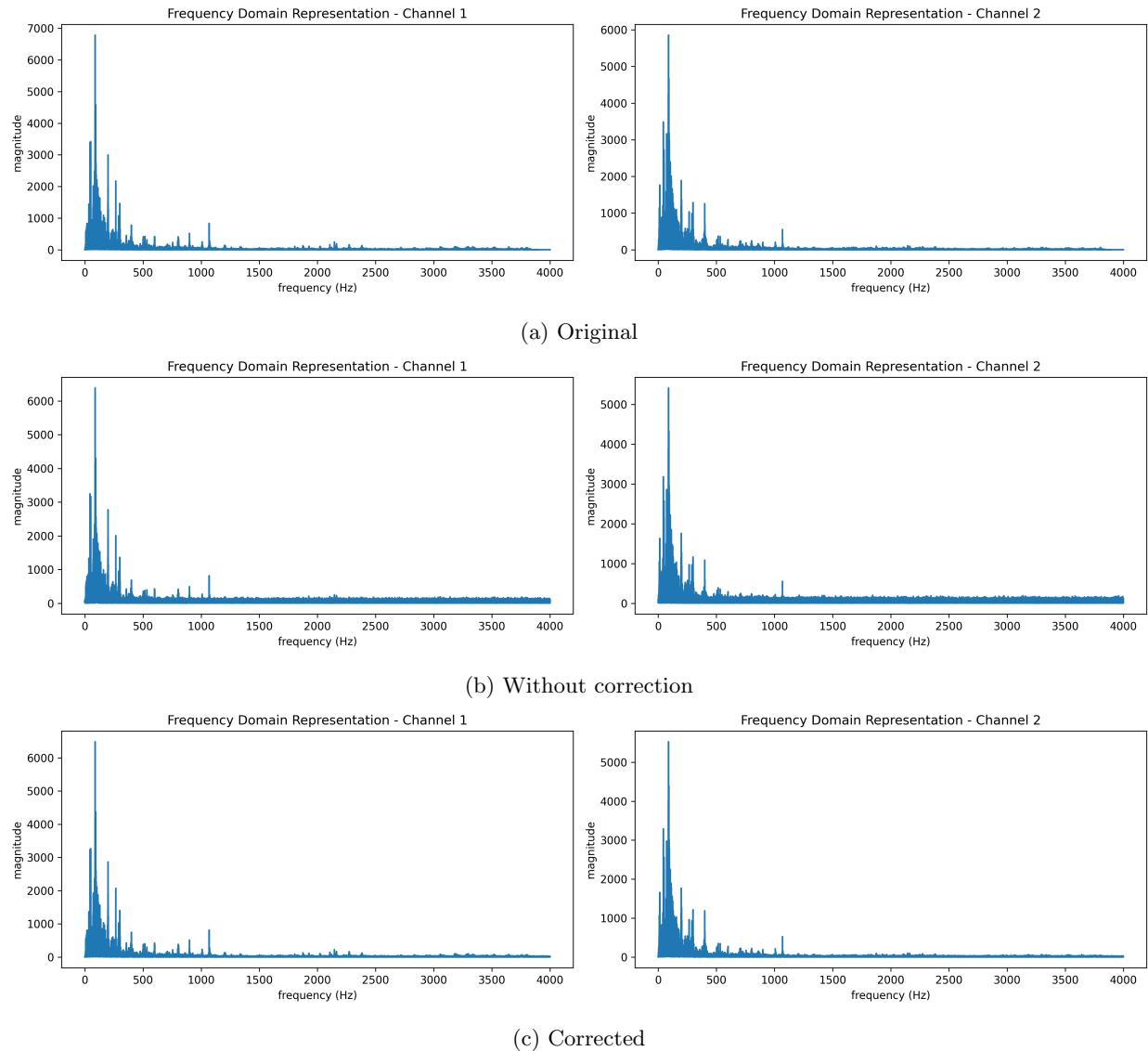


Figure 13: Audio encoded with Cyclic Hamming passed through BSC

```

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.
108    #     sample_width , audio.channels
109
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/
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])
169
170     shape, sample_rate = source.read_wav("Resource/file_example_WAV_1MG.wav")
171     print(' ')
172     print(shape, sample_rate)
173     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 # Create a logger in this module
7 logger = logging.getLogger(__name__)
8
9
10 def create_parity_check_matrix(G):
11     """
12         Create the parity-check matrix,  $H = [I_{n-k} \mid P \cdot T]$ 
13
14         @type G: ndarray
15         @param G: generator matrix in systematic form,  $G = [P \mid I_k]$ 
16
17         @rtype: ndarray
18         @return: parity-check matrix
19     """
20
21     # Get the size of the generator matrix
22     k, n = G.shape
23
24     # Create the parity-check matrix
25     #  $G = [P \mid I_k]$ , Extract P
26     P = G[:, :n-k]
27
28     #  $H = [I_{n-k} \mid P \cdot T]$ 
29     H = np.hstack((np.eye(n-k, dtype=np.uint8), P.T))
30
31     return H
32
33
34 def create_syndrome_table(H):
35     """
36         Create a table of all possible syndromes and their corresponding error
37         vectors (syndrome look-up table)
38
39         @type H: ndarray
40         @param H: parity-check matrix
41     """
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)

```

```

45     coset_leader = np.vstack((np.zeros(n, dtype=np.uint8), np.eye(n, dtype=np.
46                               uint8)))
47     possible_syndromes = (coset_leader @ H.T) % 2
48     syndrome_table = {str(possible_syndromes[i]) : coset_leader[i] for i in
49                         range(n+1)}
50
51
52
53 class Linear_Code:
54     """
55     (7, 4) Systematic Linear Block (Hamming) Code
56     """
57     def __init__(self):
58         self.n, self.k = 7, 4
59         self.G = np.array([[1, 1, 0, 1, 0, 0, 0],
60                           [0, 1, 1, 0, 1, 0, 0],
61                           [1, 1, 1, 0, 0, 1, 0],
62                           [1, 0, 1, 0, 0, 0, 1]], dtype=np.uint8)
63         self.H = np.array([[1, 0, 0, 1, 0, 1, 1],
64                           [0, 1, 0, 1, 1, 1, 0],
65                           [0, 0, 1, 0, 1, 1, 1]], dtype=np.uint8)
66         self.syndrome_table = {'[0_0_0]' : np.array([0, 0, 0, 0, 0, 0, 0]),
67                               '[1_0_0]' : np.array([1, 0, 0, 0, 0, 0, 0]),
68                               '[0_1_0]' : np.array([0, 1, 0, 0, 0, 0, 0]),
69                               '[0_0_1]' : np.array([0, 0, 1, 0, 0, 0, 0]),
70                               '[1_1_0]' : np.array([0, 0, 0, 1, 0, 0, 0]),
71                               '[0_1_1]' : np.array([0, 0, 0, 0, 1, 0, 0]),
72                               '[1_1_1]' : np.array([0, 0, 0, 0, 0, 1, 0]),
73                               '[1_0_1]' : np.array([0, 0, 0, 0, 0, 0, 1])}
74
75     def encoder_systematic(self, bits):
76         """
77             Systematic - Encode the to-be-transmitted binary bits message with
78             (7,4) hamming encoder, return the to-be-transmitted codewords
79
80             @type bits: ndarray
81             @param bits: TX message
82
83             @rtype: ndarray
84             @return: TX codewords
85         """
86         encoded_array = np.zeros((len(bits) // self.k * self.n), dtype=np.
87                                   uint8)
88
89         for i in range(0, len(bits), self.k):
90             message = bits[i:i + self.k]
91             codeword = np.dot(message, self.G) % 2
92             encoded_array[i // self.k * self.n:(i // self.k + 1) * self.n] =
93                         codeword
94
95         return encoded_array

```

```

94
95     def decoder_systematic(self, encoded_array):
96         """
97             Systematic – Decode the received binary bits codeword with (7,4)
98                 Hamming decoder, return the received message
99
100            @type encoded_array: ndarray
101            @param encoded_array: RX codewords
102            @rtype: ndarray
103            @return: RX message
104        """
105        decoded_array = np.zeros((len(encoded_array) // self.n * self.k),
106                                dtype=np.uint8)
106
107        err_count = 0
108
109        for i in range(0, len(encoded_array), self.n):
110            received_codeword = encoded_array[i:i + self.n]
111            syndrome = np.dot(self.H, received_codeword) % 2
112            if np.any(syndrome): # If there are errors, count and keep it
113                err_count += 1
114            decoded_array[i // self.n * self.k : (i // self.n + 1) * self.k] =
115                received_codeword[self.n - self.k:self.n]
116
116            logger.info(f"Error_codeword_rate:{err_count/(len(encoded_array)//self.n)}")
117
118        return decoded_array
119
120
121    def corrector_syndrome(self, received_array):
122        """
123            Systematic – Correct the received binary bits codeword with (7,4)
124                Hamming syndrome look-up table corrector,
125            return the estimated TX codeword = (RX codeword + error pattern)
126
126            @type received_array: ndarray
127            @param received_array: RX codewords
128
129            @rtype: ndarray
130            @return: estimated TX codewords
131        """
132        corrected_array = received_array.copy()
133
134        for i in range(0, len(received_array), self.n):
135            received_codeword = received_array[i:i + self.n]
136            syndrome = np.dot(self.H, received_codeword) % 2
137            if np.any(syndrome):
138                corrected_array[i:i + self.n] = (received_codeword + self.
139                                                syndrome_table[str(syndrome)]) % 2
140
140        return corrected_array
141

```

```

142
143
144 class Cyclic_Code(Linear_Code):
145     """
146     (n, k) Systematic Cyclic (Hamming) Code
147     """
148     def __init__(self, G):
149         self.G = G
150         self.k, self.n = self.G.shape
151         self.H = create_parity_check_matrix(self.G)
152
153         self.syndrome_table = create_syndrome_table(self.H)
154
155
156     def corrector_lfsr(self, received_array):
157         """
158             Systematic – Correct the received binary bits codeword with (n, k)
159             Hamming LFSR corrector,
160             return the estimated TX codeword = (RX codeword + error pattern)
161
162             @type received_array: ndarray
163             @param received_array: RX codewords
164
165             @rtype: ndarray
166             @return: estimated TX codewords
167         """
168         corrected_array = received_array.copy()
169         pass
170
171
172 class Channel:
173     """
174     Channel
175     """
176     def binary_symmetric_channel(self, input_bits, p):
177         """
178             BSC – binary symmetric channel with adjustable error probability
179
180             @type input_bits: ndarray
181             @param input_bits: TX codewords
182
183             @type p: float
184             @param p: error-probability
185
186             @rtype: ndarray
187             @return: RX codewords
188         """
189         output_bits = np.copy(input_bits)
190         for i in range(len(output_bits)):
191             if np.random.random() < p:
192                 output_bits[i] = 1 - output_bits[i] # flip the bit
193
194         return output_bits

```

```

195
196 if __name__ == '__main__':
197     # test
198     tx_msg = np.array([0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1], dtype=np.uint8)
199     print("Original_bits:", tx_msg)
200
201     # Channel
202     channel = Channel()
203
204     # Linear Code
205     linear_code = Linear_Code()
206
207     # Encoding
208     tx_codewords = linear_code.encoder_systematic(tx_msg)
209     print("Encoded_bits:", tx_codewords)
210
211     # Passing through the channel
212     rx_codewords = channel.binary_symmetric_channel(tx_codewords, 0.1)
213     print("Bits_after_channel:", rx_codewords)
214
215     # Correction with syndrome look-up table
216     estimated_tx_codewords = linear_code.corrector_syndrome(rx_codewords)
217     print("Estimated_TX_bits:", estimated_tx_codewords)
218
219     # Decoding
220     rx_msg = linear_code.decoder_systematic(estimated_tx_codewords)
221     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     @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
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
112     # Create a new image from the pixel values
113     new_image = Image.fromarray(self._analogue_data)
114
115     # Save the new image to a file
116     new_image.save(dest_path)
117
118     # Convert the img_array to binary format and flatten the array
119     bits = np.unpackbits(self._analogue_data.astype(np.uint8))
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

```

```

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
146     store the audio information as bit array in _digital_data
147
148         @type dest_path: string
149         @param dest_path: destination file path, with extension
150     """
151     # Write the array to a wav file
152     # wavfile.write(dest_path, sample_rate, self._analogue_data)
153     sf.write(dest_path, self._analogue_data, sample_rate)
154
155     # Convert the audio_array to binary format and flatten the array
156     bits = np.unpackbits(self._analogue_data.astype(np.int16)).view(np.
157         uint8))
158
159     # Store the binary bits
160     self._digital_data = bits
161
162     # def write_mp3_from_digital(self, frame_rate, sample_width, channels,
163     # dest_path):
164     """
165     # Write audio information in bits array to a mp3 file
166
167         @type dest_path: string
168         @param dest_path: destination file path, with extension
169     """
170     # Convert the bit array to uint16 array
171     # uint16_array = np.packbits(self._digital_data).astype(np.int16)
172
173     # Create a new AudioSegment object from the modified audio array
174     # modified_audio = AudioSegment(uint16_array.tobytes(), frame_rate,
175     # sample_width, channels)
176
177     # Export the modified audio to a new MP3 file
178     # modified_audio.export(dest_path, format="mp3")

```

```

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 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(
76                                         magnitude_spectrum_channel_1))
77
78     axs[0].plot(frequencies_channel_1[:int(len(frequencies_channel_1)/2)],
79                 magnitude_spectrum_channel_1[:int(len(magnitude_spectrum_channel_1)/2)])
    # plot only first half of frequencies
    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(
86         magnitude_spectrum_channel_2))
87
88     axs[1].plot(frequencies_channel_2[:int(len(frequencies_channel_2)/2)],
89                 magnitude_spectrum_channel_2[:int(len(magnitude_spectrum_channel_2)/2)])
89     # plot only first half of frequencies
90     axs[1].set_xlabel('frequency_(Hz)')
91     axs[1].set_ylabel('magnitude')
92     axs[1].set_title('Frequency-Domain-Representation--Channel-2')
93
94     # Display the plot
95     plt.tight_layout()
96     plt.savefig(dest_path, dpi=300)
97     plt.close()

```

```

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 def order(p):
12     p = p >> 1
13     order = 0
14     while p:
15         p = p >> 1
16         order += 1
17     return order
18
19 def bitRev(p, pOrder = None):
20     if p == 0:
21         return 0
22     if pOrder == None:
23         pOrder = order(p)
24
25     retVal = 0
26     for i in range(pOrder+1):
27         retVal = (retVal << 1) + (p %2)
28         p = p >>1
29     return retVal
30
31 def polyDiv(dividend, divisor, dividendOrder = None, divisorOrder = None):
32     if dividendOrder == None or divisorOrder == None:

```

```

33     dividendOrder = order(dividend)
34     divisorOrder = order(divisor)
35
36     sizeDiff = dividendOrder - divisorOrder
37     rem = dividend
38     remOrder = dividendOrder
39     result = 0
40     for i in range(sizeDiff, -1, -1):
41         if(rem >> remOrder):
42             rem = rem ^ (divisor << i)
43             result = result + (1 << i)
44     remOrder -= 1
45
46     # if rem:
47     #     print(f'{dividend:b}', ':', f'{divisor:b}', '=', f'{result:b}', 'R', f'{rem:0{divisorOrder}b}', sep='')
48     # else:
49     #     print(f'{dividend:b}', ':', f'{divisor:b}', '=', f'{result:b}')
50     return result, rem
51
52 def findGen(n,k):
53     target = (1 << n) + 1
54     gen = (1 << (n-k)) + 3
55     parity, rem = polyDiv(target, gen, n, n-k)
56     while rem:
57         gen += 2
58         if gen >= target:
59             return None, None
60         parity, rem = polyDiv(target, gen, n, n-k)
61         logger.debug('generator =', f'{gen:b}')
62         logger.debug('parity =', f'{parity:b}')
63     return gen, parity
64
65 def buildMatrix(n, k, gen, parity):
66     genMatrix = [gen]
67     # print(f'{gen:0{n}b}')
68     for i in range(k-1):
69         nextLine = genMatrix[i] << 1
70         if (nextLine >> (n-k))%2:
71             nextLine = nextLine ^ gen
72         genMatrix.append(nextLine)
73         # print(f'{nextLine:0{n}b}')
74     logger.debug('Generator matrix')
75     for i in range(k):
76         genMatrix[i] = bitRev(genMatrix[i], n-1)
77         logger.debug(f'{genMatrix[i]:0{n}b}')
78     return genMatrix
79
80 def findMatrix(n,k):
81     gen, parity = findGen(n,k)
82     if gen == None:
83         return None
84     return buildMatrix(n, k, gen, parity)

```

```

85
86     def encode(data, gen):
87         if order(data)+1 > len(gen):
88             return None
89         result = 0
90         data = bitRev(data, len(gen)-1)
91         for row in gen:
92             result = result ^ ((data & 1)*row)
93             data = data >> 1
94         return result
95
96
97     def genMatrixDecmial2Ndarray(genMatrix_decimal, n):
98         """
99             Convert the generator matrix in decimal form to numpy array form
100
101            @type genMatrix_decimal: list
102            @param genMatrix_decimal: generator matrix in decimal form
103
104            @type n: int
105            @param n: number of bits in a codeword
106
107            @rtype: ndarray
108            @return: generator matrix in numpy array form
109        """
110        # Convert the generator matrix in decimal form to binary form
111        genMatrix_binary = [[int(bit) for bit in f'{num:0{n}b}'] for num in
112                            genMatrix_decimal]
113
114        # Convert the binary generator matrix to a numpy array
115        G = np.array(genMatrix_binary, dtype=np.uint8)
116
117        return G

```

## 6.5 Linear code

```

1  # Copyright (c) 2023 Chenye Yang
2
3  import logging
4
5  from Source import source
6  from Channel import channel
7  from Destination import destination
8  from Utils import plot_wav
9
10
11 # Configure the logging
12 logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
13
14 # Create a logger in the main module
15 logger = logging.getLogger(__name__)
16

```

```

17
18
19 def linear_txt():
20     """
21         Source – Channel encoder – Channel – Channel decoder – Destination
22
23         txt      – (7,4) linear      – bsc      – (7,4) linear      – txt
24     """
25     src = source.Source()
26     chl = channel.Channel()
27     linear_code = channel.Linear_Code()
28     dest = destination.Destination()
29
30
31     src.read_txt("Resource/hardcoded.txt")
32     tx_msg = src.get_digital_data()
33
34     tx_codeword = linear_code.encoder_systematic(tx_msg)
35
36     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
37
38     # without error correction
39     rx_msg = linear_code.decoder_systematic(rx_codeword)
40     dest.set_digital_data(rx_msg)
41     dest.write_txt("Result/linear-bsc-output.txt")
42
43     # with error correction
44     estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
45     rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
46     dest.set_digital_data(rx_msg)
47     dest.write_txt("Result/linear-bsc-output-syndrome-corrected.txt")
48
49
50 def linear_png():
51     """
52         Source – Channel encoder – Channel – Channel decoder – Destination
53
54         png      – (7,4) linear      – bsc      – (7,4) linear      – png
55     """
56     src = source.Source()
57     chl = channel.Channel()
58     linear_code = channel.Linear_Code()
59     dest = destination.Destination()
60
61
62     height, width, channels = src.read_png("Resource/image.png")
63     tx_msg = src.get_digital_data()
64
65     tx_codeword = linear_code.encoder_systematic(tx_msg)
66
67     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
68
69     # without error correction
70     rx_msg = linear_code.decoder_systematic(rx_codeword)

```

```

71     dest.set_digital_data(rx_msg)
72     dest.write_png_from_digital("Result/linear-bsc-output.png", height, width,
73                               channels)
74
75     # with error correction
76     estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
77     rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
78     dest.set_digital_data(rx_msg)
79     dest.write_png_from_digital("Result/linear-bsc-output-syndrome-corrected.
80                               png", height, width, channels)
81
82 def linear_wav():
83     """
84     Source - Channel encoder - Channel - Channel decoder - Destination
85
86     wav      - (7,4) linear      - bsc      - (7,4) linear      - wav
87     """
88     src = source.Source()
89     chl = channel.Channel()
90     linear_code = channel.Linear_Code()
91     dest = destination.Destination()
92
93     shape, sample_rate = src.read_wav("Resource/file-example_WAV_1MG.wav")
94     plot_wav.plot_wav_time_domain(src.get_analogue_data(), sample_rate, "
95                                   Result/wav-time-domain-TX.png")
96     plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, "
97                                   Result/wav-frequency-domain-TX.png")
98
99     # tx_msg = src.get_analogue_data()
100    # rx_msg = tx_msg
101    # dest.set_analogue_data(rx_msg)
102    # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
103                                  Result/wav-time-domain-RX.png")
104    # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
105                                         "Result/wav-frequency-domain-RX.png")
106    # dest.write_wav_from_analogue(sample_rate, "Result/hamming-bsc-output.wav"
107                                     )
108
109    tx_msg = src.get_digital_data()
110
111    tx_codeword = linear_code.encoder_systematic(tx_msg)
112
113    rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
114
115    # without error correction
116    rx_msg = linear_code.decoder_systematic(rx_codeword)
117
118    dest.set_digital_data(rx_msg)
119    dest.write_wav_from_digital(shape, sample_rate, "Result/linear-bsc-output.
120                               wav")

```

```

117     plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
118         Result/linear-bsc-wav-time-domain-RX.png")
119     plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
120         "Result/linear-bsc-wav-frequency-domain-RX.png")
121
122     # with error correction
123     estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
124     rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
125
126     dest.set_digital_data(rx_msg)
127     dest.write_wav_from_digital(shape, sample_rate, "Result/linear-bsc-output-
128         syndrome-corrected.wav")
129
130
131 if __name__ == '__main__':
132     linear_txt()
133     linear_png()
134     linear_wav()

```

## 6.6 Cyclic code

```

1  # Copyright (c) 2023 Chenye Yang
2
3 import logging
4
5 from Source import source
6 from Channel import channel
7 from Destination import destination
8 from Utils import plot_wav, polyTools
9
10
11 # Configure the logging
12 logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
13
14 # Create a logger in the main module
15 logger = logging.getLogger(__name__)
16
17
18
19 # Work with (3, 1) cyclic code
20 n = 3
21 k = 1
22
23 # Create the generator matrix
24 genMatrix_decimal = polyTools.findMatrix(n, k)
25 G = polyTools.genMatrixDecmial2Ndarray(genMatrix_decimal, n)
26

```

```

27
28 def cyclic_txt():
29     """
30     Source – Channel encoder – Channel – Channel decoder – Destination
31
32     txt      – (n, k) cyclic      – bsc      – (n, k) cyclic      – txt
33     """
34     src = source.Source()
35     chl = channel.Channel()
36     cyclic_code = channel.Cyclic_Code(G)
37     dest = destination.Destination()
38
39
40     src.read_txt("Resource/hardcoded.txt")
41     tx_msg = src.get_digital_data()
42
43     tx_codeword = cyclic_code.encoder_systematic(tx_msg)
44
45     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
46
47     # without error correction
48     rx_msg = cyclic_code.decoder_systematic(rx_codeword)
49     dest.set_digital_data(rx_msg)
50     dest.write_txt("Result/cyclic-bsc-output.txt")
51
52     # with error correction
53     estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
54     rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword)
55     dest.set_digital_data(rx_msg)
56     dest.write_txt("Result/cyclic-bsc-output-syndrome-corrected.txt")
57
58
59 def cyclic_png():
60     """
61     Source – Channel encoder – Channel – Channel decoder – Destination
62
63     png      – (n, k) cyclic      – bsc      – (n, k) cyclic      – png
64     """
65     src = source.Source()
66     chl = channel.Channel()
67     cyclic_code = channel.Cyclic_Code(G)
68     dest = destination.Destination()
69
70
71     height, width, channels = src.read_png("Resource/image.png")
72     tx_msg = src.get_digital_data()
73
74     tx_codeword = cyclic_code.encoder_systematic(tx_msg)
75
76     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
77
78     # without error correction
79     rx_msg = cyclic_code.decoder_systematic(rx_codeword)
80     dest.set_digital_data(rx_msg)

```

```

81 dest.write_png_from_digital("Result/cyclic-bsc-output.png", height, width,
82 channels)
83
84 # with error correction
85 estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
86 rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword)
87 dest.set_digital_data(rx_msg)
88 dest.write_png_from_digital("Result/cyclic-bsc-output-syndrome-corrected.
89 png", height, width, channels)
90
91
92 def cyclic_wav():
93     """
94     Source - Channel encoder - Channel - Channel decoder - Destination
95
96     wav      - (n, k) cyclic      - bsc      - (n, k) cyclic      - wav
97     """
98
99     src = source.Source()
100    chl = channel.Channel()
101    cyclic_code = channel.Cyclic_Code(G)
102    dest = destination.Destination()
103
104
105    shape, sample_rate = src.read_wav("Resource/file-example-WAV_1MG.wav")
106    plot_wav.plot_wav_time_domain(src.get_analogue_data(), sample_rate, "
107        Result/wav-time-domain-TX.png")
108    plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, "
109        Result/wav-frequency-domain-TX.png")
110
111    # tx_msg = src.get_analogue_data()
112    # rx_msg = tx_msg
113    # dest.set_analogue_data(rx_msg)
114    # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
115        Result/wav-time-domain-RX.png")
116    # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
117        ", "Result/wav-frequency-domain-RX.png")
118    # dest.write_wav_from_analogue(sample_rate, "Result/hamming-bsc-output.wav"
119    ")
120
121
122    tx_msg = src.get_digital_data()
123
124    tx_codeword = cyclic_code.encoder_systematic(tx_msg)
125
126    rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
127
128    # without error correction
129    rx_msg = cyclic_code.decoder_systematic(rx_codeword)
130
131    dest.set_digital_data(rx_msg)
132    dest.write_wav_from_digital(shape, sample_rate, "Result/cyclic-bsc-output.
133    wav")
134
135

```

```

126 plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
127     Result/cyclic-bsc-wav-time-domain-RX.png")
128 plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
129     "Result/cyclic-bsc-wav-frequency-domain-RX.png")
130
131 # with error correction
132 estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
133 rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword)
134
135 dest.set_digital_data(rx_msg)
136 dest.write_wav_from_digital(shape, sample_rate, "Result/cyclic-bsc-output-
137     syndrome-corrected.wav")
138
139
140 if __name__ == '__main__':
141     cyclic_txt()
142     cyclic_png()
143     cyclic_wav()

```