

EEC269A - Error Correcting Codes I

Project Report

Chenyue Yang, Pranav Kharche, Parisa Oftadeh

June 5, 2023

Contents

1 Workload	2
2 Source & Destination	2
2.1 Source	2
2.1.1 Text string	2
2.1.2 Image	2
2.1.3 Audio	2
2.1.4 Copyright	3
2.2 Destination	3
3 Binary Symmetric Channel (BSC)	3
4 (7,4) Systematic Linear Block (Hamming) Code	3
4.1 Syndrome Decoder	3
4.1.1 Text string	3
4.1.2 Image	3
4.1.3 Audio	3
5 (n, k) Systematic Cyclic (Hamming) Code	3
5.1 Syndrome Decoder	3
5.1.1 Text string	3
5.1.2 Image	6
5.1.3 Audio	6
5.2 LFSR Decoder	6
6 Appendix: Python source code	9
6.1 Source	9
6.2 Channel	12
6.3 Destination	16
6.4 Utilities	20
6.5 Linear code	24
6.6 Cyclic code	27

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 the text file, the following string is stored in ASCII format:

Hello World!
EEC269A Error Correcting Code Demo

Each character is represented by 8 bits. 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

1

2.1.3 Audio

2

¹The image file 'image.png' used in this project is photographed by *Chenye Yang* at Joshua Tree National Park.

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



(a) Original

(b) Without correction

(c) Corrected

Figure 1: Image encoded with Linear Hamming passed through BSC (entire)

2.1.4 Copyright

2.2 Destination

3 Binary Symmetric Channel (BSC)

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

4.1 Syndrome Decoder

4.1.1 Text string

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

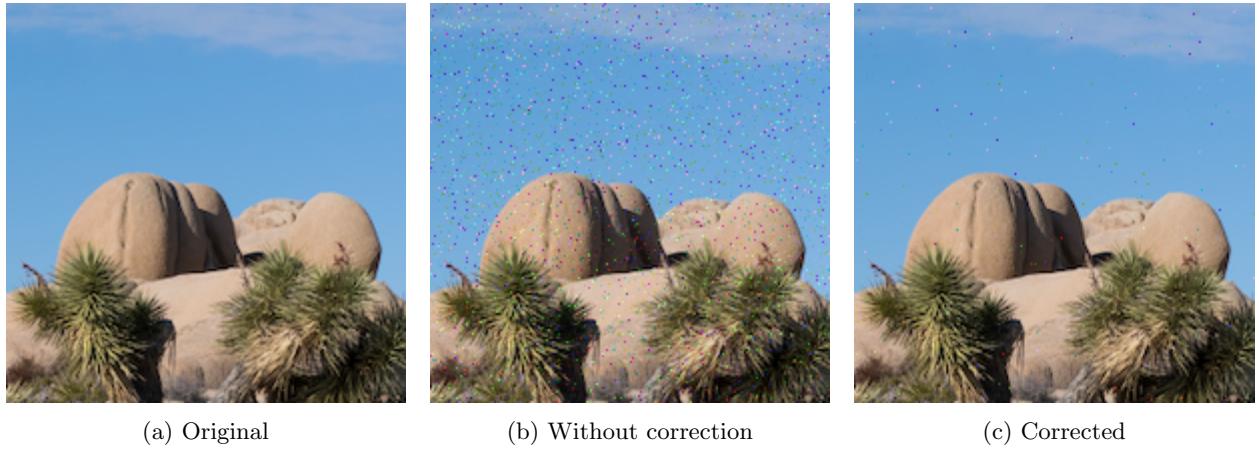
4.1.3 Audio

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

5.1 Syndrome Decoder

5.1.1 Text string

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



(a) Original

(b) Without correction

(c) Corrected

Figure 2: Image encoded with Linear Hamming passed through BSC (details)

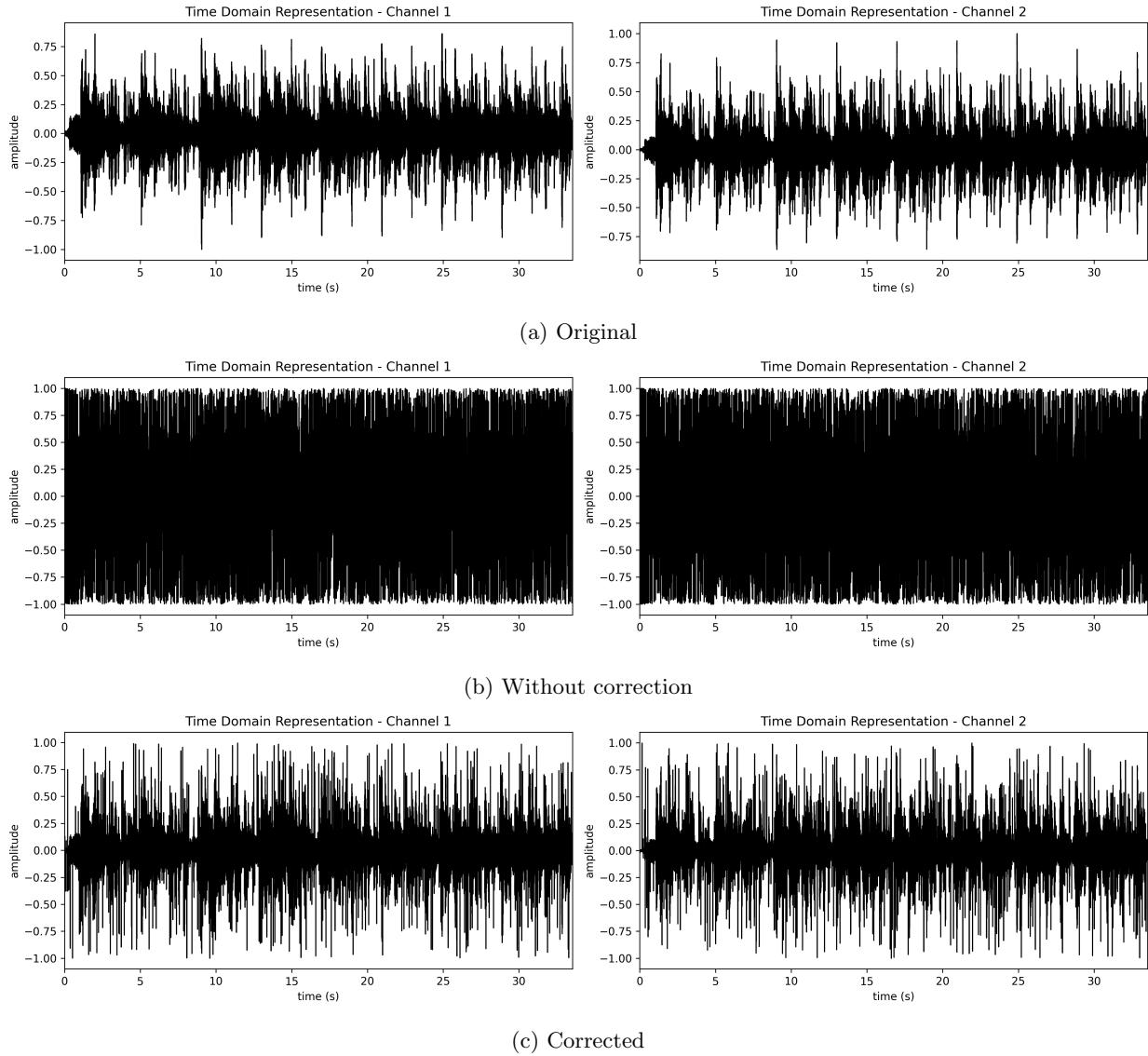


Figure 3: Audio encoded with Linear Hamming passed through BSC

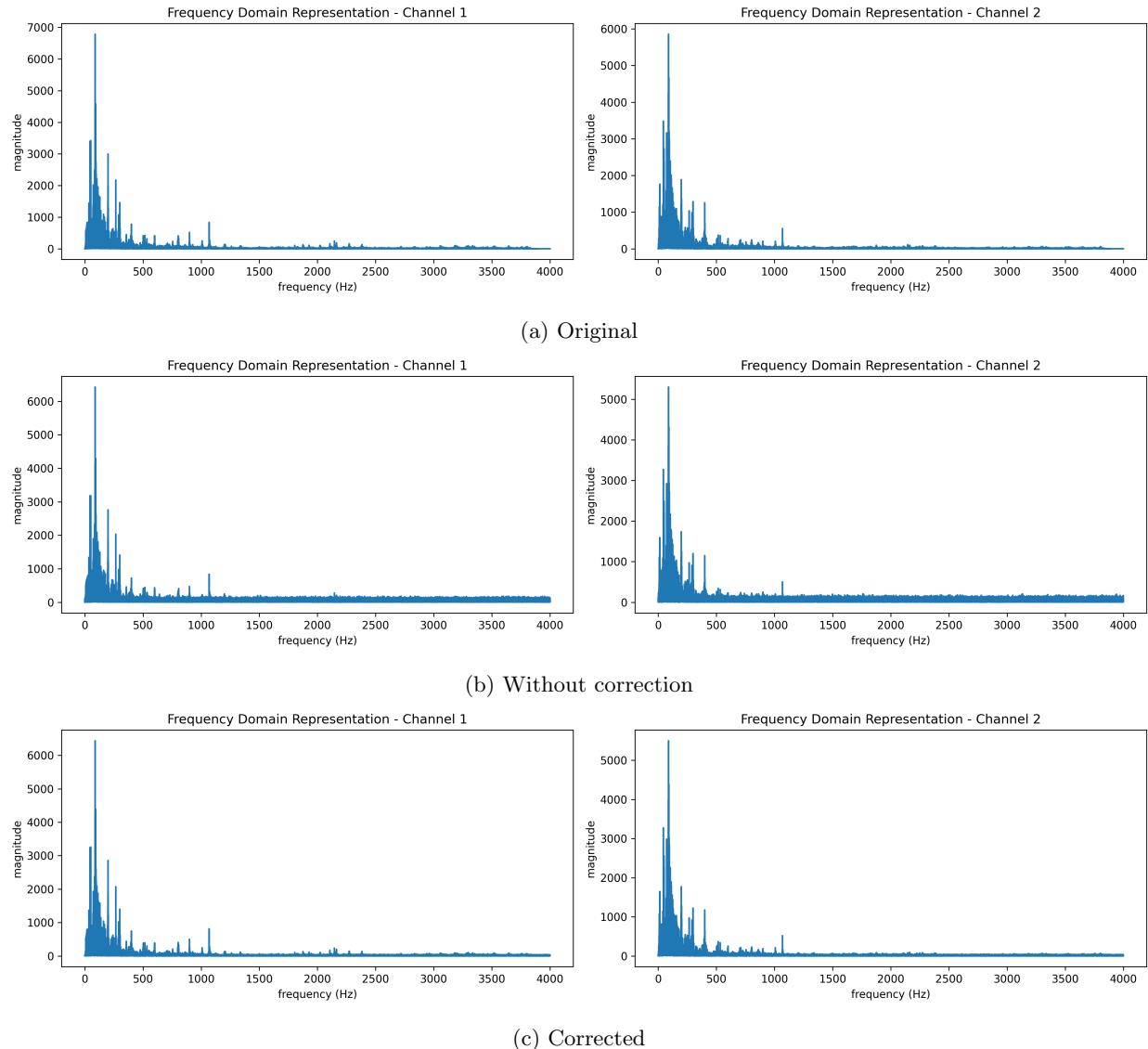


Figure 4: Audio encoded with Linear Hamming passed through BSC



(a) Original

(b) Without correction

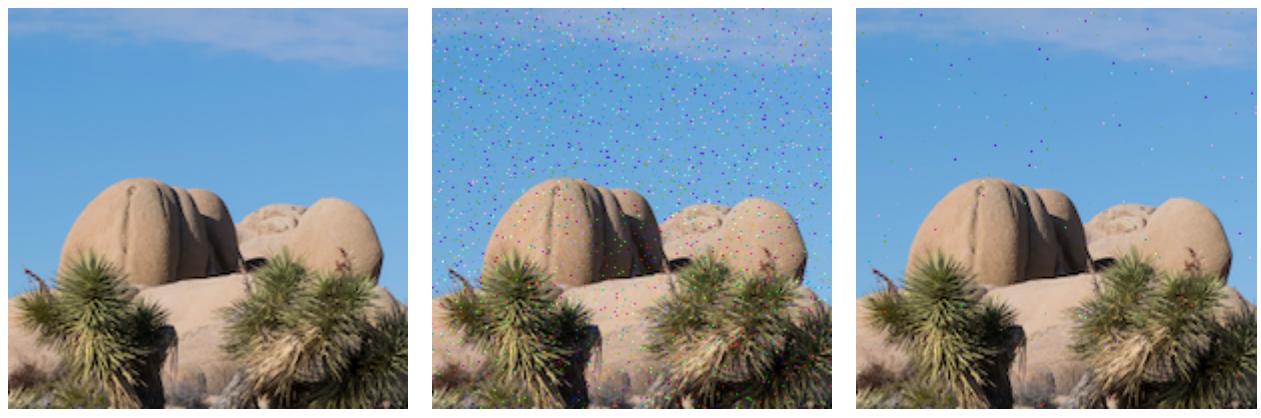
(c) Corrected

Figure 5: Image encoded with Cyclic Hamming passed through BSC (entire)

5.1.2 Image

5.1.3 Audio

5.2 LFSR Decoder



(a) Original

(b) Without correction

(c) Corrected

Figure 6: Image encoded with Cyclic Hamming passed through BSC (details)

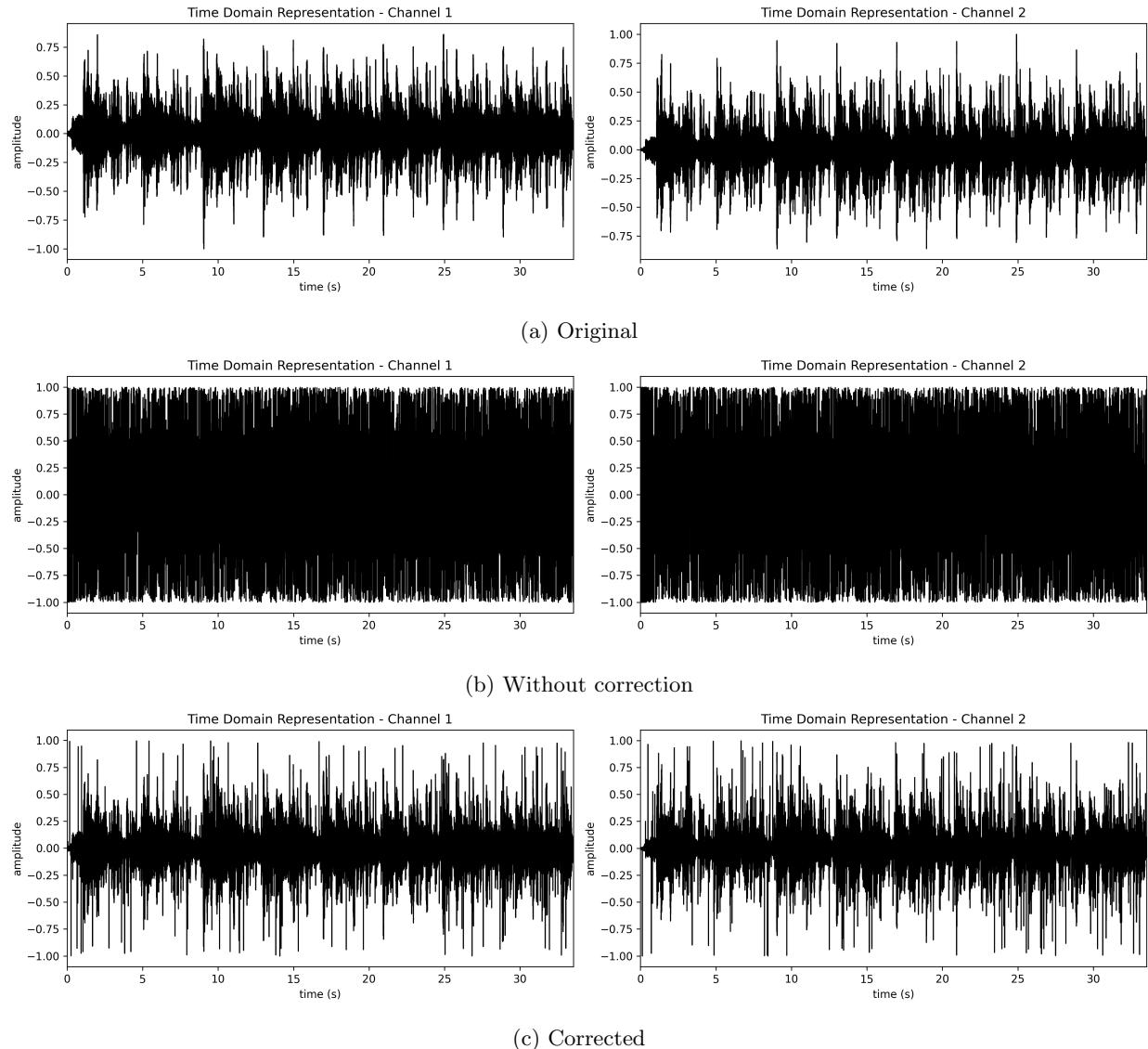


Figure 7: Audio encoded with Cyclic Hamming passed through BSC

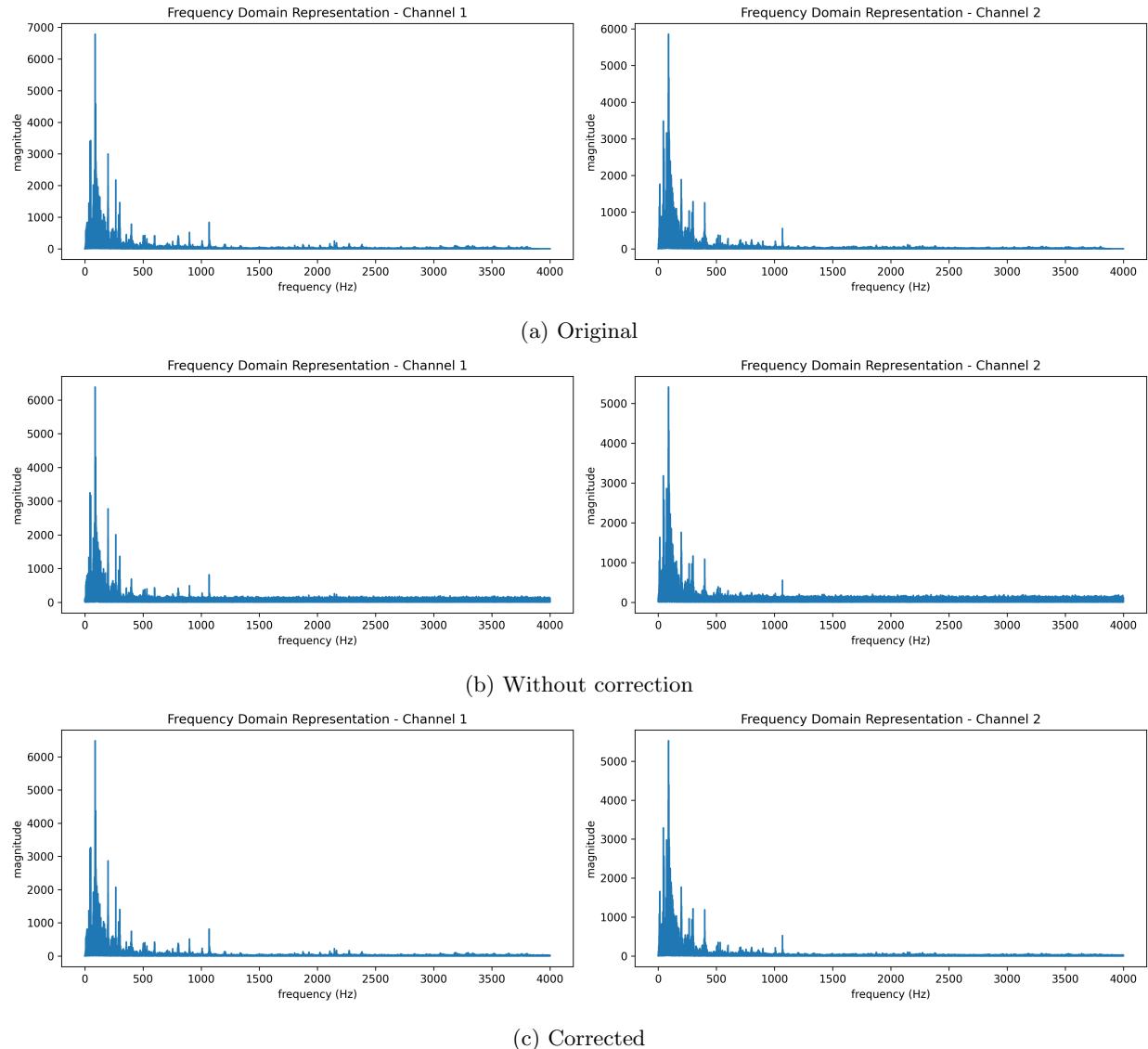


Figure 8: Audio encoded with Cyclic Hamming passed through BSC

6 Appendix: Python source code

6.1 Source

```
1 # Copyright (c) 2023 Chenye Yang
2
3 import numpy as np
4 from PIL import Image
5 import soundfile as sf
6 # from scipy.io import wavfile
7 import logging
8
9 # Create a logger in this module
10 logger = logging.getLogger(__name__)
11
12
13 class Source:
14     """
15     The data source
16     """
17
18     def read_txt(self, src_path):
19         """
20             Read the text file as binary bits
21
22             @type src_path: string
23             @param src_path: source file path, with extension
24         """
25         with open(src_path, 'rb') as file:
26             byte_array = bytearray(file.read())
27             self._digital_data = np.unpackbits(np.frombuffer(byte_array, dtype=np.uint8))
28
29
30     def read_png(self, src_path):
31         """
32             Read the png file,
33             store the color information as binary bits in _digital_data,
34             store the color information as ndarray in _analogue_data.
35             Return the height, width, channels of the image.
36
37             @type src_path: string
38             @param src_path: source file path, with extension
39
40             @rtype: tuple
41             @return: height, width, channels
42         """
43         # Open the image file
44         image = Image.open(src_path)
45
46         # Convert the image to a NumPy array
47         img_array = np.array(image)
48
```

```

49     # Get the shape of the array (height , width , channels)
50     height , width , channels = img_array.shape
51
52     # Convert the img_array to binary format and flatten the array
53     bits = np.unpackbits(img_array.astype(np.uint8))
54
55     # Store the binary bits
56     self._digital_data = bits
57     # Store the analogue data
58     self._analogue_data = img_array
59
60     return height , width , channels
61
62
63 def read_wav(self , src_path):
64     """
65     Read the wav file ,
66     store the audio information as binary bits in _digital_data ,
67     store the audio information as ndarray in _analogue_data .
68     Return the frame_rate , sample_width , channels of the audio .
69
70     @type src_path: string
71     @param src_path: source file path , with extension
72
73     @rtype: tuple
74     @return: frame rate , sample width , channels
75     """
76
77     # Open the audio file
78     # sample_rate , audio_array = wavfile.read(src_path)
79     audio_array , sample_rate = sf.read(src_path , dtype='int16')
80     shape = audio_array.shape
81
82     # Convert the audio_array to binary format and flatten the array
83     bits = np.unpackbits(audio_array.astype(np.int16).view(np.uint8))
84
85     # Store the binary bits
86     self._digital_data = bits
87     # Store the analogue data
88     self._analogue_data = audio_array
89
90     return shape , sample_rate
91
92
93     # def read_mp3(self , src_path):
94     """
95     # Read the mp3 file ,
96     # store the audio information as binary bits in _digital_data ,
97     # store the audio information as ndarray in _analogue_data .
98     # Return the frame_rate , sample_width , channels of the audio .
99
100    #
101    #     @type src_path: string
102    #     @param src_path: source file path , with extension
103
104    #

```

```

103     #         @return: frame rate, sample width, channels
104     #
105     # Open the audio file
106     # audio = AudioSegment.from_file(src_path, format="mp3")
107     # frame_rate, sample_width, channels = audio.frame_rate, audio.
108     #           sample_width, audio.channels
109
110    #     # Convert the audio to a NumPy array
111    #     audio_array = np.array(audio.get_array_of_samples())
112    #     # print(audio_array.dtype)
113
114    #     # Get the shape of the array: e.g. (2392270,)
115    #     # shape = audio_array.shape
116
117    #     # Convert the audio_array to binary format and flatten the array
118    #     bits = np.unpackbits(audio_array.astype(np.uint8))
119
120    #     # Store the binary bits
121    #     self._digital_data = bits
122    #     # Store the analogue data
123    #     self._analogue_data = audio_array
124
125
126
127
128 def get_digital_data(self):
129     """
130     Get the bits to be transmitted
131
132         @rtype: ndarray
133         @return: data bits
134     """
135
136
137
138 def get_analogue_data(self):
139     """
140     Get the analogue data to be transmitted
141
142         @rtype: ndarray
143         @return: analogue data
144     """
145
146
147
148 if __name__ == '__main__':
149     # test
150     source = Source()
151     source.read_txt("Resource/hardcoded.txt")
152     text_bits = source.get_digital_data()
153     print(text_bits)
154     print(type(text_bits))
155

```

```

156     source.read_png("Resource/image.png")
157     image_bits = source.get_digital_data()
158     print(image_bits[:16])
159     image_pixels = source.get_analogue_data()
160     print(image_pixels[0][0][:2])
161
162     # frame_rate, sample_width, channels = source.read_mp3("Resource/
163     # file-example-MP3_1MG.mp3")
164     # print(frame_rate, sample_width, channels)
165     # audio_bits = source.get_digital_data()
166     # print(audio_bits[:16])
167     # audio_array = source.get_analogue_data()
168     # print(audio_array[:2])
169
170     source.read_wav("Resource/file-example-WAV_1MG.wav")
171     audio_bits = source.get_digital_data()
172     print(audio_bits[:16])
173     audio_array = source.get_analogue_data()
174     print(audio_array[:2])

```

6.2 Channel

```

1  # Copyright (c) 2023 Chenye Yang
2
3  import numpy as np
4  import logging
5
6  # Create a logger in this module
7  logger = logging.getLogger(__name__)
8
9
10
11 def create_parity_check_matrix(G):
12     """
13         Create the parity-check matrix,  $H = [I_{n-k} \mid P.T]$ 
14
15         @type G: ndarray
16         @param G: generator matrix in systematic form,  $G = [P \mid I_k]$ 
17
18         @rtype: ndarray
19         @return: parity-check matrix
20     """
21
22     # Get the size of the generator matrix
23     k, n = G.shape
24
25     # Create the parity-check matrix
26     #  $G = [P \mid I_k]$ , Extract P
27     P = G[:, :n-k]
28
29     #  $H = [I_{n-k} \mid P.T]$ 
30     H = np.hstack((np.eye(n-k, dtype=np.uint8), P.T))
31
32     return H

```

```

32
33
34 def create_syndrome_table(H):
35     """
36     Create a table of all possible syndromes and their corresponding error
37     vectors (syndrome look-up table)
38
39     @type H: ndarray
40     @param H: parity-check matrix
41     """
42
43     # Get the size of the parity-check matrix
44     _, n = H.shape
45
46     # Create a table of all possible syndromes and their corresponding error
47     # vectors (syndrome look-up table)
48     coset_leader = np.vstack((np.zeros(n, dtype=np.uint8), np.eye(n, dtype=np.
49     uint8)))
50     possible_syndromes = (coset_leader @ H.T) % 2
51     syndrome_table = {str(possible_syndromes[i]) : coset_leader[i] for i in
52     range(n+1)}
53
54     return syndrome_table
55
56
57 class Linear_Code:
58     """
59     (7, 4) Systematic Linear Block (Hamming) Code
60     """
61
62     def __init__(self):
63         self.n, self.k = 7, 4
64         self.G = np.array([[1, 1, 0, 1, 0, 0, 0],
65                           [0, 1, 1, 0, 1, 0, 0],
66                           [1, 1, 1, 0, 0, 1, 0],
67                           [1, 0, 1, 0, 0, 0, 1]], dtype=np.uint8)
68         self.H = np.array([[1, 0, 0, 1, 0, 1, 1],
69                           [0, 1, 0, 1, 1, 1, 0],
70                           [0, 0, 1, 0, 1, 1, 1]], dtype=np.uint8)
71         self.syndrome_table = {'[0_0_0]' : np.array([0, 0, 0, 0, 0, 0, 0]),
72                               '[1_0_0]' : np.array([1, 0, 0, 0, 0, 0, 0]),
73                               '[0_1_0]' : np.array([0, 1, 0, 0, 0, 0, 0]),
74                               '[0_0_1]' : np.array([0, 0, 1, 0, 0, 0, 0]),
75                               '[1_1_0]' : np.array([0, 0, 0, 1, 0, 0, 0]),
76                               '[0_1_1]' : np.array([0, 0, 0, 0, 1, 0, 0]),
77                               '[1_1_1]' : np.array([0, 0, 0, 0, 0, 1, 0]),
78                               '[1_0_1]' : np.array([0, 0, 0, 0, 0, 0, 1])}
79
80     def encoder_systematic(self, bits):
81         """
82         Systematic - Encode the to-be-transmitted binary bits message with
83         (7,4) hamming encoder, return the to-be-transmitted codewords
84
85         @type bits: ndarray
86         @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.
86                             uint8)
87
88     for i in range(0, len(bits), self.k):
89         message = bits[i:i + self.k]
90         codeword = np.dot(message, self.G) % 2
91         encoded_array[i // self.k * self.n:(i // self.k + 1) * self.n] =
92             codeword
93
94
95     return encoded_array
96
97
98
99     def decoder_systematic(self, encoded_array):
100        """
101        Systematic – Decode the received binary bits codeword with (7,4)
102        Hamming decoder, return the received message
103
104        @type  encoded_array: ndarray
105        @param encoded_array: RX codewords
106
107        @rtype:    ndarray
108        @return:   RX message
109    """
110
111     decoded_array = np.zeros((len(encoded_array) // self.n * self.k),
112                             dtype=np.uint8)
113
114     err_count = 0
115
116     for i in range(0, len(encoded_array), self.n):
117         received_codeword = encoded_array[i:i + self.n]
118         syndrome = np.dot(self.H, received_codeword) % 2
119         if np.any(syndrome): # If there are errors, count and keep it
120             err_count += 1
121         decoded_array[i // self.n * self.k : (i // self.n + 1) * self.k] =
122             received_codeword[self.n - self.k:self.n]
123
124     logger.info(f"Error_codeword_rate:{err_count/(len(encoded_array)//
125                                         self.n)}")
126
127     return decoded_array
128
129
130
131     def corrector_syndrome(self, received_array):
132        """
133        Systematic – Correct the received binary bits codeword with (7,4)
134        Hamming syndrome look-up table corrector,
135        return the estimated TX codeword = (RX codeword + error pattern)
136
137        @type  received_array: ndarray
138        @param received_array: RX codewords

```

```

128
129         @rtype:    ndarray
130         @return:   estimated TX codewords
131     """
132     corrected_array = received_array.copy()
133
134     for i in range(0, len(received_array), self.n):
135         received_codeword = received_array[i:i + self.n]
136         syndrome = np.dot(self.H, received_codeword) % 2
137         if np.any(syndrome):
138             corrected_array[i:i + self.n] = (received_codeword + self.
139                                             syndrome_table[str(syndrome)]) % 2
140
141
142
143 class Cyclic_Code(Linear_Code):
144     """
145     (n, k) Systematic Cyclic (Hamming) Code
146     """
147     def __init__(self, G):
148         self.G = G
149         self.k, self.n = self.G.shape
150         self.H = create_parity_check_matrix(self.G)
151
152         self.syndrome_table = create_syndrome_table(self.H)
153
154
155     def corrector_lfsr(self, received_array):
156         """
157         Systematic - Correct the received binary bits codeword with (n, k)
158         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
189     output_bits = np.copy(input_bits)
190     for i in range(len(output_bits)):
191         if np.random.random() < p:
192             output_bits[i] = 1 - output_bits[i] # flip the bit
193
194
195     if __name__ == '__main__':
196         # test
197         tx_msg = np.array([0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1], dtype=np.uint8)
198         print("Original_bits:", tx_msg)
199
200     # Channel
201     channel = Channel()
202
203     # Linear Code
204     linear_code = Linear_Code()
205
206     # Encoding
207     tx_codewords = linear_code.encoder_systematic(tx_msg)
208     print("Encoded_bits:", tx_codewords)
209
210     # Passing through the channel
211     rx_codewords = channel.binary_symmetric_channel(tx_codewords, 0.1)
212     print("Bits_after_channel:", rx_codewords)
213
214     # Correction with syndrome look-up table
215     estimated_tx_codewords = linear_code.corrector_syndrome(rx_codewords)
216     print("Estimated_TX_bits:", estimated_tx_codewords)
217
218     # Decoding
219     rx_msg = linear_code.decoder_systematic(estimated_tx_codewords)
220     print("Decoded_bits:", rx_msg)

```

6.3 Destination

```

1 # Copyright (c) 2023 Chenye Yang
2
3 import numpy as np
4 from PIL import Image
5 import soundfile as sf
6 # from scipy.io import wavfile
7 import logging
8

```

```

9  # Create a logger in this module
10 logger = logging.getLogger(__name__)
11
12
13 class Destination:
14     """
15     The destination
16     """
17
18     def set_digital_data(self, bits):
19         """
20             Record the received digital data
21
22             @type bits: ndarray
23             @param bits: data bits
24         """
25         self._digital_data = bits
26
27
28     def set_analogue_data(self, array):
29         """
30             Record the received analogue data
31
32             @type array: ndarray
33             @param array: analogue data array
34         """
35         self._analogue_data = array
36
37
38     def get_digital_data(self):
39         """
40             Get the received digital data
41
42             @rtype: ndarray
43             @return: data bits
44         """
45         return self._digital_data
46
47
48     def get_analogue_data(self):
49         """
50             Get the received analogue data
51
52             @rtype: ndarray
53             @return: analogue data
54         """
55         return self._analogue_data
56
57
58     def write_txt(self, dest_path):
59         """
60             Write data to a text file
61
62             @type dest_path: string

```

```

63         @param dest_path: destination file path, with extension
64         """
65     byte_array = bytearray(np.packbits(self._digital_data).tobytes())
66     with open(dest_path, 'wb') as file:
67         file.write(byte_array)
68
69
70     def write_png_from_digital(self, dest_path, height, width, channels):
71         """
72             Write color information in bits array to a png file, at the same time
73                 store the color information as ndarray in _analogue_data
74
75             @type dest_path: string
76             @param dest_path: destination file path, with extension
77
78             @type height: int
79             @param height: height of the image
80
81             @type width: int
82             @param width: width of the image
83
84             @type channels: int
85             @param channels: channels of the image
86         """
87         # Convert the bit array to uint8 array
88         uint8_array = np.packbits(self._digital_data)
89
90         # Reshape the array to a 3D array of pixels (height, width, channels)
91         pixels = uint8_array.reshape(height, width, channels)
92
93         # Store the analogue data
94         self._analogue_data = pixels
95
96         # Create a new image from the pixel values
97         new_image = Image.fromarray(pixels)
98
99         # Save the new image to a file
100        new_image.save(dest_path)
101
102    def write_png_from_analogue(self, dest_path):
103        """
104            Write color information in (height, width, channels) array to a png
105                file, at the same time store the color information as bit array in
106                _digital_data
107
108            @type dest_path: string
109            @param dest_path: destination file path, with extension
110
111            # Create a new image from the pixel values
112            new_image = Image.fromarray(self._analogue_data)
113
114            # Save the new image to a file
115            new_image.save(dest_path)

```

```

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
125             store the audio information as ndarray in _analogue_data
126
127         @type dest_path: string
128         @param dest_path: destination file path, with extension
129     """
130     # Convert the bit array to int16 array
131     int16_array = np.packbits(self._digital_data).view(np.int16)
132
133     # Reshape the array to a 2D array of samples (channels, samples)
134     audio_array = int16_array.reshape(shape)
135
136     # Store the analogue data
137     self._analogue_data = audio_array
138
139     # Write the array to a wav file
140     # wavfile.write(dest_path, sample_rate, audio_array)
141     sf.write(dest_path, audio_array, sample_rate)
142
143
144 def write_wav_from_analogue(self, sample_rate, dest_path):
145     """
146         Write audio information in audio array to a wav file, at the same time
147             store the audio information as bit array in _digital_data
148
149         @type dest_path: string
150         @param dest_path: destination file path, with extension
151     """
152     # Write the array to a wav file
153     # wavfile.write(dest_path, sample_rate, self._analogue_data)
154     sf.write(dest_path, self._analogue_data, sample_rate)
155
156     # Convert the audio_array to binary format and flatten the array
157     bits = np.unpackbits(self._analogue_data.astype(np.int16).view(np.
158         uint8))
159
160     # Store the binary bits
161     self._digital_data = bits
162
163     # def write_mp3_from_digital(self, frame_rate, sample_width, channels,
164     #     dest_path):
165     #     """
166     #         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,
173     #                                 sample_width, channels)
174
175     # Export the modified audio to a new MP3 file
176     # modified_audio.export(dest_path, format="mp3")
177
178     # def write_mp3_from_analogue(self, frame_rate, sample_width, channels,
179     #                             dest_path):
180     """
181     # Write audio information in audio array to a mp3 file
182
183     #         @type dest_path: string
184     #         @param dest_path: destination file path, with extension
185     """
186     # Create a new AudioSegment object from the modified audio array
187     # modified_audio = AudioSegment(self._analogue_data.tobytes(),
188     #                               frame_rate, sample_width, channels)
189
190     # Export the modified audio to a new MP3 file
191     # modified_audio.export(dest_path, format="mp3")
192
193 if __name__ == '__main__':
194     # test
195     bits = np.array([0, 1, 0, 0, 1, 0, 0, 0]) # H
196     destination = Destination()
197     destination.set_data(bits)
198     destination.write_file("output.txt")

```

6.4 Utilities

```

1  # Copyright (c) 2023 Chenye Yang
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from scipy.fft import fft
6
7
8  def plot_wav_time_domain(audio_array, sample_rate, dest_path):
9      """
10     Plot the audio signal in the time domain.
11
12     @type audio_array: ndarray

```

```

13     @param audio_array: audio data array
14
15     @type sample_rate: int
16     @param sample_rate: sample rate of the audio
17
18     @type dest_path: string
19     @param dest_path: destination file path, with extension
20 """
21 # Copy the audio array
22 data = audio_array.copy()
23 # Normalize to [-1, 1]
24 data = data / np.max(np.abs(data), axis=0)
25
26 times = np.arange(len(data))/float(sample_rate)
27
28 # Prepare the subplots
29 fig, axs = plt.subplots(1, 2, figsize=(15, 4))
30
31 # Time domain representation for channel 1
32 axs[0].fill_between(times, data[:, 0], color='k')
33 axs[0].set_xlim(times[0], times[-1])
34 axs[0].set_xlabel('time_(s)')
35 axs[0].set_ylabel('amplitude')
36 axs[0].set_title('Time-Domain-Representation--Channel_1')
37
38 # Time domain representation for channel 2
39 axs[1].fill_between(times, data[:, 1], color='k')
40 axs[1].set_xlim(times[0], times[-1])
41 axs[1].set_xlabel('time_(s)')
42 axs[1].set_ylabel('amplitude')
43 axs[1].set_title('Time-Domain-Representation--Channel_2')
44
45 # Display the plot
46 plt.tight_layout()
47 plt.savefig(dest_path, dpi=300)
48 plt.close()
49
50
51 def plot_wav_frequency_domain(audio_array, sample_rate, dest_path):
52 """
53 Plot the audio signal in the frequency domain.
54
55     @type audio_array: ndarray
56     @param audio_array: audio data array
57
58     @type sample_rate: int
59     @param sample_rate: sample rate of the audio
60
61     @type dest_path: string
62     @param dest_path: destination file path, with extension
63 """
64 # Copy the audio array
65 data = audio_array.copy()
66 # Normalize to [-1, 1]

```

```

67     data = data / np.max(np.abs(data), axis=0)
68
69     # Prepare the subplots
70     fig, axs = plt.subplots(1, 2, figsize=(15, 4))
71
72     # Frequency domain representation for channel 1
73     fft_out_channel_1 = fft(data[:, 0])
74     magnitude_spectrum_channel_1 = np.abs(fft_out_channel_1)
75     frequencies_channel_1 = np.linspace(0, sample_rate, len(
76         magnitude_spectrum_channel_1))
77
77     axs[0].plot(frequencies_channel_1[:int(len(frequencies_channel_1)/2)],
78                 magnitude_spectrum_channel_1[:int(len(magnitude_spectrum_channel_1)/2)])
79     # plot only first half of frequencies
80     axs[0].set_xlabel('frequency_(Hz)')
81     axs[0].set_ylabel('magnitude')
82     axs[0].set_title('Frequency_Domain_Representation--Channel_1')
83
84     # Frequency domain representation for channel 2
85     fft_out_channel_2 = fft(data[:, 1])
86     magnitude_spectrum_channel_2 = np.abs(fft_out_channel_2)
87     frequencies_channel_2 = np.linspace(0, sample_rate, len(
88         magnitude_spectrum_channel_2))
89
90     axs[1].plot(frequencies_channel_2[:int(len(frequencies_channel_2)/2)],
91                 magnitude_spectrum_channel_2[:int(len(magnitude_spectrum_channel_2)/2)])
92     # plot only first half of frequencies
93     axs[1].set_xlabel('frequency_(Hz)')
94     axs[1].set_ylabel('magnitude')
95     axs[1].set_title('Frequency_Domain_Representation--Channel_2')

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

```

```

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

```

```

18
19 def bitRev(p, pOrder = None):
20     if p == 0:
21         return 0
22     if pOrder == None:
23         pOrder = order(p)
24
25     retVal = 0
26     for i in range(pOrder+1):
27         retVal = (retVal << 1) + (p %2)
28         p = p >>1
29     return retVal
30
31 def polyDiv(dividend, divisor, dividendOrder = None, divisorOrder = None):
32     if dividendOrder == None or divisorOrder == None:
33         dividendOrder = order(dividend)
34         divisorOrder = order(divisor)
35
36     sizeDiff = dividendOrder - divisorOrder
37     rem = dividend
38     remOrder = dividendOrder
39     result = 0
40     for i in range(sizeDiff, -1, -1):
41         if(rem >> remOrder):
42             rem = rem ^ (divisor << i)
43             result = result + (1 << i)
44         remOrder -= 1
45
46     # if rem:
47     #     print(f'{dividend:b}', ':', f'{divisor:b}', '=', f'{result:b}', 'R', f'{rem:0{divisorOrder}b}', sep=' ')
48     # else:
49     #     print(f'{dividend:b}', ':', f'{divisor:b}', '=', f'{result:b}')
50     return result, rem
51
52 def findGen(n,k):
53     target = (1 << n) + 1
54     gen = (1 << (n-k)) + 3
55     parity, rem = polyDiv(target, gen, n, n-k)
56     while rem:
57         gen += 2
58         if gen >= target:
59             return None, None
60         parity, rem = polyDiv(target, gen, n, n-k)
61         logger.debug('generator =', f'{gen:b}')
62         logger.debug('parity =', f'{parity:b}')
63     return gen, parity
64
65 def buildMatrix(n, k, gen, parity):
66     genMatrix = [gen]
67     # print(f'{gen:0{n}b}')
68     for i in range(k-1):
69         nextLine = genMatrix[i] << 1

```

```

70         if (nextLine >> (n-k))%2:
71             nextLine = nextLine ^ gen
72             genMatrix.append(nextLine)
73             # print(f'{nextLine:0{n}b}')
74             logger.debug('Generator_matrix')
75         for i in range(k):
76             genMatrix[i] = bitRev(genMatrix[i], n-1)
77             logger.debug(f'{genMatrix[i]:0{n}b}')
78     return genMatrix
79
80 def findMatrix(n,k):
81     gen, parity = findGen(n,k)
82     if gen == None:
83         return None
84     return buildMatrix(n, k, gen, parity)
85
86 def encode(data, gen):
87     if order(data)+1 > len(gen):
88         return None
89     result = 0
90     data = bitRev(data, len(gen)-1)
91     for row in gen:
92         result = result ^ ((data & 1)*row)
93         data = data >> 1
94     return result
95
96
97 def genMatrixDecmial2Ndarray(genMatrix_decimal, n):
98     """
99         Convert the generator matrix in decimal form to numpy array form
100
101     @type genMatrix_decimal: list
102     @param genMatrix_decimal: generator matrix in decimal form
103
104     @type n: int
105     @param n: number of bits in a codeword
106
107     @rtype: ndarray
108     @return: generator matrix in numpy array form
109     """
110     # Convert the generator matrix in decimal form to binary form
111     genMatrix_binary = [[int(bit) for bit in f'{num:0{n}b}'] for num in
112                         genMatrix_decimal]
113
114     # Convert the binary generator matrix to a numpy array
115     G = np.array(genMatrix_binary, dtype=np.uint8)
116
117     return G

```

6.5 Linear code

```

1 # Copyright (c) 2023 Chenye Yang
2

```

```

3 import logging
4
5 from Source import source
6 from Channel import channel
7 from Destination import destination
8 from Utils import plot_wav
9
10
11 # Configure the logging
12 logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
13
14 # Create a logger in the main module
15 logger = logging.getLogger(__name__)
16
17
18 def linear_txt():
19     """
20         Source - Channel encoder - Channel - Channel decoder - Destination
21
22         txt      - (7,4) linear      - bsc      - (7,4) linear      - txt
23     """
24
25     src = source.Source()
26     chl = channel.Channel()
27     linear_code = channel.Linear_Code()
28     dest = destination.Destination()
29
30
31     src.read_txt("Resource/hardcoded.txt")
32     tx_msg = src.get_digital_data()
33
34     tx_codeword = linear_code.encoder_systematic(tx_msg)
35
36     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
37
38     # without error correction
39     rx_msg = linear_code.decoder_systematic(rx_codeword)
40     dest.set_digital_data(rx_msg)
41     dest.write_txt("Result/linear-bsc-output.txt")
42
43     # with error correction
44     estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
45     rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
46     dest.set_digital_data(rx_msg)
47     dest.write_txt("Result/linear-bsc-output-syndrome-corrected.txt")
48
49
50 def linear_png():
51     """
52         Source - Channel encoder - Channel - Channel decoder - Destination
53
54         png      - (7,4) linear      - bsc      - (7,4) linear      - png
55     """

```

```

56     src = source.Source()
57     chl = channel.Channel()
58     linear_code = channel.Linear_Code()
59     dest = destination.Destination()
60
61
62     height , width , channels = src.read_png("Resource/image.png")
63     tx_msg = src.get_digital_data()
64
65     tx_codeword = linear_code.encoder_systematic(tx_msg)
66
67     rx_codeword = chl.binary_symmetric_channel(tx_codeword , 0.01)
68
69     # without error correction
70     rx_msg = linear_code.decoder_systematic(rx_codeword)
71     dest.set_digital_data(rx_msg)
72     dest.write_png_from_digital("Result/linear-bsc-output.png" , height , width ,
73                               channels)
74
75     # with error correction
76     estimated_tx_codeword = linear_code.corrector_syndrome(rx_codeword)
77     rx_msg = linear_code.decoder_systematic(estimated_tx_codeword)
78     dest.set_digital_data(rx_msg)
79     dest.write_png_from_digital("Result/linear-bsc-output-syndrome-corrected .
80                               png" , height , width , channels)
81
82 def linear_wav():
83     """
84     Source - Channel encoder - Channel - Channel decoder - Destination
85
86     wav      - (7,4) linear      - bsc      - (7,4) linear      - wav
87     """
88
89     src = source.Source()
90     chl = channel.Channel()
91     linear_code = channel.Linear_Code()
92     dest = destination.Destination()
93
94
95     shape , sample_rate = src.read_wav("Resource/file-example_WAV_1MG.wav")
96     plot_wav.plot_wav_time_domain(src.get_analogue_data() , sample_rate , "
97                                     Result/wav-time-domain-TX.png")
98     plot_wav.plot_wav_frequency_domain(src.get_analogue_data() , sample_rate , "
99                                     Result/wav-frequency-domain-TX.png")
100
101    # tx_msg = src.get_analogue_data()
102    # rx_msg = tx_msg
103    # dest.set_analogue_data(rx_msg)
104    # plot_wav.plot_wav_time_domain(dest.get_analogue_data() , sample_rate , "
105                                    Result/wav-time-domain-RX.png")
106    # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data() , sample_rate ,
107                                         , "Result/wav-frequency-domain-RX.png")
108    # dest.write_wav_from_analogue(sample_rate , "Result/hamming-bsc-output.wav"
109                                   )

```

```

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

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

```

6.6 Cyclic code

```

1 # Copyright (c) 2023 Chenye Yang
2
3 import logging
4
5 from Source import source
6 from Channel import channel
7 from Destination import destination
8 from Utils import plot_wav, polyTools
9
10
11 # Configure the logging

```

```

12     logging.basicConfig(level=logging.INFO, format='%(asctime)s--%(name)s--%(levelname)s--%(message)s')
13
14 # Create a logger in the main module
15 logger = logging.getLogger(__name__)
16
17
18
19 # Work with (3, 1) cyclic code
20 n = 3
21 k = 1
22
23 # Create the generator matrix
24 genMatrix_decimal = polyTools.findMatrix(n, k)
25 G = polyTools.genMatrixDecmial2Ndarray(genMatrix_decimal, n)
26
27
28 def cyclic_txt():
29     """
30         Source - Channel encoder - Channel - Channel decoder - Destination
31
32         txt      - (n, k) cyclic      - bsc      - (n, k) cyclic      - txt
33     """
34     src = source.Source()
35     chl = channel.Channel()
36     cyclic_code = channel.Cyclic_Code(G)
37     dest = destination.Destination()
38
39
40     src.read_txt("Resource/hardcoded.txt")
41     tx_msg = src.get_digital_data()
42
43     tx_codeword = cyclic_code.encoder_systematic(tx_msg)
44
45     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
46
47     # without error correction
48     rx_msg = cyclic_code.decoder_systematic(rx_codeword)
49     dest.set_digital_data(rx_msg)
50     dest.write_txt("Result/cyclic-bsc-output.txt")
51
52     # with error correction
53     estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
54     rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword)
55     dest.set_digital_data(rx_msg)
56     dest.write_txt("Result/cyclic-bsc-output-syndrome-corrected.txt")
57
58
59 def cyclic_png():
60     """
61         Source - Channel encoder - Channel - Channel decoder - Destination
62
63         png      - (n, k) cyclic      - bsc      - (n, k) cyclic      - png
64     """

```

```

65     src = source.Source()
66     chl = channel.Channel()
67     cyclic_code = channel.Cyclic_Code(G)
68     dest = destination.Destination()
69
70
71     height, width, channels = src.read_png("Resource/image.png")
72     tx_msg = src.get_digital_data()
73
74     tx_codeword = cyclic_code.encoder_systematic(tx_msg)
75
76     rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
77
78     # without error correction
79     rx_msg = cyclic_code.decoder_systematic(rx_codeword)
80     dest.set_digital_data(rx_msg)
81     dest.write_png_from_digital("Result/cyclic-bsc-output.png", height, width,
82                                 channels)
83
84     # with error correction
85     estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
86     rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword)
87     dest.set_digital_data(rx_msg)
88     dest.write_png_from_digital("Result/cyclic-bsc-output-syndrome-corrected.
89                                 png", height, width, channels)
90
91
92 def cyclic_wav():
93     """
94     Source - Channel encoder - Channel - Channel decoder - Destination
95
96     wav      - (n, k) cyclic      - bsc      - (n, k) cyclic      - wav
97     """
98
99     src = source.Source()
100    chl = channel.Channel()
101    cyclic_code = channel.Cyclic_Code(G)
102    dest = destination.Destination()
103
104
105    shape, sample_rate = src.read_wav("Resource/file-example_WAV_1MG.wav")
106    plot_wav.plot_wav_time_domain(src.get_analogue_data(), sample_rate, "
107                                  Result/wav-time-domain-TX.png")
108    plot_wav.plot_wav_frequency_domain(src.get_analogue_data(), sample_rate, "
109                                  Result/wav-frequency-domain-TX.png")
110
111    # tx_msg = src.get_analogue_data()
112    # rx_msg = tx_msg
113    # dest.set_analogue_data(rx_msg)
114    # plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
115                                  Result/wav-time-domain-RX.png")
116    # plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate
117                                      , "Result/wav-frequency-domain-RX.png")
118    # dest.write_wav_from_analogue(sample_rate, "Result/hamming-bsc-output.wav"
119                                )

```

```

112
113 tx_msg = src.get_digital_data()
114
115 tx_codeword = cyclic_code.encoder_systematic(tx_msg)
116
117 rx_codeword = chl.binary_symmetric_channel(tx_codeword, 0.01)
118
119 # without error correction
120 rx_msg = cyclic_code.decoder_systematic(rx_codeword)
121
122 dest.set_digital_data(rx_msg)
123 dest.write_wav_from_digital(shape, sample_rate, "Result/cyclic-bsc-output."
124     wav")
125
126 plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
127     Result/cyclic-bsc-wav-time-domain-RX.png")
128 plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
129     "Result/cyclic-bsc-wav-frequency-domain-RX.png")
130
131 # with error correction
132 estimated_tx_codeword = cyclic_code.corrector_syndrome(rx_codeword)
133 rx_msg = cyclic_code.decoder_systematic(estimated_tx_codeword)
134
135 dest.set_digital_data(rx_msg)
136 dest.write_wav_from_digital(shape, sample_rate, "Result/cyclic-bsc-output-
137     syndrome-corrected.wav")
138
139 plot_wav.plot_wav_time_domain(dest.get_analogue_data(), sample_rate, "
140     Result/cyclic-bsc-wav-time-domain-RX-syndrome-corrected.png")
141 plot_wav.plot_wav_frequency_domain(dest.get_analogue_data(), sample_rate,
142     "Result/cyclic-bsc-wav-frequency-domain-RX-syndrome-corrected.png")
143
144 if __name__ == '__main__':
145     cyclic_txt()
146     cyclic_png()
147     cyclic_wav()

```