# Digital Communications Laboratory Report

Forward Error

Correction

by JIE TIAN 2802461t

Rankine 329

March 8, 2024

# I. Introduction

This laboratory project introduces some issues that occur in digital communication channels. In particular, we will study Forward Error Correction (FEC) as a technique to overcome the detrimental effects of noise on the communication channel. Here we will circumvent some of the low-level coding by using the komm Python library. The Komm library provides functions for PSK modulation and demodulation and for simulating an additive white Gaussian noise source, we will use the following code to implement QPSK coding, propagate the signal in a noisy channel, and QPSK decoding. The modulated psk waveform is by default unit average power. The scalar snr specifies the signal-to-noise ratio for the channel.

```
# Quadra - ture Phase Shift Keying (QPSK)

psk = komm.PSKModulation(4, phase_offset=np.pi / 4)

awgn = komm.AWGNChannel(snr=10 ** ((snrdb / 10) ), signal_power=1.0)



# modulate(QPSK)

tx_QPSK_BCH_stream = psk.modulate(tx_BCH_stream)

# simulate Noisy Channel

rx_QPSK_BCH_stream = awgn(tx_QPSK_BCH_stream)

# Demodulate(QPSK)

rx_BCH_stream = psk.demodulate(rx_QPSK_BCH_stream)
```

# II. Importing libraries and Obtaining Digital Data

Before we start parity checking, we need to import the libraries that we will need to use in this project, they can provide functions such as reading pictures by greyscale, some specific mathematical algorithms, plotting, modulation and demodulation, simulation of signal channels, and so on, here is the corresponding code.

```
import numpy as np
```

*from PIL import Image*

*from matplotlib import pyplot as plt*

*import scipy*

*import komm*

A number of 8-bit depth grayscale images of various sizes have been provided for use in this laboratory project. We can read the greyscale image and convert it to the corresponding binary file with the following code.


*tx_im = Image.open("/Users/george/Project/DC-Experiment-/Forward Error Correction/DC4_150x100.pgm")*

*Npixels = tx_im.size[1]*tx_im.size[0]*

*plt.figure()*

*plt.imshow(np.array(tx_im),cmap="gray",vmin=0,vmax=255)*

*plt.show()*

*tx_bin = np.unpackbits(np.array(tx_im))*

# III. Block Coding: BCH codes

Block codes work on fixed-size blocks (packets) of bits or symbols of predetermined size. Practical block codes can generally be hard-decoded in polynomial time to their block length. Bose-Chaudhuri-Hocquenghem (BCH) codes form a class of cyclic error-correcting codes that are constructed using polynomials over a finite field (also called Galois field). One of the key features of BCH codes is that during code design, there is precise control over the number of symbol errors correctable by the code. In particular, it is possible to design binary BCH codes that can correct multiple-bit errors. Another advantage of BCH codes is the ease with which they can be decoded, namely, via an algebraic method known as syndrome decoding. This simplifies the design of the decoder for these codes, using small low-power electronic hardware. We can use the following code to initialize the BCH encoder and decoder.

*code = komm.BCHCode(mu=3, delta=3)*

*n,k = code.length, code.dimension*

*encoder = komm.BlockEncoder(code)*

*decoder = komm.BlockDecoder(code)*

*print(code, n, k)*

## 1. (7,4) code alphabet

The ($n$ = 7, $k$ = 4) BCH code is equivalent to the Hamming (7,4) code. We can generate the alphabet of valid 7-bit codewords by examining code.codewords and obtain the corresponding generator polynomial from code.generator_polynomial. The following code examines the process of encoding and decoding a 4-bit number via the BCH method and taking any two codewords in the alphabet, their bitwise XOR results are still in the alphabet.

*print(code.codewords, code.generator_polynomial)*

*print(code.codewords[1] ^ code.codewords[5])*

*test = np.array([1, 1, 0, 1])*

*encoder_test = encoder(test)*

*print(decoder(encoder_test))*

*# the results of the code*

*BCHCode(mu=3, delta=3) 7 4*

*[1 1 0 1]*

## 2. Forward error correction using BCH codes

In this part, use quadrature phase shift keying (QPSK) with unit average power per symbol. Having set up the code as shown above, coding and decoding are done with:

*coded_word = encoder(message_word)*

*message_word = decoder(coded_word)*

where the message_word is a fixed length of k bits and coded_word is a fixed length of n bits. We can use the following loop to code (and decode) the entire binary data.

```
for snrdb in range(-30, 91, 2):
    print(snrdb)
    # Quadra - ture Phase Shift Keying (QPSK)
    psk = komm.PSKModulation(4, phase_offset=np.pi / 4)
    awgn = komm.AWGNChannel(snr=10 ** ((snrdb / 10) / 10), signal_power=1.0)


    rx_bin = np.empty(0)
    rx_bin_withoutBCH = np.empty(0)


    for num in range(0, Npixels * 8, 8):
```

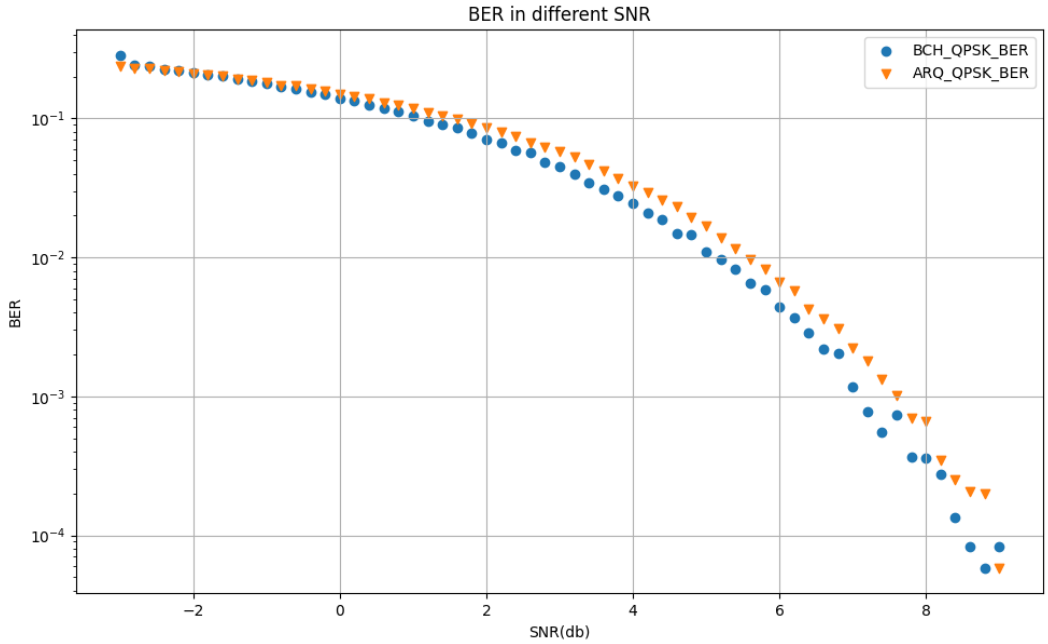
```
        # Using BCH and QPSK methods to encode the information
        tx_BCH_stream = encoder(tx_bin[num:num + 4])
        tx_BCH_stream = np.append(tx_BCH_stream, encoder(tx_bin[num + 4:num
+ 8]))
        # add one bit at the end of the tx_BCH_stream to
        # ensure tx_BCH_stream can divide by psk.bits_per_symbol
        # tx_8bit_BCH_stream = np.append(tx_BCH_stream, 0)
        # tx_QPSK_BCH_stream = psk.modulate(tx_8bit_BCH_stream)
        # print(tx_BCH_stream)
        tx_QPSK_BCH_stream = psk.modulate(tx_BCH_stream)

        # simulate Noisy Channel
        rx_QPSK_BCH_stream = awgn(tx_QPSK_BCH_stream)

        # Demodulate(QPSK)
        rx_BCH_stream = psk.demodulate(rx_QPSK_BCH_stream)
        rx_bin_bit_stream = decoder(rx_BCH_stream[:7])
        rx_bin_bit_stream = np.append(rx_bin_bit_stream,
decoder(rx_BCH_stream[7:]))

        rx_bin = np.append(rx_bin, rx_bin_bit_stream)
```

Plot *ber* (logarithmic axis) vs *snr* (in dB) over a suitable range of signal-to-noise, and also compare the results with QPSK without error correction.




# IV. Convolutional Coding

Convolutional codes work on bit or symbol streams of arbitrary length. They are most often soft-decoded with the Viterbi algorithm. The komm library provides functions for encoding and decoding convolutional codes. There are several options. In particular, the best convolutional code performance is normally with soft-decision decoding on the real values returned from the demodulator based on the Euclidean distance from the symbol, thus providing a confidence level to the Viterbi decoder.

## 1. Convolutional Coding with "hard" and "soft" decoding

Use the following code to initialize the encoder decoder_hard and decoder_soft. tblen is a positive integer scalar that specifies the traceback depth in the Viterbi algorithm. When used in stream (continuous) form, the decoder has a delay (latency) equal to tblen for a single input stream code. Therefore we should append tblen zeros to the input binary stream, and discard the first tblen bits of the output stream. Typical values for a traceback depth tblen are about five or six times the constraint length *L*.

*code = komm.ConvolutionalCode(feedforward_polynomials=[[0o7, 0o5]])*

*tblen = 18*

*encoder = komm.ConvolutionalStreamEncoder(code)*

*decoder_hard = komm.ConvolutionalStreamDecoder(code,*

*traceback_length=tblen, input_type="hard")*

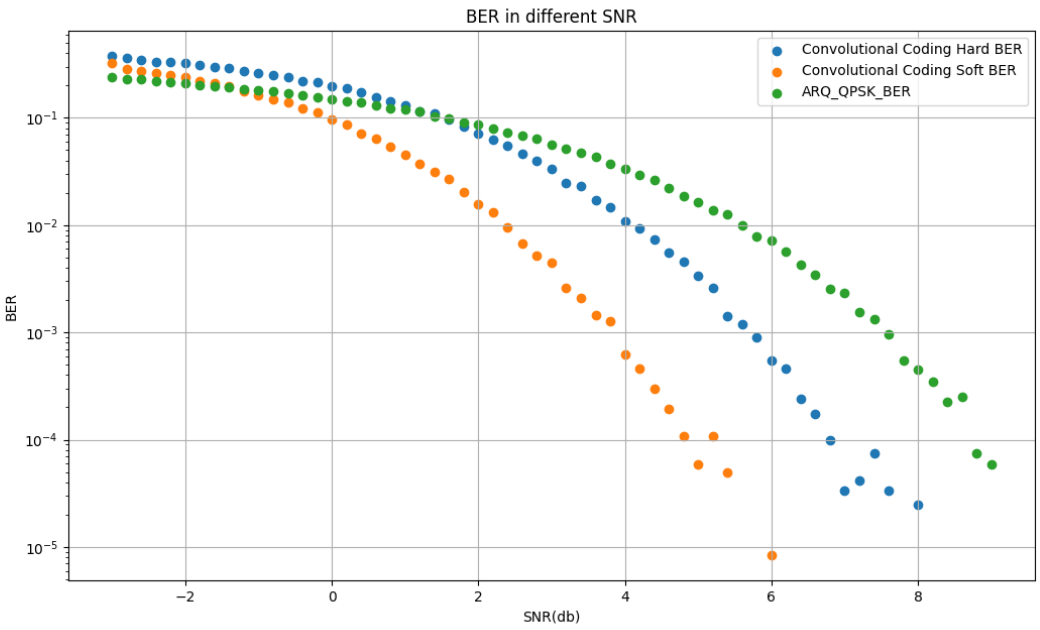
*decoder_soft = komm.ConvolutionalStreamDecoder(code,*

*traceback_length=tblen, input_type="soft")*

*tx_bin = np.append(tx_bin, np.zeros(tblen))*

*rx_hard_bin = rx_hard_bin[tblen:]*

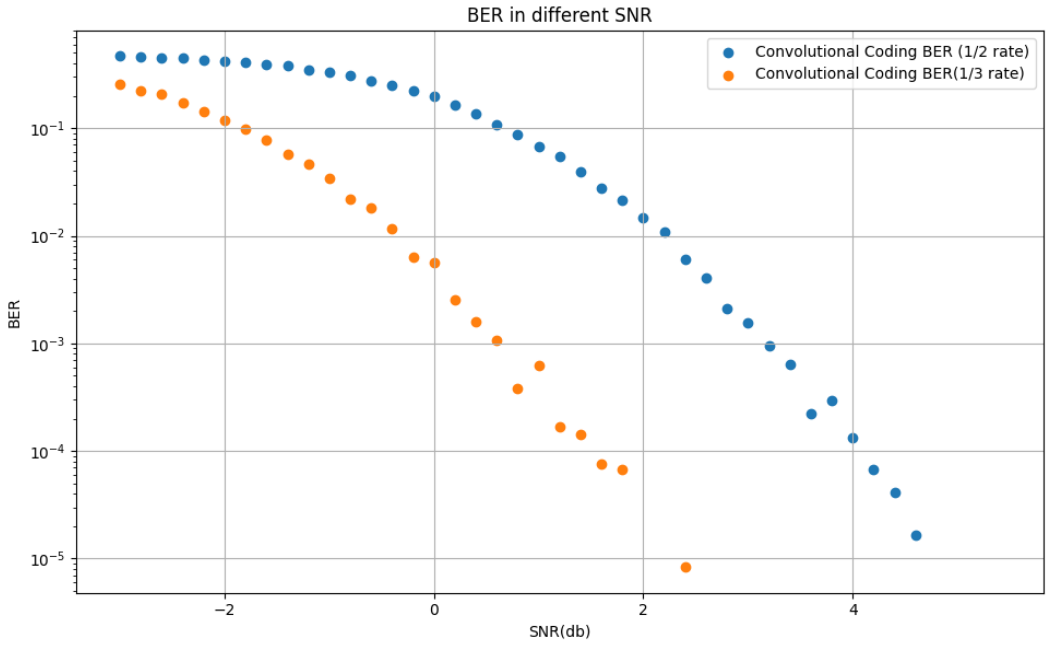
*rx_soft_bin = rx_soft_bin[tblen:]*




## 2. 7 constraint length of convolutional codes

We can use the parameter *feedforward_polynomials = [[0o155, 0o117]]* and *feedforward_polynomials = [[0o155, 0o117, 0o127]]* to initialize the new encoder and the decoder. Again, plot *ber* (logarithmic axis) vs *snr* (in dB) over a suitable

range of signal-to-noise for each of these using "soft" decoding.




# V. Comparing Codes

Compare the various error-correction techniques (simple parity check and ARQs, BCH block codes, Convolutional Code with hard/soft decision), simple parity check and ARQs provide the best error-correcting for the data provided in cases of values for *snr* =3 dB using QPSK. Here are the results.

*# BCH_QPSK_BER in snr = 3dB*

*0.10224166666666666*

*# ARQ_QPSK_BER in snr = 3dB*

*0.057508333333333335*

*Convolutional Coding with Hard BER in snr = 3dB*

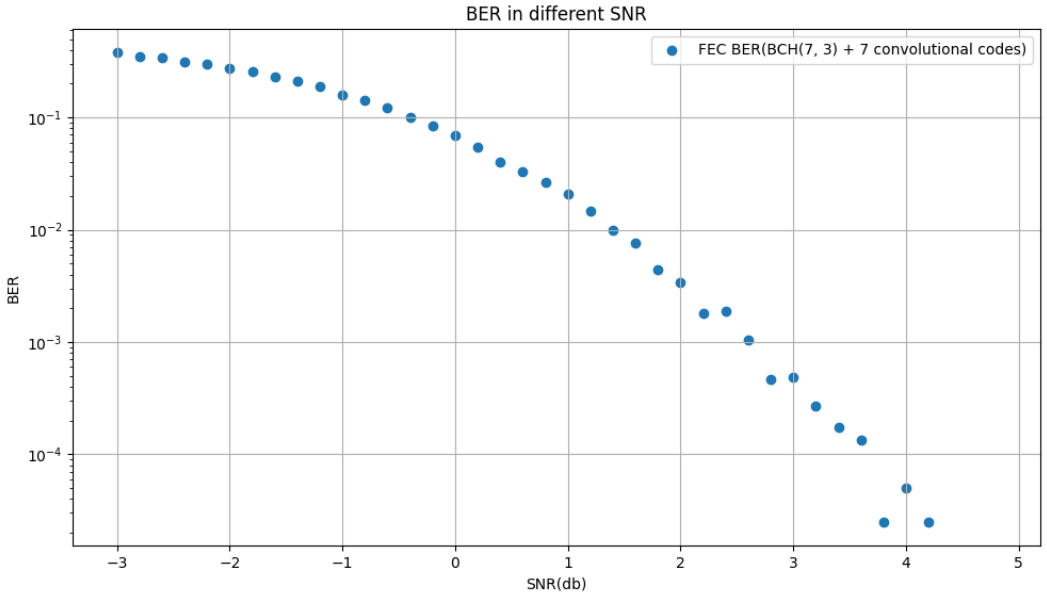
*0.09111666666666667*

*Convolutional Coding with Soft BER in snr = 3dB*

*0.067075*

# VI. Concatenated Codes

Many implementations of FEC can tolerate even lower signal-to-noise ratio by concatenating two different FEC code methods. Now use a convolutional code as an inner code, modulating as QPSK and passing the result through an AWGN channel, demodulate and decode using the soft-decision Viterbi method. Use this as an effective transmission channel and apply a BCH code as the outer code. Here is the ber in different SNR values. We can get ber equals 0.067425 where the signal power is equal to the noise, i.e. snr = 0 dB.




# VII. Conclusion

As the SNR increases, the error rates of the different codes gradually decrease, but Convolutional Code with soft decision is the best.

```
 1 import numpy as np
 2 from PIL import Image
 3 from matplotlib import pyplot as plt
 4 import scipy
 5 import komm
 6
 7 # Obtaining Digital Data from file
 8 tx_im = Image.open("/Users/george/
   Project/DC-Experiment-/Forward Error
    Correction/DC4_150x100.pgm")
 9 Npixels = tx_im.size[1] * tx_im.size
   [0]
10 plt.figure()
11 plt.imshow(np.array(tx_im), cmap="
   gray", vmin=0, vmax=255)
12 plt.show()
13 tx_bin = np.unpackbits(np.array(
   tx_im))
14 # print(tx_bin, tx_bin.dtype)
15
16 code = komm.BCHCode(mu=3, delta=3)
17 n,k = code.length, code.dimension
18 encoder = komm.BlockEncoder(code)
19 decoder = komm.BlockDecoder(code)
20
21 print(code, n, k)
22 print(code.codewords, code.
   generator_polynomial)
23 print(code.codewords[1] ^ code.
```

```
23 codewords[5])
24 test = np.array([1, 1, 0, 1])
25 encoder_test = encoder(test)
26 print(decoder(encoder_test))
27
28 x = np.empty(0)
29 y_BCH_QPSK_ber = np.empty(0)
30 y_QPSK_ber = np.empty(0)
31
32 for snrdb in range(30, 91, 2):
33     print(snrdb)
34     # Quadra - ture Phase Shift
   Keying (QPSK)
35     psk = komm.PSKModulation(4,
   phase_offset=np.pi / 4)
36     awgn = komm.AWGNChannel(snr=10
    ** ((snrdb / 10) / 10),
   signal_power=1.0)
37
38     rx_bin = np.empty(0)
39     rx_bin_withoutBCH = np.empty(0)
40
41     for num in range(0, Npixels * 8
   , 8):
42         # Using BCH and QPSK methods
    to encode the information
43         tx_BCH_stream = encoder(
   tx_bin[num:num + 4])
44         tx_BCH_stream = np.append(
```

```
44 tx_BCH_stream, encoder(tx_bin[num +
   4:num + 8]))
45             # add one bit at the end of
   the tx_BCH_stream to
46             # ensure tx_BCH_stream can
   divide by psk.bits_per_symbol
47             # tx_8bit_BCH_stream = np.
   append(tx_BCH_stream, 0)
48             # tx_QPSK_BCH_stream = psk.
   modulate(tx_8bit_BCH_stream)
49             # print(tx_BCH_stream)
50             tx_QPSK_BCH_stream = psk.
   modulate(tx_BCH_stream)
51
52             # simulate Noisy Channel
53             rx_QPSK_BCH_stream = awgn(
   tx_QPSK_BCH_stream)
54
55             # Demodulate(QPSK)
56             rx_BCH_stream = psk.
   demodulate(rx_QPSK_BCH_stream)
57             rx_bin_bit_stream = decoder(
   rx_BCH_stream[:7])
58             rx_bin_bit_stream = np.
   append(rx_bin_bit_stream, decoder(
   rx_BCH_stream[7:]))
59
60             rx_bin = np.append(rx_bin,
   rx_bin_bit_stream)
```

```
61
62          # without BCH
63          tx_bit_stream = np.array(
    tx_bin[num:num + 8])
64          tx_bit_stream[7] = (np.sum(
    tx_bit_stream) - tx_bit_stream[7
    ]) % 2
65          tx_bin[num:num + 8] =
    tx_bit_stream
66
67          tx_data_bit_stream = psk.
    modulate(tx_bit_stream)
68
69          # simulate Noisy Channel
70          rx_data_bit_stream = awgn(
    tx_data_bit_stream)
71
72          rx_bit_stream = psk.
    demodulate(rx_data_bit_stream)
73
74          # parity test
75          if (np.sum(rx_bit_stream) %
    2):
76              # Automatic Repeat-
    reQuest (ARQ)
77              rx_data_bit_stream =
    awgn(tx_data_bit_stream)
78              rx_bit_stream = psk.
    demodulate(rx_data_bit_stream)
```

```
79
80          rx_bin_withoutBCH = np.
   append(rx_bin_withoutBCH,
   rx_bit_stream)
81
82
83      BCH_QPSK_ber = np.sum(tx_bin
    != rx_bin) / (Npixels * 8)
84      QPSK_ber = np.sum(tx_bin !=
   rx_bin_withoutBCH) / (Npixels * 8)
85      print(BCH_QPSK_ber, QPSK_ber)
86
87      y_BCH_QPSK_ber = np.append(
   y_BCH_QPSK_ber, BCH_QPSK_ber)
88      y_QPSK_ber = np.append(
   y_QPSK_ber, QPSK_ber)
89
90      x = np.append(x, (snrdb / 10))
91
92 plt.figure()
93
94 # plot ber
95 plt.scatter(x, y_BCH_QPSK_ber,
   label="BCH_QPSK_BER")
96
97 plt.scatter(x, y_QPSK_ber, marker="
   v", label="ARQ_QPSK_BER")
98
99 plt.title("BER in different SNR")
```

```
100 plt.xlabel("SNR(db)")
101 plt.ylabel("BER")
102
103 plt.yscale("log")
104 plt.grid(True)
105 plt.legend(loc="upper right")
106 plt.show()
107
108
```

```
 1 import numpy as np
 2 from PIL import Image
 3 from matplotlib import pyplot as plt
 4 import scipy
 5 import komm
 6
 7 # Obtaining Digital Data from file
 8 tx_im = Image.open("/Users/george/
   Project/DC-Experiment-/Forward Error
    Correction/DC4_150x100.pgm")
 9 Npixels = tx_im.size[1] * tx_im.size
   [0]
10 plt.figure()
11 plt.imshow(np.array(tx_im), cmap="
   gray", vmin=0, vmax=255)
12 plt.show()
13 tx_bin = np.unpackbits(np.array(
   tx_im))
14 # print(tx_bin, tx_bin.dtype)
15
16 code = komm.ConvolutionalCode(
   feedforward_polynomials=[[0o7, 0o5
   ]])
17 tblen = 18
18 encoder = komm.
   ConvolutionalStreamEncoder(code)
19 decoder_hard = komm.
   ConvolutionalStreamDecoder(code,
20
```

```
20              traceback_length=tblen,
    input_type="hard")
21 decoder_soft = komm.
    ConvolutionalStreamDecoder(code,
22
                traceback_length=tblen,
    input_type="soft")
23
24 # Test convolutional coding
25 # print(code)
26 # test = np.array([1, 1, 0, 1])
27 # test = np.append(test, np.zeros(18
    ))
28 # encode_test = encoder(test)
29 # print(encode_test)
30 # print(decoder(encode_test)[18:])
31
32 x = np.empty(0)
33 y_Conv_Hard = np.empty(0)
34 y_Conv_Soft = np.empty(0)
35 y_Conv_Arq = np.empty(0)
36
37 for snrdb in range(30, 91, 2):
38     print(snrdb)
39     # Quadra - ture Phase Shift
    Keying (QPSK)
40     psk = komm.PSKModulation(4,
    phase_offset=np.pi / 4)
41     awgn = komm.AWGNChannel(snr=10
```

```
41  ** ((snrdb / 10) / 10),
   signal_power=1.0)
42
43      rx_bin_hard = np.empty(0)
44      rx_bin_soft = np.empty(0)
45      rx_bin_ARQ = np.empty(0)
46
47      # append tblen zeros to the
   input binary stream
48      tx_bin = np.append(tx_bin, np.
   zeros(tblen))
49      tx_Conv_coding = encoder(tx_bin)
50      tx_QPSK_Conv_Coding = psk.
   modulate(tx_Conv_coding)
51
52      # simulate Noisy Channel
53      rx_QPSK_Conv_Coding = awgn(
   tx_QPSK_Conv_Coding)
54
55      # "hard" demodulate and decoding
56      rx_Conv_Coding_hard = psk.
   demodulate(rx_QPSK_Conv_Coding,
   decision_method="hard")
57      rx_hard_bin = decoder_hard(
   rx_Conv_Coding_hard)
58
59      # "soft" demodulate and decoding
60      rx_Conv_Coding_soft = psk.
   demodulate(rx_QPSK_Conv_Coding,
```

```
60 decision_method="soft")
61     rx_soft_bin = decoder_soft(
   rx_Conv_Coding_soft)
62
63     # ARQ Coding
64     for num in range(0, Npixels * 8
   , 8):
65         # ARQ
66         tx_bit_stream = np.array(
   tx_bin[num:num + 8])
67         tx_bit_stream[7] = (np.sum(
   tx_bit_stream) - tx_bit_stream[7
   ]) % 2
68         tx_bin[num:num + 8] =
   tx_bit_stream
69
70         tx_data_bit_stream = psk.
   modulate(tx_bit_stream)
71
72         # simulate Noisy Channel
73         rx_data_bit_stream = awgn(
   tx_data_bit_stream)
74
75         rx_bit_stream = psk.
   demodulate(rx_data_bit_stream)
76
77         # parity test
78         if (np.sum(rx_bit_stream) %
   2):
```

```
79                # Automatic Repeat-reQuest (ARQ)
80                rx_data_bit_stream = awgn(tx_data_bit_stream)
81                rx_bit_stream = psk.demodulate(rx_data_bit_stream)
82
83            rx_bin_ARQ = np.append(rx_bin_ARQ, rx_bit_stream)
84
85        tx_bin = tx_bin[:Npixels * 8]
86        Conv_Coding_Hard = np.sum(tx_bin != rx_hard_bin[tblen:]) / (Npixels * 8)
87        Conv_Coding_Soft = np.sum(tx_bin != rx_soft_bin[tblen:]) / (Npixels * 8)
88        Conv_Coding_Arq = np.sum(tx_bin != rx_bin_ARQ) / (Npixels * 8)
89        print(Conv_Coding_Hard, Conv_Coding_Soft, Conv_Coding_Arq)
90
91        y_Conv_Hard = np.append(y_Conv_Hard, Conv_Coding_Hard)
92        y_Conv_Soft = np.append(y_Conv_Soft, Conv_Coding_Soft)
93        y_Conv_Arq = np.append(y_Conv_Arq, Conv_Coding_Arq)
94
```

```
 95         x = np.append(x, (snrdb / 10))
 96
 97 plt.figure()
 98
 99 # plot ber
100 plt.scatter(x, y_Conv_Hard, label="
    Convolutional Coding Hard BER")
101 plt.scatter(x, y_Conv_Soft, label="
    Convolutional Coding Soft BER")
102 plt.scatter(x, y_Conv_Arq, label="
    ARQ_QPSK_BER")
103
104 plt.title("BER in different SNR")
105 plt.xlabel("SNR(db)")
106 plt.ylabel("BER")
107
108 plt.yscale("log")
109 plt.grid(True)
110 plt.legend(loc="upper right")
111 plt.show()
112
113
114
115
```

```
 1 import numpy as np
 2 from PIL import Image
 3 from matplotlib import pyplot as plt
 4 import scipy
 5 import komm
 6
 7 # Obtaining Digital Data from file
 8 tx_im = Image.open("/Users/george/
   Project/DC-Experiment-/Forward Error
    Correction/DC4_150x100.pgm")
 9 Npixels = tx_im.size[1] * tx_im.size
   [0]
10 plt.figure()
11 plt.imshow(np.array(tx_im), cmap="
   gray", vmin=0, vmax=255)
12 plt.show()
13 tx_bin = np.unpackbits(np.array(
   tx_im))
14 # print(tx_bin, tx_bin.dtype)
15
16 code_one_twice = komm.
   ConvolutionalCode(
   feedforward_polynomials=[[0o155,
   0o117]])
17 tblen = 7 * 6
18 encoder_one_twice = komm.
   ConvolutionalStreamEncoder(
   code_one_twice)
19
```

```
20 decoder_one_twice = komm.
   ConvolutionalStreamDecoder(
   code_one_twice,
21
                traceback_length=tblen,
   input_type="soft")
22
23 code_one_third = komm.
   ConvolutionalCode(
   feedforward_polynomials=[[0o155,
   0o117, 0o127]])
24 encoder_one_third = komm.
   ConvolutionalStreamEncoder(
   code_one_third)
25
26 decoder_one_third = komm.
   ConvolutionalStreamDecoder(
   code_one_third,
27
                traceback_length=tblen,
   input_type="soft")
28
29 x = np.empty(0)
30 y_Conv_One_Twice = np.empty(0)
31 y_Conv_One_Third = np.empty(0)
32
33 for snrdb in range(-30, 55, 2):
34     print(snrdb)
35     # Quadra - ture Phase Shift
```

```
35 Keying (QPSK)
36     psk = komm.PSKModulation(4,
   phase_offset=np.pi / 4)
37     awgn = komm.AWGNChannel(snr=10
    ** ((snrdb / 10) / 10),
   signal_power=1.0)
38
39     rx_bin_one_twice = np.empty(0)
40     rx_bin_one_third = np.empty(0)
41
42     # append tblen zeros to the
   input binary stream
43     tx_bin = np.append(tx_bin, np.
   zeros(tblen))
44
45     # The rate 1/2 code
46     tx_Conv_coding_one_twice =
   encoder_one_twice(tx_bin)
47     tx_QPSK_Conv_Coding_one_twice =
   psk.modulate(
   tx_Conv_coding_one_twice)
48
49     # simulate Noisy Channel
50     rx_QPSK_Conv_Coding_one_twice =
   awgn(tx_QPSK_Conv_Coding_one_twice)
51
52     # The rate 1/2 code "soft"
   demodulate and decoding
53     rx_Conv_Coding_soft_one_twice =
```

```
53 psk.demodulate(
   rx_QPSK_Conv_Coding_one_twice,
   decision_method="soft")
54     rx_bin_one_twice =
   decoder_one_twice(
   rx_Conv_Coding_soft_one_twice)
55
56     # The rate 1/3 code
57     tx_Conv_coding_one_third =
   encoder_one_third(tx_bin)
58     tx_QPSK_Conv_Coding_one_third =
   psk.modulate(
   tx_Conv_coding_one_third)
59
60     # simulate Noisy Channel
61     rx_QPSK_Conv_Coding_one_third =
   awgn(tx_QPSK_Conv_Coding_one_third)
62
63     # The rate 1/3 code "soft"
   demodulate and decoding
64     rx_Conv_Coding_soft_one_third =
   psk.demodulate(
   rx_QPSK_Conv_Coding_one_third,
   decision_method="soft")
65     rx_bin_one_third =
   decoder_one_third(
   rx_Conv_Coding_soft_one_third)
66
67     tx_bin = tx_bin[:Npixels * 8]
```

```
68
69      Conv_Coding_One_Twice = np.sum(
    tx_bin != rx_bin_one_twice[tblen
    :]) / (Npixels * 8)
70      Conv_Coding_One_third = np.sum(
    tx_bin != rx_bin_one_third[tblen
    :]) / (Npixels * 8)
71
72      print(Conv_Coding_One_Twice,
    Conv_Coding_One_third)
73
74      y_Conv_One_Twice = np.append(
    y_Conv_One_Twice,
    Conv_Coding_One_Twice)
75      y_Conv_One_Third = np.append(
    y_Conv_One_Third,
    Conv_Coding_One_third)
76
77      x = np.append(x, (snrdb / 10))
78
79  plt.figure()
80
81  # plot ber
82  plt.scatter(x, y_Conv_One_Twice,
    label="Convolutional Coding BER (1/2
     rate)")
83  plt.scatter(x, y_Conv_One_Third,
    label="Convolutional Coding BER(1/3
    rate)")
```

```
84
85 plt.title("BER in different SNR")
86 plt.xlabel("SNR(db)")
87 plt.ylabel("BER")
88
89 plt.yscale("log")
90 plt.grid(True)
91 plt.legend(loc="upper right")
92 plt.show()
93
94
95
96
```

```
 1 import numpy as np
 2 from PIL import Image
 3 from matplotlib import pyplot as plt
 4 import scipy
 5 import komm
 6
 7 # Obtaining Digital Data from file
 8 tx_im = Image.open("/Users/george/
   Project/DC-Experiment-/Forward Error
    Correction/DC4_150x100.pgm")
 9 Npixels = tx_im.size[1] * tx_im.size
   [0]
10 plt.figure()
11 plt.imshow(np.array(tx_im), cmap="
   gray", vmin=0, vmax=255)
12 plt.show()
13 tx_bin = np.unpackbits(np.array(
   tx_im))
14 # print(tx_bin, tx_bin.dtype)
15
16 # Concolutional Encode and Decode
17 code_one_third = komm.
   ConvolutionalCode(
   feedforward_polynomials=[[0o155,
   0o117, 0o127]])
18 encoder_one_third = komm.
   ConvolutionalStreamEncoder(
   code_one_third)
19 tblen = 7 * 6
```

```
20 decoder_one_third = komm.
   ConvolutionalStreamDecoder(
   code_one_third, traceback_length=
   tblen)
21
22 # BCH encode and decode
23 code = komm.BCHCode(mu=3, delta=3)
24 encoder = komm.BlockEncoder(code)
25 decoder = komm.BlockDecoder(code)
26
27 x = np.empty(0)
28 y_Conv_One_Third = np.empty(0)
29
30 for snrdb in range(0, 50, 2):
31     print(snrdb)
32     # Quadra - ture Phase Shift
   Keying (QPSK)
33     psk = komm.PSKModulation(4,
   phase_offset=np.pi / 4)
34     awgn = komm.AWGNChannel(snr=10
    ** ((snrdb / 10) / 10),
   signal_power=1.0)
35
36     rx_bin_one_third = np.empty(0)
37
38     # append tblen zeros to the
   input binary stream
39     tx_bin = np.append(tx_bin, np.
   zeros(tblen))
```

```
40
41      # The rate 1/3 code
42      tx_Conv_coding_one_third =
    encoder_one_third(tx_bin)
43      print(tx_Conv_coding_one_third,
    tx_Conv_coding_one_third.size)
44
45      tx_Conv_coding_one_third = np.
    append(tx_Conv_coding_one_third, np.
    zeros(2))
46      print(tx_Conv_coding_one_third.
    size)
47
48      rx_bin_Conv = np.empty(0)
49
50      for num in range(0, 360128, 8):
51          # Using BCH and QPSK methods
     to encode the information
52          tx_BCH_stream = encoder(
    tx_Conv_coding_one_third[num:num + 4
    ])
53          tx_BCH_stream = np.append(
    tx_BCH_stream, encoder(
    tx_Conv_coding_one_third[num + 4:num
     + 8]))
54          tx_QPSK_BCH_stream = psk.
    modulate(tx_BCH_stream)
55
56          # simulate Noisy Channel
```

```
57          rx_QPSK_BCH_stream = awgn(
    tx_QPSK_BCH_stream)
58
59          # Demodulate(QPSK)
60          rx_BCH_stream = psk.
    demodulate(rx_QPSK_BCH_stream)
61          rx_bin_bit_stream = decoder(
    rx_BCH_stream[:7])
62          rx_bin_bit_stream = np.
    append(rx_bin_bit_stream, decoder(
    rx_BCH_stream[7:]))
63
64          rx_bin_Conv = np.append(
    rx_bin_Conv, rx_bin_bit_stream)
65
66      rx_bin_Conv = rx_bin_Conv[:
    360126]
67
68      # The rate 1/3 code demodulate
    and decoding
69      rx_bin_one_third =
    decoder_one_third(rx_bin_Conv)
70      print(rx_bin_one_third.size)
71
72      tx_bin = tx_bin[:Npixels * 8]
73
74      Conv_Coding_One_third = np.sum(
    tx_bin != rx_bin_one_third[tblen
    :]) / (Npixels * 8)
```

```
75      print(Conv_Coding_One_third)
76
77      y_Conv_One_Third = np.append(
    y_Conv_One_Third,
    Conv_Coding_One_third)
78
79      x = np.append(x, (snrdb / 10))
80
81  plt.figure()
82
83  # plot ber
84  plt.scatter(x, y_Conv_One_Third,
    label="FEC BER(BCH(7, 3) + 7
    convolutional codes)")
85
86  plt.title("BER in different SNR")
87  plt.xlabel("SNR(db)")
88  plt.ylabel("BER")
89
90  plt.yscale("log")
91  plt.grid(True)
92  plt.legend(loc="upper right")
93  plt.show()
```