令和四年度後期 数理工学実験 課題レポート

連続最適化

情報学科 2 回生 平田蓮

学生番号: 1029342830

実験日: 11 月 29, 12 月 5, 6, 12 日

実験場所: 京都大学工学部総合校舎数理計算機室

12 月 19 日 提出

目次

1 目的

本実験では連続な関数についての最適化手法を学び、それに関する演習を Python を用いて行う。2.1 節では関数の零点探索、2.2 では 1 変数関数の停留点探索を行い、2.3, 2.4, 2.5 節では 2 変数関数における最適化問題解く。

2 課題

2.1 課題 1

まず、本課題で用いるアルゴリズム、二分法とニュートン法について説明する。これらはどちらも連続な 1 変数関数 $f : \mathbb{R} \to \mathbb{R}$ における零点 ($x \in \mathbb{R}$ s.t. $f(x) = 0$) を探索するアルゴリズムである。

2.1.1 二分法

初期値として、$f(a) < 0, f(b) \geq 0$ を満たす $a, b \in \mathbb{R}$ を選ぶ。f は連続なので、零点が a と b の間に存在する。よって、これらの中点 $x = \dfrac{a+b}{2}$ と a, b を置き換えながら範囲を狭めていくことで零点を探索する。このアルゴリズムは次のように書ける。

1. 探索範囲の境界 a, b について、中点 $x = \dfrac{a+b}{2}$ を計算
2. $f(x) < 0$ なら $a \leftarrow x$、そうでないなら $b \leftarrow x$
3. $|a - b|$ が一定の値を下回るまで繰り返す

2.1.2 ニュートン法

適当な初期値 x_k を取る。$f(x)$ を x_k の周りで一次関数で近似し、それと x 軸の交点を新しい x_k とする。このアルゴリズムは次のように書ける。

1. x_k について、$y = f'(x_k)(x - x_k) + f(x_k)$ を考える
2. $y = 0$ となる $x = x_k - \dfrac{f(x_k)}{f'(x_k)}$ について、$x_k \leftarrow x$
3. $|f(x_k)|$ が一定の値を下回るまで繰り返す

以下では $f = x^3 + 2x^2 - 5x - 6$ について考える。

2.1.3 課題 (a)

f を $-10 \leq x, y \leq 10$ の範囲で描画する。作成したソースコードをリスト 1, 2 に示す。

コード 1 f の実装

```
class Function:
    def __call__(self, x:float) -> float:
        return x**3 + 2 * x**2 - 5 * x - 6

    def __str__(self) -> str:
        return '$f␣=␣x^3␣+␣2x^2␣-␣5x␣-␣6$'

```

```
8     def df(self, x:float): # 微分
9         return 3 * x**2 + 4 * x - 5
```

コード 2　f を描画するソースコード

```
1  import numpy
2  from matplotlib import pyplot
3
4
5  pyplot.rcParams["text.usetex"] = True
6  pyplot.rcParams["font.size"] = 12
7
8
9  def plot(
10     f:Function,
11     xlim:tuple[int],
12     ylim:tuple[int],
13     save_path:str,
14     n=10**7
15 ) -> None:
16     x = numpy.linspace(*xlim, n)
17     pyplot.plot(x, f(x), label=str(f))
18
19     pyplot.xlim(xlim)
20     pyplot.ylim(ylim)
21
22     pyplot.xlabel('$x$')
23     pyplot.ylabel('$f$', rotation=0)
24
25     pyplot.grid()
26     pyplot.legend()
27     pyplot.savefig(save_path)
28
29
30 if __name__ == '__main__':
31     f = Function()
32     plot(f, xlim=(-10, 10), ylim=(-10, 10), save_path='a.pdf')
```

描画した結果を図 1 に示す。

2.1.4　課題 (b)

二分法を用いて f の零点を全て求める。図 1 より、f の零点は-5 と-2.5 の間、-2.5 と 0 の間、0 と 2.5 の間にそれぞれあることがわかるので、これらを初期値に用いる。作成したソースコードをリスト 3 に示す。今回は、$|a-b| \leq 10^{-7}$ になるまで反復する。

コード 3　二分法の実装

```
1  def get_zero_points(
2      algorithm:Callable,
3      f:Function,
4      starts:list
5  ) -> list[float]:
6      return [algorithm(f, start) for start in starts]
7
8  def get_zero_point_with_bisection(
```

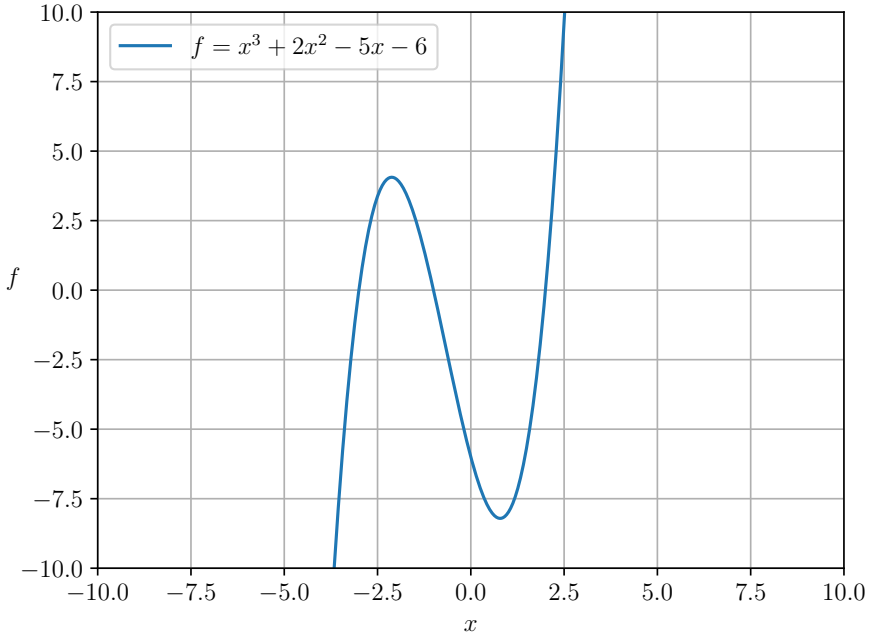

図 1　f の描画結果

```
    f:Function, start:tuple[float],
    terminate_threshold:float=10**-7
) -> list[float]:
    left:float
    right:float
    left, right = start

    assert f(left) * f(right) <= 0

    mid:float = None
    mids:list[float] = []
    while abs(left - right) > terminate_threshold:
        mid = (left + right) / 2
        mids.append(mid)
        if f(mid) * f(left) > 0:
            left = mid
        else:
            right = mid

    mids.append(mid)
    return mids


if __name__ == '__main__':
    f:Function = Function()
    zero_points:list[float] = get_zero_points(get_zero_point_with_bisection, f, starts=[
        (-5, -2.5),
        (-2.5, 0),
```

```
37          (0, 2.5)
38      ])
```

`get_zero_point_with_bisection()` は、最終的な零点だけでなく、反復時の $\frac{a+b}{2}$ の遷移を返す。求めた零点を図 2 に示す。

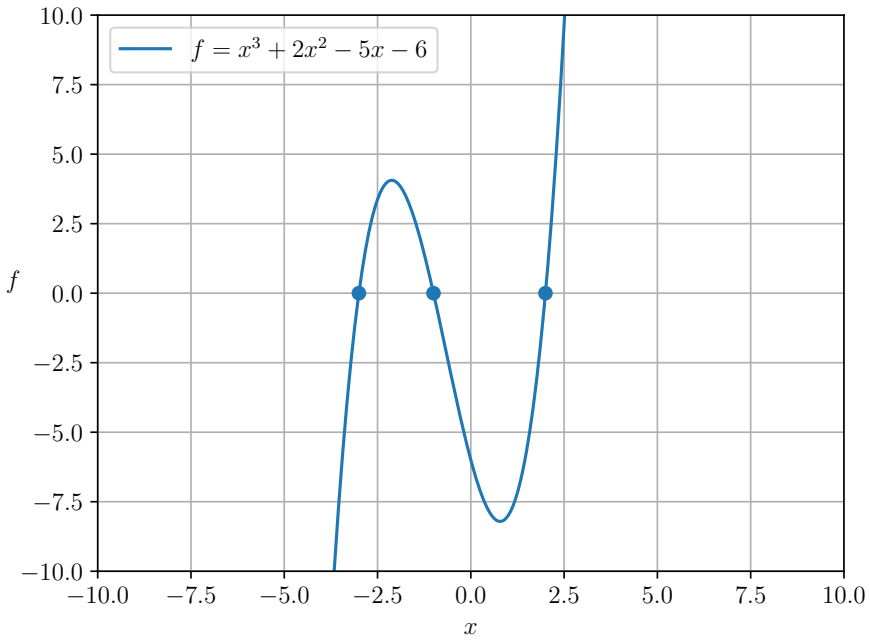

図 2　f の零点

零点は $x = -3, -1, 2$ であった。図より実際にこれらが零点となっていることがわかる。

2.1.5　課題 (c)

ニュートン法を用いて f の零点を用いる。ニュートン法の初期値は、零点の十分近くを取らないと収束しないことが知られているため、今回は $x = -4, 0, 4$ を用い、$|f(x)| \leq 10^{-7}$ になるまで反復した。作成したソースコードをリスト 4 に示す。

コード 4　ニュートン法の実装

```
1  def get_zero_point_with_newton(
2      f:Function,
3      start:float,
4      terminate_threshold:float=10**-7
5  ) -> list[float]:
6      x:float = start
7      xs:list[float] = []
8      while abs(f(x)) > terminate_threshold:
9          xs.append(x)
10         x -= f(x) / f.df(x)
11
```

```
12      xs.append(x)
13      return xs
14
15
16  if __name__ == '__main__':
17      f:Function = Function()
18      zero_points:list[float] = get_zero_points(get_zero_point_with_newton, f, starts=[
19          -4,
20          0,
21          4
22      ])
```

二分法と同様の結果が得られたため、結果は省略する。

2.1.6 考察

零点の理論解 x_0 と、探索による値 x の誤差を $E = |x_0 - x|$ と定義する。各種法における反復回数 k 回時点の x について、E をプロットした図を図 3, 4, 5 に示す。

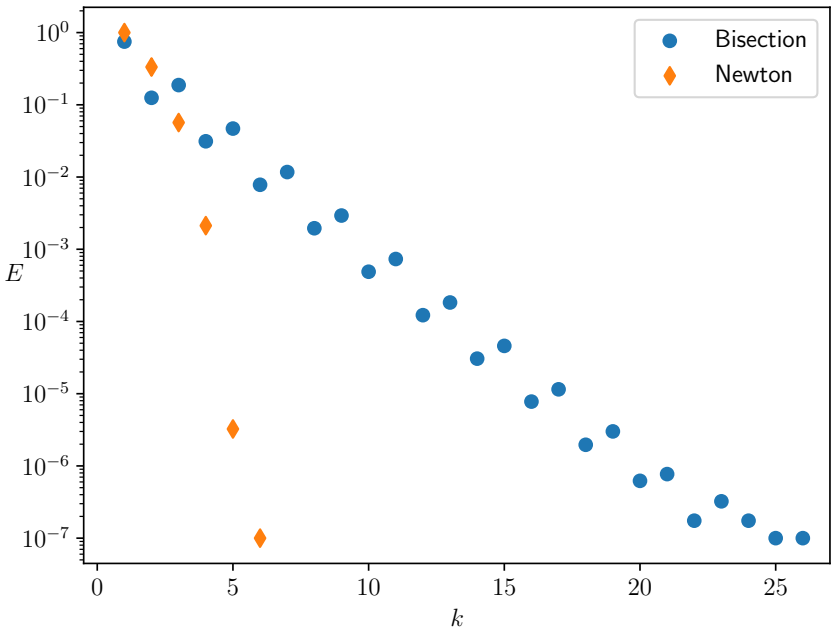

図 3　零点 $x = -3$ と探索解の誤差。Bisection が二分法、Newton がニュートン法を示す。

三つの図より、ニュートン法の方が二分法と比べて早く結果が収束していることがわかる。

2.2 課題 2

はじめに、本課題で用いるアルゴリズム、最急降下法とニュートン法の拡張について説明する。

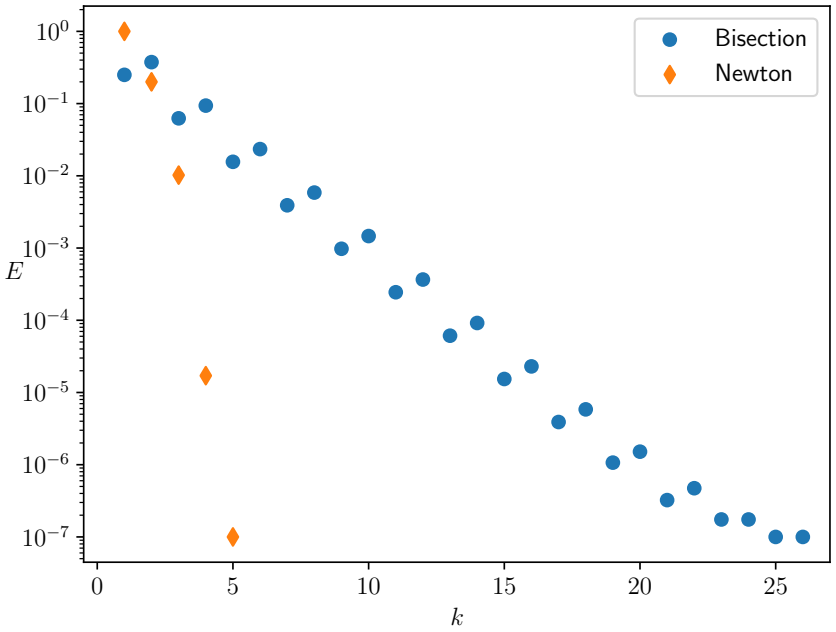

図 4　零点 $x = -1$ と探索解の誤差。Bisection が二分法、Newton がニュートン法を示す。

2.2.1　最急降下法

ニュートン法のように、$x \leftarrow x + td_k$ と x を更新しながら値を探索するアルゴリズムを降下法と呼ぶ。ここで、t はステップ幅、d_k は変化方向である。最急降下法は次に示す最急降下方向を用いることで関数の停留点を求めることができる。

$$\text{最急降下方向 } d_k = -\nabla f(x)$$

2.2.2　ニュートン法の拡張

次に示すニュートン方向を用いた降下法でも、関数の停留点を求めることができる。

$$\text{ニュートン方向 } d_k = -\nabla^2 f(x)^{-1} \nabla f(x)$$

ここで、$\nabla^2 f(x)$ は f のヘッセ行列である。

以下では関数 $f = \dfrac{x^3}{3} - x^2 - 3x + \dfrac{5}{3}$ を考える。

2.2.3　課題 (a)

最急降下法を用いて f の停留点を求める。初期値は $x = 0.5$, k ステップ目 $(k = 0, 1, \cdots)$ においてステップ幅は $t = \dfrac{1}{k+1}$ とする。反復の終了条件は $|\nabla f(x)| \leq 10^{-7}$ とした。

作成したソースコードをリスト 5, 6 に示す。

コード 5　f の実装

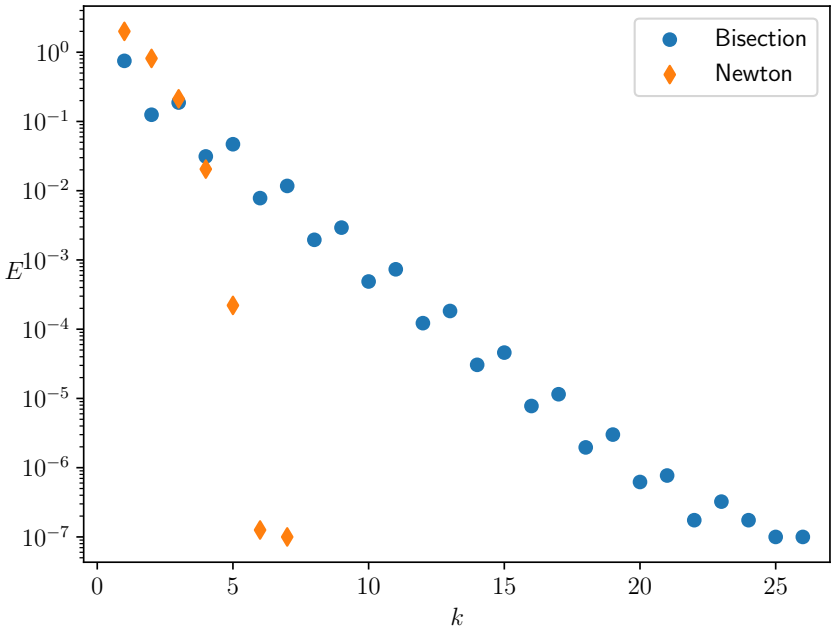

図 5　零点 $x = 2$ と探索解の誤差。Bisection が二分法、Newton がニュートン法を示す。

```
class Function:
    def __call__(self, x:float) -> float:
        return x**3 / 3 - x**2 - 3 * x + 5 / 3

    def __str__(self) -> str:
        return r'$\displaystyle\frac{1}{3}x^3␣-␣x^2␣-␣3x␣+␣\frac{5}{3}$'

    def df(self, x:float): # 一階微分
        return x**2 - 2 * x - 3

    def d2f(self, x:float): # 二階微分
        return 2 * x - 2
```

コード 6　最急降下法の実装

```
def get_critical_point_with_gradient(
    f:Function,
    start:float,
    terminate_threshold:float=10**-7
) -> list[float]:
    x:float = start
    k = 0

    xs:list[float] = [x]
    last_f:float = float('inf')
    while abs(last_f - f(x)) > terminate_threshold:
        last_f = f(x)
        x -= f.df(x) * 1 / (k + 1)

```

```
        xs.append(x)
        k += 1

    return xs


if __name__ == '__main__':
    f:Function = Function()
    critical_point:list[float] = get_critical_point_with_gradient(f, 1/2)
```

求めた停留点を図 6 に示す。

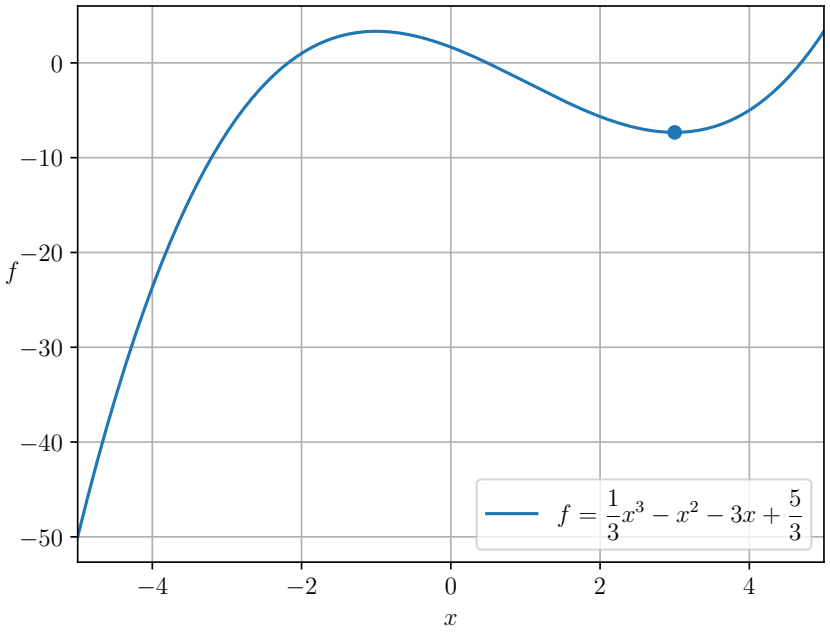

図 6　f の停留点

停留点は $x = 3$ と求まり、これが正しいことが図よりわかる。

2.2.4　課題 (b)

ニュートン法を用いて f の停留点を求める。初期値は $x = 5$, ステップ幅は $t = k$ とする。終了条件は $|\nabla^2 f(x)^{-1} \nabla f(x)| \leq 10^{-7}$ とした。

作成したソースコードをリスト 7 に示す。

コード 7　ニュートン法の実装

```
def get_critical_point_with_newton(
    f:Function,
    start:float,
    terminate_threshold:float=10**-7
) -> list[float]:
    x:float = start
```

```
    k = 0

    xs:list[float] = [x]
    last_f:float = float('inf')
    while abs(last_f - f(x)) > terminate_threshold:
        last_f = f(x)
        x -= f.df(x) / f.d2f(x)

        xs.append(x)
        k += 1

    return xs


if __name__ == '__main__':
    f:Function = Function()
    critical_point:list[float] = get_critical_point_with_newton(f, 5)
```

最急降下法と同様の結果が得られたため、結果は省略する。

2.3 課題 3

次の最適化問題を考える。

$$\text{minimize } f(\boldsymbol{x}) = x_0^2 + e^{x_0} + x_1^4 + x_1^2 - 2x_0x_1 + 3$$
$$\boldsymbol{x} = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \in \mathbb{R}^2$$

$\nabla f(x), \nabla^2 f(x)$ は、次の形で表せる。

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

$f(x)$, 勾配 $\nabla f(x)$, ヘッセ行列 $\nabla^2 f(x)$ を出力するソースコードをリスト 8 に示す。

コード 8　$f, \nabla f, \nabla^2 f$ の実装

```
import numpy

class Function:
    def __call__(self, x:tuple[float]) -> float: # f
        assert 2 == len(x)
        return x[0]**2 + numpy.exp(x[0]) + x[1]**4 + x[1]**2 - 2 * x[0] * x[1] + 3

    def __str__(self) -> str:
        return r'$\displaystyle\frac{1}{3}x^3␣-␣x^2␣-␣3x␣+␣\frac{5}{3}$'

    def df(self, x:tuple[float]) -> numpy.array: # 勾配ベクトル
        assert 2 == len(x)
        return numpy.array([
            2 * x[0] + numpy.exp(x[0]) - 2 * x[1],
            4 * x[1]**3 + 2 * x[1] - 2 * x[0]
        ])

```

```
18      def d2f(self, x:tuple[float]) -> numpy.matrix: # ヘッセ行列
19          assert 2 == len(x)
20          return numpy.matrix([
21              [2 + numpy.exp(x[0]), -2],
22              [-2, 12 * x[1]**2 + 2]
23          ])
```

2.4 課題 4

反復時のステップ幅 t_k を適切に計算するアルゴリズム、バックトラック法を説明する。

2.4.1 バックトラック法

バックトラック法はステップ幅 t を用いて更新後の $f(x_k + td_k)$ が $f(x_k) + \xi t\langle d_k, \nabla f(x_k)\rangle$ よりも小さくなるようにステップ幅を選ぶ方法である。これは、次のアルゴリズムで表せる。

1. 定数 $\xi, \rho \in (0, 1)$, 初期ステップ幅 t を決める
2. 次式を満たす間、$t \leftarrow \rho t$ とする
   $f(x_k + td_k) \leq f(x_k) + \xi t\langle d_k, \nabla f(x_k)\rangle$

以下では課題 3 と同様の最適化問題を考える。

2.4.2 課題 (a)

バックトラック法を適用した最急降下法を用いて最適解、最適値、反復回数を求める。初期値は $\boldsymbol{x} = \begin{pmatrix}1\\1\end{pmatrix}$, バックトラック法における定数と初期ステップ幅は $\xi = 10^{-4}, \rho = 0.5, t = 1$ とし、終了条件は 2.2.3 節と同様とした。

作成したソースコードをリスト 9, 10 に示す。

コード 9　バックトラック法の実装

```
1  import numpy
2
3  def backtrack(
4      f:Function,
5      x:numpy.array,
6      d:numpy.array,
7      zeta:float,
8      rho:float,
9      t:float
10 ) -> float:
11     while f(x + t * d) > f(x) + zeta * t * numpy.dot(d, f.df(x)):
12         t *= rho
13
14     return t
```

コード 10　最急降下法の実装

```
1  import numpy
2
3  def get_critical_point_with_gradient(
```

```
    f:Function,
    start:numpy.array,
    terminate_threshold:float=10**-7
) -> list[numpy.array]:
    x:numpy.array = start
    k = 0

    xs:list[tuple[float]] = [x]
    last_f:float = float('inf')
    while abs(last_f - f(x)) > terminate_threshold:
        d:numpy.array = -f.df(x)
        t:float = backtrack(f, x, d, 10**-4, 0.5, 1)

        last_f = f(x)
        x += d * t

        xs.append(x)
        k += 1

    return xs


if __name__ == '__main__':
    f:Function = Function()
    critical_point:list[float] = get_critical_point_with_gradient(f, numpy.array([1.0,
        1.0]))
```

最適解は $\boldsymbol{x} = \begin{pmatrix} -0.7333 \\ -0.4932 \end{pmatrix}$, 最適値は $f = 3.5972$ となり、反復回数は 19 回であった。

2.4.3 課題 (b)

バックトラック法を適用したニュートン法を用いて最適解、最適値、反復回数を求める。初期値及び定数は 2.4.2 節と同様とし、終了条件は 2.2.4 節と同様とした。

作成したソースコードをリスト 11 に示す。

コード 11　ニュートン法の実装

```
import numpy

def get_critical_point_with_newton(
    f:Function,
    start:numpy.array,
    terminate_threshold:float=10**-7
) -> list[numpy.array]:
    x:numpy.array = start
    k = 0

    xs:list[tuple[float]] = [x]
    last_f:float = float('inf')
    while abs(last_f - f(x)) > terminate_threshold:
        d:numpy.array = numpy.array(-f.d2f(x).I @ f.df(x))[0]
        t:float = backtrack(f, x, d, 10**-4, 0.5, 1)

        last_f = f(x)
        x += d * t

```

```
        xs.append(x)
        k += 1

    return xs


if __name__ == '__main__':
    f:Function = Function()
    critical_point:list[float] = get_critical_point_with_newton(f, numpy.array([1.0, 1.0]))
```

最適解は $\boldsymbol{x} = \begin{pmatrix} -0.7335 \\ -0.4933 \end{pmatrix}$, 最適値は $f = 3.5971$ となり、反復回数は 7 回であった。

2.4.4 考察

最急降下法のほうが、収束が遅い結果となり、さらに同様の終了条件では最急降下法によって得られた最適値はニュートン法による解よりわずかに大きかった。最急降下法の終了条件を $|\nabla f(x)| \leq 10^{-15}$ とすると、反復回数は 36 回となり、ニュートン法を用いた際と同値の最適解及び最適値を得られた。一方、ニュートン法は終了条件を厳しくすれども、反復回数が増えることはなかった。

2.5 課題 5

次の最適化問題を考える。

$$\text{minimize } f(\boldsymbol{x}) = \sum_{i=0}^{2} f_i(\boldsymbol{x})^2 \quad \left(f_i(\boldsymbol{x}) = y_i - x_0(1 - x_1^{i+1})\right)$$
$$\boldsymbol{x} = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \in \mathbb{R}^2$$

ここで、$y_0 = 1.5, y_1 = 2.25, y_2 = 2.625$ とする。

これを最急降下法とニュートン法で解く。初期値は $\boldsymbol{x} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$ とし、バックトラック法の定数及び終了条件は課題 4 と同様とする。

作成したソースコードをリスト 12 に示す。各アルゴリズムは、課題 4 で実装したものと同様のものを用いた。

コード 12　f の実装

```
import numpy

class Function:
    y:list[float] = [1.5, 2.25, 2.625]

    def f(self, i:int, x:numpy.array) -> float: # fi
        assert 2 == len(x)
        return self.y[i] - x[0] * (1 - x[1] ** (i + 1))

    def dfi(self, i:int, x:tuple[float]) -> numpy.array: # fi gradient
        assert 2 == len(x)
        return numpy.array([
            -1 + x[1] ** (i + 1),
            (i + 1) * x[0] * (x[1] ** i)
        ])
```

```
16
17      def dfi2(self, i:int, x:tuple[float]) -> numpy.matrix: # fi hessian
18          assert 2 == len(x)
19          return numpy.matrix([
20              [self.dfi(i, x)[0] * self.dfi(i, x)[0], self.dfi(i, x)[0] * self.dfi(i, x)[1]],
21              [self.dfi(i, x)[1] * self.dfi(i, x)[0], self.dfi(i, x)[1] * self.dfi(i, x)[1]]
22          ])
23
24      def d2fi(self, i:int, x:tuple[float]) -> numpy.matrix: # fi gradient squared
25          assert 2 == len(x)
26          return numpy.matrix([
27              [0, (i + 1) * (x[1] ** i)],
28              [(i + 1) * (x[1] ** i), 0 if 0 == i else i * (i + 1) * x[0] * (x[1] ** (i - 1))
                    ]
29          ])
30
31      def __call__(self, x:tuple[float]) -> float: # f
32          return sum(self.f(i, x)**2 for i in range(3))
33
34      def df(self, x:numpy.array) -> numpy.array: # 勾配
35          return 2 * sum(self.f(i, x) * self.dfi(i, x) for i in range(3))
36
37      def d2f(self, x:numpy.array) -> numpy.matrix: # ヘッセ行列
38          return 2 * sum(self.f(i, x) * self.d2fi(i, x) + self.dfi2(i, x) for i in range(3))
```

結果として、最急降下法では最適値 $2.549^{-7}, \boldsymbol{x} = \begin{pmatrix} 2.9877 \\ 0.4971 \end{pmatrix}$ が反復回数 170 回で得られ、ニュートン法では最適値 $9.470^{-15}, \boldsymbol{x} = \begin{pmatrix} 3.0000 \\ 0.5000 \end{pmatrix}$ が反復回数 6 回で得られた。この問題の最適値は 0、最適解は $\boldsymbol{x} = \begin{pmatrix} 3 \\ 0.5 \end{pmatrix}$ なので、正しい解が得られていることがわかる。

2.5.1 考察

課題 4 の際と同様に、最急降下法の方が収束が遅く、さらに、最適解と理論解の誤差が大きく現れる結果となった。最急降下法の収束条件を $|\nabla f(x)| \leq 10^{-17}$ とすると、ニュートン法に近い最適値 8.673^{-15} が得られたが、976 回の反復回数を要した。一方、ニュートン法の終了条件を $|\nabla^2 f(x)^{-1} \nabla f(x)| \leq 10^{-8}$ と少しだけ厳しくすると、その反復回数は 1 回だけ増え 7 回となり最適値 1.399^{-27} を得て、大幅に理論値に近づいた。

参考文献

[1] 実験演習ワーキンググループ、“数理工学実験 2022 年度版”、京都大学工学部情報学科数理工学コース (2022)