信州大学工学部

学士論文

ラテン方陣の消失通信路に対する復号誤り特性

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　14T2062H
氏名　豊村 聖也

2019 年 2 月 20 日

目次

1　はじめに

ラテン方陣とは図 1,2,3 のような各行各列の数字または記号が重複のないように配列したものである．ラテン方陣はオイラーが 1779 年に出した「36 人士官の問題」を機会に数学的な研究が行われてきた [1]．また，ラテン方陣から派生して様々なパズルが生み出された．その一種であるサイズ 9 のラテン方陣にルールを追加した「数独」は世界的に人気であり、様々な分野の研究者が研究対象に取り上げていて，数独を解くという行為が消失通信路の復号に一致していることが知られている [2]．本研究では，この先行研究に着想を得て，ラテン方陣のますを空白にして，そこを埋める過程を通信における受信語から符号語を復号する過程とみなし，消失通信路に対するサイズ 3,4,5,6 のラテン方陣の復号誤りを調べた．

本論文の構成は以下の通りである．2 章では同値関係の定義を述べる．次に，3 章ではラテン方陣の性質を説明する．そして，4 章ではラテン方陣の性質を利用した通信システムについて説明する．さらに，5 章では復号誤り特性の計算方法及び計測結果を述べる．最後に 6 章でまとめを述べる．

1	2	3
2	3	1
3	1	2

図 1　3 次のラテン方陣の例

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

図 2　4 次のラテン方陣の例

1	2	3	4	5
2	1	4	5	3
3	5	1	2	4
4	3	5	1	2
5	4	2	3	1

図 3　5 次のラテン方陣の例

2　同値関係

ある空でない集合 A 上での二項関係 R とは $A \times A$ の部分集合 $R \subset A \times A$ のことである．例えば，A={0,1,2,3} として “<” の関係を定義すると R= {(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)} となり，(2,1) や (1,1) は含まれない．R^{-1} は逆関係を表し，$R^{-1} = \{(y,x) : (x,y) \in R\}$ である．二項関係の中で特によく用いられる関係に同値関係がある．

定義　同値関係

ある集合 A 上の同値関係 E とは次の条件を満たす二項関係 $E \subset A \times A$ である．
(1) 全ての $x \in A$ に対し，$(x,x) \in E$ である．(反射律)
(2) $(x,y) \in E$ ならば $(y,x) \in E$ である．(対称律)
(3) $(x,y) \in E$ かつ $(y,z) \in E$ ならば $(x,z) \in E$ である．(推移律)

$(x, y) \in E$ のとき，$x \sim y$ と書き，「x と y は同値である」という．任意の要素 $x \in A$ に対して，$[x]_E = \{y \in A : y \sim x\}$，すなわち x と同値関係 E にある全ての要素の集合を $[x]_E$ と書く．その集合を x の同値類といい，そのときの x を代表元と呼ぶ．もし $y \in [x]_E$ なら $[y]_E = [x]_E$ となる．すなわち，同じ同値類から，異なる代表元を取っても，その同値類は同じとなる [3].

3　ラテン方陣の性質

n 個の文字からなる集合 Ω の各文字を n 回ずつ使って，合計 n^2 個を n 行 n 列の正方形のますに配置し，各行と各列に文字の重複がないものをサイズ n のラテン方陣という．ラテン方陣 L に対して，(a) 行を並べ替えること，(b) 列を並べ替えること，(c) 記号を置き換えること，を行ってもやはりラテン方陣が得られる．そのほかに，(d) 行列の転置に当たる操作を施してもやはりラテン方陣が得られる．以上の操作によって，ラテン方陣の同値関係を定義する [4]．(a),(b),(c) のみの操作で作ったラテン方陣は英語で “isotopy classes” と表され，(a),(b),(c),(d) の操作で作ったラテン方陣を英語で “main classes” と表される．本論文での同値類は “main classes”’ を指す．

6 次ラテン方陣に対する (a),(b),(c),(d) の操作の例を図 4,5,6,7 にそれぞれ示す．

1	2	3	4	5	6
2	3	1	6	4	5
3	1	2	5	6	4
4	5	6	1	2	3
5	6	4	3	1	2
6	4	5	2	3	1

$\longrightarrow$

6	4	5	2	3	1
2	3	1	6	4	5
3	1	2	5	6	4
4	5	6	1	2	3
5	6	4	3	1	2
1	2	3	4	5	6

図 4　1 行目と 6 行目の入れ替え

1	2	3	4	5	6
2	3	1	6	4	5
3	1	2	5	6	4
4	5	6	1	2	3
5	6	4	3	1	2
6	4	5	2	3	1

$\longrightarrow$

6	2	3	4	5	1
5	3	1	6	4	2
4	1	2	5	6	3
2	5	6	1	2	4
3	6	4	3	1	5
1	4	5	2	3	6

図 5　1 列目と 6 列目の入れ替え

1	2	3	4	5	6
2	3	1	6	4	5
3	1	2	5	6	4
4	5	6	1	2	3
5	6	4	3	1	2
6	4	5	2	3	1

$\longrightarrow$

6	2	3	4	5	1
2	3	6	1	4	5
3	6	2	5	1	4
4	5	1	6	2	3
5	1	4	3	6	2
1	4	5	2	3	6

図 6　1 と 6 を置き換え

1	2	3	4	5	6
2	3	1	6	4	5
3	1	2	5	6	4
4	5	6	1	2	3
5	6	4	3	1	2
6	4	5	2	3	1

$\longrightarrow$

1	2	3	4	5	6
2	3	1	5	6	4
3	1	2	6	4	5
4	6	5	1	3	2
5	4	6	2	1	3
6	5	4	3	2	1

図 7　正方形の対角線を中心に行と列を入れ替え

ラテン方陣の列を任意に並べ変えること，第一行を固定して第 2 行以下を任意に並べ変えることだけを採用して分類すると，第 1 行と第 1 列が $1, 2, 3, \cdots, n$ の順になる標準形のラテン方陣ができる．サイズ n のラテン方陣の総数は，標準形のラテン方陣の個数の $n!(n-1)!$ 倍に等しい．

$n \leq 11$ に対して，サイズ n のラテン方陣の同値類及び標準形の数が知られている．[5]．表 1 にラテン方陣の同値類と標準形の数を示す．

ここで，本研究の計測対象であるサイズ 3,4,5,6 のラテン方陣の個数の詳細を述べる．N を標準形のラテン方陣の個数とする．まず，サイズ 3 の場合は 3 次巡回群 Z_3 の乗積表の同値類のラテン方陣 (図 8) しかない．次に，サイズ 4 の場合は Z_4(図 9) と $Z_2 \times Z_2$(図 10) の二つの同値類があり，それぞれ N=3,N=1 である．また，サイズ 5 の場合は Z_5 の同値類 (図 11) 以外に図 12 で代表される 2 種類の同値類があり，それぞれ N=6，N=50 である．最後に，サイズ 6 の場合は同値類が 12 種類あり，図 13 の A は N=20，B は N=40，C は N=60，D は N=108，E は N=180，F は N=360，G は N=540，H は N=1080，I は N=1080，J は N=1080，K は N=1620，L は N=3240 である [6]．

表 1　サイズ n のラテン方陣の同値類と標準形の数

サイズ	同値類の数	標準形の数
1	1	1
2	1	1
3	1	1
4	2	4
5	2	56
6	12	9408
7	147	16,942,080
8	283,657	535,281,401,856
9	19,270,853,541	377,597,570,964,258,816
10	34,817,397,894,749,939	7,580,721,483,160,132,811,489,280
11	2,036,029,552,582,883,134,196,099	5,363,937,773,277,371,298,119,673,540,771,840

1	2	3
2	3	1
3	1	2

図 8　同値類を代表するラテン方陣 (サイズ 3)

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

図 9　同値類を代表するラテン方陣 (サイズ 4)

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

図 10　同値類を代表するラテン方陣 (サイズ 4)

1	2	3	4	5
2	3	4	5	1
3	4	5	1	2
4	5	1	2	3
5	1	2	3	4

図 11　同値類を代表するラテン方陣 (サイズ 5)

1	2	3	4	5
2	1	4	5	3
3	5	1	2	4
4	3	5	1	2
5	4	2	3	1

図 12　同値類を代表するラテン方陣 (サイズ 5)

A

1	2	3	4	5	6
2	3	1	6	4	5
3	1	2	5	6	4
4	5	6	1	2	3
5	6	4	3	1	2
6	4	5	2	3	1

B

1	2	3	4	5	6
2	3	1	6	4	5
3	1	2	5	6	4
4	5	6	2	3	1
5	6	4	1	2	3
6	4	5	3	1	2

C

1	2	3	4	5	6
2	3	1	5	6	4
3	1	2	6	4	5
4	5	6	2	1	3
5	6	4	1	3	2
6	4	5	3	2	1

D

1	2	3	4	5	6
2	1	6	5	4	3
3	4	1	2	6	5
4	6	5	1	3	2
5	3	2	6	1	4
6	5	4	3	2	1

E

1	2	3	4	5	6
2	3	4	5	6	1
3	4	2	6	1	5
4	5	6	1	2	3
5	6	1	2	3	4
6	1	5	3	4	2

F

1	2	3	4	5	6
2	3	1	5	6	4
3	1	2	6	4	5
4	5	6	1	2	3
5	6	4	3	1	2
6	4	5	2	3	1

G

1	2	3	4	5	6
2	3	4	6	1	5
3	4	2	5	6	1
4	5	6	1	2	3
5	6	1	3	4	2
6	1	5	2	3	4

H

1	2	3	4	5	6
2	1	6	5	4	3
3	5	1	6	2	4
4	3	2	1	6	5
5	6	4	2	3	1
6	4	5	3	1	2

I

1	2	3	4	5	6
2	1	5	3	6	4
3	4	6	1	2	5
4	5	2	6	1	3
5	6	4	2	3	1
6	3	1	5	4	2

J

1	2	3	4	5	6
2	1	5	6	3	4
3	6	1	2	4	5
4	5	2	1	6	3
5	4	6	3	1	2
6	3	4	5	2	1

K

1	2	3	4	5	6
2	1	5	6	3	4
3	4	2	1	6	5
4	5	6	2	1	3
5	6	4	3	2	1
6	3	1	5	4	2

L

1	2	3	4	5	6
2	1	4	3	6	5
3	5	6	1	4	2
4	3	5	6	2	1
5	6	1	2	3	4
6	4	2	5	1	3

図 13　同値類を代表するラテン方陣 (サイズ 6)

4　ラテン方陣による通信システム

ラテン方陣は上記に示したように各行各列で数字が重複しないという性質をもつため，いくつか数字を消しても元のラテン方陣を得ることができる．この性質を利用して，ラテン方陣を使用した通信を試みる．想定するシステムは次の通りである．サイズ n のラテン方陣を符号語として用いる．符号語は全てのラテン方陣の中から等確率に選ばれる．1 個の符号語に対し，定常独立な消失通信路が n^2 回使用されて受信語が復号器に届く．消失シンボルは空白のますとして受信される．受信語に対し，消失しなかった数字をヒントとして符号語に使用したラテン方陣が求められる．解が一意であれば誤りなく復号されたことを意味し、複数の解が存在すればそれらは等確率で正しい．

符号語数はサイズ n のラテン方陣の総数である．サイズ n のラテン方陣の個数を M_n とおくと，この符号の符号化率は

$$\frac{1}{n^2}\log_n M_n \tag{1}$$

と表される．消失確率 ε の消失通信路の通信路容量 C が

$$C = 1 - \varepsilon \tag{2}$$

であることから，ラテン方陣と同じ符号化率の符号を用いたときに，任意に小さい復号誤りを達成できる消失確率の上限 (シャノン限界) は

$$\varepsilon^* = 1 - \frac{1}{n^2}\log_n M_n \tag{3}$$

と表される．本研究の興味の 1 つは，ラテン方陣の復号誤り特性をシャノン限界と比較することである．

5　復号誤り特性

本研究の目的は，消失通信路をラテン方陣によって符号化したときの平均復号誤り確率を求めることである．一般に，受信語に対して，消失シンボルの数と復号誤りの間には明確な関係は無い．また，符号語によって復号誤り特性が異なる．しかし，だからといって全ての各ラテン方陣に対してあらゆる消失パターンを試すには膨大な時間がかかる．そのため，各サイズの同値類を代表するラテン方陣の復号誤り特性を計測する．計測対象が同値類を代表するラテン方陣で良い理由は，同じ同値類に属するラテン方陣の復号誤り特性は全て同じとなるからである．

本研究では，計測簡単のため，一意に解が定まらない場合は誤りとみなす．受信語の空白のますの数が 3 以下のときは解は一意に定まるので，復号誤りは起きないが，4 以上では複数の解がある場合がある．また，ラテン方陣のますを全て空白にした場合は必ず復号誤りが起こる．よって，計測対象は空白のますの個数が 4 から n^2-1 の範囲である．具体的な計測方法は以下の通りである．空白のますの個数を 4 から n^2-1 の間で指定した後，ランダムに場所を指定し，指定した場所を空白にする．空白が k 個のときの平均復号誤り確率 f_k を付録のプログラムに同値類を代表するラテン方陣を入力し，そのラテン方陣のますを 4 から n^2-1 個空白にし，それぞれ 10000 通りの消失パターンを試して，解が一意であったときのみ復号誤りが無いとみなして計測し，次のように計算を行うことで復号誤り特性を求めた．なお，プログラムに使用した疑似乱数生成器は現在最も高い評価を得ているメルセンヌ・ツイスタ [7] を使用した．

消失確率が ε のとき k 個のマスが空白になる確率 $p_k(\varepsilon)$ は二項分布を使用して，

$$p_k(\varepsilon) = \binom{n^2}{k} (\varepsilon)^k \times (1-\varepsilon)^{n^2-k} \tag{4}$$

と表される．また，復号誤り特性 $F(\varepsilon)$ は

$$F(\varepsilon) = \sum_{k=0}^{n^2} p_k(\varepsilon) \times f_k \tag{5}$$

となる．

本研究では，サイズ 3,4,5,6 のラテン方陣に対して計測を行った．結果を図 14 に示す．縦軸は平均復号誤り確率，横軸は消失確率となっており，グラフ中にある y 軸平行線は各ラテン方陣におけるシャノン限界を示している．また，表 2 にサイズ 3,4,5,6 のラテン方陣の符号化率とシャノン限界値を示す．

2 元符号が広く使用されていること，サイズが大きいほど送る情報量が大きくなることから，サイズが小さい順に復号誤り確率が低くなると予想した．ほとんどその通りであったが，消失確率が 0.4 より小さい場合は，サイズ 5 の復号誤り確率がサイズ 4 の復号誤り確率よりも低かった．シャノン限界値と計測結果を比較したが，図 14 の通り，ラテン方陣を符号語として用いる通信は復号誤り確率が高いものであることが明らかとなった．

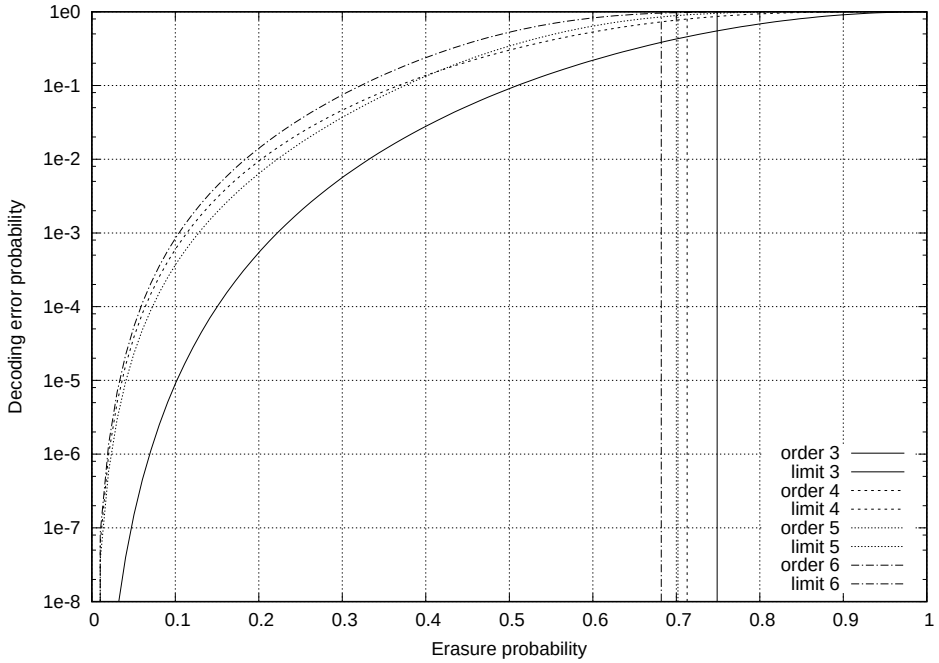

図 14　ラテン方陣における復号誤り特性

表 2　各サイズのラテン方陣の符号化率とシャノン限界値

サイズ	符号化率	シャノン限界値
3	0.251	0.749
4	0.287	0.713
5	0.298	0.702
6	0.318	0.682

6　まとめ

本研究では，ラテン方陣のますを空白にして，そこを埋める過程を通信における受信語から符号語を復号する過程とみなし，消失通信路に対するサイズ 3,4,5,6 のラテン方陣の復号誤り特性を計測した．計測の際，複数の解が発生した場合は誤りとみなしたが，複数の解が存在するときそれらはそれぞれ等確率で正しいと仮定すれば，より正確な特性が得られると考えられる．しかし，その場合は空白が多いほど時間がかかると思われる．ほとんどサイズが小さい順に復号誤り確率が小さくなったが，消失確率の値によっては異なる場合もあった．また，シャ

ノン限界と計測結果を比較し，ラテン方陣を符号語に用いる通信は復号誤り確率が高いことが分かった．しかし，有限体の概念を使用しない通信の一例を示すことができた．そのほかに，同値類の性質を利用して効率的な計測を行うことに成功した．今後の課題としては，消失通信路ではない一般の通信路の復号誤り特性を考えることがあげられる．この問題では，数字や文字が受信の際に変化した場合，ラテン方陣の性質を利用した復号方法が使えないため，代わりの復号方法を考える必要がある．

謝辞

本研究を行うにあたり、指導してくださった指導教員の西新幹彦准教授に感謝の意を表する．

参考文献

[1] 芳沢光雄，群論入門：対称性をはかる数学，講談社，2015.
[2] 比田井亮，「数独の消失通信路に対する復号誤り特性」，信州大学大学院理工学研究科修士論文 (指導教員：西新幹彦)，2014 年 3 月.
[3] 藤原良，神保雅一，符号と暗号の数理，共立出版，1993.
[4] 山本幸一，組合せ数学，朝倉書店，1989.
[5] Brendan D. McKay, Ian M. Wanless, “On the number of Latin squares”, Annals Combinatorics, vol.9, no.3, pp.335–344, Oct. 2005.
[6] 山本幸一，「ラテン方陣について」，数学，vol.12，no.2，pp.67–79，1960.
[7] Mersenne Twister Home Page, http://www.math.sci.hiroshima-u.ac.jp/∼m-mat/MT/mt.html, 2018 年 11 月閲覧.

付録 A　ソースコード

A.1　3 次ラテン方陣の同値類の代表元の復号誤り確率を求めるプログラム

```
#include <stdio.h>
#include <stdlib.h>
#include "MT64.h"

int
bit_count(unsigned int x0)
{
    unsigned int x1 = (x0 & 0x55555555) + ((x0 & 0xaaaaaaaa) >> 1);
    unsigned int x2 = (x1 & 0x33333333) + ((x1 & 0xcccccccc) >> 2);
    unsigned int x3 = (x2 & 0x0f0f0f0f) + ((x2 & 0xf0f0f0f0) >> 4);
    unsigned int x4 = (x3 & 0x00ff00ff) + ((x3 & 0xff00ff00) >> 8);
    unsigned int x5 = (x4 & 0x0000ffff) + ((x4 & 0xffff0000) >> 16);
    return(x5);
}

int
deduce(char *sq)
{
    int app;
    int i, j, s;

    app = 0;
    for (i = 0; i < 16; i++) {
        if (bit_count(sq[i]) != 1) continue;
        s = (i / 3) * 3;
        for (j = s; j < s + 3; j++) {
            if (j == i) continue;
            if (sq[j] & sq[i]) {
                sq[j] &= ~sq[i];
                app++;
            }
        }
        s = i % 3;
        for (j = s; j < s + 9; j += 3) {
            if (j == i) continue;
            if (sq[j] & sq[i]) {
                sq[j] &= ~sq[i];
                app++;
            }
        }
    }
    return(app);
}

void
printlatin(char *sq)
{
    int i;

    for (i = 0; i < 9; i++) {
        printf("%2d", sq[i]);
        if (i % 3 ==2 ) {
            printf("\n");
        }
    }
    printf("\n");

    return;
}

int
```

```
latin(char *sq)
{
    int i, idx, n, num;
    int bc;

    while (deduce(sq) > 0) {
        ;
    }

    n = 4;
    for (i = 0; i < 9; i++) {
        if (sq[i] == 0) {
            return(0);
        }
        bc = bit_count(sq[i]);
        if (bc > 1 && bc < n) {
            n = bc;
            idx = i;
        }
    }
    if (n == 4) {
        //printlatin(sq);
        return(1);
    }

    num = 0;
    for (i = 0; i < 3; i++) {
        char sq_dup[9];
        int j;
        if ((sq[idx] & (1 << i)) == 0) continue;
        for (j = 0; j < 9; j++) {
            sq_dup[j] = sq[j];
        }
        sq_dup[idx] = 1 << i;
        num += latin(sq_dup);
        if (num >= 2)
            break;
    }
    return(num);
}

int
erase(int n, char sq[])
{
    int a[9];
    int i;

    for (i = 0; i < 9; i++) {
        a[i] = i;
    }

    for (i = 0; i < n; i++) {
        int k = (int)(genrand64_real2() * (9 - i));
        //printf("%d\n", a[k]);
        sq[a[k]] = 0xf;
        a[k] = a[8 - i];
    }
    return (0);
}

double
errprob(int n_erase)
{
    char seed[9] = {
        0x1, 0x2, 0x4,
        0x2, 0x4, 0x1,
        0x4, 0x1, 0x2,
    };
```

```
    char sq[9];
    int count;
    int i;
    int n;

    for (count = 0, n = 0; count < 10000; n++) {
        for (i = 0; i < 9; i++) {
            sq[i] = seed[i];
        }
        erase(n_erase, sq);

        int num = latin(sq);
        if (num == 0) {
            printf("[%d]\n", __LINE__);
            exit(1);
        }
        if (num > 1) {
            count++;
            putchar('.');
        }
    }
    return((double)count / n);
}

int
main()
{
    double err[9];
    int i;

    init_genrand64(10);
    for (i = 0; i < 6; i++) {
        err[i] = 0.0;
    }
    for (i = 6; i < 9; i++) {
        err[i] = errprob(i);
    }

    putchar('\n');
    for (i = 0; i < 9; i++) {
        printf("%g,", err[i]);
    }
    putchar('\n');

    return(0);
}
```

A.2　4 次ラテン方陣の同値類の代表元の復号誤り確率を求めるプログラム

```
#include <stdio.h>
#include <stdlib.h>
#include"MT64.h"

int
bit_count(int x0)
{
    int x1 = (x0 & 0x05) + ((x0 & 0x0a) >> 1);
    int x2 = (x1 & 0x03) + ((x1 & 0x0c) >> 2);
    return(x2);
}

int
deduce(char *sq)
{
    int app;
```

```
    int i, j, s;

    app = 0;
    for (i = 0; i < 16; i++) {
        if (bit_count(sq[i]) != 1) continue;
        s = (i / 4) * 4;
        for (j = s; j < s + 4; j++) {
            if (j == i) continue;
            if (sq[j] & sq[i]) {
                sq[j] &= ~sq[i];
                app++;
            }
        }
        s = i % 4;
        for (j = s; j < s + 16; j += 4) {
            if (j == i) continue;
            if (sq[j] & sq[i]) {
                sq[j] &= ~sq[i];
                app++;
            }
        }
    }
    return(app);
}

void
printlatin(char *sq)
{
    int i;

    for (i = 0; i < 16; i++) {
        printf("%2d", sq[i]);
        if (i % 4 == 3) {
            printf("\n");
        }
    }
    printf("\n");

    return;
}

int
latin(char *sq)
{
    int i, idx, n, num;
    int bc;

    while (deduce(sq) > 0) {
        ;
    }

    n = 5;
    for (i = 0; i < 16; i++) {
        if (sq[i] == 0) {
            return(0);
        }
        bc = bit_count(sq[i]);
        if (bc > 1 && bc < n) {
            n = bc;
            idx = i;
        }
    }
    if (n == 5) {
        //printlatin(sq);
        return(1);
    }

    num = 0;
    for (i = 0; i < 4; i++) {
```

```
        char sq_dup[16];
        int j;
        if ((sq[idx] & (1 << i)) == 0) continue;
        for (j = 0; j < 16; j++) {
            sq_dup[j] = sq[j];
        }
        sq_dup[idx] = 1 << i;
        num += latin(sq_dup);
        if (num >= 2)
            break;
    }
    return(num);
}

int
erase(int n, char sq[])
{
    int a[16];
    int i;

    for (i = 0; i < 16; i++) {
        a[i] = i;
    }

    for (i = 0; i < n; i++) {
        int k = (int)(genrand64_real2() * (16 - i));
        //printf("%d\n", a[k]);
        sq[a[k]] = 0xf;
        a[k] = a[15 - i];
    }
    return (0);
}

double
errprob(int n_erase)
{
    char seed[16] = {
        0x1, 0x2, 0x4, 0x8,
        0x2, 0x1, 0x8, 0x4,
        0x4, 0x8, 0x1, 0x2,
        0x8, 0x4, 0x2, 0x1,
    };

    char sq[16];
    int count;
    int i;
    int n;

    for (count = 0, n = 0; count < 10000; n++) {
        for (i = 0; i < 16; i++) {
            sq[i] = seed[i];
        }
        erase(n_erase, sq);

        int num = latin(sq);
        if (num == 0) {
            printf("[%d]\n", __LINE__);
            exit(1);
        }
        if (num > 1) {
            count++;
            putchar('.');
        }
    }
    return((double)count / n);
}

int
main()
```

```
{
    double err[16];
    int i;

    init_genrand64(10);
    for (i = 0; i < 4; i++) {
        err[i] = 0.0;
    }
    for (i = 4; i < 16; i++) {
        err[i] = errprob(i);
    }

    putchar('\n');
    for (i = 0; i < 16; i++) {
        printf("%g,", err[i]);
    }
    putchar('\n');

    return(0);
}
```

A.3　5 次ラテン方陣の同値類の代表元の復号誤り確率を求めるプログラム

```
#include <stdio.h>
#include <stdlib.h>
#include"MT64.h"

int
bit_count(unsigned int x0)
{
    unsigned int x1 = (x0 & 0x55555555) + ((x0 & 0xaaaaaaaa) >> 1);
    unsigned int x2 = (x1 & 0x33333333) + ((x1 & 0xcccccccc) >> 2);
    unsigned int x3 = (x2 & 0x0f0f0f0f) + ((x2 & 0xf0f0f0f0) >> 4);
    unsigned int x4 = (x3 & 0x00ff00ff) + ((x3 & 0xff00ff00) >> 8);
    unsigned int x5 = (x4 & 0x0000ffff) + ((x4 & 0xffff0000) >> 16);
    return(x5);
}

int
deduce(char *sq)
{
    int app;
    int i, j, s;

    app = 0;
    for (i = 0; i < 25; i++) {
        if (bit_count(sq[i]) != 1) continue;
        s = (i / 5) * 5;
        for (j = s; j < s + 5; j++) {
            if (j == i) continue;
            if (sq[j] & sq[i]) {
                sq[j] &= ~sq[i];
                app++;
            }
        }
        s = i % 5;
        for (j = s; j < s + 25; j += 5) {
            if (j == i) continue;
            if (sq[j] & sq[i]) {
                sq[j] &= ~sq[i];
                app++;
            }
        }
    }
```

```
    return(app);
}

void
printlatin(char *sq)
{
    int i;

    for (i = 0; i < 25; i++) {
        printf("%3d", sq[i]);
        if (i % 5 == 4) {
            printf("\n");
        }
    }
    printf("\n");

    return;
}

int
latin(char *sq)
{
    int i, idx, n, num;
    int bc;

    while (deduce(sq) > 0) {
        ;
    }

    n = 6;
    for (i = 0; i < 25; i++) {
        if (sq[i] == 0) {
            return(0);
        }
        bc = bit_count(sq[i]);
        if (bc > 1 && bc < n) {
            n = bc;
            idx = i;
        }
    }
    if (n == 6) {
        //printlatin(sq);
        return(1);
    }

    num = 0;
    for (i = 0; i < 5; i++) {
        char sq_dup[25];
        int j;
        if ((sq[idx] & (1 << i)) == 0) continue;
        for (j = 0; j < 25; j++) {
            sq_dup[j] = sq[j];
        }
        sq_dup[idx] = 1 << i;
        num += latin(sq_dup);
        if (num >= 2)
            break;
    }
    return(num);
}

int
erase(int n, char sq[])
{
    int a[25];
    int i;

    for (i = 0; i < 25; i++) {
        a[i] = i;
```

```
    }
    for (i = 0; i < n; i++) {
        int k = (int)(genrand64_real2() * (25 - i));
        //printf("%d\n" , a[k]);
        sq[a[k]] = 0x1f;
        a[k] = a[24 - i];
    }
    return(0);
}

double
errprob(int n_erase)
{
    char seed[25] = {
        0x01, 0x02, 0x04, 0x08, 0x10,
        0x02, 0x04, 0x08, 0x10, 0x01,
        0x04, 0x08, 0x10, 0x01, 0x02,
        0x08, 0x10, 0x01, 0x02, 0x04,
        0x10, 0x01, 0x02, 0x04, 0x08,
    };
    char sq[25];
    int count;
    int i;
    int n;

    for (count = 0, n = 0; count < 10000; n++) {
        for (i = 0; i < 25; i++) {
            sq[i] = seed[i];
        }
        erase(n_erase, sq);

        int num = latin(sq);
        if (num == 0) {
            printf("[%d]\n", __LINE__);
            exit(1);
        }
        if (num > 1) {
            count++;
            putchar('.');
        }
    }
    return((double)count / n);
}

int
main()
{
    double err[25];
    int i;

    init_genrand64(10);
    for (i = 0; i < 7; i++) {
        err[i] = 0.0;
    }
    for (i = 7; i < 25; i++) {
        err[i] = errprob(i);
    }

    putchar('\n');
    for (i = 0; i < 25; i++) {
        printf("%g, ", err[i]);
    }
    putchar('\n');

    return(0);
}
```

A.4　6 次ラテン方陣の同値類の代表元の復号誤り確率を求めるプログラム

```
#include<stdio.h>
#include<stdlib.h>
#include"MT64.h"

int
bit_count(unsigned int x0)
{
    unsigned int x1 = (x0 & 0x55555555) + ((x0 & 0xaaaaaaaa) >> 1);
    unsigned int x2 = (x1 & 0x33333333) + ((x1 & 0xcccccccc) >> 2);
    unsigned int x3 = (x2 & 0x0f0f0f0f) + ((x2 & 0xf0f0f0f0) >> 4);
    unsigned int x4 = (x3 & 0x00ff00ff) + ((x3 & 0xff00ff00) >> 8);
    unsigned int x5 = (x4 & 0x0000ffff) + ((x4 & 0xffff0000) >> 16);
    return(x5);
}

int
deduce(char *sq)
{
    int app;
    int i, j, s;

    app = 0;
    for (i = 0; i < 36; i++) {
        if (bit_count(sq[i]) != 1) continue;
        s = (i / 6) * 6;
        for (j = s; j < s + 6; j++) {
            if (j == i) continue;
            if (sq[j] & sq[i]) {
                sq[j] &= ~sq[i];
                app++;
            }
        }
        s = i % 6;
        for (j = s; j < s + 36; j += 6) {
            if (j == i) continue;
            if (sq[j] & sq[i]) {
                sq[j] &= ~sq[i];
                app++;
            }
        }
    }
    return(app);
}

void
printlatin(char *sq)
{
    int i;

    for (i = 0; i < 36; i++) {
        printf("%3d", sq[i]);
        if (i % 6 == 5) {
            printf("\n");
        }
    }
    printf("\n");

    return;
}

int
latin(char *sq)
{
```

```
    int i, idx, n, num;
    int bc;

    while (deduce(sq) > 0) {
        ;
    }

    n = 7;
    for (i = 0; i < 36; i++) {
        if (sq[i] == 0) {
            return(0);
        }
        bc = bit_count(sq[i]);
        if (bc > 1 && bc < n) {
            n = bc;
            idx = i;
        }
    }
    if (n == 7) {
        //printlatin(sq);
        return(1);
    }

    num = 0;
    for (i = 0; i < 6; i++) {
        char sq_dup[36];
        int j;
        if ((sq[idx] & (1 << i)) == 0) continue;
        for (j = 0; j < 36; j++) {
            sq_dup[j] = sq[j];
        }
        sq_dup[idx] = 1 << i;
        num += latin(sq_dup);
        if (num >= 2)
            break;
    }
    return(num);
}

int
erase(int n, char sq[])
{
    int a[36];
    int i;

    for (i = 0; i < 36; i++) {
        a[i] = i;
    }
    for (i = 0; i < n; i++) {
        int k = (int)(genrand64_real2() * (36 - i));
        //printf("%d\n", a[k]);
        sq[a[k]] = 0x3f;
        a[k] = a[35 - i];
    }
    return(0);
}

double
errprob(int n_erase)
{
    char seed[36] = {
        0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
        0x02, 0x04, 0x01, 0x20, 0x08, 0x10,
        0x04, 0x01, 0x02, 0x10, 0x20, 0x08,
        0x08, 0x10, 0x20, 0x01, 0x02, 0x04,
        0x10, 0x20, 0x08, 0x04, 0x01, 0x02,
        0x20, 0x08, 0x10, 0x02, 0x04, 0x01,
    };
    char sq[36];
```

```
    int count;
    int i;
    int n;

    for (count = 0, n = 0; count < 10000; n++) {
        for (i = 0; i < 36; i++) {
            sq[i] = seed[i];
        }
        erase(n_erase, sq);

        int num = latin(sq);
        if (num == 0) {
            printf("[%d]\n", __LINE__);
            exit(1);
        }
        if (num > 1) {
            count++;
            putchar('.');
        }
    }
    return((double)count / n);
}

int
main()
{
    double err[36];
    int i;

    init_genrand64(10);
    for (i = 0; i < 4; i++) {
        err[i] = 0.0;
    }
    for (i = 4; i < 36; i++) {
        err[i] = errprob(i);
    }

    putchar('\n');
    for (i = 0; i < 36; i++) {
        printf("%g, ", err[i]);
    }
    putchar('\n');

    return(0);
}
```

A.5 Mersenne Twister

```
#pragma once/*
   A C-program for MT19937-64 (2004/9/29 version).
   Coded by Takuji Nishimura and Makoto Matsumoto.

   This is a 64-bit version of Mersenne Twister pseudorandom number
   generator.

   Before using, initialize the state by using init_genrand64(seed)
   or init_by_array64(init_key, key_length).

   Copyright (C) 2004, Makoto Matsumoto and Takuji Nishimura,
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions
   are met:
```

```
     1. Redistributions of source code must retain the above copyright
        notice, this list of conditions and the following disclaimer.

     2. Redistributions in binary form must reproduce the above copyright
        notice, this list of conditions and the following disclaimer in the
        documentation and/or other materials provided with the distribution.

     3. The names of its contributors may not be used to endorse or promote
        products derived from this software without specific prior written
        permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
   "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
   LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
   A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
   CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
   PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
   LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
   NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
   SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

   References:
   T. Nishimura, ``Tables of 64-bit Mersenne Twisters''
     ACM Transactions on Modeling and
     Computer Simulation 10. (2000) 348--357.
   M. Matsumoto and T. Nishimura,
     ``Mersenne Twister: a 623-dimensionally equidistributed
       uniform pseudorandom number generator''
     ACM Transactions on Modeling and
     Computer Simulation 8. (Jan. 1998) 3--30.

   Any feedback is very welcome.
   http://www.math.hiroshima-u.ac.jp/~m-mat/MT/emt.html
   email: m-mat @ math.sci.hiroshima-u.ac.jp (remove spaces)
*/



#define NN 312
#define MM 156
#define MATRIX_A 0xB5026F5AA96619E9ULL
#define UM 0xFFFFFFFF80000000ULL /* Most significant 33 bits */
#define LM 0x7FFFFFFFULL /* Least significant 31 bits */


/* The array for the state vector */
static unsigned long long mt[NN];
/* mti==NN+1 means mt[NN] is not initialized */
static int mti = NN + 1;

/* initializes mt[NN] with a seed */
void init_genrand64(unsigned long long seed)
{
    mt[0] = seed;
    for (mti = 1; mti<NN; mti++)
        mt[mti] = (6364136223846793005ULL * (mt[mti - 1] ^ (mt[mti - 1] >> 62)) + mti);
}

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
void init_by_array64(unsigned long long init_key[],
    unsigned long long key_length)
{
    unsigned long long i, j, k;
    init_genrand64(19650218ULL);
    i = 1; j = 0;
```

```
    k = (NN>key_length ? NN : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i - 1] ^ (mt[i - 1] >> 62)) * 3935559000370003845ULL))
            + init_key[j] + j; /* non linear */
        i++; j++;
        if (i >= NN) { mt[0] = mt[NN - 1]; i = 1; }
        if (j >= key_length) j = 0;
    }
    for (k = NN - 1; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i - 1] ^ (mt[i - 1] >> 62)) * 2862933555777941757ULL))
            - i; /* non linear */
        i++;
        if (i >= NN) { mt[0] = mt[NN - 1]; i = 1; }
    }

    mt[0] = 1ULL << 63; /* MSB is 1; assuring non-zero initial array */
}

/* generates a random number on [0, 2^64-1]-interval */
unsigned long long genrand64_int64(void)
{
    int i;
    unsigned long long x;
    static unsigned long long mag01[2] = { 0ULL, MATRIX_A };

    if (mti >= NN) { /* generate NN words at one time */

                     /* if init_genrand64() has not been called, */
                     /* a default initial seed is used     */
        if (mti == NN + 1)
            init_genrand64(5489ULL);

        for (i = 0; i<NN - MM; i++) {
            x = (mt[i] & UM) | (mt[i + 1] & LM);
            mt[i] = mt[i + MM] ^ (x >> 1) ^ mag01[(int)(x & 1ULL)];
        }
        for (; i<NN - 1; i++) {
            x = (mt[i] & UM) | (mt[i + 1] & LM);
            mt[i] = mt[i + (MM - NN)] ^ (x >> 1) ^ mag01[(int)(x & 1ULL)];
        }
        x = (mt[NN - 1] & UM) | (mt[0] & LM);
        mt[NN - 1] = mt[MM - 1] ^ (x >> 1) ^ mag01[(int)(x & 1ULL)];

        mti = 0;
    }

    x = mt[mti++];

    x ^= (x >> 29) & 0x5555555555555555ULL;
    x ^= (x << 17) & 0x71D67FFFEDA60000ULL;
    x ^= (x << 37) & 0xFFF7EEE000000000ULL;
    x ^= (x >> 43);

    return x;
}

/* generates a random number on [0, 2^63-1]-interval */
long long genrand64_int63(void)
{
    return (long long)(genrand64_int64() >> 1);
}

/* generates a random number on [0,1]-real-interval */
double genrand64_real1(void)
{
    return (genrand64_int64() >> 11) * (1.0 / 9007199254740991.0);
}

/* generates a random number on [0,1)-real-interval */
```

```
double genrand64_real2(void)
{
    return (genrand64_int64() >> 11) * (1.0 / 9007199254740992.0);
}

/* generates a random number on (0,1)-real-interval */
double genrand64_real3(void)
{
    return ((genrand64_int64() >> 12) + 0.5) * (1.0 / 4503599627370496.0);
}
```