2025 年度 科学技術計算

学生番号 36714143
情報工学科知能情報分野
和田小百合

2025 年 12 月 2 日

目次

1 課題 04-1

1.1 課題の内容

演習資料のコードを利用して，行列 A の LU 分解 $A = PLU$ を行う関数 l_decomposition() を作成しました。この関数は，行列 A を引数に取り、置換行列 P、下三角行列 L、上三角行列 U を返します。

1.2 PLU 分解について

PLU 分解は行列の連立方程式をコンピュータで解くときの解法の一つです。手計算で解くような解法 ($x = A^{-1}b$) は、逆行列を計算することになります。数学的には，行列の行列式が $\det(A) \neq 0$ であるならば，逆行列 A^{-1} が存在します。つまり、連立方程式を解く前に行列 A の行列式が 0 でないかどうかを確認する必要があります。しかし、実際に数値計算を行う際には、行列式計算における浮動小数点数に起因する誤差があり、そのためオーバーフローしない数値的に安定な計算方法が必要です。さらに，行列式の計算は非常に計算コストがかかることが多いため、行列式を求める際には計算量にも配慮する必要があります。よって、丸め誤差などで正しい値が出力される可能性が低いため、PLU 分解という入力行列 A に対して分解を行い逆行列を計算しない連立方程式をコンピュータ上で計算して厳密解を出力しています。

1.3 関数について

PLU 分解を行う関数 (lu_decomposition) は以下のように制作しました。

```
def lu_decomposition( A_org: np.ndarray ) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """LU decomposition of A = PLU

    Args:
        A (np.ndarray): nxn matrix A

    Returns:
        (np.ndarray): nxn permutation matrix P
        (np.ndarray): nxn lower matrix L with 1s in diagonal
        (np.ndarray): nxn strictly upper matrix U
    """
    assert A_org.ndim == 2
    assert A_org.shape[0] == A_org.shape[1]

    #前進消去(分解PLU)を行うコード
    n = A_org.shape[0]
    A = A_org.copy()
    L = np.eye(n)    #単位行列生成
    P = np.eye(n)    #置換行列

    for j in range(n - 1):
        #ここから資料にあった部分ピポッド選択月も前進消去の実装
        #  部分ピボット選択
        j_max = np.abs(A[j:, j]).argmax() + j    #行中の一番大きい値を抽出
        if j != j_max:
            # A の行交換
            A[[j, j_max]] = A[[j_max, j]]
```

```
28
29              # L の行交換（前の列のみ）
30              if j > 0:
31                  L[[j, j_max], :j] = L[[j_max, j], :j]
32
33              # P の行交換
34              P[[j, j_max]] = P[[j_max, j]]
35
36          #print("A\n", A, sep="")
37          #print("P\n", P)
38
39          # 前進消去
40          remaining_rows = n - (j + 1)
41          r_ij = A[j + 1:, j] / A[j, j]
42          A[j + 1:, j + 1:] -= np.tile(A[j, j + 1:], (remaining_rows, 1)) * r_ij.
                reshape(remaining_rows, -1)
43          A[j + 1:, j] = 0
44          L[j + 1:, j] = r_ij
45          #print("A\n", A, sep="")
46          #$print("L\n", L, sep="")
47
48      U = A     #完成された変形したが上三角行列A
49
50      P = P.T  # 自作の分解は「LUPA = 」という形式でLU P を作っている。一方 scipy.linalg.lu は
            「A = P L 」を返すので、両者のU P を揃えるために転置して合わせている。
51
52      return P, L, U
```

基本的な PLU 分解は講義資料に書かれていた通りの前進消去をもとに製作しました。また、置換行列 P については、自作の LU 分解は PA = LU の形式で P を作成しているため、scipy.linalg.lu の出力形式 (A = PLU) と一致させるため転置して合わせています。

1.4 実行コード/実行結果

行列サイズを 2-10 の入力行列 A を生成して、その行列を lu_decomposition(A_org: np.ndarray) を用い PLU 分解したものがライブラリ scipy と一致しているか確認するコードです。

```
1   for i in tqdm(range(50)):
2     n = rng.integers(low = 2, high = 10) #サイズ2-10
3     A = rng.random(size=(n, n))    #行列生成
4
5     P, L, U = lu_decomposition(A)
6     Psp, Lsp, Usp = scipy.linalg.lu(A)
7
8     assert np.allclose(L, Lsp), "values don't match"
9     assert np.allclose(U, Usp), "values don't match"
10    assert np.allclose(P, Psp), "values don't match"
```

出力結果は error"values don't match"が出てこなかったため、すべての行列が scipy と一致していることがわかる。また、課題でランダムに行列を生成すると，正則でない行列が生成される可能性がある。その場合、LU 分解がうまくいかないことがあるため、結果が一致しない場合，バグによるものか，行列の性質によるものかを考慮せよ。という課題があったが、10 回以上同じコードを実行したがエラーが出てこなくわかりませんでした。

2 課題 04-2

2.1 課題の内容

課題 4-1 で作成した関数 lu_decomposition() を用いて、連立方程式 $A\boldsymbol{x} = \boldsymbol{b}$ を解く関数 solve_lu() を実装しました。

2.2 関数について

連立方程式を解く関数 (solve_lu()) は以下のように実装しました。

```python
def solve_lu( P: np.ndarray, L: np.ndarray, U: np.ndarray, b: np.ndarray,) -> np.ndarray:
    """Solve Ax = b with PLU, LU decomposition of A = PLU

    Args:
        P (np.ndarray): nxn permutation matrix P
        L (np.ndarray): nxn lower matrix L with 1s in diagonal
        U (np.ndarray): nxn strictly upper matrix U
        b (np.ndarray): n-d rhs vector b

    Returns:
        (np.ndarray): solution x of Ax = PLUx = b
    """
    assert b.ndim == 1
    n = b.shape[0]
    assert P.shape == (n, n)
    assert L.shape == (n, n)
    assert U.shape == (n, n)
    b = b.copy()

    # solve Ly = P^T b
    a = P.T @ b  #Ly = と置いているa
    # 前進代入
    y = np.zeros(n)
    for i in range(n):
        y[i] = a[i] - np.dot(L[i, :i], y[:i])
        # L[i,i] = （単位下三角行列）なので割り算不要1

    # 後退代入
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = (y[i] - np.dot(U[i, i+1:], x[i+1:])) / U[i, i]

    return x
```

この関数は、lu_decomposition() の出力である置換行列 P、下三角行列 L、上三角行列 U、およびベクトル $\boldsymbol{b}$ を引数に取り、連立方程式の解 $\boldsymbol{x}$ を返す。LU 分解が与えられるた、計算には前進代入と後退代入をすることで x を出力しました。

2.3 実行コード/実行結果

実行コードは以下の通りです。

```
  for i in tqdm(range(50)):
    n = rng.integers(low=2, high=10)
    A = rng.random(size=(n, n))
    b = rng.random(n)

    P, L, U = lu_decomposition(A)
    x_with_lu = solve_lu(P, L, U, b) #こっちが自作

    x_numpy = np.linalg.solve(A, b) #こっちがライブラリにあるやつ

    assert np.allclose(x_numpy, x_with_lu ), "values don't match"
```

出力は eeror が出なかったため np.linalg.solve() と一致していることがわかります。

3 課題 04-4

3.1 課題の内容

lu_decomposition() の結果 $A = PLU$ を用いて、行列式を計算する関数 det_lu() を作りました。

3.2 行列式の計算と P の計算について

$A = PLU$ より，行列式は $\det(A) = \det(P)\det(L)\det(U)$ と変形することができる。P の行列式は課題の資料より計算することができます。L は単位下三角行列なので $\det(L) = 1$ です。

3.3 行列 U の計算方法について

U は上三角行列なので $\det(U)$ は U の対角要素の積である。アンダーフロー・オーバーフローに配慮した実装を行うため、対数を用いて $\log|U_{ii}|$ を計算し，符号とともに最後に exp で復元する。なお U_{ii} が負の場合にも対処するために，符号を別に計算します。U の行列式を含む A の行列式を計算する関数 lu_decomposition() は以下のようになります。

```
  #の行列式計算行う関数P
def det_P(A: np.ndarray) -> float:
    """det(P) of permutation matrix P
    Args:
        P (np.ndarray): nxn permutation matrix of 0 and 1

    Returns:
        float: det(P) that is either 1 or -1
    """

    assert P.ndim == 2
    assert P.shape[0] == P.shape[1]

    for p in P.ravel()[np.nonzero(P.ravel())]:
        assert np.isclose(p, 1.0)
```

```
16
17      n = P.shape[0]
18      assert np.allclose(P.sum(axis=0), np.ones(n))
19      assert np.allclose(P.sum(axis=1), np.ones(n))
20
21      ## below three lines of code are helped with ChatGPT-4o because of its
            complexisity
22
23      # Extract the permutation using np.nonzero
24      permutation = np.nonzero(P)[1]
25
26      # Calculate the number of inversions in the permutation
27      inversions = sum(1 for i in range(len(permutation)) for j in range(i + 1,
            len(permutation)) if permutation[i] > permutation[j])
28
29      # Return 1 if inversions are even, otherwise -1
30      return 1 if inversions % 2 == 0 else -1
31
32
33  #の行列式を分解を用いて行列式の計算行う関数APLU
34  def det_lu( A_org: np.ndarray,) -> float:
35      """compute det(A)
36      Args:
37          A (np.ndarray): nxn square matrix
38
39      Returns:
40          (float): det(A)
41      """
42      A = A_org.copy()
43      assert A.ndim == 2
44      assert A.shape[0] == A.shape[1]
45      n = A.shape[0]
46
47      P, L, U = lu_decomposition(A)     #分解を行うPLU
48      signP = det_P(P)    #の行列式の計算P
49
50      #の対角成分の積U
51      diagU = np.diag(U) #対角成分だけとる
52
53      # 符号の計算
54      log_symbol_U = np.prod(np.sign(diagU))  #prod 掛け合わせ sign 符号
55
56      # 対角絶対値の log の和
57      log_abs_U = np.sum(np.log(np.abs(diagU))) #をとるlog
58
59      signU = log_symbol_U * np.exp(log_abs_U)
60      detA = signP * signU
61
62      return detA
```

3.4 実行結果について

以下のように自作関数と solve() で行列式計算の比較を行いました。

```
  #実行
for i in tqdm(range(10)):
    n = rng.integers(low=2, high=5)
    B = rng.random(size=(n, n))

    #行列式計算自作関数()
    det_A = det_lu(A)
    # numpy の行列式計算
    A_numpy_det = np.linalg.det(A)
    assert np.isclose( det_A, A_numpy_det), "values don't match"
```

結果としては error が出ていないため、自作の関数があっていることがわかりました。

4 課題 04-7

4.1 課題の内容

連立方程式を求める SOR 法の関数を実装し、同じ計算方法 (反復法) である Jacobi 法や Gauss-Seidel 法と比較を行いました。

4.2 SOR 法について

Gauss-Sidel 法を改良したものが SOR 法です。SOR 法では下の式で解を更新します。

$$\tilde{\boldsymbol{x}}^k = D^{-1}(\boldsymbol{b} - L\boldsymbol{x}^k - U\boldsymbol{x}^{k-1})$$
$$\boldsymbol{x}^k = \boldsymbol{x}^{k-1} + \omega(\tilde{\boldsymbol{x}}^k - \boldsymbol{x}^{k-1})$$

この式をもとに関数を作ったコードが以下のようになります。

```
def sor(A: np.ndarray, b: np.ndarray, x0: np.ndarray | None = None, omega:
    float = 1.5, maxiter: int = 80, tol: float = 1e-8, callback: callable = None
    ) -> np.ndarray:
    """SOR method for solving Ax=b
    Args:
        A (np.ndarray): nxn matrix A
        b (np.ndarray): n-d vector b
        xk (np.ndarray): n-d vector of initial value x0
        omega (float): omega of SOR
        maxiter (int, optional): max iterations. Defaults to 200.
        tol (float, optional): tolerance. Defaults to 1e-8.
        callback (callable, ``callback(diff: float, norm: float)``, optional):
            callback function. Defaults to None.

    Returns:
        (np.ndarray): n-d vector of the solution x
    """
    assert b.ndim == 1
    n = b.shape[0]
    assert A.shape == (n, n)
```

```
18
19      if x0 is None:
20          x0 = np.zeros(n) #定義されていない場合ベクトルを生成する。
21      else:
22          assert x0.shape == (n,)
23      xk_1 = x0.copy()
24      x_tilde_k = x0.copy()
25      xk = x0.copy()
26
27      D = diag(A) #対角行列
28      LU = A.copy()
29      np.fill_diagonal(LU, 0) #狭義上三角行列と狭義した三角行列の和
30
31      for _ in range(maxiter):
32          for i in range(n):
33              x_tilde_k[i] = ( b[i] - (LU[i, :i] @ xk[:i] + LU[i, (i + 1):] @
                    xk_1[(i + 1):])) / D[i]
34              xk[i] = xk_1[i] + omega * (x_tilde_k[i] - xk_1[i])
35
36          diff_k = norm(xk - xk_1)
37          residual_k = norm(b - A @ xk)
38
39          if callback:
40              callback(diff_k, residual_k)
41
42          if diff_k < tol:
43              break
44          xk_1 = xk.copy()
45      return xk
```

$$\tilde{x}^k = D^{-1}(b - Lx^k - Ux^{k-1}$$

の部分は Gauss-Seidel の式の資料をもとに作りました。

4.3 solve と自作関数の比較

以下のように numpy の solve() と一致しているか確認しました。

```
1   for i in range(10):
2     n = rng.integers(low=2, high=1000)
3     A = rng.random(size=(n, n))
4     A = A.T @ A
5     np.fill_diagonal(A, A.sum(axis=1))  #優対角
6     b = rng.random(n)
7
8
9     x_numpy = np.linalg.solve(A, b)
10
11    cache_callback_sor = CacheDiffRes("SOR")
12    x_mysor = sor(A, b,callback=cache_callback_sor)
13
14    # print(f"x_mysor = {x_mysor}")
```

```
    # print(f"x_numpy = {x_numpy}")
    assert np.allclose(x_numpy, x_mysor), "values doesn't match"
```

行列のサイズは 2 から 1000 で連立方程式の計算を行いました。また、行列 A を優対角にしている理由は、反復法の収束条件が、A が優対角行列または正定値対称行列ならば収束する。行列 A が優対角である（もしくは対角優位である）とは，次式が成り立つことです。

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

これは対角成分がその行の非対角成分の和よりも大きい場合であり，対角行列は優対角である。実行結果としては有対角行列において error が出ていないため、numpy の関数と一致することがわかりました。一方で優対角行列ではない行列を自作関数 SOR で計算すると実際の値よりとてつもなく大きい数字が出てしまうことがわかりました。これは収束することができず発散してしまったと考えています。

4.4 加速係数 ω について

SOR 法では加速係数によって誤差の収束速度が変わります。ω の値の変化による誤差の収束速度をグラフにしたものが以下のようになります。行列のサイズは 10 で固定して計算をおこなっています。

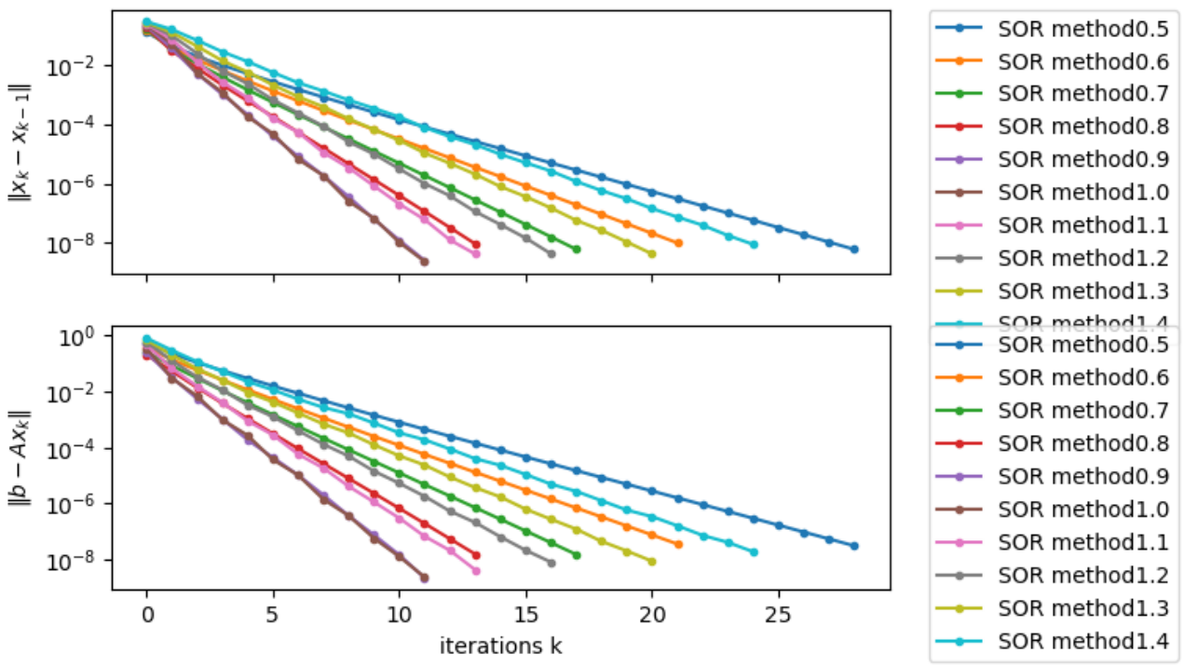


図 1　ω を 0.1 刻みで変化させた図

ω が 0.5 から 1.4 までプロットしたときに 0.5 から 1.0 にかけて収束速度が速くなりそれ以降はおそおくなることがわかります。これをもとに ω さらに細かく 0.05 刻みでプロットしたものが以下のグラフになります。

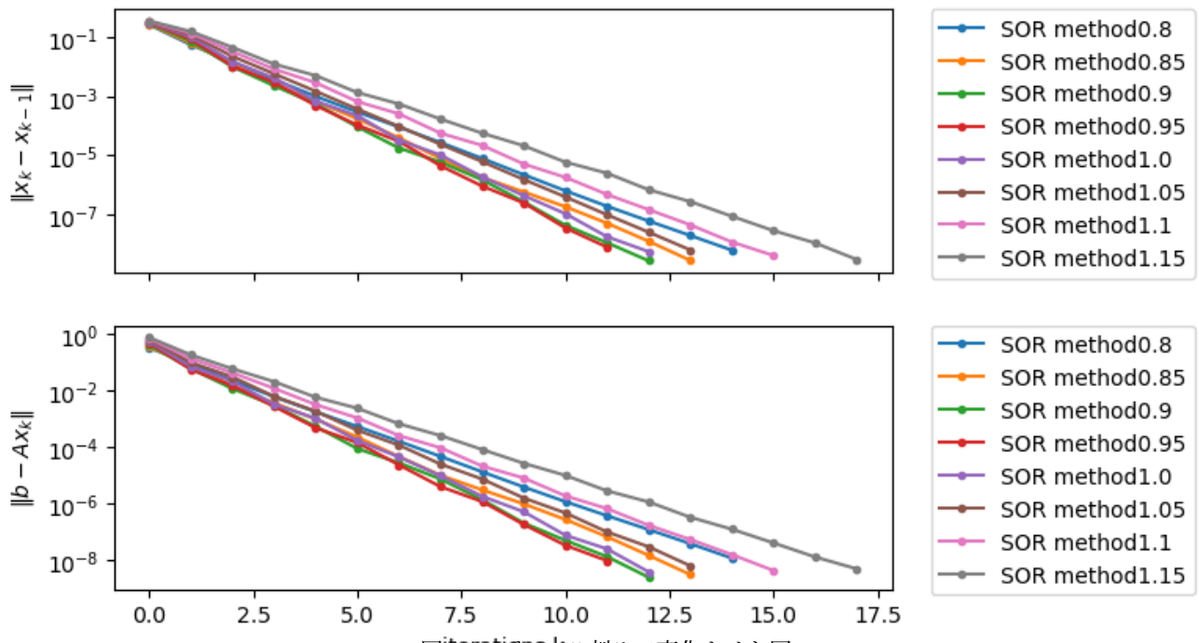


図 2　ω を 0.05 刻みで変化させた図

さらに見やすくするため ω を 0.8 から 1.15 までをグラフにしました。このグラフから、1.0(Gauss-Seidel 法) より週速度が速いものは 0.9,0.95 であることがわかります。さらに詳しく見るために ω を 0.01 刻みでプ

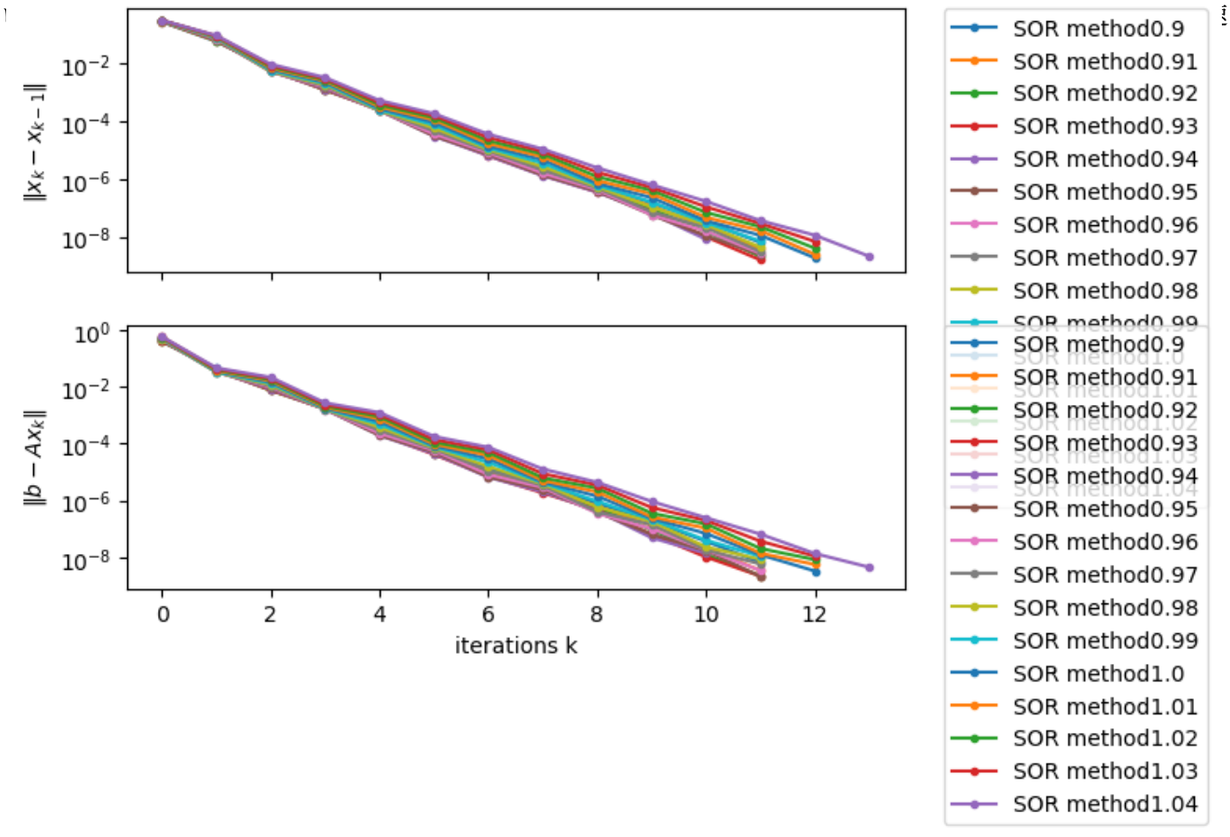


図 3　ω を 0.05 刻みで変化させた図

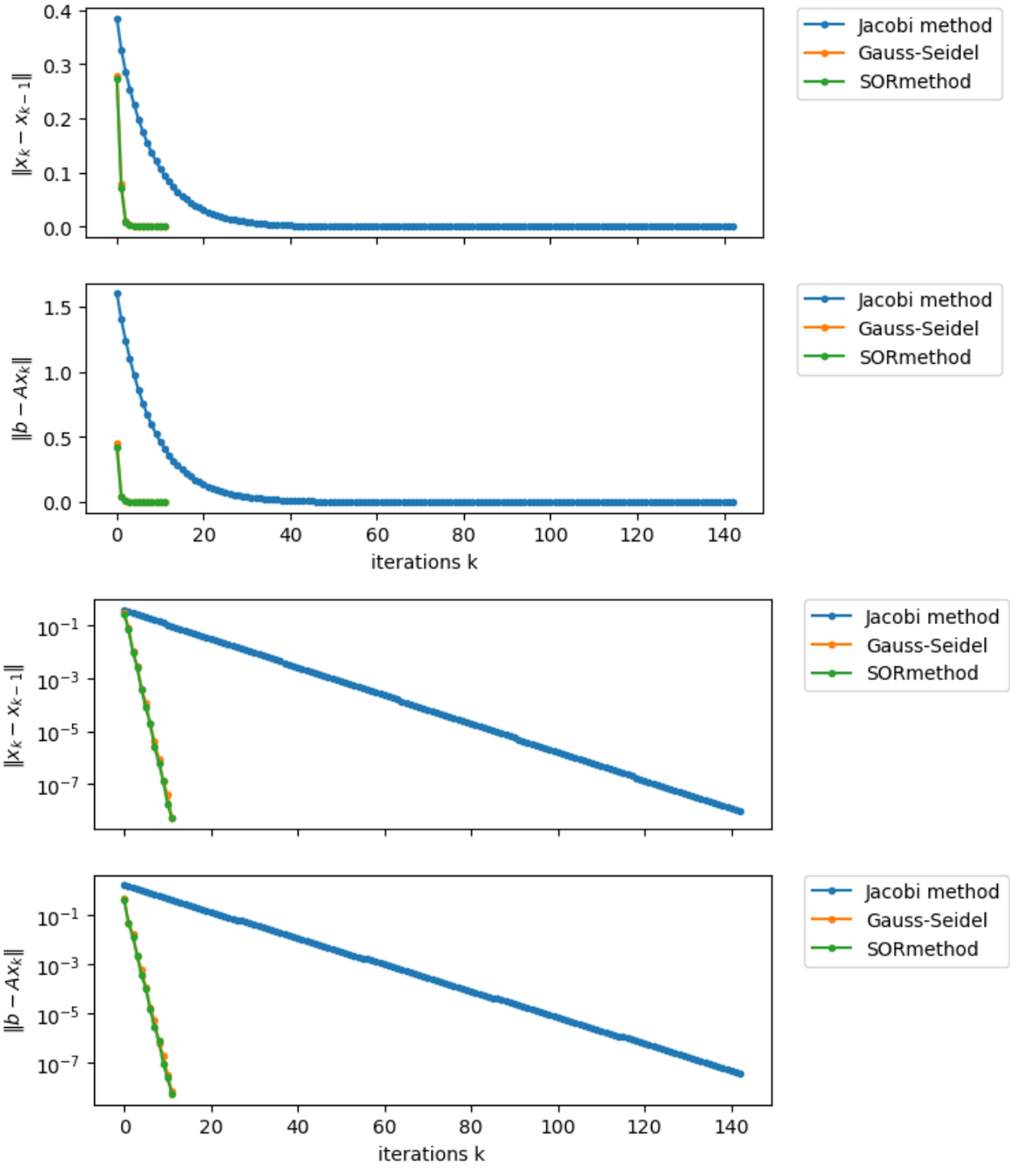
度が速いことがわかる。

4.5　Jacobi 法や Gauss-Seidel 法との比較

実行コードは以下のようになります。行列のサイズは 10、ω の値は 0.97 と設定しました。

```
  A = rng.random(size=(10, 10))
np.fill_diagonal(A, A.sum(axis=1))  #優対角
b = rng.random(10)

cache_callback_SOR = CacheDiffRes("SORmethod")
x_SOR = sor(A, b,omega = 0.97,  callback=cache_callback_SOR)

cache_callback_jacobi = CacheDiffRes("Jacobi method")
x = jacobi_method(A, b, callback=cache_callback_jacobi)

cache_callback_gauss_seidel = CacheDiffRes("Gauss-Seidel")
x = gauss_seidel(A, b, callback=cache_callback_gauss_seidel)


plot_nya = PlotCacheDiffRes(
    [cache_callback_jacobi, cache_callback_gauss_seidel, cache_callback_SOR]
)

fig = plot_nya.plot()
fig = plot_nya.plot(y_logscale=True)
```

このコードを実行し得られたグラフが以下のようになります。この結果から ω の値によって変化はするも



のの誤差の収束速度は SOR 法、Gauss-Seidel 法、Jacobi 法の順であることがわかりました。