

数理工学実験レポート

第 6 章（連続最適化）

学籍番号 1029366161 中塚一瑛

2026 年 1 月 4 日

目次

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	3
1.1	原理と方法	3
1.2	実験方法	3
1.3	結果	4
1.4	考察	4
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	5
2.1	原理と方法	5
2.2	実装	5
2.3	結果	5
2.4	考察	6
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	6
3.1	微分結果	7
3.2	コードとの対応	7
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	8
4.1	原理と方法	8
4.2	実験方法	8
4.3	結果	8
4.4	考察	9
5	課題 19：総合演習	9
5.1	原理と方法	9
5.2	実験方法	10
5.3	結果	10
5.4	考察	10
6	追加課題：制約付き凸最適化問題	11

6.1	課題 1	11
6.2	課題 2	12
6.3	課題 3	12
6.4	課題 4	13
付録 A	ソースコード	14
A.1	課題 15 のコード	14
A.2	課題 16 のコード	16
A.3	課題 17 のコード	20
A.4	課題 18 のコード	21
A.5	課題 19 のコード	26
A.6	追加課題のコード	32

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

$$f(x) = x^3 + 2x^2 - 5x - 6 \quad (1)$$

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、関数がある点 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

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

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

$$f'(x) = 3x^2 + 4x - 5 \quad (3)$$

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

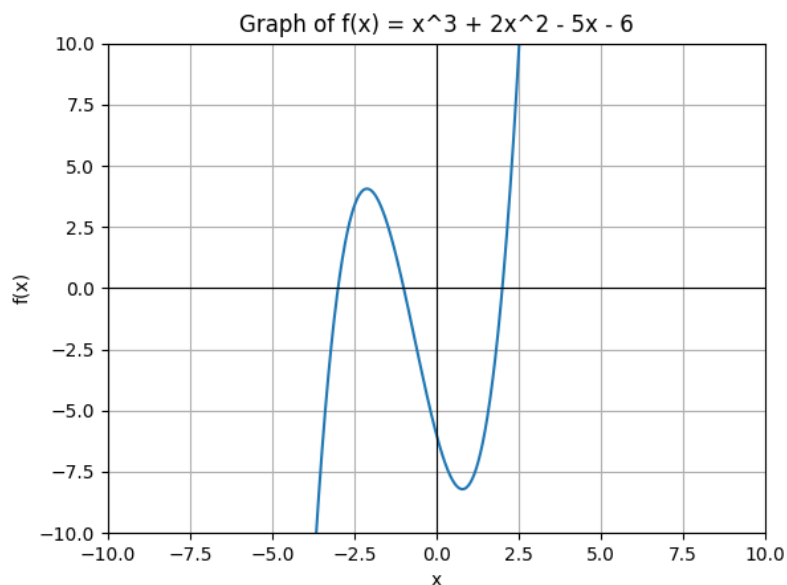


図 1: $f(x) = x^3 + 2x^2 - 5x - 6$ のグラフ

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	1.421×10^{-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

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

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

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

$$f''(x) = 2x - 2 \quad (6)$$

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

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

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

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

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

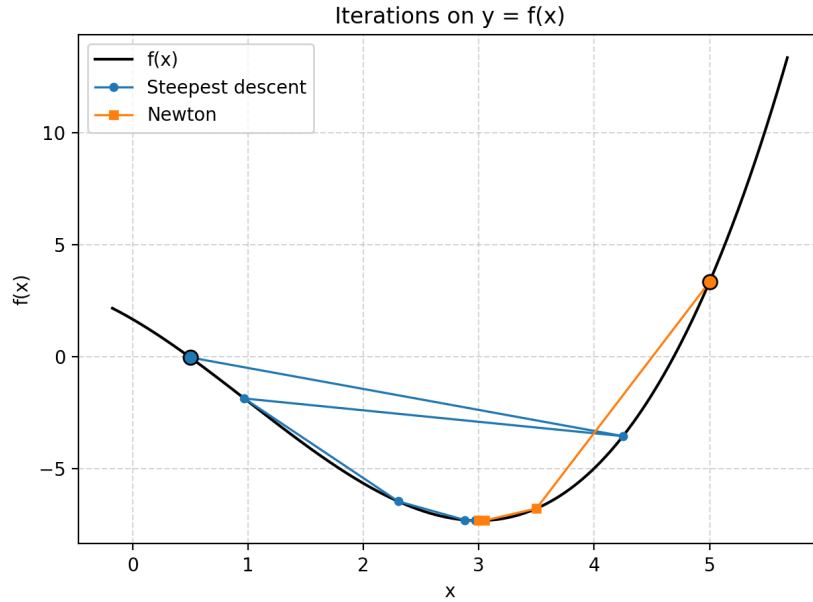


図 2: 関数 $y = f(x)$ 上における反復点の推移. 青丸は最急降下法, 橙四角はニュートン法による反復点を表す.

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

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

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

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

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

である．従って勾配ベクトルは

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

と表される [1].

次に，2 階微分よりヘッセ行列は

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

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

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

- 勾配

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

- ヘッセ行列

```
1 h00 = 2 + exp(x0)
2 h01 = -2
3 h10 = -2
4 h11 = 12*x1**2 + 2
5 H = [[h00, h01],
6      [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

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

を最小化する。勾配およびヘッセ行列は

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

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

である。更新は

$$x_{k+1} = x_k + t_k d_k \quad (17)$$

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 はバックトラッキング法 (Armijo 条件) で決定した [1]. すなわち、 $t \leftarrow t_{\text{init}}$ から開始し、

$$f(x_k + t d_k) \leq f(x_k) + \xi t \langle \nabla f(x_k), d_k \rangle \quad (18)$$

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 $x_0 = (1, 1)^T$ とし、共通パラメータを

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

とした。停止条件は $\|\nabla f(x_k)\| \leq 1.0 \times 10^{-6}$ 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) バックトラッキング法付き最急降下法および (b) バックトラッキング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。

table 3 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 $x_0 = (1, 1)^T$)

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

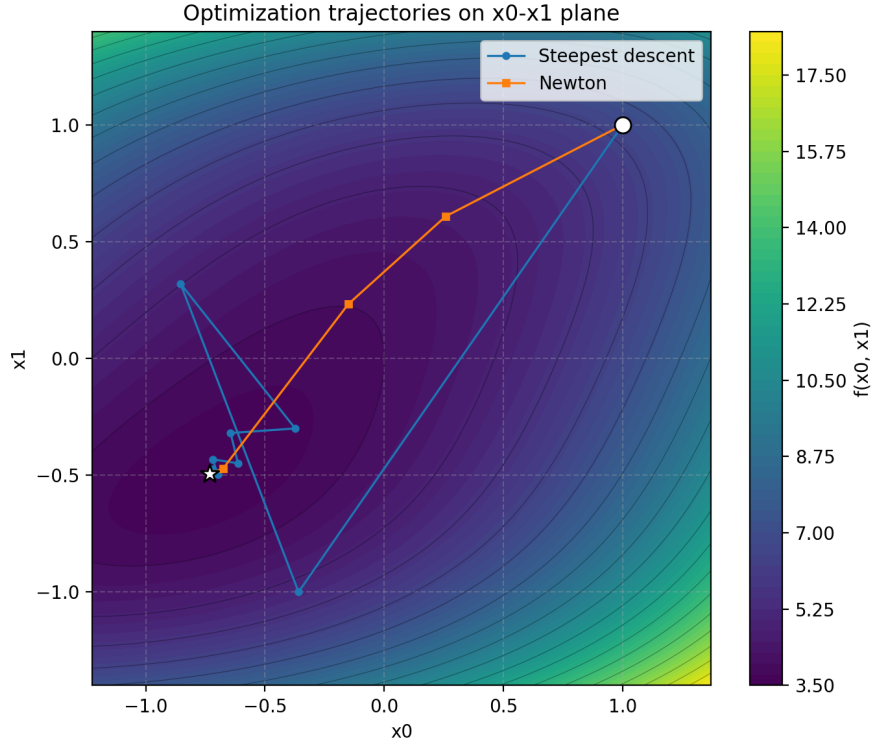


図 3: x_0 - x_1 平面における等高線図と反復点列. \circ が最急降下法 +BT, \square がニュートン法 +BT の反復点列を表す.

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, $x = (x_0, x_1)^T \in \mathbb{R}^2$ に対し

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

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

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

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

である. よって, 勾配・ヘッセ行列は課題 18 と同様に

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

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

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

$$\xi = 1.0 \times 10^{-4}, \quad \rho = 0.5, \quad t_{\text{init}} = 1 \quad (25)$$

とした. 停止条件は $\|\nabla f(x_k)\| \leq 1.0 \times 10^{-6}$ 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	7.68×10^{-13}	791
ニュートン法 +BT	3.00000013	0.50000005	9.47×10^{-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.

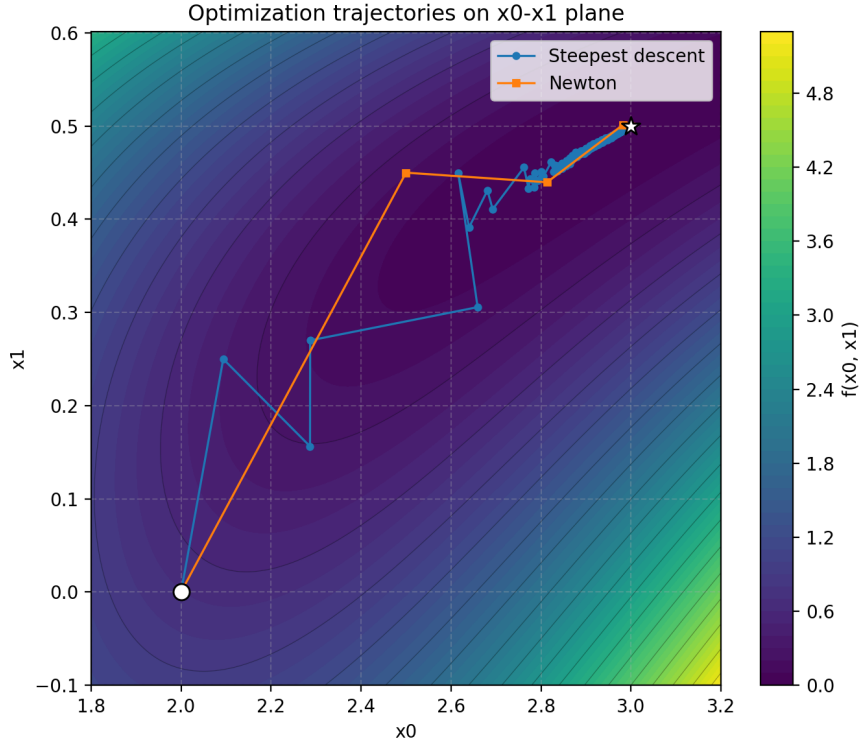


図 4: x_0 - x_1 平面における等高線図と反復点列. \circ が最急降下法 +BT, \square がニュートン法 +BT の反復点列を表す.

6 追加課題：制約付き凸最適化問題

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} x^\top Q x + c^\top x \quad \text{subject to} \quad A x \leq b,$$

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 $(a_i, y_i) \in \mathbb{R}^n \times \mathbb{R}$ ($i = 1, \dots, m$) に対し、

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

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

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

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

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

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ので、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 $Y = \text{diag}(y_1, \dots, y_m)$, $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_m)$, $e = (1, \dots, 1)^\top$, $\sigma \in (0, 1)$ は固定定数である。 $\sigma\mu > 0$ の解は central path を成し、 $\sigma\mu \rightarrow 0$ 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

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

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, $\|r_p\|$, $\|r_d\|$, μ 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 $Ax \leq b$ 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。

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

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

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

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。

$$\begin{aligned} \mathbf{x}^* &= \begin{pmatrix} -2.809 \times 10^{-13} \\ -5.618 \times 10^{-13} \end{pmatrix} \\ y &= (8.427 \times 10^{-13}) \\ \lambda &= (1.000) \end{aligned}$$

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

つまり, 最適解は

$$\begin{aligned} x^* &\approx \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ y &\approx (0) \\ \lambda &\approx (1) \end{aligned}$$

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

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import numpy as np
5  import matplotlib.pyplot as plt
6
7  # -----
8  # 課題15: 関数の定義
9  #  $f(x) = x^3 + 2x^2 - 5x - 6$ 
10 # -----
11
12 def f(x):
13     return x**3 + 2*x**2 - 5*x - 6
14
15 def df(x):
16     """ $f(x)$  の導関数:  $f'(x) = 3x^2 + 4x - 5$ """
17     return 3*x**2 + 4*x - 5
18
19
20 # -----
21 # (a) グラフ描画
22 # -----
23
24 def plot_function():
25     x = np.linspace(-10, 10, 2000)
26     y = f(x)
27
28     plt.figure()
29     plt.plot(x, y)
30     plt.axhline(0, color="black", linewidth=0.8) # x軸
31     plt.axvline(0, color="black", linewidth=0.8) # y軸
32     plt.xlim(-10, 10)
33     plt.ylim(-10, 10)
34     plt.xlabel("x")
35     plt.ylabel("f(x)")
36     plt.title("Graph of  $f(x) = x^3 + 2x^2 - 5x - 6$ ")
37     plt.grid(True)
38     plt.savefig("task15.png")
39     plt.show()
40
```

```

41
42 # -----
43 # (b) 二分法
44 # -----
45
46 def bisection(f, a, b, eps=1e-10, max_iter=1000):
47     """[a, b] で二分法により  $f(x) = 0$  の解を求める.
48          $f(a)$  と  $f(b)$  の符号は異なることが前提。
49     """
50     fa = f(a)
51     fb = f(b)
52     if fa * fb > 0:
53         raise ValueError("f(a) と f(b) の符号が同じです: a={}, b={}".format(a, b))
54
55     for _ in range(max_iter):
56         c = 0.5 * (a + b)
57         fc = f(c)
58
59         if abs(fc) <= eps or 0.5 * (b - a) < eps:
60             return c
61
62         # 符号でどちらの区間を残すか決める
63         if fa * fc < 0:
64             b = c
65             fb = fc
66         else:
67             a = c
68             fa = fc
69
70     # 最大反復に達した場合
71     return 0.5 * (a + b)
72
73
74 def solve_with_bisection():
75     # グラフから零点が -3, -1, 2 付近にあることが分かるので
76     # それを挟む区間を手で指定する
77     intervals = [
78         (-4.0, -2.0), # -3 付近
79         (-2.0, 0.0), # -1 付近
80         (1.0, 3.0), # 2 付近
81     ]
82
83     roots = []
84     for (a, b) in intervals:
85         r = bisection(f, a, b)
86         roots.append(r)
87     return roots
88
89 # -----
90 # (c) ニュートン法
91 # -----
92
93
94 def newton(f, df, x0, eps=1e-10, max_iter=1000):
95     """ニュートン法:  $f(x) = 0$  の解を初期値  $x_0$  から探索."""
96     x = x0
97     for _ in range(max_iter):
98         fx = f(x)

```

```

99         dfx = df(x)
100
101         if abs(fx) <= eps:
102             return x
103
104         if dfx == 0:
105             # 導関数が 0 になると更新できない
106             raise ZeroDivisionError("f'(x) = 0 となったため打ち切り (x={})".format(x))
107
108         x = x - fx / dfx
109
110     return x # 収束しなかった場合は最後の値を返す
111
112
113 def solve_with_newton():
114     # グラフから零点が -3, -1, 2 付近にあることを利用
115     initial_points = [-2.5, -0.5, 1.5]
116     roots = []
117     for x0 in initial_points:
118         r = newton(f, df, x0)
119         roots.append(r)
120     return roots
121
122
123 # -----
124 # メイン
125 # -----
126
127 def main():
128     # (a) グラフ描画
129     plot_function()
130
131     # (b) 二分法
132     bisection_roots = solve_with_bisection()
133     print("Bisection method roots:")
134     for r in bisection_roots:
135         print(" x ≈ {:.10f}, f(x) ≈ {:.3e}".format(r, f(r)))
136
137     # (c) ニュートン法
138     newton_roots = solve_with_newton()
139     print("\nNewton method roots:")
140     for r in newton_roots:
141         print(" x0 -> root ≈ {:.10f}, f(x) ≈ {:.3e}".format(r, f(r)))
142
143
144 if __name__ == "__main__":
145     main()

```

A.2 課題 16 のコード

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  # 課題16
5  # f(x) = (1/3)x^3 - x^2 - 3x + 5/3 の停留点を求める
6  # (a) 最急降下法 (x0 = 1/2, tk = 1/(k+1))

```



```

7 # (b) ニュートン法 (x0 = 5, tk = 1)
8
9 from __future__ import annotations
10
11 from dataclasses import dataclass
12 from pathlib import Path
13
14
15 def f(x):
16     """目的関数  $f(x) = \frac{1}{3}x^3 - x^2 - 3x + \frac{5}{3}$ """
17     return (1.0 / 3.0) * x**3 - x**2 - 3.0 * x + 5.0 / 3.0
18
19
20 def df(x):
21     """1階微分  $f'(x) = x^2 - 2x - 3$ """
22     return x**2 - 2.0 * x - 3.0
23
24
25 def d2f(x):
26     """2階微分  $f''(x) = 2x - 2$ """
27     return 2.0 * x - 2.0
28
29
30 @dataclass(frozen=True)
31 class IterationHistory:
32     x: list[float]
33     f: list[float]
34     grad_abs: list[float]
35
36
37 # -----
38 # (a) 最急降下法
39 # -----
40
41
42 def steepest_descent(x0, eps=1e-8, max_iter=1000, *, return_history=False):
43     """
44     最急降下法 (1次元版)
45      $x_{k+1} = x_k - t_k * f'(x_k)$ 
46      $t_k = 1 / (k+1)$ 
47     """
48     x = x0
49     history_x = [float(x)]
50     history_f = [float(f(x))]
51     history_grad_abs = [float(abs(df(x)))]
52
53     for k in range(max_iter):
54         g = df(x) # 勾配 (1次元なので単なる導関数)
55         if abs(g) <= eps:
56             # 停留点に到達したとみなす
57             if return_history:
58                 return x, k, IterationHistory(history_x, history_f, history_grad_abs)
59             return x, k
60
61         t_k = 1.0 / (k + 1) # 指定どおりのステップ幅
62         x = x - t_k * g
63
64         history_x.append(float(x))

```

```

65     history_f.append(float(f(x)))
66     history_grad_abs.append(float(abs(df(x))))
67
68     # 最大反復に到達した場合
69     if return_history:
70         return x, max_iter, IterationHistory(history_x, history_f, history_grad_abs)
71     return x, max_iter
72
73
74 # -----
75 # (b) ニュートン法
76 # -----
77
78
79 def newton_method(x0, eps=1e-8, max_iter=1000, *, return_history=False):
80     """
81     ニュートン法
82      $x_{k+1} = x_k - t_k * f'(x_k) / f''(x_k)$ 
83     ここでは  $t_k = 1$ 
84     """
85     x = x0
86     history_x = [float(x)]
87     history_f = [float(f(x))]
88     history_grad_abs = [float(abs(df(x)))]
89
90     for k in range(max_iter):
91         g = df(x)
92         h = d2f(x)
93
94         if abs(g) <= eps:
95             # 停留点に到達したとみなす
96             if return_history:
97                 return x, k, IterationHistory(history_x, history_f, history_grad_abs)
98             return x, k
99
100         if h == 0.0:
101             raise ZeroDivisionError(f"f''(x) = 0 となったため更新できません (x = {x})")
102
103         t_k = 1.0 # 指定どおり常に1
104         x = x - t_k * g / h
105
106         history_x.append(float(x))
107         history_f.append(float(f(x)))
108         history_grad_abs.append(float(abs(df(x))))
109
110     if return_history:
111         return x, max_iter, IterationHistory(history_x, history_f, history_grad_abs)
112     return x, max_iter
113
114 # -----
115 # メイン
116 # -----
117
118
119
120 def main():
121     print("=== (a) 最急降下法 ===")

```

```

122 x0_sd = 0.5 # 初期点 x0 = 1/2
123 x_star_sd, it_sd, hist_sd = steepest_descent(x0_sd, return_history=True)
124 print("初期値 x0 = {:.6f}".format(x0_sd))
125 print("近似停留点 x* ? {:.10f}".format(x_star_sd))
126 print("f'(x*) ? {:.3e}".format(df(x_star_sd)))
127 print("反復回数 k =", it_sd)
128 print()
129
130 print("=== (b) ニュートン法 ===")
131 x0_nt = 5.0 # 初期点 x0 = 5
132 x_star_nt, it_nt, hist_nt = newton_method(x0_nt, return_history=True)
133 print("初期値 x0 = {:.6f}".format(x0_nt))
134 print("近似停留点 x* ? {:.10f}".format(x_star_nt))
135 print("f'(x*) ? {:.3e}".format(df(x_star_nt)))
136 print("反復回数 k =", it_nt)
137 print()
138
139 # 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
140 try:
141     import matplotlib.pyplot as plt
142 except ImportError:
143     print("matplotlib が見つからないため、プロットをスキップします。")
144     return
145
146 all_x = hist_sd.x + hist_nt.x
147 x_min = min(all_x)
148 x_max = max(all_x)
149 margin = 0.15 * (x_max - x_min) if x_max > x_min else 1.0
150 x_left = x_min - margin
151 x_right = x_max + margin
152
153 n = 600
154 xs = [x_left + (x_right - x_left) * i / n for i in range(n + 1)]
155 ys = [f(x) for x in xs]
156
157 plt.figure()
158 plt.plot(xs, ys, color="black", linewidth=1.5, label="f(x)")
159
160 plt.plot(hist_sd.x, hist_sd.f, marker="o", markersize=4, linewidth=1.2, label="
    Steepest descent")
161 plt.plot(hist_nt.x, hist_nt.f, marker="s", markersize=4, linewidth=1.2, label="
    Newton")
162
163 plt.scatter([hist_sd.x[0]], [hist_sd.f[0]], s=60, edgecolors="black", zorder=3)
164 plt.scatter([hist_nt.x[0]], [hist_nt.f[0]], s=60, edgecolors="black", zorder=3)
165
166 plt.xlabel("x")
167 plt.ylabel("f(x)")
168 plt.title("Iterations on y = f(x)")
169 plt.grid(True, linestyle="--", alpha=0.5)
170 plt.legend()
171
172 out_path = Path(__file__).with_name("task16.png")
173 plt.tight_layout()
174 plt.savefig(out_path, dpi=200)
175 print(f"プロットを保存しました: {out_path}")
176 import matplotlib
177

```

```

178     if "agg" not in matplotlib.get_backend().lower():
179         plt.show()
180
181
182 if __name__ == "__main__":
183     main()

```

A.3 課題 17 のコード

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import numpy as np
5
6  #  $f(x) = x_0^2 + \exp(x_0) + x_1^4 + x_1^2 - 2 x_0 x_1 + 3$ 
7
8  def evalf(x):
9      """
10     目的関数の値  $f(x)$  を計算する
11     x: np.array shape (2,)
12     """
13     x0, x1 = x[0], x[1]
14     f = x0**2 + np.exp(x0) + x1**4 + x1**2 - 2.0*x0*x1 + 3.0
15     return f
16
17  def evalg(x):
18      """
19     勾配ベクトル  $\nabla f(x)$  を計算する
20      $\nabla f(x) = [ 2x_0 + \exp(x_0) - 2x_1, 4x_1^3 + 2x_1 - 2x_0 ]$ 
21     """
22
23     x0, x1 = x[0], x[1]
24     g0 = 2.0*x0 + np.exp(x0) - 2.0*x1
25     g1 = 4.0*x1**3 + 2.0*x1 - 2.0*x0
26     g = np.array([g0, g1])
27     return g
28
29  def evalh(x):
30      """
31     ヘッセ行列  $\nabla^2 f(x)$  を計算する
32      $\nabla^2 f(x) =$ 
33      $\begin{bmatrix} 2 + \exp(x_0) & -2 \\ -2 & 12 x_1^2 + 2 \end{bmatrix}$ 
34     """
35
36     x0, x1 = x[0], x[1]
37     h00 = 2.0 + np.exp(x0)
38     h01 = -2.0
39     h10 = -2.0
40     h11 = 12.0*x1**2 + 2.0
41     H = np.array([[h00, h01],
42                   [h10, h11]])
43     return H
44
45  def main():
46     # 動作確認用
47     x = np.array([0.3, 5.0])

```

```

48     f = evalf(x)
49     g = evalg(x)
50     H = evalh(x)
51
52     print("x =", x)
53     print("f(x) =", f)
54     print("g(x) =", g)
55     print("H(x) =\n", H)
56
57 if __name__ == "__main__":
58     main()

```

A.4 課題 18 のコード

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from __future__ import annotations
5
6  from dataclasses import dataclass
7  from pathlib import Path
8
9  import numpy as np
10
11 # =====
12 # 課題17の設定
13 #  $f(x) = x_0^2 + \exp(x_0) + x_1^4 + x_1^2 - 2 x_0 x_1 + 3$ 
14 # =====
15
16
17 def evalf(x):
18     """
19     目的関数  $f(x)$  を計算する
20     x: np.array shape (2,)
21     """
22     x0, x1 = x[0], x[1]
23     f = x0**2 + np.exp(x0) + x1**4 + x1**2 - 2.0 * x0 * x1 + 3.0
24     return f
25
26
27 def evalg(x):
28     """
29     勾配ベクトル  $\nabla f(x)$  を計算する
30      $\nabla f(x) = [ 2x_0 + \exp(x_0) - 2x_1,$ 
31                   $4x_1^3 + 2x_1 - 2x_0 ]$ 
32     """
33     x0, x1 = x[0], x[1]
34     g0 = 2.0 * x0 + np.exp(x0) - 2.0 * x1
35     g1 = 4.0 * x1**3 + 2.0 * x1 - 2.0 * x0
36     return np.array([g0, g1])
37
38
39 def evalh(x):
40     """
41     ヘッセ行列  $\nabla^2 f(x)$  を計算する
42      $\nabla^2 f(x) =$ 

```

```

43     [[ 2 + exp(x0),      -2      ],
44      [      -2      , 12 x1^2 + 2  ]]
45     """
46     x0, x1 = x[0], x[1]
47     h00 = 2.0 + np.exp(x0)
48     h01 = -2.0
49     h10 = -2.0
50     h11 = 12.0 * x1**2 + 2.0
51     return np.array([[h00, h01], [h10, h11]])
52
53
54 @dataclass(frozen=True)
55 class IterationHistory2D:
56     x: np.ndarray # shape (n, 2)
57     f: np.ndarray # shape (n,)
58
59
60 # =====
61 # バックトラック法（共通で使う）
62 # =====
63
64
65 def backtracking(xk, dk, evalf, evalg, t_init=1.0, rho=0.5, xi=1e-4):
66     """
67     バックトラック法（アルミホ条件）
68     与えられた点 xk と方向 dk に対してステップ幅 t を返す。
69     """
70     t = t_init
71     fk = evalf(xk)
72     gk = evalg(xk)
73     # アルミホ条件:  $f(x_k + t dk) \leq f(x_k) + \xi t \langle dk, gk \rangle$ 
74     while True:
75         x_new = xk + t * dk
76         if evalf(x_new) <= fk + xi * t * np.dot(dk, gk):
77             break
78         t *= rho
79     return t
80
81
82 # =====
83 # (a) バックトラック法付き最急降下法
84 # =====
85
86
87 def steepest_descent_bt(
88     x0,
89     evalf,
90     evalg,
91     eps=1e-6,
92     max_iter=1000,
93     xi=1e-4,
94     rho=0.5,
95     t_init=1.0,
96     *,
97     return_history=False,
98 ):
99     """
100     バックトラック法付き最急降下法

```

```

101 x_{k+1} = x_k + t_k d_k, d_k = -∇f(x_k)
102 """
103 x = x0.copy()
104 history_x = [x.copy()]
105 history_f = [float(evalf(x))]
106
107 for k in range(max_iter):
108     g = evalg(x)
109     norm_g = np.linalg.norm(g)
110     if norm_g <= eps:
111         if return_history:
112             hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(
113                 history_f))
114             return x, evalf(x), k, hist
115         return x, evalf(x), k
116
117     dk = -g
118     tk = backtracking(x, dk, evalf, evalg, t_init=t_init, rho=rho, xi=xi)
119     x = x + tk * dk
120
121     history_x.append(x.copy())
122     history_f.append(float(evalf(x)))
123
124 if return_history:
125     hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(history_f))
126     return x, evalf(x), max_iter, hist
127 return x, evalf(x), max_iter
128
129 # =====
130 # (b) バックトラック法付きニュートン法
131 # =====
132
133
134 def newton_bt(
135     x0,
136     evalf,
137     evalg,
138     evalh,
139     eps=1e-6,
140     max_iter=1000,
141     xi=1e-4,
142     rho=0.5,
143     t_init=1.0,
144     *,
145     return_history=False,
146 ):
147     """
148     バックトラック法付きニュートン法
149     d_k はヘッセ行列を用いて ∇^2 f(x_k) d_k = -∇f(x_k) を解く
150     """
151     x = x0.copy()
152     history_x = [x.copy()]
153     history_f = [float(evalf(x))]
154
155     for k in range(max_iter):
156         g = evalg(x)
157         norm_g = np.linalg.norm(g)

```

```

158         if norm_g <= eps:
159             if return_history:
160                 hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(
161                     history_f))
162                 return x, evalf(x), k, hist
163             return x, evalf(x), k
164
165         H = evalh(x)
166         # ニュートン方向を解く: H d = -g
167         try:
168             dk = np.linalg.solve(H, -g)
169         except np.linalg.LinAlgError:
170             break
171
172         tk = backtracking(x, dk, evalf, evalg, t_init=t_init, rho=rho, xi=xi)
173         x = x + tk * dk
174
175         history_x.append(x.copy())
176         history_f.append(float(evalf(x)))
177
178     if return_history:
179         hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(history_f))
180         return x, evalf(x), max_iter, hist
181     return x, evalf(x), max_iter
182
183 # =====
184 # メイン
185 # =====
186
187
188 def main():
189     # 共通パラメータ (課題文指定)
190     xi = 1e-4
191     rho = 0.5
192     t_init = 1.0
193     x0 = np.array([1.0, 1.0]) # 初期点 (1, 1)^T
194
195     print("初期点 x0 =", x0)
196
197     # (a) 最急降下法
198     print("\n=== (a) バックトラック法付き最急降下法 ===")
199     x_sd, f_sd, k_sd, hist_sd = steepest_descent_bt(
200         x0,
201         evalf,
202         evalg,
203         eps=1e-6,
204         max_iter=1000,
205         xi=xi,
206         rho=rho,
207         t_init=t_init,
208         return_history=True,
209     )
210     print("最適解近似 x* ?", x_sd)
211     print("最適値近似 f(x*) ?", f_sd)
212     print("反復回数 =", k_sd)
213
214     # (b) ニュートン法

```



```

215 print("\n=== (b) バックトラック法付きニュートン法 ===")
216 x_nt, f_nt, k_nt, hist_nt = newton_bt(
217     x0,
218     evalf,
219     evalg,
220     evalh,
221     eps=1e-6,
222     max_iter=1000,
223     xi=xi,
224     rho=rho,
225     t_init=t_init,
226     return_history=True,
227 )
228 print("最適解近似 x* ?", x_nt)
229 print("最適値近似 f(x*) ?", f_nt)
230 print("反復回数 =", k_nt)
231
232 # 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
233 try:
234     import matplotlib.pyplot as plt
235 except ImportError:
236     print("matplotlib が見つからないため、プロットをスキップします。")
237     return
238
239 all_points = np.vstack([hist_sd.x, hist_nt.x])
240 x0_min, x1_min = np.min(all_points, axis=0)
241 x0_max, x1_max = np.max(all_points, axis=0)
242
243 margin0 = 0.2 * (x0_max - x0_min) if x0_max > x0_min else 1.0
244 margin1 = 0.2 * (x1_max - x1_min) if x1_max > x1_min else 1.0
245 x0_left, x0_right = x0_min - margin0, x0_max + margin0
246 x1_bottom, x1_top = x1_min - margin1, x1_max + margin1
247
248 n = 250
249 grid_x0 = np.linspace(x0_left, x0_right, n)
250 grid_x1 = np.linspace(x1_bottom, x1_top, n)
251 X0, X1 = np.meshgrid(grid_x0, grid_x1)
252 F = X0**2 + np.exp(X0) + X1**4 + X1**2 - 2.0 * X0 * X1 + 3.0
253
254 plt.figure(figsize=(7, 6))
255 cf = plt.contourf(X0, X1, F, levels=60, cmap="viridis")
256 plt.contour(X0, X1, F, levels=20, colors="k", linewidths=0.4, alpha=0.35)
257 cbar = plt.colorbar(cf)
258 cbar.set_label("f(x0, x1)")
259
260 plt.plot(hist_sd.x[:, 0], hist_sd.x[:, 1], "-o", markersize=3.5, linewidth=1.2,
261         label="Steepest descent")
262 plt.plot(hist_nt.x[:, 0], hist_nt.x[:, 1], "-s", markersize=3.5, linewidth=1.2,
263         label="Newton")
264
265 plt.scatter([hist_sd.x[0, 0]], [hist_sd.x[0, 1]], s=80, c="white", edgecolors="
266             black", zorder=4)
267 plt.scatter([hist_nt.x[0, 0]], [hist_nt.x[0, 1]], s=80, c="white", edgecolors="
268             black", zorder=4)
269 plt.scatter([hist_sd.x[-1, 0]], [hist_sd.x[-1, 1]], s=110, marker="*", c="white",
270             edgecolors="black", zorder=5)
271 plt.scatter([hist_nt.x[-1, 0]], [hist_nt.x[-1, 1]], s=110, marker="*", c="white",
272             edgecolors="black", zorder=5)

```

```

267
268     plt.xlabel("x0")
269     plt.ylabel("x1")
270     plt.title("Optimization trajectories on x0-x1 plane")
271     plt.grid(True, linestyle="--", alpha=0.3)
272     plt.legend()
273
274     out_path = Path(__file__).with_name("task18.png")
275     plt.tight_layout()
276     plt.savefig(out_path, dpi=200)
277     print(f"プロットを保存しました: {out_path}")
278
279     import matplotlib
280
281     if "agg" not in matplotlib.get_backend().lower():
282         plt.show()
283
284
285 if __name__ == "__main__":
286     main()

```

A.5 課題 19 のコード

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from __future__ import annotations
5
6  from dataclasses import dataclass
7  from pathlib import Path
8
9  import numpy as np
10
11  # =====
12  # 課題19の設定
13  #  $f(x) = \sum_{i=0}^2 f_i(x)^2$ 
14  #  $f_i(x) = y_i - x_0 (1 - x_1^{i+1})$ 
15  #  $y_0=1.5, y_1=2.25, y_2=2.625$ 
16  # =====
17
18
19 Y_VALUES = np.array([1.5, 2.25, 2.625], dtype=float)
20
21
22 def fi_and_derivatives(x):
23     """
24      $f_i(x)$ , grad  $f_i(x)$ , Hessian  $f_i(x)$  をまとめて返す
25     """
26     x0, x1 = x[0], x[1]
27     fi_list = []
28     gi_list = []
29     hi_list = []
30
31     for i, yi in enumerate(Y_VALUES):
32         a = i + 1
33         x1a = x1**a

```

```

34     fi = yi - x0 + x0 * x1a
35
36     # grad f_i
37     dfi_dx0 = -1.0 + x1a
38     dfi_dx1 = x0 * a * (x1 ** (a - 1))
39     gi = np.array([dfi_dx0, dfi_dx1], dtype=float)
40
41     # Hessian f_i
42     d2fi_dx0dx0 = 0.0
43     d2fi_dx0dx1 = a * (x1 ** (a - 1))
44     if a == 1:
45         d2fi_dx1dx1 = 0.0
46     else:
47         d2fi_dx1dx1 = x0 * a * (a - 1) * (x1 ** (a - 2))
48     hi = np.array(
49         [[d2fi_dx0dx0, d2fi_dx0dx1], [d2fi_dx0dx1, d2fi_dx1dx1]],
50         dtype=float,
51     )
52
53     fi_list.append(fi)
54     gi_list.append(gi)
55     hi_list.append(hi)
56
57     return fi_list, gi_list, hi_list
58
59
60 def evalf(x):
61     """
62     目的関数  $f(x) = \sum f_i(x)^2$ 
63     """
64     fi_list, _, _ = fi_and_derivatives(x)
65     return float(np.sum(np.square(fi_list)))
66
67
68 def evalg(x):
69     """
70     勾配ベクトル  $\nabla f(x) = 2 * \sum f_i(x) * \nabla f_i(x)$ 
71     """
72     fi_list, gi_list, _ = fi_and_derivatives(x)
73     grad = np.zeros(2, dtype=float)
74     for fi, gi in zip(fi_list, gi_list):
75         grad += 2.0 * fi * gi
76     return grad
77
78
79 def evalh(x):
80     """
81     ヘッセ行列  $\nabla^2 f(x)$ 
82      $= 2 * \sum ( f_i(x) * \nabla^2 f_i(x) + \nabla f_i(x) \nabla f_i(x)^T )$ 
83     """
84     fi_list, gi_list, hi_list = fi_and_derivatives(x)
85     hess = np.zeros((2, 2), dtype=float)
86     for fi, gi, hi in zip(fi_list, gi_list, hi_list):
87         hess += 2.0 * (fi * hi + np.outer(gi, gi))
88     return hess
89
90
91 @dataclass(frozen=True)

```

```

92 class IterationHistory2D:
93     x: np.ndarray # shape (n, 2)
94     f: np.ndarray # shape (n,)
95
96
97 # =====
98 # バックトラック法（共通で使う）
99 # =====
100
101
102 def backtracking(xk, dk, evalf, evalg, t_init=1.0, rho=0.5, xi=1e-4):
103     """
104     バックトラック法（アルミホ条件）
105     """
106     t = t_init
107     fk = evalf(xk)
108     gk = evalg(xk)
109     while True:
110         x_new = xk + t * dk
111         if evalf(x_new) <= fk + xi * t * np.dot(dk, gk):
112             break
113         t *= rho
114     return t
115
116
117 # =====
118 # (a) バックトラック法付き最急降下法
119 # =====
120
121
122 def steepest_descent_bt(
123     x0,
124     evalf,
125     evalg,
126     eps=1e-6,
127     max_iter=1000,
128     xi=1e-4,
129     rho=0.5,
130     t_init=1.0,
131     *,
132     return_history=False,
133 ):
134     x = x0.copy()
135     history_x = [x.copy()]
136     history_f = [float(evalf(x))]
137
138     for k in range(max_iter):
139         g = evalg(x)
140         if np.linalg.norm(g) <= eps:
141             if return_history:
142                 hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(
143                     history_f))
144                 return x, evalf(x), k, hist
145             return x, evalf(x), k
146
147         dk = -g
148         tk = backtracking(x, dk, evalf, evalg, t_init=t_init, rho=rho, xi=xi)
149         x = x + tk * dk

```

```

149         history_x.append(x.copy())
150         history_f.append(float(evalf(x)))
151
152     if return_history:
153         hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(history_f))
154         return x, evalf(x), max_iter, hist
155     return x, evalf(x), max_iter
156
157
158
159 # =====
160 # (b) バックトラック法付きニュートン法
161 # =====
162
163
164 def newton_bt(
165     x0,
166     evalf,
167     evalg,
168     evalh,
169     eps=1e-6,
170     max_iter=1000,
171     xi=1e-4,
172     rho=0.5,
173     t_init=1.0,
174     *,
175     return_history=False,
176 ):
177     x = x0.copy()
178     history_x = [x.copy()]
179     history_f = [float(evalf(x))]
180
181     for k in range(max_iter):
182         g = evalg(x)
183         if np.linalg.norm(g) <= eps:
184             if return_history:
185                 hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(
186                     history_f))
187                 return x, evalf(x), k, hist
188             return x, evalf(x), k
189
190         H = evalh(x)
191         try:
192             dk = np.linalg.solve(H, -g)
193         except np.linalg.LinAlgError:
194             break
195
196         tk = backtracking(x, dk, evalf, evalg, t_init=t_init, rho=rho, xi=xi)
197         x = x + tk * dk
198
199         history_x.append(x.copy())
200         history_f.append(float(evalf(x)))
201
202     if return_history:
203         hist = IterationHistory2D(np.stack(history_x, axis=0), np.array(history_f))
204         return x, evalf(x), max_iter, hist
205     return x, evalf(x), max_iter

```

```

206
207 # =====
208 # メイン
209 # =====
210
211
212 def main():
213     xi = 1e-4
214     rho = 0.5
215     t_init = 1.0
216     x0 = np.array([2.0, 0.0])
217
218     print("初期点 x0 =", x0)
219
220     print("\n=== (a) バックトラック法付き最急降下法 ===")
221     x_sd, f_sd, k_sd, hist_sd = steepest_descent_bt(
222         x0,
223         evalf,
224         evalg,
225         eps=1e-6,
226         max_iter=1000,
227         xi=xi,
228         rho=rho,
229         t_init=t_init,
230         return_history=True,
231     )
232     print("最適解近似 x* ?", x_sd)
233     print("最適値近似 f(x*) ?", f_sd)
234     print("反復回数 =", k_sd)
235
236     print("\n=== (b) バックトラック法付きニュートン法 ===")
237     x_nt, f_nt, k_nt, hist_nt = newton_bt(
238         x0,
239         evalf,
240         evalg,
241         evalh,
242         eps=1e-6,
243         max_iter=1000,
244         xi=xi,
245         rho=rho,
246         t_init=t_init,
247         return_history=True,
248     )
249     print("最適解近似 x* ?", x_nt)
250     print("最適値近似 f(x*) ?", f_nt)
251     print("反復回数 =", k_nt)
252
253     # プロット (任意)
254     try:
255         import matplotlib.pyplot as plt
256     except ImportError:
257         print("matplotlib が見つからないため、プロットをスキップします。")
258         return
259
260     all_points = np.vstack([hist_sd.x, hist_nt.x])
261     x0_min, x1_min = np.min(all_points, axis=0)
262     x0_max, x1_max = np.max(all_points, axis=0)
263     margin0 = 0.2 * (x0_max - x0_min) if x0_max > x0_min else 1.0

```

```

264     margin1 = 0.2 * (x1_max - x1_min) if x1_max > x1_min else 1.0
265
266     x0_left, x0_right = x0_min - margin0, x0_max + margin0
267     x1_bottom, x1_top = x1_min - margin1, x1_max + margin1
268
269     n = 250
270     grid_x0 = np.linspace(x0_left, x0_right, n)
271     grid_x1 = np.linspace(x1_bottom, x1_top, n)
272     X0, X1 = np.meshgrid(grid_x0, grid_x1)
273
274     F = np.zeros_like(X0)
275     for i, yi in enumerate(Y_VALUES):
276         a = i + 1
277         F += (yi - X0 + X0 * (X1**a)) ** 2
278
279     plt.figure(figsize=(7, 6))
280     cf = plt.contourf(X0, X1, F, levels=60, cmap="viridis")
281     plt.contour(X0, X1, F, levels=20, colors="k", linewidths=0.4, alpha=0.35)
282     cbar = plt.colorbar(cf)
283     cbar.set_label("f(x0, x1)")
284
285     plt.plot(hist_sd.x[:, 0], hist_sd.x[:, 1], "-o", markersize=3.5, linewidth=1.2,
286             label="Steepest descent")
287     plt.plot(hist_nt.x[:, 0], hist_nt.x[:, 1], "-s", markersize=3.5, linewidth=1.2,
288             label="Newton")
289
290     plt.scatter([hist_sd.x[0, 0]], [hist_sd.x[0, 1]], s=80, c="white", edgecolors="
291                 black", zorder=4)
292     plt.scatter([hist_nt.x[0, 0]], [hist_nt.x[0, 1]], s=80, c="white", edgecolors="
293                 black", zorder=4)
294     plt.scatter([hist_sd.x[-1, 0]], [hist_sd.x[-1, 1]], s=110, marker="*", c="white",
295                 edgecolors="black", zorder=5)
296     plt.scatter([hist_nt.x[-1, 0]], [hist_nt.x[-1, 1]], s=110, marker="*", c="white",
297                 edgecolors="black", zorder=5)
298
299     plt.xlabel("x0")
300     plt.ylabel("x1")
301     plt.title("Optimization trajectories on x0-x1 plane")
302     plt.grid(True, linestyle="--", alpha=0.3)
303     plt.legend()
304
305     out_path = Path(__file__).with_name("task19.png")
306     plt.tight_layout()
307     plt.savefig(out_path, dpi=200)
308     print(f"プロットを保存しました: {out_path}")
309
310     import matplotlib
311
312     if "agg" not in matplotlib.get_backend().lower():
313         plt.show()
314
315 if __name__ == "__main__":
316     main()

```

A.6 追加課題のコード

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import numpy as np
5
6
7  def primal_dual_ipm_qp(Q, c, A, b, *, max_iter=80, tol=1e-12, sigma=0.1, tau=0.99):
8      """
9      Primal-Dual Interior-Point (Newton) for convex QP:
10         minimize    1/2 x^T Q x + c^T x
11         subject to  A x <= b    (b is scalar or length-m)
12
13         Slack: y >= 0 such that  A x + y - b = 0
14         Dual:   λ >= 0
15         Perturbed complementarity: y_i λ_i = sigma * mu,
16         mu = (y^T λ)/m
17     """
18     Q = np.asarray(Q, dtype=float)
19     c = np.asarray(c, dtype=float).reshape(-1)
20     A = np.asarray(A, dtype=float)
21     b = np.asarray(b, dtype=float).reshape(-1)
22
23     n = Q.shape[0]
24     m = A.shape[0]
25
26     # 半正定値チェック (数値誤差許容)
27     Qsym = 0.5 * (Q + Q.T)
28     if np.linalg.eigvalsh(Qsym).min() < -1e-10:
29         raise ValueError("Q must be positive semidefinite.")
30
31     # 初期点 (内点)
32     x = np.zeros(n)
33     y = np.ones(m)
34     lam = np.ones(m)
35     e = np.ones(m)
36
37     for k in range(max_iter):
38         # 残差
39         rd = Qsym @ x + c + A.T @ lam           # dual residual
40         rp = A @ x + y - b                       # primal residual
41         mu = (y @ lam) / m                       # complementarity
42         rc = y * lam - sigma * mu * e           # perturbed complementarity
43
44         # 収束判定
45         if max(np.linalg.norm(rd), np.linalg.norm(rp), mu) < tol:
46             return x, y, lam, {"iter": k, "rd": np.linalg.norm(rd),
47                                "rp": np.linalg.norm(rp), "mu": mu}
48
49         # Newton 方程式
50         # [ Q   0   A^T ] [dx] = -[rd]
51         # [ A   I   0   ] [dy]  -[rp]
52         # [ 0   λ   Y   ] [dλ]  -[rc]
53         K = np.block([
54             [Qsym,
55              np.zeros((n, m)),
56              A.T],
```



```

55         [A,                                np.eye(m),                np.zeros((m, m))],
56         [np.zeros((m, n)),                np.diag(lam),                np.diag(y)]
57     ])
58     rhs = -np.concatenate([rd, rp, rc])
59
60     delta = np.linalg.solve(K, rhs)
61     dx = delta[:n]
62     dy = delta[n:n+m]
63     dlam = delta[n+m:]
64
65     # 正性を保つステップ長
66     alpha = 1.0
67     if np.any(dy < 0):
68         alpha = min(alpha, tau * np.min(-y[dy < 0] / dy[dy < 0]))
69     if np.any(dlam < 0):
70         alpha = min(alpha, tau * np.min(-lam[dlam < 0] / dlam[dlam < 0]))
71
72     # 更新
73     x += alpha * dx
74     y += alpha * dy
75     lam += alpha * dlam
76
77     # 数値安全
78     y = np.maximum(y, 1e-18)
79     lam = np.maximum(lam, 1e-18)
80
81     return x, y, lam, {"iter": max_iter, "rd": np.linalg.norm(rd),
82                        "rp": np.linalg.norm(rp), "mu": mu}
83
84
85 if __name__ == "__main__":
86     # 課題4: パラメータ
87     Q = np.array([[2.0, 0.0],
88                  [0.0, 1.0]])      # 半正定値行列
89     c = -np.array([1.0, 1.0])
90
91     # b はスカラー 0 と解釈
92     A = np.array([[1.0, 1.0]])      # 1 本の制約
93     b = 0.0
94
95     x_star, y_star, lam_star, info = primal_dual_ipm_qp(Q, c, A, b)
96
97     print("x* =", x_star)
98     print("slack y =", y_star)
99     print("dual λ =", lam_star)
100    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.