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 11 月 17 日

目次

1　課題 03-1

1.1　課題の内容

　自作関数とライブラリの比較
python では、numpy の組み込み演算子 @ によって、ベクトル行列の積、ノルムの計算ができます。実際に手計算で用いる方法をもとに作成した自作の関数をつくり、自作関数とライブラリではどれほど実行時間に差があるかを検証しました。

1.2　自作関数について

　今回は行列同士の積である my_mm() 関数を作りました。以下が関数のコードです。

```
#行列の積の計算する関数

def my_mm(A: np.ndarray, B: np.ndarray) -> np.ndarray:
    """行列積 を計算する関数AB引数

    :
        A (ndarray): m x n matrix
        B (ndarray): n x p matrix

    Returns:
        (ndarray): m x p matrix AB
    """
    #a,b が二次元行列であることを確認

    assert A.ndim == 2 and B.ndim == 2
    assert A.shape[1] == B.shape[0]

    #サイズを読み出し
    print("A\n", A)
    print("B\n", B)

    m, n = A.shape
    n, p  = B.shape

    C = np.zeros ((m, p))

    # 4*3 の行列の積を計算3*4
    for i in range (0, m):
        for j in range (0, p):
            for k in range (0, n):
                C[i][j] += A[i][k] * B[k][j]

    print("C\n", C)

    return C
```

Listing 1　課題 03-1 行列同士の積を計算する自作関数

計算方法は手計算と同じように A の行の要素と B の列の要素をかけ合わせています。このコードでは、3 重

にの for 文があるため、A の行列のサイズを m * n B の行列のサイズを n * p とすると O(mnp) となり、とてつもない実行時間になることが考えられます。また、この関数がライブラリにある関数と一致するかどうかは以下のコードで検証しました。

```
for i in range(5):
  #行列のサイズ
  m = rng.integers(low=2, high=1000)
  n = rng.integers(low=2, high=1000)
  p = rng.integers(low=2, high=1000)
  A = rng.random(size=(m, n))
  B = rng.random(size=(n, p))

  print("A\n" ,A)
  print("B\n" ,B)

  #ライブラリの計算
  numpy_AB = A @ B

  #自作関数による計算
  my_AB = my_mm(A, B)
  print("my_AB\n" ,my_AB)

  assert np.allclose( #一致
      numpy_AB, my_AB
  ), "values doesn't match"
```

Listing 2　自作関数とライブラリ関数が一致しているか確認するコード

このコードでは、行列 A, B に対してランダムでサイズと要素を設定し、積を求めるコードです。また、ライブラリと自作の関数が一致しなかった場合はエラーメッセージ"values doesn't match"と表示されるよう担っています。

1.3　実行結果について

実行結果以下のようになります。(一部)

$$A = \begin{bmatrix} 0.70974492 & 0.79651253 & 0.84954348 & \dots & 0.66854599 & 0.72500591 & 0.78944874 \\ 0.25122687 & 0.64723943 & 0.0972914 & \dots & 0.74355117 & 0.64713535 & 0.18485971 \\ 0.04402379 & 0.66377313 & 0.90496276 & \dots & 0.90931646 & 0.61013497 & 0.73291278 \\ \multicolumn{7}{c}{\dots} \\ 0.4885968 & 0.77509604 & 0.57938034 & \dots & 0.15688472 & 0.78496526 & 0.97638023 \\ 0.69416792 & 0.79729174 & 0.78738776 & \dots & 0.42972533 & 0.55685229 & 0.90223696 \\ 0.42809744 & 0.11909929 & 0.03933435 & \dots & 0.49868688 & 0.09877086 & 0.40503104 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.83804628 & 0.44177747 & 0.81277235 & \dots & 0.54226729 & 0.27328268 & 0.83025194 \\ 0.17007421 & 0.76381704 & 0.28155452 & \dots & 0.80159514 & 0.07905584 & 0.15211501 \\ 0.33229212 & 0.03311804 & 0.64202386 & \dots & 0.67368798 & 0.40483224 & 0.53516115 \\ \multicolumn{7}{c}{\dots} \\ 0.65906175 & 0.45434139 & 0.24773743 & \dots & 0.87422966 & 0.80044577 & 0.56448645 \\ 0.53226811 & 0.22774403 & 0.7979622 & \dots & 0.01226688 & 0.47532615 & 0.01150642 \end{bmatrix}$$

$$my\_AB = \begin{bmatrix} 114.2765884 & 119.62820304 & 113.76999405 & \dots & 118.76371431 & 111.07872293 & 118.06790006 \\ 113.61134799 & 116.21301428 & 114.42227599 & \dots & 115.47155127 & 108.25191373 & 115.92286384 \\ 113.64554776 & 116.21137796 & 115.07647386 & \dots & 120.252282 & 106.04939983 & 115.9337235 \\ \dots & & & & & & \\ 115.55854973 & 119.56462886 & 109.05837756 & \dots & 120.69881438 & 112.6311495 & 116.03741038 \\ 109.39942515 & 113.0417849 & 107.49366562 & \dots & 116.12619225 & 100.99115659 & 110.76058338 \\ 113.38235227 & 117.64312228 & 115.75947469 & \dots & 120.82544618 & 115.48086097 & 121.30729031 \end{bmatrix}$$

$$A = \begin{bmatrix} 0.88113884 & 0.15077192 & 0.52805537 & \dots & 0.08432179 & 0.90704737 & 0.14581677 \\ 0.74361463 & 0.76712877 & 0.95864335 & \dots & 0.54245853 & 0.22738509 & 0.04675882 \\ 0.16493641 & 0.60140535 & 0.78237793 & \dots & 0.09241676 & 0.59398118 & 0.71721253 \\ \dots & & & & & & \end{bmatrix}$$

このように出力された行列 A, B について行列の積を出力した後エラーメッセージなく、for 文を回っているためライブラリ関数と自作関数が一致していることがわかります。

1.4 実行時間について

実行時間絵を計算するコードは以下のように設定しました。

```
for i in range(0, 5):
    #行列のサイズ
    m = [2,5,10, 100,1000]
    n = [2,5,10, 100,1000]
    p = [2,5,10, 100,1000]
    A = rng.random(size=(m[i], n[i]))
    B = rng.random(size=(n[i], p[i]))

    #実行時間の計測
    %timeit numpy_inner_product = A @ B
    %timeit my_inner_product = my_mm(A, B)
```

行列のサイズを 2, 5, 10, 100, 1000、要素はランダムである行列を生成しました。そして、これらの行列の積の計算を行いました。これをもとにグラフを出力しました。
以下が実行時間を比較したグラフです。

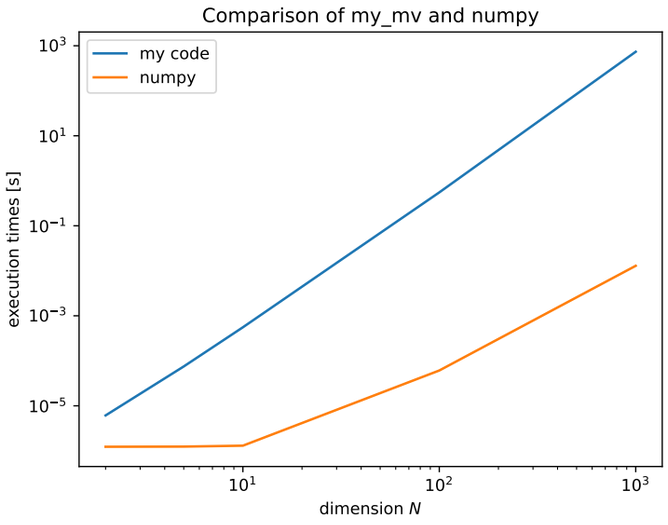

図 1　行列の積を計算する自作関数とライブラリの関数の実行時間の比較

グラフのように自作関数は、行列のサイズが大きくなればなるほど実行時間が指数関数的に遅くなる一方で、ライブラリは比較的なだらかな実行時間の変化であることがわかりました。

2 課題 03-3

2.1 課題の内容

CSR 形式から密行列への変換を自前で実装し、一致するかどうか検証した

2.2 疎行列 / 密行列とは

疎行列 (sparse matrix) は、行列の大部分の要素が 0 で、非ゼロ要素が少ない行列です。大規模な問題を扱うときには，疎行列の特性を生かして計算を効率化することが多い。
密行列 (dense matrix) とは、行列の要素の大部分が非ゼロである行列を指します。数値計算において一般的な行列は密行列であり，通常はそのまま計算することが多い。しかし、密行列が大規模になると、メモリを大量に消費し、計算にも膨大な時間を要することがあります。

2.3 CSR 形式とは

CSR(Compressed Sparse Row) 形式は、行方向に非ゼロ要素を格納する形式であり。行列とベクトルの積を効率的に計算するために適しています。CSR 形式は、以下の 3 つの配列で表現されます。

indptr: 各行の非ゼロ要素の開始位置を示す配列。
indices: 非ゼロ要素が存在する列のインデックスを示す配列。
data: 非ゼロ要素の実際の値を格納する配列。

この三つの配列によって非ゼロの値がどの位置に入っているかの情報がわかります。

2.4 CSR 形式から密行列の変換の実装方法

以下のコードが CSR 形式から密行列の変換を行う関数です。

```
  # 形式の行列をもとの式に変換する関数の作成CSR
def sparse_to_dense(A: scipy.sparse._csr.csr_array) -> np.ndarray:
    indptr = A.indptr
    indices = A.indices
    data = A.data

    #元の配列のサイズを特定したい
    n_rows, n_cols = A.shape
    answer = np.zeros((n_rows, n_cols))

    for i in range (0, len(indptr) - 1 ):
        for j in range (indptr[i],indptr[i + 1]):
            answer[i][indices[j]] = data[j]

    return answer
```

csr 形式の行列から 3 つの要素 (indptr, indices, data) を読みだします。次にもともとの配列のサイズを CSR 形式の行列 A から読み取り、すべての要素を 0 にします。その後、CSR 形式の変換と同様に SCR 形式の要素の配列から非ゼロの値がもともとどこにあったかを確認し、代入します。

2.5 実行結果

以下が実行コードです。

```
  #ランダム生成する
a = scipy.sparse.random_array((5, 5), density=0.2, )
print(a)
print()

# 形式に変換CSR
A_csr = scipy.sparse.csr_array(a)


#戻す
answer = sparse_to_dense(A_csr)
#コンピュータ側のやつ
true_dense = a.toarray()


print("自作関数の結果：\n", answer)
print()
print("の結果：SciPy\n", true_dense)


#結果を比較
print("一致しているか? →", np.allclose(answer, true_dense))
```

結果は以下のようになりました。

```
自作関数の結果：
 [[0.         0.77908705 0.         0.         0.        ]
 [0.         0.         0.         0.85851858 0.        ]
 [0.         0.         0.         0.         0.54087523]
 [0.         0.         0.         0.         0.        ]
 [0.26877557 0.         0.         0.         0.95604159]]

SciPyの結果：
 [[0.         0.77908705 0.         0.         0.        ]
 [0.         0.         0.         0.85851858 0.        ]
 [0.         0.         0.         0.         0.54087523]
 [0.         0.         0.         0.         0.        ]
 [0.26877557 0.         0.         0.         0.95604159]]
一致しているか? → True
```

図 2　SCR 形式変換し元に戻した自作行列と scipy の変換行列

3　課題 03-4

3.1　課題の内容

行列 A が与えられたとき，その行列が単位上三角行列か、または上ヘッセンベルク行列かどうか判定する関数を作りました。それぞれ、単位上三角行列 (is_unit_upper_triangular())、上ヘッセンベルグ行列 (is_upper_hessenberg()) を作成しました。判定は True,False で返しました。

3.2　単位上三角行列について

単位上三角行列は以下のような関数を作成しました。

```
  #単位上三角行列判定
def is_upper_triangular( A: np.ndarray, atol: float = 1e-8, rtol: float = 1e
    -8,) -> bool:
    #要するに対角がで下の要素が全部であればいい1,0
    #対角がかをまず判定1
    if(np.allclose(np.diag(A), 1, atol, rtol)):
        for i in range(1, n):
            if not(np.allclose(np.diag(A, k = -i), 0, atol, rtol)):
                return False
        else:
                return True
    else:
        return False
```

まず対角成分が 1 かどうかを判別します。そのあとに劣対角成分がが 0 かどうかを判定します。劣対角成分を判定する関数 (diag(i,k =)) を用いて k の値を変えることによって、対角線上に 0 かどうか判定しました。これらの条件をすべて満たしたときに True を返すように条件分岐をしました。

3.3　上ヘッセンベルク行列について

上ヘッセンベルク行列について以下のような関数を作成しました。

```
  #上ヘッセンベルク行列判定
def is_upper_hessenberg( A: np.ndarray, atol: float = 1e-8, rtol: float = 1e
    -8,) -> bool:
    #要するに対角い越した以下がであればいい0
    for i in range(2, n):
        if not(np.allclose(np.diag(A, k = -i), 0, atol, rtol)):
            return False
    else:
        return True
```

上ヘッセンベルク行列は劣対角成分まで要素がありそれ以下の要素が 0 である必要があります。なので、diag(i,k =) の値を i = 2 からにすることで判定しました。

3.4 実行結果

多くの行列を生成し、関数に読み込んで判定しました。

```
#実行
#ランダムのやーつ
n = 5
A = rng.random(size=(n, n))
print("A\n", A)
print("は単位上三角行列かA ->",is_upper_triangular(A))
print("は上ヘッセンベルク行列かA ->",is_upper_hessenberg(A))
print()

#上三角行列を生成
A_tri_upper = np.triu(A)
print("upper triangular matrix\n", A_tri_upper)
print("は単位上三角行列かA ->",is_upper_triangular(A_tri_upper))
print("は上ヘッセンベルク行列かA ->",is_upper_hessenberg(A_tri_upper))
print()

#わかりやすい行列だよ
B = np.array([[1,2,3,4,5],
      [0,1,3,4,5],
      [0,0,1,4,5],
      [0,0,0,1,5],
      [0,10,0,0,1]])
print("B matrix\n",B)
print("は単位上三角行列かB ->",is_upper_triangular(B))
print("は上ヘッセンベルク行列かB ->",is_upper_hessenberg(B))
print()

#上三角行列を生成
A_tri_tani_upper = np.triu(A)
np.fill_diagonal(A_tri_tani_upper, 1)
print("upper tani triangular matrix\n", A_tri_tani_upper)
print("は単位上三角行列かA ->",is_upper_triangular(A_tri_tani_upper))
print("は上ヘッセンベルク行列かA ->",is_upper_hessenberg(A_tri_tani_upper))
print()

#上ヘッセンベルクだよ
A_upper_hessenberg = np.triu(A, k=-1)
print("upper Hessenbrerg matrix\n", A_upper_hessenberg)
print("は上三角行列かA ->\n",is_upper_triangular(A_upper_hessenberg))
print("は上ヘッセンベルク行列かA ->",is_upper_hessenberg(A_upper_hessenberg))
```

結果はこのようになります。

```
  A
 [[0.36120944 0.09708664 0.51093833 0.03046645 0.38993994]
 [0.41209796 0.98835302 0.14855196 0.80909799 0.128218  ]
 [0.84798785 0.50746525 0.87033755 0.93491136 0.44816159]
 [0.48530231 0.71511643 0.68474405 0.66940809 0.46289694]
 [0.2114649  0.76787082 0.48361928 0.42200012 0.76178458]]は単位上三角行列か
A -> Falseは上ヘッセンベルク行列か
```

```
A -> False

upper triangular matrix
 [[0.36120944 0.09708664 0.51093833 0.03046645 0.38993994]
 [0.          0.98835302 0.14855196 0.80909799 0.128218  ]
 [0.          0.          0.87033755 0.93491136 0.44816159]
 [0.          0.          0.          0.66940809 0.46289694]
 [0.          0.          0.          0.          0.76178458]]は単位上三角行列か
A -> Falseは上ヘッセンベルク行列か
A -> True

B matrix
 [[ 1  2  3  4  5]
 [ 0  1  3  4  5]
 [ 0  0  1  4  5]
 [ 0  0  0  1  5]
 [ 0 10  0  0  1]]は単位上三角行列か
B -> Falseは上ヘッセンベルク行列か
B -> False

upper tani triangular matrix
 [[1.          0.09708664 0.51093833 0.03046645 0.38993994]
 [0.          1.          0.14855196 0.80909799 0.128218  ]
 [0.          0.          1.          0.93491136 0.44816159]
 [0.          0.          0.          1.          0.46289694]
 [0.          0.          0.          0.          1.         ]]は単位上三角行列か
A -> Trueは上ヘッセンベルク行列か
A -> True

upper Hessenbrerg matrix
 [[0.36120944 0.09708664 0.51093833 0.03046645 0.38993994]
 [0.41209796 0.98835302 0.14855196 0.80909799 0.128218  ]
 [0.          0.50746525 0.87033755 0.93491136 0.44816159]
 [0.          0.          0.68474405 0.66940809 0.46289694]
 [0.          0.          0.          0.42200012 0.76178458]]は上三角行列か
A ->
 Falseは上ヘッセンベルク行列か
A -> True
```

すべて結果があっていました。

4　課題 03-5

4.1　課題の内容

行と列のそれぞれについて定数倍・置換・線形和の 3 種類を非破壊版と in-place 版で作成しました。in-place 版とは，引数に与えた元の変数の値そのものを直接書き換える操作を指します。非破壊版は，引数に与えた元の変数の値をコピーしてから処理を行い、そのコピーを返す方式である。

4.2　定数倍の関数について

行列のそれぞれの定数倍する関数は以下になります。

```
  def row_scale(A: np.ndarray, i: int, alpha: float) -> np.ndarray:
  """行を定数倍する関数（非破壊版）変更した行列を返す行列は変更されないA"""
  B = A.copy()      # コピーを作る
  I_ic = np.eye(len(A))
  I_ic[i, i] = alpha
  """
  print("I_ic\n", I_ic)
  print("I_ic A\n", I_ic @ A)  # 行 i を c 倍
  """
  B = I_ic @ A
  return B

def row_scale_inplace(A: np.ndarray, i: int, alpha: float) -> None:
  """行を定数倍する関数（in-版）行列を直接変更するplaceA 返り値を持たない"""
  A[i] *= alpha
  return None

def col_scale(A: np.ndarray, i: int, alpha: float) -> np.ndarray:
  """列を定数倍する関数（非破壊版）変更した行列を返す行列は変更されないA"""
  B = A.copy()      # コピーを作る
  I_ic = np.eye(len(A))
  I_ic[i, i] = alpha
  """
  print("I_ic\n", I_ic)
  print("A I_ic\n", A @ I_ic)  # 列 i を c 倍
  """
  B = A @ I_ic
  return B


def col_scale_inplace(A: np.ndarray, i: int, alpha: float) -> None:
  """列を定数倍する関数（in-版）行列を直接変更するplaceA 返り値を持たない"""
  A[:, i] *= alpha
  return None
```

非破壊版の行の定数倍の関数は、まず配列をコピーし、コピー元の行列のサイズの単位行列を生成します。その単位行列変形し、元の行列と行列積にかけることで実現しました。また、in-place 版では、行列の特定の列にそのまま定数倍して関数を作成しました。列母定数倍も同じように作っています。

4.3 置換変換の関数について

```
  def row_swap(A: np.ndarray, i: int, j: int) -> np.ndarray:
  """行 i と行 j を置換する関数非破壊版()変更した行列を返す行列は変更されないA"""
  B = A.copy()
  #print("A\n", A)
  M = np.eye(len(A))
  M[i, i] = 0
  M[j, j] = 0
  M[i, j] = 1
  M[j, i] = 1
  """
  print("M\n", M)
```

```
    print("M A\n", M @ A)  # 行 p と q の入れ替え
    """
    B = M @ A
    return B

def row_swap_inplace(A: np.ndarray, i: int, j: int) -> None:
    """行 i と行 j を置換する関数(in-版) 行列を直接変更するplaceA 返り値を持たない"""
    A[[i, j]] = A[[j, i]]
    return None

def col_swap(A: np.ndarray, i: int, j: int) -> np.ndarray:
    """列 i と列 j を置換する関数非破壊版()変更した行列を返す行列は変更されないA"""
    B = A.copy()
    #print("A\n", A)
    M = np.eye(A.shape[1])
    M[i, i] = 0
    M[j, j] = 0
    M[i, j] = 1
    M[j, i] = 1
    """
    print("M\n", M)
    print("A M\n", A @ M)  # 列 p と q の入れ替え
    """
    B = A @ M
    return B

def col_swap_inplace(A: np.ndarray, i: int, j: int) -> None:
    """列 i と列 j を置換する関数(in-版) 行列を直接変更するplaceA 返り値を持たない"""
    A[:, [i, j]] = A[:, [j, i]]
    return None
```

非破壊版の行列の行置換の方法は、まず行列のコピーを作成してそのサイズの単位行列を作成します。その単位行列の置換する二つの行の要素を交換し、その単位行列をコピーした行列との積をとることで実現しました。また、in-place 版では、直接変更する A[[i, j]] = A[[j, i]] を用いました。同様な操作を列の置換でも行っています。

4.4 線形和の関数について

```
    def row_axpy(A: np.ndarray, i: int, j: int, alpha: float, beta: float, gamma:
        float, delta: float) ->np.ndarray:
    #行 i を alpha * 行i + beta * 行にj行, j を gamma * 行i + delta * 行にするj.
    B = A.copy()

    row_i_old = A[i, :]
    row_j_old = A[j, :]

    B[i, :] = alpha * row_i_old + beta  * row_j_old
    B[j, :] = gamma * row_i_old + delta * row_j_old

    return B

```

```
def row_axpy_inplace(A: np.ndarray, i: int, j: int, alpha: float, beta: float, ganmma: float, delta: float) -> None:
  #行 i を alpha * 行i + beta * 行にj行, j を gamma * 行i + delta * 行にするj.(in-版)place
  row_i_old = A[i, :].copy()   # 元の行 i
  row_j_old = A[j, :].copy()   # 元の行 j

  A[i, :] = alpha * row_i_old + beta  * row_j_old
  A[j, :] = gamma * row_i_old + delta * row_j_old
  return None


def col_axpy(A: np.ndarray, i: int, j: int, alpha: float, beta: float, gamma: float, delta: float) ->np.ndarray:
  #列 i を alpha * 列i + beta * 列にj列, j を gamma * 列i + delta * 行にするj.

  B = A.copy()
  col_i_old = A[:, i]
  col_j_old = A[:, j]

  B[:, i] = alpha * col_i_old + beta * col_j_old
  B[:, j] = gamma * col_i_old + delta * col_j_old

  return B

def col_axpy_inplace(A: np.ndarray, i: int, j: int, alpha: float, beta: float, ganmma: float, delta: float) -> None:
  #列 i を alpha * 列i + beta * 列にj列, j を gamma * 列i + delta * 行にするj.(in-版place)
  col_i_old = A[:, i].copy()   # 元の列 i を保存
  col_j_old = A[:, j].copy()   # 元の列 j を保存

  A[:, i] = alpha * col_i_old + beta  * col_j_old
  A[:, j] = gamma * col_i_old + delta * col_j_old
  return None
```

非破壊版の線形和は、入力行列 A を変更せず新しい行列 B を生成します。そのあと、操作対象となる行は, row_i_old = A[i, :].copy() のようにコピーを一時保存する。これにより、元の行（列）を保持したまま計算が行えます。コピーをもとに計算を行うことで実現しました。一方で in-place 版では、一時変数として行のコピーを保存し， row_i_old = A[i, :]、row_j_old = A[j, :] に保存し、これをもとに線形和を計算しました。列の置換も同じような操作で実現しています。

4.5 実行結果

それぞれランダムに行列を製作し、非破壊版→ inplace 版→非破壊版の順番で定数倍、置換、線形和を行っています。この結果を見ると in-place 版で値が直接書き換えられていることがわかります。

```
  n = 5
A = rng.random(size=(n, n))

print("A =\n" ,A)

alpha = 2
```

```
beta = 1
gamma = 2
delta = 5
i = 2
j = 3
k = 4

#行操作
print("row_scale\n",row_scale(A, i, alpha)) #行を倍ialpha
print()
print("row_scale_inplace",row_scale_inplace(A, i, alpha))
print(A)
print("row_scale\n",row_scale(A, i, alpha)) #行を倍ialpha
print()
```

実行結果 (一部/一例)

```
  A =
 [[0.10697645 0.55787988 0.61883169 0.75897576 0.6432531 ]
 [0.62692667 0.46756794 0.42274748 0.50420321 0.54114282]
 [0.51970615 0.36019001 0.51816072 0.39244666 0.45370231]
 [0.67342661 0.75719158 0.40247333 0.87590668 0.7340864 ]
 [0.89503677 0.80626965 0.78134647 0.0787148  0.84589685]]
row_scale
 [[0.10697645 0.55787988 0.61883169 0.75897576 0.6432531 ]
 [0.62692667 0.46756794 0.42274748 0.50420321 0.54114282]
 [1.03941231 0.72038003 1.03632144 0.78489333 0.90740461]
 [0.67342661 0.75719158 0.40247333 0.87590668 0.7340864 ]
 [0.89503677 0.80626965 0.78134647 0.0787148  0.84589685]]

row_scale_inplace None
[[0.10697645 0.55787988 0.61883169 0.75897576 0.6432531 ]
 [0.62692667 0.46756794 0.42274748 0.50420321 0.54114282]
 [1.03941231 0.72038003 1.03632144 0.78489333 0.90740461]
 [0.67342661 0.75719158 0.40247333 0.87590668 0.7340864 ]
 [0.89503677 0.80626965 0.78134647 0.0787148  0.84589685]]
row_scale
 [[0.10697645 0.55787988 0.61883169 0.75897576 0.6432531 ]
 [0.62692667 0.46756794 0.42274748 0.50420321 0.54114282]
 [2.07882462 1.44076005 2.07264288 1.56978665 1.81480922]
 [0.67342661 0.75719158 0.40247333 0.87590668 0.7340864 ]
 [0.89503677 0.80626965 0.78134647 0.0787148  0.84589685]]
```