

EE326 Homework 4

20210203 RHO UKJIN

November 17, 2025

1) (7, 4, 3) Hamming Code Simulation

a) Generate 10^6 messages

```
1 import random
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 def hamming743(p : float):
6     '''
7     0) form a parity check matrix H
8     '''
9     H = np.array([
10         [0,0,0,1,1,1,1],
11         [0,1,1,0,0,1,1],
12         [1,0,1,0,1,0,1]
13     ])
14     msg_list = []
15     output_list = []
16     for _ in range(10**6):
17         '''
18         a) make 10^6 length-4 binary codeword
19         '''
20         msg_list.append(list(map(int, f"{random.randint(0, 15):04b}")))
```

Before generate 10^6 random binary sequence, I defined parity check matrix H as an array. And made 10^6 number of binary sequence using for and random.randint function. Because random.randint(0, 15):04b function generates integer between 0, 15 with same probability, this binary sequence is generated uniformly.

b) Send the corresponding length-7 binary codeword through binary symmetric channel with transition probability p .

```
1     for msg in msg_list:
2         '''
3         b) for each message, make codeword
4         used p1=b+c+d, p2=a+c+d, p3=a+b+d
5         and according to p and BSC, send codeword
6         '''
7         msg.append((msg[1]+msg[2]+msg[3])%2)
8         msg.append((msg[0]+msg[2]+msg[3])%2)
9         msg.append((msg[0]+msg[1]+msg[3])%2)
10        output_list.append(msg.copy())
11        for i in range(7):
12            if random.random() < p:
```

```

13         msg[i] = (msg[i] + 1) % 2
14     temp = msg_list
15     msg_list = output_list
16     output_list = temp

```

From this code, for each message, I added 3 parity bits behind 4 binary bits using the condition below.

$$p_1 = b + c + d, p_2 = a + c + d, p_3 = a + b + d$$

I copied this length-7 binary codeword to another list named "output_list". And used random.random function to implement binary symmetric channel with probability p. Not to confuse, I interchanged msg_list and output_list.

c) Estimate the transmitted codeword according to the following decoding procedure

```

1     '''
2     c) if Hr == 0, estimate as r.
3     if Hr != 0, correct r.
4     '''
5     count_undetected = 0
6     count_corrected = 0
7     count_not_single_error = 0
8     for i in range(10**6):
9         Hr = np.dot(H, np.array(output_list[i])) % 2
10        h_i = Hr[0] * 4 + Hr[1] * 2 + Hr[2]
11        if h_i != 0:
12            output_list[i][h_i - 1] = (output_list[i][h_i - 1] + 1) % 2
13            if output_list[i] == msg_list[i]:
14                count_corrected += 1
15            else: count_not_single_error += 1
16        else:
17            if output_list[i] != msg_list[i]:
18                count_undetected += 1

```

By multiplying two matrix H and output_list[i], we can get exact position of flipped bit. Because every '+' represents the modulo 2-sum, I added every summation code to '%2'. After we get Hr , we can get h_i by changing binary to integer. Then, we can correct the error by flipping the i -th bit(if h_i is not 0).

d)-g) Compute the probability and plot it

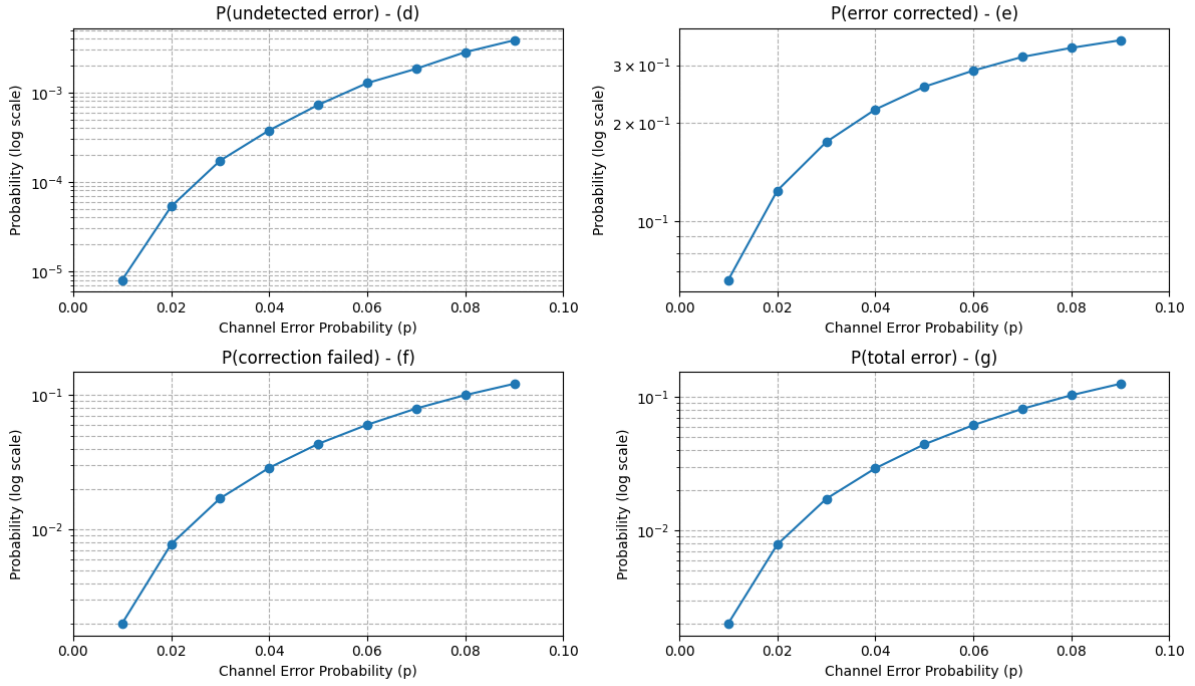
```

1     '''
2     d) e) f) g) calculate the probability and plot it
3     '''
4     P_u = count_undetected / 10**6
5     P_dc = count_corrected / 10**6
6     P_du = count_not_single_error / 10**6
7     P_t = P_u + P_du
8     return P_u, P_dc, P_du, P_t
9
10 def plot_hamming(p_values):
11     ...

```

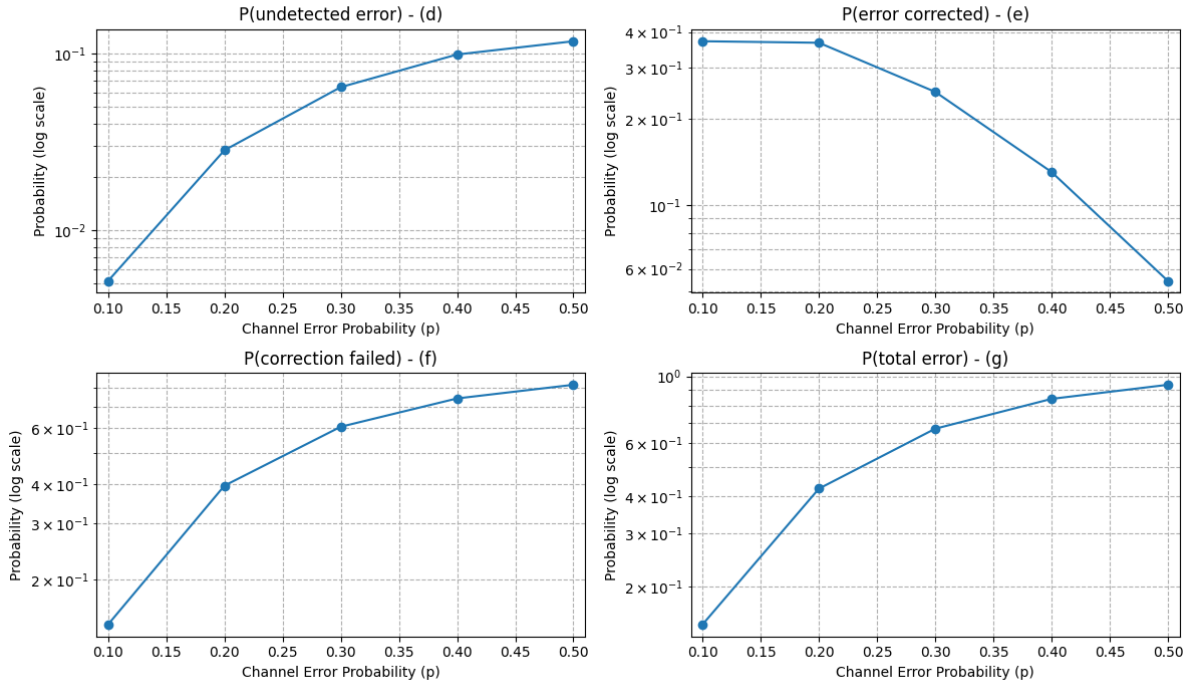
In code c), we added variable count_undetected, count_corrected, count_not_single_error to calculate the number of events of d), e), f). And I got probability by dividing to 10^6 and got P_t

Hamming Code Performance (Log Scale)



(a) Hamming code performance for $p = 0.01$ to 0.09

Hamming Code Performance (Log Scale)



(b) Hamming code performance for $p = 0.1$ to 0.5

by adding P_u and P_{du} .

Discuss d)-g):

d) Probability of undetected error: This plot represents the probability that an error occurred,

but the value of H_r is 0 fooling the decoder. The (7,4,3) Hamming code has a minimum distance $d_{min} = 3$. Therefore, this event requires at least 3 errors. The probability is dominated by 3-bit errors, $P(E = 3) = \binom{7}{3}p^3(1-p)^4 \approx 35p^3$ for small p . Because P_u is proportional to p^3 , it is significantly smaller than P_{du} ($\propto p^2$) and P_{dc} ($\propto p$) at low error rates.

e) Probability of detected and corrected: In this case, the probability of this event is $P(E = 1) = \binom{7}{1}p(1-p)^6$. The plot shows P_{dc} peaking around $p \approx 0.1$ and then decreasing. This is expected, as p grows, the $(1-p)^6$ term begins to shrink, and the probability of multiple errors (which are not corrected) becomes more significant than the probability of a single error.

f) Probability of detected but uncorrected: This plot represents the probability that the correction algorithm failed, resulting in an incorrect final codeword. This event is dominated by 2-bit errors. The probability is $P(E = 2) = \binom{7}{2}p^2(1-p)^5 \approx 21p^2$ for small p . As seen in the plots, this p^2 relationship means P_{du} starts much lower than P_{dc} (which is $\propto p$) but rises much more steeply.

g) Probability of total error: This plot shows the total probability of decoding failure, which is the sum $P_t = P_u + P_{du}$. A key observation is that the plot for P_t is visually identical to the plot for P_{du} . This confirms that the total failure rate of the (7,4) Hamming code is overwhelmingly dominated by the "correction failed" events (2-bit errors). The probability of an undetected error (3-bit errors) is negligible in comparison.

2) Repeat 1) for the (15, 11, 3) Hamming code

The main code difference of (15,11,3) Hamming code and (7,4,3) Hamming code

```
1 ...
2 H = np.array([
3     [0,0,0,0,0,0,0,1,1,1,1,1,1,1,1],
4     [0,0,0,1,1,1,1,0,0,0,0,1,1,1,1],
5     [0,1,1,0,0,1,1,0,0,1,1,0,0,1,1],
6     [1,0,1,0,1,0,1,0,1,0,1,0,1,0,1]
7 ])
8 ...
9     msg_list.append(list(map(int, f"{random.randint(0, 2**11 - 1)
10         :011b}"))))
11
12 for j in range(10**6):
13     '''
14     b) for each message, make codeword
15     and according to p and BSC, send codeword
16     '''
17     codeword = [0] * 15
18
19     data_indices = [2, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14]
20     for i in range(11):
21         codeword[data_indices[i]] = msg_list[j][i]
22
23     codeword[7] = (codeword[8] + codeword[9] + codeword[10] +
24         codeword[11] +
25         codeword[12] + codeword[13] + codeword[14]) % 2
26     codeword[3] = (codeword[4] + codeword[5] + codeword[6] +
27         codeword[11] +
28         codeword[12] + codeword[13] + codeword[14]) % 2
29     codeword[1] = (codeword[2] + codeword[5] + codeword[6] +
30         codeword[9] +
31         codeword[10] + codeword[13] + codeword[14]) % 2
32     codeword[0] = (codeword[2] + codeword[4] + codeword[6] +
33         codeword[8] +
34         codeword[10] + codeword[12] + codeword[14]) % 2
35
36     output_list.append(codeword.copy())
37     for i in range(15):
38         if random.random() < p:
39             codeword[i] = (codeword[i] + 1) % 2
40     msg_list[j] = codeword
41 temp = msg_list
42 msg_list = output_list
43 output_list = temp
44 ...
45 count_undetected = 0
46 count_corrected = 0
47 count_not_single_error = 0
48 for i in range(10**6):
49     Hr = np.dot(H, np.array(output_list[i])) % 2
50     h_i = Hr[0] * 8 + Hr[1] * 4 + Hr[2] * 2 + Hr[3]
51     if h_i != 0:
52         output_list[i][h_i - 1] = (output_list[i][h_i - 1] + 1) % 2
53         if output_list[i] == msg_list[i]:
54             count_corrected += 1
55         else: count_not_single_error += 1
```

```

51         else:
52             if output_list[i] != msg_list[i]:
53                 count_undetected += 1
54     ...

```

In this code, we changed H to (15,11,3) parity check matrix, random.randint variables to match the dimension. Also, to add the parity bits, for every messages, made codeword with 4 parity bits. codeword = [p1 p2 x1 p3 x2 x3 x4 p4 x5 x6 x7 x8 x9 x10 x11]. And made copy of codeword to output_list. Other things are similar as before.

The main difference between 1) and 2) is overall error probability, performance crossover point and code rate.

1. Higher overall error probability

P_{dc} : The (15, 11, 3) plot is higher because it has more than double the chance of experiencing a 1-bit error.

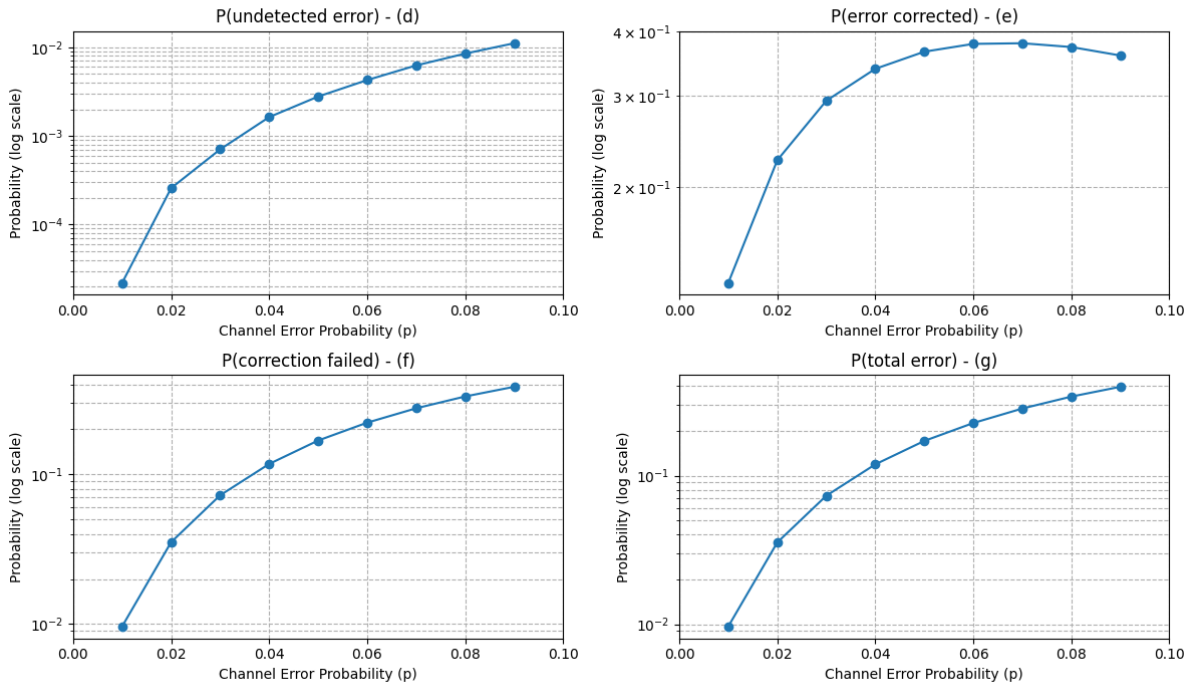
P_{du} : The (15, 11, 3) code is 5 times more likely to suffer a 2-bit error than the (7, 4, 3) code. This is the most significant difference and is visually apparent in the higher, steeper P_{du} curve.

2. Earlier performance crossover point

A Hamming code is only useful as long as it corrects more errors than it creates (i.e., $P_{dc} > P_{du}$). The point where $P_{dc} \approx P_{du}$ marks the limit of the code's usefulness. (7, 4, 3) code has point on between 0.1 and 0.15 whereas (15, 11, 3) code has point on about 0.2.

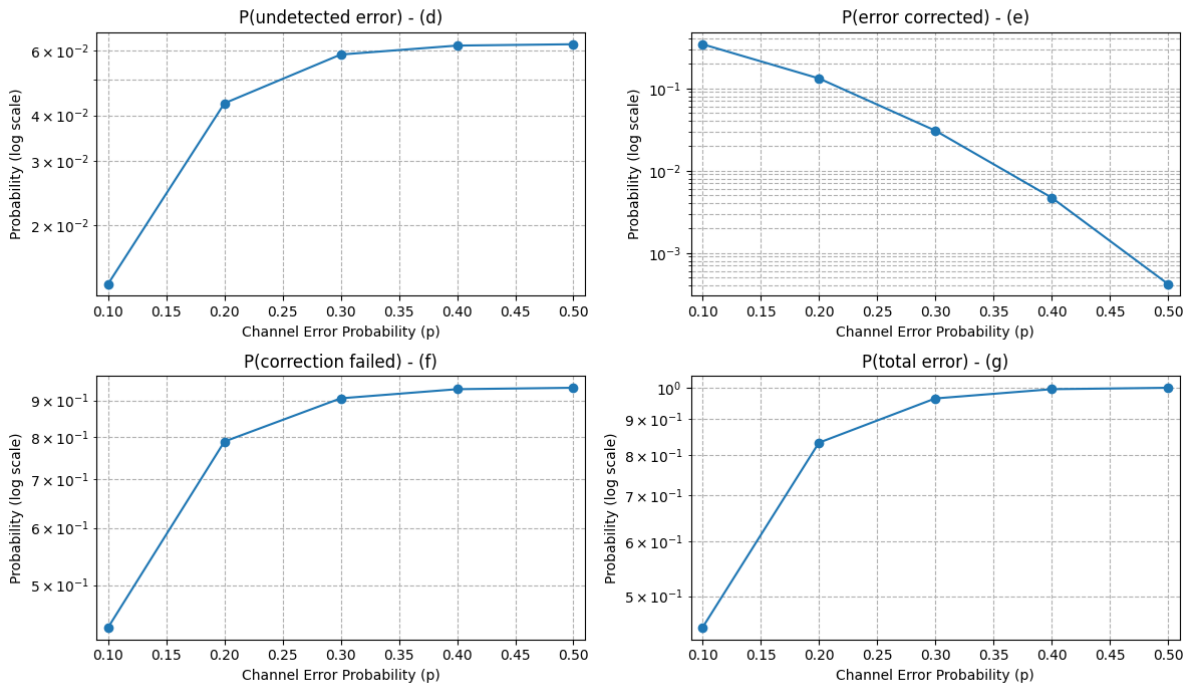
3. Higher code rate The (7, 4, 3) code has a low rate ($R = 4/7 \approx 0.57$) but the (15, 11, 3) code has a much higher, more efficient rate ($R = 11/15 \approx 0.73$). But (7, 4, 3) code has lower probability of error which is a tradeoff of code rate.

Hamming Code Performance (Log Scale)



(a) Hamming code performance for $p = 0.01$ to 0.09

Hamming Code Performance (Log Scale)



(b) Hamming code performance for $p = 0.1$ to 0.5