数理工学実験レポート

第 8 章（偏微分方程式）

学籍番号 1029366161　中塚一瑳

2026 年 1 月 19 日

目次

はじめに

今回は、偏微分方程式の数値解法について学習した。課題 1 では拡散方程式に対するオイラー陽解法とクランク-ニコルソン法を実装し、Dirichlet 境界条件および Neumann 境界条件のもとで数値解を求めた。課題 2 では Fisher 方程式（反応拡散方程式）の数値解を求め、進行波解の振る舞いを観察した。課題 3 ではシュレディンガー方程式の数値解法を実装し、調和振動子ポテンシャル中の波束の時間発展を追跡した。

1　課題 1：拡散方程式の数値解法

1.1　問題設定

1 次元拡散方程式

$$\frac{\partial u}{\partial t}(x,t) = \frac{\partial^2 u}{\partial x^2}(x,t) \tag{1}$$

を、区間 $[0, L]$（$L = 10$）上で数値的に解く。ここで $u(x,t)$ は位置 x・時刻 t における濃度を表す。初期条件は

$$u_0(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(x-5)^2\right) \tag{2}$$

とし、Dirichlet 境界条件および Neumann 境界条件の両方について計算を行った。

1.2　原理と方法

1.2.1　離散化と無次元パラメータ c の導出

空間区間 $[0, L]$ を幅 Δx で分割し、$L = N\Delta x$ とする。時間も刻み幅 Δt で離散化する。区間 $[(j-1)\Delta x, j\Delta x]$ における平均値として

$$u_j^n = \frac{1}{\Delta x} \int_{(j-1)\Delta x}^{j\Delta x} u(y, n\Delta t)\, dy \tag{3}$$

を定義する。[1]

拡散方程式を時間区間 $[n\Delta t, (n+1)\Delta t]$ で積分すると、

$$u(x,(n+1)\Delta t) - u(x, n\Delta t) = \int_{n\Delta t}^{(n+1)\Delta t} \frac{\partial^2 u}{\partial x^2}(x,s)\, ds \tag{4}$$

を得る。右辺を時刻 $n\Delta t$ の値で近似すると、

$$u(x,(n+1)\Delta t) - u(x, n\Delta t) \simeq \Delta t \frac{\partial^2 u}{\partial x^2}(x, n\Delta t) \tag{5}$$

となる。さらに空間微分を中心差分で近似すると、

$$\frac{\partial^2 u}{\partial x^2}(x, n\Delta t) \simeq \frac{u_{j-1}^n - 2u_j^n + u_{j+1}^n}{\Delta x^2} \tag{6}$$

である。これらをまとめると、

$$u_j^{n+1} - u_j^n = \frac{\Delta t}{\Delta x^2}\left(u_{j-1}^n - 2u_j^n + u_{j+1}^n\right) \tag{7}$$

を得る。ここで

$$\boxed{c \equiv \frac{\Delta t}{\Delta x^2}} \tag{8}$$

と定義する。この c は時間刻みと空間刻みの比を表し、数値安定性を支配する重要な無次元量である。

1.2.2 オイラー陽解法

差分方程式をそのまま用いると、

$$u_j^{n+1} = u_j^n + c\left(u_{j-1}^n - 2u_j^n + u_{j+1}^n\right) \tag{9}$$

となる。これは既知の時刻 n の値から直接 $n+1$ を計算できるため、オイラー陽解法と呼ばれる。[1]

■特徴

- 実装が非常に簡単
- ただし安定性条件 $c \leq 1/2$ を満たさないと数値解が発散する
- Δx を小さくすると Δt も強く制限される

1.2.3 クランク–ニコルソン法

時間積分を台形公式で近似すると、

$$u_j^{n+1} - u_j^n = \frac{c}{2}\left[(u_{j-1}^{n+1} - 2u_j^{n+1} + u_{j+1}^{n+1}) + (u_{j-1}^n - 2u_j^n + u_{j+1}^n)\right] \tag{10}$$

を得る。整理すると、

$$-\frac{c}{2}u_{j-1}^{n+1} + (1+c)u_j^{n+1} - \frac{c}{2}u_{j+1}^{n+1} = (1-c)u_j^n + \frac{c}{2}(u_{j-1}^n + u_{j+1}^n) \tag{11}$$

となる。未知数 u_j^{n+1} が連立方程式として現れるため、陰解法に分類される。[1]

■行列表示と三重対角構造　内部格子点 $j = 1, \ldots, N$ に対して、未知ベクトル $\mathbf{x} = (u_1^{n+1}, u_2^{n+1}, \ldots, u_N^{n+1})^T$ を定義すると、式 (11) は

$$A\mathbf{x} = \mathbf{z} \tag{12}$$

と書ける。係数行列 A は三重対角行列となる。

■LU 分解による解法　三重対角行列 A を

$$A = LU \tag{13}$$

と下三角行列 L と上三角行列 U の積に分解する。L と U の係数は漸化式により前処理として一度だけ計算できる。

連立方程式 $A\mathbf{x} = \mathbf{z}$ は、$LU\mathbf{x} = \mathbf{z}$ とおき、

1. 前進代入：$L\mathbf{y} = \mathbf{z}$ を解く
2. 後退代入：$U\mathbf{x} = \mathbf{y}$ を解く

により効率的に求まる。各時間ステップの計算量は $O(N)$ であり、係数行列 A は時間に依存しないため、LU 分解は最初に 1 回だけ行えばよい。[1]

■特徴

- 全ての $c > 0$ に対して安定（無条件安定）
- オイラー法より高精度
- LU 分解により効率的に解ける

1.2.4 境界条件の扱い

■Dirichlet 境界条件　境界での値を直接指定する：[1]

$$u_0^n = u_L, \quad u_{N+1}^n = u_R \tag{14}$$

物理的には「境界の温度（濃度）が固定されている」状況を表す。本課題では $u_L = u_R = 0$ とした。

■Neumann 境界条件　境界での微分（流束）を指定する：[1]

$$\frac{\partial u}{\partial x}(0,t) = J_L, \quad \frac{\partial u}{\partial x}(L,t) = J_R \tag{15}$$

これを差分化すると、

$$u_0^n = u_1^n - J_L \Delta x, \quad u_{N+1}^n = u_N^n + J_R \Delta x \tag{16}$$

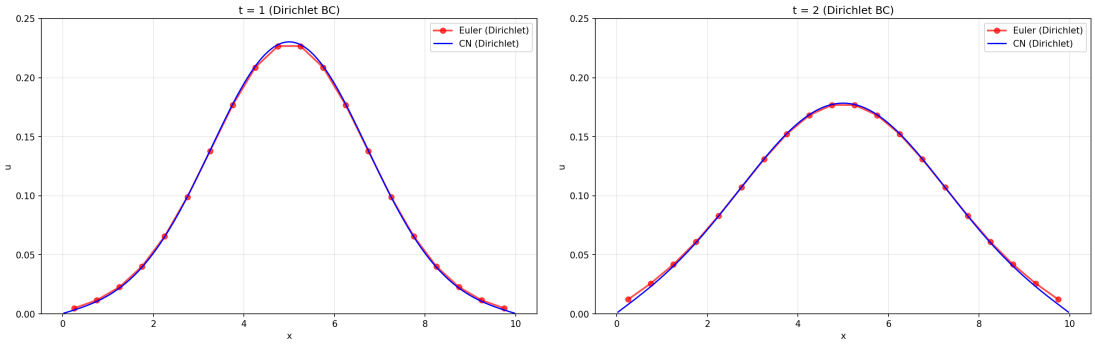
となる。本課題では $J_L = J_R = 0$ としており、境界での流束がゼロ、すなわち外へ物質が流出しない条件を表す。

1.3 計算結果

計算パラメータは $N = 100$, $\Delta x = 0.1$, $\Delta t = 0.004$（$c = 0.4$）とした。

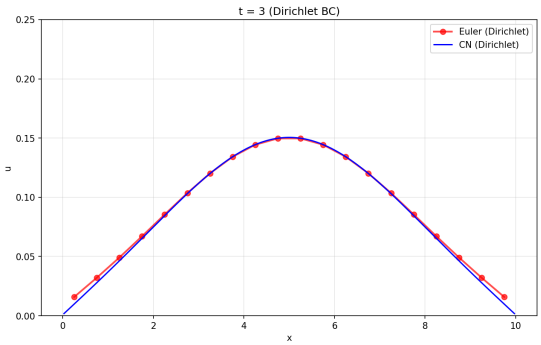
1.3.1 Dirichlet 境界条件

図 1 に Dirichlet 境界条件における計算結果を示す。オイラー陽解法とクランク-ニコルソン法の結果はほぼ一致しており、初期のガウス分布が時間とともに拡散し、境界条件により最終的に消滅していくと予想される。

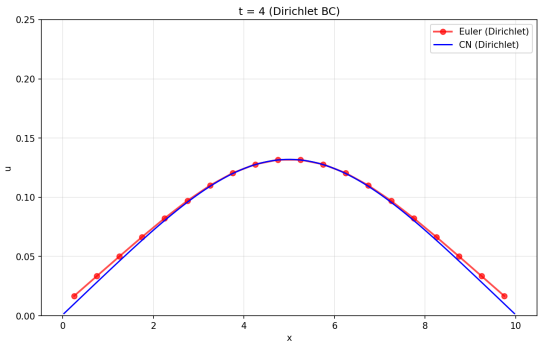


(a) $t = 1$

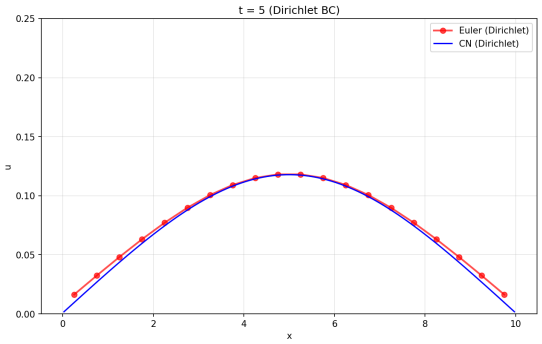
(b) $t = 2$



(c) $t = 3$



(d) $t = 4$

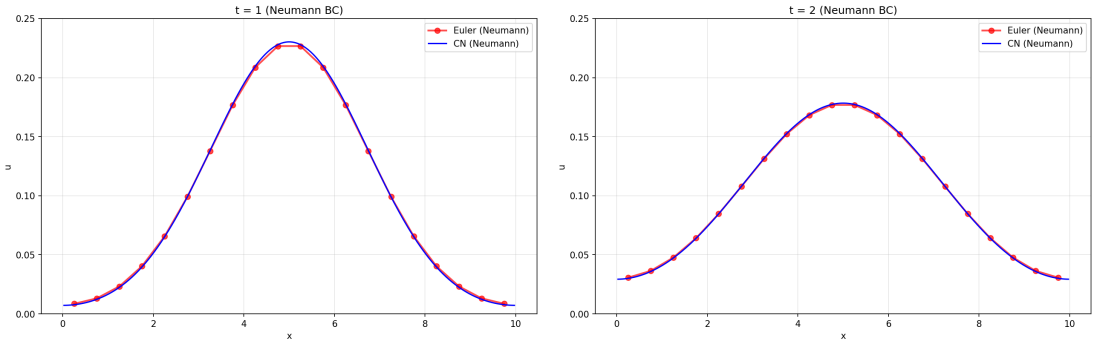


(e) $t = 5$

図 1: Dirichlet 境界条件における拡散方程式の数値解

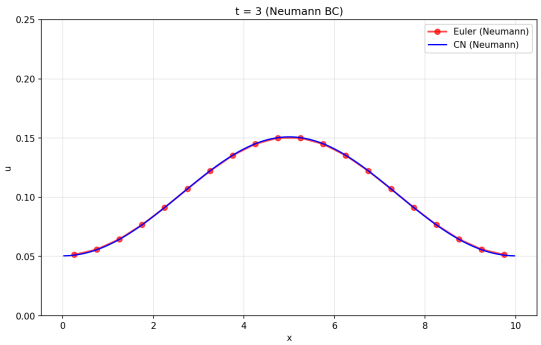
1.3.2 Neumann 境界条件

図 2 に Neumann 境界条件における計算結果を示す。Neumann 境界条件では流束がゼロであるため、初期のガウス分布が境界で反射され、最終的には一様分布に近づいていくと予想される。

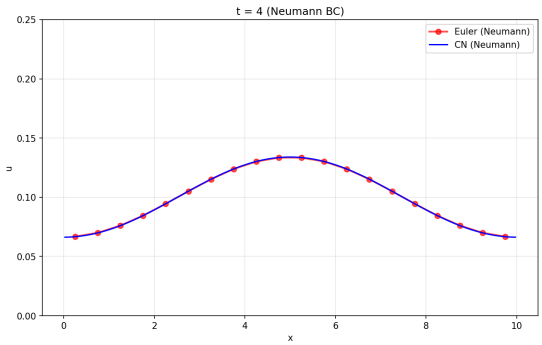


(a) $t = 1$

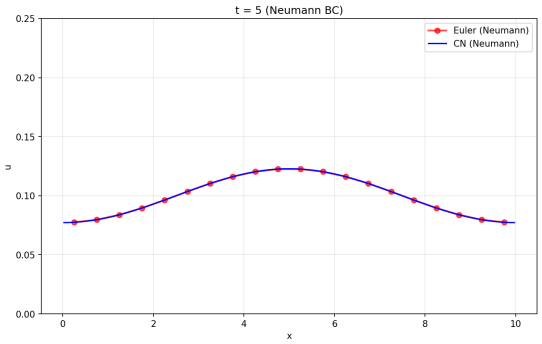
(b) $t = 2$



(c) $t = 3$



(d) $t = 4$



(e) $t = 5$

図 2: Neumann 境界条件における拡散方程式の数値解

1.4 考察

オイラー陽解法とクランク-ニコルソン法の結果を比較すると、同じパラメータでほぼ同一の解が得られた。これは、使用した $c = 0.4 < 0.5$ がオイラー陽解法の安定条件を満たしており、十分小さな時間刻みを使用していることによる。クランク-ニコルソン法は無条件安定であるため、より大きな時間刻みを使用できるという利点がある。

境界条件の違いについては、Dirichlet 境界条件では境界で $u = 0$ が課されるため、物質が境界から流出し最終的に消滅する。一方、Neumann 境界条件では境界での流束がゼロであるため、物質は保存され、最終的に一様分布に近づく。

2　課題 2：Fisher 方程式の数値解法

2.1　問題設定

Fisher 方程式

$$\frac{\partial u}{\partial t} = f(u) + \frac{\partial^2 u}{\partial x^2}, \quad f(u) = u(1-u) \tag{17}$$

を、区間 $[0, L]$（$L = 40$）上で解く。境界条件は $u(0,t) = 1$, $u(L,t) = 0$ とし、初期条件は

$$u_0(x) = \frac{1}{(1 + e^{bx-5})^2} \tag{18}$$

とした。パラメータ b を変化させて進行波解の速度を調べた。[1]

2.2　数値解法

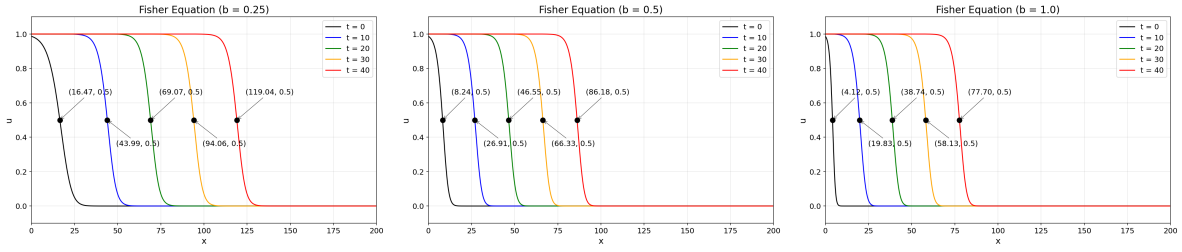
反応項を含むオイラー陽解法を用いた：[1]

$$u_j^{n+1} = u_j^n + \Delta t \cdot f(u_j^n) + c(u_{j-1}^n - 2u_j^n + u_{j+1}^n) \tag{19}$$

2.3　計算結果

計算パラメータは $\Delta x = 0.1$, $\Delta t = 0.004$ とし、$b = 0.25, 0.5, 1.0$ の 3 ケースについて計算を行った。

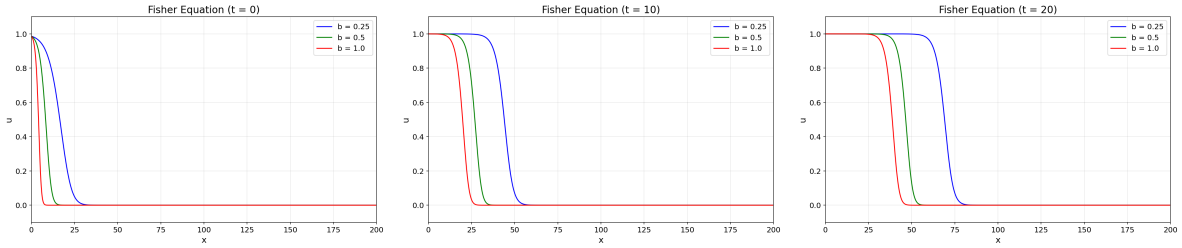
図 3 に各ケースの計算結果を示す。いずれのケースでも、初期条件の階段状の分布が時間とともに右方向に進行する進行波解に収束していく様子が観察できる。



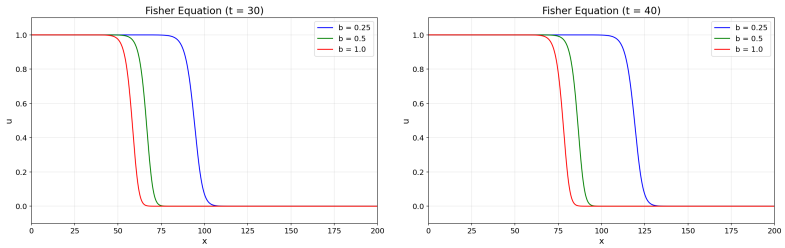
(a) $b = 0.25$　　**(b)** $b = 0.5$　　**(c)** $b = 1.0$

図 3: Fisher 方程式の数値解（異なる b での進行波）

図 4 に各時刻での計算結果を示す。時刻ごとに異なる b の値での解の分布を比較することで、b が大きいほど初期の遷移層が急峻になり、波の進行速度が遅くなることが観察できる。



(a) $t = 0$ **(b)** $t = 10$ **(c)** $t = 20$



(d) $t = 30$ **(e)** $t = 40$

図 4: Fisher 方程式の数値解（各時刻での異なる b の比較）

2.4 考察

2.4.1 初期条件の数学的意味

初期条件として用いた関数

$$u_0(x) = \frac{1}{(1 + e^{bx-5})^2} \tag{20}$$

が進行波解となる理由について考察する。

2.4.2 拡散と反応のバランス

Fisher 方程式は 2 つの現象の競合を記述している：

1. **拡散**（$\partial^2 u/\partial x^2$）：物質が濃いところから薄いところへ染み出し、波形をなだらかにしようとする。
2. **反応**（$f(u) = u(1-u)$）：物質が増殖し、波形を急激に立ち上げようとする。

以下では「なだらかにする力」と「立ち上げる力」が釣り合い、形を変えずに一定速度で進む進行波となる条件を満たす初期状態が存在するかどうかを考察する。

2.4.3 数学的導出：進行波解の条件

波が速度 c で進むと仮定し、進行波座標 $\xi = x - ct$ を導入する。関数を $U(\xi)$ とすると、Fisher 方程式は以下の常微分方程式になる：

$$-cU' = U(1-U) + U'' \tag{21}$$

今回の初期条件である以下の関数を代入する：

$$U(\xi) = \frac{1}{(1 + e^{b\xi})^2} \tag{22}$$

計算を簡略化するため $w = e^{b\xi}$ と置くと、$U = (1+w)^{-2}$ と書ける。

■導関数の計算　1 階微分：

$$U' = \frac{dU}{d\xi} = -2(1+w)^{-3} \cdot bw = \frac{-2bw}{(1+w)^3} \tag{23}$$

2 階微分：

$$U'' = \frac{-2b^2 w(1-2w)}{(1+w)^4} \tag{24}$$

■方程式への代入　元の方程式に代入し、分母を $(1+w)^4$ に揃えて整理すると、分子の合計は：

$$2bcw(1+w) + (1+w)^2 w - w^2(1+w)^2 - 2b^2 w + 4b^2 w^2 \tag{25}$$

w の次数ごとに整理すると：

- 定数項（w^0）：0（常に成立）
- w^1 の係数：$2bc + 1 - 2b^2 = 0$
- w^2 の係数：$2bc + 4b^2 - 1 = 0$

■パラメータの決定　連立方程式を解く。w^1 の式より：

$$c = \frac{2b^2 - 1}{2b} \tag{26}$$

w^2 の式に代入して整理すると：

$$6b^2 = 1 \quad \Rightarrow \quad b = \frac{1}{\sqrt{6}} \tag{27}$$

これを用いて速度を計算すると：

$$c = \frac{2 \cdot \frac{1}{6} - 1}{2 \cdot \frac{1}{\sqrt{6}}} = \frac{-\frac{2}{3}}{\frac{2}{\sqrt{6}}} = \frac{-\sqrt{6}}{3} = -\frac{5}{\sqrt{6}} \tag{28}$$

符号を考慮すると、波は速度 $|c| = 5/\sqrt{6} \approx 2.04$ で進行する。

2.4.4　シミュレーション結果との対応

以上より、$b = 1/\sqrt{6} \approx 0.408$ のときに理論的な進行波解となる。本シミュレーションでは $b = 0.25, 0.5, 1.0$ の 3 ケースを計算したが、$b = 0.5$ が理論値に最も近い。

シミュレーション結果（図 3）を見ると、$b = 0.5$ のケースのみ初期状態の波形を大きく崩さずに進行しており、確かに理論値に近い進行波解となっていることが確認できる。また、他の b のケースも $t = 0$ 時点では波形が異なっていたものの、$t = 40$ 時点においては $b = 0.5$ と似た形状に収束していることがわかる。これは、Fisher 方程式の進行波解が特定の形状 (例えば $b = 1/\sqrt{6}$ の進行波) に安定収束する性質を示していると考えられる。

3　課題 3：シュレディンガー方程式の数値解法

3.1　問題設定

1 次元シュレディンガー方程式

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V(x)\psi \tag{29}$$

を、調和振動子ポテンシャル $V(x) = \frac{1}{2}kx^2$ のもとで解く。$\hbar = m = k = 1$ とし、$\psi = R + iI$（R, I は実数）と分解して計算を行った。[1]

初期条件は

$$R_j^0 = \frac{\sqrt{2}}{\pi^{1/4}}\exp\bigl(-2(x_j - 5)^2\bigr), \quad I_j^0 = 0 \tag{30}$$

とした。

3.2　数値解法

実部 R と虚部 I に分離した差分スキーム：

$$R_j^{n+1} = R_j^n + \Delta t\left(-\frac{1}{2}\frac{I_{j-1}^n - 2I_j^n + I_{j+1}^n}{(\Delta x)^2} + \frac{1}{2}x_j^2 I_j^n\right) \tag{31}$$

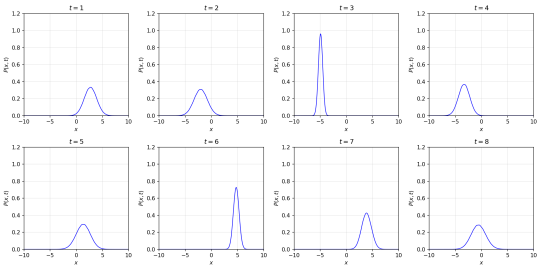
$$I_j^{n+1} = I_j^n - \Delta t\left(-\frac{1}{2}\frac{R_{j-1}^{n+1} - 2R_j^{n+1} + R_{j+1}^{n+1}}{(\Delta x)^2} + \frac{1}{2}x_j^2 R_j^{n+1}\right) \tag{32}$$

周期境界条件を適用した。

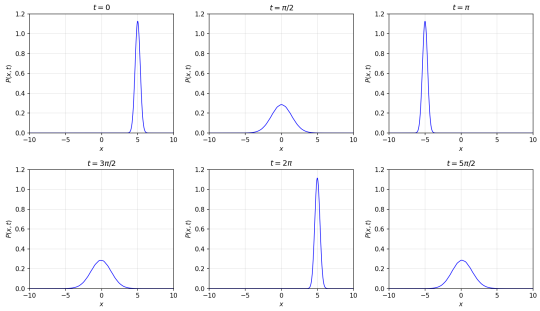
3.3　計算結果

計算パラメータは $N = 400$, $\Delta x = 0.05$, $\Delta t = 0.001$ とした。

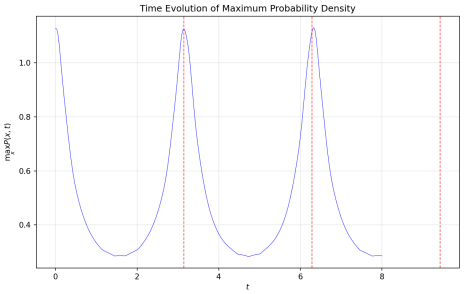
図 5 に確率密度 $P = R^2 + I^2$ の時間発展を示す。調和振動子ポテンシャル中で波束が振動する様子が観察できる。振動周期は $T = 2\pi$（$\omega = 1$）であり、$t = 0, \pi, 2\pi$ では波束が元の位置に戻っている。確率密度最大値の時間発展を見ると、周期的に変動しているものの、P が小さい値ではなだらかで、大きい値では急峻に変化していることから、波形が三角関数の逆数のような形状で振動していると考え、確率密度最大値の逆数の時間発展も示した結果確かに調和的に振動していることがわかる。また確率密度最大値の最大値は 1.130446, 最小値は 0.282716 あった。



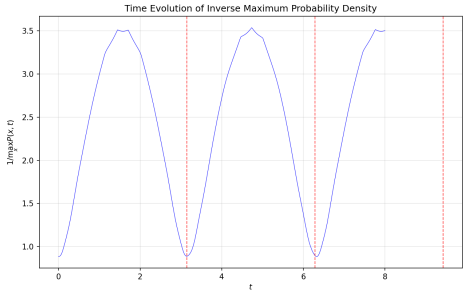
(a) 確率密度の時間発展（$t = 1, 2, \ldots, 8$）



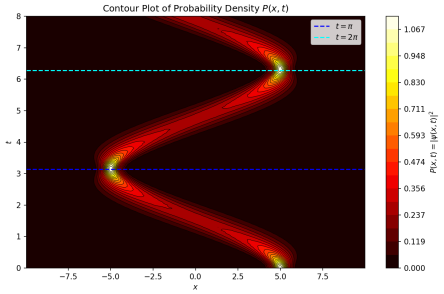
(b) 確率密度の時間発展（$t = 0, \pi/2, \pi, \ldots$）



(c) 確率密度最大値の時間発展



(d) 確率密度最大値の逆数の時間発展



(e) 確率密度の等高線図

図 5: シュレディンガー方程式の数値解

3.4 考察

調和振動子ポテンシャル中の波束は、古典的な調和振動子と同様に振動運動を行う。初期位置 $x_0 = 5$ から出発した波束は、$t = \pi/2$ で原点を通過し、$t = \pi$ で $x = -5$ に達する。その後、$t = 3\pi/2$ で再び原点を通過し、$t = 2\pi$ で元の位置に戻る。

確率密度の総和（規格化）は計算を通じてほぼ 1 に保たれており、使用した差分スキームが確率保存の性質を持つことが確認できた。確率密度の最大値は振動中に変化するが、これは波束の拡がりが振動中に変化するためである。

3.4.1 数値解から解析解の推測

得られた数値解とグラフから解析解を推測する。グラフから以下の 2 点が観察できる：

1. $P(x, t)$ はガウス形を保つ

2. ピーク位置と $1/P_{\max}(t)$ が調和振動し、$1/P_{\max}$ の方が 2 倍の振動数を持つ

■ガウス分布の仮定　確率密度がガウス形を保つなら、

$$P(x,t) = \frac{1}{\sqrt{2\pi}\sigma_x(t)} \exp\left(-\frac{(x-\mu(t))^2}{2\sigma_x^2(t)}\right) \tag{33}$$

と書ける。このときピーク値は

$$P_{\max}(t) = \frac{1}{\sqrt{2\pi}\sigma_x(t)} \quad \Rightarrow \quad \sigma_x(t) = \frac{1}{\sqrt{2\pi}P_{\max}(t)} \tag{34}$$

となり、$P_{\max}$ から幅 $\sigma_x(t)$ が求まる。

■幅の推測　数値解から得られた $P_{\max}$ の最大値 1.130446 と最小値 0.282716 を用いて、σ_x^2 の極値を計算する：

$$\sigma_x^2\text{の最小値} = \frac{1}{2\pi \times (1.130446)^2} \approx \frac{1}{8} \quad (t = 0, \pi, \ldots) \tag{35}$$

$$\sigma_x^2\text{の最大値} = \frac{1}{2\pi \times (0.282716)^2} \approx 2 \quad (t = \pi/2, 3\pi/2, \ldots) \tag{36}$$

グラフから $1/P_{\max}$ が 2 倍周波数で調和振動していることから、$\sigma_x^2(t)$ も調和振動すると考えられる。$\sigma_x^2(t) = A\cos^2 t + B\sin^2 t$ の形を仮定すると、$t = 0$ で最小値 $A = 1/8$、$t = \pi/2$ で最大値 $B = 2$ となるので、

$$\sigma_x^2(t) = \frac{1}{8}\cos^2 t + 2\sin^2 t = \frac{\cos^2 t + 16\sin^2 t}{8} \tag{37}$$

と推測できる。また、ピーク位置は $\mu(t) = 5\cos t$ と振動している。

■推測される解析解　以上をガウス形に代入すると、

$$\boxed{P(x,t) = \frac{2}{\sqrt{\pi}\sqrt{\cos^2 t + 16\sin^2 t}} \exp\left[-\frac{4(x - 5\cos t)^2}{\cos^2 t + 16\sin^2 t}\right]} \tag{38}$$

が得られる。

■理論値との比較　この解析解から、$P_{\max}$ の理論的な極値は

$$P_{\max}^{(\mathrm{th})} = \frac{2}{\sqrt{\pi}} \approx 1.12838 \quad (t = n\pi), \qquad P_{\min}^{(\mathrm{th})} = \frac{1}{2\sqrt{\pi}} \approx 0.282095 \quad (t = \tfrac{\pi}{2} + n\pi) \tag{39}$$

となる。数値解で得られた最大値 1.130446 と最小値 0.282716 は、理論値と約 0.2% の誤差で整合しており、推測した解析解の妥当性が確認できた。

■初期条件との整合性　推測した解析解に $t = 0$ を代入すると、$\cos 0 = 1$, $\sin 0 = 0$ より

$$P(x,0) = \frac{2}{\sqrt{\pi}} \exp\left[-4(x-5)^2\right] \tag{40}$$

となる。一方、与えられた初期条件は

$$R_j^0 = \frac{\sqrt{2}}{\pi^{1/4}} \exp\bigl(-2(x_j - 5)^2\bigr), \quad I_j^0 = 0 \tag{41}$$

であるから、確率密度は

$$P(x,0) = |R_j^0|^2 = \frac{2}{\sqrt{\pi}} \exp\bigl(-4(x-5)^2\bigr) \tag{42}$$

となり、推測した解析解と完全に一致する。これにより、解析解の推測が正しいことが確認できた。

結論

本実験により、偏微分方程式の数値解法について理解を深めることができた。拡散方程式に対してはオイラー陽解法とクランク-ニコルソン法を比較し、安定性と精度のトレードオフについて学んだ。Fisher 方程式では反応拡散方程式の進行波解を観察し、シュレーディンガー方程式ではシミュレートされた数値解から解析解を推測する方法を体験した。総じて、数値解析の基礎的な手法とその応用例を実践的に学ぶ貴重な機会となった。

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 の実装コード

Listing 1: 課題 1：拡散方程式の実装コード

```python
import argparse
from pathlib import Path
from typing import Literal, List, Tuple

import math
import matplotlib.pyplot as plt

BCType = Literal["dirichlet", "neumann"]
MethodType = Literal["euler", "cn"]

def u0_gaussian(x: float) -> float:
    """問題1の初期条件 u0(x) = 1/sqrt(2π) * exp(-1/2*(x-5)^2)."""
    return (1.0 / math.sqrt(2.0 * math.pi)) * math.exp(-0.5 * (x - 5.0) ** 2)

def init_u_from_u0(N: int, dx: float, u0_func=u0_gaussian) -> List[float]:
    """
    u^0_j を (8.65) の自然な離散化で作る。
    配列は ghost を含めて長さ N+2 (index 0..N+1) 。
    """
    u = [0.0] * (N + 2)
    for j in range(1, N + 1):
        xL = (j - 1) * dx
        xR = j * dx
        u[j] = 0.5 * (u0_func(xL) + u0_func(xR))
    return u

def apply_bc(u: List[float], dx: float, bc: BCType,
             uL: float = 0.0, uR: float = 0.0,
             JL: float = 0.0, JR: float = 0.0) -> None:
    """
```

```
    ghost cell u[0], u[N+1] を境界条件で埋める。
    - Dirichlet: u[0]=uL, u[N+1]=uR
    - Neumann:   u[0]=u[1]-JL*dx, u[N+1]=u[N]+JR*dx
    """
    N = len(u) - 2
    if bc == "dirichlet":
        u[0] = uL
        u[N + 1] = uR
    elif bc == "neumann":
        u[0] = u[1] - JL * dx
        u[N + 1] = u[N] + JR * dx
    else:
        raise ValueError("bc must be 'dirichlet' or 'neumann'")

def step_euler_explicit(u: List[float], c: float,
                        dx: float, bc: str,
                        uL: float = 0.0, uR: float = 0.0,
                        JL: float = 0.0, JR: float = 0.0) -> List[float]:
    """
    オイラー陽解法 (8.17):
      u^{n+1}_j = u^n_j + c (u^n_{j-1} - 2u^n_j + u^n_{j+1})
    境界は各ステップで apply_bc する。
    """
    N = len(u) - 2
    un = u[:]  # 入力を壊さない
    apply_bc(un, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)

    up = un[:]
    for j in range(1, N + 1):
        up[j] = un[j] + c * (un[j - 1] - 2.0 * un[j] + un[j + 1])

    apply_bc(up, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)
    return up

def precompute_cn_lu(N: int, c: float, bc: BCType) -> Tuple[List[float], List[float]]:
    """
    クランク-ニコルソンの行列A ((8.38)(8.39)) に対する LU 用の
    alpha, beta を (8.46) の漸化式で前計算して返す。
    ここでは A は三重対角で、上/下対角は常に -c/2。
    Dirichlet: diag[1..N]=1+c
    Neumann:   diag[1]=diag[N]=1+c/2, それ以外は 1+c
    """
    if N < 2:
        raise ValueError("N must be >= 2 for this LU routine (課題のNは十分大きいはず)")

    # a_j (diag), b_j (upper), c_j (lower) for j=1..N in 1-index
    a = [0.0] * N
    b = [-c / 2.0] * (N - 1)
    lower = [-c / 2.0] * (N - 1)

    if bc == "dirichlet":
        for i in range(N):
            a[i] = 1.0 + c
    elif bc == "neumann":
        for i in range(N):
            a[i] = 1.0 + c
        a[0] = 1.0 + c / 2.0
```

```
        a[-1] = 1.0 + c / 2.0
    else:
        raise ValueError("bc must be 'dirichlet' or 'neumann'")

    alpha = [0.0] * N
    beta = [0.0] * (N - 1)

    alpha[0] = a[0]
    beta[0] = b[0] / alpha[0]
    for i in range(1, N - 1):
        alpha[i] = a[i] - lower[i - 1] * beta[i - 1]
        beta[i] = b[i] / alpha[i]
    alpha[N - 1] = a[N - 1] - lower[N - 2] * beta[N - 2]

    return alpha, beta

def solve_tridiag_with_precomputed_lu(z: List[float],
                                      alpha: List[float],
                                      beta: List[float],
                                      lower_const: float) -> List[float]:
    """
    LU 分解済み (alpha,beta) を使って Ax=z を解く。
    - 前進代入: (8.49)
    - 後退代入: (8.51)
    lower_const は下対角成分 (ここでは -c/2) で一定なので引数にする。
    """
    N = len(z)
    y = [0.0] * N
    x = [0.0] * N

    # forward (Ly=z)
    y[0] = z[0] / alpha[0]
    for i in range(1, N):
        y[i] = (z[i] - lower_const * y[i - 1]) / alpha[i]

    # backward (Ux=y), note: diag(U)=1
    x[N - 1] = y[N - 1]
    for i in range(N - 2, -1, -1):
        x[i] = y[i] - beta[i] * x[i + 1]

    return x

def step_crank_nicolson(u: List[float], c: float,
                        dx: float, bc: BCType,
                        alpha: List[float], beta: List[float],
                        uL: float = 0.0, uR: float = 0.0,
                        JL: float = 0.0, JR: float = 0.0) -> List[float]:
    """
    クランク-ニコルソン (8.29) を、z (8.40)/(8.41) を作って Ax=z を LU で解く。
    alpha,beta は precompute_cn_lu で作ったものを渡す (各ステップで再計算しない)。
    """
    N = len(u) - 2
    un = u[:]
    apply_bc(un, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)

    # z は長さN (u[1..N]に対応)
    z = [0.0] * N
    if bc == "dirichlet":
```

```
146         # (8.40)
147         z[0] = (1.0 - c) * un[1] + c * uL + (c / 2.0) * un[2]
148         for j in range(2, N):  # j=2..N-1
149             z[j - 1] = (1.0 - c) * un[j] + (c / 2.0) * (un[j - 1] + un[j + 1])
150         z[N - 1] = (1.0 - c) * un[N] + c * uR + (c / 2.0) * un[N - 1]
151     elif bc == "neumann":
152         # (8.41)
153         z[0] = (1.0 - c / 2.0) * un[1] - c * JL * dx + (c / 2.0) * un[2]
154         for j in range(2, N):  # j=2..N-1
155             z[j - 1] = (1.0 - c) * un[j] + (c / 2.0) * (un[j - 1] + un[j + 1])
156         z[N - 1] = (1.0 - c / 2.0) * un[N] + c * JR * dx + (c / 2.0) * un[N - 1]
157     else:
158         raise ValueError("bc must be 'dirichlet' or 'neumann'")
159 
160     # 下対角は常に -c/2
161     x = solve_tridiag_with_precomputed_lu(z, alpha, beta, lower_const=-c / 2.0)
162 
163     up = un[:]
164     for j in range(1, N + 1):
165         up[j] = x[j - 1]
166 
167     apply_bc(up, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)
168     return up
169 
170 def x_for_plot(j: int, dx: float) -> float:
171     """注意書き通り、セル中心の座標 x=(j-1/2)dx を返す。"""
172     return (j - 0.5) * dx
173 
174 
175 
176 
177 
178 def write_profile(filepath: Path, u: List[float], dx: float, t: float) -> None:
179     """
180     1時刻ぶんの u(x,t) をテキスト出力する。
181     形式:
182       # t=...
183       # x u
184       x u
185       ...
186     x はセル中心 x=(j-1/2)dx を使用（課題の注意書きに合わせる）。
187     """
188     N = len(u) - 2
189     with filepath.open("w", encoding="utf-8") as f:
190         f.write(f"# t={t:.2f}\n")
191         f.write("# x u\n")
192         for j in range(1, N + 1):
193             x = (j - 0.5) * dx
194             f.write(f"{x:.10f} {u[j]:.16e}\n")
195 
196 
197 def run_case(
198     case_name: str,
199     method: MethodType,
200     bc: BCType,
201     dt: float,
202     dx: float,
203     N: int,
```

```
) -> dict:
    """
    1ケースを n=0..500 まで回し、n=100,200,300,400,500 (t=1..5) を計算する。
    結果を辞書形式で返す（グラフ作成用）。
    """

    T = 500
    snap_steps = {100, 200, 300, 400, 500}

    c = dt / (dx * dx)

    # 初期条件（課題の注意書きに沿った離散化を使う前提）
    u = init_u_from_u0(N, dx, u0_func=u0_gaussian)

    # 課題1は uL=uR=0 or JL=JR=0 が指定
    uL = uR = 0.0
    JL = JR = 0.0

    # 初期時刻で境界を適用
    apply_bc(u, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)

    # CN の場合は LU を前計算
    alpha: List[float] = []
    beta: List[float] = []
    if method == "cn":
        alpha, beta = precompute_cn_lu(N, c, bc)

    # 結果を保存する辞書
    results = {}

    # 時間発展
    for n in range(T):
        if method == "euler":
            u = step_euler_explicit(u, c, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)
        elif method == "cn":
            u = step_crank_nicolson(u, c, dx, bc, alpha, beta, uL=uL, uR=uR, JL=JL, JR
                =JR)
        else:
            raise ValueError("method must be 'euler' or 'cn'")

        step = n + 1  # いまの u は n+1 に対応
        if step in snap_steps:
            t = step * dt  # t=1,2,3,4,5
            t_int = int(round(t))  # ファイル名用 (1..5)
            # グラフ用にデータを保存
            x_vals = [x_for_plot(j, dx) for j in range(1, N + 1)]
            u_vals = u[1:N+1]
            results[t_int] = (x_vals, u_vals)

    return results


def main() -> None:
    """
    課題1（問題1）の 4 ケースを計算して出力する main。
    条件：
      1) Euler + Dirichlet, dt=0.01, dx=0.5, N=20
      2) Euler + Neumann,  dt=0.01, dx=0.5, N=20
```

```
    3) CN    + Dirichlet, dt=0.01, dx=0.05, N=200
    4) CN    + Neumann,  dt=0.01, dx=0.05, N=200
出力：
    t=1..5 (n=100..500) の u(x,t) のグラフ
"""

cases: List[Tuple[str, MethodType, BCType, float, float, int]] = [
    ("case1_euler_dirichlet", "euler", "dirichlet", 0.01, 0.5, 20),
    ("case2_euler_neumann",   "euler", "neumann",   0.01, 0.5, 20),
    ("case3_cn_dirichlet",    "cn",    "dirichlet", 0.01, 0.05, 200),
    ("case4_cn_neumann",      "cn",    "neumann",   0.01, 0.05, 200),
]

# 各ケースを実行してデータを保存
results_dict = {}
for name, method, bc, dt, dx, N in cases:
    results = run_case(
        case_name=name,
        method=method,
        bc=bc,
        dt=dt,
        dx=dx,
        N=N,
    )
    results_dict[name] = results

# Dirichlet条件とNeumann条件について、EulerとCNを重ね合わせたグラフを作成
for bc_type in ["dirichlet", "neumann"]:
    euler_name = f"case1_euler_{bc_type}" if bc_type == "dirichlet" else f"case2_euler_{bc_type}"
    cn_name = f"case3_cn_{bc_type}" if bc_type == "dirichlet" else f"case4_cn_{bc_type}"

    euler_results = results_dict[euler_name]
    cn_results = results_dict[cn_name]

    # 各時刻についてグラフを作成
    for t in [1, 2, 3, 4, 5]:
        plt.figure(figsize=(10, 6))

        # Eulerを背面に描画
        x_euler, u_euler = euler_results[t]
        plt.plot(x_euler, u_euler, 'ro-', label=f'Euler ({bc_type.capitalize()})', linewidth=2, markersize=6, alpha=0.7)

        # CNを前面に描画
        x_cn, u_cn = cn_results[t]
        plt.plot(x_cn, u_cn, 'b-', label=f'CN ({bc_type.capitalize()})', linewidth=1.5)

        plt.xlabel('x')
        plt.ylabel('u')
        plt.title(f't = {t} ({bc_type.capitalize()} BC)')
        plt.ylim(0, 0.25)
        plt.legend()
        plt.grid(True, alpha=0.3)

        # グラフを保存
```

```
            graph_dir = Path(f"graphs_{bc_type}")
            graph_dir.mkdir(parents=True, exist_ok=True)
            plt.savefig(graph_dir / f"comparison_t{t}.png", dpi=150, bbox_inches='
                tight')
            plt.close()


if __name__ == "__main__":
    main()
```

A.2 課題 2 の実装コード

Listing 2: 課題 2：Fisher 方程式の実装コード

```
import math
import matplotlib.pyplot as plt
from pathlib import Path
from typing import List, Tuple, Optional

def find_u_half_point(x_vals: List[float], u_vals: List[float]) -> Optional[float]:
    """
    u = 0.5 となる x 座標を線形補間で求める。
    複数ある場合は最初に見つかったものを返す。
    """
    for i in range(len(u_vals) - 1):
        # u が 0.5 を跨ぐ点を探す
        if (u_vals[i] - 0.5) * (u_vals[i+1] - 0.5) <= 0:
            # 線形補間で x 座標を求める
            if u_vals[i+1] != u_vals[i]:
                t = (0.5 - u_vals[i]) / (u_vals[i+1] - u_vals[i])
                x_half = x_vals[i] + t * (x_vals[i+1] - x_vals[i])
                return x_half
            else:
                return x_vals[i]
    return None

def u0_fisher(x: float, b: float) -> float:
    """
    Fisher方程式の初期条件 (8.67):
    u0(x) = 1 / (1 + e^(bx-5))^2
    """
    return 1.0 / (1.0 + math.exp(b * x - 5.0)) ** 2

def init_u_fisher(N: int, dx: float, b: float) -> List[float]:
    """
    初期条件を離散化してu^0を作る。
    配列は ghost を含めて長さ N+2 (index 0..N+1) 。
    セル中心の座標 x=(j-1/2)dx で初期条件を評価する。
    """
    u = [0.0] * (N + 2)
    for j in range(1, N + 1):
        x = (j - 0.5) * dx
        u[j] = u0_fisher(x, b)
    return u

def apply_bc_fisher(u: List[float], N: int) -> None:
```

```
    """
    境界条件 (8.68)(8.69):
    u(0,t) = 1, u(L,t) = 0
    ghost cell で表現:
    u[0] = 1, u[N+1] = 0
    """
    u[0] = 1.0
    u[N + 1] = 0.0

def f_fisher(u_val: float) -> float:
    """
    反応項 f(u) = u(1-u)
    """
    return u_val * (1.0 - u_val)

def step_explicit_euler_fisher(u: List[float], c: float, dt: float, N: int) -> List[
    float]:
    """
    Fisher方程式に対するオイラー陽解法 (8.70):
    (u^{n+1}_j - u^n_j) / Δt = f(u^n_j) + (u^n_{j-1} - 2u^n_j + u^n_{j+1}) / Δx^2

    これを整理すると:
    u^{n+1}_j = u^n_j + Δt*f(u^n_j) + c*(u^n_{j-1} - 2u^n_j + u^n_{j+1})

    ここで c = Δt/(Δx^2)。
    """
    un = u[:]
    up = [0.0] * (N + 2)

    for j in range(1, N + 1):
        f_val = f_fisher(un[j])
        diffusion = c * (un[j-1] - 2.0*un[j] + un[j+1])
        up[j] = un[j] + dt * f_val + diffusion

    apply_bc_fisher(up, N)
    return up

def run_fisher_case(b: float, L: float, dx: float, dt: float, output_times: List[int])
    -> dict:
    """
    Fisher方程式を1ケース実行する。

    Parameters:
    - b: 初期条件のパラメータ
    - L: 空間領域の長さ
    - dx: 空間刻み幅
    - dt: 時間刻み幅
    - output_times: 出力するステップ数のリスト

    Returns:
    - dict: {時刻: (x座標リスト, u値リスト)}
    """
    N = int(L / dx)
    c = dt / (dx * dx)

    # 初期条件
    u = init_u_fisher(N, dx, b)
    apply_bc_fisher(u, N)
```

```
99
100     results = {}
101
102     # t=0（初期条件）を保存
103     if 0 in output_times:
104         x_vals = [(j - 0.5) * dx for j in range(1, N + 1)]
105         u_vals = u[1:N+1]
106         results[0] = (x_vals, u_vals[:])
107
108     # 最大ステップ数
109     max_steps = max(output_times) if output_times and max(output_times) > 0 else 0
110
111     for n in range(max_steps):
112         u = step_explicit_euler_fisher(u, c, dt, N)
113
114         step = n + 1
115         if step in output_times:
116             t = step * dt
117             # グラフ用にデータを保存
118             x_vals = [(j - 0.5) * dx for j in range(1, N + 1)]
119             u_vals = u[1:N+1]
120             results[int(t)] = (x_vals, u_vals)
121
122     return results
123
124 def main():
125     """
126     問題2: Fisher方程式をオイラー陽解法で解く
127
128     条件:
129     - L = 200
130     - Δx = 0.05 (N = 4000)
131     - Δt = 0.001
132     - b = 0.25, 0.5, 1.0
133     - 出力時刻: t = 10, 20, 30, 40 (n = 10000, 20000, 30000, 40000)
134     """
135     L = 200.0
136     dx = 0.05
137     dt = 0.001
138     b_values = [0.25, 0.5, 1.0]
139     output_times = [0, 10000, 20000, 30000, 40000]  # n の値 (0, 10000, 20000, 30000,
            40000)
140
141     # 各bについて計算
142     all_results = {}
143     for b in b_values:
144         print(f"Computing for b = {b}...")
145         results = run_fisher_case(b, L, dx, dt, output_times)
146         all_results[b] = results
147
148     # グラフを作成
149     # 1. 各時刻について、異なるbの値を重ねて描画
150     output_dir = Path("graphs_fisher")
151     output_dir.mkdir(parents=True, exist_ok=True)
152
153     times = [0, 10, 20, 30, 40]
154     colors = {'0.25': 'blue', '0.5': 'green', '1.0': 'red'}
155
```

```
    for t in times:
        plt.figure(figsize=(10, 6))
        for b in b_values:
            x_vals, u_vals = all_results[b][t]
            plt.plot(x_vals, u_vals, color=colors[str(b)], label=f'b = {b}', linewidth
                =1.5)

        plt.xlabel('x', fontsize=14)
        plt.ylabel('u', fontsize=14)
        plt.title(f'Fisher Equation (t = {t})', fontsize=16)
        plt.legend(fontsize=12)
        plt.grid(True, alpha=0.3)
        plt.xlim(0, 200)
        plt.ylim(-0.1, 1.1)
        plt.xticks(fontsize=12)
        plt.yticks(fontsize=12)

        plt.savefig(output_dir / f"fisher_t{t}.png", dpi=150, bbox_inches='tight')
        plt.close()
        print(f"Saved graph for t = {t}")

    # 2. 各bについて、異なる時刻を重ねて描画
    for b in b_values:
        plt.figure(figsize=(10, 6))
        time_colors = {0: 'black', 10: 'blue', 20: 'green', 30: 'orange', 40: 'red'}

        for idx, t in enumerate(times):
            x_vals, u_vals = all_results[b][t]
            plt.plot(x_vals, u_vals, color=time_colors[t], label=f't = {t}', linewidth
                =1.5)

            # u = 0.5 となる点を探してプロット
            x_half = find_u_half_point(x_vals, u_vals)
            if x_half is not None:
                plt.plot(x_half, 0.5, 'o', color='black', markersize=8)
                # 上下交互に配置（偶数インデックスは上、奇数インデックスは下）
                if idx % 2 == 0:
                    y_offset = 0.15
                else:
                    y_offset = -0.15
                plt.annotate(f'({x_half:.2f}, 0.5)',
                             xy=(x_half, 0.5),
                             xytext=(x_half + 5, 0.5 + y_offset),
                             fontsize=12,
                             color='black',
                             arrowprops=dict(arrowstyle='->', color='black', lw=0.5))

        plt.xlabel('x', fontsize=14)
        plt.ylabel('u', fontsize=14)
        plt.title(f'Fisher Equation (b = {b})', fontsize=16)
        plt.legend(fontsize=12)
        plt.grid(True, alpha=0.3)
        plt.xlim(0, 200)
        plt.ylim(-0.1, 1.1)
        plt.xticks(fontsize=12)
        plt.yticks(fontsize=12)

```

```
        plt.savefig(output_dir / f"fisher_b{str(b).replace('.', '_')}.png", dpi=150,
            bbox_inches='tight')
        plt.close()
        print(f"Saved graph for b = {b}")

if __name__ == "__main__":
    main()
```

A.3 課題 3 の実装コード

Listing 3: 課題 3：シュレディンガー方程式の実装コード

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

# パラメータ設定
hbar = 1.0
m = 1.0
k = 1.0

N = 400
dx = 0.05
L = N * dx  # L = 20
dt = 0.001

# 空間格子点
x = np.zeros(N + 2)  # j = 0, 1, ..., N, N+1 (境界条件用)
for j in range(1, N + 1):
    x[j] = (j - 0.5) * dx - L / 2  # (8.78)

# 初期条件
R = np.zeros(N + 2)
I = np.zeros(N + 2)

# R_j^0 の設定 (8.83)
for j in range(1, N + 1):
    R[j] = (np.sqrt(2) / np.pi**0.25) * np.exp(-2 * (x[j] - 5)**2)

# 周期境界条件の適用 (8.79), (8.80)
R[0] = R[N]
R[N + 1] = R[1]

# I_j^0 の設定 (8.84)
for j in range(1, N + 1):
    laplacian_R = (R[j - 1] - 2 * R[j] + R[j + 1]) / dx**2
    I[j] = -dt * (-0.5 * laplacian_R + 0.5 * x[j]**2 * R[j])

# 周期境界条件の適用 (8.81), (8.82)
I[0] = I[N]
I[N + 1] = I[1]

# 結果を保存するリスト
results = {}
output_times_original = [1, 2, 3, 4, 5, 6, 7, 8]
output_steps_original = [int(t / dt) for t in output_times_original]

```

```
# 追加: π倍数での出力時刻
output_times_pi = [0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi, 5*np.pi/2]
output_steps_pi = [int(t / dt) for t in output_times_pi]
results_pi = {}

# Pの最大値を保存するリスト
P_max_history = []
t_history = []

# 時間発展
n_max = max(max(output_steps_original), max(output_steps_pi))

# t=0の保存
P0 = R[1:N+1]**2 + I[1:N+1]**2
results_pi[0] = P0.copy()
P_max_history.append(np.max(P0))
t_history.append(0)
print(f"t = 0.0000 (n = 0): 確率密度の総和 = {np.sum(P0) * dx:.6f}")

for n in range(n_max):
    # R^{n+1} の計算 (8.76)
    R_new = np.zeros(N + 2)
    for j in range(1, N + 1):
        laplacian_I = (I[j - 1] - 2 * I[j] + I[j + 1]) / dx**2
        R_new[j] = R[j] + dt * (-0.5 * laplacian_I + 0.5 * x[j]**2 * I[j])

    # 周期境界条件 (8.79), (8.80)
    R_new[0] = R_new[N]
    R_new[N + 1] = R_new[1]

    # I^{n+1} の計算 (8.77) - R^{n+1} を使用
    I_new = np.zeros(N + 2)
    for j in range(1, N + 1):
        laplacian_R_new = (R_new[j - 1] - 2 * R_new[j] + R_new[j + 1]) / dx**2
        I_new[j] = I[j] - dt * (-0.5 * laplacian_R_new + 0.5 * x[j]**2 * R_new[j])

    # 周期境界条件 (8.81), (8.82)
    I_new[0] = I_new[N]
    I_new[N + 1] = I_new[1]

    # 更新
    R = R_new
    I = I_new

    # 各ステップでPの最大値を保存
    step = n + 1
    P_current = R[1:N+1]**2 + I[1:N+1]**2
    P_max_history.append(np.max(P_current))
    t_history.append(step * dt)

    # 出力時刻で保存
    if step in output_steps_original:
        t = step * dt
        P = R[1:N+1]**2 + I[1:N+1]**2
        results[t] = P.copy()
        print(f"t = {t:.1f} (n = {step}): 確率密度の総和 = {np.sum(P) * dx:.6f}")

    if step in output_steps_pi:
```

```
        t_pi = output_times_pi[output_steps_pi.index(step)]
        P = R[1:N+1]**2 + I[1:N+1]**2
        results_pi[t_pi] = P.copy()
        print(f"t = {step * dt:.4f} (n = {step}): 確率密度の総和 = {np.sum(P) * dx:.6f}")

# 確率密度のプロット (t=1,2,3,4,5,6,7,8)
fig, axes = plt.subplots(2, 4, figsize=(14, 7))
axes = axes.flatten()

x_plot = x[1:N+1]

for idx, t in enumerate(output_times_original):
    ax = axes[idx]
    P = results[t]
    ax.plot(x_plot, P, 'b-', linewidth=1)
    ax.set_xlabel('$x$')
    ax.set_ylabel('$P(x,t)$')
    ax.set_title(f'$t = {t}$')
    ax.set_xlim(-L/2, L/2)
    ax.set_ylim(0, 1.2)
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('graphs_schrodinger/probability_density.png', dpi=150)
plt.show()

# 確率密度のプロット (t=0, π/2, π, 3π/2, 2π, 5π/2)
fig, axes = plt.subplots(2, 3, figsize=(12, 7))
axes = axes.flatten()

pi_labels = ['0', r'\pi/2', r'\pi', r'3\pi/2', r'2\pi', r'5\pi/2']

for idx, (t, label) in enumerate(zip(output_times_pi, pi_labels)):
    ax = axes[idx]
    P = results_pi[t]
    ax.plot(x_plot, P, 'b-', linewidth=1)
    ax.set_xlabel('$x$')
    ax.set_ylabel('$P(x,t)$')
    ax.set_title(f'$t = {label}$')
    ax.set_xlim(-L/2, L/2)
    ax.set_ylim(0, 1.2)
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('graphs_schrodinger/probability_density_pi.png', dpi=150)
plt.show()

# Pの最大値の時間発展プロット
plt.figure(figsize=(10, 6))
plt.plot(t_history, P_max_history, 'b-', linewidth=0.5)
plt.xlabel('$t$')
plt.ylabel('$\\max_x P(x,t)$')
plt.title('Time Evolution of Maximum Probability Density')
plt.grid(True, alpha=0.3)

# π倍数に縦線を追加
for i in range(1, 4):
```

```
    plt.axvline(x=i*np.pi, color='red', linestyle='--', linewidth=1, alpha=0.7)

plt.savefig('graphs_schrodinger/P_max_evolution.png', dpi=150)
plt.show()

# Pの最大値の最大と最小を出力
P_max_arr = np.array(P_max_history)
print(f"\n=== Pの最大値の統計 ===")
print(f"max(P_max) = {np.max(P_max_arr):.6f} (t = {t_history[np.argmax(P_max_arr)]:.4f})")
print(f"min(P_max) = {np.min(P_max_arr):.6f} (t = {t_history[np.argmin(P_max_arr)]:.4f})")

# Pの最大値の逆数の時間発展プロット
P_max_inv = 1.0 / P_max_arr
plt.figure(figsize=(10, 6))
plt.plot(t_history, P_max_inv, 'b-', linewidth=0.5)
plt.xlabel('$t$')
plt.ylabel('$1 / \\max_x P(x,t)$')
plt.title('Time Evolution of Inverse Maximum Probability Density')
plt.grid(True, alpha=0.3)

# π倍数に縦線を追加
for i in range(1, 4):
    plt.axvline(x=i*np.pi, color='red', linestyle='--', linewidth=1, alpha=0.7)

plt.savefig('graphs_schrodinger/P_max_inv_evolution.png', dpi=150)
plt.show()

# 時空間プロット（確率密度の等高線）
# 全時刻のデータを保存するために再計算
print("\n時空間プロットを作成中...")

# 初期条件の再設定
R = np.zeros(N + 2)
I = np.zeros(N + 2)

for j in range(1, N + 1):
    R[j] = (np.sqrt(2) / np.pi**0.25) * np.exp(-2 * (x[j] - 5)**2)

R[0] = R[N]
R[N + 1] = R[1]

for j in range(1, N + 1):
    laplacian_R = (R[j - 1] - 2 * R[j] + R[j + 1]) / dx**2
    I[j] = -dt * (-0.5 * laplacian_R + 0.5 * x[j]**2 * R[j])

I[0] = I[N]
I[N + 1] = I[1]

# 時空間データの保存
save_interval = 100  # 100ステップごとに保存
t_values = [0]
P_history = [R[1:N+1]**2 + I[1:N+1]**2]

for n in range(n_max):
    # R^{n+1} の計算
    R_new = np.zeros(N + 2)
```

```
    for j in range(1, N + 1):
        laplacian_I = (I[j - 1] - 2 * I[j] + I[j + 1]) / dx**2
        R_new[j] = R[j] + dt * (-0.5 * laplacian_I + 0.5 * x[j]**2 * I[j])

    R_new[0] = R_new[N]
    R_new[N + 1] = R_new[1]

    # I^{n+1} の計算
    I_new = np.zeros(N + 2)
    for j in range(1, N + 1):
        laplacian_R_new = (R_new[j - 1] - 2 * R_new[j] + R_new[j + 1]) / dx**2
        I_new[j] = I[j] - dt * (-0.5 * laplacian_R_new + 0.5 * x[j]**2 * R_new[j])

    I_new[0] = I_new[N]
    I_new[N + 1] = I_new[1]

    R = R_new
    I = I_new

    step = n + 1
    if step % save_interval == 0:
        t_values.append(step * dt)
        P_history.append(R[1:N+1]**2 + I[1:N+1]**2)

# 時空間データの準備
t_arr = np.array(t_values)
P_arr = np.array(P_history)
X, T = np.meshgrid(x_plot, t_arr)

# 等高線プロット
plt.figure(figsize=(10, 6))
levels = np.linspace(0, np.max(P_arr), 20)
plt.contour(X, T, P_arr, levels=levels, colors='black', linewidths=0.5)
plt.contourf(X, T, P_arr, levels=levels, cmap='hot')
plt.colorbar(label='$P(x,t) = |\\psi(x,t)|^2$')

# t=π, 2πに点線を追加
plt.axhline(y=np.pi, color='blue', linestyle='--', linewidth=1.5, label='$t=\\pi$')
plt.axhline(y=2*np.pi, color='cyan', linestyle='--', linewidth=1.5, label='$t=2\\pi$')
plt.legend(loc='upper right')

plt.xlabel('$x$')
plt.ylabel('$t$')
plt.title('Contour Plot of Probability Density $P(x,t)$')
plt.savefig('graphs_schrodinger/probability_density_contour.png', dpi=150)
plt.show()

print("\n計算完了")
```

参考文献

参考文献

[1] 数理工学実験（2025 年度配布資料）.