EEC269A - Error Correcting Codes I
Project Report

Chenye Yang, Pranav Kharche, Parisa Oftadeh

June 5, 2023

Contents

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. For example, the image file we are using has more than one image data (IDAT) chunks, each of which contains a portion of the image, shown in Table 3.

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

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

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

1. *height* is the number of pixels in the vertical direction (i.e. the number of rows of pixels);

2. *width* is the number of pixels in the horizontal direction (i.e. the number of columns of pixels);

3. *channels* is the number of color channels per pixel. This value depends on what type of image it is:

    - In an RGB image, the three channels correspond to Red, Green, and Blue, respectively. Each channel value usually ranges from 0 to 255, where 0 indicates none of that color is present and 255 indicates that color is fully present.

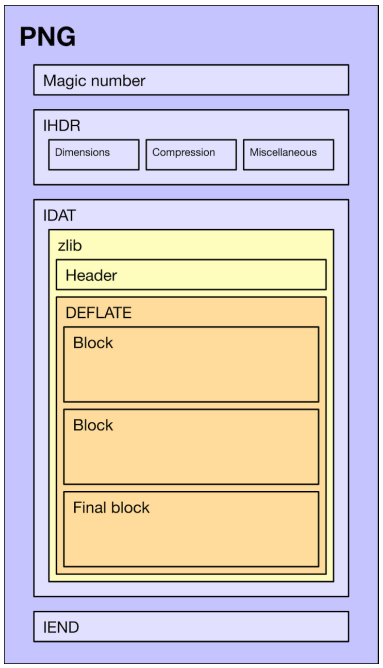


Figure 1: PNG file structure

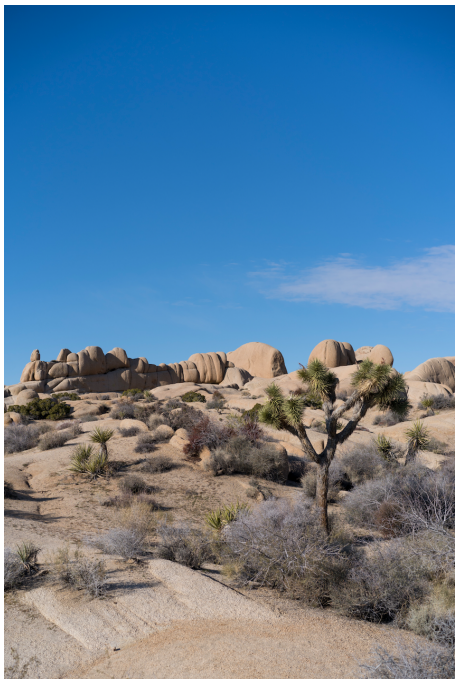


Figure 2: Our testing PNG image

- 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[1], 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. 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).

[1]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.

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

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.

For example, as for our testing audio file[2], 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.

[2]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.

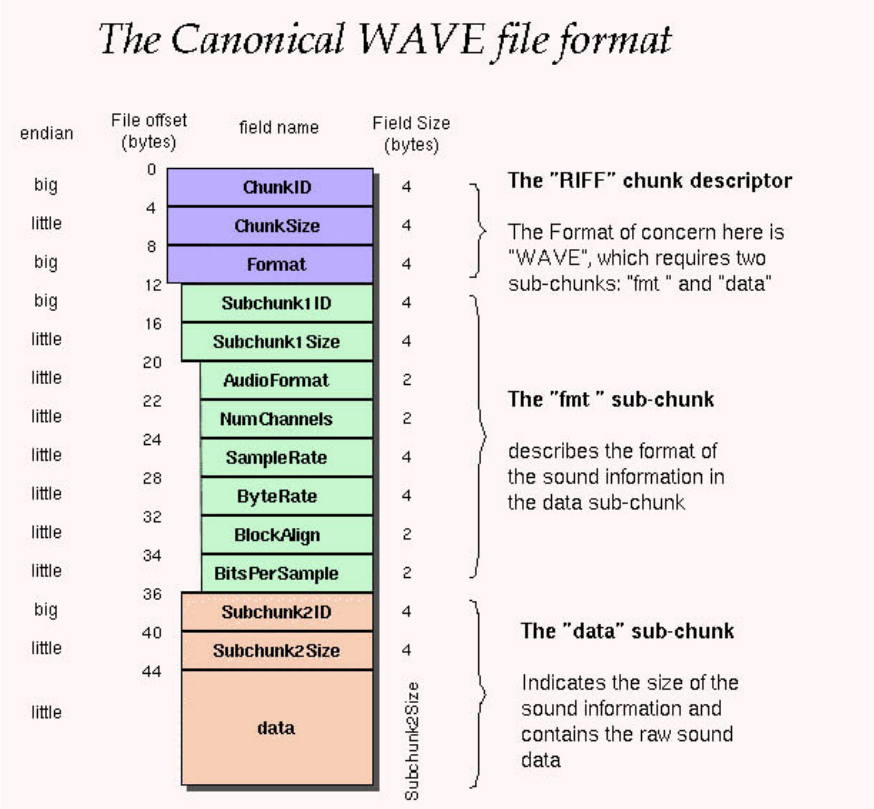


Figure 3: Canonical WAV file structure

2.2 Destination

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

3 Channel

3.1 Binary Symmetric Channel (BSC)

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

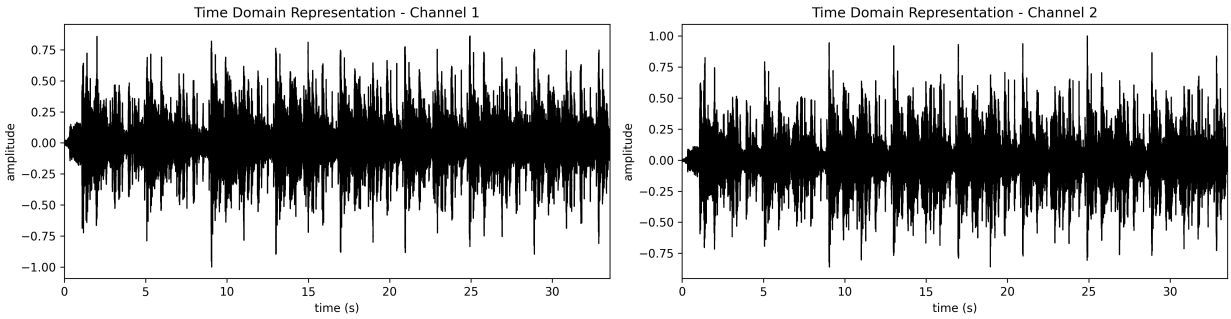


Figure 4: Time domain waveform of our testing audio file

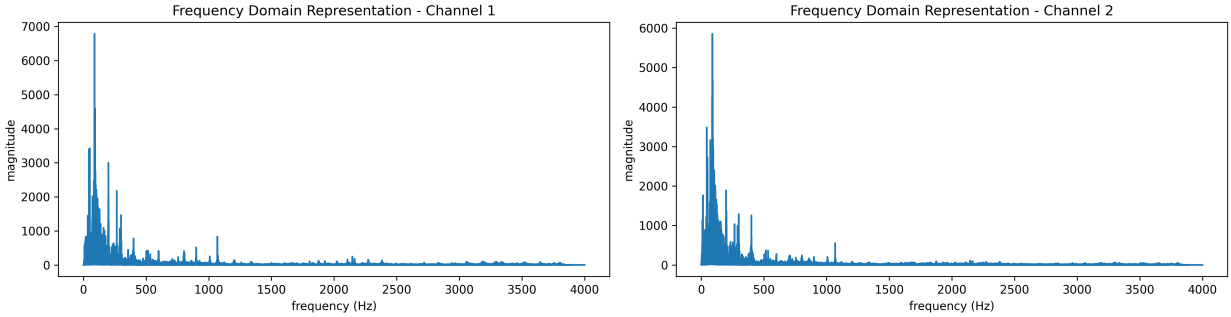


Figure 5: Frequency domain waveform of our testing audio file

4.1.2 Image

4.1.3 Audio

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

5.1 Syndrome Decoder

5.1.1 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

5.1.2 Image

5.1.3 Audio

5.2 LFSR Decoder

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

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

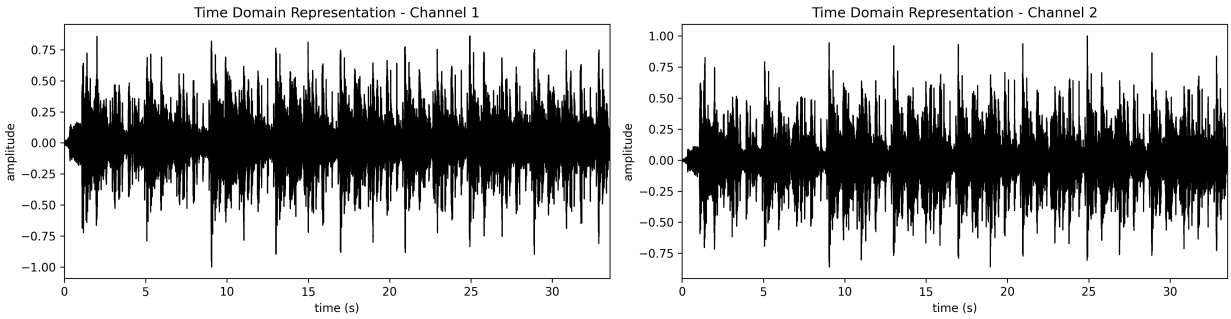
6 Appendix: Python source code

6.1 Source

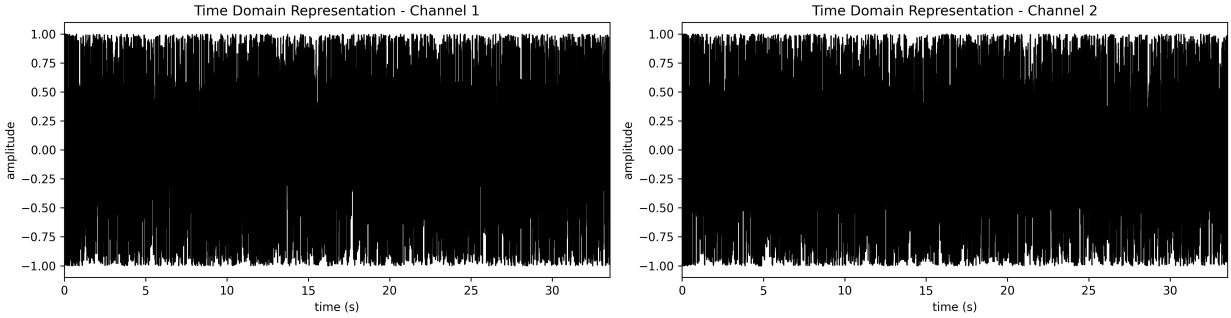
```
# Copyright (c) 2023 Chenye Yang

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

```

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

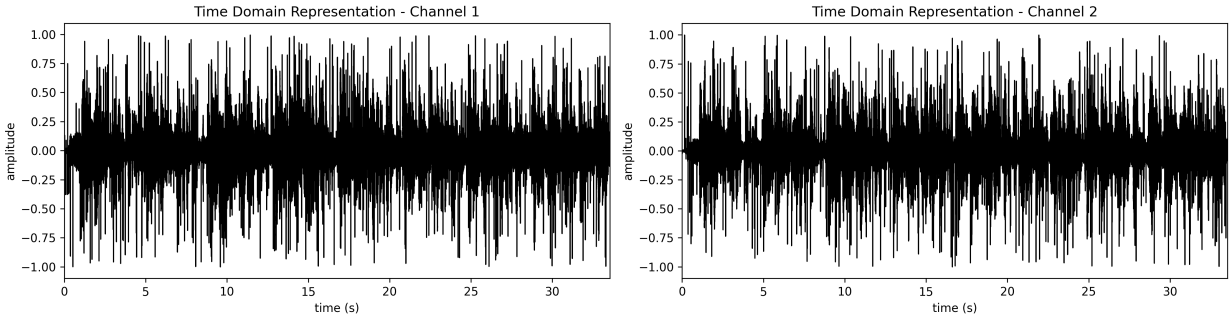
Figure 7: Image encoded with Linear Hamming passed through BSC (details)



(a) Original



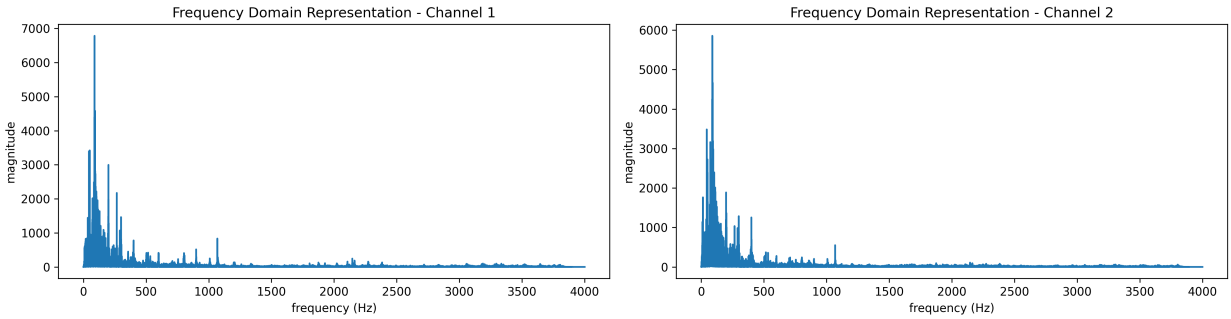
(b) Without correction



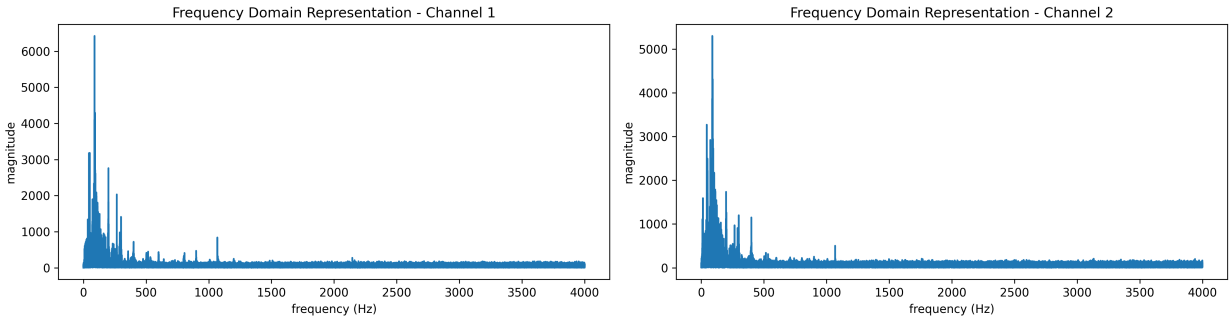
(c) Corrected

Figure 8: Audio encoded with Linear Hamming passed through BSC

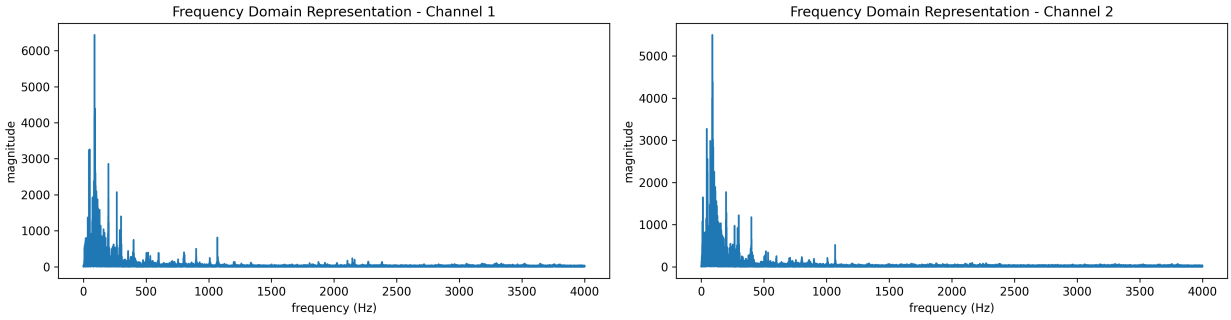
```
# Create a logger in this module
logger = logging.getLogger(__name__)


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
```



(a) Original



(b) Without correction



(c) Corrected

Figure 9: Audio encoded with Linear Hamming passed through BSC

```
        """
        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):
        """
        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
```

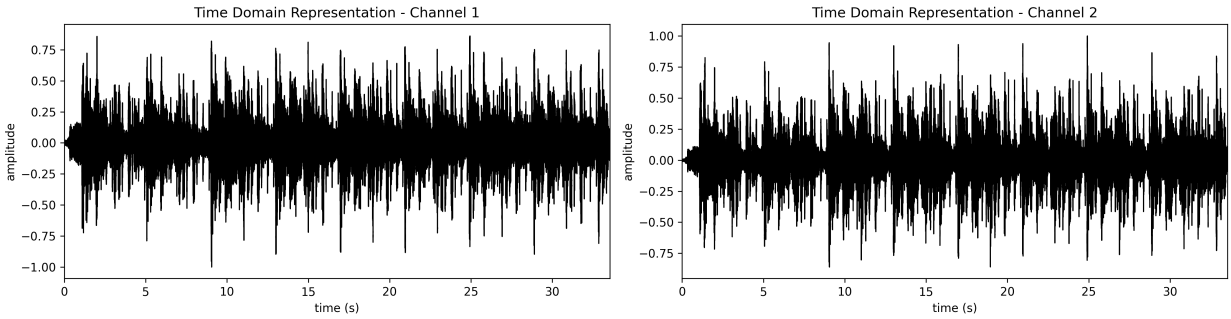
(a) Original (b) Without correction (c) Corrected

Figure 10: Image encoded with Cyclic Hamming passed through BSC (entire)

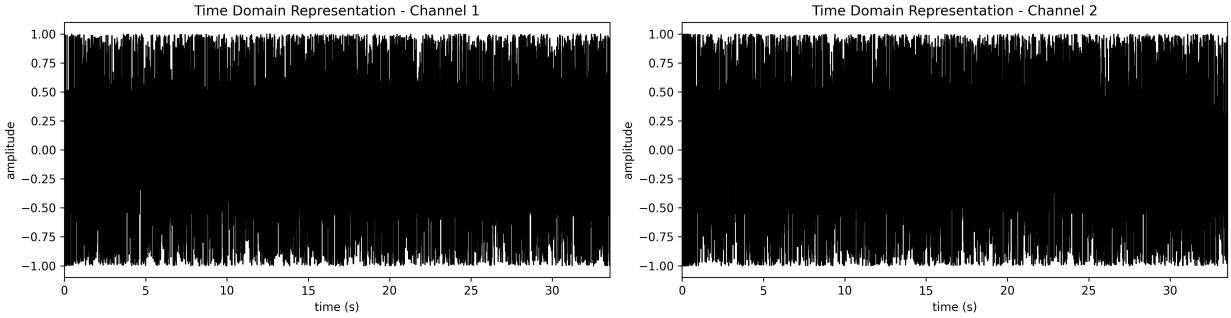
```
38          @param src_path: source file path, with extension
39
40          @rtype:    tuple
41          @return:   height, width, channels
42      """
43      # Open the image file
44      image = Image.open(src_path)
45
46      # Convert the image to a NumPy array
47      img_array = np.array(image)
48
49      # Get the shape of the array (height, width, channels)
```

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

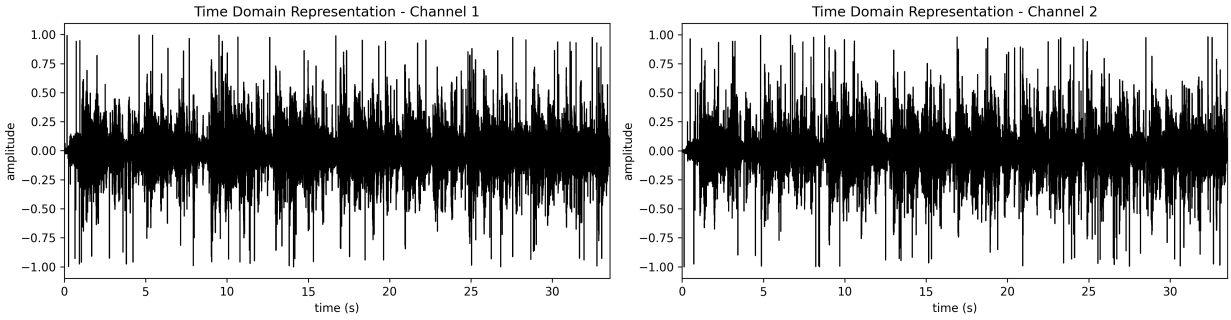
Figure 11: Image encoded with Cyclic Hamming passed through BSC (details)



(a) Original



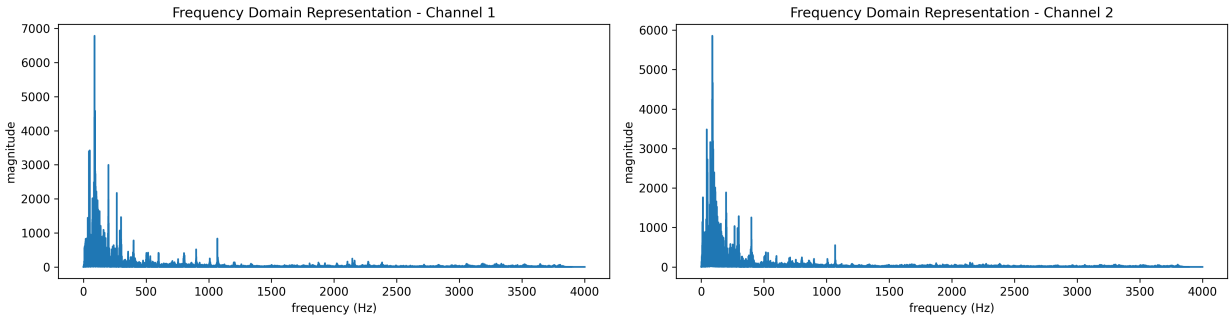
(b) Without correction



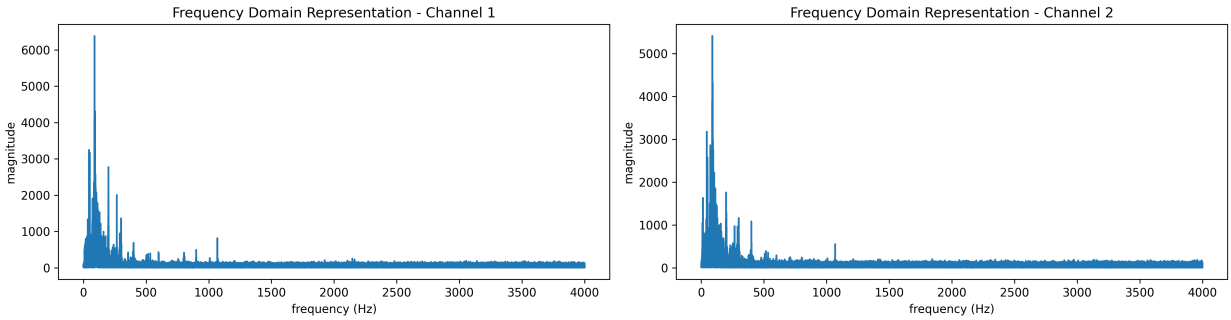
(c) Corrected

Figure 12: Audio encoded with Cyclic Hamming passed through BSC

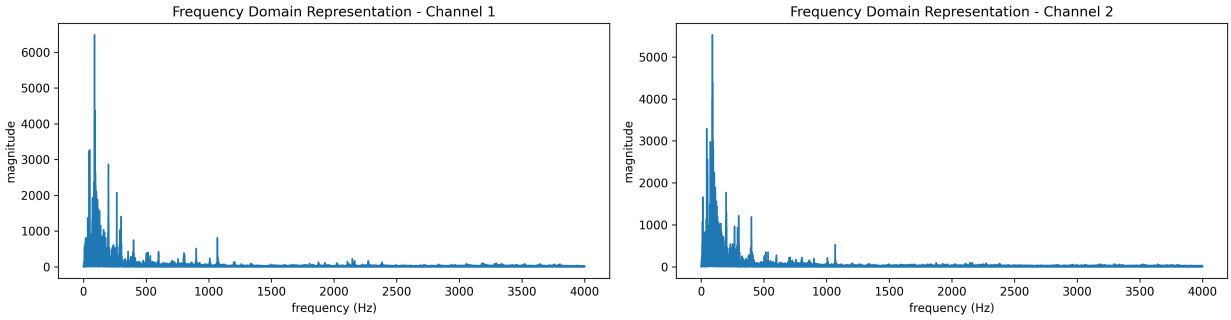
```
        height, width, channels = img_array.shape

        # Convert the img_array to binary format and flatten the array
        bits = np.unpackbits(img_array.astype(np.uint8))

        # Store the binary bits
        self._digital_data = bits
        # Store the analogue data
        self._analogue_data = img_array

        return height, width, channels


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



(a) Original



(b) Without correction



(c) Corrected

Figure 13: Audio encoded with Cyclic Hamming passed through BSC

```
        Read the wav file,
        store the audio information as binary bits in _digital_data,
        store the audio information as ndarray in _analogue_data.
        Return the frame_rate, sample_width, channels of the audio.

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

            @rtype:    tuple, int
            @return:   shape, sample_rate
        """
        # Open the audio file
        # sample_rate, audio_array = wavfile.read(src_path)
        audio_array, sample_rate = sf.read(src_path, dtype='int16')
        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
```

```
            @return:  data bits
        """
        return self._digital_data


    def get_analogue_data(self):
        """
        Get the analogue data to be transmitted

            @rtype:   ndarray
            @return:  analogue data
        """
        return self._analogue_data


if __name__ == '__main__':
    # test
    source = Source()
    source.read_txt("Resource/hardcoded.txt")
    text_bits = source.get_digital_data()
    print(text_bits)
    print(type(text_bits))

    source.read_png("Resource/image.png")
    image_bits = source.get_digital_data()
    print(image_bits[:16])
    image_pixels = source.get_analogue_data()
    print(image_pixels[0][0][:2])

    # frame_rate, sample_width, channels = source.read_mp3("Resource/
        file_example_MP3_1MG.mp3")
    # print(frame_rate, sample_width, channels)
    # audio_bits = source.get_digital_data()
    # print(audio_bits[:16])
    # audio_array = source.get_analogue_data()
    # print(audio_array[:2])

    shape, sample_rate = source.read_wav("Resource/file_example_WAV_1MG.wav")
    print('␣')
    print(shape, sample_rate)
    audio_bits = source.get_digital_data()
    print(audio_bits[:16])
    audio_array = source.get_analogue_data()
    print(audio_array[0])
```

6.2 Channel

```
# Copyright (c) 2023 Chenye Yang

import numpy as np
import logging

# Create a logger in this module
```

```
7  logger = logging.getLogger(__name__)
8
9
10
11 def create_parity_check_matrix(G):
12     """
13     Create the parity-check matrix, H = [I_{n-k} | P.T]
14
15         @type  G: ndarray
16         @param G: generator matrix in systematic form, G = [P | I_k]
17
18         @rtype:   ndarray
19         @return:  parity-check matrix
20     """
21     # Get the size of the generator matrix
22     k, n = G.shape
23
24     # Create the parity-check matrix
25     # G = [P | I_k], Extract P
26     P = G[:, :n-k]
27
28     # H = [I_{n-k} | P.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
           vectors (syndrome look-up table)
37
38         @type  H: ndarray
39         @param H: parity-check matrix
40     """
41     # Get the size of the parity-check matrix
42     _, n = H.shape
43
44     # Create a table of all possible syndromes and their corresponding error
           vectors (syndrome look-up table)
45     coset_leader = np.vstack((np.zeros(n, dtype=np.uint8), np.eye(n, dtype=np.
           uint8)))
46     possible_syndromes = (coset_leader @ H.T) % 2
47     syndrome_table = {str(possible_syndromes[i]) : coset_leader[i] for i in
           range(n+1)}
48
49     return syndrome_table
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
                (7,4) hamming encoder, return the to-be-transmitted codewords
78
79              @type   bits: ndarray
80              @param bits: TX message
81
82              @rtype:    ndarray
83              @return:   TX codewords
84          """
85          encoded_array = np.zeros((len(bits) // self.k * self.n), dtype=np.
                uint8)
86
87          for i in range(0, len(bits), self.k):
88              message = bits[i:i + self.k]
89              codeword = np.dot(message, self.G) % 2
90              encoded_array[i // self.k * self.n:(i // self.k + 1) * self.n] =
                    codeword
91
92          return encoded_array
93
94
95      def decoder_systematic(self, encoded_array):
96          """
97          Systematic - Decode the received binary bits codeword with (7,4)
                Hamming decoder, return the received message
98
99              @type   encoded_array: ndarray
100             @param encoded_array: RX codewords
101
102             @rtype:    ndarray
103             @return:   RX message
104         """
105         decoded_array = np.zeros((len(encoded_array) // self.n * self.k),
                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] =
                    received_codeword[self.n - self.k:self.n]
115
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)
                Hamming syndrome look-up table corrector,
124         return the estimated TX codeword = (RX codeword + error pattern)
125
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.
                        syndrome_table[str(syndrome)]) % 2
139
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)
                Hamming LFSR corrector,
159         return the estimated TX codeword = (RX codeword + error pattern)
160 
161             @type  received_array: ndarray
162             @param received_array: RX codewords
163 
164             @rtype:   ndarray
165             @return:  estimated TX codewords
166         """
167         corrected_array = received_array.copy()
168         pass
169 
170 
171 class Channel:
172     """
173     Channel
174     """
175     def binary_symmetric_channel(self, input_bits, p):
176         """
177         BSC - binary symmetric channel with adjustable error probability
178 
179             @type  input_bits: ndarray
180             @param input_bits: TX codewords
181 
182             @type  p: float
183             @param p: error_probability
184 
185             @rtype:   ndarray
186             @return:  RX codewords
187         """
188         output_bits = np.copy(input_bits)
189         for i in range(len(output_bits)):
190             if np.random.random() < p:
191                 output_bits[i] = 1 - output_bits[i]  # flip the bit
192         return output_bits
193 
194 
195 
196 if __name__ == '__main__':
197     # test
198     tx_msg = np.array([0, 1, 0, 0, 1, 0, 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)
```

```
    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
    rx_msg = linear_code.decoder_systematic(estimated_tx_codewords)
    print("Decoded bits :", rx_msg)
```

6.3 Destination

```
# 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 Destination:
    """
    The destination
    """

    def set_digital_data(self, bits):
        """
        Record the received digital data

            @type  bits: ndarray
            @param bits: data bits
        """
        self._digital_data = bits


    def set_analogue_data(self, array):
        """
        Record the received analogue data

            @type  array: ndarray
            @param array: analogue data array
        """
        self._analogue_data = array


```

```
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 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')
```

```
    axs[1].set_title('Frequency Domain Representation - Channel 2')

    # Display the plot
    plt.tight_layout()
    plt.savefig(dest_path, dpi=300)
    plt.close()
```

```
# Copyright (c) 2023 Pranav Kharche, Chenye Yang
# Toolbox for polynomials and CRC

import numpy as np
import logging

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


def order(p):
        p = p >> 1
        order = 0
        while p:
                p = p >> 1
                order += 1
        return order

def bitRev(p, pOrder = None):
        if p == 0:
                return 0
        if pOrder == None:
                pOrder = order(p)

        retVal = 0
        for i in range(pOrder+1):
                retVal = (retVal << 1) + (p %2)
                p = p >>1
        return retVal

def polyDiv(dividend, divisor, dividendOrder = None, divisorOrder = None):
        if dividendOrder == None or divisorOrder == None:
                dividendOrder = order(dividend)
                divisorOrder = order(divisor)

        sizeDiff = dividendOrder - divisorOrder
        rem = dividend
        remOrder = dividendOrder
        result = 0
        for i in range(sizeDiff, -1, -1):
                if(rem >> remOrder):
                        rem = rem ^ (divisor << i)
                        result = result + (1 << i)
                remOrder -= 1

        # 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          """
```

```
        Convert the generator matrix in decimal form to numpy array form

                @type  genMatrix_decimal: list
                @param genMatrix_decimal: generator matrix in decimal form

                @type  n: int
                @param n: number of bits in a codeword

                @rtype:    ndarray
                @return:   generator matrix in numpy array form
        """
        # Convert the generator matrix in decimal form to binary form
        genMatrix_binary = [[int(bit) for bit in f'{num:0{n}b}'] for num in
            genMatrix_decimal]

        # Convert the binary generator matrix to a numpy array
        G = np.array(genMatrix_binary, dtype=np.uint8)

        return G
```

6.5 Linear code

```
# Copyright (c) 2023 Chenye Yang

import logging

from Source import source
from Channel import channel
from Destination import destination
from Utils import plot_wav


# Configure the logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(
    levelname)s - %(message)s')

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



def linear_txt():
    """
    Source - Channel encoder - Channel - Channel decoder - Destination

    txt    - (7,4) linear    - bsc     - (7,4) linear    - txt
    """
    src = source.Source()
    chl = channel.Channel()
    linear_code = channel.Linear_Code()
    dest = destination.Destination()


```

```
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,
            channels)
73
74      # with error correction
75      estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
76      rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
77      dest.set_digital_data(rx_msg)
78      dest.write_png_from_digital("Result/linear-bsc-output-syndrome-corrected.
            png", height, width, channels)
79
80
81  def linear_wav():
82      """
```

```
    Source - Channel encoder - Channel - Channel decoder - Destination

    wav     - (7,4) linear     - bsc      - (7,4) linear     - wav
    """
    src = source.Source()
    chl = channel.Channel()
    linear_code = channel.Linear_Code()
    dest = destination.Destination()


    shape, sample_rate = src.read_wav("Resource/file_example_WAV_1MG.wav")
    plot_wav.plot_wav_time_domain(src.get_analogue_data(), sample_rate, "Result/wav-time-domain-TX.png")
    plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, "Result/wav-frequency-domain-TX.png")

    # tx_msg = src.get_analogue_data()
    # rx_msg = tx_msg
    # dest.set_analogue_data(rx_msg)
    # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "Result/wav-time-domain-RX.png")
    # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate, "Result/wav-frequency-domain-RX.png")
    # dest.write_wav_from_analogue(sample_rate, "Result/hamming-bsc-output.wav")


    tx_msg = src.get_digital_data()

    tx_codeword = linear_code.encoder_systematic(tx_msg)

    rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)

    # without error correction
    rx_msg = linear_code.decoder_systematic(rx_codeword)

    dest.set_digital_data(rx_msg)
    dest.write_wav_from_digital(shape, sample_rate, "Result/linear-bsc-output.wav")

    plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "Result/linear-bsc-wav-time-domain-RX.png")
    plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate, "Result/linear-bsc-wav-frequency-domain-RX.png")

    # with error correction
    estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
    rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)

    dest.set_digital_data(rx_msg)
    dest.write_wav_from_digital(shape, sample_rate, "Result/linear-bsc-output-syndrome-corrected.wav")

```

```
    plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "Result/linear-bsc-wav-time-domain-RX-syndrome-corrected.png")
    plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate, "Result/linear-bsc-wav-frequency-domain-RX-syndrome-corrected.png")


if __name__ == '__main__':
    linear_txt()
    linear_png()
    linear_wav()
```

6.6 Cyclic code

```
# Copyright (c) 2023 Chenye Yang

import logging

from Source import source
from Channel import channel
from Destination import destination
from Utils import plot_wav, polyTools


# Configure the logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')

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



# Work with (3, 1) cyclic code
n = 3
k = 1

# Create the generator matrix
genMatrix_decimal = polyTools.findMatrix(n, k)
G = polyTools.genMatrixDecmial2Ndarray(genMatrix_decimal, n)


def cyclic_txt():
    """
    Source - Channel encoder - Channel - Channel decoder - Destination

    txt    - (n,k) cyclic    - bsc     - (n,k) cyclic    - txt
    """
    src = source.Source()
    chl = channel.Channel()
    cyclic_code = channel.Cyclic_Code(G)
    dest = destination.Destination()


```

```
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,
            channels)
82
83      # with error correction
84      estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
85      rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword)
86      dest.set_digital_data(rx_msg)
87      dest.write_png_from_digital("Result/cyclic-bsc-output-syndrome-corrected.
            png", height, width, channels)
88
89
90  def cyclic_wav():
91      """
```

```
    Source - Channel encoder - Channel - Channel decoder - Destination

    wav    - (n,k) cyclic    - bsc     - (n,k) cyclic    - wav
    """
    src = source.Source()
    chl = channel.Channel()
    cyclic_code = channel.Cyclic_Code(G)
    dest = destination.Destination()


    shape, sample_rate = src.read_wav("Resource/file_example_WAV_1MG.wav")
    plot_wav.plot_wav_time_domain(src.get_analogue_data(), sample_rate, "
        Result/wav-time-domain-TX.png")
    plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, "
        Result/wav-frequency-domain-TX.png")

    # tx_msg = src.get_analogue_data()
    # rx_msg = tx_msg
    # dest.set_analogue_data(rx_msg)
    # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
        Result/wav-time-domain-RX.png")
    # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate
        , "Result/wav-frequency-domain-RX.png")
    # dest.write_wav_from_analogue(sample_rate, "Result/hamming-bsc-output.wav
        ")


    tx_msg = src.get_digital_data()

    tx_codeword = cyclic_code.encoder_systematic(tx_msg)

    rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)

    # without error correction
    rx_msg = cyclic_code.decoder_systematic(rx_codeword)

    dest.set_digital_data(rx_msg)
    dest.write_wav_from_digital(shape, sample_rate, "Result/cyclic-bsc-output.
        wav")

    plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
        Result/cyclic-bsc-wav-time-domain-RX.png")
    plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
        "Result/cyclic-bsc-wav-frequency-domain-RX.png")

    # with error correction
    estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
    rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword)

    dest.set_digital_data(rx_msg)
    dest.write_wav_from_digital(shape, sample_rate, "Result/cyclic-bsc-output-
        syndrome-corrected.wav")

```

```
136     plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
            Result/cyclic-bsc-wav-time-domain-RX-syndrome-corrected.png")
137     plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
            "Result/cyclic-bsc-wav-frequency-domain-RX-syndrome-corrected.png")
138
139
140 if __name__ == '__main__':
141     cyclic_txt()
142     cyclic_png()
143     cyclic_wav()
```