# 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 |
| 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** | **Create generator matrix for** $(n,k)$ **cyclic code** | Pranav |
| | **Adjustable** $(n,k)$ | 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.

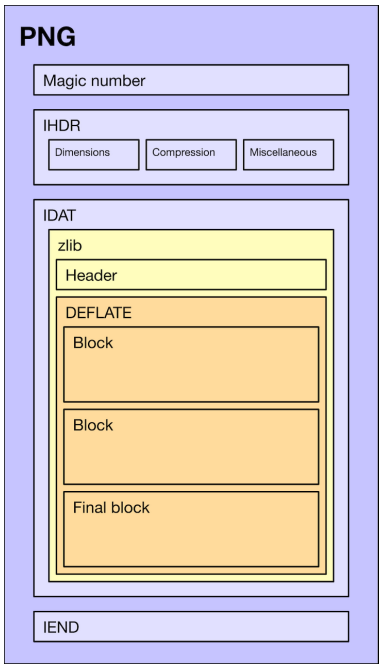



Figure 1: PNG file structure

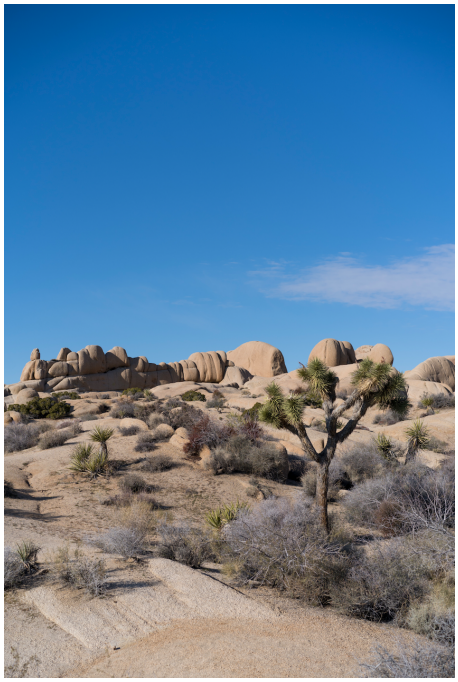


Figure 2: Our testing PNG image

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.

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 (HSV, LAB, etc.) 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.

[1]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.

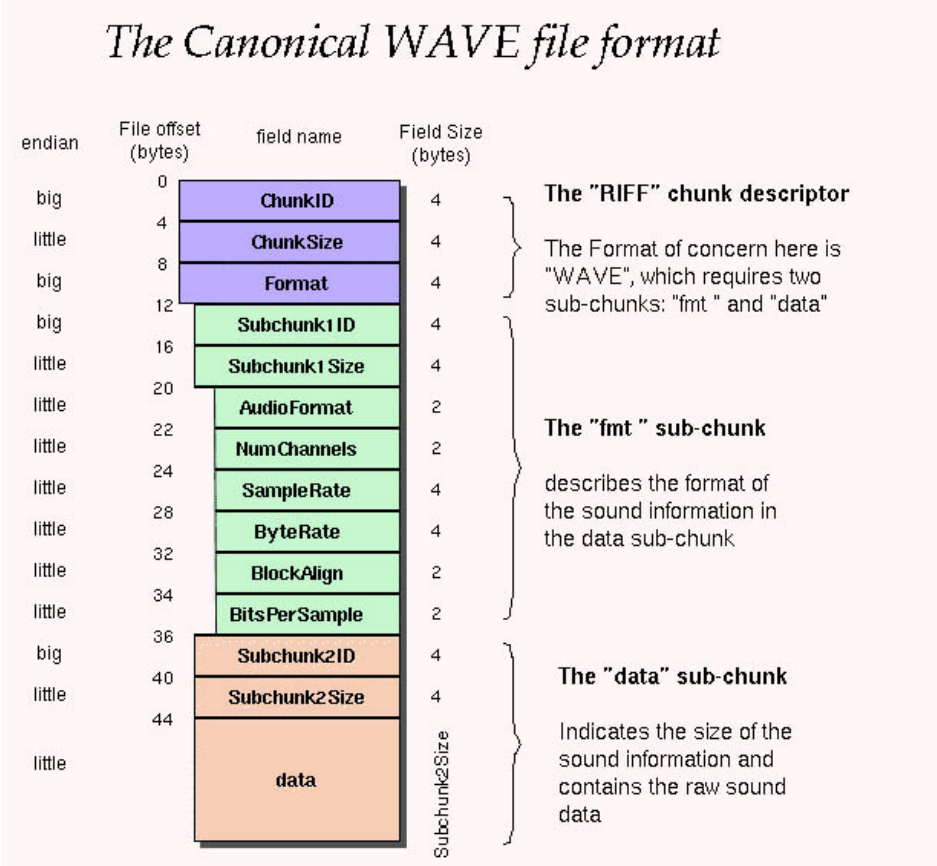



Figure 3: Canonical WAV file structure

[2]

[2]The audio file '*file_example_WAV_1MG.wav*' used in this project is free downloaded from file-examples.com.

### 2.2 Destination

## 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 4: Image encoded with Linear Hamming passed through BSC (entire)

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

Figure 5: Image encoded with Linear Hamming passed through BSC (details)

#### 4.1.3 Audio

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

## 5.1 Syndrome Decoder

### 5.1.1 Text string

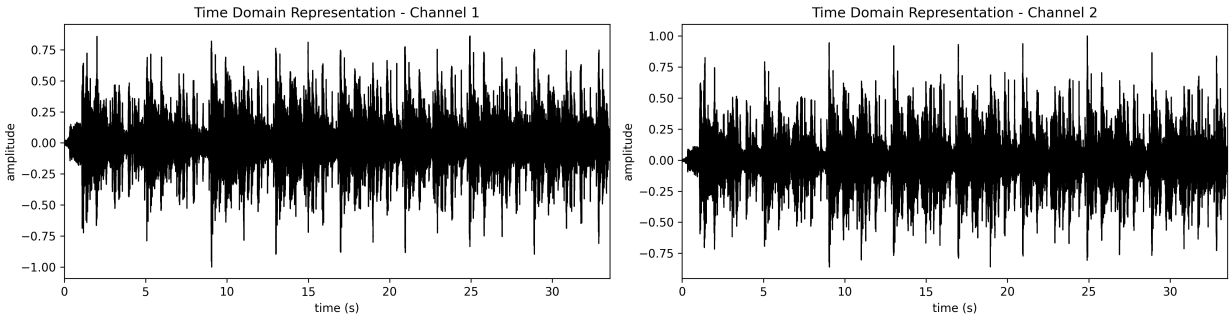
Table 5: Text string encoded with Cyclic Hamming passed through BSC

| **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 |

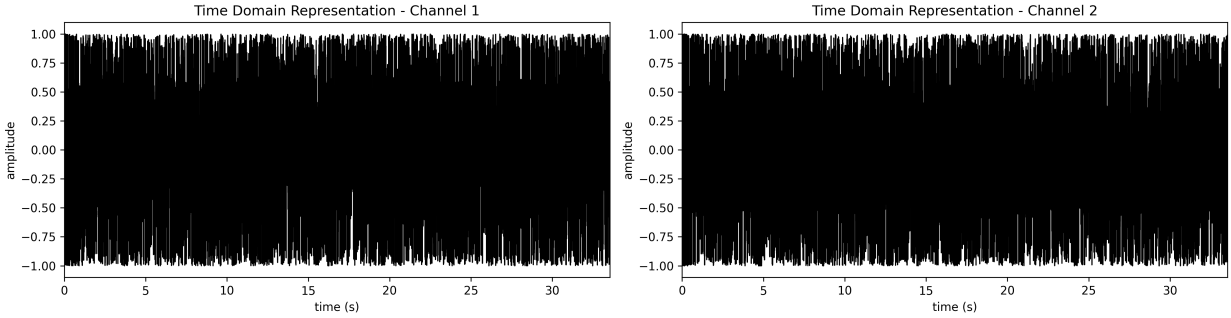
### 5.1.2 Image

### 5.1.3 Audio

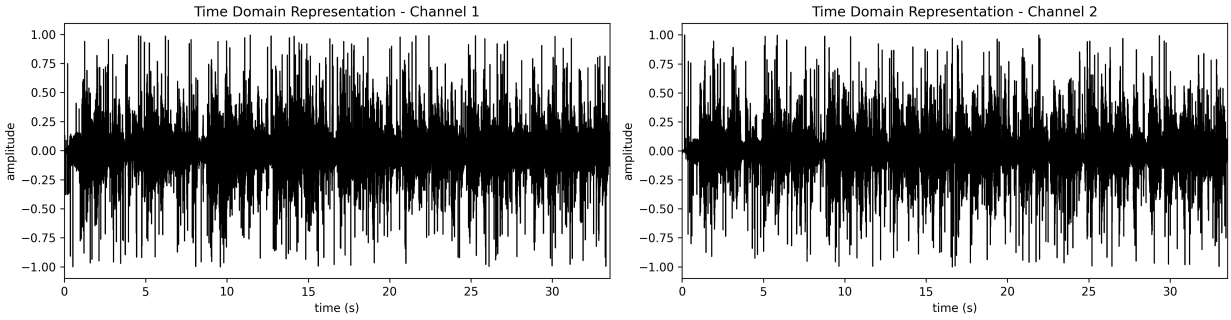
## 5.2 LFSR Decoder




(a) Original




(b) Without correction



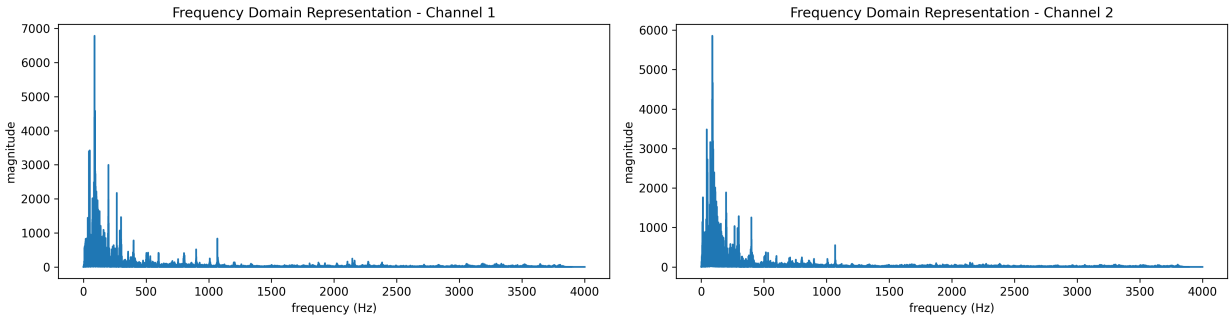

(c) Corrected

Figure 6: Audio encoded with Linear Hamming passed through BSC

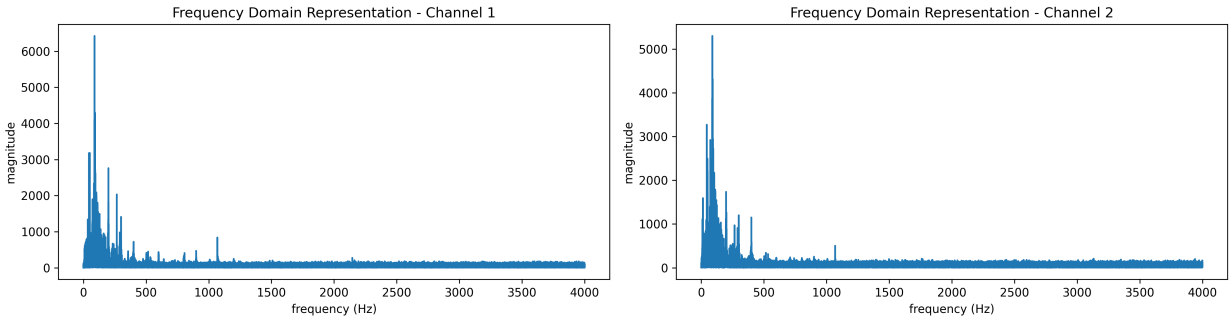
# 6 Appendix: Python source code

## 6.1 Source

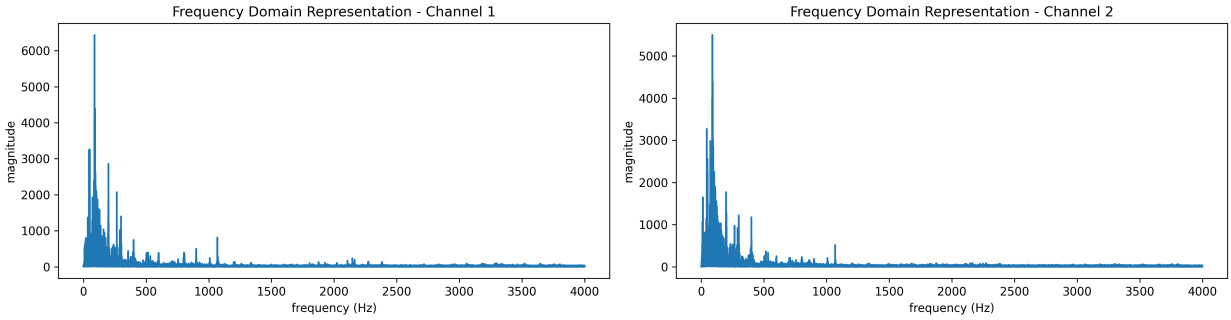
```
# Copyright (c) 2023 Chenye Yang

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

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




(a) Original




(b) Without correction




(c) Corrected

Figure 7: 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:
```

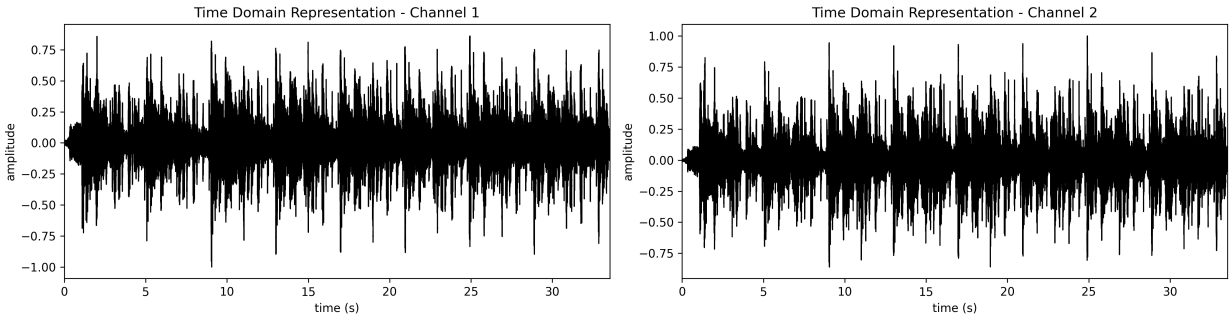
(a) Original (b) Without correction (c) Corrected

Figure 8: Image encoded with Cyclic Hamming passed through BSC (entire)

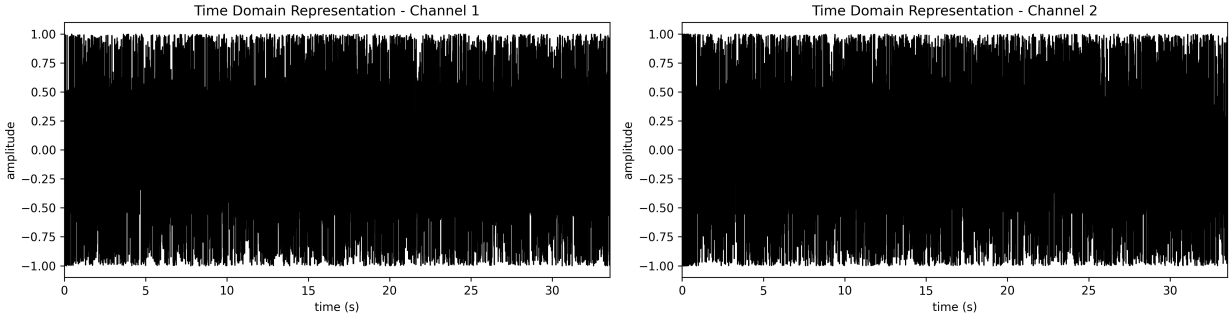
```
            byte_array = bytearray(file.read())
            self._digital_data = np.unpackbits(np.frombuffer(byte_array, dtype
                =np.uint8))


    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.
```

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

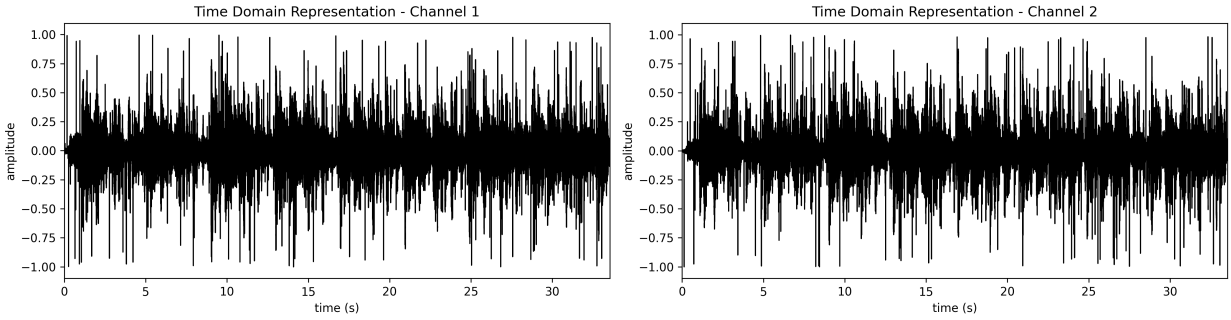
Figure 9: Image encoded with Cyclic Hamming passed through BSC (details)




(a) Original



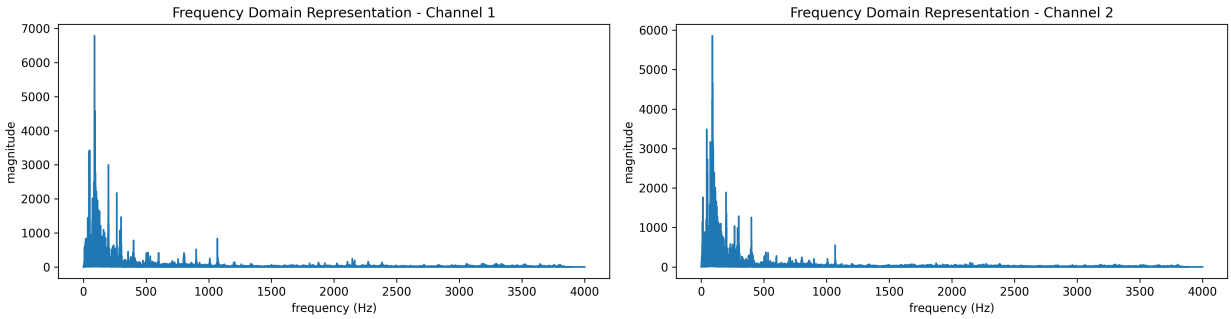

(b) Without correction



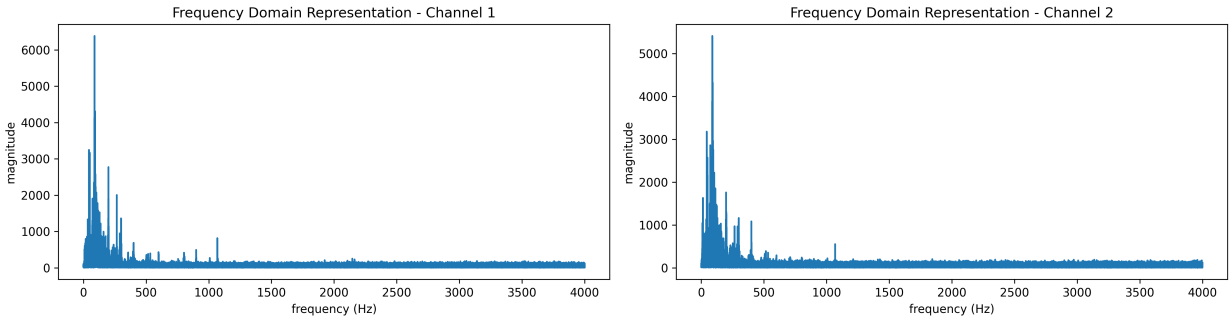

(c) Corrected

Figure 10: Audio encoded with Cyclic Hamming passed through BSC

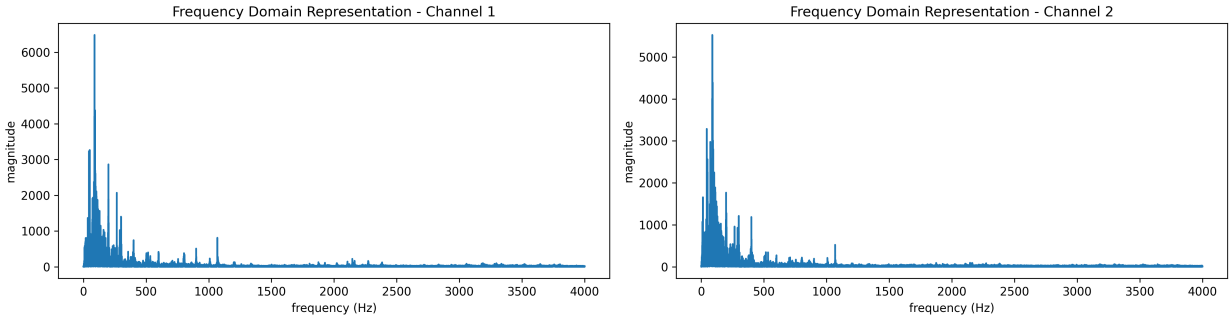
```
        @type   src_path: string
        @param src_path: source file path, with extension

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

    # Convert the image to a NumPy array
    img_array = np.array(image)

    # Get the shape of the array (height, width, channels)
    height, width, channels = img_array.shape

```




(a) Original




(b) Without correction




(c) Corrected

Figure 11: Audio encoded with Cyclic Hamming passed through BSC

```
        # 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):
        """
        Read the wav file,
        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
74              @return:  frame rate, sample width, channels
75          """
76          # Open the audio file
77          # sample_rate, audio_array = wavfile.read(src_path)
78          audio_array, sample_rate = sf.read(src_path, dtype='int16')
79          shape = audio_array.shape
80
81          # Convert the audio_array to binary format and flatten the array
82          bits = np.unpackbits(audio_array.astype(np.int16).view(np.uint8))
83
84          # Store the binary bits
85          self._digital_data = bits
86          # Store the analogue data
87          self._analogue_data = audio_array
88
89          return shape, sample_rate
90
91
92      # def read_mp3(self, src_path):
93      #     """
94      #     Read the mp3 file,
95      #     store the audio information as binary bits in _digital_data,
96      #     store the audio information as ndarray in _analogue_data.
97      #     Return the frame_rate, sample_width, channels of the audio.
98
99      #         @type  src_path: string
100     #         @param src_path: source file path, with extension
101
102     #         @rtype:   tuple
103     #         @return:  frame rate, sample width, channels
104     #     """
105     #     # Open the audio file
106     #     audio = AudioSegment.from_file(src_path, format="mp3")
107     #     frame_rate, sample_width, channels = audio.frame_rate, audio.
        sample_width, audio.channels
108
109     #     # Convert the audio to a NumPy array
110     #     audio_array = np.array(audio.get_array_of_samples())
111     #     # print(audio_array.dtype)
112
113     #     # Get the shape of the array: e.g. (2392270,)
114     #     # shape = audio_array.shape
115
116     #     # Convert the audio_array to binary format and flatten the array
117     #     bits = np.unpackbits(audio_array.astype(np.uint8))
118
119     #     # Store the binary bits
```

```
120         #       self._digital_data = bits
121         #      # Store the analogue data
122         #       self._analogue_data = audio_array
123 
124         #       return frame_rate, sample_width, channels
125 
126 
127 
128         def get_digital_data(self):
129             """
130             Get the bits to be transmitted
131 
132                 @rtype:    ndarray
133                 @return:   data bits
134             """
135             return self._digital_data
136 
137 
138         def get_analogue_data(self):
139             """
140             Get the analogue data to be transmitted
141 
142                 @rtype:    ndarray
143                 @return:   analogue data
144             """
145             return self._analogue_data
146 
147 
148 if __name__ == '__main__':
149         # test
150         source = Source()
151         source.read_txt("Resource/hardcoded.txt")
152         text_bits = source.get_digital_data()
153         print(text_bits)
154         print(type(text_bits))
155 
156         source.read_png("Resource/image.png")
157         image_bits = source.get_digital_data()
158         print(image_bits[:16])
159         image_pixels = source.get_analogue_data()
160         print(image_pixels[0][0][:2])
161 
162         # frame_rate, sample_width, channels = source.read_mp3("Resource/
                file_example_MP3_1MG.mp3")
163         # print(frame_rate, sample_width, channels)
164         # audio_bits = source.get_digital_data()
165         # print(audio_bits[:16])
166         # audio_array = source.get_analogue_data()
167         # print(audio_array[:2])
168 
169         source.read_wav("Resource/file_example_WAV_1MG.wav")
170         audio_bits = source.get_digital_data()
171         print(audio_bits[:16])
172         audio_array = source.get_analogue_data()
```

```
    print(audio_array[:2])
```

## 6.2 Channel

```
# Copyright (c) 2023 Chenye Yang

import numpy as np
import logging

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



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

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

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

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

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

    return H


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

        @type  H: ndarray
        @param H: parity-check matrix
    """
    # Get the size of the parity-check matrix
    _, n = H.shape

    # Create a table of all possible syndromes and their corresponding error
        vectors (syndrome look-up table)
    coset_leader = np.vstack((np.zeros(n, dtype=np.uint8), np.eye(n, dtype=np.
        uint8)))
    possible_syndromes = (coset_leader @ H.T) % 2
```

```
    syndrome_table = {str(possible_syndromes[i]) : coset_leader[i] for i in
        range(n+1)}

    return syndrome_table



class Linear_Code:
    """
    (7, 4) Systematic Linear Block (Hamming) Code
    """
    def __init__(self):
        self.n, self.k = 7, 4
        self.G = np.array([[1, 1, 0, 1, 0, 0, 0],
                           [0, 1, 1, 0, 1, 0, 0],
                           [1, 1, 1, 0, 0, 1, 0],
                           [1, 0, 1, 0, 0, 0, 1]], dtype=np.uint8)
        self.H = np.array([[1, 0, 0, 1, 0, 1, 1],
                           [0, 1, 0, 1, 1, 1, 0],
                           [0, 0, 1, 0, 1, 1, 1]], dtype=np.uint8)
        self.syndrome_table = {'[0 0 0]' : np.array([0, 0, 0, 0, 0, 0, 0]),
                               '[1 0 0]' : np.array([1, 0, 0, 0, 0, 0, 0]),
                               '[0 1 0]' : np.array([0, 1, 0, 0, 0, 0, 0]),
                               '[0 0 1]' : np.array([0, 0, 1, 0, 0, 0, 0]),
                               '[1 1 0]' : np.array([0, 0, 0, 1, 0, 0, 0]),
                               '[0 1 1]' : np.array([0, 0, 0, 0, 1, 0, 0]),
                               '[1 1 1]' : np.array([0, 0, 0, 0, 0, 1, 0]),
                               '[1 0 1]' : np.array([0, 0, 0, 0, 0, 0, 1])}

    def encoder_systematic(self, bits):
        """
        Systematic - Encode the to-be-transmitted binary bits message with
            (7,4) hamming encoder, return the to-be-transmitted codewords

            @type   bits: ndarray
            @param  bits: TX message

            @rtype:    ndarray
            @return:   TX codewords
        """
        encoded_array = np.zeros((len(bits) // self.k * self.n), dtype=np.
            uint8)

        for i in range(0, len(bits), self.k):
            message = bits[i:i + self.k]
            codeword = np.dot(message, self.G) % 2
            encoded_array[i // self.k * self.n:(i // self.k + 1) * self.n] =
                codeword

        return encoded_array


    def decoder_systematic(self, encoded_array):
        """
```

```
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
```

```
    tx_msg = np.array([0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1], dtype=np.uint8)
    print("Original bits:", tx_msg)

    # Channel
    channel = Channel()

    # Linear Code
    linear_code = Linear_Code()

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

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

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

    # Decoding
    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

```

```
27
28      def set_analogue_data(self, array):
29          """
30          Record the received analogue data
31
32              @type  array: ndarray
33              @param array: analogue data array
34          """
35          self._analogue_data = array
36
37
38      def get_digital_data(self):
39          """
40          Get the received digital data
41
42              @rtype:   ndarray
43              @return:  data bits
44          """
45          return self._digital_data
46
47
48      def get_analogue_data(self):
49          """
50          Get the received analogue data
51
52              @rtype:   ndarray
53              @return:  analogue data
54          """
55          return self._analogue_data
56
57
58      def write_txt(self, dest_path):
59          """
60          Write data to a text file
61
62              @type  dest_path: string
63              @param dest_path: destination file path, with extension
64          """
65          byte_array = bytearray(np.packbits(self._digital_data).tobytes())
66          with open(dest_path, 'wb') as file:
67              file.write(byte_array)
68
69
70      def write_png_from_digital(self, dest_path, height, width, channels):
71          """
72          Write color information in bits array to a png file, at the same time
                store the color information as ndarray in _analogue_data
73
74              @type  dest_path: string
75              @param dest_path: destination file path, with extension
76
77              @type  height: int
78              @param height: height of the image
79
```

```
            @type  width: int
            @param width: width of the image

            @type  channels: int
            @param channels: channels of the image
        """
        # Convert the bit array to uint8 array
        uint8_array = np.packbits(self._digital_data)

        # Reshape the array to a 3D array of pixels (height, width, channels)
        pixels = uint8_array.reshape(height, width, channels)

        # Store the analogue data
        self._analogue_data = pixels

        # Create a new image from the pixel values
        new_image = Image.fromarray(pixels)

        # Save the new image to a file
        new_image.save(dest_path)


    def write_png_from_analogue(self, dest_path):
        """
        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

            @type  dest_path: string
            @param dest_path: destination file path, with extension
        """
        # Create a new image from the pixel values
        new_image = Image.fromarray(self._analogue_data)

        # Save the new image to a file
        new_image.save(dest_path)

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

        # Store the binary bits
        self._digital_data = bits


    def write_wav_from_digital(self, shape, sample_rate, dest_path):
        """
        Write audio information in bits array to a wav file, at the same time
            store the audio information as ndarray in _analogue_data

            @type  dest_path: string
            @param dest_path: destination file path, with extension
        """
        # Convert the bit array to int16 array
        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         #     """
```

```
    #       Write audio information in audio array to a mp3 file

    #           @type  dest_path: string
    #           @param dest_path: destination file path, with extension
    #       """
    #       # Create a new AudioSegment object from the modified audio array
    #       modified_audio = AudioSegment(self._analogue_data.tobytes(),
        frame_rate, sample_width, channels)

    #       # Export the modified audio to a new MP3 file
    #       modified_audio.export(dest_path, format="mp3")



if __name__ == '__main__':
    # test
    bits = np.array([0, 1, 0, 0, 1, 0, 0, 0]) # H
    destination = Destination()
    destination.set_data(bits)
    destination.write_file("output.txt")
```

### 6.4 Utilities

```
# Copyright (c) 2023 Chenye Yang

import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft


def plot_wav_time_domain(audio_array, sample_rate, dest_path):
    """
    Plot the audio signal in the time domain.

        @type  audio_array: ndarray
        @param audio_array: audio data array

        @type  sample_rate: int
        @param sample_rate: sample rate of the audio

        @type  dest_path: string
        @param dest_path: destination file path, with extension
    """
    # Copy the audio array
    data = audio_array.copy()
    # Normalize to [-1, 1]
    data = data / np.max(np.abs(data),axis=0)

    times = np.arange(len(data))/float(sample_rate)

    # Prepare the subplots
    fig, axs = plt.subplots(1, 2, figsize=(15, 4))

```

```
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
```

```
    # Frequency domain representation for channel 2
    fft_out_channel_2 = fft(data[:, 1])
    magnitude_spectrum_channel_2 = np.abs(fft_out_channel_2)
    frequencies_channel_2 = np.linspace(0, sample_rate, len(
        magnitude_spectrum_channel_2))

    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
    axs[1].set_xlabel('frequency (Hz)')
    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)

```

```
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):
```

```
                return None
        result = 0
        data = bitRev(data, len(gen)-1)
        for row in gen:
                result = result ^ ((data & 1)*row)
                data = data >> 1
        return result


def genMatrixDecmial2Ndarray(genMatrix_decimal, n):
        """
        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():
```

```
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,
            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      """
83      Source - Channel encoder - Channel - Channel decoder - Destination
84
85      wav    - (7,4) linear    - bsc     - (7,4) linear    - wav
86      """
87      src = source.Source()
88      chl = channel.Channel()
89      linear_code = channel.Linear_Code()
90      dest = destination.Destination()
91
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, "
            Result/wav-time-domain-TX.png")
95      plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, "
            Result/wav-frequency-domain-TX.png")
96
97      # tx_msg = src.get_analogue_data()
98      # rx_msg = tx_msg
99      # dest.set_analogue_data(rx_msg)
100     # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
            Result/wav-time-domain-RX.png")
101     # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate
            , "Result/wav-frequency-domain-RX.png")
102     # dest.write_wav_from_analogue(sample_rate, "Result/hamming-bsc-output.wav
            ")
103
104
105     tx_msg = src.get_digital_data()
106
107     tx_codeword = linear_code.encoder_systematic(tx_msg)
108
109     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
110
111     # without error correction
112     rx_msg = linear_code.decoder_systematic(rx_codeword)
113
114     dest.set_digital_data(rx_msg)
115     dest.write_wav_from_digital(shape, sample_rate, "Result/linear-bsc-output.
            wav")
116
117     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():
```

```
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,
            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      """
92      Source - Channel encoder - Channel - Channel decoder - Destination
93
94      wav    - (n,k) cyclic    - bsc     - (n,k) cyclic    - wav
95      """
96      src = source.Source()
97      chl = channel.Channel()
98      cyclic_code = channel.Cyclic_Code(G)
99      dest = destination.Destination()
100
101
102     shape, sample_rate = src.read_wav("Resource/file_example_WAV_1MG.wav")
103     plot_wav.plot_wav_time_domain(src.get_analogue_data(), sample_rate, "
            Result/wav-time-domain-TX.png")
104     plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, "
            Result/wav-frequency-domain-TX.png")
105
106     # tx_msg = src.get_analogue_data()
107     # rx_msg = tx_msg
108     # dest.set_analogue_data(rx_msg)
109     # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
            Result/wav-time-domain-RX.png")
110     # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate
            , "Result/wav-frequency-domain-RX.png")
111     # dest.write_wav_from_analogue(sample_rate, "Result/hamming-bsc-output.wav
            ")
112
113
114     tx_msg = src.get_digital_data()
115
116     tx_codeword = cyclic_code.encoder_systematic(tx_msg)
117
118     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
119
120     # without error correction
121     rx_msg = cyclic_code.decoder_systematic(rx_codeword)
122
123     dest.set_digital_data(rx_msg)
124     dest.write_wav_from_digital(shape, sample_rate, "Result/cyclic-bsc-output.
            wav")
125
126     plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
            Result/cyclic-bsc-wav-time-domain-RX.png")
```

```
127     plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
            "Result/cyclic-bsc-wav-frequency-domain-RX.png")
128
129     # with error correction
130     estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
131     rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword)
132
133     dest.set_digital_data(rx_msg)
134     dest.write_wav_from_digital(shape, sample_rate, "Result/cyclic-bsc-output-
            syndrome-corrected.wav")
135
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()
```