レポート提出票

科目名: 情報工学実験2

実験テーマ: 実験テーマ1 数理計画法

実施日: 2020年 10月 26日

学籍番号: 4619055

氏名: 辰川力駆

共同実験者:

1　実験の要旨

実験を通して、最急降下法とニュートン法についての理解を深めるとともに、二次元平面にプロットするなどをして議論をする。

2　実験の目的

非線形最適化問題に対し、最急降下法とニュートン法を実装・適用を通し、アルゴリズムの特性を理解する。

3　実験の原理 (理論)

3.1　無制約最小化問題に対する基礎理論

無制約最小化問題とは、n 変数関数 $f : \mathbb{R}^n \to \mathbb{R}$ に対して定義される以下の問題である。

$$\text{Minimize } f(\boldsymbol{x}) \text{ subject to } \boldsymbol{x} = (x_1, x_2, \cdots, x_n)^T \in \mathbb{R}^n$$

3.1.1　諸定義

n 変数関数 $f : \mathbb{R}^n \to \mathbb{R}$ に対し、勾配ベクトル $\nabla f(\boldsymbol{x})$ とヘッセ行列 $\nabla^2 f(\boldsymbol{x})$ はそれぞれ以下のように定義されるベクトルと行列である。

$$\nabla f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}, \ \nabla^2 f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial^2 x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial^2 x_n} \end{pmatrix}$$

勾配ベクトルとヘッセ行列はそれぞれ 1 変数関数 $f(\boldsymbol{x})$ の微分係数 $f'(\boldsymbol{x})$ と 2 階微分係数 $f''(\boldsymbol{x})$ を n 変数関数に拡張したものである。ヘッセ行列は対称行列になることに注意する。勾配ベクトルとヘッセ行列は f のテイラー展開に現れる。具体的には、$\boldsymbol{x} = \boldsymbol{a}$ の周りで 2 次の項まで求めると、以下のようになる。

$$f(\boldsymbol{a} + \boldsymbol{d}) = f(\boldsymbol{a}) + \nabla f(\boldsymbol{a})^T \boldsymbol{d} + \frac{1}{2} \boldsymbol{d}^T \nabla^2 f(\boldsymbol{a}) \boldsymbol{d} + (\text{残差})$$

また、n 次実正方行列 X に対し、

- X が半正定値とは、$X^T = X$ と、$\forall \boldsymbol{d} \in \mathbb{R}^n,\ \boldsymbol{d}^T X \boldsymbol{d} \geq 0$ が成り立つことを言い、
- X が正定値とは、$X^T = X$ と、$\forall \boldsymbol{d} \in \mathbb{R}^n \backslash \{\boldsymbol{0}\},\ \boldsymbol{d}^T X \boldsymbol{d} > 0$ が成り立つことを言う。

さらに、n 次実対称行列 X に対し、「X が半正定値 (正定値)$\Leftrightarrow$ X の固有値がすべて正」が成り立つ。

3.1.2　最適性条件

一般に非線形最適化問題において大域的最適解を求めることは極めて難しく、多くの場合は局所的最適解を求めることを目指す。局所最適解については以下の最適性条件が成り立つ。

定理 (最適性の必要条件) $\boldsymbol{x}^* \in \mathbb{R}^n$ を無制約最小化問題の局所的最適解とする。このとき、つぎが成り立つ。

- f が連続的微分可能 $\Rightarrow \nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$
- f が 2 回連続的微分可能 $\Rightarrow \nabla^2 f(\boldsymbol{x}^*)$ は半正定値行列

1 次の必要条件 $\nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$ を満たす $\boldsymbol{x}^*$ を停留点という。

定理 (2 次の十分条件) 関数 f が 2 回連続的微分可能とする。$\boldsymbol{x}^* \in \mathbb{R}^n$ が以下の条件を満たすならば、$\boldsymbol{x}^* \in \mathbb{R}^n$ は無制約最小化問題の局所的最適解である。

- $\nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$
- $\nabla^2 f(\boldsymbol{x}^*)$ が正定値行列

関数が凸性と呼ばれる「良い」性質を持つ場合には、実は、大域的最適解を求めることが比較的容易になる。関数 $f : \mathbb{R}^n \to \mathbb{R}$ が以下の条件を満たすとき、f は凸関数という。

$$f(\lambda \boldsymbol{x} + (1-\lambda)\boldsymbol{y}) \leq \lambda f(\boldsymbol{x}) + (1-\lambda) f(\boldsymbol{y}),\ \forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n,\ \forall \lambda \in (0,1)$$

f が 2 回連続的微分可能であるとき、以下の事実が成り立つ。

$$f \text{ が凸関数} \Leftrightarrow \ \nabla^2 f(\boldsymbol{x}) \text{ が半正定値行列} \ \ \forall \boldsymbol{x} \in \mathbb{R}^n$$

さらに、凸関数は次の「局所最適性=大域的最適性」を満たす。

$$f \text{ が凸関数のとき、} \boldsymbol{x}^* \text{ が } f \text{ の局所的最適解} \Rightarrow \boldsymbol{x}^* \text{ は } f \text{ の大域的最適解}$$

これにより、微分可能な凸関数に対しては、$\nabla f(\boldsymbol{x}^*) = \boldsymbol{0}$ を満たす $\boldsymbol{x}^*$ を求めれば、その $\boldsymbol{x}^*$ は最適解であることが保証される。このように、大域的最適解が必ず求まる関数のクラスとして、凸関数は重要なクラスである。

3.2 反復法

適当な初期点 $\boldsymbol{x_0} \in \mathbb{R}^n$ からスタートし、以下の更新式で次々と点 $\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots,$ を生成するアルゴリズムを反復法という。

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$$

この式における $\boldsymbol{d}_k$ を探索方向、α_k をステップ幅という。

探索方向としては降下方向になっているものを用いるのが一般的である。具体的には、$\boldsymbol{x} \in \mathbb{R}^n$ における降下方向とは以下の条件を満たす $\boldsymbol{d} \in \mathbb{R}^n$ であり、目的関数値が減少する方向である。

$$\nabla f(\boldsymbol{x})^T \boldsymbol{d} < 0$$

$\nabla f(\boldsymbol{x}) \neq \boldsymbol{0}$ であれば、$-\nabla f(\boldsymbol{x})$ は自明な降下方向である。これを最急降下方向という。

また、ステップ幅は次小節で説明する直線探索を利用して求めることが多い。直線探索を用いた反復法を形式的に書くと以下のようになる。

<u>直線探索を用いた反復法</u>

ステップ 0: 初期点 $\boldsymbol{x}_0$ を選び、$k := 0$ とする

ステップ 1: 停止基準が満たされていれば終了とする

ステップ 2: 探査方向 $\boldsymbol{d}_k$ を定める

ステップ 3: 直線探索を用いてステップ幅 α_k を定める

ステップ 4: $\boldsymbol{x}_{k+1} := \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k,\ k := k + 1$ としてステップ幅 1 に戻る

ステップ 1 の停止基準としては、勾配ベクトルの大きさ $||\nabla f(\boldsymbol{x}_k)||$ が十分小さくなったことや、解の更新幅 $||\boldsymbol{x}_{k+1} - \boldsymbol{x}_k||$ が十分小さくなったことなどに設定する。

3.3 直線探索

点 $\boldsymbol{x}_k \in \mathbb{R}^n$ と $\boldsymbol{x}_k$ における降下方向 $\boldsymbol{d}_k$ が与えられたときに可能ならばステップ幅 α_k を

$$f(\boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k) = \min\{f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) | \alpha > 0\}$$

となるように選びたい。関数 f が凸 2 次関数の場合は正確に α_k が求まるが、そうでなければこの問題は非常に難しい (それ自体が 1 変数の最小化問題)。多くの場合は次のアルミホ基準を満たすように α_k を求める。

定義 (アルミホ基準) $0 < \xi < 1$ を満たす定数 ξ に対し、

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) \le f(\boldsymbol{x}_k) + \xi \alpha \nabla f(\boldsymbol{x}_k)^T \boldsymbol{d}_k$$

<u>アルミホ条件に対する直線探索</u> ($\boldsymbol{x}_k, \boldsymbol{d}_k$ は所与とする)

ステップ 0: パラメータ $0 < \xi < 1, 0 < \tau < 1$ を選び、$\alpha := 1$ とする

ステップ 1: $f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) \le f(\boldsymbol{x}_k) + \xi \alpha \nabla f(\boldsymbol{x}_k)^T \boldsymbol{d}_k$(アルミホ条件) が成り立てば終了する

ステップ 2: $\alpha := \tau \alpha$ としてステップ 1 に戻る

なお、アルミホ条件に $\nabla f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ に関する条件を付加したウルフ条件も実用的に多く用いられている。

3.4 実験で扱うアルゴリズム

3.4.1 最急降下法

最急降下法とは、反復法におけるステップ 2 の探索方向として、$\boldsymbol{x}_k$ における最急降下方向 $-\nabla f(\boldsymbol{x}_k)$ を常に用いる方法である。本実験で用いる最急降下法は以下の通りである。

<u>最急降下法</u>

ステップ 0: 初期点 $\boldsymbol{x}_k$ を選び、$k := 0$ とする。また、$\epsilon := 10^{-8}$ とする

ステップ 1: $||\nabla f(\boldsymbol{x}_{\boldsymbol{k}})|| < \epsilon$ が満たされていれば終了する

ステップ 2: 探索方向を $\boldsymbol{d}_k = -\nabla f(\boldsymbol{x}_k)$ とする

ステップ 3: アルミホ条件による直線探索を用いてステップ幅 α_k を定める

ステップ 4: $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k,\ k := k + 1$ としてステップ 2 に戻る

最急降下法の大域的収束性 関数 $f(\boldsymbol{x})$ に関するいくつかの仮定の下ではウルフ条件を用いた最急降下法は任意の初期点に対して以下の式が成り立つ。

$$\lim_{k \to \infty} ||\nabla f(\boldsymbol{x}_k)|| = 0$$

このように、初期点に依存せずに停留点に収束するのは最急降下法の強みであるが、その一方で収束スピードが非常に遅い欠点を持つ。

最急降下法の 1 次収束性 最急降下法で生成される点列 $\{\boldsymbol{x}_k\}$ の収束先を $\boldsymbol{x}^*$ とすると以下の式が成り立つ定数 $0 < c < 1$ が存在する。

$$||\boldsymbol{x}_{k+1} - \boldsymbol{x}^*|| \le c||\boldsymbol{x}_k - \boldsymbol{x}^*||$$

この式では一見、収束が遅いようには見えないが、$\boldsymbol{x}^*$ 付近では $||\boldsymbol{x}_k - \boldsymbol{x}^*||$ が非常に微小のため、最急降下法が収束するまでに要する反復数は非常に多い。

3.4.2 ニュートン法

ニュートン法は $f(\boldsymbol{x})$2 次の項までのテイラー展開を最小化することを繰り返す方法である。すなわち、

$$q(\boldsymbol{d}) = f(\boldsymbol{x}_k + \boldsymbol{d}) = f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)^T \boldsymbol{d} + \frac{1}{2}\boldsymbol{d}^T \nabla^2 f(\boldsymbol{x}_k)\boldsymbol{d}$$

として、$\nabla q(\boldsymbol{d}) = \nabla f(\boldsymbol{x}_k) + \nabla^2 f(\boldsymbol{x}_k)\boldsymbol{d} = \boldsymbol{0}$ を満たす $\boldsymbol{d}$ を探索方向とする。方向 $\boldsymbol{d}$ はニュートン方向と呼ばれ、以下のニュートン方程式を解くことで得られる。

$$\nabla^2 f(\boldsymbol{x}_k)\boldsymbol{d} = -\nabla f(\boldsymbol{x}_k)$$

本実験で用いるニュートン法は以下の通りである。

ニュートン法

ステップ 0: 初期点 $\boldsymbol{x}_0$ を選び、$k = 0$ とする。$\epsilon = 10^{-8}$ とする

ステップ 1: $||\nabla f(\boldsymbol{x_k})|| < \epsilon$ が満たされていれば終了する

ステップ 2: 方程式 $\nabla^2 f(\boldsymbol{x}_k)\boldsymbol{d} = -\nabla f(\boldsymbol{x}_k)$ を解いて探索方向 $\boldsymbol{d}_k$ を求める

ステップ 3: $\boldsymbol{x}_{k+1} := \boldsymbol{x}_k + \boldsymbol{d}_k,\ k := k + 1$ としてステップ 1 に戻る

ニュートン法の局所的 2 次収束性 初期点 $\boldsymbol{x}_0$ を $\boldsymbol{x}^*$ の十分近くに点を取ると、ニュートン法で生成される点列 $\{\boldsymbol{x}_k\}$ は収束し、その収束先を $\boldsymbol{x}^*$ とすると以下を満たす定数 $c \geq 0$ が存在する。

$$||\boldsymbol{x}_{k+1} - \boldsymbol{x}^*|| \leq c||\boldsymbol{x}_k - \boldsymbol{x}^*||^2$$

1 次収束と異なり、上の 2 次収束は非常に速い。一方で、初期点の取り方が悪ければニュートン法は収束しなかったり、局所最適解でない点に収束することもしばしばある。

4　実験課題

次の各項目に取り組む。

(1) 実験目的とアルゴリズムに関する数学的理論について概要を述べる。

(2) 以下の 2 種類の 2 変数関数に対して最急降下法を実行するプログラムを完成させる。

$$
\begin{aligned}
f_1(x_1, x_2) &= \frac{1}{2}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}^T \begin{pmatrix} p & q \\ q & r \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}^T \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + e^{(x_1 - x_2)^2} \\
f_2(x_1, x_2) &= \sum_{i=0}^{4}\sum_{j=0}^{2} a_{ij} x_1^i x_2^j
\end{aligned}
$$

(3) (2) で作成したプログラムを動かしてふるまいを観察し、性能を考察する。

(4) 関数 f_1 と関数 f_2 に対してニュートン法を実行するプログラムを完成させる。

(5) (4) で作成したプログラムを動かしてふるまいを観察し、性能を考察する。

5　実験結果・検討・考察

(1)

上記 (目的、原理) で述べた。

(2)

それぞれの関数に対して作成したソースコード 2 つを付録に載せた。f_1 のソースコードも f_2 のソースコードも仕組みは同じなのでソースコード 1(4619055_辰川力駆_SD_1.c) に対しての説明をする。

まず、27 行目から 48 行目でファイルを読み取り、最急降下法を使う行列を受け取っている。そして、確認のために 50 行目から 71 行目で出力している。次に、メインの最急降下法 (77 行目から 114 行目) について説明する。79 行目から 83 行目までは $\nabla f(\boldsymbol{x}_k)$ と $\boldsymbol{d}_k$ を求めている。その後、ノルム、関数値を求めた後に、アルミホ条件による直線探索を用いてステップ幅を定めている。アルミホ条件によるステップ幅の決定は 116 行目から 142 行目の関数で求めている。

最後に $\boldsymbol{x}_k$ を更新したあと、101 行目から 108 行目で終了条件が満たされていれば終了する。満たされていなければさらに実行というようにしている。$||\nabla f(\boldsymbol{x_k})|| < \epsilon$ という条件だけでは、最急降下法はたくさんの回数を要するため時間がかかってしまう可能性があるので反復回数が 1000 回を越えたら停止するように工夫している。

出力部分に関しては、反復回数、現在位置、目的関数、勾配ベクトル、ノルムを表示するようにした。表示する頻度は多すぎるといけないので、0,1,2,3,4,5,10,20,30,40,50,60,70,80,90,100,200,300,400,500,600,700,800,900,1000 とするように工夫をした。

(3)

学籍番号は 4619055 なので seed 値を 55 としてデータを発生させた。実験に用いるデータは下記である。なお、図 2 の一番右のデータは data55.txt のデータである。

```
data set 1
matrix Q
   6.0   -5.0
  -5.0   13.0

vector c
  -5.0    1.0

three initial solutions
   9.0    0.0
 -13.0    1.0
   8.0   -1.0
```

```
data set 2
matrix Q
  10.0    6.0
   6.0    7.0

vector c
  -7.0   -8.0

three initial solutions
  10.0   -1.0
 -11.0   -3.0
   1.0   -6.0
```

```
data set 3
matrix Q
   2.0    1.0
   1.0    9.0

vector c
  -7.0   -6.0

three initial solutions
  -8.0   -3.0
  -5.0   -4.0
   1.0   -7.0
```

図 1: f_1 で用いる実験データ

```
data set 1
coefficient matrix A
   0.0  -27.0    1.0
  32.0  -11.0    0.0
   4.0    0.0    0.0
  -3.0    0.0    0.0
   1.0    0.0    0.0

three initial solutions
 -17.0  -36.0
 -33.0  -33.0
  27.0   11.0
```

```
data set 2
coefficient matrix A
   0.0  -33.0    1.0
 -15.0  -31.0    0.0
  -8.0    0.0    0.0
   9.0    0.0    0.0
   1.0    0.0    0.0

three initial solutions
 -24.0  -33.0
  36.0  -15.0
  25.0  -13.0
```

```
coefficient matrix A
   0.0    2.0    1.0
  -4.0   14.0    0.0
   4.0    0.0    0.0
   7.0    0.0    0.0
   1.0    0.0    0.0
three initial solutions A
 -41.0   32.0
  -3.0  -50.0
  31.0  -21.0
```

図 2: f_2 で用いる実験データ

実行結果は以下のようになった。

```
data set 1
coefficient matrix A
   0.0  -27.0    1.0
  32.0  -11.0    0.0
   4.0    0.0    0.0
  -3.0    0.0    0.0
   1.0    0.0    0.0

three initial solutions
 -17.0  -36.0
 -33.0  -33.0
  27.0   11.0
```

```
data set 2
coefficient matrix A
   0.0  -33.0    1.0
 -15.0  -31.0    0.0
  -8.0    0.0    0.0
   9.0    0.0    0.0
   1.0    0.0    0.0

three initial solutions
 -24.0  -33.0
  36.0  -15.0
  25.0  -13.0
```

```
coefficient matrix A
   0.0    2.0    1.0
  -4.0   14.0    0.0
   4.0    0.0    0.0
   7.0    0.0    0.0
   1.0    0.0    0.0
three initial solutions A
 -41.0   32.0
  -3.0  -50.0
  31.0  -21.0
```

図 3: f_1, data set 1, $\boldsymbol{x}_0$ の値 1 行目の結果

```
data set 1
coefficient matrix A
   0.0  -27.0    1.0
  32.0  -11.0    0.0
   4.0    0.0    0.0
  -3.0    0.0    0.0
   1.0    0.0    0.0

three initial solutions
 -17.0  -36.0
 -33.0  -33.0
  27.0   11.0
```

```
data set 2
coefficient matrix A
   0.0  -33.0    1.0
 -15.0  -31.0    0.0
  -8.0    0.0    0.0
   9.0    0.0    0.0
   1.0    0.0    0.0

three initial solutions
 -24.0  -33.0
  36.0  -15.0
  25.0  -13.0
```

```
coefficient matrix A
   0.0    2.0    1.0
  -4.0   14.0    0.0
   4.0    0.0    0.0
   7.0    0.0    0.0
   1.0    0.0    0.0
three initial solutions A
 -41.0   32.0
  -3.0  -50.0
  31.0  -21.0
```

図 4: f_1, data set 1, $\boldsymbol{x}_0$ の値 2 行目の結果

```
data set 1
coefficient matrix A
   0.0  -27.0    1.0
  32.0  -11.0    0.0
   4.0    0.0    0.0
  -3.0    0.0    0.0
   1.0    0.0    0.0

three initial solutions
 -17.0  -36.0
 -33.0  -33.0
  27.0   11.0
```

```
data set 2
coefficient matrix A
   0.0  -33.0    1.0
 -15.0  -31.0    0.0
  -8.0    0.0    0.0
   9.0    0.0    0.0
   1.0    0.0    0.0

three initial solutions
 -24.0  -33.0
  36.0  -15.0
  25.0  -13.0
```

```
coefficient matrix A
   0.0    2.0    1.0
  -4.0   14.0    0.0
   4.0    0.0    0.0
   7.0    0.0    0.0
   1.0    0.0    0.0
three initial solutions A
 -41.0   32.0
  -3.0  -50.0
  31.0  -21.0
```

図 5: f_1, data set 1, $\boldsymbol{x}_0$ の値 3 行目の結果

(4)

それぞれの関数に対して作成したソースコード 2 つを付録に載せた。f_1 のソースコードも f_2 のソースコードも仕組みは同じなのでソースコード 3(4619055_辰川力駆_newton_1.c) に対しての説明をする。

まず、29 行目から 50 行目でファイルを読み取り、ニュートン法を使う行列を受け取っている。そして、確認のために 52 行目から 73 行目で出力している。次に、メインの最急降下法 (79 行目から 111 行目) について説明する。81,82 行目では $\nabla f(\boldsymbol{x}_k)$ を求めている。その後、ノルム、探索方向 $\boldsymbol{d}_k$、関数値を求めている。探索方向はニュートン法を使っている。ニュートン法は関数化している。ニュートン法では、まずヘッセ行列 $\nabla^2 f(\boldsymbol{x_k})$ を求めて、正則でなければ、$\boldsymbol{d} = \mathrm{inf}$ としている。正則であればヘッセ行列の逆行列を求めて、$\boldsymbol{d}$ を計算している。

最後に $\boldsymbol{x}_k$ を更新したあと、98 行目から 110 行目で終了条件が満たされていれば終了する。満たされていなければさらに実行というようにしている。$||\nabla f(\boldsymbol{x_k})|| < \epsilon$ という停止条件と反復回数が 500 回を超えるという停止条件以外に $\nabla^2 f(\boldsymbol{x_k})$ が非正則なら停止するようにしている。

出力部分に関しては最急降下法と同様に、反復回数、現在位置、目的関数、勾配ベクトル、ノルムを表示するようにした。表示する頻度は多すぎるといけないので、0,1,2,3,4,5,10,20,30,40,50,60,70, 80,90,100,200,300,400,500,600,700,800,900,1000 とするように工夫をした。

(5)

ニュートン法は非正則になったら終了しないといけないのでつらい

6　まとめ

最急降下法とニュートン法を C 言語で実装した。また、それらの適用を通し、アルゴリズムの特性を理解することができた。

参考文献

[1] 東京理科大学工学部情報工学科 情報工学実験 2 2020 年度東京理科大学工学部情報工学科出版

[2] 関数グラフ - GeoGebra

https://www.geogebra.org/graphing?lang=ja

最終閲覧日 2020/10/29

A 付録

ソースコード 1: 4619055_辰川力駆_SD_1.c

```c
//4619055 辰川力駆
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define N 1000
#define res 0.00000001 ///停止条件

double func(double *x); ///f1の式を用いている
double Armijo(double *x, double *d, double *nabla); //Armijo

double Q[2][2]; //Qのデータを保存
double c[2]; //cのデータを保存

int main()
{
    char fname[128];
    double x[2];
    double p, q, r;
    double d[2], norm;
    double nabla[2];
    double f;
    FILE *fp;
    int i, j, k;
    double alpha;

    printf("input filename:");
    fgets(fname, sizeof(fname), stdin);
    fname[strlen(fname) - 1] = '\0';
    fflush(stdin);
    fp = fopen(fname, "r");

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            fscanf(fp, "%lf", &Q[i][j]);
        }
    }
    for (i = 0; i < 2; i++)
    {
        fscanf(fp, "%lf", &c[i]);
    }
    for (i = 0; i < 2; i++)
    {
        fscanf(fp, "%lf", &x[i]);
    }
    fclose(fp);
```

```c

    printf("Q=\n");
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            printf("%4.1f", Q[i][j]);
        }
        printf("\n");
    }
    printf("C=\n");
    for (i = 0; i < 2; i++)
    {
        printf("%4.1f", c[i]);
    }
    printf("\n");
    printf("x=\n");
    for (i = 0; i < 2; i++)
    {
        printf("%4.1f", x[i]);
    }
    printf("\n");
    printf("-----\n");

    p = Q[0][0];
    q = Q[0][1];
    r = Q[1][1];
    k = 0;
    while (1)
    {
        nabla[0] = p * x[0] + q * x[1] + c[0] + 2 * (x[0] - x[1]) * exp((x[0] - x[1]) * (x[0] - x[1]));
        nabla[1] = q * x[0] + r * x[1] + c[1] - 2 * (x[0] - x[1]) * exp((x[0] - x[1]) * (x[0] - x[1]));

        d[0] = -nabla[0];
        d[1] = -nabla[1];

        norm = sqrt(nabla[0] * nabla[0] + nabla[1] * nabla[1]); ///ノルムを計算

        f = func(x);
        alpha = Armijo(x, d, nabla);

        if (k <= 5 || (k <= 100 && k % 10 == 0) || k % 100 == 0) //5回目までと10n回目,100n回目(n=1,2,3,...10)を表示
        {
            printf("%d回目\n", k);
            printf("現在位置=(%.8lf,%.8lf)\n目的関数=%.8lf\n勾配ベクトル=(%.8lf,%.8lf)\nノルム=%.8lf\n", x[0], x[1], f, nabla[0], nabla[1], norm);
            printf("-----\n");
        }

```

```c
            x[0] = x[0] + alpha * d[0];
            x[1] = x[1] + alpha * d[1];
            k++;

            if (norm < res) ///nablaのノルムが10^{-8}を越えていたら停止
            {
                break;
            }
            if (k > N) ///反復回数が1000回越えたら停止
            {
                break;
            }
        }

        printf("--結果--\n");
        printf("%d回\n現在位置=(%.8lf,%.8lf)\n目的関数=%.8lf\n勾配ベクトル=(%.8lf
            ,%.8lf)\nノルム=%.8lf\n", k - 1, x[0], x[1], f, nabla[0], nabla[1], norm);
        return 0;
    }

    double Armijo(double *x, double *d, double *nabla)
    {
        double alpha = 1.0;
        double xi = 0.1; ///今回の課題条件
        double tau = 0.5; ///今回の課題条件
        double fx, fy;
        double y[2];

        ///f(x_k)を求める
        fx = func(x);

        while (1)
        {
            ///新しいalphaに対してのfyを求める
            y[0] = x[0] + alpha * d[0];
            y[1] = x[1] + alpha * d[1];
            fy = func(y);

            if (fy <= fx + xi * (nabla[0] * d[0] + nabla[1] * d[1]) * alpha) //Armijo条件
            {
                break;
            }

            alpha = tau * alpha; //alphaの更新
        }
        return alpha;
    }

    // f1の式を用いている
    double func(double *x)
    {
        double p, q, r;
```

```c
    p = Q[0][0];
    q = Q[0][1];
    r = Q[1][1];
    return 0.5 * (p * x[0] * x[0] + 2 * q * x[0] * x[1] + r * x[1] * x[1]) + c[0] * x[0] + c[1] * x[1] + exp((x[0] - x[1]) * (x[0] - x[1]));
}
```

ソースコード 2: 4619055_辰川力駆_SD_2.c

```c
//4619055 辰川力駆
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define N 1000
#define res 0.00000001 ///停止条件

double func(double *x); ///f2の式を用いている
double Armijo(double *x, double *d, double *nabla); //Armijo

double A[5][3]; //Aのデータを保存

int main()
{
    char fname[128];
    double x[2];
    double d[2], norm;
    double nabla[2];
    double f;
    FILE *fp;
    int i, j, k;
    double alpha;

    printf("input filename:");
    fgets(fname, sizeof(fname), stdin);
    fname[strlen(fname) - 1] = '\0';
    fflush(stdin);
    fp = fopen(fname, "r");

    for (i = 0; i < 5; i++)
    {
        for (j = 0; j < 3; j++)
        {
            fscanf(fp, "%lf", &A[i][j]);
        }
    }
    for (i = 0; i < 2; i++)
    {
        fscanf(fp, "%lf", &x[i]);
    }
    fclose(fp);

    printf("A=\n");
```

```
        for (i = 0; i < 5; i++)
        {
            for (j = 0; j < 3; j++)
            {
                printf("%4.1f", A[i][j]);
            }
            printf("\n");
        }
        printf("x=\n");
        for (i = 0; i < 2; i++)
        {
            printf("%4.1f", x[i]);
        }
        printf("\n");
        printf("-----\n");

        k = 0;
        while (1)
        {
            nabla[0] = A[1][0] + A[1][1] * x[1] + 2 * A[2][0] * x[0] + 3 * A[3][0] * pow(x[0], 2) + 4 * A[4][0] * pow(x[0], 3);
            nabla[1] = A[0][1] + 2 * A[0][2] * x[1] + A[1][1] * x[0];

            d[0] = -nabla[0];
            d[1] = -nabla[1];

            norm = sqrt(nabla[0] * nabla[0] + nabla[1] * nabla[1]); ///ノルムを計算

            f = func(x);
            alpha = Armijo(x, d, nabla);

            if (k <= 5 || (k <= 100 && k % 10 == 0) || k % 100 == 0) //5回目までと10n回目,100n回目(n=1,2,3,...10)を表示
            {
                printf("%d回目\n", k);
                printf("現在位置=(%.8lf,%.8lf)\n目的関数=%.8lf\n勾配ベクトル=(%.8lf,%.8lf)\nノルム=%.8lf\n", x[0], x[1], f, nabla[0], nabla[1], norm);
                printf("-----\n");
            }

            x[0] = x[0] + alpha * d[0];
            x[1] = x[1] + alpha * d[1];
            k++;

            if (norm < res) ///nablaのノルムが10^{-8}を越えていたら停止
            {
                break;
            }
            if (k > N) ///反復回数が1000回越えたら停止
            {
                break;
            }
```

```c
    }

    printf("--結果--\n");
    printf("%d回\n現在位置=(%.8lf,%.8lf)\n目的関数=%.8lf\n勾配ベクトル=(%.8lf,%.8lf)\nノルム=%.8lf\n", k - 1, x[0], x[1], f, nabla[0], nabla[1], norm);
    return 0;
}

double Armijo(double *x, double *d, double *nabla)
{
    double alpha = 1.0;
    double xi = 0.1; ///今回の課題条件
    double tau = 0.5; ///今回の課題条件
    double fx, fy;
    double y[2];

    ///f(x_k)を求める
    fx = func(x);

    while (1)
    {
        ///新しいalphaに対してのfyを求める
        y[0] = x[0] + alpha * d[0];
        y[1] = x[1] + alpha * d[1];
        fy = func(y);

        if (fy <= fx + xi * (nabla[0] * d[0] + nabla[1] * d[1]) * alpha) //Armijo条件
        {
            break;
        }

        alpha = tau * alpha; //alphaの更新
    }
    return alpha;
}

//f2の式を用いている
double func(double *x)
{
    return A[0][1] * x[1] + A[0][2] * pow(x[1], 2) + A[1][0] * x[0] + A[1][1] * x[0] * x[1] + A[2][0] * pow(x[0], 2) + A[3][0] * pow(x[0], 3) + A[4][0] * pow(x[0], 4);
}
```

ソースコード 3: 4619055_辰川力駆_newton_1.c

```c
//4619055 辰川力駆
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define inf 10000000000000
#define N 500
#define res 0.00000001 ///停止条件
```

```c

double func(double *x); ///f1の式を用いている
void Hf(double *x, double *d); ///ヘッセ行列を求める
void Newton(double *x, double *d, double *nabla); //dを計算する

double Q[2][2]; //Qのデータを保存
double c[2]; //cのデータを保存
double Hessian[2][2]; ///ヘッセ行列を保存

int main()
{
    char fname[128];
    double x[2];
    double p, q, r;
    double d[2], norm;
    double nabla[2];
    double f;
    FILE *fp;
    int i, j, k;

    printf("input filename:");
    fgets(fname, sizeof(fname), stdin);
    fname[strlen(fname) - 1] = '\0';
    fflush(stdin);
    fp = fopen(fname, "r");

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            fscanf(fp, "%lf", &Q[i][j]);
        }
    }
    for (i = 0; i < 2; i++)
    {
        fscanf(fp, "%lf", &c[i]);
    }
    for (i = 0; i < 2; i++)
    {
        fscanf(fp, "%lf", &x[i]);
    }
    fclose(fp);

    printf("Q=\n");
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            printf("%4.1lf", Q[i][j]);
        }
        printf("\n");
    }
```

```c
        printf("C=\n");
        for (i = 0; i < 2; i++)
        {
            printf("%4.1f", c[i]);
        }
        printf("\n");
        printf("x=\n");
        for (i = 0; i < 2; i++)
        {
            printf("%4.1f", x[i]);
        }
        printf("\n");
        printf("-----\n");

        p = Q[0][0];
        q = Q[0][1];
        r = Q[1][1];
        k = 0;
        while (1)
        {
            nabla[0] = p * x[0] + q * x[1] + c[0] + 2 * (x[0] - x[1]) * exp((x[0] - x[1]) * (x[0] - x[1]));
            nabla[1] = q * x[0] + r * x[1] + c[1] - 2 * (x[0] - x[1]) * exp((x[0] - x[1]) * (x[0] - x[1]));

            norm = sqrt(nabla[0] * nabla[0] + nabla[1] * nabla[1]); ///ノルムを計算

            Newton(x, d, nabla); //dを計算
            f = func(x);

            if (k <= 5 || (k <= 100 && k % 10 == 0) || k % 100 == 0) //5回目までと10n回目,100n回目(n=1,2,3,...10)を表示
            {
                printf("%d回目\n", k);
                printf("現在位置=(%.8lf,%.8lf)\n目的関数=%.8lf\n勾配ベクトル=(%.8lf,%.8lf)\nノルム=%.8lf\n", x[0], x[1], f, nabla[0], nabla[1], norm);
                printf("-----\n");
            }
            x[0] = x[0] + d[0];
            x[1] = x[1] + d[1];
            k++;
            if (norm < res) ///nablaのノルムが10^{-8}を越えていたら停止
            {
                break;
            }
            if (k > N) ///反復回数が500回越えたら停止
            {
                break;
            }
            if (d[0] == inf) ///nabla^2が非正則なら停止
            {
                printf("非正則なので終了します\n");
```

```
                break;
            }
        }
        printf("--結果--\n");
        printf("%d 回\n 現在位置=(%.8lf,%.8lf)\n 目的関数=%.8lf\n 勾配ベクトル=(%.8lf,%.8lf)\n ノルム=%.8lf\n", k - 1, x[0], x[1], f, nabla[0], nabla[1], norm);
        return 0;
    }

    void Newton(double *x, double *d, double *nabla)
    {
        double Hessian_inverse[2][2];
        Hf(x, d);
        if (Hessian[0][0] * Hessian[1][1] - Hessian[0][1] * Hessian[1][0] == 0) ///正則かどうか
        {
            d[0] = inf;
            d[1] = inf;
        }
        else
        {
            Hessian_inverse[0][0] = 1 / (Hessian[0][0] * Hessian[1][1] - Hessian[0][1] * Hessian[1][0]) * Hessian[1][1];
            Hessian_inverse[0][1] = 1 / (Hessian[0][0] * Hessian[1][1] - Hessian[0][1] * Hessian[1][0]) * -Hessian[0][1];
            Hessian_inverse[1][0] = 1 / (Hessian[0][0] * Hessian[1][1] - Hessian[0][1] * Hessian[1][0]) * -Hessian[1][0];
            Hessian_inverse[1][1] = 1 / (Hessian[0][0] * Hessian[1][1] - Hessian[0][1] * Hessian[1][0]) * Hessian[0][0];
            d[0] = -(Hessian_inverse[0][0] * nabla[0] + Hessian_inverse[1][0] * nabla[1]);
            d[1] = -(Hessian_inverse[0][1] * nabla[0] + Hessian_inverse[1][1] * nabla[1]);
        }
    }

    void Hf(double *x, double *d)
    {
        double p, q, r;
        p = Q[0][0];
        q = Q[0][1];
        r = Q[1][1];
        Hessian[0][0] = p + 2 * (1 + 2 * pow(x[0] - x[1], 2)) * exp(pow(x[0] - x[1], 2));
        Hessian[0][1] = q - 2 * (1 + 2 * pow(x[0] - x[1], 2)) * exp(pow(x[0] - x[1], 2));
        Hessian[1][0] = Hessian[0][1];
        Hessian[1][1] = r + 2 * (1 + 2 * pow(x[0] - x[1], 2)) * exp(pow(x[0] - x[1], 2));
    }

    // f1 の式を用いている
    double func(double *x)
    {
        double p, q, r;
        p = Q[0][0];
        q = Q[0][1];
        r = Q[1][1];
```

```c
    return 0.5 * (p * x[0] * x[0] + 2 * q * x[0] * x[1] + r * x[1] * x[1]) + c[0] * x[0] + c[1] * x[1] + exp((x[0] - x[1]) * (x[0] - x[1]));
}
```

ソースコード 4: 4619055_辰川力駆_newton_2.c

```c
//4619055 辰川力駆
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define inf 10000000000000
#define N 500
#define res 0.00000001 ///停止条件

double func(double *x); ///f2 の式を用いている
void Hf(double *x, double *d); ///ヘッセ行列を求める
void Newton(double *x, double *d, double *nabla); //d を計算する

double A[5][3]; //A のデータを保存
double Hessian[2][2]; ///ヘッセ行列を保存

int main()
{
    char fname[128];
    double x[2];
    double d[2], norm;
    double nabla[2];
    double f;
    FILE *fp;
    int i, j, k;

    printf("input filename:");
    fgets(fname, sizeof(fname), stdin);
    fname[strlen(fname) - 1] = '\0';
    fflush(stdin);
    fp = fopen(fname, "r");

    for (i = 0; i < 5; i++)
    {
        for (j = 0; j < 3; j++)
        {
            fscanf(fp, "%lf", &A[i][j]);
        }
    }
    for (i = 0; i < 2; i++)
    {
        fscanf(fp, "%lf", &x[i]);
    }
    fclose(fp);

    printf("A=\n");
    for (i = 0; i < 5; i++)
```

```
        {
            for (j = 0; j < 3; j++)
            {
                printf("%4.1f", A[i][j]);
            }
            printf("\n");
        }
        printf("x=\n");
        for (i = 0; i < 2; i++)
        {
            printf("%4.1f", x[i]);
        }
        printf("\n");
        printf("-----\n");

        k = 0;
        while (1)
        {
            nabla[0] = A[1][0] + A[1][1] * x[1] + 2 * A[2][0] * x[0] + 3 * A[3][0] * pow(x[0],
                2) + 4 * A[4][0] * pow(x[0], 3);
            nabla[1] = A[0][1] + 2 * A[0][2] * x[1] + A[1][1] * x[0];

            norm = sqrt(nabla[0] * nabla[0] + nabla[1] * nabla[1]); ///ノルムを計算

            Newton(x, d, nabla); //dを計算
            f = func(x);
            if (k <= 5 || (k <= 100 && k % 10 == 0) || k % 100 == 0) //5回目までと10
                n回目,100n回目(n=1,2,3,...10)を表示
            {
                printf("%d回目\n", k);
                printf("現在位置=(%.8lf,%.8lf)\n目的関数=%.8lf\n勾配ベクトル=(%.8lf
                    ,%.8lf)\nノルム=%.8lf\n", x[0], x[1], f, nabla[0], nabla[1], norm);
                printf("-----\n");
            }
            x[0] = x[0] + d[0];
            x[1] = x[1] + d[1];
            k++;
            if (norm < res) ///nablaのノルムが10^{-8}を越えていたら停止
            {
                break;
            }
            if (k > N) ///反復回数が500回越えたら停止
            {
                break;
            }
            if (d[0] == inf) ///nabla^2が非正則なら停止
            {
                printf("非正則なので終了します\n");
                break;
            }
        }

```

```
        printf("--結果--\n");
        printf("%d 回\n 現在位置=(%.8lf,%.8lf)\n 目的関数=%.8lf\n 勾配ベクトル=(%.8lf,%.8lf)\n ノルム=%.8lf\n", k - 1, x[0], x[1], f, nabla[0], nabla[1], norm);
        return 0;
    }

    void Newton(double *x, double *d, double *nabla)
    {
        double Hessian_inverse[2][2];
        Hf(x, d);
        if (Hessian[0][0] * Hessian[1][1] - Hessian[0][1] * Hessian[1][0] == 0) ///正則かどうか
        {
            d[0] = inf;
            d[1] = inf;
        }
        else
        {
            Hessian_inverse[0][0] = 1 / (Hessian[0][0] * Hessian[1][1] - Hessian[0][1] * Hessian[1][0]) * Hessian[1][1];
            Hessian_inverse[0][1] = 1 / (Hessian[0][0] * Hessian[1][1] - Hessian[0][1] * Hessian[1][0]) * -Hessian[0][1];
            Hessian_inverse[1][0] = 1 / (Hessian[0][0] * Hessian[1][1] - Hessian[0][1] * Hessian[1][0]) * -Hessian[1][0];
            Hessian_inverse[1][1] = 1 / (Hessian[0][0] * Hessian[1][1] - Hessian[0][1] * Hessian[1][0]) * Hessian[0][0];
            d[0] = -(Hessian_inverse[0][0] * nabla[0] + Hessian_inverse[1][0] * nabla[1]);
            d[1] = -(Hessian_inverse[0][1] * nabla[0] + Hessian_inverse[1][1] * nabla[1]);
        }
    }

    void Hf(double *x, double *d)
    {
        Hessian[0][0] = 2 * A[2][0] + 6 * A[3][0] * x[0] + 12 * A[4][0] * pow(x[0], 2);
        Hessian[0][1] = A[1][1];
        Hessian[1][0] = Hessian[0][1];
        Hessian[1][1] = 2 * A[0][2];
    }

    //f2 の式を用いている
    double func(double *x)
    {
        return A[0][1] * x[1] + A[0][2] * pow(x[1], 2) + A[1][0] * x[0] + A[1][1] * x[0] * x[1] + A[2][0] * pow(x[0], 2) + A[3][0] * pow(x[0], 3) + A[4][0] * pow(x[0], 4);
    }
```