学籍番号: 4617041 氏名: 白江涼雅

課題3 ハミング符号の繰り返しシミュレーション

1 目的

課題 1,2 で行った内容を踏まえてシミュレーションプログラムを作成し、仮想的な通信環境を構築する技術を身につける.

2 実験装置

- windows X シリーズ
- Visualstdio2013

3 実験結果

実行結果は以下の通りになった. またグラフは最後のページにまとめた.

表 3: 試行回数 100000 の時のブロック誤り率と理論値

ε	ブロック誤り率の出力結果
0.002500	0.000045
0.012500	0.001442
0.022500	0.004248
0.032500	0.008690
0.042500	0.014722
0.052500	0.021600
0.062500	0.028650
0.072500	0.038812
0.082500	0.047460
0.092500	0.058490
0.102500	0.069642

4 検討事項

1. 今回計測した ε の区間で BER が 0 になる確率が一番高い $\varepsilon = 0.0025$ について考える.
   検討事項 2 にある通り,$P_e = {}_7C_2(0.0025)^2$ より, $P_e \to 0$ とみなし、試行回数を $n \to \infty$ とすると、ポアソン分布に従うとわかる.$P_e * n = \lambda$ とし, 式に表すと

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!} \tag{1}$$

   となる.$k \geq 1$ となれば BER が 0 にならず計算できるため排反を考え,$P(X = 0) \simeq 0$ となる値をみつける. すなわち $P(X = 0) = e^{-\lambda} \simeq 0$ を考えれば良い.

表 2:試行回数 n に対するハミング符号の誤り率 $P(X=0)$ の値

試行回数 n	λ	$P(X=0)=e-^{\lambda}$
100	$1.31*10^{-2}$	0.99
1000	$1.31*10^{-1}$	0.88
10000	1.31	0.27
100000	13.1	$2.04*10^{-6}$

直感的ではあるが,$n=10000$ までは BER が 0 になる可能性があが,$n=100000$ ではほぼ 0 となる. これより 10 万回以上の試行を行うことが推薦される.

2. ハミング符号を用いない場合は P_e と ε は同等の確率で誤る. 一方, 実験課題 2 であったように, ハミング符号を用いると 7bit の中にも誤りがない場合と 1 つ誤りがある場合は正しく受け取ることができる. よって BER は

$$P_e = {}_7C_2 * \varepsilon^2 \tag{2}$$

となる. よってハミング符号を用いた時の方が P_e は低くなる.

3. 以下の表にまとめる.

表 3: 試行回数 100000 の時のブロック誤り率と理論値

ε	ブロック誤り率の出力結果	理論値
0.002500	0.000100	0.00013
0.012500	0.003300	0.00327
0.022500	0.009850	0.01063
0.032500	0.020100	0.0221
0.042500	0.033820	0.038
0.052500	0.049110	0.058
0.062500	0.065020	0.062
0.072500	0.087920	0.082
0.082500	0.106780	0.110
0.092500	0.131650	0.143
0.102500	0.155610	0.181

比較すると, 全体的に理論値のほうがやや大きい.

5 まとめ

今回の実験では, ハミング符号の役割や通信技術についての理解が深まった. また, シミュレーションプログラムの作成及び, 結果について検討することでシミュレーションプログラムの利点や扱い方, 注意しなければならないことを理解することができた.

6 ソースコード

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <random>
#define gyo 7 //ハミング符号
#define retu 4 //ハミング符号
#define k 3 //シンドロームの長さ
#define SIM 100000 //試行回数
#define probability 11 //桁上がり

int main(){
int G[gyo][retu];//生成行列
int H[gyo][k]; //検査行列
int w[retu];  //情報系列
int x[gyo];  //送信系列
int y[gyo];  //受信系列
int e[gyo];  //誤り符号
int s[gyo]; //シンドローム生成
double ran;
int i, j,l,m;
int tmp;
int miss;
double delta_plus = 0.01; //はじめの桁
double delta;
double syoki = 0.0025;
int miss_count;
double answer;

//乱数発生準備
std::mt19937 mt(41);
std::uniform_real_distribution<double> r_rand(0.0, 1.0);


//生成行列で必要な任意にきめるところの生成
G[0][0] = 1; G[0][1] = 0; G[0][2] = 0; G[0][3] = 0;
G[1][0] = 0; G[1][1] = 1; G[1][2] = 0; G[1][3] = 0;
G[2][0] = 0; G[2][1] = 0; G[2][2] = 1; G[2][3] = 0;
G[3][0] = 0; G[3][1] = 0; G[3][2] = 0; G[3][3] = 1;
G[4][0] = 1; G[4][1] = 1; G[4][2] = 1; G[4][3] = 0;
G[5][0] = 1; G[5][1] = 1; G[5][2] = 0; G[5][3] = 1;
G[6][0] = 1; G[6][1] = 0; G[6][2] = 1; G[6][3] = 1;

H[0][0] = 1; H[0][1] = 1; H[0][2] = 1;
H[1][0] = 1; H[1][1] = 1; H[1][2] = 0;
H[2][0] = 1; H[2][1] = 0; H[2][2] = 1;
H[3][0] = 0; H[3][1] = 1; H[3][2] = 1;
H[4][0] = 1; H[4][1] = 0; H[4][2] = 0;
H[5][0] = 0; H[5][1] = 1; H[5][2] = 0;
```

```
H[6][0] = 0; H[6][1] = 0; H[6][2] = 1;


for (m = 0; m < probability; m++){
delta = syoki + delta_plus * m;
miss_count = 0;
for (l = 0; l < SIM; l++){
//w の生成
for (i = 0; i < retu; i++){
ran = r_rand(mt);
if (ran < 0.5){
w[i] = 0;
}
else{
w[i] = 1;
}
}

//x の生成
for (i = 0; i < gyo; i++){
tmp = 0;
for (j = 0; j < retu; j++){
tmp += w[j] * G[i][j];
}
if (tmp % 2 == 0){
x[i] = 0;
}
else{
x[i] = 1;
}
}

//誤り e の生成
for (i = 0; i < gyo; i++){
ran = r_rand(mt);
if (ran < delta){
e[i] = 1;
}
else{
e[i] = 0;
}
}

//送信行列に誤り e を干渉させ, 送信行列をつくる
for (i = 0; i < gyo; i++){
```

```
y[i] = (x[i] + e[i]) % 2;
}

//s の生成
for (i = 0; i < k; i++){
s[i] = 0;
for (j = 0; j < gyo; j++){
s[i] += y[j] * H[j][i];
}
if (s[i] % 2 == 1){
s[i] = 1;
}
else{
s[i] = 0;
}
}

//どこ反転させるかの判定
if (s[0] == 1 && s[1] == 1 && s[2] == 1){
miss = 1;
}
else if (s[0] == 1 && s[1] == 1 && s[2] == 0){
miss = 2;
}
else if (s[0] == 1 && s[1] == 0 && s[2] == 1){
miss = 3;
}
else if (s[0] == 0 && s[1] == 1 && s[2] == 1){
miss = 4;
}
else if (s[0] == 1 && s[1] == 0 && s[2] == 0){
miss = 5;
}
else if (s[0] == 0 && s[1] == 1 && s[2] == 0){
miss = 6;
}
else if (s[0] == 0 && s[1] == 0 && s[2] == 1){
miss = 7;
}
else{
miss = 0;
}

/*printf("%d ビット目を反転させます\n", miss);*/
```

```
if (y[miss - 1] == 0){
y[miss - 1] = 1;
}
else{
y[miss - 1] = 0;
}

for (i = 0; i < retu; i++){
if (y[i] != x[i]){
miss_count++;
}
}
}
answer = (double)miss_count / (SIM * retu);
printf("εが%f のとき:%f\n", delta,answer);
}
}
```

図 2:ハミング符号を用いたシミュレーションのソースコード

7　参考文献

参考文献

[1] 統計 WEB https://bellcurve.jp/statistics/course/6984.html (2018/1012 アクセス)