山田龍

2020 年 11 月 24 日

1 Gauss-Seidel 法

SOR 法の導出を知りたかったので、ここに示した。Gauss-Seidel 法と SOR 法の導出については [1] の導出を参考にしながら行間をできる限り埋めた。

$$A = D + L + U \tag{1}$$

の順番で対角成分、下三角成分、上三角成分を定義する。解くべき連立方程式は、

$$(D + L + U)\mathbf{x} = \mathbf{b} \tag{2}$$

$$\mathbf{x} = -D^{-1}(L + U)\mathbf{x} + D^{-1}\mathbf{b} \tag{3}$$

ヤコビ法では、以下の漸化式が成立すれば $x^{(k)}$ は連立方程式の解である。

$$\mathbf{x}^{(k+1)} = -D^{-1}L\mathbf{x}^{(k)} - D^{-1}U\mathbf{x}^{(k)} + D^{-1}\mathbf{b} \tag{4}$$

Gauss-Seidel 法では、以下の漸化式が成立すれば $x^{(k)}$ は連立方程式の解である。

$$\mathbf{x}^{(k+1)} = -D^{-1}L\mathbf{x}^{(k+1)} - D^{-1}U\mathbf{x}^{(k)} + D^{-1}\mathbf{b} \tag{5}$$

$$= -D^{-1}(L + U)\mathbf{x}^{(k)} + D^{-1}\mathbf{b} \tag{6}$$

反復の計算は、

$$\begin{aligned}
(E + D^{-1}L)\mathbf{x}^{(k+1)} &= -D^{-1}U\mathbf{x}^{(k)} + D^{-1}\mathbf{b} \\
(D + L)D^{-1}\mathbf{x}^{(k+1)} &= -D^{-1}U\mathbf{x}^{(k)} + D^{-1}\mathbf{b} \\
\mathbf{x}^{(k+1)} &= -(D + L)^{-1}U\mathbf{x}^{(k)} + (D + L)^{-1}\mathbf{b}
\end{aligned}$$

ここで、$\mathbf{d}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ を定義する。

$$\begin{aligned}
\mathbf{d}^{(k+1)} &= -\left[(D + L)^{-1}U + E\right]\mathbf{x}^{(k)} + (D + L)^{-1}\mathbf{b} \\
&= -\left[(D + L)^{-1}U - D^{-1}U + D^{-1}U + E\right]\mathbf{x}^{(k)} \\
&+ \left[(D + L)^{-1} - D^{-1} + D^{-1}\right]\mathbf{b} \\
&= -\left[-L(D + L)^{-1}D^{-1}U + D^{-1}U + E\right]\mathbf{x}^{(k)} \\
&+ \left[-L(D + L)^{-1}D^{-1} + D^{-1}\right]\mathbf{b} \\
&= -D^{-1}L\mathbf{x}^{(k+1)} - \left(D^{-1}U + E\right)\mathbf{x}^{(k)} + D^{-1}\mathbf{b}
\end{aligned}$$

更に変形する。

$$
\begin{aligned}
(E+D^{-1}L)\mathbf{d}^{(k+1)} \\
&= -D^{-1}L(-D^{-1}U\mathbf{x}^{(k)}+D^{-1}\mathbf{b}) \\
&- \left(D^{-1}U+E\right)(-D^{-1}U\mathbf{x}^{(k-1)}+D^{-1}\mathbf{b}) \\
&+ (D^{-1}+D^{-1}LD^{-1})\mathbf{b} \\
&= -D^{-1}LD^{-1}U\mathbf{x}^{(k)} - D^{-1}LD^{-1}\mathbf{b} \\
&+ \left(D^{-1}UD^{-1}U+D^{-1}U\right)\mathbf{x}^{(k-1)} \\
&- (D^{-1}UD^{-1}+D^{-1})\mathbf{b} \\
&+ (D^{-1}+D^{-1}LD^{-1})\mathbf{b}
\end{aligned}
$$

両辺変形して、

$$
\begin{aligned}
D^{-1}(D+L)\mathbf{d}^{(k+1)} &= D^{-1}[-LD^{-1}U\mathbf{x}^{(k)} - LD^{-1}\mathbf{b} \\
&+ \left(UD^{-1}U+U\right)\mathbf{x}^{(k-1)} \\
&+ (-UD^{-1}+LD^{-1})\mathbf{b}] \\
(D+L)\mathbf{d}^{(k+1)} &= -LD^{-1}U\mathbf{x}^{(k)} - LD^{-1}\mathbf{b} \\
&+ \left(UD^{-1}U+U\right)\mathbf{x}^{(k-1)} \\
&+ (-UD^{-1}+LD^{-1})\mathbf{b}
\end{aligned}
$$

左辺の係数の逆行列を両辺にかけて、

$$
\begin{aligned}
\mathbf{d}^{(k+1)} &= (D+L)^{-1}U[-LD^{-1}\mathbf{x}^{(k)} + \left(D^{-1}U+E\right)\mathbf{x}^{(k-1)} - D^{-1}\mathbf{b}] \\
&= (D+L)^{-1}U\mathbf{d}^{(k)}
\end{aligned}
$$

よって Gauss-Seidel 法において解が収束する条件は、

$$||(D+L)^{-1}U||_2 < 1 \tag{7}$$

反復すべき式は、

$$D\mathbf{x}^{(k+1)} = -L\mathbf{x}^{(k+1)} - U\mathbf{x}^{(k)} + \mathbf{b} \tag{8}$$

$$a_{ii}\mathbf{x}_i^{(k+1)} = -\sum_{j=1}^{i-1} a_{ij}\mathbf{x}_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\mathbf{x}_j^{(k+1)} + \mathbf{b}_i \tag{9}$$

2 SOR 法

Gauss-Seidel 法において、緩和係数 ω を持ち込んで改良した解法を SOR 法という。Gauss-Seidel 法の計算式を変形する。

$$(E+D^{-1}L)\mathbf{x}^{(k+1)} = -D^{-1}U\mathbf{x}^{(k)} + D^{-1}\mathbf{b} \tag{10}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - D^{-1}L\mathbf{x}^{(k+1)} - (D^{-1}U+E)\mathbf{x}^{(k)} + D^{-1}\mathbf{b} \tag{11}$$

SOR 法では緩和係数 ω を導入する。

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega \left[-D^{-1}L\mathbf{x}^{(k+1)} - (D^{-1}U + E)\mathbf{x}^{(k)} + D^{-1}\mathbf{b} \right] \tag{12}$$

$$(E + \omega D^{-1}L)\mathbf{x}^{(k+1)} = \left[E - \omega(D^{-1}U + E) \right] \mathbf{x}^{(k)} + \omega D^{-1}\mathbf{b} \tag{13}$$

反復すべき式は、

$$D\mathbf{x}^{(k+1)} = D\mathbf{x}^{(k)} + \omega \left[-L\mathbf{x}^{(k+1)} - (U + D)\mathbf{x}^{(k)} + \mathbf{b} \right] \tag{14}$$

$$D\mathbf{x}^{(k+1)} = -\omega L\mathbf{x}^{(k+1)} + [D - \omega(U + D)]\, \mathbf{x}^{(k)} + \omega\mathbf{b} \tag{15}$$

成分表示すれば、

$$a_{ii}\mathbf{x}_i^{(k+1)} = -\omega \sum_{j=1}^{i-1} a_{ij}\mathbf{x}_j^{(k+1)} + (1-\omega)a_{ii}\mathbf{x}_i^{(k)} - \omega \sum_{j=i+1}^{n} a_{ij}\mathbf{x}_j^{(k)} + \omega\mathbf{b}_i \tag{16}$$

緩和係数は問題に応じて選択される。

3 課題 1

$$\begin{pmatrix} 10 & 1 & 4 & 0 \\ 1 & 10 & 5 & -1 \\ 4 & 5 & 10 & 7 \\ 0 & -1 & 7 & 9 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 15 \\ 15 \\ 26 \\ 15 \end{pmatrix} \tag{17}$$

$$\begin{pmatrix} 10 & 1 & 4 & 0 \\ 1 & 10 & 5 & -1 \\ 4 & 5 & 10 & 7 \\ 0 & -1 & 7 & 9 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 16 \\ 16 \\ 25 \\ 16 \end{pmatrix} \tag{18}$$

の解を Gauss の消去法を用いて求める。

3.1 単精度と倍精度

計算機において実数を表示するためには浮動小数点法が使われる。計算機の内部において 2 進数が使われていたとき、その表現は、以下のよう記述できる。

$$\pm(1.a_1a_2\cdots a_m) \times 2^e \tag{19}$$

では、単精度の場合に指数部が 8 ビットであったとしよう。いま、10 進数で 12.5 の数値を浮動小数点表示したいとする。まず符号は + なので符号の部分のビットは 0 である。12.5 は 2 進数表示で 1.1001×2^3 と表される。指数部分はバイアス $128 - 1 = 127$ を使った記法を用いれば、$(3 + 127)_{10} = (130)_{10} = (10000010)_2$ となる。仮数部は、正規化された 2 進数表示の場合には必ず先頭は 1 になるので、先頭を省略して $10010\cdots0$ となる。図示すれば以下のようになる。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$\cdots$	32
符号	指数部	-	-	-	-	-	-	-	仮数部	-	-	-	-	-	$\cdots$	-
0	1	0	0	0	0	0	1	0	1	0	0	1	0	0	$\cdots$	0

numpy の float32、float64 においては、numpy.finfo 関数を使ってそれぞれの仕様を確認することができる。

```
  Machine parameters for float32
precision = 6 resolution = 1.0000000e-06
machep = -23 eps = 1.1920929e-07
negep = -24 epsneg = 5.9604645e-08
minexp = -126 tiny = 1.1754944e-38
maxexp = 128 max = 3.4028235e+38
nexp = 8 min = -max
```

```
  Machine parameters for float64
precision = 15 resolution = 1.0000000000000001e-15
machep = -52 eps = 2.2204460492503131e-16
negep = -53 epsneg = 1.1102230246251565e-16
minexp = -1022 tiny = 2.2250738585072014e-308
maxexp = 1024 max = 1.7976931348623157e+308
nexp = 11 min = -max
```

ここから、単精度では指数部が $8bit$ で仮数部が $32-1-8=23bit$ あるので 10 進数では 6 桁の精度、倍精度では指数部が $11bit$ で仮数部が $64-1-11=52bit$ あるので 10 進数では 15 桁の精度があることがわかる。

3.2 $b=(15,15,26,15)^T$ の場合

Gauss の消去法を使って $A\mathbf{x}=\mathbf{b}$ の解 $\mathbf{x}$ を求めた。単精度では、

```
  x: [1. 1.0000001 0.9999999 1. ]
  A * x: [14.99999964 15.0000006 25.9999994 14.99999905]
```

倍精度では、

```
  x: [1. 1. 1. 1.]
  A * x: [15. 15. 26. 15.]
```

比較すると、単精度の場合には 7 桁目以降で誤差が生じている。

3.3 $b=(16,16,25,16)^T$ の場合

Gauss の消去法を使って $A\mathbf{x}=\mathbf{b}$ の解 $\mathbf{x}$ を求めた。単精度では、

```
  x: [ 832.18555 1324.2953 -2407.5376 2021.451 ]
  A * x: [16.00036621 15.99938965 25. 16.00097656]
```

倍精度では、

```
  x: [ 832. 1324. -2407. 2021.]
  A * x: [16. 16. 25. 16.]
```

比較すると、4 桁目以降に誤差が生じており、単精度の範囲での演算では誤差が大きくなっている。

4 課題 2

$$\begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 0 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{20}$$

の解を SOR 法を用いて求める。

4.1　解と反復係数

解を計算によって求めた。

x: [0.72727273 −0.27272727 −0.18181818 −0.09090909]

横軸を反復回数、縦軸を真の解からの誤差のノルムの対数にとる。また、計算は 10 桁よりも発散が大きくなったら打ち切る。この条件のもとで、ω の値を変えてプロットすると以下のようになる。$w <= 1$ では収

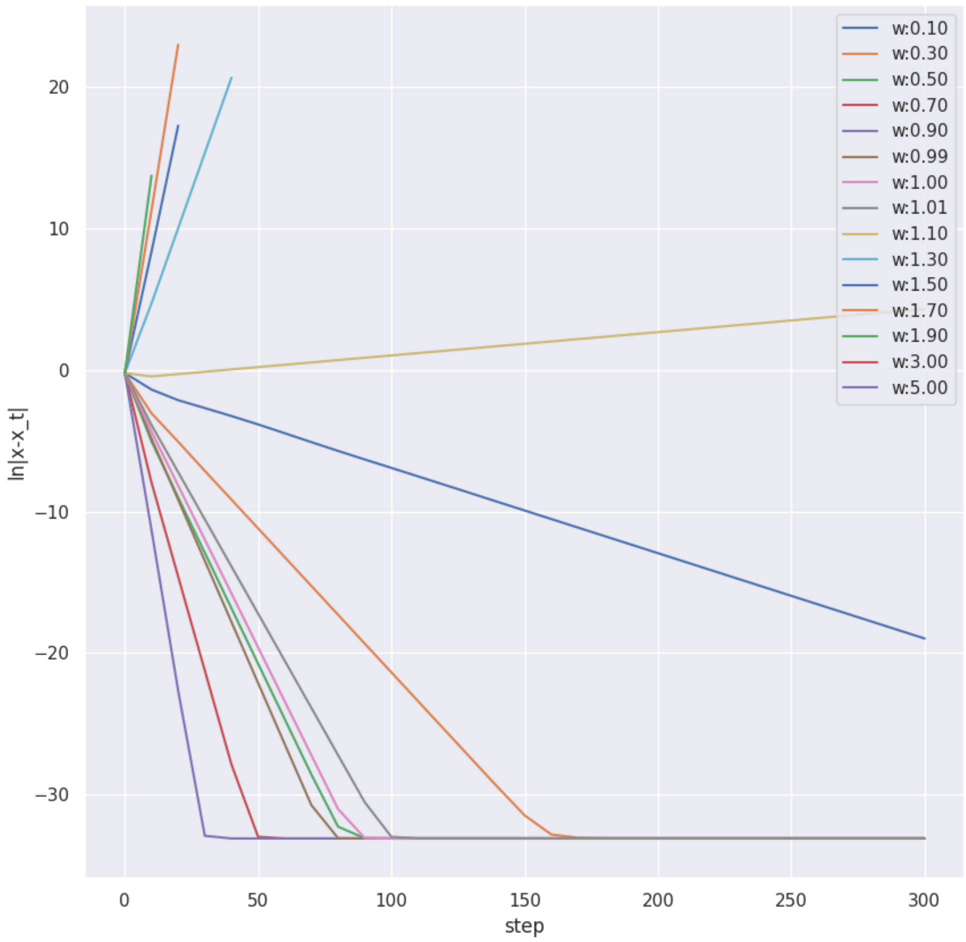

図 1　SOR 法の誤差

束、$w > 1$ では発散している。プログラムのミスによるものかを確認するために、例えば、課題 1 で指定されたような連立方程式に対して SOR 法を用いると、以下のようになる。

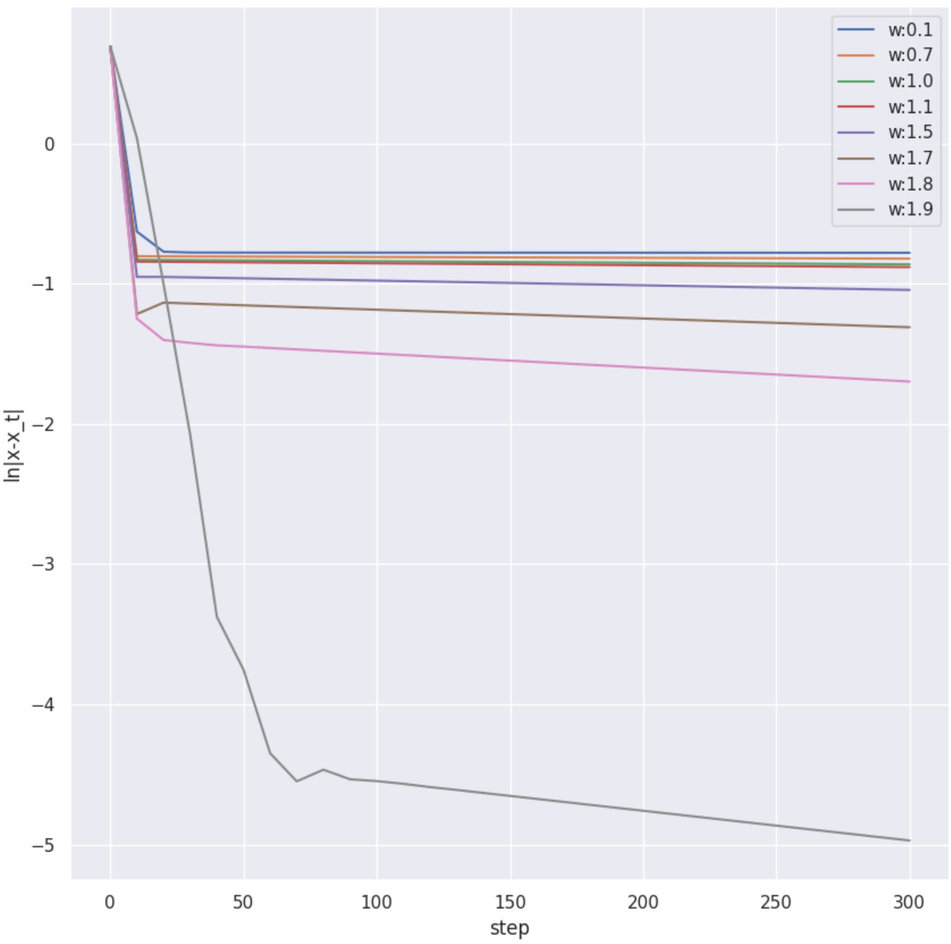

図 2　SOR 法の誤差

すべての $0 < \omega < 2$ の緩和係数に対して収束している。また、$\omega = 1.9$ のときだけ精度が他の緩和係数のときよりも高いこともわかった。

プログラムは正しく走っていると考えられるので、最初の場合に ω が 1 よりも少し高いところから先で発散している理由を考えた。私は、行列の固有値が $3, 2.2469796037175, -0.80193773580484, 0.55495813208737$ で絶対値が 1 より大きいものがあることから、本当は Gauss-Seidel 法が使えない状況であるために w が 1 より小さい場合にのみ収束したと考えた。

5 プログラム

ソースコードを添付しますが、全て https://github.com/tychy/NumericalAnalysisPlayground にあります。各関数のテストもついています。

ソースコード 1 gauss の消去法

```
import numpy as np


def scale(A, x):
    scale_ls = np.zeros_like(x)
    for i in range(A.shape[0]):
        max_idx = 0
        for j in range(A.shape[0]):
            if np.abs(A[i, j]) > np.abs(A[i, max_idx]):
                max_idx = j
        scale_ls[i] = np.abs(A[i, max_idx])
        x[i] = x[i] / np.abs(A[i, max_idx])
        A[i, :] = A[i, :] / np.abs(A[i, max_idx])
    return scale_ls


def pivot(A, x, idx):
    max_idx = idx
    for i in range(idx, A.shape[0]):
        if A[idx, i] > A[idx, max_idx]:
            max_idx = i
    if max_idx != idx:
        buff = np.copy(A[max_idx, :])
        A[max_idx, :] = A[idx, :]
        A[idx, :] = buff
        buffx = x[max_idx]
        x[max_idx] = x[idx]
        x[idx] = buffx
    return


def execute(A, b):
    assert A.shape[0] == A.shape[1], "A must be n * n"
    assert A.shape[0] == b.shape[0], "size of A and b must match"
    print("A:", A)
    print("b:", b)
    L = np.zeros_like(A)
    for i in range(A.shape[0]):
        L[i, i] = 1
    scale_ls = scale(A, b)
    print("Scaled A:", A)

    pivot(A, b, 0)
    for i in range(1, A.shape[0]):
        pivot(A, b, i)
        for j in range(i, A.shape[0]):
            coef = A[j][i - 1] / A[i - 1][i - 1]
            A[j, :] = A[j, :] - A[i - 1, :] * coef
            b[j] = b[j] - b[i - 1] * coef
```

```
            L[j, i - 1] = coef
    x = np.zeros_like(b)
    print("L:", L)
    for i in range(A.shape[0] - 1, -1, -1):
        x[i] += b[i]
        for j in range(i, A.shape[0] - 1):
            x[i] -= A[i][j + 1] * x[j + 1]
        x[i] /= A[i][i]
    print("x:", x)
    for i in range(A.shape[0]):
        A[i, :] = A[i, :] * scale_ls[i]

    return x, L, A


def single(A, b):
    print("-----single-----")
    A_copy = np.copy(A).astype(np.float32)
    b_copy = np.copy(b).astype(np.float32)
    fi32 = np.finfo(np.float32)
    print(fi32)
    x, L, U = execute(A_copy, b_copy)
    print("A * x:", A @ x.T)
    print("-----END-----")


def double(A, b):
    print("double")

    A_copy = np.copy(A).astype(np.float64)
    b_copy = np.copy(b).astype(np.float64)
    fi64 = np.finfo(np.float64)
    print(fi64)
    x, L, U = execute(A_copy, b_copy)
    print("A * x:", A @ x.T)
    print("-----END-----")


if __name__ == "__main__":
    A = np.array(
        [
            [10.0, 1.0, 4.0, 0.0],
            [1.0, 10.0, 5.0, -1.0],
            [4.0, 5.0, 10.0, 7.0],
            [0.0, -1.0, 7.0, 9.0],
        ],
    )
    b = np.array([15.0, 15.0, 26.0, 15.0])
    c = np.array([16.0, 16.0, 25.0, 16.0])

    single(A, b)
    double(A, b)
    single(A, c)
    double(A, c)
```

ソースコード 2　SOR 法

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns


def split_matrix(A):
    D = np.zeros_like(A)
    L = np.zeros_like(A)
    U = np.zeros_like(A)
    for i in range(A.shape[0]):
        D[i, i] = A[i, i]

    for i in range(A.shape[0]):
        for j in range(i):
            L[i, j] = A[i, j]

    for i in range(A.shape[0]):
        for j in range(i + 1, A.shape[0]):
            U[i, j] = A[i, j]
    print("D:", D)
    print("L:", L)
    print("U:", U)
    return D, L, U


def execute(A, b, w, x_true):
    assert A.shape[0] == A.shape[1], "A must be n * n"
    assert A.shape[0] == b.shape[0], "size of A and b must match"
    print("A:", A)
    print("b:", b)
    # D, L, U = split_matrix(A)
    x_prev = np.zeros_like(b)
    x = np.zeros_like(b)
    step_ls = []
    x_ls = []
    step = 0
    max_step = 300
    while step <= max_step:
        if np.linalg.norm(x - x_true) > np.power(10, 10):
            break
        if step % 10 == 0:
            step_ls.append(step)
            x_ls.append(np.log(np.linalg.norm(x - x_true)))
        x_prev = np.copy(x)
        for i in range(A.shape[0]):
            mida = 0
            midb = 0
            for j in range(i):
                mida += A[i, j] * x[j]
            for j in range(i + 1, A.shape[0]):
                midb += A[i, j] * x_prev[j]
            x[i] = (
                -w * mida + (1 - w) * A[i, i] * x_prev[i] - w * midb + w * b[i]
            ) / A[i, i]
        step += 1
    print("x:", x)
    return x, step_ls, x_ls
```

```python
def double(A, b, w, x_true):
    print("-----double-----")
    print("omega:", w)
    A_copy = np.copy(A).astype(np.float64)
    b_copy = np.copy(b).astype(np.float64)
    x, step_ls, x_ls = execute(A_copy, b_copy, w, x_true)
    print("A * x:", A @ x.T)
    print("-----END-----")
    return step_ls, x_ls


def plot_sor():
    sns.set_theme()
    A = np.array(
        [
            [1.0, -1.0, 0.0, 0.0],
            [-1.0, -2.0, -1.0, 0.0],
            [0.0, -1.0, 2.0, -1.0],
            [0.0, 0.0, -1.0, 2.0],
        ],
    )
    b = np.array([1.0, 0.0, 0.0, 0.0])
    x_true = np.array(
        [0.72727272727273, -0.27272727272727, -0.18181818181818, -0.090909090909091]
    )
    fig = plt.figure(figsize=(10, 10))
    for w in [0.1, 0.3, 0.5, 0.7, 0.9, 0.99, 1.0, 1.01, 1.1, 1.3, 1.5, 1.7, 1.9, 3, 5]:
        step_ls, x_ls = double(A, b, w, x_true)
        plt.plot(step_ls, x_ls, label="w:{:.2f}".format(w))

    plt.xlabel("step")
    plt.ylabel("ln|x-x_t|")
    plt.legend()
    plt.savefig("sor.png")


def plot_sor_debug():
    sns.set_theme()
    A = np.array(
        [
            [10.0, 1.0, 4.0, 0.0],
            [1.0, 10.0, 5.0, -1.0],
            [4.0, 5.0, 10.0, 7.0],
            [0.0, -1.0, 7.0, 9.0],
        ],
    )
    b = np.array([15.0, 15.0, 26.0, 15.0])

    x_true = np.array([1, 1, 1, 1])
    fig = plt.figure(figsize=(10, 10))

    for w in [0.1, 0.7, 1.0, 1.1, 1.5, 1.7, 1.8, 1.9]:
        step_ls, x_ls = double(A, b, w, x_true)
        plt.plot(step_ls, x_ls, label="w:{:.1f}".format(w))
```

```
    plt.xlabel("step")
    plt.ylabel("ln|x-x_t|")
    plt.legend()
    plt.savefig("sor_debug.png")


if __name__ == "__main__":
    plot_sor()
    plot_sor_debug()
```

参考文献

[1] 石原卓水島二郎. 理工学のための数値計算法. 2002.