レポート提出票

科目名: 情報工学実験2

実験テーマ: 実験テーマ3 情報通信シミュレーション

実施日: 2020年 12月 14日

学籍番号: 4619055

氏名: 辰川力駆

共同実験者:

1　実験概要

ハミング符号を用いたシミュレーションプログラムを作成し、誤り率を求め、グラフを参考にしながら、情報通信の特性や原理を理解する。

2　目的

通信システムの「シミュレーション」を通し、ディジタル通信システムと誤り訂正符号の理解を深める。ここでは、コンピューター・シミュレーションと呼ばれるもので、コンピューター上で仮想的な通信環境を構築し実験を行うものである。

3　原理

3.1　誤り訂正符号

情報を伝えるとき、できるだけ正確に伝えるための仕組みである。送信メッセージに $N-K$ ビットの冗長性をもたせることで、通信路で発生した誤りを訂正することが可能である。

- 符号化 (Encode)

  送信者は K ビットの送りたい情報 $\boldsymbol{w}$ をルールに従って N ビットの符号語 $\boldsymbol{x}$ に変換する。

- 復号 (Decode)

  受信者は $\boldsymbol{x}$ がわからず、ノイズ付きの情報 $\boldsymbol{y}$ のみを知っている。$\boldsymbol{y}$ から $\hat{\boldsymbol{x}}$ および $\hat{\boldsymbol{w}}$ を推定する。

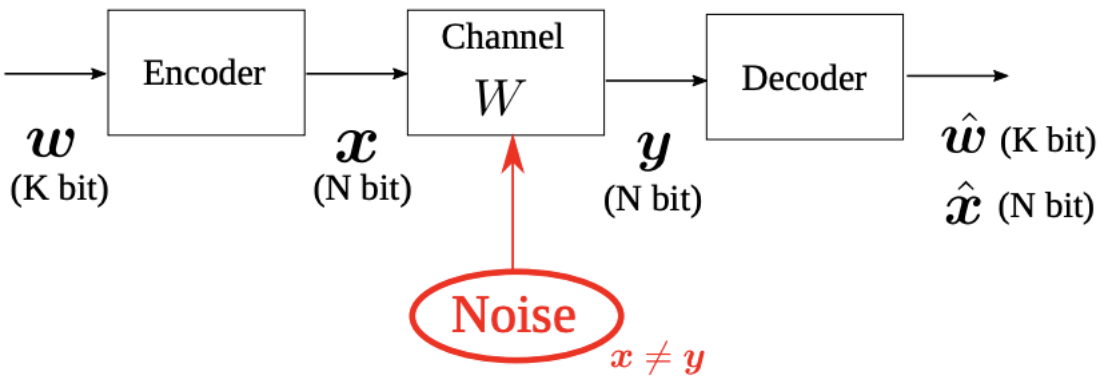

図 1: 通信路モデル

3.2　ハミング符号

ハミング符号はブロック符号の一種である。デジタル情報をブロック (K ビット) に分け、それぞれに対して符号化を行う (N ビットになる)。

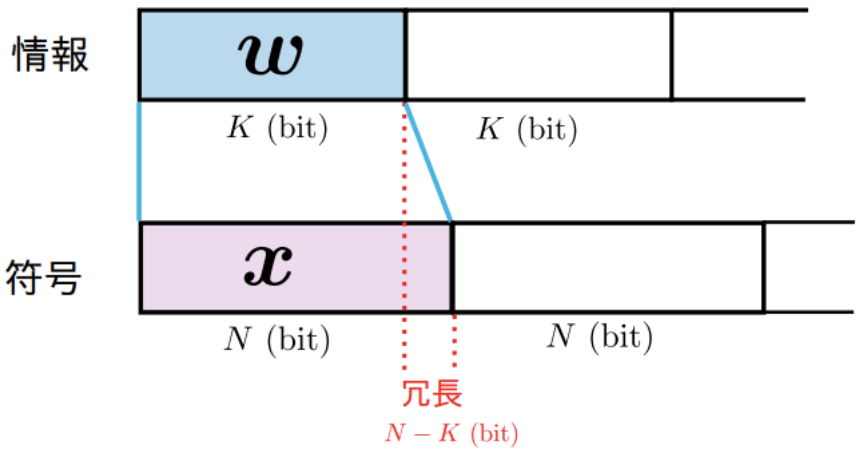

図 2: ブロック符号化

4　実験手順

1. $K = 4$ ビットの情報 $\boldsymbol{w}$ を乱数を用いて生成
2. 符号化により、$\boldsymbol{w}$ から $\boldsymbol{x}$ を生成
3. 乱数を用いて $N = 7$ ビットの雑音ベクトル $\boldsymbol{e}$ を生成
4. $\boldsymbol{y} = \boldsymbol{x} \oplus \boldsymbol{e}$ を計算
5. $\boldsymbol{y}$ から復号して $\hat{\boldsymbol{x}}$、$\hat{\boldsymbol{w}}$ を得る
6. $\boldsymbol{w}$ と $\hat{\boldsymbol{w}}$ を比較し、ビット誤り数をカウント
7. 1 から 6 を SIM 回行い、ビット/ブロック誤り率を計算

5　実験結果

ソースコードは付録に記述した。そのソースコードを実行した結果を下記に示す。

なお、誤り率を $\epsilon = 0.005, 0.009, 0.013, 0.017, 0.021$、シミュレーション回数を $\mathrm{SIM} = 1000000$ に設定した。

また、これらの結果をグラフにプロットした。

```
# SIM:1000000
#BSCの誤り率 #復号後ビット誤り率 #復号後ブロック誤り率
0.005        0.00021625         0.000514
0.009        0.00072750         0.001684
0.013        0.00146725         0.003375
0.017        0.00245400         0.005629
0.021        0.00375825         0.008681
```

図 3: ソースコードの実行結果

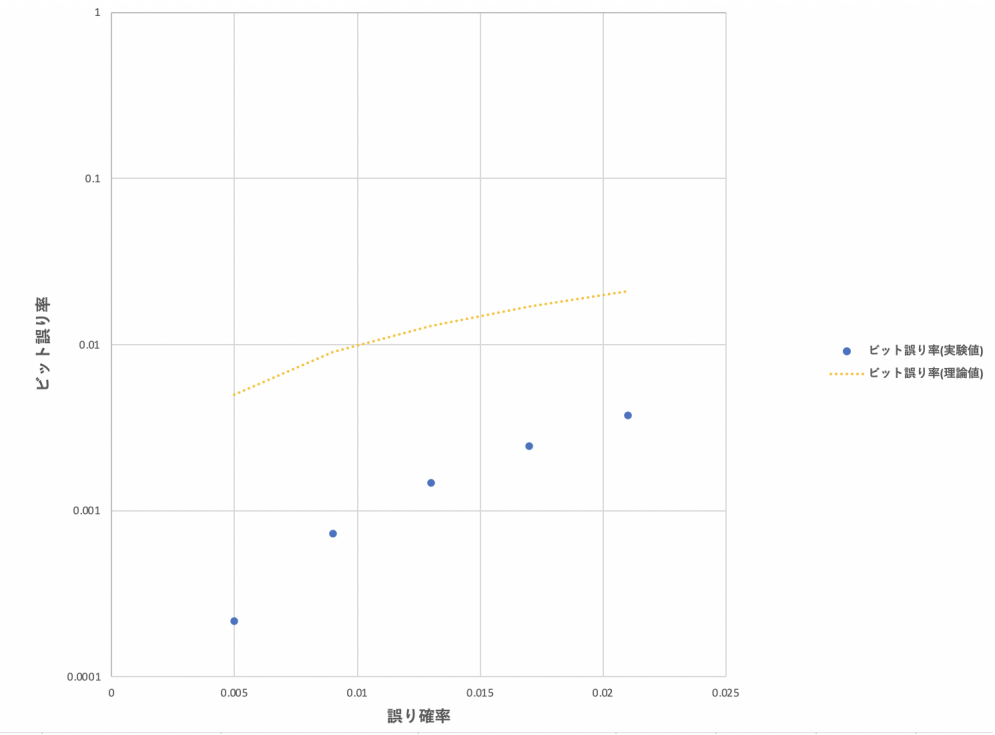

図 4: ビット誤り率のグラフ

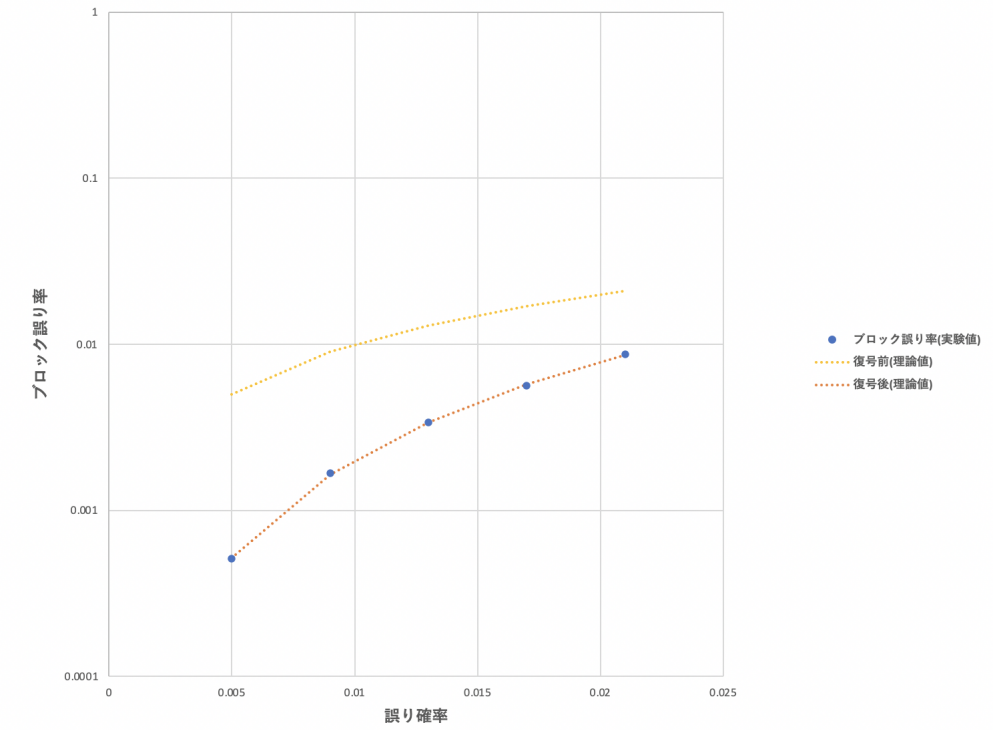

図 5: ブロック誤り率のグラフ

6　検討

6.1　課題 1

ϵ を非常に小さくした場合 (10^{-4} 程度)、誤り率を正しく測定するためにはどの程度のシミュレーション回数が必要か。

今回はシミュレーション回数を 1000000 回で行ったが、試しに 100, 1000, ... 回で行うと、1000000 回の時より 10 %前後誤差があった。したがって、正しく測定するためには 1000000 回くらいは必要であると考える。

6.2 課題 2

ハミング符号によって P_e はどの程度改善されたか。

7 ビットの中で一つも誤りが生じない確率は $(1-\epsilon)^7$ である。よって 1 つ以上の誤りが生じる確率は、余事象を考えて $1-(1-\epsilon)^7$ である。これが本来の誤り確率である。

次にハミング符号を用いた場合を考える。ハミング符号を用いると 1 ビットまでなら誤りを訂正することができる。1 ビット誤りが生じる確率は $7(1-\epsilon)^6\epsilon$ であるから、ハミング符号を用いた場合の誤り確率は $1-(1-\epsilon)^7-7(1-\epsilon)^6\epsilon$ となる。よって、$7(1-\epsilon)^6\epsilon$ だけ誤り確率が減ると考える。

6.3 課題 3

ブロック誤り率の復号前の理論値、復号後の理論値を考察し、実験で得られた値と比較せよ。

実験結果の図 5 のグラフを見ると、当たり前ではあるが、実験値は復号前の理論値より常に下に存在する。

また、実験値は復号後の理論値とほぼ等しくなった。これは、ブロック誤り率の定義として、1 ビットの誤りがあれば、ブロックが誤りとして判定しているからである。

7 結論

(7,4) ハミング符号において、1 ビットの誤りを訂正することができ、グラフから分かるように明らかに復号前より誤りが減ることが分かった。また、実験を行う際はシミュレーションを 1 万回程度じゃなく 100 万回以上行わなければならない。

A 付録

ソースコード 1: kadai3_3.cpp

```cpp
//4619055 辰川力駆
#include <random> // 乱数生成
#include <stdio.h>
#include <iostream>
#include <iomanip>

using namespace std;

#define N 7 ///符号化後のビット数
#define K 4 ///デジタル情報の分けるブロックのビット数
#define seed 55 ///学籍番号下２桁
#define SIM 1000000 ///シミュレーション回数

mt19937 mt(seed); ///メルセンヌ・ツイスタ

int main()
{
    uniform_real_distribution<double> rand_real(0, 1);
    normal_distribution<double> rand_n(0, 0.3);

    int w[K]; //4ビットの情報w
    int x[N]; //7ビットの符号語x
    int e[N]; //7ビットの雑音ベクトルe
    int y[N]; //7ビットの受信語y
    int s[3]; ///シンドロームs
    int bit_count = 0; ///シミュレーションごとの毎回のビット誤り数を計算
    int total_bit_count = 0; ///誤ったビットの総数
    int total_block_count = 0; ///ブロック単位の誤りの総数
    int EstimationPosition; ///誤り位置推定場所
    double ep; ///誤り率

    cout << "# SIM:" << SIM << endl;
    cout << "#BSC の誤り率 #復号後ビット誤り率 #復号後ブロック誤り率" << endl;

    for (int k = 0; k < 5; k++)
    {
        ep = 0.005 + k * 0.004;
        total_bit_count = 0;
        total_block_count = 0;

        for (int sim = 0; sim < SIM; sim++)
        {
            bit_count = 0;

            for (int i = 0; i < K; i++) //K=4 ビットの情報 w を乱数を用いて生成
            {
                w[i] = rand_real(mt) * 2;
            }
```

```
49
50              for (int i = 0; i < K; i++) ///符号化により，w から x を生成
51              {
52                  x[i] = w[i];
53              }
54              x[N − K + 1] = w[0] ^ w[1] ^ w[2];
55              x[N − K + 2] = w[1] ^ w[2] ^ w[3];
56              x[N − K + 3] = w[0] ^ w[1] ^ w[3];
57
58              for (int i = 0; i < N; i++) ///乱数を用いてN=7ビットの雑音ベクトル
                    e を生成
59              {
60                  if (rand_real(mt) <= ep)
61                  {
62                      e[i] = 1;
63                  }
64                  else
65                  {
66                      e[i] = 0;
67                  }
68              }
69
70              for (int i = 0; i < N; i++) //x と e の排他的論理和を計算して y とする
71              {
72                  y[i] = x[i] ^ e[i];
73              }
74
75              s[0] = y[0] ^ y[1] ^ y[2] ^ y[4]; //y から復号して x ハット、w ハットを得る
76              s[1] = y[1] ^ y[2] ^ y[3] ^ y[5];
77              s[2] = y[0] ^ y[1] ^ y[3] ^ y[6];
78
79              int point = 0;
80              for (int i = 0; i < 3; i++)
81              {
82                  point += s[i] * pow(2, 2 − i);
83              }
84              switch (point)
85              {
86              case 5:
87                  EstimationPosition = 1;
88                  break;
89              case 7:
90                  EstimationPosition = 2;
91                  break;
92              case 6:
93                  EstimationPosition = 3;
94                  break;
95              case 3:
96                  EstimationPosition = 4;
97                  break;
98              case 4:
99                  EstimationPosition = 5;
```

```
100                     break;
101                 case 2:
102                     EstimationPosition = 6;
103                     break;
104                 case 1:
105                     EstimationPosition = 7;
106                     break;
107                 default:
108                     EstimationPosition = -1;
109                     break;
110                 }
111                 if (EstimationPosition != -1)
112                 {
113                     y[EstimationPosition - 1] = y[EstimationPosition - 1] ^ 1;
114                 }
115
116                 for (int i = 0; i < K; i++) //w と w ハットを比較し、ビット誤り数をカウント（つまり 4ビットまでを見れば良い）
117                 {
118                     bit_count += w[i] ^ y[i];
119                 }
120
121                 total_bit_count += bit_count;
122
123                 if (bit_count != 0)
124                 {
125                     total_block_count += 1;
126                 }
127             }
128             cout << ep;
129             cout << fixed << setprecision(8) << "␣␣␣␣␣␣␣␣" << (double)total_bit_count / (K * SIM);
130             cout << fixed << setprecision(6) << "␣␣␣␣␣␣␣␣␣␣" << (double)total_block_count / SIM << endl;
131             cout.unsetf(ios::fixed); ///体裁を整えている
132         }
133         return 0;
134     }
```