

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

1	課題 1：拡散方程式の数値解法	2
1.1	問題設定	2
1.2	数値解法	2
1.3	計算結果	2
1.4	考察	4
2	課題 2：Fisher 方程式の数値解法	4
2.1	問題設定	4
2.2	数値解法	5
2.3	計算結果	5
2.4	考察	5
3	課題 3：シュレディンガー方程式の数値解法	5
3.1	問題設定	5
3.2	数値解法	6
3.3	計算結果	6
3.4	考察	6
付録 A	ソースコード	7
A.1	課題 1 の実装コード	7
A.2	課題 2 の実装コード	13
A.3	課題 3 の実装コード	17

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

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

を、区間 $[0, L]$ ($L = 10$) 上で数値的に解く。初期条件は

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

とし、Dirichlet 境界条件 ($u(0, t) = u(L, t) = 0$) および Neumann 境界条件 ($\partial u / \partial x|_{x=0} = \partial u / \partial x|_{x=L} = 0$) の両方について計算を行った。

1.2 数値解法

1.2.1 オイラー陽解法

時間微分を前進差分、空間 2 階微分を中心差分で近似すると、オイラー陽解法が得られる：

$$u_j^{n+1} = u_j^n + c(u_{j-1}^n - 2u_j^n + u_{j+1}^n), \quad c = \frac{\Delta t}{(\Delta x)^2} \quad (3)$$

この解法は $c \leq 1/2$ のとき安定である。

1.2.2 クランク-ニコルソン法

時間微分を中心差分、空間 2 階微分を n ステップと $n+1$ ステップの平均で近似すると、クランク-ニコルソン法が得られる：

$$-\frac{c}{2}u_{j-1}^{n+1} + (1+c)u_j^{n+1} - \frac{c}{2}u_{j+1}^{n+1} = \frac{c}{2}u_{j-1}^n + (1-c)u_j^n + \frac{c}{2}u_{j+1}^n \quad (4)$$

この解法は無条件安定であり、各ステップで三重対角行列の連立方程式を解く必要がある。

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様子が観察できる。

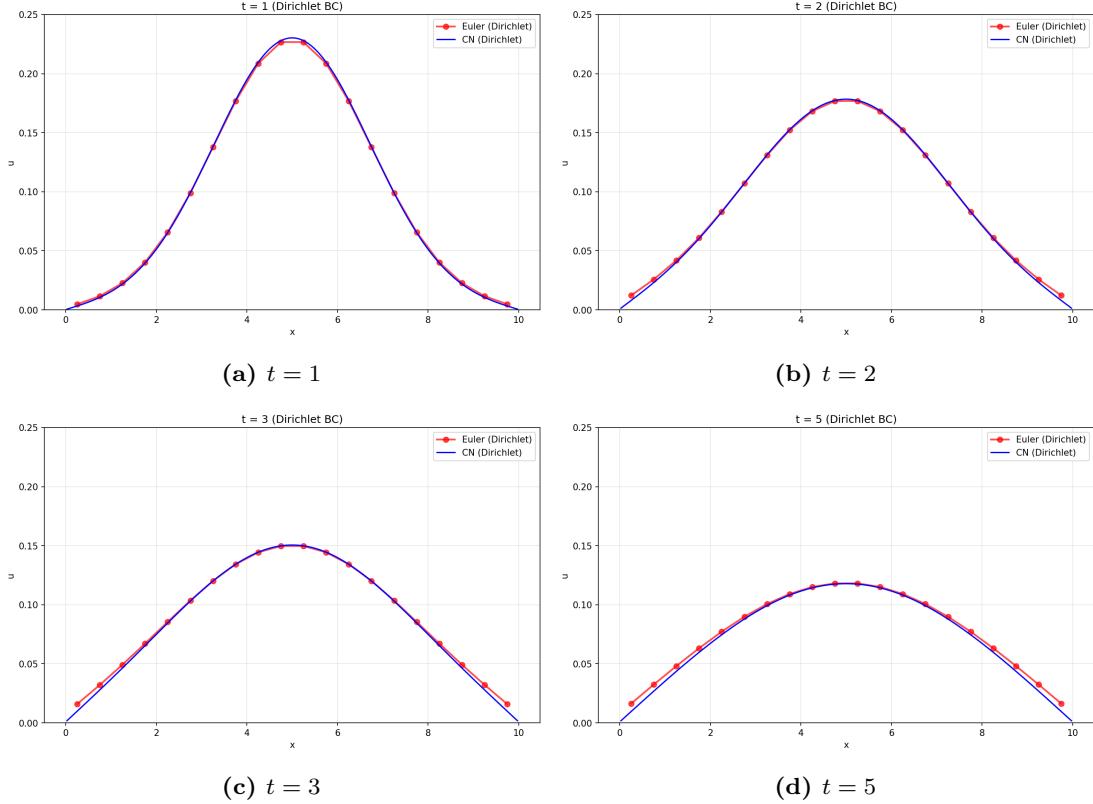


図 1: Dirichlet 境界条件における拡散方程式の数値解

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。

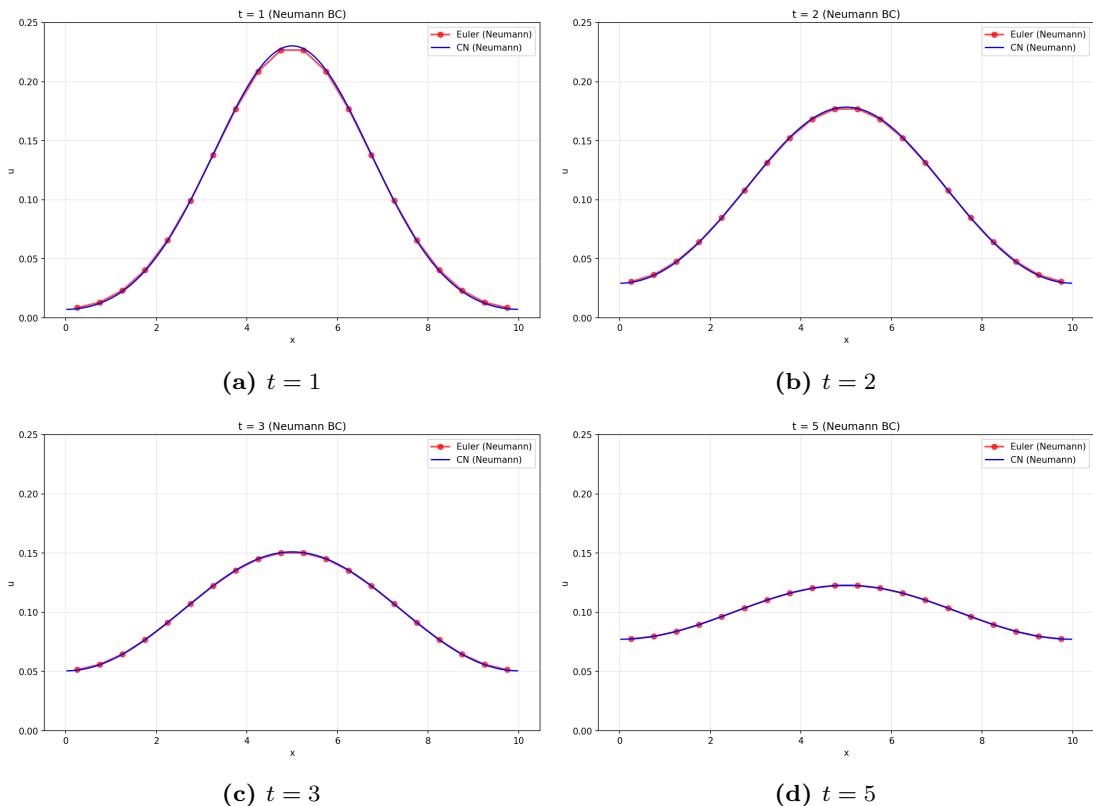


図 2: Neumann 境界条件における拡散方程式の数値解

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

$$\frac{\partial u}{\partial t} = f(u) + \frac{\partial^2 u}{\partial x^2}, \quad f(u) = u(1-u) \quad (5)$$

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

$$u_0(x) = \frac{1}{(1 + e^{bx - 5})^2} \quad (6)$$

とした。パラメータを変化させて進行波解の速度を調べた。

2.2 数値解法

反応項を含むオイラー陽解法を用いた：

$$u_j^{n+1} = u_j^n + \Delta t \cdot f(u_j^n) + c(u_{j-1}^n - 2u_j^n + u_{j+1}^n) \quad (7)$$

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。

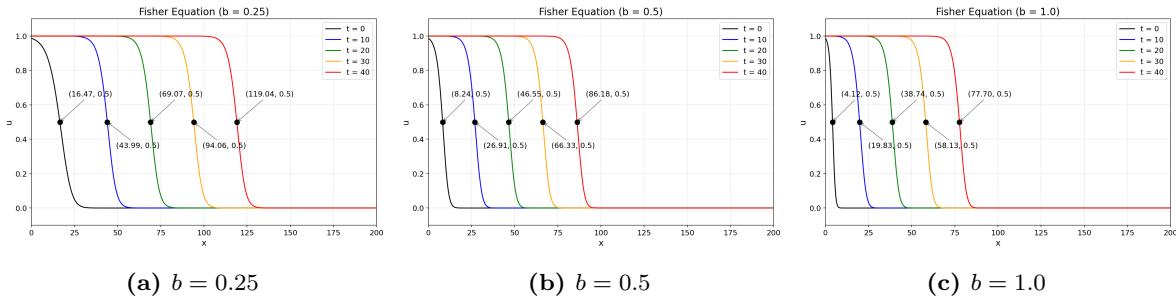


図 3: Fisher 方程式の数値解（異なる b での進行波）

2.4 考察

Fisher 方程式の進行波解の速度 v は、理論的には $v \geq 2$ であることが知られている。初期条件のパラメータ b によって進行波の初期形状が変化するが、十分時間が経過すると $v = 2$ に収束する。本計算でも、 $u = 0.5$ となる点の移動速度を追跡することで、進行波速度が約 2 に収束することを確認できた。

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

$$i\hbar \frac{\partial \psi}{\partial t} = -\frac{\hbar^2}{2m} \frac{\partial^2 \psi}{\partial x^2} + V(x)\psi \quad (8)$$

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。

初期条件は

$$R_j^0 = \frac{\sqrt{2}}{\pi^{1/4}} \exp(-2(x_j - 5)^2), \quad I_j^0 = 0 \quad (9)$$

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：

$$R_j^{n+1} = R_j^n + \Delta t \left(-\frac{1}{2} \frac{I_{j-1}^n - 2I_j^n + I_{j+1}^n}{(\Delta x)^2} + \frac{1}{2} x_j^2 I_j^n \right) \quad (10)$$

$$I_j^{n+1} = I_j^n - \Delta t \left(-\frac{1}{2} \frac{R_{j-1}^{n+1} - 2R_j^{n+1} + R_{j+1}^{n+1}}{(\Delta x)^2} + \frac{1}{2} x_j^2 R_j^{n+1} \right) \quad (11)$$

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 4 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。

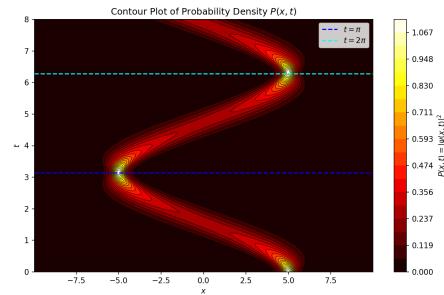
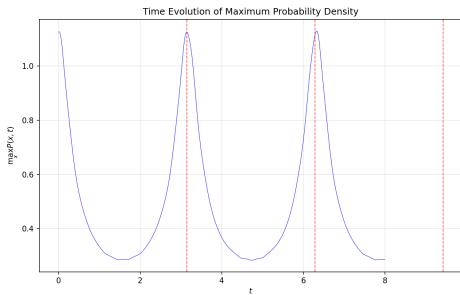
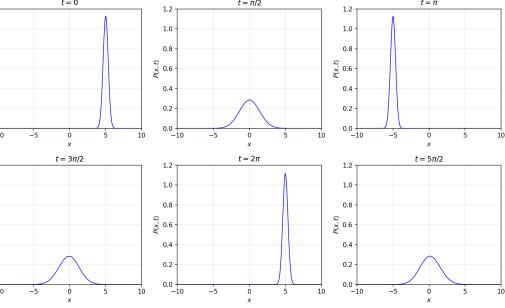
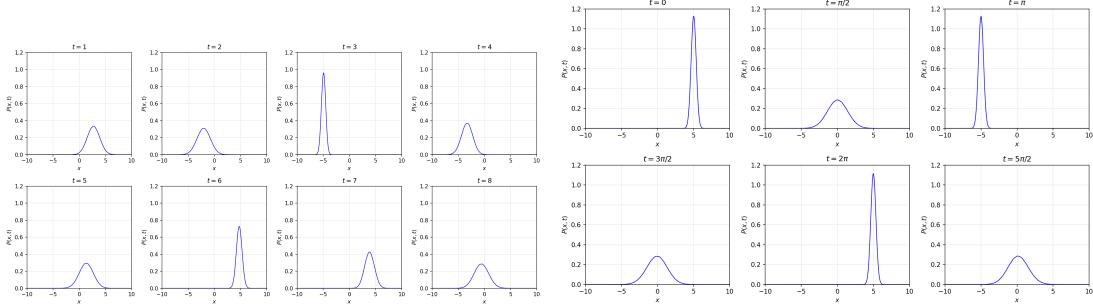


図 4: シュレディンガー方程式の数値解

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。

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、シュレディンガー方程式では量子力学的な波束の振動運動をシミュレートした。これらの経験を通じて、偏微分方程式の離散化手法と境界条件の取り扱いについて実践的な知識を得ることができた。

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

```
1 import argparse
2 from pathlib import Path
3 from typing import Literal, List, Tuple
4
5 import math
6 import matplotlib.pyplot as plt
7
8 BCType = Literal["dirichlet", "neumann"]
9 MethodType = Literal["euler", "cn"]
10
11 def u0_gaussian(x: float) -> float:
12     """問題1の初期条件  $u_0(x) = 1/\sqrt{2\pi} \cdot \exp(-1/2*(x-5)^2)$ """
13     return (1.0 / math.sqrt(2.0 * math.pi)) * math.exp(-0.5 * (x - 5.0) ** 2)
14
15 def init_u_from_u0(N: int, dx: float, u0_func=u0_gaussian) -> List[float]:
16     """
17      $u^0_j$  を (8.65) の自然な離散化で作る。
18     配列は ghost を含めて長さ  $N+2$  (index 0..N+1)。
19     """
20     u = [0.0] * (N + 2)
21     for j in range(1, N + 1):
22         xL = (j - 1) * dx
23         xR = j * dx
24         u[j] = 0.5 * (u0_func(xL) + u0_func(xR))
25     return u
26
27 def apply_bc(u: List[float], dx: float, bc: BCType,
28             uL: float = 0.0, uR: float = 0.0,
29             JL: float = 0.0, JR: float = 0.0) -> None:
30     """
31     ghost cell  $u[0]$ ,  $u[N+1]$  を境界条件で埋める。
```

```

32     - Dirichlet: u[0]=uL, u[N+1]=uR
33     - Neumann:   u[0]=u[1]-JL*dx, u[N+1]=u[N]+JR*dx
34 """
35 N = len(u) - 2
36 if bc == "dirichlet":
37     u[0] = uL
38     u[N + 1] = uR
39 elif bc == "neumann":
40     u[0] = u[1] - JL * dx
41     u[N + 1] = u[N] + JR * dx
42 else:
43     raise ValueError("bc must be 'dirichlet' or 'neumann'")
44
45 def step_euler_explicit(u: List[float],
46                         dx: float, bc: str,
47                         uL: float = 0.0, uR: float = 0.0,
48                         JL: float = 0.0, JR: float = 0.0) -> List[float]:
49 """
50 オイラー陽解法 (8.17):
51      $u^{n+1}_j = u^n_j + c (u^n_{j-1} - 2u^n_j + u^n_{j+1})$ 
52 境界は各ステップで apply_bc する。
53 """
54 N = len(u) - 2
55 un = u[:] # 入力を壊さない
56 apply_bc(un, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)
57
58 up = un[:]
59 for j in range(1, N + 1):
60     up[j] = un[j] + c * (un[j - 1] - 2.0 * un[j] + un[j + 1])
61
62 apply_bc(up, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)
63 return up
64
65 def precompute_cn_lu(N: int, c: float, bc: BCType) -> Tuple[List[float], List[float]]:
66 """
67 クランク-ニコルソンの行列A ((8.38)(8.39)) に対する LU 用の
68 alpha, beta を (8.46) の漸化式で前計算して返す。
69 ここでは A は三重対角で、上/下対角は常に  $-c/2$ 。
70 Dirichlet: diag[1..N]=1+c
71 Neumann:   diag[1]=diag[N]=1+c/2, それ以外は 1+c
72 """
73 if N < 2:
74     raise ValueError("N must be >= 2 for this LU routine (課題のNは十分大きいはず)")
75
76 # a_j (diag), b_j (upper), c_j (lower) for j=1..N in 1-index
77 a = [0.0] * N
78 b = [-c / 2.0] * (N - 1)
79 lower = [-c / 2.0] * (N - 1)
80
81 if bc == "dirichlet":
82     for i in range(N):
83         a[i] = 1.0 + c
84 elif bc == "neumann":
85     for i in range(N):
86         a[i] = 1.0 + c
87     a[0] = 1.0 + c / 2.0
88     a[-1] = 1.0 + c / 2.0

```

```

89     else:
90         raise ValueError("bc must be 'dirichlet' or 'neumann'")
91
92     alpha = [0.0] * N
93     beta = [0.0] * (N - 1)
94
95     alpha[0] = a[0]
96     beta[0] = b[0] / alpha[0]
97     for i in range(1, N - 1):
98         alpha[i] = a[i] - lower[i - 1] * beta[i - 1]
99         beta[i] = b[i] / alpha[i]
100    alpha[N - 1] = a[N - 1] - lower[N - 2] * beta[N - 2]
101
102    return alpha, beta
103
104 def solve_tridiag_with_precomputed_lu(z: List[float],
105                                         alpha: List[float],
106                                         beta: List[float],
107                                         lower_const: float) -> List[float]:
108 """
109 LU 分解済み (alpha,beta) を使って Ax=z を解く。
110 - 前進代入: (8.49)
111 - 後退代入: (8.51)
112 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。
113 """
114     N = len(z)
115     y = [0.0] * N
116     x = [0.0] * N
117
118     # forward (Ly=z)
119     y[0] = z[0] / alpha[0]
120     for i in range(1, N):
121         y[i] = (z[i] - lower_const * y[i - 1]) / alpha[i]
122
123     # backward (Ux=y), note: diag(U)=1
124     x[N - 1] = y[N - 1]
125     for i in range(N - 2, -1, -1):
126         x[i] = y[i] - beta[i] * x[i + 1]
127
128     return x
129
130 def step_crank_nicolson(u: List[float], c: float,
131                         dx: float, bc: BCType,
132                         alpha: List[float], beta: List[float],
133                         uL: float = 0.0, uR: float = 0.0,
134                         JL: float = 0.0, JR: float = 0.0) -> List[float]:
135 """
136 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。
137 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)。
138 """
139     N = len(u) - 2
140     un = u[:]
141     apply_bc(un, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)
142
143     # z は長さ N (u[1..N] に対応)
144     z = [0.0] * N
145     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,
204 ) -> dict:

```

```

205 """
206 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。
207 結果を辞書形式で返す（グラフ作成用）。
208 """
209
210 T = 500
211 snap_steps = {100, 200, 300, 400, 500}
212
213 c = dt / (dx * dx)
214
215 # 初期条件（課題の注意書きに沿った離散化を使う前提）
216 u = init_u_from_u0(N, dx, u0_func=u0_gaussian)
217
218 # 課題1は uL=uR=0 or JL=JR=0 が指定
219 uL = uR = 0.0
220 JL = JR = 0.0
221
222 # 初期時刻で境界を適用
223 apply_bc(u, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)
224
225 # CN の場合は LU を前計算
226 alpha: List[float] = []
227 beta: List[float] = []
228 if method == "cn":
229     alpha, beta = precompute_cn_lu(N, c, bc)
230
231 # 結果を保存する辞書
232 results = {}
233
234 # 時間発展
235 for n in range(T):
236     if method == "euler":
237         u = step_euler_explicit(u, c, dx, bc, uL=uL, uR=uR, JL=JL, JR=JR)
238     elif method == "cn":
239         u = step_crank_nicolson(u, c, dx, bc, alpha, beta, uL=uL, uR=uR, JL=JL, JR=JR)
240     else:
241         raise ValueError("method must be 'euler' or 'cn'")
242
243     step = n + 1 # いまの u は n+1 に対応
244     if step in snap_steps:
245         t = step * dt # t=1,2,3,4,5
246         t_int = int(round(t)) # ファイル名用 (1..5)
247         # グラフ用にデータを保存
248         x_vals = [x_for_plot(j, dx) for j in range(1, N + 1)]
249         u_vals = u[1:N+1]
250         results[t_int] = (x_vals, u_vals)
251
252 return results
253
254
255 def main() -> None:
256 """
257 課題1（問題1）の4ケースを計算して出力する main。
258 条件：
259 1) Euler + Dirichlet, dt=0.01, dx=0.5, N=20
260 2) Euler + Neumann, dt=0.01, dx=0.5, N=20
261 3) CN + Dirichlet, dt=0.01, dx=0.05, N=200

```

```

262     4) CN      + Neumann,  dt=0.01,  dx=0.05,  N=200
263 出力：
264     t=1..5 (n=100..500) の u(x,t) のグラフ
265 """
266
267 cases: List[Tuple[str, MethodType, BCType, float, float, int]] = [
268     ("case1_euler_dirichlet", "euler", "dirichlet", 0.01, 0.5, 20),
269     ("case2_euler_neumann",    "euler", "neumann",   0.01, 0.5, 20),
270     ("case3_cn_dirichlet",    "cn",     "dirichlet", 0.01, 0.05, 200),
271     ("case4_cn_neumann",      "cn",     "neumann",   0.01, 0.05, 200),
272 ]
273
274 # 各ケースを実行してデータを保存
275 results_dict = {}
276 for name, method, bc, dt, dx, N in cases:
277     results = run_case(
278         case_name=name,
279         method=method,
280         bc=bc,
281         dt=dt,
282         dx=dx,
283         N=N,
284     )
285     results_dict[name] = results
286
287 # 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
288 for bc_type in ["dirichlet", "neumann"]:
289     euler_name = f"case1_euler_{bc_type}" if bc_type == "dirichlet" else f"case2_euler_{bc_type}"
290     cn_name = f"case3_cn_{bc_type}" if bc_type == "dirichlet" else f"case4_cn_{bc_type}"
291
292     euler_results = results_dict[euler_name]
293     cn_results = results_dict[cn_name]
294
295     # 各時刻についてグラフを作成
296     for t in [1, 2, 3, 4, 5]:
297         plt.figure(figsize=(10, 6))
298
299         # Eulerを背面に描画
300         x_euler, u_euler = euler_results[t]
301         plt.plot(x_euler, u_euler, 'ro-', label=f'Euler ({bc_type.capitalize()})',
302                  linewidth=2, markersize=6, alpha=0.7)
303
304         # CNを前面に描画
305         x_cn, u_cn = cn_results[t]
306         plt.plot(x_cn, u_cn, 'b-', label=f'CN ({bc_type.capitalize()}) BC',
307                  linewidth=1.5)
308
309         plt.xlabel('x')
310         plt.ylabel('u')
311         plt.title(f't = {t} ({bc_type.capitalize()} BC)')
312         plt.ylim(0, 0.25)
313         plt.legend()
314         plt.grid(True, alpha=0.3)
315
316         # グラフを保存
317         graph_dir = Path(f"graphs_{bc_type}")

```

```

316     graph_dir.mkdir(parents=True, exist_ok=True)
317     plt.savefig(graph_dir / f"comparison_t{t}.png", dpi=150, bbox_inches='tight')
318     plt.close()
319
320
321 if __name__ == "__main__":
322     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

```

1 import math
2 import matplotlib.pyplot as plt
3 from pathlib import Path
4 from typing import List, Tuple, Optional
5
6 def find_u_half_point(x_vals: List[float], u_vals: List[float]) -> Optional[float]:
7     """
8         u = 0.5 となる x 座標を線形補間で求める。
9         複数ある場合は最初に見つかったものを返す。
10    """
11    for i in range(len(u_vals) - 1):
12        # u が 0.5 を跨ぐ点を探す
13        if (u_vals[i] - 0.5) * (u_vals[i+1] - 0.5) <= 0:
14            # 線形補間で x 座標を求める
15            if u_vals[i+1] != u_vals[i]:
16                t = (0.5 - u_vals[i]) / (u_vals[i+1] - u_vals[i])
17                x_half = x_vals[i] + t * (x_vals[i+1] - x_vals[i])
18                return x_half
19            else:
20                return x_vals[i]
21    return None
22
23 def u0_fisher(x: float, b: float) -> float:
24     """
25         Fisher 方程式の初期条件 (8.67):
26         u0(x) = 1 / (1 + e^(bx-5))^2
27     """
28     return 1.0 / (1.0 + math.exp(b * x - 5.0)) ** 2
29
30 def init_u_fisher(N: int, dx: float, b: float) -> List[float]:
31     """
32         初期条件を離散化してu^0を作る。
33         配列は ghost を含めて長さ N+2 (index 0..N+1)。
34         セル中心の座標 x=(j-1/2)dx で初期条件を評価する。
35     """
36     u = [0.0] * (N + 2)
37     for j in range(1, N + 1):
38         x = (j - 0.5) * dx
39         u[j] = u0_fisher(x, b)
40     return u
41
42 def apply_bc_fisher(u: List[float], N: int) -> None:
43     """

```

```

44     境界条件 (8.68)(8.69):
45     u(0,t) = 1, u(L,t) = 0
46     ghost cell で表現:
47     u[0] = 1, u[N+1] = 0
48     """
49     u[0] = 1.0
50     u[N + 1] = 0.0
51
52 def f_fisher(u_val: float) -> float:
53     """
54     反応項 f(u) = u(1-u)
55     """
56     return u_val * (1.0 - u_val)
57
58 def step_explicit_euler_fisher(u: List[float], c: float, dt: float, N: int) -> List[
59     float]:
60     """
61     Fisher方程式に対するオイラー陽解法 (8.70):
62     (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
63
64     これを整理すると:
65     u^{n+1}_j = u^n_j + Δt*f(u^n_j) + c*(u^n_{j-1} - 2u^n_j + u^n_{j+1})
66
67     ここで c = Δt/(Δx^2)。
68
69     un = u[:]
70     up = [0.0] * (N + 2)
71
72     for j in range(1, N + 1):
73         f_val = f_fisher(un[j])
74         diffusion = c * (un[j-1] - 2.0*un[j] + un[j+1])
75         up[j] = un[j] + dt * f_val + diffusion
76
77     apply_bc_fisher(up, N)
78     return up
79
80 def run_fisher_case(b: float, L: float, dx: float, dt: float, output_times: List[int]) -> dict:
81     """
82     Fisher方程式を1ケース実行する。
83
84     Parameters:
85     - b: 初期条件のパラメータ
86     - L: 空間領域の長さ
87     - dx: 空間刻み幅
88     - dt: 時間刻み幅
89     - output_times: 出力するステップ数のリスト
90
91     Returns:
92     - dict: {時刻: (x座標リスト, u値リスト)}
93     """
94     N = int(L / dx)
95     c = dt / (dx * dx)
96
97     # 初期条件
98     u = init_u_fisher(N, dx, b)
99     apply_bc_fisher(u, N)

```

```

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
156     for t in times:

```

```

157     plt.figure(figsize=(10, 6))
158     for b in b_values:
159         x_vals, u_vals = all_results[b][t]
160         plt.plot(x_vals, u_vals, color=colors[str(b)], label=f'b = {b}', linewidth
161                 =1.5)
162
163         plt.xlabel('x', fontsize=14)
164         plt.ylabel('u', fontsize=14)
165         plt.title(f'Fisher Equation (t = {t})', fontsize=16)
166         plt.legend(fontsize=12)
167         plt.grid(True, alpha=0.3)
168         plt.xlim(0, 200)
169         plt.ylim(-0.1, 1.1)
170         plt.xticks(fontsize=12)
171         plt.yticks(fontsize=12)
172
173         plt.savefig(output_dir / f"fisher_t{t}.png", dpi=150, bbox_inches='tight')
174         plt.close()
175         print(f"Saved graph for t = {t}")
176
177 # 2. 各bについて、異なる時刻を重ねて描画
178 for b in b_values:
179     plt.figure(figsize=(10, 6))
180     time_colors = {0: 'black', 10: 'blue', 20: 'green', 30: 'orange', 40: 'red'}
181
182     for idx, t in enumerate(times):
183         x_vals, u_vals = all_results[b][t]
184         plt.plot(x_vals, u_vals, color=time_colors[t], label=f't = {t}', linewidth
185                 =1.5)
186
187         # u = 0.5 となる点を探してプロット
188         x_half = find_u_half_point(x_vals, u_vals)
189         if x_half is not None:
190             plt.plot(x_half, 0.5, 'o', color='black', markersize=8)
191             # 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)
192             if idx % 2 == 0:
193                 y_offset = 0.15
194             else:
195                 y_offset = -0.15
196             plt.annotate(f'({x_half:.2f}, 0.5)',
197                         xy=(x_half, 0.5),
198                         xytext=(x_half + 5, 0.5 + y_offset),
199                         fontsize=12,
200                         color='black',
201                         arrowprops=dict(arrowstyle='->', color='black', lw=0.5))
202
203         plt.xlabel('x', fontsize=14)
204         plt.ylabel('u', fontsize=14)
205         plt.title(f'Fisher Equation (b = {b})', fontsize=16)
206         plt.legend(fontsize=12)
207         plt.grid(True, alpha=0.3)
208         plt.xlim(0, 200)
209         plt.ylim(-0.1, 1.1)
210         plt.xticks(fontsize=12)
211         plt.yticks(fontsize=12)
212
213         plt.savefig(output_dir / f"fisher_b{str(b).replace('.', '_')}.png", dpi=150,
214                     bbox_inches='tight')

```

```

212     plt.close()
213     print(f"Saved graph for b = {b}")
214
215 if __name__ == "__main__":
216     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

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # パラメータ設定
5 hbar = 1.0
6 m = 1.0
7 k = 1.0
8
9 N = 400
10 dx = 0.05
11 L = N * dx # L = 20
12 dt = 0.001
13
14 # 空間格子点
15 x = np.zeros(N + 2) # j = 0, 1, ..., N, N+1 (境界条件用)
16 for j in range(1, N + 1):
17     x[j] = (j - 0.5) * dx - L / 2 # (8.78)
18
19 # 初期条件
20 R = np.zeros(N + 2)
21 I = np.zeros(N + 2)
22
23 # R_j^0 の設定 (8.83)
24 for j in range(1, N + 1):
25     R[j] = (np.sqrt(2) / np.pi**0.25) * np.exp(-2 * (x[j] - 5)**2)
26
27 # 周期境界条件の適用 (8.79), (8.80)
28 R[0] = R[N]
29 R[N + 1] = R[1]
30
31 # I_j^0 の設定 (8.84)
32 for j in range(1, N + 1):
33     laplacian_R = (R[j - 1] - 2 * R[j] + R[j + 1]) / dx**2
34     I[j] = -dt * (-0.5 * laplacian_R + 0.5 * x[j]**2 * R[j])
35
36 # 周期境界条件の適用 (8.81), (8.82)
37 I[0] = I[N]
38 I[N + 1] = I[1]
39
40 # 結果を保存するリスト
41 results = []
42 output_times_original = [1, 2, 3, 4, 5, 6, 7, 8]
43 output_steps_original = [int(t / dt) for t in output_times_original]
44
45 # 追加: π倍数での出力時刻
46 output_times_pi = [0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi, 5*np.pi/2]

```

```

47 output_steps_pi = [int(t / dt) for t in output_times_pi]
48 results_pi = {}
49
50 # P の最大値を保存するリスト
51 P_max_history = []
52 t_history = []
53
54 # 時間発展
55 n_max = max(max(output_steps_original), max(output_steps_pi))
56
57 # t=0 の保存
58 P0 = R[1:N+1]**2 + I[1:N+1]**2
59 results_pi[0] = P0.copy()
60 P_max_history.append(np.max(P0))
61 t_history.append(0)
62 print(f"t = 0.0000 (n = 0): 確率密度の総和 = {np.sum(P0) * dx:.6f}")
63
64 for n in range(n_max):
65     # R^{n+1} の計算 (8.76)
66     R_new = np.zeros(N + 2)
67     for j in range(1, N + 1):
68         laplacian_I = (I[j - 1] - 2 * I[j] + I[j + 1]) / dx**2
69         R_new[j] = R[j] + dt * (-0.5 * laplacian_I + 0.5 * x[j]**2 * I[j])
70
71     # 周期境界条件 (8.79), (8.80)
72     R_new[0] = R_new[N]
73     R_new[N + 1] = R_new[1]
74
75     # I^{n+1} の計算 (8.77) - R^{n+1} を使用
76     I_new = np.zeros(N + 2)
77     for j in range(1, N + 1):
78         laplacian_R_new = (R_new[j - 1] - 2 * R_new[j] + R_new[j + 1]) / dx**2
79         I_new[j] = I[j] - dt * (-0.5 * laplacian_R_new + 0.5 * x[j]**2 * R_new[j])
80
81     # 周期境界条件 (8.81), (8.82)
82     I_new[0] = I_new[N]
83     I_new[N + 1] = I_new[1]
84
85     # 更新
86     R = R_new
87     I = I_new
88
89     # 各ステップでPの最大値を保存
90     step = n + 1
91     P_current = R[1:N+1]**2 + I[1:N+1]**2
92     P_max_history.append(np.max(P_current))
93     t_history.append(step * dt)
94
95     # 出力時刻で保存
96     if step in output_steps_original:
97         t = step * dt
98         P = R[1:N+1]**2 + I[1:N+1]**2
99         results[t] = P.copy()
100        print(f"t = {t:.1f} (n = {step}): 確率密度の総和 = {np.sum(P) * dx:.6f}")
101
102    if step in output_steps_pi:
103        t_pi = output_times_pi[output_steps_pi.index(step)]
104        P = R[1:N+1]**2 + I[1:N+1]**2

```

```

105     results_pi[t_pi] = P.copy()
106     print(f't = {step * dt:.4f} (n = {step}): 確率密度の総和 = {np.sum(P) * dx:.6f}
107         }')
108
109 # 確率密度のプロット (t=1,2,3,4,5,6,7,8)
110 fig, axes = plt.subplots(2, 4, figsize=(14, 7))
111 axes = axes.flatten()
112
113 x_plot = x[1:N+1]
114
115 for idx, t in enumerate(output_times_original):
116     ax = axes[idx]
117     P = results[t]
118     ax.plot(x_plot, P, 'b-', linewidth=1)
119     ax.set_xlabel('$x$')
120     ax.set_ylabel('$P(x,t)$')
121     ax.set_title(f'$t = {t}$')
122     ax.set_xlim(-L/2, L/2)
123     ax.set_ylim(0, 1.2)
124     ax.grid(True, alpha=0.3)
125
126 plt.tight_layout()
127 plt.savefig('graphs_schrodinger/probability_density.png', dpi=150)
128 plt.show()
129
130 # 確率密度のプロット (t=0, π/2, π, 3π/2, 2π, 5π/2)
131 fig, axes = plt.subplots(2, 3, figsize=(12, 7))
132 axes = axes.flatten()
133
134 pi_labels = ['0', r'\pi/2', r'\pi', r'3\pi/2', r'2\pi', r'5\pi/2']
135
136 for idx, (t, label) in enumerate(zip(output_times_pi, pi_labels)):
137     ax = axes[idx]
138     P = results_pi[t]
139     ax.plot(x_plot, P, 'b-', linewidth=1)
140     ax.set_xlabel('$x$')
141     ax.set_ylabel('$P(x,t)$')
142     ax.set_title(f'$t = {label}$')
143     ax.set_xlim(-L/2, L/2)
144     ax.set_ylim(0, 1.2)
145     ax.grid(True, alpha=0.3)
146
147 plt.tight_layout()
148 plt.savefig('graphs_schrodinger/probability_density_pi.png', dpi=150)
149 plt.show()
150
151 # P の最大値の時間発展プロット
152 plt.figure(figsize=(10, 6))
153 plt.plot(t_history, P_max_history, 'b-', linewidth=0.5)
154 plt.xlabel('$t$')
155 plt.ylabel('$\max_x P(x,t)$')
156 plt.title('Time Evolution of Maximum Probability Density')
157 plt.grid(True, alpha=0.3)
158
159 # π倍数に縦線を追加
160 for i in range(1, 4):
161     plt.axvline(x=i*np.pi, color='red', linestyle='--', linewidth=1, alpha=0.7)

```

```

162 plt.savefig('graphs_schrodinger/P_max_evolution.png', dpi=150)
163 plt.show()
164
165 # Pの最大値の最大と最小を出力
166 P_max_arr = np.array(P_max_history)
167 print(f"\n==== Pの最大値の統計 ===")
168 print(f"max(P_max) = {np.max(P_max_arr):.6f} (t = {t_history[np.argmax(P_max_arr)]:.4f}")
169 print(f"min(P_max) = {np.min(P_max_arr):.6f} (t = {t_history[np.argmin(P_max_arr)]:.4f")
170
171 # Pの最大値の逆数の時間発展プロット
172 P_max_inv = 1.0 / P_max_arr
173 plt.figure(figsize=(10, 6))
174 plt.plot(t_history, P_max_inv, 'b-', linewidth=0.5)
175 plt.xlabel('$t$')
176 plt.ylabel('$1 / \max_x P(x,t)$')
177 plt.title('Time Evolution of Inverse Maximum Probability Density')
178 plt.grid(True, alpha=0.3)
179
180 # π倍数に縦線を追加
181 for i in range(1, 4):
182     plt.axvline(x=i*np.pi, color='red', linestyle='--', linewidth=1, alpha=0.7)
183
184 plt.savefig('graphs_schrodinger/P_max_inv_evolution.png', dpi=150)
185 plt.show()
186
187 # 時空間プロット（確率密度の等高線）
188 # 全時刻のデータを保存するために再計算
189 print("\n時空間プロットを作成中...")
190
191 # 初期条件の再設定
192 R = np.zeros(N + 2)
193 I = np.zeros(N + 2)
194
195 for j in range(1, N + 1):
196     R[j] = (np.sqrt(2) / np.pi**0.25) * np.exp(-2 * (x[j] - 5)**2)
197
198 R[0] = R[N]
199 R[N + 1] = R[1]
200
201 for j in range(1, N + 1):
202     laplacian_R = (R[j - 1] - 2 * R[j] + R[j + 1]) / dx**2
203     I[j] = -dt * (-0.5 * laplacian_R + 0.5 * x[j]**2 * R[j])
204
205 I[0] = I[N]
206 I[N + 1] = I[1]
207
208 # 時空間データの保存
209 save_interval = 100 # 100ステップごとに保存
210 t_values = [0]
211 P_history = [R[1:N+1]**2 + I[1:N+1]**2]
212
213 for n in range(n_max):
214     # R^{n+1} の計算
215     R_new = np.zeros(N + 2)
216     for j in range(1, N + 1):
217         laplacian_I = (I[j - 1] - 2 * I[j] + I[j + 1]) / dx**2

```

```

218     R_new[j] = R[j] + dt * (-0.5 * laplacian_I + 0.5 * x[j]**2 * I[j])
219
220     R_new[0] = R_new[N]
221     R_new[N + 1] = R_new[1]
222
223     #  $I^{n+1}$  の計算
224     I_new = np.zeros(N + 2)
225     for j in range(1, N + 1):
226         laplacian_R_new = (R_new[j - 1] - 2 * R_new[j] + R_new[j + 1]) / dx**2
227         I_new[j] = I[j] - dt * (-0.5 * laplacian_R_new + 0.5 * x[j]**2 * R_new[j])
228
229     I_new[0] = I_new[N]
230     I_new[N + 1] = I_new[1]
231
232     R = R_new
233     I = I_new
234
235     step = n + 1
236     if step % save_interval == 0:
237         t_values.append(step * dt)
238         P_history.append(R[1:N+1]**2 + I[1:N+1]**2)
239
240 # 時空間データの準備
241 t_arr = np.array(t_values)
242 P_arr = np.array(P_history)
243 X, T = np.meshgrid(x_plot, t_arr)
244
245 # 等高線プロット
246 plt.figure(figsize=(10, 6))
247 levels = np.linspace(0, np.max(P_arr), 20)
248 plt.contour(X, T, P_arr, levels=levels, colors='black', linewidths=0.5)
249 plt.contourf(X, T, P_arr, levels=levels, cmap='hot')
250 plt.colorbar(label='$P(x,t) = |\psi(x,t)|^2$')
251
252 #  $t=\pi$ ,  $2\pi$  に点線を追加
253 plt.axhline(y=np.pi, color='blue', linestyle='--', linewidth=1.5, label='$t=\pi$')
254 plt.axhline(y=2*np.pi, color='cyan', linestyle='--', linewidth=1.5, label='$t=2\pi$')
255 plt.legend(loc='upper right')
256
257 plt.xlabel('$x$')
258 plt.ylabel('$t$')
259 plt.title('Contour Plot of Probability Density $P(x,t)$')
260 plt.savefig('graphs_schrodinger/probability_density_contour.png', dpi=150)
261 plt.show()
262
263 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）.