# 数理工学実験レポート

第 6 章（連続最適化）

学籍番号 1029366161　中塚一瑳

2026 年 1 月 4 日

## 目次

# はじめに

今回は連続最適化の様々な手法を用いて、与えられた関数の最小値を求める課題に取り組む。

# 1　課題 15：ニュートン法および二分法による多項式の零点計算

本課題では，多項式

$$f(x) = x^3 + 2x^2 - 5x - 6 \tag{1}$$

の零点を数値的に求める．まず関数のグラフを描画して零点の存在を確認し，その後，二分法およびニュートン法を用いて零点を計算する．

## 1.1　原理と方法

### 1.1.1　二分法

二分法は，区間 $[a, b]$ において $f(a)$ と $f(b)$ の符号が異なるとき，その区間内に零点が存在することを利用した反復法である [1]．中点 $c = (a + b)/2$ を取り，$f(c)$ の符号に応じて零点を含む半区間に更新する操作を繰り返すことで，区間幅を徐々に縮小し零点へ収束させる．零点を挟む区間が与えられれば必ず収束するが，収束速度は比較的遅い．

### 1.1.2　ニュートン法

ニュートン法は，関数をある点 $x_k$ の周りで一次近似し，その接線と $x$ 軸の交点を次の近似値とする方法である [1]．反復公式は

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{2}$$

で与えられる．一般に収束は速いが，初期値の選び方によっては収束しない場合がある．本課題で用いた導関数は

$$f'(x) = 3x^2 + 4x - 5 \tag{3}$$

である．

## 1.2　実験方法

Python を用いて $x \in [-10, 10]$ の範囲で $f(x)$ を描画し，グラフから零点が $x \approx -3, -1, 2$ 付近に存在することを確認した．二分法では，それぞれの零点を挟む区間として

$$[-4, -2],\ [-2, 0],\ [1, 3]$$

を与えた．ニュートン法では初期値として

$$x_0 = -2.5,\ -0.5,\ 1.5$$

を用いた．停止条件は $|f(x)| \leq 10^{-10}$ とした．

## 1.3 結果

### 1.3.1 関数のグラフ

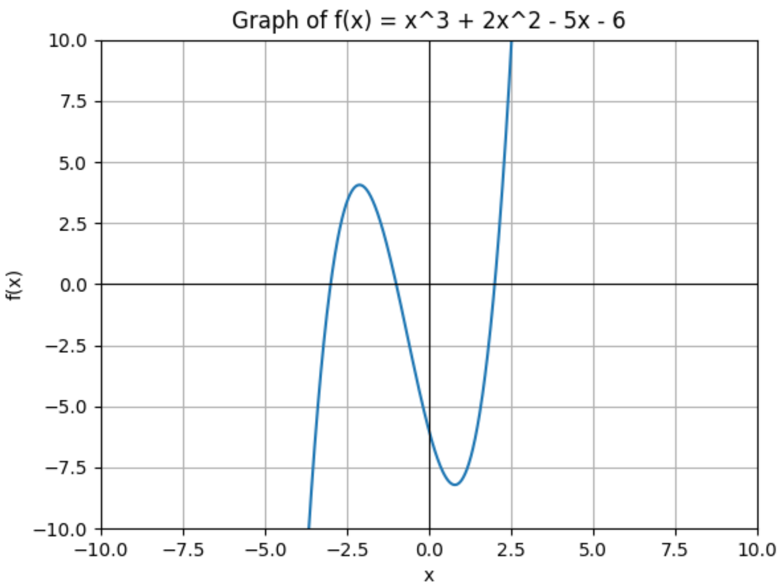


**図 1:** $f(x) = x^3 + 2x^2 - 5x - 6$ のグラフ

### 1.3.2 零点

数値計算によって得られた零点を表 1 に示す．

**表 1:** 二分法およびニュートン法で求めた零点

| 手法 | 近似解 $x$ | 残差 $f(x)$ |
|---|---|---|
| 二分法 | $-3.000$ | 0 |
| 二分法 | $-1.000$ | 0 |
| 二分法 | 2.000 | 0 |
| ニュートン法 | $-3.000$ | 0 |
| ニュートン法 | $-1.000$ | 0 |
| ニュートン法 | 2.000 | $1.421 \times 10^{-14}$ |

## 1.4 考察

$f(x) = 0$ は因数分解により

$$f(x) = (x+3)(x+1)(x-2)$$

と書け，解析解は $x = -3, -1, 2$ である．数値計算によって得られた零点はこれらと一致しており，手法が正しく実装されていることが確認できる．ニュートン法において残差が完全に 0 とならない場合があるのは，浮動小数点演算誤差によるものである．

# 2　課題 16：最急降下法およびニュートン法による停留点計算

## 2.1　原理と方法

本課題では，次の関数

$$f(x) := \frac{1}{3}x^3 - x^2 - 3x + \frac{5}{3} \tag{4}$$

の停留点（$f'(x) = 0$ を満たす点）を，(a) 最急降下法，(b) ニュートン法により数値的に求める．まず微分は

$$f'(x) = x^2 - 2x - 3 = (x-3)(x+1), \tag{5}$$

$$f''(x) = 2x - 2 \tag{6}$$

である．従って停留点は解析的には $x = -1, 3$ の 2 点である．

### 2.1.1　(a) 最急降下法

1 次元では勾配 $\nabla f(x)$ は $f'(x)$ に一致するため，最急降下法は

$$x_{k+1} = x_k - t_k f'(x_k), \qquad t_k = \frac{1}{k+1} \tag{7}$$

で与えられる [1]．初期点は $x_0 = 1/2$ とする．停止判定は $|f'(x_k)| \leq \varepsilon$（$\varepsilon = 10^{-8}$）とし，最大反復回数も設ける．

### 2.1.2　(b) ニュートン法

停留点探索（$f'(x) = 0$ の零点探索）としてのニュートン法は

$$x_{k+1} = x_k - t_k \frac{f'(x_k)}{f''(x_k)}, \qquad t_k = 1 \tag{8}$$

で与えられる [1]．初期点は $x_0 = 5$ とし，(a) と同様に $|f'(x_k)| \leq \varepsilon$ を停止判定とした．なお，本問題では $f''(x) = 0$（すなわち $x = 1$）で更新が不能となるため，実装では $f''(x_k) = 0$ の場合を例外として扱う．

## 2.2　実装

Python により $f, f', f''$ をそれぞれ関数として実装し，(a) と (b) の更新式をそのまま反復した．各反復で $(x_k, f(x_k), |f'(x_k)|)$ を履歴として保存し，収束挙動の可視化に用いた．

## 2.3　結果

(a) 最急降下法 ($x_0 = 0.5,\ t_k = 1/(k+1)$) および (b) ニュートン法 ($x_0 = 5,\ t_k = 1$) の結果を表 2 に示す．両手法とも停留点 $x^* = 3$ に収束した（もう一つの停留点 $x = -1$ には到達しなかった）．

両方法による反復点の推移を，関数 $y = f(x)$ 上に重ねて図 2 に示す．

表 2: 課題 16 の計算結果（$\varepsilon = 10^{-8}$）

| 手法 | 初期値 $x_0$ | 近似解 $x^*$ | $f(x^*)$ | 反復回数 $k$ |
|---|---|---|---|---|
| 最急降下法 | 0.5000 | 3.000 | -7.333 | 193 |
| ニュートン法 | 5.000 | 3.000 | -7.333 | 5 |

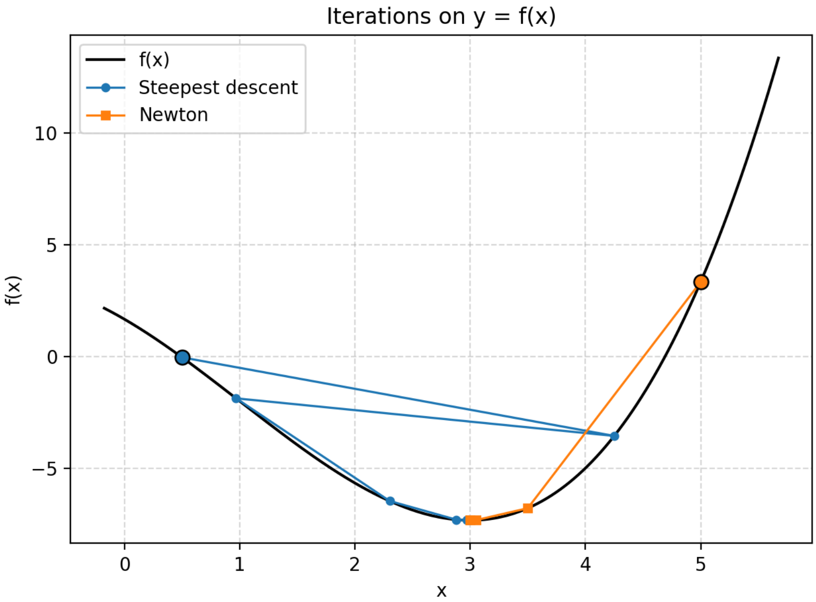


**図 2:** 関数 $y = f(x)$ 上における反復点の推移．青丸は最急降下法，橙四角はニュートン法による反復点を表す．

## 2.4 考察

図 2 より，最急降下法では反復点が谷に沿って緩やかに移動しており，ステップサイズ $t_k = 1/(k+1)$ が減少することで後半の収束が遅くなっていることが視覚的に確認できる．一方，ニュートン法では局所的な 2 次近似に基づき更新が行われるため，停留点近傍では大きなジャンプを伴い，極めて高速に収束している．

# 3 課題 17：勾配およびヘッセ行列の解析的導出と実装との対応

本課題では，次の関数

$$f(x_0, x_1) = x_0^2 + e^{x_0} + x_1^4 + x_1^2 - 2x_0x_1 + 3 \tag{9}$$

について，勾配およびヘッセ行列を解析的に求め，それを実装したコードとの対応を示す．

### 3.1　微分結果

まず，各変数による 1 階微分は

$$\frac{\partial f}{\partial x_0} = 2x_0 + e^{x_0} - 2x_1, \tag{10}$$

$$\frac{\partial f}{\partial x_1} = 4x_1^3 + 2x_1 - 2x_0 \tag{11}$$

である．従って勾配ベクトルは

$$\nabla f(x) = \begin{pmatrix} 2x_0 + e^{x_0} - 2x_1 \\ 4x_1^3 + 2x_1 - 2x_0 \end{pmatrix} \tag{12}$$

と表される [1].

次に，2 階微分よりヘッセ行列は

$$\nabla^2 f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_0^2} & \frac{\partial^2 f}{\partial x_0 \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_0} & \frac{\partial^2 f}{\partial x_1^2} \end{pmatrix} = \begin{pmatrix} 2 + e^{x_0} & -2 \\ -2 & 12x_1^2 + 2 \end{pmatrix} \tag{13}$$

となる．

### 3.2　コードとの対応

上記の解析結果は，以下の Python コードとして実装されている．

- 目的関数

```
f = x0**2 + exp(x0) + x1**4 + x1**2 - 2*x0*x1 + 3
```

- 勾配

```
g0 = 2*x0 + exp(x0) - 2*x1
g1 = 4*x1**3 + 2*x1 - 2*x0
g  = [g0, g1]
```

- ヘッセ行列

```
h00 = 2 + exp(x0)
h01 = -2
h10 = -2
h11 = 12*x1**2 + 2
H = [[h00, h01],
     [h10, h11]]
```

これにより，関数値・勾配・ヘッセ行列が理論式と一致して計算されていることが確認できる．

## 4 課題 18：最急降下法およびニュートン法による 2 変数関数の最小化

### 4.1 原理と方法

本課題では，2 変数関数

$$f(x_0, x_1) = x_0^2 + e^{x_0} + x_1^4 + x_1^2 - 2x_0x_1 + 3 \tag{14}$$

を最小化する．勾配およびヘッセ行列は

$$\nabla f(x) = \begin{pmatrix} 2x_0 + e^{x_0} - 2x_1 \\ 4x_1^3 + 2x_1 - 2x_0 \end{pmatrix}, \tag{15}$$

$$\nabla^2 f(x) = \begin{pmatrix} 2 + e^{x_0} & -2 \\ -2 & 12x_1^2 + 2 \end{pmatrix} \tag{16}$$

である．更新は

$$x_{k+1} = x_k + t_k d_k \tag{17}$$

とし，方向 $d_k$ は (a) 最急降下法 $d_k = -\nabla f(x_k)$，(b) ニュートン法 $\nabla^2 f(x_k) d_k = -\nabla f(x_k)$ の解として定める [1].

ステップ幅 $t_k$ はバックトラック法（Armijo 条件）で決定した [1]．すなわち，$t \leftarrow t_{\text{init}}$ から開始し，

$$f(x_k + td_k) \le f(x_k) + \xi t \langle \nabla f(x_k), d_k \rangle \tag{18}$$

を満たすまで $t \leftarrow \rho t$ により縮小する．

### 4.2 実験方法

初期点は $x_0 = (1, 1)^\mathsf{T}$ とし，共通パラメータを

$$\xi = 1.0 \times 10^{-4}, \qquad \rho = 0.5, \qquad t_{\text{init}} = 1 \tag{19}$$

とした. 停止条件は $\|\nabla f(x_k)\| \le 1.0 \times 10^{-6}$ とし, 最大反復回数は 1000 とした. 各反復で $(x_k, f(x_k))$ を保存し，$x_0$–$x_1$ 平面上の等高線図に反復点列を重ねて可視化した（図 3）.

### 4.3 結果

(a) バックトラック法付き最急降下法および (b) バックトラック法付きニュートン法の収束結果を表 3 に示す．両手法は同一の解に収束し，ニュートン法の方が少ない反復回数で収束した.

table3 より，ニュートン法は最急降下法に比べて訳 5 分の 1 の反復回数で収束している.

反復点列を等高線図上に重ねた結果を図 3 に示す.

表 3: 課題 18 の計算結果（初期点 $x_0 = (1, 1)^\mathsf{T}$）

| 手法 | $x_0^*$ | $x_1^*$ | $f(x^*)$ | 反復回数 $k$ |
|---|---|---|---|---|
| 最急降下法 +BT | $-0.7335$ | $-0.4933$ | 3.597 | 31 |
| ニュートン法 +BT | $-0.7335$ | $-0.4933$ | 3.597 | 6 |

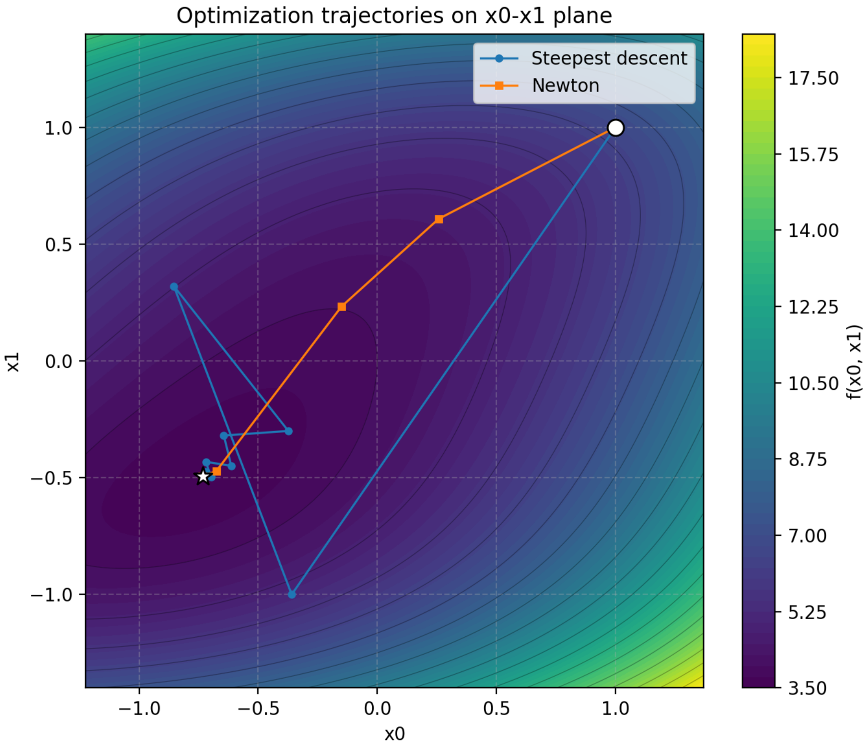


図 3: $x_0$–$x_1$ 平面における等高線図と反復点列．〇が最急降下法 +BT，□がニュートン法 +BT の反復点列を表す．

### 4.4 考察

図 3 より，最急降下法では反復点同士を結ぶ線分が等高線に対して直交しているのに対し，ニュートン法では直交していないものの、あたかも低い場所を知っているかのように、等高線の谷に沿って効率的に移動している様子がわかる．これは，ニュートン法がヘッセ行列を用いて関数の曲率を考慮しているためであり，その結果として少ない反復回数で収束していると考えられる．

## 5 課題 19：総合演習

### 5.1 原理と方法

本課題では，$x = (x_0, x_1)^\mathsf{T} \in \mathbb{R}^2$ に対し

$$f(x) = \sum_{i=0}^{2} f_i(x)^2, \qquad f_i(x) = y_i - x_0\left(1 - x_1^{i+1}\right) \tag{20}$$

を最小化する. 定数は $y_0 = 1.5,\ y_1 = 2.25,\ y_2 = 2.625$ とする（最適値は 0, 最適解は $x^* = (3, 0.5)^\top$）.

$f_i$ の勾配およびヘッセ行列は

$$\nabla f_i(x) = \begin{pmatrix} -1 + x_1^{i+1} \\ x_0(i+1)x_1^i \end{pmatrix}, \tag{21}$$

$$\nabla^2 f_0(x) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad \nabla^2 f_i(x) = \begin{pmatrix} 0 & (i+1)x_1^i \\ (i+1)x_1^i & x_0(i+1)i\,x_1^{i-1} \end{pmatrix}\ (i = 1, 2) \tag{22}$$

である． よって，勾配・ヘッセ行列は課題 18 と同様に

$$\nabla f(x) = 2\sum_{i=0}^{2} f_i(x)\nabla f_i(x), \tag{23}$$

$$\nabla^2 f(x) = 2\sum_{i=0}^{2} \left(f_i(x)\nabla^2 f_i(x) + \nabla f_i(x)\nabla f_i(x)^\top\right) \tag{24}$$

より計算できる [1]． 更新式 $x_{k+1} = x_k + t_k d_k$，方向の選び方（最急降下法／ニュートン法），および バックトラック法（Armijo 条件）によるステップ幅決定は課題 18 と同様である．

## 5.2 実験方法

初期点は $x_0 = (2, 0)^\top$ とし，共通パラメータを

$$\xi = 1.0 \times 10^{-4}, \qquad \rho = 0.5, \qquad t_{\mathrm{init}} = 1 \tag{25}$$

とした. 停止条件は $\|\nabla f(x_k)\| \le 1.0 \times 10^{-6}$ とし, 最大反復回数は 1000 とした. 各反復で $(x_k, f(x_k))$ を保存し，$x_0$–$x_1$ 平面上の等高線図に反復点列を重ねて可視化した（図 4）.

## 5.3 結果

(a) バックトラック法付き最急降下法および (b) バックトラック法付きニュートン法の収束結果を表 4 に示す． 両手法はほぼ同一の解に収束し，ニュートン法の方が少ない反復回数で収束した．

**表 4:** 課題 19 の計算結果（初期点 $x_0 = (2, 0)^\top$）

| 手法 | $x_0^*$ | $x_1^*$ | $f(x^*)$ | 反復回数 $k$ |
|---|---|---|---|---|
| 最急降下法 +BT | 2.99999781 | 0.49999946 | $7.68 \times 10^{-13}$ | 791 |
| ニュートン法 +BT | 3.00000013 | 0.50000005 | $9.47 \times 10^{-15}$ | 5 |

表 4 より，ニュートン法は最急降下法に比べて大幅に少ない反復回数で収束している． 反復点列を等高線図上に重ねた結果を図 4 に示す．

## 5.4 考察

図 4 より，最急降下法では最適解近傍でジグザグに進む傾向が見られ，収束までに多くの反復を要した． 一方，ニュートン法はヘッセ行列により曲率情報を用いるため，より適切な方向・スケールで更新でき，少ない反復回数で効率的に収束したと考えられる．

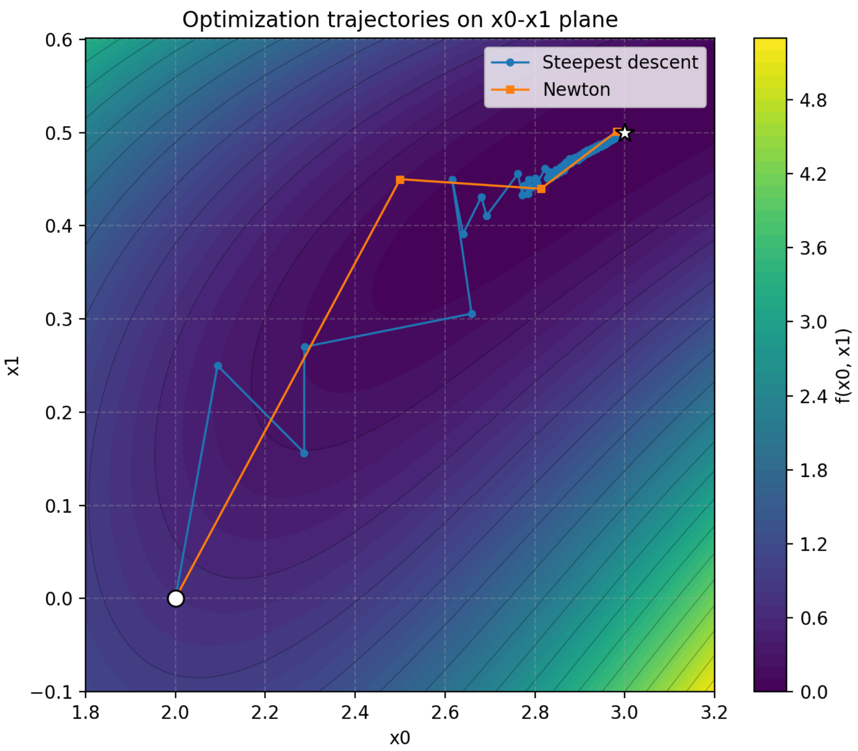


**図 4:** $x_0$–$x_1$ 平面における等高線図と反復点列．○が最急降下法 +BT，□がニュートン法 +BT の反復点列を表す．

# 6　追加課題：制約付き凸最適化問題

$$\min_{x\in\mathbb{R}^n} \ \frac{1}{2}x^\top Qx + c^\top x \quad \text{subject to} \quad Ax \le b,$$

ただし $Q$ は対称半正定値行列とする。

## 6.1　課題 1

二次計画問題として定式化することが自然な問題の一つに、最小二乗法による制約付き回帰問題がある。与えられたデータ点 $(a_i, y_i) \in \mathbb{R}^n \times \mathbb{R}$ $(i = 1, \ldots, m)$ に対し、

$$\min_{x\in\mathbb{R}^n} \ \frac{1}{2}\sum_{i=1}^{m}(y_i - a_i^\top x)^2 \quad \text{subject to} \quad Ax \le b$$

を解くことで、線形モデル $y \approx a^\top x$ のパラメータ $x$ を求めることができる。この問題は、行列 $A \in \mathbb{R}^{m\times n}$、ベクトル $c \in \mathbb{R}^m$ を用いて

$$\min_{x\in\mathbb{R}^n} \ \frac{1}{2}\|Ax - y\|^2 = \min_{x\in\mathbb{R}^n} \ \frac{1}{2}x^\top(A^\top A)x - y^\top Ax + \frac{1}{2}y^\top y$$

と変形でき、$Q = A^\top A$、$c = -A^\top y$ とすれば二次計画問題の形に帰着できる。

## 6.2 課題 2

課題 3 で後ほど説明するように、Q が半正定値である場合は KKT 条件と内点法を用いて最適解を求めることができる。しかし、Q が正定値でない、例えば Q の固有値に 1 つでも負の値がある場合は、KKT 条件を満たす点が最適解とは限らず、NP 困難な問題となることが知られている。[2][3]。

## 6.3 課題 3

本課題では，University of California, Irvine の Numerical Optimization のチャプター 16.6 "INTERIOR-POINT METHODS" を参考に，内点法を実装した。文献で扱われている $Ax \geq b$ の形式に合わせるため，

$$Ax \leq b \iff (-A)x \geq (-b)$$

と左右反転し，$\tilde{A} = -A,\ \tilde{b} = -b$ を用いて

$$\tilde{A}x \geq \tilde{b}$$

として以下を進める（記号の簡略化のため，再び $A, b$ と書く）。

制約 $Ax \geq b$ に対し，スラック変数 $y \geq 0$ を導入すると，

$$Ax - y - b = 0$$

と書ける。これに対する KKT 条件は

$$\begin{cases} Qx - A^\top \lambda + c = 0, \\ Ax - y - b = 0, \\ y_i \lambda_i = 0 \quad (i = 1, \ldots, m), \\ y \geq 0,\ \lambda \geq 0 \end{cases}$$

である。問題は凸であるため，これらの条件は必要十分である。

相補性条件 $y_i \lambda_i = 0$ は境界条件であり直接扱いにくいため，補完性の平均尺度

$$\mu = \frac{y^\top \lambda}{m}$$

を導入し，摂動 KKT 系

$$F(x, y, \lambda; \sigma\mu) = \begin{pmatrix} Qx - A^\top \lambda + c \\ Ax - y - b \\ Y\Lambda e - \sigma\mu e \end{pmatrix} = 0$$

を考える。ここで $Y = \mathrm{diag}(y_1, \ldots, y_m)$，$\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_m)$，$e = (1, \ldots, 1)^\top$，$\sigma \in (0, 1)$ は固定定数である。$\sigma\mu > 0$ の解は central path を成し，$\sigma\mu \to 0$ により元の KKT 解（最適解）に収束する。

現在の反復点 $z = (x, y, \lambda)$ において，Newton 法により

$$F(z + \Delta z) \approx F(z) + J_F(z)\Delta z = 0$$

を解く。ヤコビ行列は

$$J_F(z) = \begin{pmatrix} Q & 0 & -A^\top \\ A & -I & 0 \\ 0 & \Lambda & Y \end{pmatrix}$$

であり，Newton 方程式は

$$\begin{pmatrix} Q & 0 & -A^\top \\ A & -I & 0 \\ 0 & 0 & Y \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta\lambda \end{pmatrix} = \begin{pmatrix} -r_d \\ -r_p \\ -\Lambda Y e + \sigma\mu e \end{pmatrix}$$

となる。ここで

$$r_d = Qx - A^\top \lambda + c, \qquad r_p = Ax - y - b$$

はそれぞれ双対残差および主残差である。

得られた方向に対して

$$(x^+, y^+, \lambda^+) = (x, y, \lambda) + \alpha(\Delta x, \Delta y, \Delta\lambda)$$

と更新する。ただし $y^+ > 0,\ \lambda^+ > 0$ を保つように $\alpha \in (0, 1]$ を選ぶ。

以上を反復し，$\|r_p\|$, $\|r_d\|$, $\mu$ が十分小さくなった時点で停止する。凸性より，停止点 $x$ は元の制約 $Ax \le b$ を満たす二次計画問題の最適解である。

## 6.4 課題 4

本課題では，以下のデータを用いて内点法を実装し，制約付き二次計画問題を解いた。

$$Q = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}, \quad c = \begin{pmatrix} -1 \\ -1 \end{pmatrix}, \quad A = \begin{pmatrix} 1 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 0 \end{pmatrix}.$$

初期点として

$$x^{(0)} = (0, 0)^\top$$

を用い，スラック変数および双対変数は

$$y^{(0)} = 1, \qquad \lambda^{(0)} = 1$$

とした。これらはいずれも正であり，内点法の初期条件を満たしている。

この条件下で課題 3 のアルゴリズムを addtask.py に実装し，反復を行った。以下が出力結果である。

$$\mathbf{x}^* = \begin{pmatrix} -2.809 \times 10^{-13} \\ -5.618 \times 10^{-13} \end{pmatrix}$$

$$y = \begin{pmatrix} 8.427 \times 10^{-13} \end{pmatrix}$$

$$\lambda = \begin{pmatrix} 1.000 \end{pmatrix}$$

$$\text{info} = \{\text{iter} : 12,\ r_d : 3.140 \times 10^{-16},\ r_p : 4.039 \times 10^{-28},\ \mu : 8.427 \times 10^{-13}\}$$

つまり, 最適解は

$$x^* \approx \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$y \approx \begin{pmatrix} 0 \end{pmatrix}$$

$$\lambda \approx \begin{pmatrix} 1 \end{pmatrix}$$

であることがわかる。

## 結論

今回はニュートン法など、様々な連続最適化の様々な手法を用いて、与えられた関数の最小値を求める課題に取り組んだ。特に、曲率を考慮するニュートン法は、二分法や、最急降下法に比べて収束が速いことが確認できた。

# 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 15 のコード

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt

# -------------------------
# 課題15: 関数の定義
# f(x) = x^3 + 2x^2 - 5x - 6
# -------------------------

def f(x):
    return x**3 + 2*x**2 - 5*x - 6

def df(x):
    """f(x) の導関数: f'(x) = 3x^2 + 4x - 5"""
    return 3*x**2 + 4*x - 5


# -------------------------
# (a) グラフ描画
# -------------------------

def plot_function():
    x = np.linspace(-10, 10, 2000)
    y = f(x)

    plt.figure()
    plt.plot(x, y)
    plt.axhline(0, color="black", linewidth=0.8)  # x軸
    plt.axvline(0, color="black", linewidth=0.8)  # y軸
    plt.xlim(-10, 10)
    plt.ylim(-10, 10)
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.title("Graph of f(x) = x^3 + 2x^2 - 5x - 6")
    plt.grid(True)
    plt.savefig("task15.png")
    plt.show()

```

```python

# ------------------------
# (b) 二分法
# ------------------------

def bisection(f, a, b, eps=1e-10, max_iter=1000):
    """[a, b] で二分法により f(x) = 0 の解を求める.
       f(a) と f(b) の符号は異なることが前提。
    """
    fa = f(a)
    fb = f(b)
    if fa * fb > 0:
        raise ValueError("f(a) と f(b) の符号が同じです: a={}, b={}".format(a, b))

    for _ in range(max_iter):
        c = 0.5 * (a + b)
        fc = f(c)

        if abs(fc) <= eps or 0.5 * (b - a) < eps:
            return c

        # 符号でどちらの区間を残すか決める
        if fa * fc < 0:
            b = c
            fb = fc
        else:
            a = c
            fa = fc

    # 最大反復に達した場合
    return 0.5 * (a + b)


def solve_with_bisection():
    # グラフから零点が -3, -1, 2 付近にあることが分かるので
    # それを挟む区間を手で指定する
    intervals = [
        (-4.0, -2.0),  # -3 付近
        (-2.0,  0.0),  # -1 付近
        ( 1.0,  3.0),  #  2 付近
    ]

    roots = []
    for (a, b) in intervals:
        r = bisection(f, a, b)
        roots.append(r)
    return roots


# ------------------------
# (c) ニュートン法
# ------------------------

def newton(f, df, x0, eps=1e-10, max_iter=1000):
    """ニュートン法: f(x) = 0 の解を初期値 x0 から探索."""
    x = x0
    for _ in range(max_iter):
        fx = f(x)
```

```
        dfx = df(x)

        if abs(fx) <= eps:
            return x

        if dfx == 0:
            # 導関数が 0 になると更新できない
            raise ZeroDivisionError("f'(x) = 0 となったため打ち切り (x={})".format(x))

        x = x - fx / dfx

    return x  # 収束しなかった場合は最後の値を返す


def solve_with_newton():
    # グラフから零点が -3, -1, 2 付近にあることを利用
    initial_points = [-2.5, -0.5, 1.5]
    roots = []
    for x0 in initial_points:
        r = newton(f, df, x0)
        roots.append(r)
    return roots


# -------------------------
# メイン
# -------------------------

def main():
    # (a) グラフ描画
    plot_function()

    # (b) 二分法
    bisection_roots = solve_with_bisection()
    print("Bisection method roots:")
    for r in bisection_roots:
        print("  x ≈ {:.10f}, f(x) ≈ {:.3e}".format(r, f(r)))

    # (c) ニュートン法
    newton_roots = solve_with_newton()
    print("\nNewton method roots:")
    for r in newton_roots:
        print("  x0 -> root ≈ {:.10f}, f(x) ≈ {:.3e}".format(r, f(r)))


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

## A.2 課題 16 のコード

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# 課題16
# f(x) = (1/3)x^3 - x^2 - 3x + 5/3 の停留点を求める
# (a) 最急降下法 (x0 = 1/2, tk = 1/(k+1))
```

```python
# (b) ニュートン法 (x0 = 5, tk = 1)

from __future__ import annotations

from dataclasses import dataclass
from pathlib import Path


def f(x):
    """目的関数 f(x) = 1/3 x^3 - x^2 - 3x + 5/3"""
    return (1.0 / 3.0) * x**3 - x**2 - 3.0 * x + 5.0 / 3.0


def df(x):
    """1階微分 f'(x) = x^2 - 2x - 3"""
    return x**2 - 2.0 * x - 3.0


def d2f(x):
    """2階微分 f''(x) = 2x - 2"""
    return 2.0 * x - 2.0


@dataclass(frozen=True)
class IterationHistory:
    x: list[float]
    f: list[float]
    grad_abs: list[float]


# -------------------------
# (a) 最急降下法
# -------------------------


def steepest_descent(x0, eps=1e-8, max_iter=1000, *, return_history=False):
    """
    最急降下法(1次元版)
    x_{k+1} = x_k - t_k * f'(x_k)
    t_k = 1 / (k+1)
    """
    x = x0
    history_x = [float(x)]
    history_f = [float(f(x))]
    history_grad_abs = [float(abs(df(x)))]

    for k in range(max_iter):
        g = df(x)  # 勾配(1次元なので単なる導関数)
        if abs(g) <= eps:
            # 停留点に到達したとみなす
            if return_history:
                return x, k, IterationHistory(history_x, history_f, history_grad_abs)
            return x, k

        t_k = 1.0 / (k + 1)  # 指定どおりのステップ幅
        x = x - t_k * g

        history_x.append(float(x))
```

```python
        history_f.append(float(f(x)))
        history_grad_abs.append(float(abs(df(x))))

    # 最大反復に到達した場合
    if return_history:
        return x, max_iter, IterationHistory(history_x, history_f, history_grad_abs)
    return x, max_iter


# ------------------------
# (b) ニュートン法
# ------------------------


def newton_method(x0, eps=1e-8, max_iter=1000, *, return_history=False):
    """
    ニュートン法
    x_{k+1} = x_k - t_k * f'(x_k) / f''(x_k)
    ここでは t_k = 1
    """
    x = x0
    history_x = [float(x)]
    history_f = [float(f(x))]
    history_grad_abs = [float(abs(df(x)))]

    for k in range(max_iter):
        g = df(x)
        h = d2f(x)

        if abs(g) <= eps:
            # 停留点に到達したとみなす
            if return_history:
                return x, k, IterationHistory(history_x, history_f, history_grad_abs)
            return x, k

        if h == 0.0:
            raise ZeroDivisionError(f"f''(x) = 0 となったため更新できません (x = {x})")

        t_k = 1.0  # 指定どおり常に1
        x = x - t_k * g / h

        history_x.append(float(x))
        history_f.append(float(f(x)))
        history_grad_abs.append(float(abs(df(x))))

    if return_history:
        return x, max_iter, IterationHistory(history_x, history_f, history_grad_abs)
    return x, max_iter


# ------------------------
# メイン
# ------------------------


def main():
    print("=== (a) 最急降下法 ===")
```

```
    x0_sd = 0.5  # 初期点 x0 = 1/2
    x_star_sd, it_sd, hist_sd = steepest_descent(x0_sd, return_history=True)
    print("初期値 x0 = {:.6f}".format(x0_sd))
    print("近似停留点 x* ? {:.10f}".format(x_star_sd))
    print("f'(x*) ? {:.3e}".format(df(x_star_sd)))
    print("反復回数 k =", it_sd)
    print()

    print("=== (b) ニュートン法 ===")
    x0_nt = 5.0  # 初期点 x0 = 5
    x_star_nt, it_nt, hist_nt = newton_method(x0_nt, return_history=True)
    print("初期値 x0 = {:.6f}".format(x0_nt))
    print("近似停留点 x* ? {:.10f}".format(x_star_nt))
    print("f'(x*) ? {:.3e}".format(df(x_star_nt)))
    print("反復回数 k =", it_nt)
    print()

    # y=f(x) 上で、反復点が収束していく様子を同一グラフにプロット
    try:
        import matplotlib.pyplot as plt
    except ImportError:
        print("matplotlib が見つからないため、プロットをスキップします。")
        return

    all_x = hist_sd.x + hist_nt.x
    x_min = min(all_x)
    x_max = max(all_x)
    margin = 0.15 * (x_max - x_min) if x_max > x_min else 1.0
    x_left = x_min - margin
    x_right = x_max + margin

    n = 600
    xs = [x_left + (x_right - x_left) * i / n for i in range(n + 1)]
    ys = [f(x) for x in xs]

    plt.figure()
    plt.plot(xs, ys, color="black", linewidth=1.5, label="f(x)")

    plt.plot(hist_sd.x, hist_sd.f, marker="o", markersize=4, linewidth=1.2, label="Steepest descent")
    plt.plot(hist_nt.x, hist_nt.f, marker="s", markersize=4, linewidth=1.2, label="Newton")

    plt.scatter([hist_sd.x[0]], [hist_sd.f[0]], s=60, edgecolors="black", zorder=3)
    plt.scatter([hist_nt.x[0]], [hist_nt.f[0]], s=60, edgecolors="black", zorder=3)

    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.title("Iterations on y = f(x)")
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.legend()

    out_path = Path(__file__).with_name("task16.png")
    plt.tight_layout()
    plt.savefig(out_path, dpi=200)
    print(f"プロットを保存しました: {out_path}")
    import matplotlib

```

```
    if "agg" not in matplotlib.get_backend().lower():
        plt.show()


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

## A.3 課題 17 のコード

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy as np

# f(x) = x0^2 + exp(x0) + x1^4 + x1^2 - 2 x0 x1 + 3

def evalf(x):
    """
    目的関数の値 f(x) を計算する
    x: np.array shape (2,)
    """
    x0, x1 = x[0], x[1]
    f = x0**2 + np.exp(x0) + x1**4 + x1**2 - 2.0*x0*x1 + 3.0
    return f

def evalg(x):
    """
    勾配ベクトル ∇f(x) を計算する
    ∇f(x) = [ 2x0 + exp(x0) - 2x1,
              4x1^3 + 2x1 - 2x0 ]
    """
    x0, x1 = x[0], x[1]
    g0 = 2.0*x0 + np.exp(x0) - 2.0*x1
    g1 = 4.0*x1**3 + 2.0*x1 - 2.0*x0
    g = np.array([g0, g1])
    return g

def evalh(x):
    """
    ヘッセ行列 ∇^2 f(x) を計算する
    ∇^2 f(x) =
      [[ 2 + exp(x0),      -2          ],
       [      -2       , 12 x1^2 + 2   ]]
    """
    x0, x1 = x[0], x[1]
    h00 = 2.0 + np.exp(x0)
    h01 = -2.0
    h10 = -2.0
    h11 = 12.0*x1**2 + 2.0
    H = np.array([[h00, h01],
                  [h10, h11]])
    return H

def main():
    # 動作確認用
    x = np.array([0.3, 5.0])
```

```python
    f = evalf(x)
    g = evalg(x)
    H = evalh(x)

    print("x =", x)
    print("f(x) =", f)
    print("g(x) =", g)
    print("H(x) =\n", H)

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

## A.4 課題 18 のコード

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import annotations

from dataclasses import dataclass
from pathlib import Path

import numpy as np

# ==========================================================
# 課題17の設定
# f(x) = x0^2 + exp(x0) + x1^4 + x1^2 - 2 x0 x1 + 3
# ==========================================================


def evalf(x):
    """
    目的関数 f(x) を計算する
    x: np.array shape (2,)
    """
    x0, x1 = x[0], x[1]
    f = x0**2 + np.exp(x0) + x1**4 + x1**2 - 2.0 * x0 * x1 + 3.0
    return f


def evalg(x):
    """
    勾配ベクトル ∇f(x) を計算する
    ∇f(x) = [ 2x0 + exp(x0) - 2x1,
              4x1^3 + 2x1 - 2x0 ]
    """
    x0, x1 = x[0], x[1]
    g0 = 2.0 * x0 + np.exp(x0) - 2.0 * x1
    g1 = 4.0 * x1**3 + 2.0 * x1 - 2.0 * x0
    return np.array([g0, g1])


def evalh(x):
    """
    ヘッセ行列 ∇^2 f(x) を計算する
    ∇^2 f(x) =
```

```
        [[ 2 + exp(x0),      -2          ],
         [      -2       , 12 x1^2 + 2   ]]
    """
    x0, x1 = x[0], x[1]
    h00 = 2.0 + np.exp(x0)
    h01 = -2.0
    h10 = -2.0
    h11 = 12.0 * x1**2 + 2.0
    return np.array([[h00, h01], [h10, h11]])


@dataclass(frozen=True)
class IterationHistory2D:
    x: np.ndarray  # shape (n, 2)
    f: np.ndarray  # shape (n,)


# ========================================================
# バックトラック法（共通で使う）
# ========================================================


def backtracking(xk, dk, evalf, evalg, t_init=1.0, rho=0.5, xi=1e-4):
    """
    バックトラック法（アルミホ条件）
    与えられた点 xk と方向 dk に対してステップ幅 t を返す。
    """
    t = t_init
    fk = evalf(xk)
    gk = evalg(xk)
    # アルミホ条件: f(xk + t dk) <= f(xk) + ξ t <dk, gk>
    while True:
        x_new = xk + t * dk
        if evalf(x_new) <= fk + xi * t * np.dot(dk, gk):
            break
        t *= rho
    return t


# ========================================================
# (a) バックトラック法付き最急降下法
# ========================================================


def steepest_descent_bt(
    x0,
    evalf,
    evalg,
    eps=1e-6,
    max_iter=1000,
    xi=1e-4,
    rho=0.5,
    t_init=1.0,
    *,
    return_history=False,
):
    """
    バックトラック法付き最急降下法
```

```python
    x_{k+1} = x_k + t_k d_k,  d_k = -∇f(x_k)
    """
    x = x0.copy()
    history_x = [x.copy()]
    history_f = [float(evalf(x))]

    for k in range(max_iter):
        g = evalg(x)
        norm_g = np.linalg.norm(g)
        if norm_g <= eps:
            if return_history:
                hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(
                    history_f))
                return x, evalf(x), k, hist
            return x, evalf(x), k

        dk = -g
        tk = backtracking(x, dk, evalf, evalg, t_init=t_init, rho=rho, xi=xi)
        x = x + tk * dk

        history_x.append(x.copy())
        history_f.append(float(evalf(x)))

    if return_history:
        hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(history_f))
        return x, evalf(x), max_iter, hist
    return x, evalf(x), max_iter


# ========================================================
# （b）バックトラック法付きニュートン法
# ========================================================


def newton_bt(
    x0,
    evalf,
    evalg,
    evalh,
    eps=1e-6,
    max_iter=1000,
    xi=1e-4,
    rho=0.5,
    t_init=1.0,
    *,
    return_history=False,
):
    """
    バックトラック法付きニュートン法
    d_k はヘッセ行列を用いて ∇^2 f(x_k) d_k = -∇f(x_k) を解く
    """
    x = x0.copy()
    history_x = [x.copy()]
    history_f = [float(evalf(x))]

    for k in range(max_iter):
        g = evalg(x)
        norm_g = np.linalg.norm(g)
```

```python
        if norm_g <= eps:
            if return_history:
                hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(
                    history_f))
                return x, evalf(x), k, hist
            return x, evalf(x), k

        H = evalh(x)
        # ニュートン方向を解く: H d = -g
        try:
            dk = np.linalg.solve(H, -g)
        except np.linalg.LinAlgError:
            break

        tk = backtracking(x, dk, evalf, evalg, t_init=t_init, rho=rho, xi=xi)
        x = x + tk * dk

        history_x.append(x.copy())
        history_f.append(float(evalf(x)))

    if return_history:
        hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(history_f))
        return x, evalf(x), max_iter, hist
    return x, evalf(x), max_iter


# ==========================================================
# メイン
# ==========================================================


def main():
    # 共通パラメータ(課題文指定)
    xi = 1e-4
    rho = 0.5
    t_init = 1.0
    x0 = np.array([1.0, 1.0])  # 初期点 (1, 1)^T

    print("初期点 x0 =", x0)

    # (a) 最急降下法
    print("\n=== (a) バックトラック法付き最急降下法 ===")
    x_sd, f_sd, k_sd, hist_sd = steepest_descent_bt(
        x0,
        evalf,
        evalg,
        eps=1e-6,
        max_iter=1000,
        xi=xi,
        rho=rho,
        t_init=t_init,
        return_history=True,
    )
    print("最適解近似 x* ?", x_sd)
    print("最適値近似 f(x*) ?", f_sd)
    print("反復回数 =", k_sd)

    # (b) ニュートン法
```

```python
    print("\n=== (b) バックトラック法付きニュートン法 ===")
    x_nt, f_nt, k_nt, hist_nt = newton_bt(
        x0,
        evalf,
        evalg,
        evalh,
        eps=1e-6,
        max_iter=1000,
        xi=xi,
        rho=rho,
        t_init=t_init,
        return_history=True,
    )
    print("最適解近似 x* ?", x_nt)
    print("最適値近似 f(x*) ?", f_nt)
    print("反復回数 =", k_nt)

    # x0-x1 平面に f の値を色で表示し、両手法の反復点の遷移を同一グラフに重ねる
    try:
        import matplotlib.pyplot as plt
    except ImportError:
        print("matplotlib が見つからないため、プロットをスキップします。")
        return

    all_points = np.vstack([hist_sd.x, hist_nt.x])
    x0_min, x1_min = np.min(all_points, axis=0)
    x0_max, x1_max = np.max(all_points, axis=0)

    margin0 = 0.2 * (x0_max - x0_min) if x0_max > x0_min else 1.0
    margin1 = 0.2 * (x1_max - x1_min) if x1_max > x1_min else 1.0
    x0_left, x0_right = x0_min - margin0, x0_max + margin0
    x1_bottom, x1_top = x1_min - margin1, x1_max + margin1

    n = 250
    grid_x0 = np.linspace(x0_left, x0_right, n)
    grid_x1 = np.linspace(x1_bottom, x1_top, n)
    X0, X1 = np.meshgrid(grid_x0, grid_x1)
    F = X0**2 + np.exp(X0) + X1**4 + X1**2 - 2.0 * X0 * X1 + 3.0

    plt.figure(figsize=(7, 6))
    cf = plt.contourf(X0, X1, F, levels=60, cmap="viridis")
    plt.contour(X0, X1, F, levels=20, colors="k", linewidths=0.4, alpha=0.35)
    cbar = plt.colorbar(cf)
    cbar.set_label("f(x0, x1)")

    plt.plot(hist_sd.x[:, 0], hist_sd.x[:, 1], "-o", markersize=3.5, linewidth=1.2,
        label="Steepest descent")
    plt.plot(hist_nt.x[:, 0], hist_nt.x[:, 1], "-s", markersize=3.5, linewidth=1.2,
        label="Newton")

    plt.scatter([hist_sd.x[0, 0]], [hist_sd.x[0, 1]], s=80, c="white", edgecolors="black", zorder=4)
    plt.scatter([hist_nt.x[0, 0]], [hist_nt.x[0, 1]], s=80, c="white", edgecolors="black", zorder=4)
    plt.scatter([hist_sd.x[-1, 0]], [hist_sd.x[-1, 1]], s=110, marker="*", c="white",
        edgecolors="black", zorder=5)
    plt.scatter([hist_nt.x[-1, 0]], [hist_nt.x[-1, 1]], s=110, marker="*", c="white",
        edgecolors="black", zorder=5)
```

```

    plt.xlabel("x0")
    plt.ylabel("x1")
    plt.title("Optimization trajectories on x0-x1 plane")
    plt.grid(True, linestyle="--", alpha=0.3)
    plt.legend()

    out_path = Path(__file__).with_name("task18.png")
    plt.tight_layout()
    plt.savefig(out_path, dpi=200)
    print(f"プロットを保存しました: {out_path}")

    import matplotlib

    if "agg" not in matplotlib.get_backend().lower():
        plt.show()


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

## A.5 課題 19 のコード

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import annotations

from dataclasses import dataclass
from pathlib import Path

import numpy as np

# =========================================================
# 課題19の設定
# f(x) = sum_{i=0}^2 f_i(x)^2
# f_i(x) = y_i - x0 (1 - x1^{i+1})
# y0=1.5, y1=2.25, y2=2.625
# =========================================================


Y_VALUES = np.array([1.5, 2.25, 2.625], dtype=float)


def fi_and_derivatives(x):
    """
    f_i(x), grad f_i(x), Hessian f_i(x) をまとめて返す
    """
    x0, x1 = x[0], x[1]
    fi_list = []
    gi_list = []
    hi_list = []

    for i, yi in enumerate(Y_VALUES):
        a = i + 1
        x1a = x1**a
```

```
        fi = yi - x0 + x0 * x1a

        # grad f_i
        dfi_dx0 = -1.0 + x1a
        dfi_dx1 = x0 * a * (x1 ** (a - 1))
        gi = np.array([dfi_dx0, dfi_dx1], dtype=float)

        # Hessian f_i
        d2fi_dx0dx0 = 0.0
        d2fi_dx0dx1 = a * (x1 ** (a - 1))
        if a == 1:
            d2fi_dx1dx1 = 0.0
        else:
            d2fi_dx1dx1 = x0 * a * (a - 1) * (x1 ** (a - 2))
        hi = np.array(
            [[d2fi_dx0dx0, d2fi_dx0dx1], [d2fi_dx0dx1, d2fi_dx1dx1]],
            dtype=float,
        )

        fi_list.append(fi)
        gi_list.append(gi)
        hi_list.append(hi)

    return fi_list, gi_list, hi_list


def evalf(x):
    """
    目的関数 f(x) = sum f_i(x)^2
    """
    fi_list, _, _ = fi_and_derivatives(x)
    return float(np.sum(np.square(fi_list)))


def evalg(x):
    """
    勾配ベクトル ∇f(x) = 2 * sum f_i(x) * ∇f_i(x)
    """
    fi_list, gi_list, _ = fi_and_derivatives(x)
    grad = np.zeros(2, dtype=float)
    for fi, gi in zip(fi_list, gi_list):
        grad += 2.0 * fi * gi
    return grad


def evalh(x):
    """
    ヘッセ行列 ∇^2 f(x)
    = 2 * sum ( f_i(x) * ∇^2 f_i(x) + ∇f_i(x) ∇f_i(x)^T )
    """
    fi_list, gi_list, hi_list = fi_and_derivatives(x)
    hess = np.zeros((2, 2), dtype=float)
    for fi, gi, hi in zip(fi_list, gi_list, hi_list):
        hess += 2.0 * (fi * hi + np.outer(gi, gi))
    return hess


@dataclass(frozen=True)
```

```python
class IterationHistory2D:
    x: np.ndarray  # shape (n, 2)
    f: np.ndarray  # shape (n,)


# ==========================================================
# バックトラック法（共通で使う）
# ==========================================================


def backtracking(xk, dk, evalf, evalg, t_init=1.0, rho=0.5, xi=1e-4):
    """
    バックトラック法（アルミホ条件）
    """
    t = t_init
    fk = evalf(xk)
    gk = evalg(xk)
    while True:
        x_new = xk + t * dk
        if evalf(x_new) <= fk + xi * t * np.dot(dk, gk):
            break
        t *= rho
    return t


# ==========================================================
# (a) バックトラック法付き最急降下法
# ==========================================================


def steepest_descent_bt(
    x0,
    evalf,
    evalg,
    eps=1e-6,
    max_iter=1000,
    xi=1e-4,
    rho=0.5,
    t_init=1.0,
    *,
    return_history=False,
):
    x = x0.copy()
    history_x = [x.copy()]
    history_f = [float(evalf(x))]

    for k in range(max_iter):
        g = evalg(x)
        if np.linalg.norm(g) <= eps:
            if return_history:
                hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(
                    history_f))
                return x, evalf(x), k, hist
            return x, evalf(x), k

        dk = -g
        tk = backtracking(x, dk, evalf, evalg, t_init=t_init, rho=rho, xi=xi)
        x = x + tk * dk
```

```python

        history_x.append(x.copy())
        history_f.append(float(evalf(x)))

    if return_history:
        hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(history_f))
        return x, evalf(x), max_iter, hist
    return x, evalf(x), max_iter


# ============================================================
# (b) バックトラック法付きニュートン法
# ============================================================


def newton_bt(
    x0,
    evalf,
    evalg,
    evalh,
    eps=1e-6,
    max_iter=1000,
    xi=1e-4,
    rho=0.5,
    t_init=1.0,
    *,
    return_history=False,
):
    x = x0.copy()
    history_x = [x.copy()]
    history_f = [float(evalf(x))]

    for k in range(max_iter):
        g = evalg(x)
        if np.linalg.norm(g) <= eps:
            if return_history:
                hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(
                    history_f))
                return x, evalf(x), k, hist
            return x, evalf(x), k

        H = evalh(x)
        try:
            dk = np.linalg.solve(H, -g)
        except np.linalg.LinAlgError:
            break

        tk = backtracking(x, dk, evalf, evalg, t_init=t_init, rho=rho, xi=xi)
        x = x + tk * dk

        history_x.append(x.copy())
        history_f.append(float(evalf(x)))

    if return_history:
        hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(history_f))
        return x, evalf(x), max_iter, hist
    return x, evalf(x), max_iter

```

```
# ========================================================
# メイン
# ========================================================


def main():
    xi = 1e-4
    rho = 0.5
    t_init = 1.0
    x0 = np.array([2.0, 0.0])

    print("初期点 x0 =", x0)

    print("\n=== (a) バックトラック法付き最急降下法 ===")
    x_sd, f_sd, k_sd, hist_sd = steepest_descent_bt(
        x0,
        evalf,
        evalg,
        eps=1e-6,
        max_iter=1000,
        xi=xi,
        rho=rho,
        t_init=t_init,
        return_history=True,
    )
    print("最適解近似 x* ?", x_sd)
    print("最適値近似 f(x*) ?", f_sd)
    print("反復回数 =", k_sd)

    print("\n=== (b) バックトラック法付きニュートン法 ===")
    x_nt, f_nt, k_nt, hist_nt = newton_bt(
        x0,
        evalf,
        evalg,
        evalh,
        eps=1e-6,
        max_iter=1000,
        xi=xi,
        rho=rho,
        t_init=t_init,
        return_history=True,
    )
    print("最適解近似 x* ?", x_nt)
    print("最適値近似 f(x*) ?", f_nt)
    print("反復回数 =", k_nt)

    # プロット（任意）
    try:
        import matplotlib.pyplot as plt
    except ImportError:
        print("matplotlib が見つからないため、プロットをスキップします。")
        return

    all_points = np.vstack([hist_sd.x, hist_nt.x])
    x0_min, x1_min = np.min(all_points, axis=0)
    x0_max, x1_max = np.max(all_points, axis=0)
    margin0 = 0.2 * (x0_max - x0_min) if x0_max > x0_min else 1.0
```

```python
    margin1 = 0.2 * (x1_max - x1_min) if x1_max > x1_min else 1.0

    x0_left, x0_right = x0_min - margin0, x0_max + margin0
    x1_bottom, x1_top = x1_min - margin1, x1_max + margin1

    n = 250
    grid_x0 = np.linspace(x0_left, x0_right, n)
    grid_x1 = np.linspace(x1_bottom, x1_top, n)
    X0, X1 = np.meshgrid(grid_x0, grid_x1)

    F = np.zeros_like(X0)
    for i, yi in enumerate(Y_VALUES):
        a = i + 1
        F += (yi - X0 + X0 * (X1**a)) ** 2

    plt.figure(figsize=(7, 6))
    cf = plt.contourf(X0, X1, F, levels=60, cmap="viridis")
    plt.contour(X0, X1, F, levels=20, colors="k", linewidths=0.4, alpha=0.35)
    cbar = plt.colorbar(cf)
    cbar.set_label("f(x0, x1)")

    plt.plot(hist_sd.x[:, 0], hist_sd.x[:, 1], "-o", markersize=3.5, linewidth=1.2, label="Steepest descent")
    plt.plot(hist_nt.x[:, 0], hist_nt.x[:, 1], "-s", markersize=3.5, linewidth=1.2, label="Newton")

    plt.scatter([hist_sd.x[0, 0]], [hist_sd.x[0, 1]], s=80, c="white", edgecolors="black", zorder=4)
    plt.scatter([hist_nt.x[0, 0]], [hist_nt.x[0, 1]], s=80, c="white", edgecolors="black", zorder=4)
    plt.scatter([hist_sd.x[-1, 0]], [hist_sd.x[-1, 1]], s=110, marker="*", c="white", edgecolors="black", zorder=5)
    plt.scatter([hist_nt.x[-1, 0]], [hist_nt.x[-1, 1]], s=110, marker="*", c="white", edgecolors="black", zorder=5)

    plt.xlabel("x0")
    plt.ylabel("x1")
    plt.title("Optimization trajectories on x0-x1 plane")
    plt.grid(True, linestyle="--", alpha=0.3)
    plt.legend()

    out_path = Path(__file__).with_name("task19.png")
    plt.tight_layout()
    plt.savefig(out_path, dpi=200)
    print(f"プロットを保存しました: {out_path}")

    import matplotlib

    if "agg" not in matplotlib.get_backend().lower():
        plt.show()


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

## A.6 追加課題のコード

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy as np


def primal_dual_ipm_qp(Q, c, A, b, *, max_iter=80, tol=1e-12, sigma=0.1, tau=0.99):
    """
    Primal-Dual Interior-Point (Newton) for convex QP:
        minimize    1/2 x^T Q x + c^T x
        subject to A x <= b    (b is scalar or length-m)

    Slack: y >= 0 such that  A x + y - b = 0
    Dual:  λ >= 0
    Perturbed complementarity: y_i λ_i = sigma * mu,
        mu = (y^T λ)/m
    """
    Q = np.asarray(Q, dtype=float)
    c = np.asarray(c, dtype=float).reshape(-1)
    A = np.asarray(A, dtype=float)
    b = np.asarray(b, dtype=float).reshape(-1)

    n = Q.shape[0]
    m = A.shape[0]

    # 半正定値チェック（数値誤差許容）
    Qsym = 0.5 * (Q + Q.T)
    if np.linalg.eigvalsh(Qsym).min() < -1e-10:
        raise ValueError("Q must be positive semidefinite.")

    # 初期点（内点）
    x = np.zeros(n)
    y = np.ones(m)
    lam = np.ones(m)
    e = np.ones(m)

    for k in range(max_iter):
        # 残差
        rd = Qsym @ x + c + A.T @ lam          # dual residual
        rp = A @ x + y - b                      # primal residual
        mu = (y @ lam) / m                      # complementarity
        rc = y * lam - sigma * mu * e           # perturbed complementarity

        # 収束判定
        if max(np.linalg.norm(rd), np.linalg.norm(rp), mu) < tol:
            return x, y, lam, {"iter": k, "rd": np.linalg.norm(rd),
                               "rp": np.linalg.norm(rp), "mu": mu}

        # Newton 方程式
        # [ Q   0    A^T ] [dx] = -[rd]
        # [ A   I    0   ] [dy]   -[rp]
        # [ 0   Λ    Y   ] [dλ]   -[rc]
        K = np.block([
            [Qsym,                  np.zeros((n, m)),      A.T],
```

```
            [A,                   np.eye(m),             np.zeros((m, m))],
            [np.zeros((m, n)),    np.diag(lam),          np.diag(y)]
        ])
        rhs = -np.concatenate([rd, rp, rc])

        delta = np.linalg.solve(K, rhs)
        dx = delta[:n]
        dy = delta[n:n+m]
        dlam = delta[n+m:]

        # 正性を保つステップ長
        alpha = 1.0
        if np.any(dy < 0):
            alpha = min(alpha, tau * np.min(-y[dy < 0] / dy[dy < 0]))
        if np.any(dlam < 0):
            alpha = min(alpha, tau * np.min(-lam[dlam < 0] / dlam[dlam < 0]))

        # 更新
        x += alpha * dx
        y += alpha * dy
        lam += alpha * dlam

        # 数値安全
        y = np.maximum(y, 1e-18)
        lam = np.maximum(lam, 1e-18)

    return x, y, lam, {"iter": max_iter, "rd": np.linalg.norm(rd),
                       "rp": np.linalg.norm(rp), "mu": mu}


if __name__ == "__main__":
    # 課題4：パラメータ
    Q = np.array([[2.0, 0.0],
                  [0.0, 1.0]])      # 半正定値行列
    c = -np.array([1.0, 1.0])

    # b はスカラー 0 と解釈
    A = np.array([[1.0, 1.0]])      # 1 本の制約
    b = 0.0

    x_star, y_star, lam_star, info = primal_dual_ipm_qp(Q, c, A, b)

    print("x* =", x_star)
    print("slack y =", y_star)
    print("dual λ =", lam_star)
    print("info =", info)
```

# 参考文献

# 参考文献


[1] 数理工学実験（2025 年度配布資料）.

[2] Pardalos, P. M. and Vavasis, S. A. (1991). “Quadratic programming with one negative eigenvalue is NP-hard,” *Journal of Global Optimization*, 1(1), pp. 15–22.

[3] Sahni, S. (1974). "Computationally related problems," *SIAM Journal on Computing*, 3, pp. 262–279.