数理工学実験レポート

第 4 章（モンテカルロシミュレーション）

学籍番号 1029366161　中塚一瑳

2025 年 12 月 9 日

概要

目次

はじめに

今回はモンテカルロ法を用いた数値計算手法について学ぶ。伝染モデルや円周率のモンテカルロ法シミュレーションを行い、さらにイジングモデルに対して熱浴法とメトロポリス法を実装し、その性能を比較する。問題の条件を満たしながら、python で効率的なコードを実装することを目指す。

1　課題 1：疫病の確率的な伝染モデル

1.1　原理と方法

本課題では，配布資料 [1] に記載の一次元確率的伝染モデルに従い，格子点 $i = 1, \ldots, 64$ に状態変数 $s_i(t) \in \{0, 1\}$ を配置した。左右近傍の感染者数を

$$n_i(t) = s_{i-1}(t) + s_{i+1}(t)$$

とし，次時刻の状態は資料式 (5.2) に従う確率 $P(1 \mid s_i(t), n_i(t))$ に基づいて決定する [1]。周期境界条件は

$$s_0(t) = s_{64}(t), \qquad s_{65}(t) = s_1(t)$$

により与えた。

モンテカルロシミュレーションでは，時刻 t の状態列 $\{s_i(t)\}$ を初期状態 $t = 0$ から順に確率的に生成する。各格子点で一様乱数 $r \in [0, 1]$ を生成し，

$$s_i(t+1) = \begin{cases} 1 & (r \leq P(1 \mid s_i(t), n_i(t))) \\ 0 & (r > P(1 \mid s_i(t), n_i(t))) \end{cases}$$

により次状態を決定した。

時刻 t の病人数は

$$m(t) = \sum_{i=1}^{64} s_i(t)$$

で定義し，独立な M 個のサンプルに対してサンプル平均

$$\overline{m(t)} = \frac{1}{M} \sum_{k=1}^{M} m^{(k)}(t)$$

を計算することで，期待値の近似を得た。

本課題では，初期条件として中央のみ感染（$s_{32}(0) = 1$）とし，シミュレーションの最終時刻は $T = 256$，サンプル数は M とした。

1.2　実装上の工夫

Python と NumPy を用いて実装を行った。Python で各格子点を for ループで逐次更新すると計算が極端に遅くなるため，本研究では全サイトをベクトル化により一括更新する方法を採用した。

まず，M 個の試行を同時に扱うために，状態を

$$s(t) \in \{0, 1\}^{M \times 64}$$

という 2 次元配列として保持した。周期境界条件に基づく左右近傍は NumPy の `roll` により

$$\texttt{right} = \texttt{roll}(s, -1), \qquad \texttt{left} = \texttt{roll}(s, 1)$$

として同時に生成し，$n(t) = \texttt{right} + \texttt{left}$ を一括して計算した。

資料式 (5.2) の遷移確率 $P(1 \mid s, n)$ は 6 通りの組 $(s, n) = (0,0), (0,1), (0,2), (1,0), (1,1), (1,2)$ に対応する値をあらかじめ一次元配列

$$\texttt{vals} = (\alpha_{00}, \alpha_{01}, \alpha_{02}, \alpha_{10}, \alpha_{11}, \alpha_{12})$$

として保持した。各格子点の組 (s_i, n_i) は

$$\texttt{index} = 3s + n$$

により 0–5 の整数に写像され,

$$\alpha = \texttt{vals[index]}$$

として全要素の遷移確率を分岐なしで取得した。

次状態の決定も同様にベクトル化し，乱数配列 $r \in [0,1]^{M \times 64}$ を生成して

$$s(t+1) = (r < \alpha)$$

という比較演算により全要素を同時更新した。病人数は `sum(axis=1)` を用いて

$$m^{(k)}(t) = \sum_i s_i^{(k)}(t)$$

を一括計算した。

以上のベクトル化により，明示的な for 文を用いず高速に M 試行 $\times$ 256 ステップのシミュレーションを実行できた。

1.3 小問 1

1.3.1 結果

パラメータ $p = 0.7$，サンプル数 $M = 10$ に対してシミュレーションを実行し, 各サンプルに対する 各時刻での m(t) を同時にプロットした（図 1）.

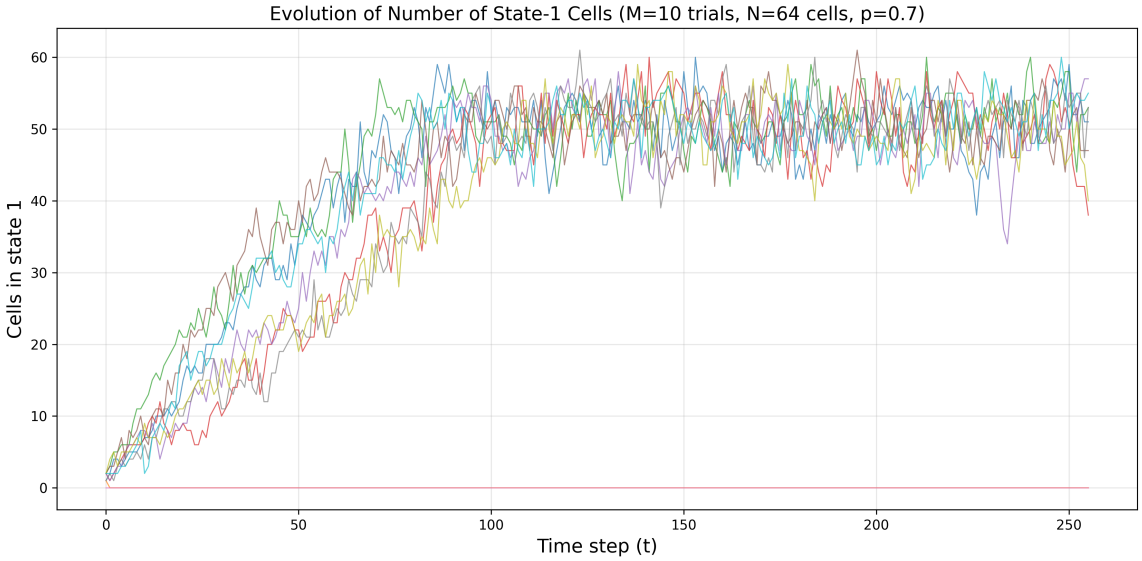

図 1: 各サンプルに対する 各時刻での $\mathrm{m(t)}$（$p = 0.7,\ M = 10$）

1.3.2 考察

図 1 に示すように, 各サンプルに対する 各時刻での $\mathrm{m(t)}$ は, 時間とともに増減を繰り返しながら増加し, 最終的には 50 前後を上下するようになっている. これは, 感染が広がるにつれて感染者数が増加する一方で, 回復によって感染者数が減少するつり合いの結果であると考えられる. また, ごく一部のサンプルでは, 開始直後に感染者が 0 となり, そのまま感染が広がらない場合も見られた. これは, 感染者が 0 となった時点で, その後の感染拡大が起こり得ないためであると考えられる.

1.4 小問 2

1.4.1 結果

パラメータ $p = 0.62, 0.625, 0.63, 0.635, 0.64, 0.645, 0.65, 0.66, 0.68, 0.7$, サンプル数 $M = 10000$ に対してシミュレーションを実行し, 各時刻での $\mathrm{m(t)}$ のサンプル平均をプロットした（図 2）.

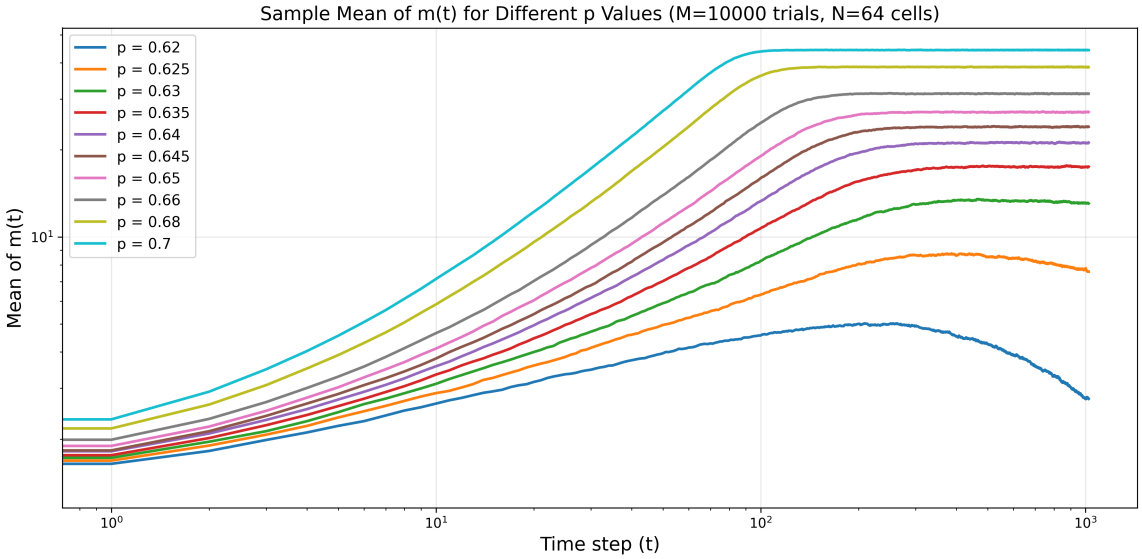

図 2: 各時刻での m(t) のサンプル平均（$p = 0.62, 0.625, 0.63, 0.635, 0.64, 0.645, 0.65, 0.66, 0.68, 0.7$, $M = 10000$）

log

1.4.2 考察

図 2 に示すように, 各時刻での m(t) のサンプル平均は, 時間とともに増加し, 最終的には一定の値に収束する p の値と、増加した後に減少する p の値が存在した. 前者は感染が広がり続ける場合, 後者は感染が収束する場合であると考えられる. 後者の p の範囲は図 2 からは明確には特定できないが, およそ 0.63 以下では明らかに減少していると考えられる. また, パラメータ p が大きくなるにつれて, 最終的な感染者数の最大値も増加する傾向が見られた. これは, 感染確率が高いほど感染が広がりやすくなり, より多くの人々が感染するためであると考えられる. また、問題では最終時刻 1024 としているが，T=100000 までシミュレーションを行った結果が以下の図である。

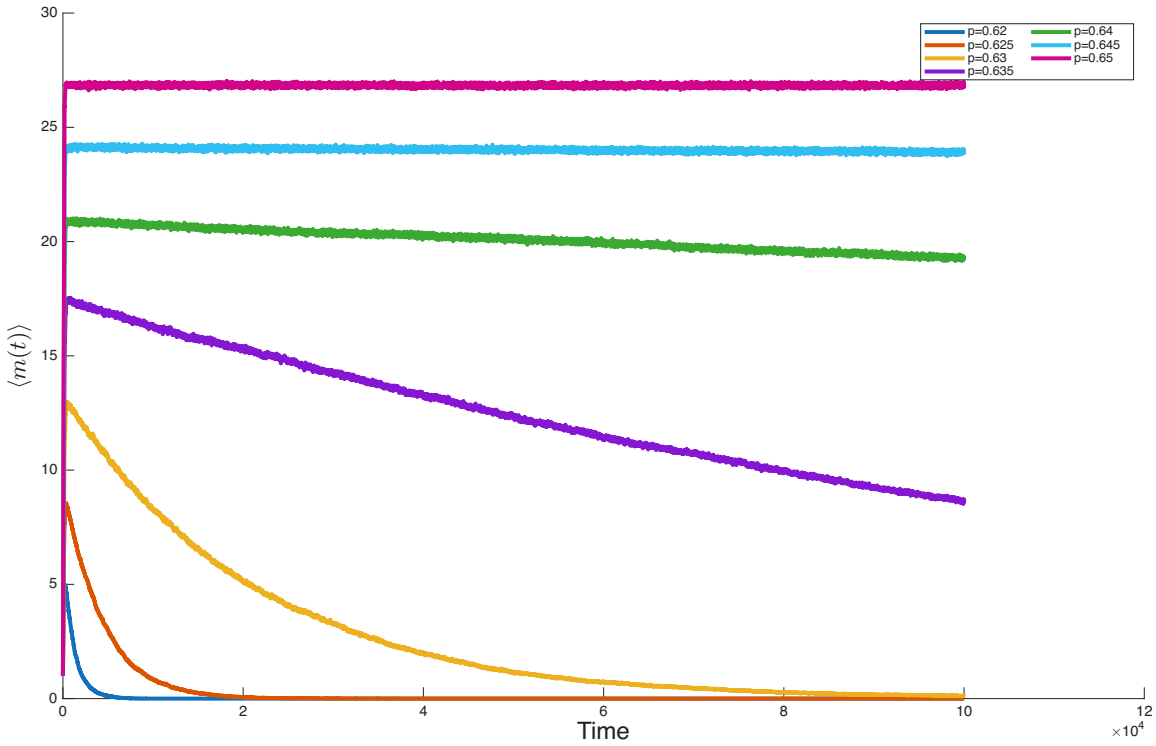

図 3: 各時刻での m(t) のサンプル平均（$p = 0.62, 0.625, 0.63, 0.635, 0.64, 0.645, 0.65$, $M = 10000$, $T = 100000$）

図 3 が示すのは、$p = 0.65$ では一旦は増加した感染者数が一定の値を保っているのに対し、$p < 0.65$ では一旦は増加した感染者数が減少する方向に向かう。これは、$0.645 < p < 0.650$ が臨界点であり、これを境に感染が広がるか収束するかが決定されるためであると考えられる。図 2 において $p = 0.64, 0.63$ が一見収束してるように見えたものの、長時間シミュレーションを行うことで、これらも最終的には減少に向かうことが確認できた。

2　課題 2：モンテカルロ法による円周率推定

2.1　原理と方法

本節では，モンテカルロ法を用いて円周率 π を推定する原理と，実際に用いる計算手順をまとめる [1].

まず，次の定積分が成り立つことを用いる [1].

$$\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x = \pi. \tag{1}$$

区間 $[0,1]$ 上の一様分布に従う確率変数 X を考えると，その確率密度関数は

$$p_X(x) = \begin{cases} 1 & (0 \le x \le 1), \\ 0 & (\text{それ以外}) \end{cases}$$

である．このとき，任意の可積分関数 $g(x)$ に対して

$$\mathbb{E}[g(X)] = \int_0^1 g(x)\,p_X(x)\,\mathrm{d}x = \int_0^1 g(x)\,\mathrm{d}x$$

が成り立つ．$g(x) = 4/(1+x^2)$ とおけば，式 (1) は

$$\pi = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x = \mathbb{E}\left[\frac{4}{1+X^2}\right] \tag{2}$$

と書き換えられる．したがって，π は確率変数 $Y = 4/(1+X^2)$ の期待値として表される．

式 (2) より，モンテカルロ法では

$$\hat{\pi}_M = \frac{1}{M}\sum_{k=1}^{M}\frac{4}{1+X_k^2}$$

を計算することで π の推定値 $\hat{\pi}_M$ を得る [1]。ここで X_k は $U[0,1]$ に従う独立な乱数である。本実験ではさらに，推定値の精度を評価するために，分割平均（blocking）に基づく分散推定を用いる。サンプル列を N_d 個のブロックに分割し，各ブロックに対して

$$\bar{f}_i = \frac{1}{N_{d,i}}\sum_{x\in\text{block }i}\frac{4}{1+x^2}$$

を計算する。これらのブロック平均の標本分散を用いると，推定値 $\hat{\pi}_M$ の分散は

$$\widehat{\mathrm{Var}}(\hat{\pi}_M) = \frac{1}{N_d(N_d-1)}\sum_{i=1}^{N_d}(\bar{f}_i - \hat{f})^2$$

と推定される。これはブロック平均が互いに独立とみなせる場合に成り立つ標準的な不偏分散推定式であり，推定量 $\hat{\pi}_M$ の標準誤差を近似的に与える。

2.2 実装上の工夫

本実験ではサンプル数 M を $10, 10^3, 10^5, 10^7$ と増加させながら，式 (5.10) に基づくモンテカルロ積分を行う．計算負荷が大きくなるため，以下の点に留意して実装した．

- **完全ベクトル化による高速化** Python の for ループは極端に遅いため，乱数生成から積分評価
$$\frac{4}{1+X^2}$$
の計算までを NumPy の配列演算として一度に実行した．これにより Python レベルのループを排除し，C 実装による高速な要素演算が可能になる．
- **形状 (10, M) の乱数生成** 10 回の独立試行を行うため，毎回乱数を生成するのではなく，あらかじめ形状 $(10, M)$ の乱数行列を生成した．これにより，乱数生成コストを 1 回にまとめつつ，同じ処理を 10 行に対して並列に実行できる．
- **in-place 演算の利用** NumPy の in-place 演算（例：`s *= s`, `np.reciprocal(out=s)`）を積極的に利用することで，中間配列の生成を避け，メモリ消費とメモリアロケーション時間を削減した．特に $M = 10^7$ のような大規模計算では，配列の再生成がボトルネックになるため有効である．
- **標本平均の同時計算** 10 行分の計算結果に対して，NumPy の `mean(axis=1)` を用いて一度に標本平均を求めることで，試行ごとのループ処理を回避した．

上記の工夫により，大規模サンプルに対しても純粋 Python 実装と比べて大幅に高速な実行が可能となった．

2.3　結果

サンプル数 $M = 10, 10^3, 10^5, 10^7$ について，各 M につき 10 回の独立試行を行い，推定値 $\hat{\pi}_M$ と精度（標準誤差推定値）を求めた。結果の平均値を表 1 にまとめる。サンプル数が増えるにつれて推

表 1: π の推定結果と精度（標準誤差推定値）

サンプル数 M	$\hat{\pi}_M$（平均）	精度（標準誤差）
10	3.2023	1.613×10^{-1}
10^3	3.1282	2.046×10^{-2}
10^5	3.1410	2.032×10^{-3}
10^7	3.14165	2.033×10^{-4}

定値は理論値 $\pi = 3.141592\ldots$ に収束し，標準誤差は $M^{-1/2}$ に比例して減少した。

2.3.1　考察

表 1 の結果より，推定値 $\hat{\pi}_M$ はサンプル数の増加とともに理論値に近づいており，モンテカルロ法が大数の法則に従って収束していることが確認できる。また，精度（標準誤差）については，解析的に求めた $f(X) = 4/(1 + X^2)$ の分散

$$\mathrm{Var}(f(X)) = 2\pi + 4 - \pi^2 \approx 0.41358$$

を用いると，推定量の理論的標準誤差は

$$\sqrt{\frac{\mathrm{Var}(f(X))}{M}} \approx \frac{0.64294}{M^{1/2}} = \frac{2.03367}{\sqrt{10M}}$$

で与えられる。これは $M^{-1/2}$ に比例して減少する。実験で得られた標準誤差もこの理論値と一致しており，特に $M \geq 10^3$ では誤差が 1% 未満の精度で一致した。したがって，式（5.9）で与えられるブロック平均に基づく分散推定が妥当であることが確認できる。さらに，$M = 10^7$ の場合には $\hat{\pi}_M = 3.14165$ と理論値との差が 6×10^{-5} 程度となり，モンテカルロ法としては十分な精度が得られた。これは、サンプル数を増やせば精度が向上する一方，収束速度が $M^{-1/2}$ に制限されるという，モンテカルロ法の特性を示している。

3　課題 3：熱よく法とメトロポリス法のつり合い条件の証明

3.1　1. メトロポリス法が詳細つり合い条件を満たすことの証明

メトロポリス法では，現状態 X から次状態候補 Y を対称な確率分布 $Q(X, Y) = Q(Y, X)$ に従って提案し，受理確率

$$A(X \to Y) = \min\left(1, \frac{W(Y)}{W(X)}\right)$$

で遷移する [1]．したがって遷移確率は

$$P(X \to Y) = Q(X, Y)A(X \to Y)$$

である．

詳細つり合い条件

$$P(X \to Y)W(X) = P(Y \to X)W(Y) \tag{5.19}$$

を示すため，$W(Y) \geq W(X)$ と $W(Y) < W(X)$ の場合に分ける．

■(i) $W(Y) \geq W(X)$ **の場合**　このとき

$$A(X \to Y) = 1, \qquad A(Y \to X) = \frac{W(X)}{W(Y)}.$$

よって

$$P(X \to Y)W(X) = Q(X,Y) \cdot 1 \cdot W(X),$$

$$P(Y \to X)W(Y) = Q(Y,X)\frac{W(X)}{W(Y)}W(Y) = Q(X,Y)W(X),$$

となり (5.19) が成り立つ．

■(ii) $W(Y) < W(X)$ **の場合**　このとき

$$A(X \to Y) = \frac{W(Y)}{W(X)}, \qquad A(Y \to X) = 1.$$

したがって

$$P(X \to Y)W(X) = Q(X,Y)\frac{W(Y)}{W(X)}W(X) = Q(X,Y)W(Y),$$

$$P(Y \to X)W(Y) = Q(Y,X) \cdot 1 \cdot W(Y) = Q(X,Y)W(Y),$$

となる．

以上より任意の X, Y について

$$P(X \to Y)W(X) = P(Y \to X)W(Y)$$

が成立し，メトロポリス法は詳細つり合い条件を満たす．

3.2　2. 熱浴法が詳細つり合い条件を満たすことの証明

状態を $X = (x_1, x_2, \ldots)$ とし，時刻 t において添字 i が状態に依存しない確率 p_i で選ばれるとする [1]．熱浴更新では i 番目の成分だけを更新し，それ以外は不変である．

X と Y が 1 箇所だけ異なる場合

$$X = (x_1, \ldots, x_i, \ldots), \qquad Y = (x_1, \ldots, y_i, \ldots)$$

となる．このとき熱浴法の条件付き確率は

$$P(x_i' = y_i) = \frac{W(Y)}{\sum_{x_i} W(x_1, \ldots, x_i, \ldots)} \equiv \frac{W(Y)}{Z_i},$$

ここで Z_i は i 番目のみを動かしたときの重みの総和で，X と Y に対して共通である．

よって遷移確率は

$$P(X \to Y) = p_i\frac{W(Y)}{Z_i}, \qquad P(Y \to X) = p_i\frac{W(X)}{Z_i}.$$

したがって

$$P(X \to Y)W(X) = p_i \frac{W(Y)}{Z_i} W(X),$$

$$P(Y \to X)W(Y) = p_i \frac{W(X)}{Z_i} W(Y),$$

より両辺は一致する.

X と Y が 2 箇所以上異なる場合には熱浴法では 1 ステップで遷移できないため

$$P(X \to Y) = P(Y \to X) = 0$$

となり，この場合も自明に詳細つり合いが成り立つ.

以上より熱浴法の遷移確率は詳細つり合い条件

$$P(X \to Y)W(X) = P(Y \to X)W(Y)$$

を満たす.

3.3 3. 詳細つり合い条件 (5.19) からつり合い条件 (5.17) が従うことの証明

詳細つり合い条件

$$P(X \to Y)W(X) = P(Y \to X)W(Y) \tag{5.19}$$

がすべての X, Y について成り立つと仮定する.

このとき任意の X について

$$\sum_Y P(Y \to X)W(Y)$$

を計算する：

$$\sum_Y P(Y \to X)W(Y) = \sum_Y P(X \to Y)W(X),$$

ここで右辺の等号は詳細つり合い (5.19) より従う.

また遷移確率の和は $\sum_Y P(X \to Y) = 1$ であるから,

$$\sum_Y P(Y \to X)W(Y) = W(X) \sum_Y P(X \to Y) = W(X).$$

よってつり合い条件

$$W(X) = \sum_Y P(Y \to X)W(Y) \tag{5.17}$$

が示された．詳細つり合いが成立すれば，W はマルコフ過程の定常分布となる.

4 課題 4：熱よく法によるイジングモデルのシミュレーション

4.1 原理と方法

本節では，二次元トーラス上の強磁性イジング模型

$$H(s) = -J \sum_{\langle i,j \rangle} s_i s_j, \qquad s_i \in \{+1, -1\},\ J = 1$$

に対し，マルコフ連鎖モンテカルロ法（MCMC）を用いて平衡状態を生成する手法について述べる [1]．格子は $L \times L$ の正方格子とし，周期境界条件（トーラス）を課す．本問では特に，熱浴法（Heat Bath 法）とメトロポリス法（Metropolis 法）を比較し，初期状態依存性が消失する様子を磁化の時間発展から観察する。

4.1.1 イジングモデルの熱浴法（小問 1）

熱浴法は，選んだ格子点 i のスピン s_i を，その条件付き分布 $P(s_i \mid s_{\setminus i})$ から**直接サンプリング**する更新法である [1]。スピン i の近傍スピン（4 近傍）を s_j とし，その和を

$$h_i = \sum_{j \in \mathrm{nn}(i)} s_j$$

と書くと，ハミルトニアンからエネルギーは

$$E(s_i) = -J s_i h_i$$

であるから，ボルツマン分布より

$$P(s_i = +1) \propto e^{\beta h_i}, \qquad P(s_i = -1) \propto e^{-\beta h_i}.$$

よって正規化すれば

$$P(s_i = +1 \mid s_{\setminus i}) = \frac{1}{1 + e^{-2\beta h_i}}, \qquad P(s_i = -1) = 1 - P(s_i = +1). \tag{3}$$

以上より，熱浴法の 1 ステップは以下のように与えられる。

1. 格子点 i をランダムに 1 つ選ぶ。
2. 近傍和 $h_i = \sum_j s_j$ を計算する。
3. $P(s_i = +1) = 1/(1 + e^{-2\beta h_i})$ を求める。
4. 一様乱数 $r \in [0, 1]$ を生成し，

$$r < P(s_i = +1) \;\Rightarrow\; s_i \leftarrow +1, \quad r \geq P(s_i = +1) \;\Rightarrow\; s_i \leftarrow -1.$$

条件付き分布からの厳密サンプリングに基づくため，自動的に平衡分布を保つマルコフ連鎖が得られる。

4.1.2 メトロポリス法

メトロポリス法では，スピン反転 $s_i \to -s_i$ を更新案とし，対応するエネルギー変化

$$\Delta E = 2 s_i h_i$$

を用いて採択判定を行う [1]。採択確率は

$$P_{\mathrm{acc}} = \min\{1, e^{-\beta \Delta E}\}$$

であり，$\Delta E \leq 0$（エネルギーが下がる場合）は必ず採択され，$\Delta E > 0$ の場合には確率的に採択される。

4.2 実装上の工夫

周期境界条件の最近接スピンは NumPy の `np.roll` を用いて

上，下，左，右

の方向に周期的に配列をシフトすることで効率的に求めることができる。また，格子をチェッカーボード状に黒白に分割し，黒マス→白マスの順に更新することで，隣接点を同時更新することによる干渉を防ぎつつ，ベクトル化された高速な一括更新が可能となる。

4.2.1 Python による実装例

以下では NumPy による簡潔な実装例を示す。

■熱浴法：

```
def sweep_heatbath(spins, beta, rng, itrations=1500):
    ms = []
    for _ in range(itrations):
        for mask in (mask_black, mask_white):

            n  = np.roll(spins,  1, axis=0)
            n += np.roll(spins, -1, axis=0)
            n += np.roll(spins,  1, axis=1)
            n += np.roll(spins, -1, axis=1)

            p = 1.0 / (1.0 + np.exp(-2 * beta * n))
            r = rng.random((L, L))

            spins[mask] = np.where(r[mask] < p[mask], 1, -1)

        ms.append(spins.mean())
    return np.array(ms)
```

■メトロポリス法：

```
def sweep_metropolis(spins, beta, rng, itrations=1500):
    ms = []
    for _ in range(itrations):
        for mask in (mask_black, mask_white):

            n  = np.roll(spins,  1, axis=0)
            n += np.roll(spins, -1, axis=0)
```

```
            n += np.roll(spins,  1, axis=1)
            n += np.roll(spins, -1, axis=1)

            s_sub = spins[mask]
            n_sub = n[mask]
            deltaE = 2.0 * s_sub * n_sub

            r = rng.random(s_sub.shape)
            accept = (deltaE <= 0) | (r < np.exp(-beta * deltaE))

            s_sub[accept] *= -1
            spins[mask] = s_sub

        ms.append(spins.mean())
    return np.array(ms)
```

4.3 メトロポリス法、熱浴法によるイジングモデルのシミュレーション（小問 2）

4.3.1 シミュレーション条件

本問では，問題文の指定に従い以下の条件でシミュレーションを行う。

- 系サイズ：$L = 64$。
- 境界条件：トーラス境界。
- 初期状態：全スピン $s_i(0) = +1$。
- 観測量：逐次ステップにおける磁化

$$m(t) = \frac{1}{L^2} \sum_i s_i(t).$$

- 温度：$z = \exp(2/T) - 1 = \sqrt{2}$ を満たす温度 T。

4.3.2 磁化の時間発展（小問 2）

初期状態 $s_i(0) = +1$ からスタートし，熱浴法およびメトロポリス法で 10000 ステップの更新を行った。結果として得られた磁化 $m(t)$ の時間発展を比較したところ、以下の図のようになった。

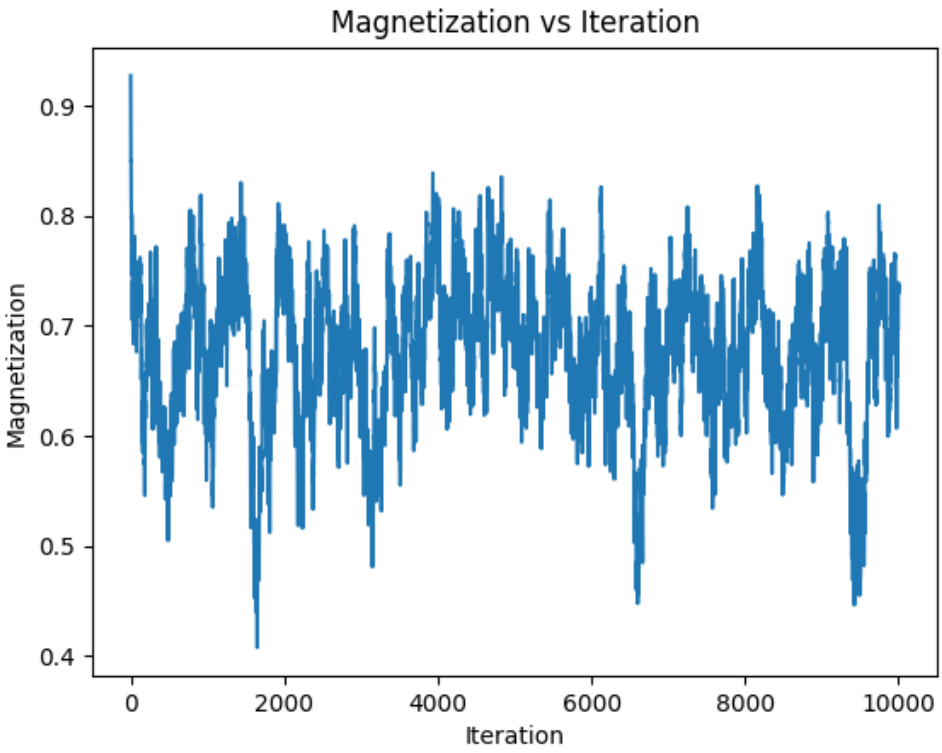

図 4: 熱よく法による磁化の時間発展

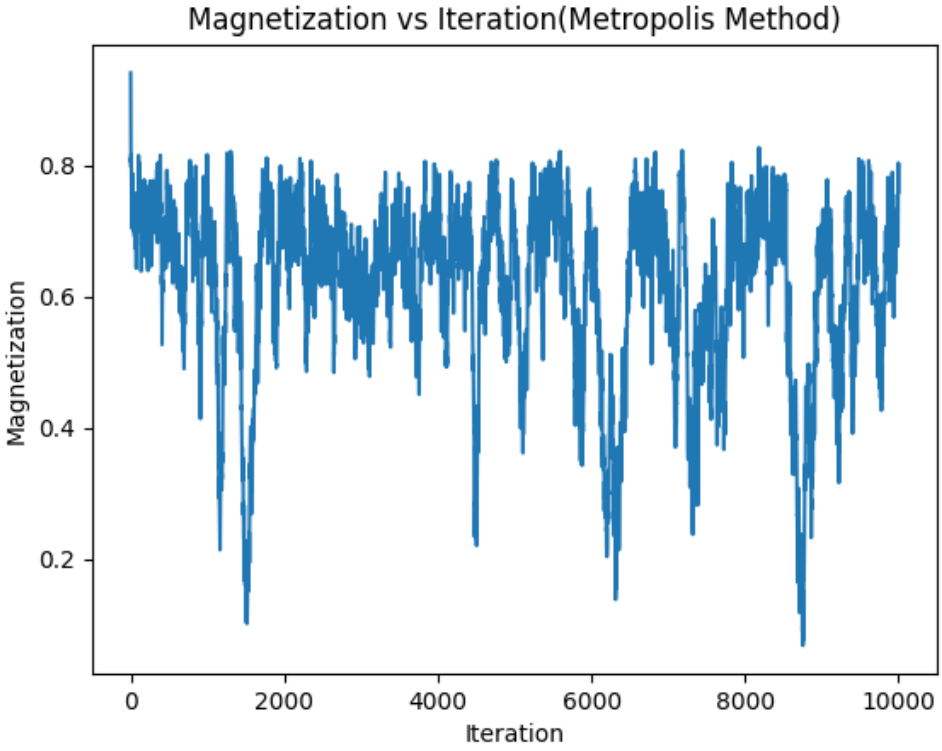

図 5: メトロポリス法による磁化の時間発展

4.3.3 考察

図 4, 図 5 から、いずれの方法でも初期の $+1$ 状態から出発して磁化の逆転を繰り返しながら平衡状態を中心に振動する様子が観察された。熱浴法とメトロポリス法の比較では、メトロポリス方法の方が磁化の振動が大きく、不安定であることがわかる。いずれの方法でも、500 ステップたてば十分に初期の $+1$ 状態からの影響が消失しており、小問 3 で用いる burn-in 期間として適切であると考えられる。

4.4 磁化の期待値と系サイズ・温度依存性（小問 3）

本問では，磁化の期待値

$$\langle m \rangle(T, L) = \left\langle \frac{1}{L^2} \sum_i s_i \right\rangle$$

の温度 T および系サイズ L への依存性を，モンテカルロ法により求める．ここで $\langle \cdot \rangle$ は平衡分布に関する期待値を表す。問題文の条件に従い，温度は

$$z = \exp(2/T) - 1$$

が区間 $z \in (1.2, 1.5]$ を等間隔に分割する 12 点以上を用いた。この区間には二次元イジング模型の臨界温度

$$T_c = \frac{2}{\ln\bigl(1+\sqrt{2}\bigr)} \quad \Longleftrightarrow \quad z_c = \sqrt{2}$$

が含まれており，秩序–無秩序転移の挙動を観察できる。

系サイズは

$$L = 12,\ 16,\ 24,\ 32,\ 48,\ 64$$

の 6 通りについて計算した。各 (L, z) ごとに，メトロポリス法と熱浴法のいずれかでマルコフ連鎖を構成し，十分に長いモンテカルロステップ (MCS) を回した。

4.4.1 サンプリング方法

問題文の指示に従い，各条件 (L, z) について総サンプリング数が 10^6 MCS 以上となるように設定した。具体的には，以下のような方法で実装した。

- それぞれの (L, z) について独立なイジング系を $N = 1000$ 個用意する。
- 各系を同じ温度 T のもとで 1500 MCS だけ時間発展させる。
- 初期値依存性を除くため，問題 (2) の結果を参考に，最初の 500 ステップを burn-in として捨て，残り 1000 ステップのみを平均に用いる。

このとき，有効サンプル数は

$$N_{\text{sample}} = N \times (1500 - 500) = 1000 \times 1000 = 10^6$$

となり，問題文の条件を満たす。

各ステップ t における k 番目の系の磁化を $m_k(t)$ と書くと，磁化の期待値の推定量は

$$\langle m \rangle(T, L) \simeq \frac{1}{N_{\text{sample}}} \sum_{k=1}^{N} \sum_{t=t_{\text{burn}}+1}^{t_{\max}} m_k(t), \qquad t_{\text{burn}} = 500, \quad t_{\max} = 1500.$$

実装上は，まず各ステップごとに格子平均 $m_k(t) = L^{-2}\sum_i s_i^{(k)}(t)$ を計算し，その後，k と t の二重和を取ることで $\langle m \rangle$ を求めた。

4.4.2 実装上の工夫

本問のシミュレーションでは，$L = 64$ に対して 10^6 サンプル以上を要求されるため，単純な Python ループに基づく逐次更新では計算量が膨大となり，実行時間が現実的でなくなる。そこで本研究では次の 3 点の最適化を施し，計算速度を大幅に改善した。

■(1) 系の並列化：$N = 1000$ 系を同時に更新　問題文にある「10^6 MCS 以上」という条件を効率的に満たすために，1 つの温度に対して $N = 1000$ 個の独立なイジング系を同時に生成し，これらを同一の温度で並列に時間発展させた。これにより，1 系あたり 1000 ステップ進めるだけで

$$N \times (1500 - 500) = 1000 \times 1000 = 10^6$$

という必要サンプル数を一度に確保でき，ループ回数を従来の 1000 分の 1 以下に削減できた。

■(2) PyTorch による GPU 並列計算　NumPy による CPU ベースの更新では，$N = 1000$ 系を同時に扱うと計算量が急激に増大する。そこで本研究では PyTorch を用いて，スピン配列 (K, N, L, L) を GPU メモリ上に展開し，以下のすべての処理を GPU 上で実行した。

- 近傍和の計算（`torch.roll` によるベクトル化）
- 熱浴法の確率計算（`torch.sigmoid` による高速化）
- メトロポリス法の採択判定
- 磁化の計算（空間平均の並列化）

これにより，CPU 実装では数十分を要する計算が，GPU ではわずか数十秒程度で完了するようになった。

■(3) 温度点 z の 12 個をひとまとめにして並列更新　問題文にある “$z \in (1.2, 1.5]$ の範囲で 12 点以上” という条件に対し，通常であれば温度を 1 つずつ変えて 12 回のシミュレーションを行う必要がある。しかし本研究では温度の逆温度 β を 12 個まとめて

$$\beta_k \quad (k = 1, \ldots, 12)$$

としてベクトル化し，スピン配列を

$$\text{spins} \in \mathbb{R}^{K \times N \times L \times L}$$

として $K = 12$ 個の温度点を同時並列に扱った。これにより，12 回の独立な実行を 1 回にまとめることができ，計算時間をさらに 1/12 に短縮できた。

■(4) 実行速度の改善と $L = 128$ への拡張　以上の最適化により，$L = 64$ および 12 個の温度点に対する 10^6 サンプルの磁化期待値計算が，

$$\text{総実行時間} \approx 2 \text{ 分}$$

という高速な実行を達成した。

計算速度に十分余裕が生まれたため，本問の指定にはないものの，

$$L = 128$$

についても同一コードで追加計算を行ったところ，GPU メモリの制約にも抵触せず，問題なく同様のシミュレーションを実行できた。

4.4.3 結果

前節で述べた方法により，$z = \exp(2/T) - 1$ を $z \in (1.2, 1.5]$ の範囲で 12 点とり，それぞれに対応する温度 T において磁化の期待値 $\langle m \rangle(T, L)$ を計算した。図 6 に，横軸を温度 T，縦軸を磁化の期待値としたプロットを示す。各曲線は $L = 12, 16, 24, 32, 48, 64, 128$ に対応しており，系サイズが大きくなるにつれて曲線がより急峻になる様子が見てとれる。

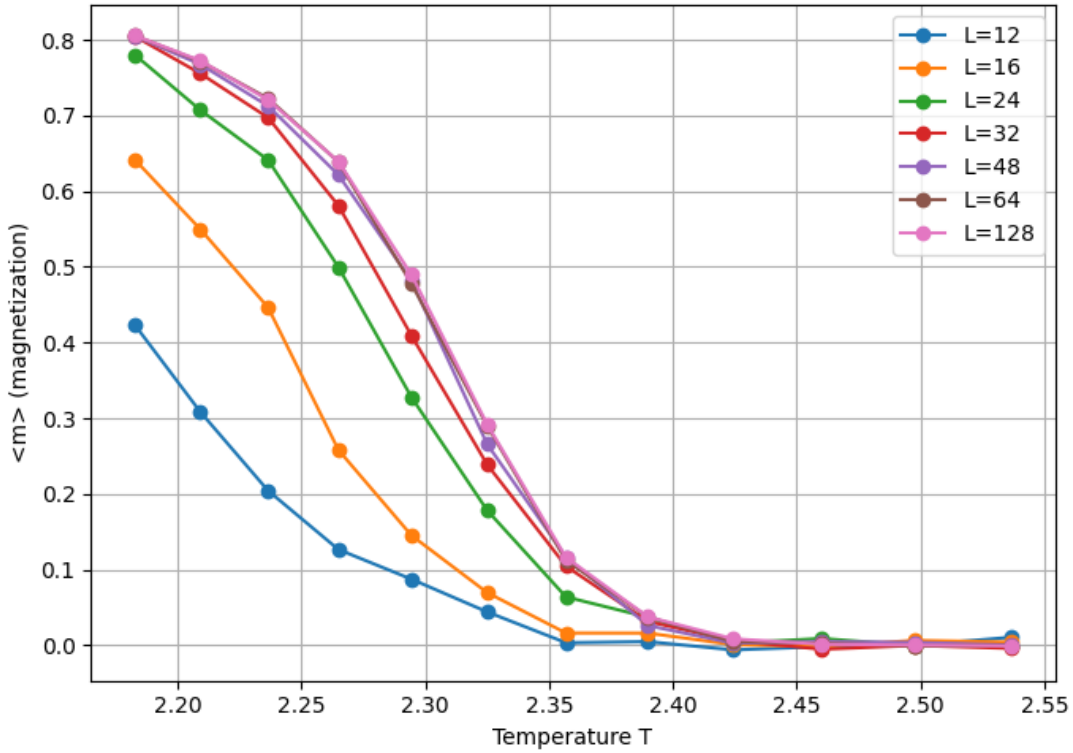

図 6: 各系サイズ L における磁化の期待値 $\langle m \rangle$ の温度 T 依存性.

4.4.4 考察

低温側（$T \lesssim 2.2$）では，大きな系サイズ $L \geq 24$ について $\langle m \rangle \approx 0.75$–$0.8$ の飽和した値をとり，強磁性秩序状態にあることが確認できる。温度を上げていくと，$T \approx 2.25$ 付近から $\langle m \rangle$ が急激に減少する領域が現れ，$T \approx 2.35$ 付近ではほとんどの系で $\langle m \rangle \approx 0$ となる。高温側（$T \gtrsim 2.4$）では，いずれの系サイズでも $\langle m \rangle$ は 0 近傍に張り付いており，常磁性状態にあることが分かる。

また，系サイズ L を変化させたときの詳細を見ると，小さい系サイズ $(L = 12, 16)$ では $\langle m \rangle$ の減少がなだらかであり、相転移が明確でない一方で、$L = 48, 64, 128$ では，$T \approx 2.25$〜2.3 の狭い温度範囲で急峻に落ち込んでいる。見かけの転移点の理論値がおよそ 2.269 であり、有限サイズ系におい

て丸められた相転移が，系サイズが大きくなるにつれて系サイズ無限のシャープな転移に近づく様子が観察できる。

図 6 から，二次元イジング模型の強磁性–常磁性転移の特徴的な振る舞いが数値的に再現されていることが分かる。低温側では系サイズに依らず大きな自発磁化が観測され，高温側では $\langle m \rangle \simeq 0$ に収束している。臨界温度近傍では，系サイズ L が大きくなるほど $\langle m \rangle(T, L)$ の曲線がより急峻となり，転移がシャープになっていく。

また，L を大きくすると低温側および高温側の曲線がほぼ重なってくる一方で，臨界付近のみ系サイズ依存性が強くなることも確認できる。$L < 64$ においては L によってグラフが大きく変化していたのに対して $L = 64, 128$ においてはグラフがほぼ重なっていることから、$L = 64$ あたりで十分に大きな系サイズに達しており、臨界挙動の解析に適していると考えられる。

結論

本実験では、モンテカルロ法を用いた数値計算手法について学んだ。疫病の確率的な伝染モデルをシミュレーションし、感染拡大の様子を観察した。また、円周率のモンテカルロ法による推定を行い、サンプル数の増加に伴う推定精度の向上を確認した。さらに、イジングモデルに対して熱浴法とメトロポリス法を実装し、磁化の期待値と系サイズ・温度依存性を比較した。実装においては、Python のベクトル化や GPU 並列計算を活用し、計算効率を大幅に改善した。これにより、大規模なシミュレーションが現実的な時間内で実行可能となった。今後は、さらに高度なモンテカルロ手法や他の物理モデルへの応用を検討したい。

付録 A　ソースコード

コード作成、レポート作成の一部に GitHub Copilot を使用した。

A.1　課題 1 のコード

A.1.1　小問 1

Listing 1: 課題 1-1 のコード

```
import numpy as np

N = 64
M = 10
T = 256
# 全試行で同じ初期条件：32番目だけ 1
s = np.zeros((M, N), dtype=int)
plot_MT = np.zeros((M, T), dtype=int)
s[:, 31] = 1
p = 0.7

def prob_from_s_n_batch(s, n, p):
        # 6 通りすべての α を p から計算しておく
    # 順番は (s,n) = (0,0),(0,1),(0,2),(1,0),(1,1),(1,2)
    vals = np.array([
        0.0,
```

```
        p**2,
        p**2*(2-p**2),
        p**2*(2-p**2),
        p**2*(p**3 - 2*p**2 - p + 3),
        p**2*(2-p)*(p**3 - 2*p**2 + 2)
    ])

    # s, n は (M, N) 配列
    index = s*3 + n          # (0,0,0,1,1,1)×3 + (0,1,2,0,1,2) → 0〜5 にマップ
    alpha = vals[index]      # 完全にブランチレスなベクトル演算
    return alpha

for t in range(T):
    right = np.roll(s, -1, axis=1)  # 右隣の状態
    left = np.roll(s, 1, axis=1)    # 左隣の状態
    n = right + left                # 隣接する 1 の数
    alpha = prob_from_s_n_batch(s, n, p)  # 各セルの α を計算
    rand = np.random.rand(M, N)    # 一様乱数を生成
    s = (rand < alpha).astype(int)  # α と比較して次の状態を決定
    num_ones = np.sum(s, axis=1)  # 各試行の 1 の数をカウント
    plot_MT[:, t] = num_ones

# プロット
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
for m in range(M):
    plt.plot(range(T), plot_MT[m, :], alpha=0.7, linewidth=0.8)

plt.xlabel('Time step (t)', fontsize=14)
plt.ylabel('Cells in state 1', fontsize=14)
plt.title(f'Evolution of Number of State-1 Cells (M={M} trials, N={N} cells, p={p})',
    fontsize=14)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('task1_1.png', dpi=300, bbox_inches='tight')
plt.show()
```

A.1.2 小問 2

Listing 2: 課題 1-2 のコード

```
import numpy as np
import matplotlib.pyplot as plt

N = 64
M = 10000
T = 1024
p_values = [0.62, 0.625, 0.63, 0.635, 0.64, 0.645, 0.65, 0.66, 0.68, 0.7]

def prob_from_s_n_batch(s, n, p):
    # 6 通りすべての α を p から計算しておく
    # 順番は (s,n) = (0,0),(0,1),(0,2),(1,0),(1,1),(1,2)
    vals = np.array([
        0.0,
        p**2,
        p**2*(2-p**2),
```

```
        p**2*(2-p**2),
        p**2*(p**3 - 2*p**2 - p + 3),
        p**2*(2-p)*(p**3 - 2*p**2 + 2)
    ])

    # s, n は (M, N) 配列
    index = s*3 + n          # (0,0,0,1,1,1)×3 + (0,1,2,0,1,2) → 0〜5 にマップ
    alpha = vals[index]      # 完全にブランチレスなベクトル演算
    return alpha

# 各 p の値に対してシミュレーションを実行
plt.figure(figsize=(12, 6))

for p in p_values:
    print(f'Computing for p = {p}...')

    # 初期化：全試行で同じ初期条件：32番目だけ 1
    s = np.zeros((M, N), dtype=int)
    s[:, 31] = 1

    # 各時刻の m(t) のサンプル平均を保存
    mean_m_t = np.zeros(T)

    for t in range(T):
        right = np.roll(s, -1, axis=1)  # 右隣の状態
        left = np.roll(s, 1, axis=1)    # 左隣の状態
        n = right + left                # 隣接する 1 の数
        alpha = prob_from_s_n_batch(s, n, p)  # 各セルの α を計算
        rand = np.random.rand(M, N)    # 一様乱数を生成
        s = (rand < alpha).astype(int)  # α と比較して次の状態を決定

        # 各試行の 1 の数をカウントし、サンプル平均を計算
        num_ones = np.sum(s, axis=1)  # (M,) の配列
        mean_m_t[t] = np.mean(num_ones)

    # プロット
    plt.plot(range(T), mean_m_t, label=f'p = {p}', linewidth=2)
    print(f'  Final mean m(T-1) = {mean_m_t[-1]:.2f}')

plt.xlabel('Time step (t)', fontsize=14)
plt.ylabel('Mean of m(t)', fontsize=14)
plt.title(f'Sample Mean of m(t) for Different p Values (M={M} trials, N={N} cells)',
    fontsize=14)
plt.legend(fontsize=11)
plt.grid(True, alpha=0.3)
plt.tight_layout()

# Save linear scale plot
plt.savefig('task1_2_sample_mean_linear.png', dpi=300, bbox_inches='tight')
print('\nSaved plot to task1_2_sample_mean_linear.png')

# Switch to log scale and save
plt.xscale('log')
plt.yscale('log')
plt.savefig('task1_2_sample_mean_log.png', dpi=300, bbox_inches='tight')
print('Saved plot to task1_2_sample_mean_log.png')

plt.show()
```

A.2 課題 2 のコード

Listing 3: 課題 2 のコード

```python
import numpy as np

def pi_monte_carlo_integration(M):
    s = np.random.rand(10, M)      # (10, M)
    s *= s                         # s = s**2
    s += 1.0                       # 1 + s**2
    np.reciprocal(s, out=s)        # 1 / (1 + s**2)
    s *= 4.0                       # 4 / (1 + s**2)
    mean_fs = s.mean(axis=1)       # 各行の平均
    # S の不偏推定分散を計算
    s -= mean_fs[:, np.newaxis]  # 各行から平均を引く
    s *= s                         # 平方
    variances = s.sum(axis=1) / (M - 1)  # 不偏分散
    accuracy = np.sqrt(variances / M)  # 標準誤差
    return mean_fs, accuracy

# M = 100_000_000
Ms = [10,1000,100000, 10000000]
for M in Ms:
    mean_fs, accuracy = pi_monte_carlo_integration(M)

    print(f"M = {M}")
    for f in mean_fs:
        print("pi")
        print(f"{f:.12f}")
        print("accuracy")
        print(f"{accuracy[0]:.12f}")
    print(f"Mean: {mean_fs.mean():.12f}")
```

A.3 課題 4 のコード

A.3.1 熱浴法

Listing 4: 課題 4-2 熱浴法のコード

```python
import numpy as np
import matplotlib.pyplot as plt
L = 64
# 温度T はz =exp(2/T) - 1 = √2の 解

# beta = 1.0 / T
beta = 0.5 * np.log(1.0 + np.sqrt(2.0))
rng = np.random.default_rng()

# 状態配列（唯一の状態配列）
spins = np.ones((L, L), dtype=np.int8)

# チェッカーボードマスク（初回だけ作る）
x = np.arange(L)[:, None]
```

```
y = np.arange(L)[None, :]
mask_black = ((x + y) % 2 == 0)
mask_white = ~mask_black


def sweep(spins, beta, rng,itrations=100):
    ms = []
    for _ in range(itrations):
        for mask in (mask_black, mask_white):

            # ----- 近傍和（最速） -----
            n  = np.roll(spins,  1, axis=0)
            n += np.roll(spins, -1, axis=0)
            n += np.roll(spins,  1, axis=1)
            n += np.roll(spins, -1, axis=1)

            # ----- 熱浴確率 -----
            p = 1.0 / (1.0 + np.exp(-2 * beta * n))

            # ----- ランダムに更新 -----
            r = rng.random((L, L))
            sub = mask
            spins[sub] = np.where(r[sub] < p[sub], 1, -1)
        ms.append(spins.mean())
    return np.array(ms)

ms = sweep(spins, beta, rng, itrations=10000)

plt.plot(ms)
plt.title('Magnetization vs Iteration(Heat Bath Method)')
plt.xlabel('Iteration')
plt.ylabel('Magnetization')
plt.title('Magnetization vs Iteration')
plt.savefig('task4_2_heat.png')
plt.show()
```

A.3.2 メトロポリス法

Listing 5: 課題 4-2 メトロポリス法のコード

```
import numpy as np
import matplotlib.pyplot as plt
L = 64
# 温度T はz =exp(2/T)−1 = √2の 解

# beta = 1.0 / T
beta = 0.5 * np.log(1.0 + np.sqrt(2.0))
rng = np.random.default_rng()

# 状態配列（唯一の状態配列）
spins = np.ones((L, L), dtype=np.int8)

# チェッカーボードマスク（初回だけ作る）
x = np.arange(L)[:, None]
y = np.arange(L)[None, :]
mask_black = ((x + y) % 2 == 0)
mask_white = ~mask_black
```

```

def sweep(spins, beta, rng, itrations=100):
    """
    J=1, H=0 のイジング模型をメトロポリス法 + チェッカーボード更新で itrations ステップ回す。
    spins : (L, L) の int 型配列（要素は ±1）
    beta  : 1 / T
    rng   : np.random.Generator
    """
    ms = []
    for _ in range(itrations):
        for mask in (mask_black, mask_white):

            # ----- 近傍和 n(i,j) = 上下左右の和 -----
            n  = np.roll(spins,  1, axis=0)
            n += np.roll(spins, -1, axis=0)
            n += np.roll(spins,  1, axis=1)
            n += np.roll(spins, -1, axis=1)

            # ----- ΔE = 2 s_i Σ_j s_j (J=1, H=0) -----
            # mask で対象サイトだけ取り出す
            s_sub = spins[mask]      # 現在の s_i
            n_sub = n[mask]          # 近傍和 Σ_j s_j

            deltaE = 2.0 * s_sub * n_sub  # ここは配列（float でも int でも可）

            # ----- メトロポリスの採択確率 -----
            # ΔE <= 0 なら必ず採択、ΔE > 0 なら exp(-beta * ΔE) で採択
            r = rng.random(s_sub.shape)   # [0,1) 一様乱数

            accept = (deltaE <= 0.0) | (r < np.exp(-beta * deltaE))

            # ----- 採択されたサイトだけスピンを反転 -----
            s_sub[accept] *= -1
            spins[mask] = s_sub

        ms.append(spins.mean())

    return np.array(ms)

ms = sweep(spins, beta, rng, itrations=10000)

plt.plot(ms)
plt.title('Magnetization vs Iteration(Metropolis Method)')
plt.xlabel('Iteration')
plt.ylabel('Magnetization')
plt.savefig('task4_2_metro.png')
plt.show()
```

A.3.3 複数温度の並列シミュレーション

Listing 6: 課題 4-3 のコード

```
import torch
import math
import numpy as np
```

```
import matplotlib.pyplot as plt

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("device =", device)

# ------------------------------------------------------------
#  K × N × L × L を同時に回す heat-bath + checkerboard
#  spins: float32, ±1.0
# ------------------------------------------------------------
def sweep_multi_beta(spins, two_betas, iterations, mask_black, mask_white):
    """
    spins      : (K, N, L, L) float32, ±1
    two_betas  : (K, 1, 1, 1) float32, 2*beta
    mask_black : (1, 1, L, L) bool
    mask_white : (1, 1, L, L) bool
    戻り値     : spins, ms (iterations, K, N)
    """
    K, N, L, _ = spins.shape
    ms = torch.empty((iterations, K, N), dtype=torch.float32, device=device)

    for t in range(iterations):
        for mask in (mask_black, mask_white):
            # 近傍和 n = 上下左右
            n  = torch.roll(spins,  1, dims=2)
            n += torch.roll(spins, -1, dims=2)
            n += torch.roll(spins,  1, dims=3)
            n += torch.roll(spins, -1, dims=3)

            # p = sigmoid(2 β n)
            p = torch.sigmoid(two_betas * n)

            # 一様乱数
            r = torch.rand_like(p)

            # 候補値（全格子）
            new_spins = torch.where(r < p, 1.0, -1.0)

            # マスクされた格子だけ更新
            spins = torch.where(mask, new_spins, spins)

        # 各ステップの各系の磁化（空間平均）
        ms[t] = spins.mean(dim=(3, 2))  # (K, N)

    return spins, ms


# ------------------------------------------------------------
#  1つの L について z_list を同時並列で回す
# ------------------------------------------------------------
def run_for_L(L, z_list, N=1000, iterations=1500, burn_in=500):
    """
    L: system size
    z_list: 温度パラメータ列 (len = K)
    N: 系の数
    iterations: 総 MCS
    burn_in: 捨てるステップ数
    戻り値: (z_array[K], m_expect[K])
    """
```

```
    K = len(z_list)

    # z = exp(2/T) - 1 から beta をすべて計算
    betas = []
    for z in z_list:
        T = 2.0 / math.log(z + 1.0)
        betas.append(1.0 / T)
    betas = np.array(betas, dtype=np.float32)

    # two_betas = 2 * beta を tensor にして shape (K,1,1,1)
    two_betas = torch.tensor(2.0 * betas, dtype=torch.float32,
                             device=device).view(K, 1, 1, 1)

    # スピン初期状態: 全て +1.0 (float32)
    spins = torch.ones((K, N, L, L), dtype=torch.float32, device=device)

    # チェッカーボードマスク (1,1,L,L)
    x = torch.arange(L, device=device).view(L, 1)
    y = torch.arange(L, device=device).view(1, L)
    mask2d_black = ((x + y) % 2 == 0)
    mask_black = mask2d_black.view(1, 1, L, L)
    mask_white = (~mask2d_black).view(1, 1, L, L)

    # シミュレーション
    spins, ms = sweep_multi_beta(spins, two_betas, iterations,
                                 mask_black, mask_white)

    # burn-in を捨てて残りで平均 (サンプル数 = (iterations-burn_in) * N)
    ms_eff = ms[burn_in:]                          # (iterations-burn_in, K, N)
    # 時間方向(dim=0) と系方向(dim=2)で平均 → shape = (K,)
    m_expect = ms_eff.mean(dim=0).mean(dim=1).cpu().numpy()

    return np.array(z_list), m_expect


# ------------------------------------------------------------
#  main: L と z を回して <m> を求める & プロット & 数値表示
# ------------------------------------------------------------
def main():
    # システムサイズ
    L_list = [12, 16, 24, 32, 48, 64, 128]

    # z ∈ [1.2, 1.5] を 12 点
    z_list = np.linspace(1.2, 1.5, 12)

    all_results = {}

    for L in L_list:
        print(f"\n=== L = {L} ===")
        z_vals, m_vals = run_for_L(L, z_list,
                                   N=1000, iterations=1500, burn_in=500)
        all_results[L] = (z_vals, m_vals)

        # ここで具体的な値を表示
        for z, m in zip(z_vals, m_vals):
            print(f"L={L:2d}, z={z:.3f}, <m>={m:.6f}")

    # プロット
```

```
    plt.figure(figsize=(7, 5))
    for L in L_list:
        z_vals, m_vals = all_results[L]
        # z = exp(2/T) - 1  =>  T = 2 / log(z+1)
        T_vals = 2.0 / np.log(z_vals + 1.0)
        plt.plot(T_vals, m_vals, "-o", label=f"L={L}")
    plt.xlabel("Temperature T")
    plt.ylabel("<m> (magnetization)")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

    return all_results


# 実行
all_results = main()
```

参考文献

参考文献

[1] 数理工学実験（2025 年度配布資料）.