信州大学
大学院総合理工学研究科

修士論文

球充填に基づく位相符号化変調の性能評価について

指導教員　西新 幹彦 准教授

専攻　工学専攻
分野　情報数理・融合システム分野
学籍番号　24W6068C
氏名　堀 達裕

2026 年 02 月 09 日

目次

1　はじめに

現代社会において通信技術はさまざまな場面で使用されており, その発達は社会基盤を支えている. なかでも, 情報の確実な伝達を実現するために, 通信の安定性の向上は重要となっている. 通信の安定性に対して悪影響を及ぼすものの一つに, 雑音がある. 通信環境には雑音が存在するため, この雑音への耐性が通信技術の性能を評価する一つの指標となる. この性能を測るために, 情報を正しく伝送することの出来る符号化レートと雑音の関係を理論的にあらわしたのが, 通信路符号化問題である. この問題では, 送信者と受信者の間に通信路が存在し, 符号器で符号化した情報を通信路で送っている. 受信者は, 通信路内にて雑音の影響を受けた符号を受け取り, それを復号器で誤り訂正することで一連の通信の流れを表している. このように, 通信の安定性向上を目指し, 通信路符号化問題の理論を応用した変調方式や符号化方式の改良に関する研究が行われている.

代表的な変調方式のひとつに位相変調がある. 位相変調とは, 送信したい情報を信号の位相で表現する変調方式であり, 0 や 1 といったデジタルデータを, 0 度や 180 度のように特定の位相で表現できるため, デジタル通信で広く使われている. さらに, 変調に符号化を組み合わせた方式として, 符号化変調が存在する. 本研究では位相変調による送信から受信に至る過程を通信路とみなし, この通信路に対して符号を設計するという立場から符号化変調を考える. このようにして, 符号化変調の設計を通信路符号化問題として定式化する.

本研究ではまず, 位相変調に対するランダム符号の性能を評価した. その結果, 符号の符号語長が大きくなると雑音耐性が良くなることが確認できた. しかし, QPSK などの既存の位相変調と比較すると, ランダム符号の方が性能が劣ることが明らかになった. この結果を受け, 次の検討として, 球充填問題に基づいて符号を設計し, 性能を評価した. 球充填問題とは, 空間内にできるだけ多くの球を互いに重ならないように配置する方法を考える数学的問題である. この球を符号語に置き換えることで, 符号語同士の距離を最大化する配置になり, 雑音による復号誤りを低減できると推測した. そして実際に, この球充填に基づく符号を導入した位相変調と, 既存の位相変調やランダム符号の性能比較を行った結果, 誤り率の低減を確認した.

2　位相変調の通信路容量

通信路のための符号を実験的に評価する際には, 対象となる通信路の容量を知る必要がある. 本研究では位相変調を通信路とみなすので, 符号の評価の際には位相変調の通信路容量が必要となる. 位相変調は連続通信路の一種であるため, まず一般に連続通信路の通信路容量を確認し, 次にそこに位相変調の特殊性を付け加えて, 本研究で対象とする位相変調の通信路容量を導出する.

2.1 連続通信路の通信路容量

連続アルファベットを持つ定常無記憶通信路に対する通信路容量を定義する. 入力アルファベットを $\mathcal{X}$, 出力アルファベットを $\mathcal{Y}$ とし, それぞれの要素を表す確率変数を X と Y とする. この X を入力, Y を出力とする通信路は, 条件付き確率

$$W(y|x) \triangleq \Pr\{\, Y = y \,|\, X = x \,\} \tag{1}$$

によって表される. この入力系列 $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$ が与えられたとき出力系列 $\boldsymbol{y} = (y_1, y_2, \cdots, y_n)$ の出現する条件付き確率は,

$$W^n(\boldsymbol{y}|\boldsymbol{x}) = \prod_{i=1}^{n} W(y_i|x_i) \tag{2}$$

と表される.

送信者が送る M_n 個のメッセージの集合を $\mathcal{M}_n = \{\, 1, 2, \cdots, M_n \,\}$ としたとき, 符号器を $\varphi_n : \mathcal{M}_n \to \mathcal{X}^n$ とし, 復号器を $\psi_n : \mathcal{Y}^n \to \mathcal{M}_n$ とする. ここで, $\varphi_n(m)$ を $m \in \mathcal{M}_n$ の符号語という. この符号の符号化レート R は,

$$R \triangleq \frac{1}{n} \log M_n \tag{3}$$

と定義される. ただし, log の底は 2 である. また, 復号誤り確率は,

$$\varepsilon_n \triangleq \frac{1}{M_n} \sum_{m \in \mathcal{M}_n} W^n(\psi_n^{-1}(m)^c | \varphi_n(m)) \tag{4}$$

と定義する. ここで, $\psi_n^{-1}(m)^c$ は復号領域 $\psi_n^{-1}(m)$ の補集合を表しており, メッセージ m が誤って復号される領域を意味する.

符号化レート R が達成可能とは,

$$\lim_{n \to \infty} \varepsilon_n = 0 \tag{5}$$

$$\liminf_{n \to \infty} \frac{1}{n} \log M_n \geq R \tag{6}$$

を満たす, 符号 (φ_n,ψ_n) が存在することである. 通信路容量 $C(W)$ を,

$$C(W) \triangleq \sup\{R \,|\, R \text{ が達成可能 }\} \tag{7}$$

と定義する.

通信路容量の数学的表現が知られている [1]. 準備のため, 確率密度関数 $f_X(x)$ をもつ確率変数 X の微分エントロピー $h(X)$ を,

$$h(X) \triangleq -\int_x f_X(x) \log f_X(x) dx \tag{8}$$

と定義する．また, 確率変数 X, Y が同時確率密度関数 $f_{XY}(x,y)$ をもつとき, 条件付き微分エントロピー $h(Y|X)$ を,

$$h(Y|X) \;=\; -\iint_{x,y} f_{XY}(x,y)\log\, f_{Y|X}(y|x)dxdy \tag{9}$$

と定義する．さらに, X と Y の間の相互情報量 $I(X;Y)$ を,

$$I(X;Y) \;\triangleq\; \iint_{x,y} f_{XY}(x,y)\log\frac{f_{XY}(x,y)}{f_X(x)f_Y(y)}dxdy \tag{10}$$

と定義する．この定義より,

$$I(X;Y) \;=\; h(Y) \;-\; h(Y|X) \tag{11}$$

が成り立つ.

以上の定義のもとで, 通信路容量は, 通信路符号化定理により,

$$C(W) \;=\; \max_X I(X;Y) \tag{12}$$

で与えられる [2]．これが一般的な連続通信路の通信路容量である．

2.2 位相変調における雑音

連続通信路に対して位相変調の特徴を加味して通信路容量を導く．まず, 通信路の入出力アルファベットは実数全体ではなく, $-\pi$ から π までの連続値である．そして, 図 1 のように, 通信路の入力 X に対してそれとは独立な雑音 N が加算されると考える．位相変調の通信路の周期性を表すため, 入力 X と雑音 N の加算は 2π 周期の剰余演算で,

$$X \oplus N \;\triangleq\; \mathrm{mod}(X+N+\pi,\; 2\pi) - \pi \tag{13}$$

とする．ここで, $\mathrm{mod}(a, 2\pi)$ は, a を 2π で割った余りを $[0, 2\pi)$ に写像する関数であり, さらに $-\pi$ することで, 結果を $[-\pi, \pi)$ に対応付けている．したがって出力は $Y = X \oplus N$ と表され, 式 (12) は,

$$C(W) \;=\; \max_X(h(Y) \;-\; h(X \oplus N|X)) \tag{14}$$

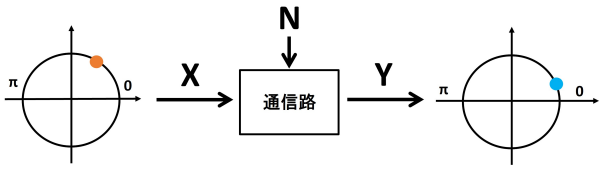


図 1: 通信路モデル

となる．さらに, X と N が独立であることから通信路容量は

$$C(W) \;=\; \max_{X}(h(Y)) - h(N) \tag{15}$$

と表される．したがって通信路容量を求めるには $h(Y)$ の最大値と $h(N)$ を求めればよい．

位相変調の $h(Y)$ は, 雑音の種類に依らず, 入力位相が $[-\pi, \pi)$ 上で一様分布となるときに最大となり, その最大値は,

$$h(Y) \;=\; \log 2\pi \tag{16}$$

で与えられる [4].

次に $h(N)$ を求める．本研究では正規分布に基づく雑音を考える．正規分布は,

$$f_N(x) \;=\; \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \;,\; -\infty < x < \infty \tag{17}$$

という密度関数を持つ確率分布である [3]．この分布の期待値は μ, 分散は σ^2 であり, $\sigma > 0$ である．本研究で扱う雑音は, $\mu = 0$ とする．なお, 正規分布の定義域は実数全体であるのに対し, 本研究の通信路の定義域は $-\pi \le x \le \pi$ である．そこで, その範囲外の部分は 0 として切り捨て, その後で正規化を行なう [4]．このときの, 密度関数は

$$\bar{f}_N(x) \;=\; \frac{f_N(x)}{\int_{-\pi}^{\pi} f_N(x)dx} \tag{18}$$

$$\;=\; \frac{e^{-\frac{x^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma \mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)} \;,\; -\pi \le x \le \pi \tag{19}$$

となる．この密度関数のグラフを図 2 に示す．$\sigma = 0.1$ のときのグラフを赤色, $\sigma = 0.5$ のときのグラフを青色で表示して σ の値による変化を示している．

本研究ではこの密度関数に従う雑音を考える．雑音の微分エントロピー $h(N)$ は,

$$h(N) \;=\; -\frac{\sqrt{2\pi}\,\mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)\sigma \ln\left(\frac{e^{-\frac{x^2}{2\sigma^2}}}{\mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)\sigma}\right) + \pi e^{-\frac{x^2}{2\sigma^2}} - \frac{\sqrt{\pi}\,\mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)\sigma}{\sqrt{2}} + \frac{\pi^{\frac{5}{2}}\mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)}{\sqrt{2}\sigma}}{\sqrt{2\pi}\ln(2)\,\mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)} \tag{20}$$

となる．これより, 本研究の通信路容量は,

$$C(W) \;=\; \log 2\pi \;+\; \frac{\sqrt{2\pi}\,\mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)\sigma \ln\left(\frac{e^{-\frac{x^2}{2\sigma^2}}}{\mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)\sigma}\right) + \pi e^{-\frac{x^2}{2\sigma^2}} - \frac{\sqrt{\pi}\,\mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)\sigma}{\sqrt{2}} + \frac{\pi^{\frac{5}{2}}\mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)}{\sqrt{2}\sigma}}{\sqrt{2\pi}\ln(2)\,\mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)} \tag{21}$$

となる．

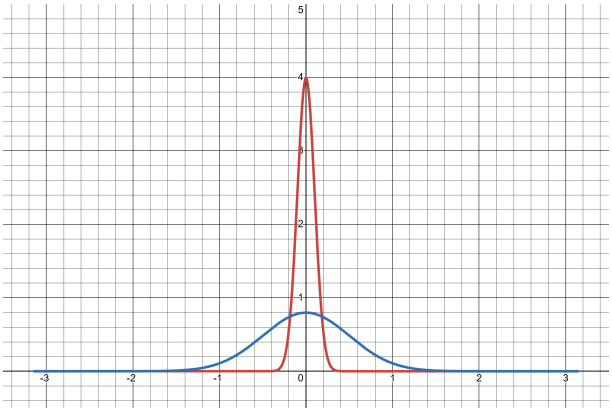


図 2: 切り捨てた正規分布

3　QPSK とランダム符号

本研究の主題は球充填に基づく符号の性能評価であるが, この章では, 比較対象の QPSK とランダム符号を説明する.

3.1　QPSK

QPSK は, 1 シンボルあたり 2 ビットを伝送できる単純かつ広く用いられている位相変調である. 図 3 のように, QPSK の符号器では, 送信する信号を $\frac{\pi}{4}$, $\frac{3\pi}{4}$, $-\frac{\pi}{4}$, $-\frac{3\pi}{4}$ の 4 つの位相から等確率で選ぶ. 送信された信号は, 通信路で雑音が加算された状態で復号器が受け取る. 復号器では, 4 つの位相に対応する範囲が割り振られており, 受信信号がどこの範囲に含まれているかを判断して誤り訂正を行う. これを n 回行うことにより, ブロック長が n の QPSK が実現される.

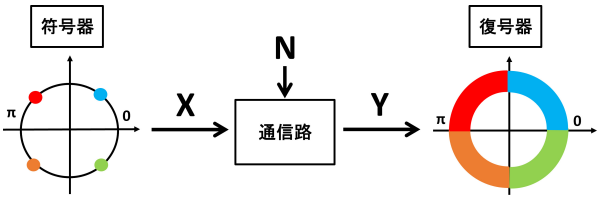


図 3: QPSK の通信路イメージ

3.2 ランダム符号

ランダム符号は, 本研究の初期段階では, 位相変調方式よりも高い通信性能を実現できる可能性がある符号化方式として検討していた. しかし検証の末, QPSK よりも性能が劣ることが分かったため, あくまで比較対象の一つとして扱うこととした.

図 4 のように, ランダム符号の符号器では, 符号語の各シンボルを, $-\pi$ から π の一様分布にしたがって生成する. この操作を n 回行うことで, 符号語長 n の符号語とする. さらにその符号語を M_n 個生成することで, 符号語リストを構成する.

このリストの中から, 送信する符号語を等確率で選択する. 受信側は, 雑音が加わった符号語を受け取り, あらかじめ送信側と共有している符号語リストを参照して復号を行う. その際の復号器は, 受信語 $\boldsymbol{y}$ が得られたもとでの各符号語 $\boldsymbol{x}_i$ の条件付き確率 $P_{Y|X}(\boldsymbol{y}|\boldsymbol{x}_i)$ を計算し, その確率が最大になるものを送信された符号語と推定する. この復号方法は最尤復号といい, その誤り訂正は, 受信語と各符号語との距離の比較に帰着できる.

3.3 最尤復号のための距離

ここでは, ランダム符号の最尤復号で用いる距離の計算方法を説明する. 最尤復号とは, 受信語 $\boldsymbol{y}$ が得られたとき, 各符号語 $\boldsymbol{x}_i$ の集合 $C = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{M_n}\}$ の中から,

$$\hat{\boldsymbol{x}} = \arg\max_{\boldsymbol{x}_i \in C} P_{Y|X}(\boldsymbol{y}|\boldsymbol{x}_i) \tag{22}$$

として, 推定語 $\hat{\boldsymbol{x}}$ を定める復号法である [5]. このように, 尤度を最大化する符号語 $\hat{\boldsymbol{x}}$ が, 送信された符号語として推定される. この確率が最大になるのは, 受信語と符号語の距離が最小になるときである. 特に, 雑音がガウス分布に従う場合は, ユークリッド距離で計算を行うと最小距離になる [1].

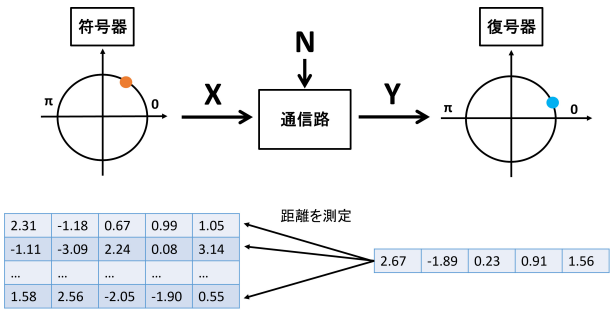


図 4: ランダム符号の通信路イメージ

通常のユークリッド距離では, 二点間の差を各成分ごとに計算し, その二乗和の平方根をとることで距離を求める. しかし, 本研究の通信路は各成分が $[-\pi, \pi)$ で周期的に繰り返すトーラス空間上にあるため, 単純な差を用いるだけでは位相の折り返しを考慮できない. そこで, 円周上の差を

$$Y \ominus X \triangleq \mathrm{mod}(Y - X + \pi,\ 2\pi) - \pi \tag{23}$$

と定義する. この差を本論文では, 位相差と呼ぶ. この定義により, 受信語と符号語の位相が π 以上離れている場合は, 差が負の値になり, 位相差は常に $[-\pi, \pi)$ の範囲に収まる. すなわち, 受信語に対して最も近い方向の位相差を表すことになる. よって, $\boldsymbol{y}$ と $\boldsymbol{x}_i$ の各シンボルにおける位相差は,

$$y_j \ominus x_{i,j} \ = \ \mathrm{mod}(y_j - x_{i,j} + \pi,\ 2\pi) - \pi, \quad j = 1, 2, \ldots, n \tag{24}$$

となる. この位相差を用いると, $\boldsymbol{y}$ と $\boldsymbol{x}_i$ の距離は,

$$\sqrt{\sum_{j=1}^{n} (y_j \ominus x_{i,j})^2} \tag{25}$$

となる. この位相差を用いた距離が最小距離になることの証明は付録 A にて示す. このようにして, 受信語と全ての符号語との距離を計算して比較することで, 誤り訂正を行う.

4 球充填に基づく符号

球充填問題とは, ユークリッド空間内において, 互いに重ならない球をできるだけ密に配置する方法を求める数学的問題である. 球充填問題は, 古くは三次元空間でのケプラー予想として知られ, 長年にわたり数学や物理などの分野で広く研究されてきた [6]. 特に, 二次元から八次元までの空間における最密充填構造はすでに数学的に明らかにされており, それぞれの次元で特徴的な配置が存在する. これらの各次元における最密充填構造は, 格子点が周期的に並ぶ格子構造によって表現される.

情報理論においては, この球の座標を符号語と捉えることができる. そうすることで, 球が重ならないということは, 復号誤りが起きにくいということになり, なおかつ, 空間内に球をできるだけ密に配置することは, 通信効率を上げることにつながる [5]. また, 球が存在する空間の次元数は, 符号語長と対応しており, n 次元空間の球の座標は符号語長 n の符号語に対応する. ただし, 球充填問題はユークリッド空間で定義される一方で, 本研究の通信路は n 次元のトーラス空間で表されるため, ユークリッド空間の結果を直接的に適用することはできない. そこで, 格子構造において, 全体を平行移動により再構成できる最小の領域として, 基本領域を導入する. 本研究では, もとのユークリッド空間からこの基本領域を取り出し, 各軸方向の大きさが

2π の整数分割となるようにスケーリングを行う. そして, 調整した基本領域を周期的に配置することで, 一辺の長さが 2π の n 次元トーラス空間に並べる.

次節より, 二次元から八次元までの各次元に対し, 数学的な最密球充填構造と符号への適用について説明する.

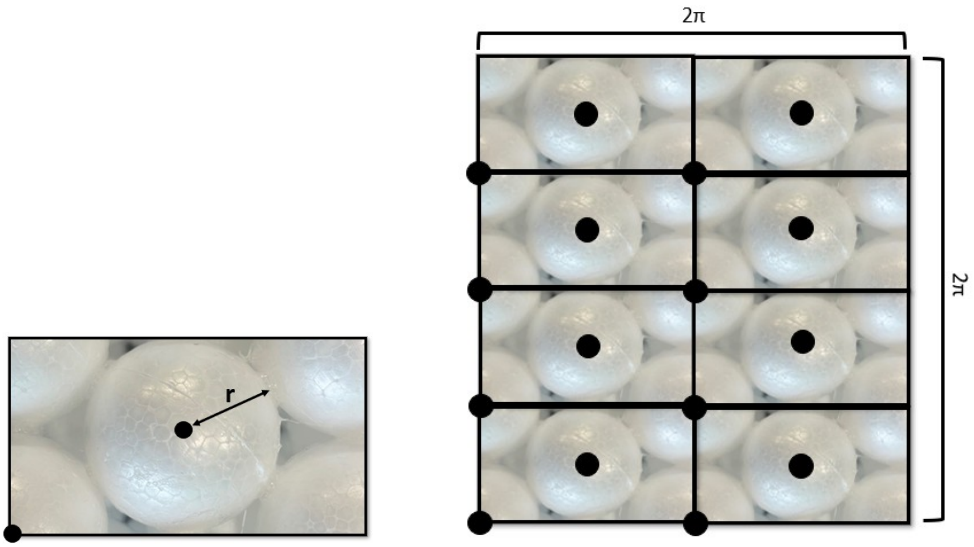
4.1 二次元

二次元ユークリッド空間における最密充填は, 六方格子である. 図 5 のように, 1 つの円のまわりに 6 つの円が接する構造をしている.

図 6 の (a) に示すように, この基本領域の中には, 中心部分に 1 つと, その周囲の四分円が 4 つ分で, 2 つの円が含まれている. 円の半径を r とすると, 基本領域の大きさは, 横が $2\sqrt{3}r$, 縦が $2r$ となる. これが縦と横に繰り返されることで, 図 5 のような六方格子の構造をしている.

この基本領域を, 二次元トーラス空間に適用する. $r = \frac{\pi}{4}$ とすると, 縦の方向には 2π の範囲に 4 つ配置することができる. 一方横は, 2 つだと 2π に満たない. そこで, 横方向については基本領域の横の長さを π に拡大することで, 2π の範囲に 2 つ並べる. この方法で, 図 6 の (b) のように, 縦に 4 つ, 横に 2 つの基本領域を並べることで, 計 16 個の符号語を配置した. これは, n が 2 の QPSK と同じ符号語数である. これ以降の次元でも同様にして, 基本領域の各軸方向の大きさを確認し, 2π の縮尺に合わせた充填構造に調整する.

図 5: 六方格子



(a) 六方格子の基本領域　　(b) 基本領域の並べ方

図 6: 二次元のスケーリング

4.2 三次元

三次元ユークリッド空間では, 面心立方格子が最密充填となる. 図 7 のように, 六方格子に置かれた球のくぼみに球を配置する構造をしている. 球の半径を r とすると, 基本領域の大きさはそれぞれ, 横が $2\sqrt{3}r$, 縦が $2r$, 高さが $\frac{4\sqrt{6}}{3}r$ である.

この基本領域を, 三次元トーラス空間に適用する. 二次元のときの基本領域を利用して縦に 4 つ, 横に 2 つ並べる. 高さは, 横の大きさと同様に 2π を 2 等分して π に拡大する. これを 2 つ積み上げることで, 三次元空間に計 64 個の符号語を配置した. なお, これは n が 3 の QPSK の符号語数 $4^3 = 64$ と同じである.

4.3 四次元と五次元

以降の次元では, 視覚化して配置の様子を示すことが出来ないので, 数学的定義を示す. 四次元ユークリッド空間と五次元ユークリッド空間の最密充填は, D_n 格子である. D_n 格子は,

$$D_n \triangleq \{(x_1, ..., x_n) \in \mathbb{Z}^n : x_1 + ... + x_n \ \text{ even } \} \tag{26}$$

と定義される [7]. $\mathbb{Z}^n$ は整数格子であり, 座標の各成分が整数である点の集合を表す. つまり, D_n 格子は, この整数格子の中から座標の和が偶数となる点で構成される. また, 整数ベクトル

図 7: 面心立方格子

で表わされた D_n 格子は, 各成分を $\frac{2\pi}{T\times k}$ 倍して, $[-\pi, \pi)$ の座標に変換する. この T は, D_n 格子の基本領域の大きさで, k はその倍数である.

D_n 格子の基本領域の大きさは各軸方向で 2 である. この説明は付録 B に示す. 原点を中心とした基本領域を考えると, 1 つの領域のなかには 0 と 1 の 2 つの整数が含まれる. なお, -1 も範囲に含まれるが, 後にトーラス空間へ適用する際, -1 と 1 は同一点として扱われるため, ここではあらかじめ除外している. したがって, D_n 格子の基本領域は, $\frac{2^n}{2} = 2^{n-1}$ 個の格子点をもつ. さらに, 基本領域の大きさを 2 倍にすると, D_4 格子は, $\frac{(2\times2)^4}{2} = 128$ 個, D_5 格子は, $\frac{(2\times2)^5}{2} = 512$ 個の格子点が存在する. 一方, QPSK の n が 4 と 5 のときの符号語数はそれぞれ, $4^4 = 256$ 個と $4^5 = 1024$ 個であり, どちらも D_n 格子点の数と一致しない. これらの数が合わないと, 同一の符号化レートでの比較ができないため, 両者の符号語数を揃えるための工夫が求められる.

この問題に対処するために, QPSK の符号語数を調整する. QPSK の n 番目のシンボルのみを BPSK にすることで, 符号語数が $4^{n-1}\times2$ 個になる. 具体的に, $n = 4$ では, $4^3\times2 = 128$ 個であり, 2 倍の基本領域の D_4 格子の格子点の数と同じになる. $n = 5$ でも同様に, $4^4\times2 = 512$ 個で, 2 倍の基本領域の D_5 格子と同じになる. このように, 符号語数が合わない場合は, QPSK の代わりに QPSK と BPSK を併用した符号を比較対象にすることで対処することもできる.

4.4 八次元

六次元, 七次元の充填構造の説明に先立ち, 八次元ユークリッド空間の充填構造の E_8 格子について述べる. その理由は, 六次元および七次元の最密充填は, E_8 格子の部分格子として定義

されるためである．そのため，この後の節では，E_8 格子をもとにして六次元と七次元の充填構造を示す．

八次元ユークリッド空間では，E_8 格子が最密な充填構造である．E_8 格子は，

$$E_8 \triangleq \{(x_1, \ldots, x_8) : \text{ all } x_i \in \mathbb{Z} \text{ or all } x_i \in \mathbb{Z} + \frac{1}{2}, \sum x_i \equiv 0 \pmod 2\} \tag{27}$$

と定義される [7]．この E_8 格子は，座標の各成分がすべて整数か，すべて半整数である点の集合のなかで，その和が偶数となる点で構成される．そして，D_n 格子と同様に，整数ベクトルで表わされた各成分を $\frac{2\pi}{T \times k}$ 倍して，$[-\pi, \pi)$ の座標に変換する．

この基本領域の大きさは各軸で 2 である．この説明は付録 B に示す．原点を中心とした領域内には，座標成分の和が偶数となる点は $2^{8-1} = 128$ 個存在する．また，各成分がすべて半整数で，その和が偶数となる点は同じく 128 個存在する．したがって，領域の中には合計 256 個の E_8 格子点が存在する．一方，八次元の QPSK の符号語数は $4^8 = 65536$ 個である．これに合わせるために，E_8 格子の基本領域の大きさを 2 倍にすることで，条件を満たす整数格子と半整数格子はそれぞれ，$\frac{4^8}{2} \times 2 = 32768$ 個になり，合計して E_8 格子は 65536 個になる．

4.5 七次元

七次元ユークリッド空間では，E_7 格子が最密充填である．E_7 格子は，

$$E_7 \triangleq \{(x_1, \ldots, x_8) \in E_8 : x_1 + \cdots + x_8 = 0\} \tag{28}$$

と定義される [7]．E_8 格子のなかで，その成分の和が 0 になる点で構成される．ただし，この状態だと八次元成分で表されているため，E_7 格子の条件を満たす正規直交基底ベクトルを用いて七次元成分に変換する．そのためにまず，7 本の正規直交基底ベクトル

$$\boldsymbol{e}_1 = \frac{1}{\sqrt{2}}\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}\right) \tag{29}$$

$$\boldsymbol{e}_2 = \frac{1}{\sqrt{2}}\left(\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}\right) \tag{30}$$

$$\boldsymbol{e}_3 = \frac{1}{\sqrt{2}}\left(\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}\right) \tag{31}$$

$$\boldsymbol{e}_4 = \frac{1}{\sqrt{2}}\left(\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, -\frac{1}{2}\right) \tag{32}$$

$$\boldsymbol{e}_5 = \frac{1}{\sqrt{2}}\left(\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, \frac{1}{2}\right) \tag{33}$$

$$\boldsymbol{e}_6 = \frac{1}{\sqrt{2}}\left(\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right) \tag{34}$$

$$\boldsymbol{e}_7 = \frac{1}{\sqrt{2}}\left(\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}\right) \tag{35}$$

を導入する．八次元成分で表された各 E_7 格子は，これらの基底ベクトルを用いて，

$$\boldsymbol{x}^{(8)} = \sum_{j=1}^{7} c_j \boldsymbol{e}_j \tag{36}$$

と表すことができる．これにより，七次元成分で

$$E_7 = \{(c_1, \ldots, c_7) : c_i = \langle \boldsymbol{x}, \boldsymbol{e}_i \rangle \, (i = 1, \ldots, 7)\} \tag{37}$$

と表せる．

このときの基本領域の大きさは，各軸の方向に $\sqrt{2}$ であり，これまでの次元と同様に，原点を中心とする領域を取る．基本領域の大きさの説明は付録 B に示す．そして，整数ベクトルを

$$c_i = \frac{2\pi}{\sqrt{2}k} \langle \boldsymbol{x}, \boldsymbol{e}_i \rangle \, (i = 1, \ldots, 7) \tag{38}$$

とし，さらに，$\mathrm{mod}(c_i + \pi,\ 2\pi) - \pi$ とすることで，$[-\pi, \pi)$ の範囲に収める．

これらの定義に基づいた符号語をプログラムにて生成したところ，七次元の符号語数は，2 倍の基本領域で 1016 個であり，3 倍で 17468 個であった．なお，符号語生成の詳細は，付録 C にてコードを示す．

4.6 六次元

六次元ユークリッド空間では，E_6 格子が最密充填である．E_6 格子は，

$$E_6 \triangleq \{(x_1, \ldots, x_8) \in E_8 : x_6 = x_7 = x_8\} \tag{39}$$

と定義される [7]．E_8 格子のなかで，6 番目と 7 番目，そして 8 番目の成分が同じになる点で構成される．ただし，この状態だと八次元成分で表されているので，E_6 格子の条件を満たす正規直交基底ベクトルを用いて六次元成分に変換する．そのために，6 本の正規直交基底ベクトル

$$\boldsymbol{e}_1 = \frac{1}{2\sqrt{3}} (0, 0, 0, 0, 0, 2, 2, 2) \tag{40}$$

$$\boldsymbol{e}_2 = \frac{1}{2} (2, 0, 0, 0, 0, 0, 0, 0) \tag{41}$$

$$\boldsymbol{e}_3 = \frac{1}{2} (0, 2, 0, 0, 0, 0, 0, 0) \tag{42}$$

$$\boldsymbol{e}_4 = \frac{1}{2} (0, 0, 2, 0, 0, 0, 0, 0) \tag{43}$$

$$\boldsymbol{e}_5 = \frac{1}{2} (0, 0, 0, 2, 0, 0, 0, 0) \tag{44}$$

$$\boldsymbol{e}_6 = \frac{1}{2} (0, 0, 0, 0, 2, 0, 0, 0) \tag{45}$$

を導入する．八次元成分で表された各 E_6 格子は，これらの基底ベクトルを用いて，

$$\boldsymbol{x}^{(8)} = \sum_{j=1}^{6} c_j \boldsymbol{e}_j \tag{46}$$

と表すことができる．これにより，六次元成分で

$$E_6 = \{(c_1, \ldots, c_6) : c_i = \langle \boldsymbol{x}, \boldsymbol{e}_i \rangle \, (i = 1, \ldots, 6)\} \tag{47}$$

と表せる．

この基本領域の大きさは，$\boldsymbol{e}_1$ の方向に $2\sqrt{3}$ で，$\boldsymbol{e}_2$ から $\boldsymbol{e}_6$ の方向は 2 である．この大きさの説明は付録 B に示す．原点を中心とする基本領域を取り，その整数ベクトルを

$$c_1 = \frac{2\pi}{2\sqrt{3}k} \langle \boldsymbol{x}, \boldsymbol{e}_1 \rangle \tag{48}$$

$$c_i = \frac{2\pi}{2k} \langle \boldsymbol{x}, \boldsymbol{e}_i \rangle \, (i = 2, \ldots, 6) \tag{49}$$

とする．そして，$\mathrm{mod}(c_i + \pi,\ 2\pi) - \pi$ とすることで，$[-\pi, \pi)$ の範囲に収める．

これらの定義に基づいた符号語をプログラムにて生成したところ，符号語数は，2 倍の基本領域で 2048 個であり，3 倍で 13107 個であった．なお，符号語生成の詳細は，付録 C にてコードを示す．

5 結果比較

本章では，シミュレーションにて計測した，球充填に基づく符号の誤り確率を示す．その際に用いたコードは付録 C にて示す．そして，QPSK やランダム符号でも同様のシミュレーションを行いその結果を比較する．

シミュレーションの流れを示す．まず，各符号化方式の定義に基づき，M_n 個の符号語リストを二次元配列に表す．その符号語リストの中から，$\frac{1}{M_n}$ の確率で符号語を 1 つ選択し，送信する符号語とする．そして選ばれた符号語に，雑音を加算する．それを受信語とし，復号する際に，3.3 節にて示した距離を計算する．その距離が最も小さいものを送信された符号語と判断し，間違っていたらカウントを 1 増やす．この一連の流れを 1000000 回実行し，誤っていた回数の相対頻度によって誤り確率を算出する．さらに，σ を変化させてシミュレーションを行うことで，通信路容量に応じた誤り確率の変化の様子を確認する．

図 8 に示すのは，符号語長 n が 2 のときの比較である．縦軸が誤り確率で，横軸が通信路容量である．グラフの右側の方が雑音が小さくて通信路容量が大きくなっている．グラフ内の赤い縦線は，符号化レートの位置を示している．グラフを確認すると，ランダム符号の誤り確率が最も大きく，性能が悪いことが分かる．そして，QPSK と球充填を比較すると，若干，球充填符号が方が誤り確率が小さいため，最も性能が良いことが分かる．

図 9 に示すのは, n が 3 ときの比較である. グラフを確認すると, 二次元同様に, ランダム符号が最も誤り確率が高く, QPSK と球充填では, 球充填の方が誤り確率が小さいことが確認できる.

図 10 に示すのは, n が 4 のときの結果である. 符号化レートが同一の 3 つの符号を実線で表示して比較する. グラフを確認すると, ランダム符号が最も誤り確率が大きく, 次に誤り確率が大きいのが, QPSK と BPSK の併用符号である. そして, 最も性能がいいのは, 球充填符号である. さらに, 球充填の優位性を示すために, QPSK2 つと BPSK2 つの併用符号を破線で示す. この符号の符号語数は 64 個であり, 他の 3 つの符号の半分の符号語数である. したがって, 符号化レートは他の符号よりも低く, 公平な性能比較ではない. しかしながら, グラフから分かるように, 併用符号の方が符号化レートが低いにもかかわらず, 球充填符号の方が誤り確率は低い. これは, 球充填符号の誤り訂正の性能の高さを表している.

図 11 には, QPSK との比較を示すため, 雑音の大きさ σ を固定し, 横軸を符号化レートにしたグラフを示す. PSK 系のグラフは, 左側の QPSK3 つと BPSK1 つの併用符号と, 右側の QPSK の誤り確率を破線で結んで表示した. また, 球充填符号のグラフは, 左側の点が 2 倍の基本領域で, 右の点が 3 倍の基本領域の場合の誤り確率である. ここで,QPSK の誤り確率を球充填符号の 2 倍および 3 倍の基本領域に対応する点で挟むことにより, 両者の誤り確率の変化傾向を比較できるようにした. グラフを確認すると, 球充填符号のグラフよりも QPSK の点が上側に位置していることが分かる. したがって, 球充填符号は QPSK よりも誤り確率が低い傾向にあるといえる. また, PSK 系のグラフの傾きが他のグラフの傾きよりも小さいことから,

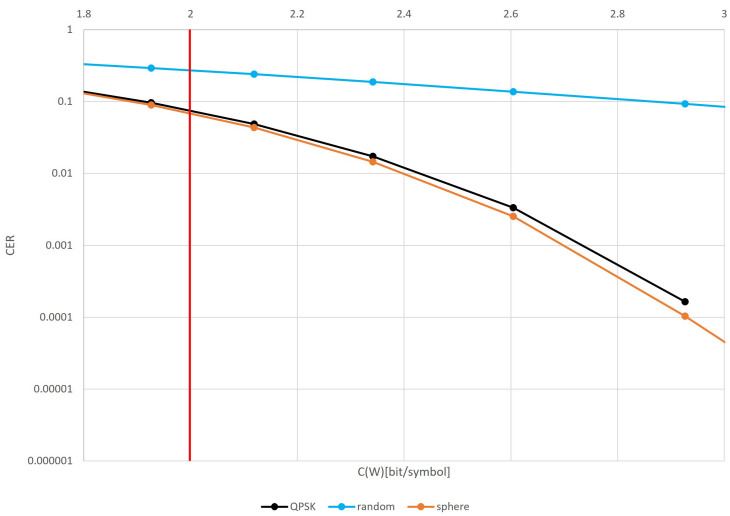


図 8: 二次元の性能比較

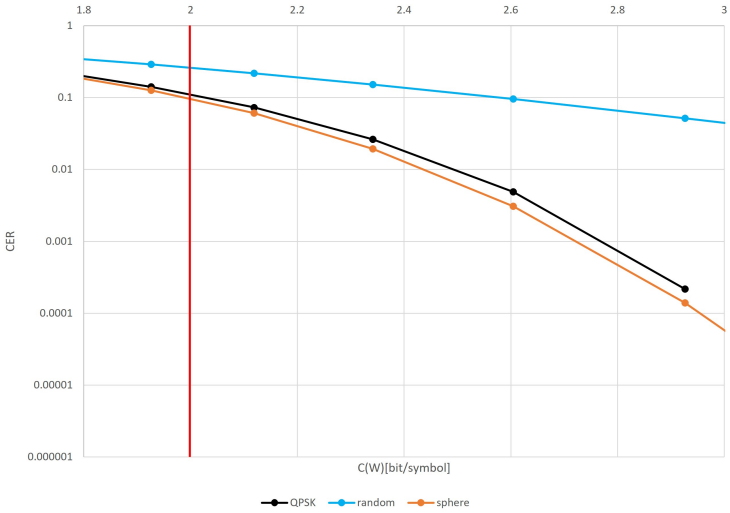


図 9: 三次元の性能比較

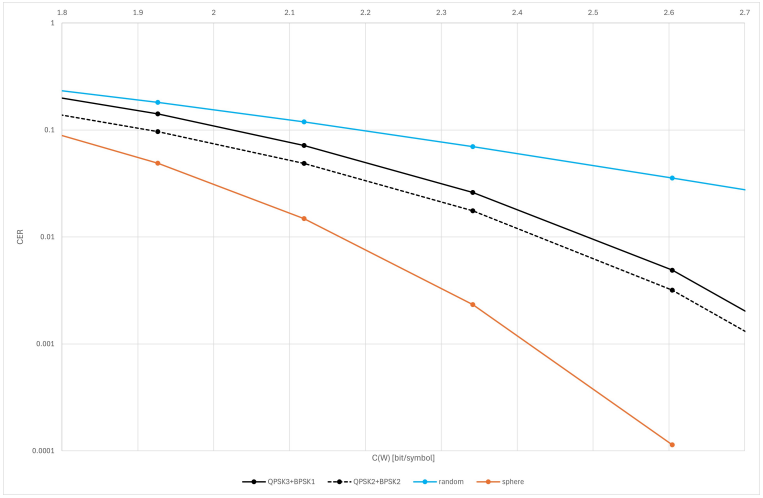


図 10: 四次元の性能比較 (同一の符号化レートの場合)

BPSK と QPSK を併用すると性能が低下することが示唆される.

図 12 に示すのは, n が 5 のときの比較である. 図 10 と同様に, 符号化レートが同一の 3 つの符号を実線で表示し, それよりも符号化レートが低い併用符号を破線で表示している. グラフを確認すると, ランダム符号が最も誤り確率が大きく, 次に誤り確率が大きいのが, QPSK と BPSK の併用符号である. そして, 最も誤り確率が低いのは球充填符号である.

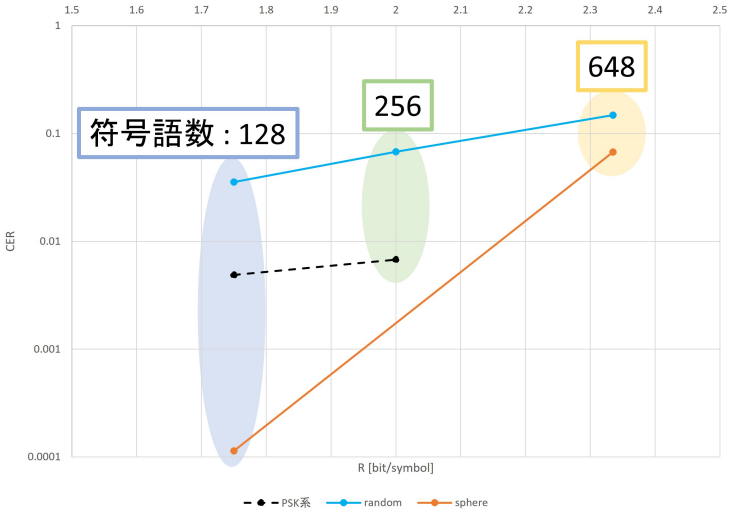


図 11: 四次元の性能比較 (横軸を符号化レートにした場合)

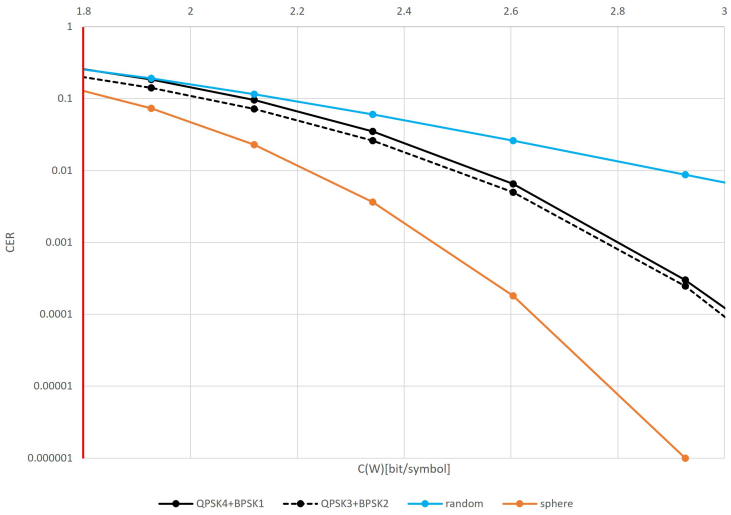


図 12: 五次元の性能比較 (同一の符号化レートの場合)

図 13 グラフは図 11 と同様に, 横軸を符号化レートにしたグラフを示す. グラフを確認すると, 五次元でも球充填符号のグラフよりも QPSK の点が上側に位置していることが分かる. したがって, 球充填符号は QPSK よりも誤り確率が低い傾向にあるといえる.

図 14 に示すのは, n が 6 のときの結果である. 四次元と五次元と同様に, 球充填符号の符号

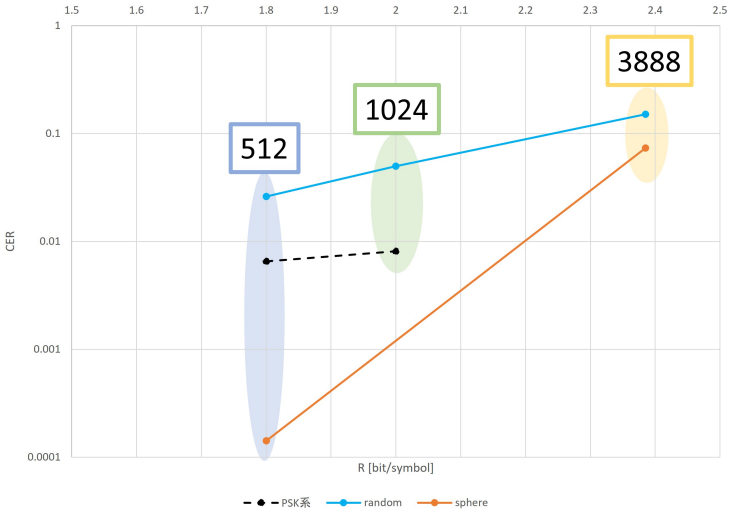


図 13: 五次元の性能比較 (横軸を符号化レートにした場合)

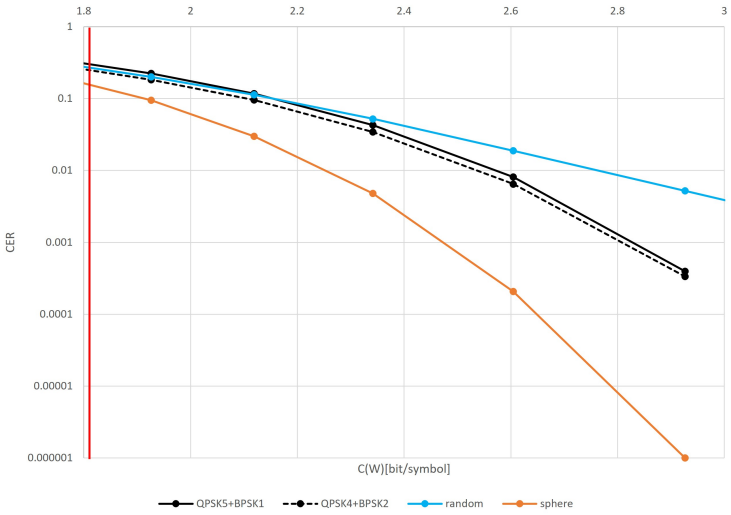


図 14: 六次元の性能比較 (同一の符号化レートの場合)

語数が QPSK と BPSK の併用符号と一致した. そのため, 同一の符号化レートの 3 つの符号を実線で示し, それよりも符号化レートが低い併用符号を破線で表示している. グラフを確認すると, 最も誤り確率が低いのは球充填符号であることが分かる.

図 15 のグラフは図 11 と同様に, 横軸を符号化レートにしたグラフを示す. グラフを確認す

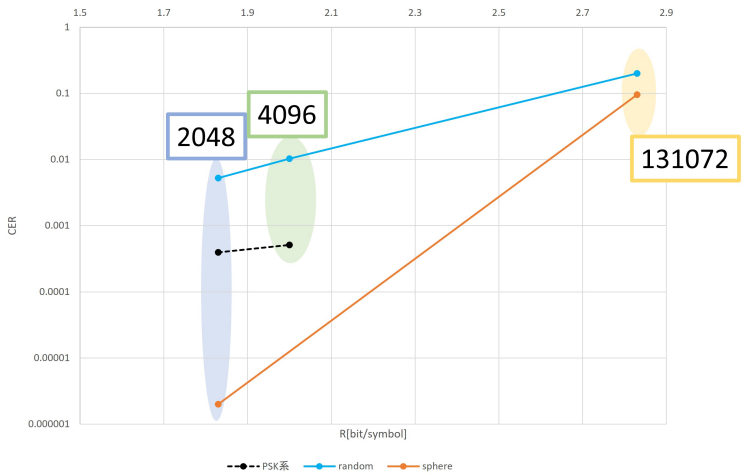


図 15: 六次元の性能比較 (横軸を符号化レートにした場合)

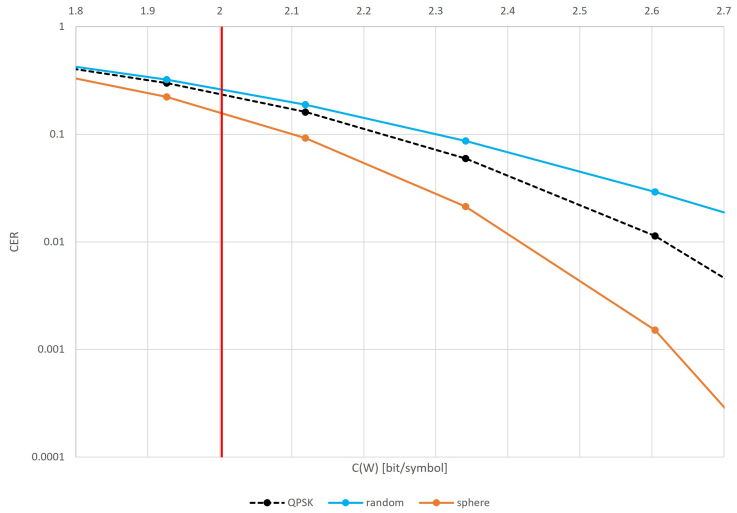


図 16: 七次元の性能比較 (同一の符号化レートの場合)

ると, 球充填符号のグラフよりも QPSK の点が上側に位置していることが分かる. したがって, 六次元でも, 球充填符号は QPSK よりも誤り確率が低い傾向にあるといえる.

図 16 に示すのは, n が 7 のときの結果である. 四次元〜六次元の方法を用いても符号語数が一致しないため, 球充填符号よりも符号化レートが低い QPSK を破線で表示して比較する. グラフを確認すると, 3 つの符号の中で最も誤り確率が小さいのは球充填符号である.

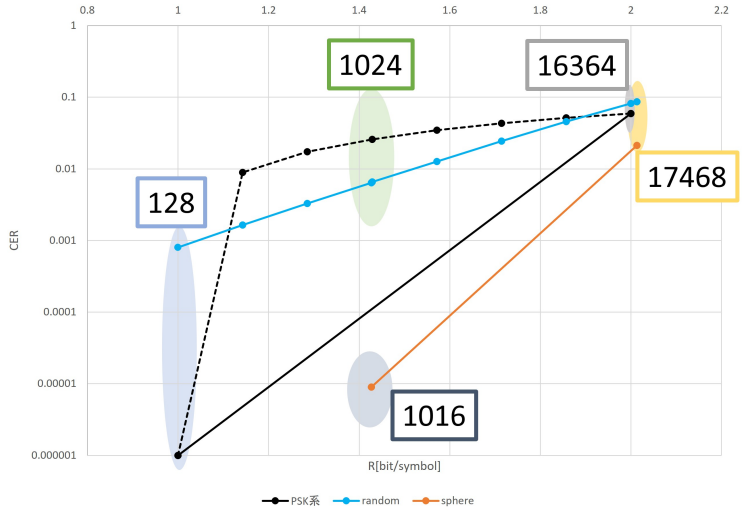


図 17: 七次元の性能比較 (横軸を符号化レートにした場合)

図 17 のグラフは図 11 と同様に, 横軸を符号化レートにしたグラフを示す. PSK 系のグラフは, 最も右の点が QPSK で, その左隣の点は一シンボルのみを BPSK にした場合の誤り確率である. さらに左に移るにつれて, 一シンボルずつ BPSK にしていき, 最も左の点が完全な BPSK でグラフを作成した. この変化の傾向を確認すると, 併用符号の誤り確率は, ランダム符号化よりも大きいことが分かる. これは, n の値が大きくなるにつれてランダム符号化の性能が向上することが一因であると考えられる. しかし, 全シンボルを QPSK とした場合および全シンボルを BPSK とした場合は, いずれもランダム符号化より誤り確率が小さいことから, 併用符号では誤り確率が著しく増大する傾向があるといえる. この要因は, 各軸ごとのシンボル間隔が不均一になることで復号領域が歪む点にある. 具体的には, 併用符号では復号領域が不規則となり雑音耐性が低下し, 誤り確率が増加する. 一方, 全シンボルを QPSK または BPSK のみとした場合は, 復号領域が規則的かつ等間隔であるため, 誤り確率は低くなる. このことから, 併用符号の構成は復号領域の歪みにより誤り確率を増大させるといえる. これは四次元, 五次元, 六次元において, PSK 系のグラフの傾きが他の符号化方式よりも小さいという結果とも整合的である. そのため, BPSK と QPSK の結果を直線で補間し, それ以外の部分は破線で結ぶことで, 性能の低い結果を含めず, PSK 系における性能変化の傾向を明確に示す. グラフを確認すると, PSK 系よりも球充填の方が誤り確率が低い傾向にあることが確認できる. よって, 最も性能が良いのは, 球充填符号である.

図 18 に示すのは, n が 8 のときの比較である. グラフを確認すると, 最も性能が良いのは球充填符号であることが確認できる.

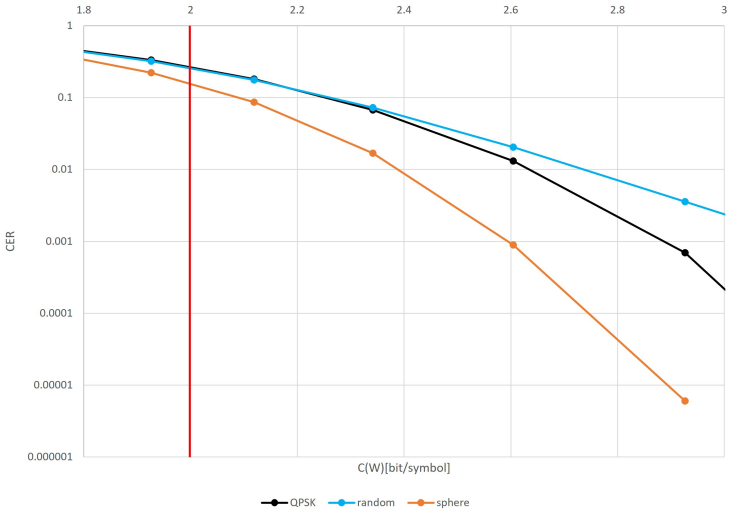


図 18: 八次元の性能比較

6 まとめ

本研究では, 位相変調の符号化の部分に着目し, 球充填問題に基づいた符号化方式を導入した. これにより, 符号の配置を理論的に最適化したことで, QPSK やランダム符号化と比較して, 球充填符号の優位性を示すことができた. 今後の課題としては, 符号語の構成方法にさらなる改良の余地があると考えている. 現段階では符号語リストを作成しているため, 計算量が符号語数や符号語長に比例して増大しており, 実用化は困難である. そのため, これを線形符号として表現できるようになれば, アルゴリズム的な生成や復号が可能となり, 実際の通信システムでの実装に近づくと考えている.

謝辞

本研究を行うにあたり, 多大なるご指導とご鞭撻を賜りました指導教員の西新幹彦准教授に心より感謝の意を表する.

参考文献

[1] Thomas M. Cover, Joy A.Thomas, Elements of Information Theory 2nd ed., Wiley, 2006.

[2] 韓太舜, 情報理論における情報スペクトル的方法, 培風館, 1998.
[3] 蓑谷千凰彦, 統計分布ハンドブック, 朝倉書店, 2003.
[4] 堀達裕, 「各種雑音に対する位相変調の通信路容量について」, 信州大学工学部卒業論文 (指導教員：西新幹彦), 2024 年 4 月.
[5] 和田山正, 誤り訂正技術の基礎, 森北出版, 2010.
[6] George G. Szpiro(青木薫訳), ケプラー予想, 新潮社, 2005.
[7] J.H. Conway, N.J.A. Sloane, Sphere Packings, Lattices and Groups 3rd ed., Springer, 1999.
[8] C 言語による乱数生成, https://omitakahiro.github.io/random/random_variables_generation.html, 2025 年 12 月閲覧.
[9] A handy approximation for the error function and its inverse, https://www.academia.edu/9730974/A_handy_approximation_for_the_error_function_and_its_inverse, 2025 年 12 月閲覧.

付録 A　最尤復号のための距離

3.3 節にて示した位相差を用いた距離が, 本研究の通信路にて最尤復号であることを証明する.

最尤復号は, 出力系列 $Y^n = Y_1 \dots Y_n$ に対して, 入力系列 $X^n = X_1 \dots X_n$ が与えられた条件付き確率

$$\Pr\{Y^n = y_1 \dots y_n | X^n = x_1 \dots x_n\} \tag{50}$$

を最大化する復号法である. また, 本研究の通信路の条件付き確率は,

$$W(y|x) \;=\; \Pr\{\, Y = y \,|\, X = x \,\} \tag{51}$$

である. ここで, 2.2 節より, $Y = X \oplus N$ であり, 入力 X と雑音 N は独立である. また, $\oplus$ と $\ominus$ は $a \oplus b \ominus b = a$ の関係にある. よって, 式 (51) は,

$$\begin{aligned} W(y|x) \;&=\; \Pr\{\, X \oplus N = y \,|\, X = x \,\} && (52)\\ &=\; \Pr\{\, N = y \ominus x \,|\, X = x \,\} && (53)\\ &=\; \Pr\{\, N = y \ominus x \,\} && (54) \end{aligned}$$

となる. この確率を雑音の密度関数 (19) を用いて,

$$\Pr\{\, N = y \ominus x \,\} \;=\; \bar{f}_N(y \ominus x) \tag{55}$$

と表すことができる. よって, 式 (50) は,

$$\begin{aligned} &\Pr\{Y^n = y_1 \dots y_n | X^n = x_1 \dots x_n\} && (56)\\ &= \prod_{i=1}^{n} W(y_i|x_i) && (57)\\ &= \prod_{i=1}^{n} \left(\frac{1}{\sqrt{2\pi}\sigma \mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)} e^{-\frac{(y_n \ominus x_n)^2}{2\sigma^2}} \right) && (58)\\ &= \left(\frac{1}{\sqrt{2\pi}\sigma \mathrm{erf}\left(\frac{\pi}{\sqrt{2}\sigma}\right)} \right)^n e^{-\frac{1}{2\sigma^2}\left(\sum_{i=1}^{n} (y_i \ominus x_i)^2\right)} && (59) \end{aligned}$$

となる. 式 (59) のなかで x に依存するのは, 指数関数の肩の

$$\sum_{i=1}^{n} (y_i \ominus x_i)^2 \tag{60}$$

の部分である．この指数関数は単調減少であるため, 条件付き確率 (50) を最大化することは, この式 (60) を最小化することと等価である．したがって, 本研究の通信路における最尤復号は, すべての符号語 C の中から,

$$\hat{\boldsymbol{x}} = \arg\min_{x \in C} \sum_{i=1}^{n} (y_i \ominus x_i)^2 \tag{61}$$

を満たす符号語を選択する操作として与えられる．となり, これは, 3.3 節の位相差を用いた距離

$$\sqrt{\sum_{i=1}^{n} (y_i \ominus x_i)^2} \tag{62}$$

を最小化する復号と一致する．以上より, 位相差を用いた距離は, 本研究の通信路において最尤復号距離であることが示された.

付録 B　基本領域の大きさ

4 章で紹介した四次元から八次元までの格子構造の基本領域の大きさを示していく．

まず初めに, D_n 格子の基本領域の大きさが各軸方向に 2 であることを示す．D_n 格子の任意の格子点 $\boldsymbol{x} = (x_1, \ldots, x_n) \in D_n$ をとる．ここで D_n は, 整数格子のなかで, 成分和が偶数となる格子全体である．そして, この $\boldsymbol{x}$ の i 番目の成分を 2 だけ増加させた格子点

$$\boldsymbol{x}' = (x_1, \ldots, x_i + 2, \ldots x_n) \tag{63}$$

を考える．このとき, $\boldsymbol{x}'$ が再び D_n に属することを確かめる．まず, D_n は整数格子であるため, $\boldsymbol{x}$ のすべての成分は整数である．したがって, 1 つの成分に整数 2 を加えた $\boldsymbol{x}'$ も整数である．よって, $\boldsymbol{x}'$ は整数格子である．次に, 成分和が偶数であることを確認する．$\boldsymbol{x} \in D_n$ より $x_1 + \cdots + x_n$ は偶数である．さらに, 偶数に 2 を加えても偶数のままであるため, $\boldsymbol{x}'$ も成分和が偶数であり, D_n の定義を満たす．以上より, 任意の $\boldsymbol{x} \in D_n$ に対して, ある 1 成分だけを 2 だけずらした格子点 $\boldsymbol{x}'$ も D_n に属する．したがって, D_n 格子の各軸方向の基本領域の大きさは 2 である.

同様にして, E_8 格子の基本領域の大きさも各軸方向に 2 であることを示す．E_8 格子の任意の格子点 $\boldsymbol{x} = (x_1, \ldots, x_8) \in E_8$ をとり, この $\boldsymbol{x}$ の i 番目の成分を 2 だけ増加させた格子点

$$\boldsymbol{x}' = (x_1, \ldots, x_i + 2, \ldots x_8) \tag{64}$$

を考える．このとき, $\boldsymbol{x}'$ が再び E_8 に属することを確かめる．E_8 格子の定義は, 整数格子と半整数格子から構成され, この格子の成分和が偶数になるものである．このうち, 整数格子の場

合は, D_n 格子と同様であるため, 任意の成分に 2 を加えても再び E_8 に属する. 半整数格子の場合も同様に, 任意の成分に 2 を加えても半整数のままであり, 成分和も偶数のまま保たれる. したがって, この場合も $\boldsymbol{x}'$ は E_8 に属する. 以上より, 任意の $\boldsymbol{x} \in E_8$ に対して, ある 1 成分を 2 だけずらした格子点 $\boldsymbol{x}'$ も E_8 に属する. したがって, E_8 格子の基本領域の大きさも各軸方向に 2 である.

次に E_7 格子の基本領域の大きさが $\sqrt{2}$ であることを示す. E_7 格子の任意の格子点 $\boldsymbol{x} = (x_1, \ldots, x_8) \in E_7$ をとる. この八次元成分から成る格子点は、式 (29) から (35) の七次元基底 $\{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_7\}$ によって

$$\boldsymbol{x} = c_1\boldsymbol{e}_1 + \cdots + c_7\boldsymbol{e}_7 \tag{65}$$

と表せる. E_7 格子は, E_8 格子に属する格子のうち, 成分和が 0 になるものである. そして, この $\boldsymbol{x}$ を基底ベクトル $\boldsymbol{e}_i$ 方向 $(i = 1, \ldots, 7)$ に $\sqrt{2}$ だけ移動させた格子点

$$\boldsymbol{x}' = \boldsymbol{x} + \sqrt{2}\boldsymbol{e}_i \tag{66}$$

を考える. このとき, $\boldsymbol{x}'$ が再び E_7 に属することを確かめる. $\sqrt{2}\boldsymbol{e}_i$ は, 八次元ベクトルとして,

$$\sqrt{2}\boldsymbol{e}_1 = \frac{1}{2}(1,1,1,1,-1,-1,-1,-1) \tag{67}$$

$$\sqrt{2}\boldsymbol{e}_2 = \frac{1}{2}(1,1,-1,-1,1,1,-1,-1) \tag{68}$$

$$\sqrt{2}\boldsymbol{e}_3 = \frac{1}{2}(1,-1,1,-1,1,-1,1,-1) \tag{69}$$

$$\sqrt{2}\boldsymbol{e}_4 = \frac{1}{2}(1,-1,-1,1,-1,1,1,-1) \tag{70}$$

$$\sqrt{2}\boldsymbol{e}_5 = \frac{1}{2}(1,-1,1,-1,-1,1,-1,1) \tag{71}$$

$$\sqrt{2}\boldsymbol{e}_6 = \frac{1}{2}(1,1,-1,-1,-1,-1,1,1) \tag{72}$$

$$\sqrt{2}\boldsymbol{e}_7 = \frac{1}{2}(1,-1,-1,1,1,-1,-1,1) \tag{73}$$

のように表される. これらの 7 つのベクトルはそれぞれ, 4 つの成分が $\frac{1}{2}$ で残りの 4 つの成分が $-\frac{1}{2}$ の半整数格子であるため, $\sqrt{2}\boldsymbol{e}_i \in E_8$ である. また, E_7 格子は E_8 格子に属するため, $\boldsymbol{x}$ は E_8 格子に属する. したがって, $\boldsymbol{x}' = \boldsymbol{x} + \sqrt{2}\boldsymbol{e}_i$ は E_8 格子同士の和であるため, $\boldsymbol{x}'$ は E_8 に属する. 次に, $\boldsymbol{x}'$ が成分和が 0 であることを確認する. $\boldsymbol{x}$ は E_7 に属するため成分和が 0 であり, $\sqrt{2}\boldsymbol{e}_i$ の成分和も, $4\left(\frac{1}{2}\right) + 4\left(-\frac{1}{2}\right) = 0$ であるため, $\boldsymbol{x}'$ は E_7 格子の定義を満たす. 以上より, $\boldsymbol{x}$ を各基底方向に $\sqrt{2}$ だけ移動しても E_7 の条件を満たすため, E_7 格子は各基底方向に大きさが $\sqrt{2}$ の基本領域をもつ.

最後に, E_6 格子の基本領域の大きさが, e_1 方向に $2\sqrt{3}$, e_2 から e_6 方向に 2 であることを示す. まず, e_1 方向の基本領域の大きさが $2\sqrt{3}$ であることを示す. E_6 格子の任意の格子点

$\boldsymbol{x} = (x_1, \ldots, x_8) \in E_6$ をとる．この八次元成分から成る格子点は、式 (40) から (45) の六次元基底 $\{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_6\}$ によって

$$\boldsymbol{x} = c_1\boldsymbol{e}_1 + \cdots + c_6\boldsymbol{e}_6 \tag{74}$$

と表せる．そして, この $\boldsymbol{x}$ を基底ベクトル $\boldsymbol{e}_1$ 方向に $2\sqrt{3}$ だけ移動させた格子点

$$\boldsymbol{x}' = \boldsymbol{x} + 2\sqrt{3}\boldsymbol{e}_1 \tag{75}$$

を考える．このとき, $\boldsymbol{x}'$ が再び E_6 に属することを確かめる．$2\sqrt{3}\boldsymbol{e}_1$ は, 八次元ベクトルとして,

$$2\sqrt{3}\boldsymbol{e}_1 = (0, 0, 0, 0, 0, 2, 2, 2) \tag{76}$$

のように表される．このベクトルは成分が全て整数であり, なおかつそれらの和が偶数であるため, $2\sqrt{3}\boldsymbol{e}_1 \in E_8$ である．また, E_6 格子は E_8 格子に属するため, $\boldsymbol{x}$ は E_8 格子に属する．したがって, $\boldsymbol{x}' = \boldsymbol{x} + 2\sqrt{3}\boldsymbol{e}_1$ は E_8 格子同士の和であるため, $\boldsymbol{x}'$ は E_8 に属する．次に, $\boldsymbol{x}'$ の 6 番目と 7 番目, そして 8 番目の成分が同じ値であるかを確かめるが, $\boldsymbol{x} \in E_6$ と式 (76) より自明である．よって, e_1 方向の基本領域の大きさは $2\sqrt{3}$ である．

同様にして, e_2 から e_6 方向の基本領域の大きさが 2 であることを示す．式 (74) を基底ベクトル $\boldsymbol{e}_2$ から $\boldsymbol{e}_6$ 方向の任意の向きに 2 だけ移動させた格子点

$$\boldsymbol{x}' = \boldsymbol{x} + 2\boldsymbol{e}_i \tag{77}$$

を考える．$2\boldsymbol{e}_i$ は, 八次元ベクトルとして,

$$2\boldsymbol{e}_2 = (2, 0, 0, 0, 0, 0, 0, 0) \tag{78}$$
$$2\boldsymbol{e}_3 = (0, 2, 0, 0, 0, 0, 0, 0) \tag{79}$$
$$2\boldsymbol{e}_4 = (0, 0, 2, 0, 0, 0, 0, 0) \tag{80}$$
$$2\boldsymbol{e}_5 = (0, 0, 0, 2, 0, 0, 0, 0) \tag{81}$$
$$2\boldsymbol{e}_6 = (0, 0, 0, 0, 2, 0, 0, 0) \tag{82}$$

のように表される．これらのベクトルは成分が全て整数であり, その和が偶数であるため, $2\boldsymbol{e}_i \in E_8$ である．また, $\boldsymbol{x}$ は E_6 格子であるため, E_8 格子に属する．したがって, $\boldsymbol{x}' = \boldsymbol{x} + 2\boldsymbol{e}_i$ は E_8 格子同士の和であるため, $\boldsymbol{x}'$ は E_8 に属する．次に, $\boldsymbol{x}'$ の 6 番目と 7 番目, そして 8 番目の成分が同じ値であるかを確かめるが, $\boldsymbol{x} \in E_6$ と式 (78)〜(82) より自明である．よって, e_2 から e_6 方向の基本領域の大きさは 2 である．以上より, E_6 格子は e_1 方向に $2\sqrt{3}$, e_2 から e_6 方向に 2 の大きさの基本領域をもつ．

付録 C　ソースコード

C.1　QPSK のシミュレーションプログラム

文献 [8],[9] を参考にしてプログラムを作成した.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "MT.h"

#define n 5     //ブロック長 (任意に設定)
#define k 5      //BPSK 部分の長さ (任意に設定)
#define M_n (1 << (2 * (n - k) + k))  //符号語数
#define N 1000000 //試行回数 (任意に設定)
#define M_PI acos(-1) //π


int codewords[M_n][n];  //符号語リスト

//一様乱数生成
double uniform_random() {
    return genrand_real1();  //0.0 から 1.0 の一様乱数を生成
}

//逆誤差関数の近似 (文献 [9] を参考にした)
double erfinv(double x) {
    double a = 0.147;
    double ln_term = log(1.0 - x * x);
    double part1 = (2.0 / (M_PI * a)) + (ln_term / 2.0);
    double part2 = ln_term / a;

    double sign_x = (x > 0) - (x < 0);  //符号関数
    return sign_x * sqrt(sqrt(part1 * part1 - part2) - part1);
}

//正規分布の乱数発生
double gauss_random(double sigma) {
    double u = uniform_random();

    return sqrt(2) * sigma * erfinv(erf(M_PI / (sqrt(2) * sigma)) * (2 * u - 1));
}

//通信路容量
double calculate_channel_capacity_gauss(double sigma) {
    return log2(2 * M_PI) + (sqrt(2 * M_PI) * erf(M_PI / (sqrt(2) * sigma))
    * sigma * log(exp(-(M_PI * M_PI) / (2 * sigma * sigma)) / (sqrt(2 * M_PI)
    * erf(M_PI / (sqrt(2) * sigma)) * sigma)) + M_PI * exp(-(M_PI * M_PI) /
    (2 * sigma * sigma)) - (sqrt(M_PI) * erf(M_PI / (sqrt(2) * sigma)) * sigma) /
    sqrt(2) + M_PI * M_PI * sqrt(M_PI) * erf(M_PI / (sqrt(2) * sigma)) / (sqrt(2) * sigma))
    / (sqrt(2 * M_PI) * log(2) * erf(M_PI / (sqrt(2) * sigma)) * sigma);
}

// QPSK 変調
double qpsk_modulate(int tx_signal) {
    switch (tx_signal) {
    case 0: return M_PI / 4;
    case 1: return 3 * M_PI / 4;
    case 2: return -3 * M_PI / 4;
    case 3: return -M_PI / 4;
    }
}

// QPSK 復調
int qpsk_demodulate(double rx_signal) {
```

```
    //-πからπの範囲に収める
    rx_signal = fmod(rx_signal + M_PI, 2 * M_PI) - M_PI;

    //範囲に基づいて復調
    if (rx_signal > 0 && rx_signal <= M_PI / 2) {
        return 0;
    }
    else if (rx_signal > M_PI / 2 && rx_signal <= M_PI) {
        return 1;
    }
    else if (rx_signal > -M_PI && rx_signal <= -M_PI / 2) {
        return 2;
    }
    else {
        return 3;
    }
}

//BPSK 変調
double bpsk_modulate(int tx_signal) {
    switch (tx_signal) {
    case 0: return M_PI / 2;
    case 1: return -M_PI / 2;
    }
}

//BPSK 復調
int bpsk_demodulate(double rx_signal) {
    //-πからπの範囲に収める
    rx_signal = fmod(rx_signal + M_PI, 2 * M_PI) - M_PI;

    //復調結果を返す
    if (rx_signal > 0 && rx_signal <= M_PI) {
        return 0;
    }
    else {
        return 1;
    }
}


//index から符号語を生成
void generate_codeword_from_index(int codeword[n], int index) {
    int qpsk_part_len = n - k;
    int bpsk_symbols = 1 << k;                    //2^k
    int qpsk_symbols = 1 << (2 * qpsk_part_len); //4^(n-k)

    int bpsk_index = index % bpsk_symbols;
    int qpsk_index = index / bpsk_symbols;

    //QPSK 部分
    for (int j = qpsk_part_len - 1; j >= 0; j--) {
        codeword[j] = qpsk_index % 4;
        qpsk_index /= 4;
    }

    //BPSK 部分
    for (int j = n - 1; j >= qpsk_part_len; j--) {
        codeword[j] = bpsk_index % 2;
        bpsk_index /= 2;
    }
}

//全符号語を配列に生成
void generate_all_codewords(void) {
    for (int i = 0; i < M_n; i++) {
        generate_codeword_from_index(codewords[i], i);
    }
}
```

```
//1 回分のシミュレーション
int simulate_mixed_R(double sigma) {
    static int print_count = 0;  //出力するためのカウンタ
    int random_index = genrand_int32() % M_n;
    int* selected_word = codewords[random_index];

    double modulated_signal[n];
    double noisy_signal[n];
    int demodulated_word[n];

    //変調
    for (int j = 0; j < n; j++) {
        if (j < n - k)
            modulated_signal[j] = qpsk_modulate(selected_word[j]);
        else
            modulated_signal[j] = bpsk_modulate(selected_word[j]);
    }

    //雑音付加
    for (int j = 0; j < n; j++) {
        double noise = gauss_random(sigma);
        noisy_signal[j] = modulated_signal[j] + noise;
    }

    // --- 復調 ---
    for (int j = 0; j < n; j++) {
        if (j < n - k)
            demodulated_word[j] = qpsk_demodulate(noisy_signal[j]);
        else
            demodulated_word[j] = bpsk_demodulate(noisy_signal[j]);
    }

    //誤り判定
    for (int j = 0; j < n; j++) {
        if (selected_word[j] != demodulated_word[j])
            return 1;  //誤りあり
    }

    return 0;  //誤りなし
}

int main(void) {

    // 乱数初期化
    srand((unsigned int)time(NULL));
    init_genrand((unsigned long)time(NULL));

    // 結果を CSV ファイルに書き込む
    FILE* file = fopen("simulation_results.csv", "w");

    // CSV への書き込み
    fprintf(file, "C_W,CER\n");

    // 全符号語生成
    generate_all_codewords();


    // sigma を変化させ、対応する C_W を計算
    for (double sigma = 0.1; sigma <= 1.5; sigma += 0.05) {
        double C_W = calculate_channel_capacity_gauss(sigma);  // 通信路容量の計算
        int total_errors = 0;

        // N 回の試行を行う
        for (int i = 0; i < N; i++) {
            total_errors += simulate_mixed_R(sigma);
        }

        // CER（符号語誤り率）を計算
```

```
        double cer = (double)total_errors / N;

        // 結果を表示
        printf("C_W = %.4f, 符号語誤り確率 (CER): %d / %d = %.8f\n", C_W, total_errors, N, cer);

        // CSV ファイルに結果を書き込む
        fprintf(file, "%.4f,%.8f\n", C_W, cer);
    }

    fclose(file); // ファイルを閉じる


    return 0;
}
```

C.2 ランダム符号のシミュレーションプログラム

文献 [8],[9] を参考にしてプログラムを作成した.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "MT.h"

#define n 5      // ブロック長 (任意に設定)
#define M_n 1024    // 符号語数 (任意に設定)
#define N 1000000 // 試行回数 (任意に設定)
#define M_PI acos(-1) // π

// 一様乱数生成
double uniform_random() {
    return genrand_real1();  // 0.0 から 1.0 の一様乱数を生成
}

// 逆誤差関数の近似 (文献 [9] を参考にした)
double erfinv(double x) {
    double a = 0.147;
    double ln_term = log(1.0 - x * x);
    double part1 = (2.0 / (M_PI * a)) + (ln_term / 2.0);
    double part2 = ln_term / a;

    double sign_x = (x > 0) - (x < 0);  // 符号関数
    return sign_x * sqrt(sqrt(part1 * part1 - part2) - part1);
}

//正規分布の乱数発生
double gauss_random(double sigma) {
    double u = uniform_random();

    return sqrt(2) * sigma * erfinv(erf(M_PI / (sqrt(2) * sigma)) * (2 * u - 1));
}

//通信路容量
double calculate_channel_capacity_gauss(double sigma) {
    return log2(2 * M_PI) + (sqrt(2 * M_PI) * erf(M_PI / (sqrt(2) * sigma))
    * sigma * log(exp(-(M_PI * M_PI) / (2 * sigma * sigma)) / (sqrt(2 * M_PI)
    * erf(M_PI / (sqrt(2) * sigma)) * sigma)) + M_PI * exp(-(M_PI * M_PI) /
    (2 * sigma * sigma)) - (sqrt(M_PI) * erf(M_PI / (sqrt(2) * sigma)) * sigma) /
    sqrt(2) + M_PI * M_PI * sqrt(M_PI) * erf(M_PI / (sqrt(2) * sigma)) / (sqrt(2) * sigma))
    / (sqrt(2 * M_PI) * log(2) * erf(M_PI / (sqrt(2) * sigma)) * sigma);
}

// 符号語の配列を生成 (-πからπの範囲)
void generate_codeword(double codeword[n]) {
    for (int i = 0; i < n; i++) {
```

```
        // 符号語をランダムに生成（-πからπの値）
        codeword[i] = genrand_real1() * 2 * M_PI - M_PI;
    }
}

// 受信信号を復号
double phase_demodulate(double rx_signal) {
    // 信号を -π から π の範囲に収める
    rx_signal = fmod(rx_signal + M_PI, 2 * M_PI) - M_PI;

    // 復号の結果として -π ～ π の数値を返す
    return rx_signal;
}

// 受信符号と各符号語の距離を計算
double calculate_distance(double received[n], double codeword[n]) {
    double total_squared_difference = 0.0;

    for (int i = 0; i < n; i++) {
        double diff = received[i] - codeword[i];
        diff = fmod(diff + M_PI, 2 * M_PI);  // [-π, π) に変換
        if (diff < 0) diff += 2 * M_PI;
        diff -= M_PI;

        total_squared_difference += diff * diff;  // 2 乗
    }

    return sqrt(total_squared_difference);  // ユークリッド距離
}

//1 回のシミュレーション
int simulate(double sigma) {
    double codeword[n];  // 一回目の正しい符号語
    double received_signal[n];  // 一回目の受信信号
    double decoded_signal[n];  // 一回目の復号された信号

    // 一回目の処理：正しい符号語を生成し、差を計算
    generate_codeword(codeword);

    // 符号語に雑音を加えて受信信号を生成
    for (int i = 0; i < n; i++) {
        double noise = gauss_random(sigma);
        received_signal[i] = codeword[i] + noise;
    }

    // 受信信号を復号
    for (int i = 0; i < n; i++) {
        decoded_signal[i] = phase_demodulate(received_signal[i]);
    }

    // 一回目の差を計算
    double initial_difference = calculate_distance(decoded_signal, codeword);

    // ダミー符号語との比較ループ
    for (int trial = 1; trial <= M_n; trial++) {
        double dummy_codeword[n];
        double dummy_received_signal[n];
        double dummy_decoded_signal[n];

        // ダミーの符号語を生成
        generate_codeword(dummy_codeword);

        // ダミー符号語に雑音を加えて受信信号を生成
        for (int i = 0; i < n; i++) {
            double noise = gauss_random(sigma);
            dummy_received_signal[i] = dummy_codeword[i] + noise;
        }

        // ダミーの受信信号を復号
```

```
        for (int i = 0; i < n; i++) {
            dummy_decoded_signal[i] = phase_demodulate(dummy_received_signal[i]);
        }

        // ダミーの復号信号と一回目の符号語の差を計算
        double dummy_difference = calculate_distance(dummy_decoded_signal, codeword);

        // 差が一回目より小さい場合、送信失敗と判定し終了
        if (dummy_difference < initial_difference) {
            return 1;  // 送信失敗
        }
    }

    // M_n 回の試行を行い、送信成功と判定
    return 0;  // 送信成功
}

int main(void) {
    // 乱数シードを設定
    srand((unsigned int)time(NULL));

    // 結果を CSV ファイルに書き込む
    FILE* file = fopen("simulation_results.csv", "w");

    // CSV ファイルのヘッダを書き込み
    fprintf(file, "C_W,CER\n");

    // sigma を 0.1 から 0.1 ずつ 2.6 まで変化させ、対応する C_W と CER を計算
    for (double sigma = 0.15; sigma <= 0.5; sigma += 0.05) {
        double C_W = calculate_channel_capacity(sigma);  // 通信路容量の計算
        int total_errors = 0;

        // N 回の試行を行い、誤り回数をカウント
        for (int i = 0; i < N; i++) {
            total_errors += simulate(sigma);
        }

        // CER（符号語誤り率）を計算
        double cer = (double)total_errors / N;

        // 結果を表示
        printf("C_W = %.4f, 符号語誤り確率 (CER): %d / %d = %.8f\n", C_W, total_errors, N, cer);

        // CSV ファイルに結果を書き込む
        fprintf(file, "%.4f,%.8f\n", C_W, cer);
    }

    fclose(file); // ファイルを閉じる

    return 0;
}
```

C.3 二次元と三次元の球充填符号のシミュレーションプログラム

文献 [8],[9] を参考にしてプログラムを作成した.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "MT.h"

#define n 3   // ブロック長（任意に設定）
#define M_n 64   // 符号語数（任意に設定）
#define N 1000000 // 試行回数（任意に設定）
#define M_PI acos(-1)
```

```
// 一様乱数生成
double uniform_random() {
    return genrand_real1();  // 0.0 から 1.0 の一様乱数を生成
}

// 逆誤差関数の近似（文献 [9] を参考にした）
double erfinv(double x) {
    double a = 0.147;
    double ln_term = log(1.0 - x * x);
    double part1 = (2.0 / (M_PI * a)) + (ln_term / 2.0);
    double part2 = ln_term / a;

    double sign_x = (x > 0) - (x < 0);  // 符号関数
    return sign_x * sqrt(sqrt(part1 * part1 - part2) - part1);
}

//正規分布の乱数発生
double gauss_random(double sigma) {
    double u = uniform_random();

    return sqrt(2) * sigma * erfinv(erf(M_PI / (sqrt(2) * sigma)) * (2 * u - 1));
}

//通信路容量
double calculate_channel_capacity_gauss(double sigma) {
    return log2(2 * M_PI) + (sqrt(2 * M_PI) * erf(M_PI / (sqrt(2) * sigma))
    * sigma * log(exp(-(M_PI * M_PI) / (2 * sigma * sigma)) / (sqrt(2 * M_PI)
    * erf(M_PI / (sqrt(2) * sigma)) * sigma)) + M_PI * exp(-(M_PI * M_PI) /
    (2 * sigma * sigma)) - (sqrt(M_PI) * erf(M_PI / (sqrt(2) * sigma)) * sigma) /
    sqrt(2) + M_PI * M_PI * sqrt(M_PI) * erf(M_PI / (sqrt(2) * sigma)) / (sqrt(2) * sigma))
    / (sqrt(2 * M_PI) * log(2) * erf(M_PI / (sqrt(2) * sigma)) * sigma);
}

//二次元の球充填符号
void sphere16(double codeword[M_n][n]) {
    codeword[0][0] = 0; codeword[0][1] = M_PI;
    codeword[1][0] = 0; codeword[1][1] = M_PI / 2;
    codeword[2][0] = 0; codeword[2][1] = 0;
    codeword[3][0] = 0; codeword[3][1] = -M_PI / 2;
    codeword[4][0] = M_PI / 2; codeword[4][1] = 3 * M_PI / 4;
    codeword[5][0] = M_PI / 2; codeword[5][1] = M_PI / 4;
    codeword[6][0] = M_PI / 2; codeword[6][1] = -M_PI / 4;
    codeword[7][0] = M_PI / 2; codeword[7][1] = -3 * M_PI / 4;
    codeword[8][0] = -M_PI / 2; codeword[8][1] = 3 * M_PI / 4;
    codeword[9][0] = -M_PI / 2; codeword[9][1] = M_PI / 4;
    codeword[10][0] = -M_PI / 2; codeword[10][1] = -M_PI / 4;
    codeword[11][0] = -M_PI / 2; codeword[11][1] = -3 * M_PI / 4;
    codeword[12][0] = M_PI; codeword[12][1] = M_PI;
    codeword[13][0] = M_PI; codeword[13][1] = M_PI / 2;
    codeword[14][0] = M_PI; codeword[14][1] = 0;
    codeword[15][0] = M_PI; codeword[15][1] = -M_PI / 2;
}

//三次元の球充填符号
void sphere64(double codeword[M_n][n]) {
    codeword[0][0] = 0; codeword[0][1] = M_PI; codeword[0][2] = M_PI;
    codeword[1][0] = 0; codeword[1][1] = M_PI / 2; codeword[1][2] = M_PI;
    codeword[2][0] = 0; codeword[2][1] = 0; codeword[2][2] = M_PI;
    codeword[3][0] = 0; codeword[3][1] = -M_PI / 2; codeword[3][2] = M_PI;
    codeword[4][0] = M_PI / 2; codeword[4][1] = 3 * M_PI / 4; codeword[4][2] = M_PI;
    codeword[5][0] = M_PI / 2; codeword[5][1] = M_PI / 4; codeword[5][2] = M_PI;
    codeword[6][0] = M_PI / 2; codeword[6][1] = -M_PI / 4; codeword[6][2] = M_PI;
    codeword[7][0] = M_PI / 2; codeword[7][1] = -3 * M_PI / 4; codeword[7][2] = M_PI;
    codeword[8][0] = -M_PI / 2; codeword[8][1] = 3 * M_PI / 4; codeword[8][2] = M_PI;
    codeword[9][0] = -M_PI / 2; codeword[9][1] = M_PI / 4; codeword[9][2] = M_PI;
    codeword[10][0] = -M_PI / 2; codeword[10][1] = -M_PI / 4; codeword[10][2] = M_PI;
    codeword[11][0] = -M_PI / 2; codeword[11][1] = -3 * M_PI / 4; codeword[11][2] = M_PI;
    codeword[12][0] = M_PI; codeword[12][1] = M_PI; codeword[12][2] = M_PI;
```

```
    codeword[13][0] = M_PI; codeword[13][1] = M_PI / 2; codeword[13][2] = M_PI;
    codeword[14][0] = M_PI; codeword[14][1] = 0; codeword[14][2] = M_PI;
    codeword[15][0] = M_PI; codeword[15][1] = -M_PI / 2; codeword[15][2] = M_PI;
    codeword[16][0] = 3 * M_PI / 4 + 0.4; codeword[16][1] = 3 * M_PI / 4; codeword[16][2] = M_PI / 2;
    codeword[17][0] = 3 * M_PI / 4 + 0.4; codeword[17][1] = M_PI / 4; codeword[17][2] = M_PI / 2;
    codeword[18][0] = 3 * M_PI / 4 + 0.4; codeword[18][1] = -M_PI / 4; codeword[18][2] = M_PI / 2;
    codeword[19][0] = 3 * M_PI / 4 + 0.4; codeword[19][1] = -3 * M_PI / 4; codeword[19][2] = M_PI / 2;
    codeword[20][0] = M_PI / 4 + 0.4; codeword[20][1] = M_PI; codeword[20][2] = M_PI / 2;
    codeword[21][0] = M_PI / 4 + 0.4; codeword[21][1] = M_PI / 2; codeword[21][2] = M_PI / 2;
    codeword[22][0] = M_PI / 4 + 0.4; codeword[22][1] = 0; codeword[22][2] = M_PI / 2;
    codeword[23][0] = M_PI / 4 + 0.4; codeword[23][1] = -M_PI / 2; codeword[23][2] = M_PI / 2;
    codeword[24][0] = -M_PI / 4 + 0.4; codeword[24][1] = 3 * M_PI / 4; codeword[24][2] = M_PI / 2;
    codeword[25][0] = -M_PI / 4 + 0.4; codeword[25][1] = M_PI / 4; codeword[25][2] = M_PI / 2;
    codeword[26][0] = -M_PI / 4 + 0.4; codeword[26][1] = -M_PI / 4; codeword[26][2] = M_PI / 2;
    codeword[27][0] = -M_PI / 4 + 0.4; codeword[27][1] = -3 * M_PI / 4; codeword[27][2] = M_PI / 2;
    codeword[28][0] = -3 * M_PI / 4 + 0.4; codeword[28][1] = M_PI; codeword[28][2] = M_PI / 2;
    codeword[29][0] = -3 * M_PI / 4 + 0.4; codeword[29][1] = M_PI / 2; codeword[29][2] = M_PI / 2;
    codeword[30][0] = -3 * M_PI / 4 + 0.4; codeword[30][1] = 0; codeword[30][2] = M_PI / 2;
    codeword[31][0] = -3 * M_PI / 4 + 0.4; codeword[31][1] = -M_PI / 2; codeword[31][2] = M_PI / 2;
    codeword[32][0] = 0; codeword[32][1] = M_PI; codeword[32][2] = 0;
    codeword[33][0] = 0; codeword[33][1] = M_PI / 2; codeword[33][2] = 0;
    codeword[34][0] = 0; codeword[34][1] = 0; codeword[34][2] = 0;
    codeword[35][0] = 0; codeword[35][1] = -M_PI / 2; codeword[35][2] = 0;
    codeword[36][0] = M_PI / 2; codeword[36][1] = 3 * M_PI / 4; codeword[36][2] = 0;
    codeword[37][0] = M_PI / 2; codeword[37][1] = M_PI / 4; codeword[37][2] = 0;
    codeword[38][0] = M_PI / 2; codeword[38][1] = -M_PI / 4; codeword[38][2] = 0;
    codeword[39][0] = M_PI / 2; codeword[39][1] = -3 * M_PI / 4; codeword[39][2] = 0;
    codeword[40][0] = -M_PI / 2; codeword[40][1] = 3 * M_PI / 4; codeword[40][2] = 0;
    codeword[41][0] = -M_PI / 2; codeword[41][1] = M_PI / 4; codeword[41][2] = 0;
    codeword[42][0] = -M_PI / 2; codeword[42][1] = -M_PI / 4; codeword[42][2] = 0;
    codeword[43][0] = -M_PI / 2; codeword[43][1] = -3 * M_PI / 4; codeword[43][2] = 0;
    codeword[44][0] = M_PI; codeword[44][1] = M_PI; codeword[44][2] = 0;
    codeword[45][0] = M_PI; codeword[45][1] = M_PI / 2; codeword[45][2] = 0;
    codeword[46][0] = M_PI; codeword[46][1] = 0; codeword[46][2] = 0;
    codeword[47][0] = M_PI; codeword[47][1] = -M_PI / 2; codeword[47][2] = 0;
    codeword[48][0] = 3 * M_PI / 4 + 0.4; codeword[48][1] = 3 * M_PI / 4; codeword[48][2] = -M_PI / 2;
    codeword[49][0] = 3 * M_PI / 4 + 0.4; codeword[49][1] = M_PI / 4; codeword[49][2] = -M_PI / 2;
    codeword[50][0] = 3 * M_PI / 4 + 0.4; codeword[50][1] = -M_PI / 4; codeword[50][2] = -M_PI / 2;
    codeword[51][0] = 3 * M_PI / 4 + 0.4; codeword[51][1] = -3 * M_PI / 4; codeword[51][2] = -M_PI / 2;
    codeword[52][0] = M_PI / 4 + 0.4; codeword[52][1] = M_PI; codeword[52][2] = -M_PI / 2;
    codeword[53][0] = M_PI / 4 + 0.4; codeword[53][1] = M_PI / 2; codeword[53][2] = -M_PI / 2;
    codeword[54][0] = M_PI / 4 + 0.4; codeword[54][1] = 0; codeword[54][2] = -M_PI / 2;
    codeword[55][0] = M_PI / 4 + 0.4; codeword[55][1] = -M_PI / 2; codeword[55][2] = -M_PI / 2;
    codeword[56][0] = -M_PI / 4 + 0.4; codeword[56][1] = 3 * M_PI / 4; codeword[56][2] = -M_PI / 2;
    codeword[57][0] = -M_PI / 4 + 0.4; codeword[57][1] = M_PI / 4; codeword[57][2] = -M_PI / 2;
    codeword[58][0] = -M_PI / 4 + 0.4; codeword[58][1] = -M_PI / 4; codeword[58][2] = -M_PI / 2;
    codeword[59][0] = -M_PI / 4 + 0.4; codeword[59][1] = -3 * M_PI / 4; codeword[59][2] = -M_PI / 2;
    codeword[60][0] = -3 * M_PI / 4 + 0.4; codeword[60][1] = M_PI; codeword[60][2] = -M_PI / 2;
    codeword[61][0] = -3 * M_PI / 4 + 0.4; codeword[61][1] = M_PI / 2; codeword[61][2] = -M_PI / 2;
    codeword[62][0] = -3 * M_PI / 4 + 0.4; codeword[62][1] = 0; codeword[62][2] = -M_PI / 2;
    codeword[63][0] = -3 * M_PI / 4 + 0.4; codeword[63][1] = -M_PI / 2; codeword[63][2] = -M_PI / 2;
}

// 受信符号と各符号語の距離を計算
double calculate_distance(double received[n], double codeword[n]) {
    double total_squared_difference = 0.0;

    for (int i = 0; i < n; i++) {
        double diff = received[i] - codeword[i];
        diff = fmod(diff + M_PI, 2 * M_PI);  // [-π, π) に変換
        if (diff < 0) diff += 2 * M_PI;
        diff -= M_PI;

        total_squared_difference += diff * diff;  // 2 乗
    }

    return sqrt(total_squared_difference);  // ユークリッド距離
}
```

```
//雑音付加
void add_laplace_noise(double signal[], double sigma, double received[]) {
    for (int i = 0; i < n; i++) {
        received[i] = signal[i] + gauss_random(sigma);
    }
}

// 復号：最も距離の近い符号語を選択
int decode(double received[n], double codeword[M_n][n]) {
    int best_index = 0;
    double min_distance = calculate_distance(received, codeword[0]);

    for (int i = 1; i < M_n; i++) {
        double dist = calculate_distance(received, codeword[i]);
        if (dist < min_distance) {
            min_distance = dist;
            best_index = i;
        }
    }

    return best_index;
}

//1 回のシミュレーション
int simulate(double sigma) {
    static int initialized = 0;
    static double codeword[M_n][n];

    // 符号語を生成
    if (!initialized) {
        sphere64R(codeword);
        initialized = 1;
    }

    // 符号語をランダムに 1 つ選択
    int transmitted_index = rand() % M_n;
    double transmitted_signal[n];
    for (int i = 0; i < n; i++) {
        transmitted_signal[i] = codeword[transmitted_index][i];
    }

    // 雑音を加えて受信信号を生成
    double received_signal[n];
    add_laplace_noise(transmitted_signal, sigma, received_signal);

    // トーラスの範囲 [-π, π] に正規化
    for (int i = 0; i < n; i++) {
        while (received_signal[i] > M_PI) received_signal[i] -= 2 * M_PI;
        while (received_signal[i] < -M_PI) received_signal[i] += 2 * M_PI;
    }

    // 復号して最も近い符号語を選択
    int decoded_index = decode(received_signal, codeword);

    // 符号語誤り：送信と復号が異なる
    return (decoded_index == transmitted_index) ? 0 : 1;
}


int main(void) {
    // 乱数シードを設定
    srand((unsigned int)time(NULL));

    // 結果を CSV ファイルに書き込む
    FILE* file = fopen("simulation_results.csv", "w");

    // CSV ファイルのヘッダを書き込み
    fprintf(file, "C_W,CER\n");
```

```
    // sigma を変化させる
    for (double sigma = 0.1; sigma <= 1.5; sigma += 0.05) {
        double C_W = calculate_channel_capacity_gauss(sigma);  // 通信路容量を計算
        int total_errors = 0;

        // N 回の通信シミュレーションを実行
        for (int i = 0; i < N; i++) {
            total_errors += simulate(sigma);
        }

        double cer = (double)total_errors / N;

        // 結果を表示
        printf("sigma = %.2f, C_W = %.4f, CER = %d / %d = %.8f\n", sigma, C_W, total_errors, N, cer);

        // CSV ファイルに出力
        fprintf(file, "%.4f,%.8f\n", C_W, cer);
    }

    fclose(file);  // ファイルを閉じる
    return 0;
}
```

C.4 四次元, 五次元, 八次元の球充填符号のシミュレーションプログラム

文献 [8],[9] を参考にしてプログラムを作成した.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "MT.h"

#define n 8  //ブロック長（任意に設定）
#define M_n 500000  //符号語数の最大
#define M_PI acos(-1)  //π
#define N 1000000  //試行回数（任意に設定）

// 一様乱数生成（0.0〜1.0）
double uniform_random() {
    return genrand_real1();
}

// 逆誤差関数の近似（文献 [9] を参考にした）
double erfinv(double x) {
    double a = 0.147;
    double ln_term = log(1.0 - x * x);
    double part1 = (2.0 / (M_PI * a)) + (ln_term / 2.0);
    double part2 = ln_term / a;

    double sign_x = (x > 0) - (x < 0);  // 符号関数
    return sign_x * sqrt(sqrt(part1 * part1 - part2) - part1);
}

//正規分布の乱数発生
double gauss_random(double sigma) {
    double u = uniform_random();

    return sqrt(2) * sigma * erfinv(erf(M_PI / (sqrt(2) * sigma)) * (2 * u - 1));
}

//通信路容量
double calculate_channel_capacity_gauss(double sigma) {
    return log2(2 * M_PI) + (sqrt(2 * M_PI) * erf(M_PI / (sqrt(2) * sigma))
    * sigma * log(exp(-(M_PI * M_PI) / (2 * sigma * sigma)) / (sqrt(2 * M_PI)
```

```
    * erf(M_PI / (sqrt(2) * sigma)) * sigma)) + M_PI * exp(-(M_PI * M_PI) /
    (2 * sigma * sigma)) - (sqrt(M_PI) * erf(M_PI / (sqrt(2) * sigma)) * sigma) /
    sqrt(2) + M_PI * M_PI * sqrt(M_PI) * erf(M_PI / (sqrt(2) * sigma)) / (sqrt(2) * sigma))
    / (sqrt(2 * M_PI) * log(2) * erf(M_PI / (sqrt(2) * sigma)) * sigma);
}

//D4 符号の生成
void d4_codewords(double codeword[M_n][n], int* actual_Mn) {
    int value[] = { -1, 0, 1, 2 };  //基本領域内の整数
    int value_count = sizeof(value) / sizeof(value[0]);

    int count = 0;

    for (int i1 = 0; i1 < value_count; i1++) {
        for (int i2 = 0; i2 < value_count; i2++) {
            for (int i3 = 0; i3 < value_count; i3++) {
                for (int i4 = 0; i4 < value_count; i4++) {
                    int x[4] = { value[i1], value[i2], value[i3], value[i4] };

                    int sum = 0;  //成分和
                    for (int i = 0; i < 4; i++) sum += x[i];
                 //D4 条件の確認
                    if (fabs(fmod(sum, 2)) == 0.0) {
                        for (int i = 0; i < 4; i++)
                            codeword[count][i] = (double)x[i] * (M_PI / 2); //単位円上の値に変換
                        count++;

                        if (count >= M_n) {
                            *actual_Mn = count;
                            return;
                        }
                    }
                }
            }
        }
    }

    *actual_Mn = count;
}

//D5 符号の生成
void d5_codewords(double codeword[M_n][n], int* actual_Mn) {
    int value[] = { -1, 0, 1, 2 };  //基本領域内の整数
    int value_count = sizeof(value) / sizeof(value[0]);

    int count = 0;

    for (int i1 = 0; i1 < value_count; i1++) {
        for (int i2 = 0; i2 < value_count; i2++) {
            for (int i3 = 0; i3 < value_count; i3++) {
                for (int i4 = 0; i4 < value_count; i4++) {
                    for (int i5 = 0; i5 < value_count; i5++) {
                        int x[5] = { value[i1], value[i2], value[i3], value[i4],  value[i5] };

                        int sum = 0;  //成分和
                        for (int i = 0; i < 5; i++) sum += x[i];
                     //D5 条件の確認
                        if (fabs(fmod(sum, 2)) == 0.0) {
                            for (int i = 0; i < 5; i++)
                                codeword[count][i] = (double)x[i] * (M_PI / 2);  //単位円上の値に変換
                            count++;

                            if (count >= M_n) {
                                *actual_Mn = count;
                                return;
                            }
                        }

                    }
```

```
                }
            }
        }
    }

    *actual_Mn = count;
}

//E8 符号の生成
void e8_codewords(double codeword[][n], int* actual_Mn) {
    // 整数座標
    int set1[] = { -1, 0, 1,2 };
    int num_values1 = sizeof(set1) / sizeof(set1[0]);

    // 半整数座標
    double set2[] = { -1.5,-0.5, 0.5,1.5 };
    int num_values2 = sizeof(set2) / sizeof(set2[0]);

    int count = 0;

    // 整数座標
    for (int i1 = 0; i1 < num_values1; i1++)
        for (int i2 = 0; i2 < num_values1; i2++)
            for (int i3 = 0; i3 < num_values1; i3++)
                for (int i4 = 0; i4 < num_values1; i4++)
                    for (int i5 = 0; i5 < num_values1; i5++)
                        for (int i6 = 0; i6 < num_values1; i6++)
                            for (int i7 = 0; i7 < num_values1; i7++)
                                for (int i8 = 0; i8 < num_values1; i8++) {
                                    int x[8] = { set1[i1], set1[i2], set1[i3], set1[i4],
                                                 set1[i5], set1[i6], set1[i7], set1[i8] };

                                    int sum = 0;  //成分和
                                    for (int i = 0; i < 8; i++) sum += x[i];

                                    // E8 条件の確認
                                    if (fabs(fmod(sum, 2.0)) == 0) {
                                        for (int i = 0; i < 8; i++)
                                            codeword[count][i] = (double)x[i] * (M_PI / 2);  //単位円上の
値に変換

                                        count++;
                                    }
                                }

    // 半整数座標
    for (int i1 = 0; i1 < num_values2; i1++)
        for (int i2 = 0; i2 < num_values2; i2++)
            for (int i3 = 0; i3 < num_values2; i3++)
                for (int i4 = 0; i4 < num_values2; i4++)
                    for (int i5 = 0; i5 < num_values2; i5++)
                        for (int i6 = 0; i6 < num_values2; i6++)
                            for (int i7 = 0; i7 < num_values2; i7++)
                                for (int i8 = 0; i8 < num_values2; i8++) {
                                    double x[8] = { set2[i1], set2[i2], set2[i3], set2[i4],
                                                    set2[i5], set2[i6], set2[i7], set2[i8] };

                                    double sum = 0.0;  //成分和
                                    for (int i = 0; i < 8; i++) sum += x[i];

                                    // E8 条件の確認
                                    if (fabs(fmod(sum, 2.0)) == 0) {
                                        for (int i = 0; i < 8; i++)
                                            codeword[count][i] = x[i] * (M_PI / 2);  //単位円上の値に変換
                                        count++;
                                    }
                                }

    *actual_Mn = count;
}
```

```
// 受信符号と各符号語の距離を計算
double calculate_distance(double received[n], double codeword[n]) {
    double total_squared_difference = 0.0;

    for (int i = 0; i < n; i++) {
        double diff = received[i] - codeword[i];
        diff = fmod(diff + M_PI, 2 * M_PI);  // [-π, π) に変換
        if (diff < 0) diff += 2 * M_PI;
        diff -= M_PI;

        total_squared_difference += diff * diff;  // 2 乗
    }

    return sqrt(total_squared_difference);  // ユークリッド距離
}


// 雑音付加
void add_noise(double signal[], double sigma, double received[]) {
    for (int i = 0; i < n; i++) {
        received[i] = signal[i] + gauss_random(sigma);
    }
}

// 復号（最も距離の近い符号語を選ぶ）
int decode(double received[n], double codeword[M_n][n], int Mn) {
    int best_index = 0;
    double min_distance = calculate_distance(received, codeword[0]);

    for (int i = 1; i < Mn; i++) {
        double dist = calculate_distance(received, codeword[i]);
        if (dist < min_distance) {
            min_distance = dist;
            best_index = i;
        }
    }

    return best_index;
}

// 通信 1 回分のシミュレーション
int simulate(double sigma, double codeword[M_n][n], int Mn) {
    int transmitted_index = genrand_int32() % Mn;

    double transmitted_signal[n];
    for (int i = 0; i < n; i++) {
        transmitted_signal[i] = codeword[transmitted_index][i];
    }

    double received_signal[n];
    add_noise(transmitted_signal, sigma, received_signal);

    // トーラス [-π, π] に収める
    for (int i = 0; i < n; i++) {
        while (received_signal[i] > M_PI) received_signal[i] -= 2 * M_PI;
        while (received_signal[i] < -M_PI) received_signal[i] += 2 * M_PI;
    }

    int decoded_index = decode(received_signal, codeword, Mn);
    return (decoded_index == transmitted_index) ? 0 : 1;
}

int main(void) {
    //乱数の初期化
    init_genrand((unsigned)time(NULL));

    //符号語配列を動的確保
    double(*codeword)[n] = malloc(sizeof(double[n]) * M_n);
```

```
    if (codeword == NULL) {
        fprintf(stderr, "メモリ確保に失敗しました。\n");
        return 1;
    }

    int Mn = 0;

    //符号語生成（シミュレーションによって随時変更）
    e8_codewords(codeword, &Mn);

     // 結果を CSV ファイルに書き込む
    FILE* file = fopen("simulation_results.csv", "w");

    // CSV ファイルのヘッダを書き込み
    fprintf(file, "C_W,CER\n");

    //  sigma を変化させながらシミュレーション
    for (double sigma = 0.1; sigma <= 0.5; sigma += 0.05) {
        double C_W = calculate_channel_capacity_gauss(sigma);  //通信路容量の計算
        int error_count = 0; //誤り回数をカウント

       //N 回のシミュレーション
        for (int i = 0; i < N; i++) {
            int result = simulate(sigma, codeword, Mn);
            if (result < 0) {
                fprintf(stderr, "simulate 内でエラーが発生\n");
                fclose(file);
                free(codeword);
                return 1;
            }
            error_count += result;
        }

        double cer = (double)error_count / N;

        // 結果を表示
        printf("sigma = %.2f, C_W = %.4f, CER = %d / %d = %.8f\n", sigma, C_W, error_count, N, cer);
        // CSV ファイルに出力
        fprintf(file, "%.4f,%.8f\n", C_W, cer);
    }

    //ファイルを閉じてメモリ解放
    fclose(file);
    free(codeword);
    return 0;
}
```

C.5　六次元, 七次元の球充填符号のシミュレーションプログラム

文献 [8],[9] を参考にしてプログラムを作成した.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "MT.h"

#define n 8  //ブロック長（任意に設定）
#define M_n 2000000  //符号語数の最大
#define M_PI acos(-1) //π
#define N 1000000 //試行回数（任意に設定）
#define L 7  //射影後のブロック長

// 一様乱数生成（0.0〜1.0）
double uniform_random() {
```

```
    return genrand_real1();
}

// 逆誤差関数の近似（文献 [9] を参考にした）
double erfinv(double x) {
    double a = 0.147;
    double ln_term = log(1.0 - x * x);
    double part1 = (2.0 / (M_PI * a)) + (ln_term / 2.0);
    double part2 = ln_term / a;

    double sign_x = (x > 0) - (x < 0);  // 符号関数
    return sign_x * sqrt(sqrt(part1 * part1 - part2) - part1);
}

//正規分布の乱数発生
double gauss_random(double sigma) {
    double u = uniform_random();

    return sqrt(2) * sigma * erfinv(erf(M_PI / (sqrt(2) * sigma)) * (2 * u - 1));
}

//通信路容量
double calculate_channel_capacity_gauss(double sigma) {
    return log2(2 * M_PI) + (sqrt(2 * M_PI) * erf(M_PI / (sqrt(2) * sigma))
    * sigma * log(exp(-(M_PI * M_PI) / (2 * sigma * sigma)) / (sqrt(2 * M_PI)
    * erf(M_PI / (sqrt(2) * sigma)) * sigma)) + M_PI * exp(-(M_PI * M_PI) /
    (2 * sigma * sigma)) - (sqrt(M_PI) * erf(M_PI / (sqrt(2) * sigma)) * sigma) /
    sqrt(2) + M_PI * M_PI * sqrt(M_PI) * erf(M_PI / (sqrt(2) * sigma)) / (sqrt(2) * sigma))
    / (sqrt(2 * M_PI) * log(2) * erf(M_PI / (sqrt(2) * sigma)) * sigma);
}

//E7 符号の生成
void e7_codewords(double codeword[][n], int* actual_Mn) {
    // 整数座標
    double set1[] = {  -2, -1, 0, 1 ,2, 3 };
    int num_values1 = sizeof(set1) / sizeof(set1[0]);

    // 半整数座標
    double set2[] = { -2.5, -1.5, -0.5, 0.5, 1.5, 2.5 };
    int num_values2 = sizeof(set2) / sizeof(set2[0]);

    int count = 0;

    // 整数座標
    for (int i1 = 0; i1 < num_values1; i1++)
        for (int i2 = 0; i2 < num_values1; i2++)
            for (int i3 = 0; i3 < num_values1; i3++)
                for (int i4 = 0; i4 < num_values1; i4++)
                    for (int i5 = 0; i5 < num_values1; i5++)
                        for (int i6 = 0; i6 < num_values1; i6++)
                            for (int i7 = 0; i7 < num_values1; i7++)
                                for (int i8 = 0; i8 < num_values1; i8++) {
                                    double x[8] = { set1[i1], set1[i2], set1[i3], set1[i4],
                                                    set1[i5], set1[i6], set1[i7], set1[i8] };

                                    double sum = 0; //成分和
                                    for (int i = 0; i < 8; i++) sum += x[i];

                                    // E8 条件の確認，成分和が偶数
                                    if (fabs(fmod(sum, 2.0)) == 0) {
                                        // E7 条件の確認，成分和が 0
                                        if (sum == 0) {
                                            for (int i = 0; i < 8; i++)
                                                codeword[count][i] = (double)x[i];
                                            count++;
                                        }
                                    }
                                }
```

```
    // 半整数座標
    for (int i1 = 0; i1 < num_values2; i1++)
        for (int i2 = 0; i2 < num_values2; i2++)
            for (int i3 = 0; i3 < num_values2; i3++)
                for (int i4 = 0; i4 < num_values2; i4++)
                    for (int i5 = 0; i5 < num_values2; i5++)
                        for (int i6 = 0; i6 < num_values2; i6++)
                            for (int i7 = 0; i7 < num_values2; i7++)
                                for (int i8 = 0; i8 < num_values2; i8++) {
                                    double x[8] = { set2[i1], set2[i2], set2[i3], set2[i4],
                                                    set2[i5], set2[i6], set2[i7], set2[i8] };

                                    double sum = 0.0; //成分和
                                    for (int i = 0; i < 8; i++) sum += x[i];

                                    // E8 条件の確認，成分和が偶数
                                    if (fabs(fmod(sum, 2.0)) == 0) {
                                        // E7 条件の確認，成分和が 0
                                        if (sum == 0) {
                                            for (int i = 0; i < 8; i++)
                                                codeword[count][i] = x[i];
                                            count++;
                                        }
                                    }
                                }

    *actual_Mn = count;
}

// 直交基底（行ベクトル：e1〜e7）
const double basis7[L][8] = {
    { 0.5,  0.5,  0.5,  0.5, -0.5, -0.5, -0.5, -0.5 },
    { 0.5,  0.5, -0.5, -0.5,  0.5,  0.5, -0.5, -0.5 },
    { 0.5, -0.5,  0.5, -0.5,  0.5, -0.5,  0.5, -0.5 },
    { 0.5, -0.5, -0.5,  0.5, -0.5,  0.5,  0.5, -0.5 },
    { 0.5, -0.5,  0.5, -0.5, -0.5,  0.5, -0.5,  0.5 },
    { 0.5,  0.5, -0.5, -0.5, -0.5, -0.5,  0.5,  0.5 },
    { 0.5, -0.5, -0.5,  0.5,  0.5, -0.5, -0.5,  0.5 }
};

//E6 符号の生成
void e6_codewords(double codeword[][n], int* actual_Mn) {
    // 整数座標
    double set1[] = { -1, 0, 1, 2 };
    double set1_last[] = { 0, 1 };         // x6, x7, x8 用
    // 半整数座標
    double set2[] = {  -1.5, -0.5, 0.5, 1.5 };
    double set2_last[] = { -0.5, 0.5  };    // x6, x7, x8 用

    int num_values1 = sizeof(set1) / sizeof(set1[0]);
    int num_values1_last = sizeof(set1_last) / sizeof(set1_last[0]);
    int num_values2 = sizeof(set2) / sizeof(set2[0]);
    int num_values2_last = sizeof(set2_last) / sizeof(set2_last[0]);

    int count = 0;

    // 整数座標
    for (int i1 = 0; i1 < num_values1; i1++)
        for (int i2 = 0; i2 < num_values1; i2++)
            for (int i3 = 0; i3 < num_values1; i3++)
                for (int i4 = 0; i4 < num_values1; i4++)
                    for (int i5 = 0; i5 < num_values1; i5++)
                        for (int i6 = 0; i6 < num_values1_last; i6++)
                            for (int i7 = 0; i7 < num_values1_last; i7++)
                                for (int i8 = 0; i8 < num_values1_last; i8++) {
                                    double x[8] = {
                                        set1[i1], set1[i2], set1[i3], set1[i4],
                                        set1[i5], set1_last[i6], set1_last[i7], set1_last[i8]
                                    };
```

```
                                    double sum = 0;  //成分和
                                    for (int i = 0; i < 8; i++) sum += x[i];

                                  // E8 条件の確認，成分和が偶数
                                    if (fabs(fmod(sum, 2.0)) == 0) {
                                       // E6 条件の確認，x6，x7，x8 が同じか
                                        if (x[5] == x[6] && x[6] == x[7]) {
                                            if (count >= M_n) {
                                                printf("Error: codeword array exceeded M_n = %d\n", M_n);
                                                exit(1);
                                            }
                                            for (int i = 0; i < 8; i++)
                                                codeword[count][i] = x[i];
                                            count++;
                                        }
                                    }
                                }

    // 半整数座標
    for (int i1 = 0; i1 < num_values2; i1++)
        for (int i2 = 0; i2 < num_values2; i2++)
            for (int i3 = 0; i3 < num_values2; i3++)
                for (int i4 = 0; i4 < num_values2; i4++)
                    for (int i5 = 0; i5 < num_values2; i5++)
                        for (int i6 = 0; i6 < num_values2_last; i6++)
                            for (int i7 = 0; i7 < num_values2_last; i7++)
                                for (int i8 = 0; i8 < num_values2_last; i8++) {
                                    double x[8] = {
                                        set2[i1], set2[i2], set2[i3], set2[i4],
                                        set2[i5], set2_last[i6], set2_last[i7], set2_last[i8]
                                    };

                                    double sum = 0.0;  //成分和
                                    for (int i = 0; i < 8; i++) sum += x[i];

                                  // E8 条件の確認，成分和が偶数
                                    if (fabs(fmod(sum, 2.0)) == 0) {
                                       // E6 条件の確認，x6，x7，x8 が同じか
                                        if (x[5] == x[6] && x[6] == x[7]) {
                                            if (count >= M_n) {
                                                printf("Error: codeword array exceeded M_n = %d\n", M_n);
                                                exit(1);
                                            }
                                            for (int i = 0; i < 8; i++)
                                                codeword[count][i] = x[i];
                                            count++;
                                        }
                                    }
                                }

    *actual_Mn = count;
}

// 直交基底（行ベクトル：e1〜e6）
const double basis6[L][8] = {
     { 0, 0, 0, 0, 0, 2, 2, 2 },
     { 2, 0, 0, 0, 0, 0, 0, 0 },
     { 0, 2, 0, 0, 0, 0, 0, 0 },
     { 0, 0, 2, 0, 0, 0, 0, 0 },
     { 0, 0, 0, 2, 0, 0, 0, 0 },
     { 0, 0, 0, 0, 2, 0, 0, 0 } };

// 内積を求める関数
double dot(const double* a, const double* b, int A) {
    double sum = 0.0;
    for (int i = 0; i < n; i++) sum += a[i] * b[i];
    return sum;
}
```

```
//7 次元ベクトルへの射影
int vector_transformation7(double codeword[][n], double codeword_proj[][L], int num_codewords) {
    // 一時確保
    double(*temp)[L] = malloc(sizeof(double[L]) * num_codewords);
    if (!temp) {
        fprintf(stderr, "malloc failed in vector_transformation\n");
        return -1;
    }

    // 変換処理
    for (int i = 0; i < num_codewords; i++) {
        for (int j = 0; j < L; j++) {
            double val = dot(codeword[i], basis7[j], n) * (M_PI / 3); //単位円上の値に変換

            // [-π, π) に収める
            val = fmod(val + M_PI, 2.0 * M_PI);
            if (val < 0) val += 2.0 * M_PI;
            val -= M_PI;

            temp[i][j] = val;
        }
    }

    // 重複削除
    int new_count = 0;
    for (int i = 0; i < num_codewords; i++) {
        int duplicate = 0;
        for (int k = 0; k < new_count; k++) {
            int same = 1;
            for (int j = 0; j < L; j++) {
                if (fabs(temp[i][j] - codeword_proj[k][j]) > 1e-9) { // 誤差を許容
                    same = 0;
                    break;
                }
            }
            if (same) {
                duplicate = 1;
                break;
            }
        }
        if (!duplicate) {
            for (int j = 0; j < L; j++) {
                codeword_proj[new_count][j] = temp[i][j];
            }
            new_count++;
        }
    }

    free(temp);
    return new_count;  // 最終的な行数を返す
}

//6 次元ベクトルへの射影
int vector_transformation6(double codeword[][n], double codeword_proj[][L], int num_codewords) {
    // 一時確保
    double(*temp)[L] = malloc(sizeof(double[L]) * num_codewords);
    if (!temp) {
        fprintf(stderr, "malloc failed in vector_transformation\n");
        return -1;
    }
    double val;

    // 変換処理
    for (int i = 0; i < num_codewords; i++) {
        for (int j = 0; j < L; j++) {
            if (j == 0) {
                val = dot(codeword[i], basis6[j], n) * (M_PI / 6);  //単位円上の値に変換
            }
```

```
            else {
                val = dot(codeword[i], basis6[j], n) * (M_PI / 4);  //単位円上の値に変換
            }
            // [-π, π) に収める
            val = fmod(val + M_PI, 2.0 * M_PI);
            if (val < 0) val += 2.0 * M_PI;
            val -= M_PI;

            temp[i][j] = val;
        }
    }

    // 重複削除
    int new_count = 0;
    for (int i = 0; i < num_codewords; i++) {
        int duplicate = 0;
        for (int k = 0; k < new_count; k++) {
            int same = 1;
            for (int j = 0; j < L; j++) {
                if (fabs(temp[i][j] - codeword_proj[k][j]) > 1e-9) { // 誤差を許容
                    same = 0;
                    break;
                }
            }
            if (same) {
                duplicate = 1;
                break;
            }
        }
        if (!duplicate) {
            for (int j = 0; j < L; j++) {
                codeword_proj[new_count][j] = temp[i][j];
            }
            new_count++;
        }
    }

    free(temp);
    return new_count;  // 最終的な行数を返す
}

// 受信符号と各符号語の距離を計算
double calculate_distance(double received[n], double codeword[n]) {
    double total_squared_difference = 0.0;

    for (int i = 0; i < n; i++) {
        double diff = received[i] - codeword[i];
        diff = fmod(diff + M_PI, 2 * M_PI);  // [-π, π) に変換
        if (diff < 0) diff += 2 * M_PI;
        diff -= M_PI;

        total_squared_difference += diff * diff;  // 2 乗
    }

    return sqrt(total_squared_difference);  // ユークリッド距離
}

//雑音付加
void add_laplace_noise(double signal[], double sigma, double received[]) {
    for (int i = 0; i < n; i++) {
        received[i] = signal[i] + gauss_random(sigma);
    }
}

// 復号：最も距離の近い符号語を選択
int decode(double received[n], double codeword[M_n][n]) {
    int best_index = 0;
    double min_distance = calculate_distance(received, codeword[0]);
```

```
    for (int i = 1; i < M_n; i++) {
        double dist = calculate_distance(received, codeword[i]);
        if (dist < min_distance) {
            min_distance = dist;
            best_index = i;
        }
    }

    return best_index;
}

// 通信 1 回分のシミュレーション
int simulate(double sigma, double codeword_proj[][L], int Mn) {
    int transmitted_index = genrand_int32() % Mn;

    double transmitted_signal[L];
    for (int i = 0; i < L; i++) {
        transmitted_signal[i] = codeword_proj[transmitted_index][i];
    }

    double received_signal[L];
    add_noise(transmitted_signal, sigma, received_signal);

    // トーラス [-π, π] に収める
    for (int i = 0; i < L; i++) {
        while (received_signal[i] > M_PI) received_signal[i] -= 2 * M_PI;
        while (received_signal[i] < -M_PI) received_signal[i] += 2 * M_PI;
    }

    int decoded_index = decode(received_signal, codeword_proj, Mn);
    return (decoded_index == transmitted_index) ? 0 : 1;
}

int main(void) {
    //乱数の初期化
    init_genrand((unsigned)time(NULL));

    // 8 次元符号語の動的確保
    double(*codeword)[n] = malloc(sizeof(double[n]) * M_n);

    // 射影後の符号語の動的確保
    double(*codeword_proj)[L] = malloc(sizeof(double[L]) * M_n);

    int Mn = 0;
    //符号語生成（シミュレーションによって随時変更）
    e6_codewords(codeword, &Mn);
    printf("8 次元符号語生成数: %d\n", Mn);

    // 射影 + π処理 + 重複削除（シミュレーションによって随時変更）
    int Mn_ld = vector_transformation6(codeword, codeword_proj, Mn);
    printf("射影後符号語（重複削除後）数: %d\n", Mn_ld);

    // 結果を CSV ファイルに書き込む
    FILE* file = fopen("simulation_results.csv", "w");

    // CSV ファイルのヘッダを書き込み
    fprintf(file, "C_W,CER\n");

    //  sigma を変化させながらシミュレーション
    for (double sigma = 0.1; sigma <= 0.5; sigma += 0.05) {
        double C_W = calculate_channel_capacity_gauss(sigma);  //通信路容量の計算
        int error_count = 0; //誤り回数をカウント

        //N 回のシミュレーション
        for (int i = 0; i < N; i++) {
            int result = simulate(sigma, codeword_proj, Mn_ld);
            if (result < 0) {
                fprintf(stderr, "simulate 内でエラーが発生\n");
                fclose(file);
```

```
                free(codeword);
                free(codeword_proj);
                return 1;
            }
            error_count += result;
        }

        double cer = (double)error_count / N;
        //結果を表示
        printf("sigma = %.2f, C_W = %.4f, CER = %d / %d = %.8f\n", sigma, C_W, error_count, N, cer);
        // CSV ファイルに出力
        fprintf(file, "%.4f,%.8f\n", C_W, cer);
    }

    //ファイルを閉じてメモリ解放
    fclose(file);
    free(codeword);
    free(codeword_proj);

    return 0;
}
```