Computer Homework 1

Matrix and Tensor Decompositions

Dr. Sepideh Hajipour

Sharif University of Technology

Electrical Engineering

Nikoo Moradi

400101934

Date: April 16, 2024

Contents

1 **2**

2 **4**

2.A 4

2.A.a 4

2.A.b 5

2.A.c 7

2.A.d 7

2.B 7

3 **10**

3.A 10

3.B 13

1

In this question, we compute the eigenvectors and eigenvalues of symmetric matrices using the Jacobi method. We compare the results from the custom-built Jacobi method with MATLAB's built-in `eig` function as well. We defined two functions

Function `Jacobi_eig`

- **Input:** A symmetric matrix `A`.
- **Output:** Matrices `V` (eigenvectors) and `D` (eigenvalues).
- **Key Components:**
  - **Initial Setup:** Matrices `D` and `V` are initialized. `D` is a copy of `A`, and `V` is initialized as an identity matrix.
  - **Tolerance Setting:** A tolerance `e` is set to 10^{-10} to determine the precision of the convergence.

Off-Diagonal Norm Calculation (Function `off`)

- **Description:** Computes the Frobenius norm of the off-diagonal elements of `A`, which is used to determine when the matrix `D` has become sufficiently diagonal.

Main Loop

- **Description:** The loop continues until the off-diagonal norm is less than `e`. Within the loop:
  - **Maximum Off-Diagonal Element:** It searches for the largest off-diagonal element in `D` to determine the indices `p` and `q`.
  - **Angle Calculation:** Computes the angle `theta` for the rotation matrix using the formula for the Jacobi rotation.
  - **Rotation Matrix (Givens Rotation):** Forms the rotation matrix `J` based on computed `c` (cosine) and `s` (sine) values.
  - **Matrix Updates:** Updates `D` and `V` using the rotation matrix to progressively diagonalize `D` while accumulating transformations in `V`.

Comparing results with `eig`

V_{ii} and D_{ii} are outputs of `eig`, A_i is our input matrix and V_i and D_i are otputs of our implemented function `Jacobi_eig`.

```
A1 =

    1.0000    1.4142    2.0000
    1.4142    3.0000    1.4142
    2.0000    1.4142    1.0000


V1 =

    0.5000   -0.5000   -0.7071
    0.7071    0.7071         0
    0.5000   -0.5000    0.7071


D1 =

    5.0000    0.0000    0.0000
    0.0000    1.0000   -0.0000
    0.0000   -0.0000   -1.0000


V11 =

    0.7071    0.5000    0.5000
         0   -0.7071    0.7071
   -0.7071    0.5000    0.5000


D11 =

   -1.0000         0         0
         0    1.0000         0
         0         0    5.0000
```

```
A2 =

     1     1     1
     1     1     1
     1     1     1


V2 =

    0.5774   -0.7071   -0.4082
    0.5774    0.7071   -0.4082
    0.5774         0    0.8165


D2 =

    3.0000    0.0000         0
    0.0000    0.0000    0.0000
    0.0000   -0.0000   -0.0000


V22 =

    0.4082    0.7071    0.5774
    0.4082   -0.7071    0.5774
   -0.8165         0    0.5774


D22 =

   -0.0000         0         0
         0         0         0
         0         0    3.0000
```

Figure 1: Q1 _ Examples

```
A3 =

     7     2     1
     2     5     1
     1     1     9


V3 =

    0.7085   -0.5070    0.4908
    0.3712    0.8593    0.3517
   -0.6001   -0.0670    0.7971


D3 =

    7.2008   -0.0000         0
   -0.0000    3.7421    0.0000
   -0.0000    0.0000   10.0571


V33 =

    0.5070    0.7085    0.4908
   -0.8593    0.3712    0.3517
    0.0670   -0.6001    0.7971


D33 =

    3.7421         0         0
         0    7.2008         0
         0         0   10.0571
```

Figure 2: Q1 _ Examples

2

2.A

2.A.a

making symmetric

$$C=\begin{bmatrix} w & x \\ y & z\end{bmatrix} \qquad B=\begin{bmatrix} c & s \\ -s & c\end{bmatrix} C \quad \text{is symmetric}$$

$$\Rightarrow B=\begin{bmatrix} c & s \\ -s & c\end{bmatrix}\begin{bmatrix} w & x \\ y & z\end{bmatrix}=\begin{bmatrix} \cdots & cx+sz \\ cy-sw & \cdots\end{bmatrix} \rightsquigarrow cx+sz=cy-sw$$

$$\Rightarrow c(x-y)+s(z+w)=0 \xrightarrow{t=\frac{s}{c}} (x-y)+t(z+w)=0 \Rightarrow t=\frac{y-x}{z+w}$$

$$c=\frac{1}{\sqrt{1+t^2}}, \quad s=tc$$

Setting to zero

$$A\in\mathbb{R}^{m\times n}, \quad A=\left.\begin{bmatrix} a_{11} & & 0 \\ & \ddots & \\ 0 & & a_{nn} \\ \hline a_{n+1,1} & \cdots & a_{n+1,n} \\ \vdots & & \\ a_{m1} & & a_{mn}\end{bmatrix}\right\} \rightsquigarrow$$ we want to set these elements to zero using the $A(p,p)$ and $A(q,p)$ elements

3x2 example: $\begin{bmatrix} c & 0 & s \\ 0 & 1 & 0 \\ -s & 0 & c\end{bmatrix}\begin{bmatrix} x & y \\ z & w \\ h & r\end{bmatrix}=\begin{bmatrix} x' & y' \\ z' & w' \\ h' & t'\end{bmatrix}$ we want $h'=A'(3,1)=0 \Rightarrow$

$$h'=0=-sx+ch \xrightarrow{t=\frac{s}{c}} 0=-tx+h \Rightarrow t=\frac{h}{x}=\frac{A(3,1)}{A(1,1)}$$

$$c=\frac{1}{\sqrt{1+t^2}}, \quad s=tc$$

$\Rightarrow$ To set the $A(q,p)$ to zero, we need $\begin{bmatrix} 1 & \cdots & & & 0 \\ \vdots & c & \cdots & s & \vdots \\ \vdots & -s & \cdots & c & \vdots \\ 0 & \cdots & & & 1\end{bmatrix}$

wher $t=\dfrac{A(q,p)}{A(p,p)}$ and $c=\dfrac{1}{\sqrt{1+t^2}}$, $s=tc$

2.A.b

In order to obtain a stable algorithm for computing the SVD of C we merge the Jacobi idea and the symmetrizing algorithm together. Firtst we make the nxn part of the A (first nth rows) symmetric, then we set the off diagonal elements to zero using the jacobi rotations and `symSchur2` algorithm, and in the end we also set the rest of the off diagonal elements that are not in the nxn part (rows from n+1 to m) to zero. here is the pseudo-code of this algorithm:

Algorithm 1 Symmetric Schur Decomposition

function SYMSCHUR2(A, p, q)
    if $A(p, q) \neq 0$ **then**
        $\tau \leftarrow (A(q, q) - A(p, p))/(2 \times A(p, q))$
        if $\tau \geq 0$ **then**
            $t \leftarrow 1/(\tau + \sqrt{1 + \tau^2})$
        else
            $t \leftarrow 1/(\tau - \sqrt{1 + \tau^2})$
        end if
        $c \leftarrow 1/\sqrt{1 + t^2}$
        $s \leftarrow c \times t$
    else
        $c \leftarrow 1$
        $s \leftarrow 0$
    end if
    return c, s
end function

Algorithm 2 Making Symmetric

function MAKING_SYMMETRIC(A, p, q)
    $app \leftarrow A(p, p)$
    $apq \leftarrow A(p, q)$
    $aqp \leftarrow A(q, p)$
    $aqq \leftarrow A(q, q)$
    $t \leftarrow (aqp - apq)/(app + aqq)$
    $c \leftarrow 1/\sqrt{1 + t^2}$
    $s \leftarrow t \times c$
    return c, s
end function

Algorithm 3 Setting to Zero

```
function SETTING_TO_ZERO(A, p, q)
    app ← A(p, p)
    aqp ← A(q, p)
    t3 ← aqp/app
    c3 ← 1/√(1 + t3^2)
    s3 ← t3 × c3
    return c3, s3
end function
```

Algorithm 4 Two-Sided Jacobi SVD

```
function JACOBI_SVD_2SIDED(A)
    m, n ← size(A)
    V ← eye(n, n)
    U ← eye(m, m)
    e ← 10^-20
    while off(A) > e do
        for p ← 1 to n do
            for q ← p + 1 to n do
                cos, sin ← making_symmetric(A, p, q)
                symmetric ← eye(m, m)
                symmetric([p, q], [p, q]) ← [cos, sin; −sin, cos]
                A ← symmetric × A
                c1, s1 ← symSchur2(A, p, q)
                J1 ← eye(m, m)
                J1([p, q], [p, q]) ← [c1, s1; −s1, c1]
                J2 ← eye(n, n)
                J2([p, q], [p, q]) ← [c1, s1; −s1, c1]
                A ← J1′ × A × J2
                U ← U × symmetric′ × J1
                V ← V × J2
            end for
            for q ← n + 1 to m do
                c3, s3 ← setting_to_zero(A, p, q)
                J3 ← eye(m, m)
                J3([p, q], [p, q]) ← [c3, −s3; s3, c3]
                A ← J3′ × A
                U ← U × J3
            end for
        end for
    end while
    S ← A
    return U, S, V
end function
```

2.A.c

$$\mathrm{off}(A)^2 = \|A\|_F^2 - \sum_{i=1}^{n} a_{ii}^2$$

تبدیل متعامد : $\|B\|_F^2 = \|A\|_F^2$ گام ② الگوریتم $\xrightarrow{\text{تبدیل متعامد}}$ $a_{pp}^2 + a_{pq}^2 + a_{qq}^2 + a_{qp}^2 = b_{pp}^2 + b_{qq}^2$

$$\mathrm{off}(B) = \|B\|_F^2 - \sum_{i=1}^{n} b_{ii}^2 = \|B\|_F^2 - \sum_{\substack{i=1 \\ i \neq p \\ i \neq q}}^{n} b_{ii}^2 - b_{pp}^2 - b_{qq}^2 = \|A\|_F^2 - \sum_{\substack{i=1 \\ i \neq p \\ i \neq q}}^{n} a_{ii}^2 - a_{pp}^2 - a_{qq}^2 - a_{pq}^2 - a_{qp}^2$$

$$= \mathrm{off}^2(A) - a_{pq}^2 - a_{qp}^2 \implies \mathrm{off}(B) \leq \mathrm{off}(A)$$

2.A.d

Everything about the algorithm was explained in other sections. Also details of the MATLAB code of two_sided jacobi will be explained in the next section.

2.B

In this question we compute the singular value decomposition (SVD) of real rectangular matrices using a two-sided Jacobi method. We compares the results from the custom Jacobi SVD implementation with MATLAB's built-in `svd` function across several test matrices.

Function Definitions

Function `symSchur2`

- **Purpose:** Calculates cosines and sines for the Jacobi rotation to annihilate off-diagonal elements, making the matrix closer to diagonal form.
- **Input:** Matrix `A`, indices `p` and `q`.
- **Output:** Cosine (`c`) and sine (`s`) values for the transformation.

Function `making_symmetric`

- **Purpose:** Adjusts the matrix to be more symmetric around indices `p` and `q`, aiding its diagonalization.
- **Input:** Matrix `A`, indices `p` and `q`.
- **Output:** Cosine (`c`) and sine (`s`) values for symmetric adjustments.

Function `Jacobi_svd_2sided`

- **Input:** A rectangular matrix `A`.
- **Output:** Left singular vectors (`U`), singular values (`S`), and right singular vectors (`V`).
- **Key Components:**
  - **Initialization:** Matrices `V` and `U` are initialized as identity matrices. The matrix `S` initially mirrors `A`.
  - **Loop Condition:** Continues until the off-diagonal norm of `A` is less than a set tolerance, applying rotations to diagonalize `A`.
  - Within the loop, applies rotations to the first nth rows of A to make it symmetric and then make it diagonal using two nested for loops that iterate over the columns and rows of A, and then set the rows from $n + 1$ till m to zero.
  - Updates U and V with rotations that aim to zero out off-diagonal elements of A, progressively converging towards a diagonal matrix S.

Function `off`

- **Purpose:** Computes the Frobenius norm of the off-diagonal elements of `A` to determine when the matrix has been sufficiently diagonalized.
- **Output:** The norm (`Off`), indicating the magnitude of off-diagonal elements.

Function `setting_to_zero`

- Specifically adjusts A to drive a single off-diagonal element to zero, used in the second loop where the column index exceeds the row index.

Comparing results with `svd`

U_{ii} , S_{ii} and V_{ii} are outputs of `svd`, A_i is our input matrix and U_i , S_i and V_i are otputs of our implemented function `Jacobi_svd_2sided`.

```
A1 =

     1     2     3
     1     5     3
     1     0     0
     0     2     2

U1 =

    0.0683    0.4814    0.8042   -0.3417
   -0.2604    0.7923   -0.5247   -0.1709
   -0.8138    0.0237    0.2728    0.5126
    0.5150    0.3742    0.0590    0.7689

S1 =

   -1.1118    0.0000   -0.0000
    0.0000    7.3986    0.0000
   -0.0000   -0.0000    1.4230
    0.0000   -0.0000         0

V1 =

    0.9047    0.1754    0.3882
    0.1218    0.7667   -0.6303
   -0.4081    0.6176    0.6723
```

```
U11 =

   -0.4814    0.8042   -0.0683   -0.3417
   -0.7923   -0.5247    0.2604   -0.1709
   -0.0237    0.2728    0.8138    0.5126
   -0.3742    0.0590   -0.5150    0.7689

S11 =

    7.3986         0         0
         0    1.4230         0
         0         0    1.1118
         0         0         0

V11 =

   -0.1754    0.3882    0.9047
   -0.7667   -0.6303    0.1218
   -0.6176    0.6723   -0.4081
```

Figure 3: Q2 _ Ex 1

```
A2 =

    1.0000         0    3.0000
    1.0000    5.0000    3.0000
    1.0000         0    7.0000
    1.0000    2.2361    2.0000

U2 =

    0.7095   -0.2334    0.3228   -0.5813
   -0.2596    0.7432    0.5248   -0.3238
   -0.3643   -0.5587    0.7227    0.1810
    0.5445    0.2847    0.3132    0.7241

S2 =

    0.6449   -0.0000    0.0000
    0.0000    4.7206   -0.0000
    0.0000   -0.0000    9.0719
    0.0000    0.0000   -0.0000

V2 =

    0.9769    0.0499    0.2076
   -0.1250    0.9220    0.3664
   -0.1731   -0.3839    0.9070
```

```
U22 =

   -0.3228    0.2334   -0.7095   -0.5813
   -0.5248   -0.7432    0.2596   -0.3238
   -0.7227    0.5587    0.3643    0.1810
   -0.3132   -0.2847   -0.5445    0.7241

S22 =

    9.0719         0         0
         0    4.7206         0
         0         0    0.6449
         0         0         0

V22 =

   -0.2076   -0.0499   -0.9769
   -0.3664   -0.9220    0.1250
   -0.9070    0.3839    0.1731
```

Figure 4: Q2 _ Ex 2

```
A3 =

        0    9.0000    2.0000
   7.0000    1.0000    4.0000
   5.0000    8.0000    2.6458
   3.0000    2.4495    1.0000


U3 =

   0.6108    0.5563   -0.5107    0.2377
  -0.7765    0.3886   -0.4957    0.0208
   0.0041    0.6853    0.5090   -0.5209
  -0.1548    0.2643    0.4842    0.8196


S3 =

  -7.6037   -0.0000   -0.0000
  -0.0000   14.2654    0.0000
   0.0000    0.0000   -1.2971
   0.0000    0.0000         0


V3 =

   0.7732    0.4865   -0.4068
  -0.5753    0.8079   -0.1274
   0.2667    0.3326    0.9046
```

```
U33 =

  -0.5563   -0.6108    0.5107    0.2377
  -0.3886    0.7765    0.4957    0.0208
  -0.6853   -0.0041   -0.5090   -0.5209
  -0.2643    0.1548   -0.4842    0.8196


S33 =

  14.2654         0         0
        0    7.6037         0
        0         0    1.2971
        0         0         0


V33 =

  -0.4865    0.7732   -0.4068
  -0.8079   -0.5753   -0.1274
  -0.3326    0.2667    0.9046
```

Figure 5: Q2 _ Ex 3

3

3.A

making - orthogonal

$\underline{x}, \underline{y} \in \mathbb{R}^m \qquad Q = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \qquad [x|y]Q \rightarrow$ is orthogonal

$$\begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_m & y_m \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} w_1 & z_1 \\ \vdots & \vdots \\ w_m & z_m \end{bmatrix} \rightarrow \text{orthogonal} \Rightarrow \underline{w}^T \underline{z} = 0$$

$$\Rightarrow w_i = cx_i - sy_i \quad , z_i = sx_i + cy_i \Rightarrow \underline{w}^T \underline{z} = \sum_{i=1}^{m} w_i z_i = 0$$

$$\Rightarrow 0 = x^T y (c^2 - s^2) + sc(x^2 - y^2) \longrightarrow \tau = \frac{y^2 - x^2}{2x^T y} \; , t = \frac{s}{c}$$

$$\Rightarrow t^2 + 2\tau t - 1 = 0 \longrightarrow t = \frac{1}{\tau + \sqrt{1+\tau^2}} \quad \tau \geq 0$$

$$\longrightarrow t = \frac{1}{\tau - \sqrt{1+\tau^2}} \quad \tau < 0$$

$$c = \frac{1}{\sqrt{1+t^2}} \; , \; s = tc$$

Here is the psuedocode of our algorithm for one_sided jacobi :

Pseudocode

Function for Making Orthogonal Transformations

Algorithm 5 Making Orthogonal

```
function MAKING_ORTHOGONAL(x, y)
    a ← norm(x)^2
    b ← norm(y)^2
    c ← dot(x, y)
    if c ≠ 0 then
        τ ← (b − a)/(2 ∗ c)
        if τ ≥ 0 then
            t ← 1/(τ + √(1 + τ^2))
        else
            t ← 1/(τ − √(1 + τ^2))
        end if
        c ← 1/√(1 + t^2)
        s ← c ∗ t
    else
        c ← 1
        s ← 0
    end if
    return c, s
end function
```

Algorithm 6 Jacobi SVD One-Sided

```
function JACOBI_SVD_1SIDED(A)
    m, n ← size(A)
    V ← identity matrix of size n × n
    S ← zero matrix of size m × n
    U ← zero matrix of size m × m
    e ← 10^−30
    not_orthogonal ← true
    err ← 0
    while not_orthogonal do
        for p = 1 to n − 1 do
            for q = p + 1 to n do
                x ← A[:, p]
                y ← A[:, q]
                cos, sin ← MAKING_ORTHOGONAL(x, y)
                J ← identity matrix of size n × n
                J[[p, q], [p, q]] ← [[cos, −sin], [sin, cos]]
                A ← A ∗ J
                V ← J′ ∗ V
            end for
        end for
        err ← 0
        for p = 1 to n do
            for q = p + 1 to n do
                err ← err + dot(A[:, p], A[:, q])
            end for
        end for
        if abs(err) < e then
            not_orthogonal ← false
        else
            not_orthogonal ← true
        end if
    end while
    for j = 1 to min(m, n) do
        S[j, j] ← norm(A[:, j])
        U[:, j] ← A[:, j]/norm(A[:, j])
    end for
    return U, S, V
end function
```

3.B

In this question we compute the Singular Value Decomposition (SVD) of real rectangular matrices using a one-sided Jacobi algorithm. Also we compare the results from this custom implementation with MATLAB's built-in `svd` function.

Test Matrices Initialization

- **Lines:** Initialization of matrices `A1`, `A2`, and `A3`.
- **Description:** The script initializes three rectangular matrices, which are then used to test the custom `Jacobi_svd_1sided` function.

Function Invocation and Comparison

- **Lines:** For each matrix (e.g., `[U1, S1, V1] = Jacobi_svd_1sided(A1)`).
- **Description:** Computes the singular value decomposition using both the custom `Jacobi_svd_1sided` function and MATLAB's built-in `svd` function. The decompositions provide the left singular vectors (`U`), the singular values (`S`), and the right singular vectors (`V`). The results are then stored for comparison.

Function Definitions

Function `making_orthogonal`

- **Purpose:** Calculates rotation angles (cosine and sine) needed to make two columns of a matrix orthogonal to each other.
- **Input:** Vectors `x` and `y` which are columns from the matrix `A`.
- **Output:** Cosine (`c`) and sine (`s`) values used to form a rotation matrix.
- **Mechanism:** Computes a parameter `tau` based on norms and dot products, which is used to compute the rotation angle `t`. The cosine and sine for the rotation matrix are derived from `t`.

Function `Jacobi_svd_1sided`

- **Input:** A real rectangular matrix `A`.
- **Output:** Matrices `U` (left singular vectors), `S` (diagonal matrix of singular values), and `V` (right singular vectors).
- **Key Components:**
  - **Initialization:** Matrices `V` and `U` are initialized as identity matrices, and `S` as a zero matrix. A very small error tolerance `e` is set.
  - **Orthogonalization Loop:** Iterates over all pairs of columns in `A`. For each pair, calculates the necessary rotation (using `making_orthogonal`) to make these columns orthogonal and applies this rotation to `A` and updates `V`.

– **Convergence Check:** A loop calculates the sum of dot products between all pairs of columns to check if they are orthogonal. If the sum is below the tolerance `e`, the loop ends.

– **Normalization:** After achieving orthogonality, each column of `A` is normalized to form the matrix `S`, and the normalized vectors form the columns of `U`.

Comparing results with `svd`

U_{ii} , S_{ii} and V_{ii} are outputs of `svd`, A_i is our input matrix and U_i , S_i and V_i are otputs of our implemented function `Jacobi_svd_1sided`.

```
A1 =

     1     2
     1     5
     1     0
     0     2


U1 =

    0.4515    0.3690         0         0
   -0.0887    0.8673         0         0
    0.8116    0.0368         0         0
   -0.3601    0.3322         0         0


S1 =

    1.2029         0
         0    5.8782
         0         0
         0         0


V1 =

    0.9763   -0.2166
    0.2166    0.9763
```

```
U11 =

   -0.3690    0.4515   -0.7894    0.1918
   -0.8673   -0.0887    0.2527   -0.4197
   -0.0368    0.8116    0.5367    0.2279
   -0.3322   -0.3601    0.1577    0.8574


S11 =

    5.8782         0
         0    1.2029
         0         0
         0         0


V11 =

   -0.2166    0.9763
   -0.9763   -0.2166
```

Figure 6: Q3 _ Ex 1

```
A2 =

     1     0
     0     1
     1     1


U2 =

    0.7071    0.4082         0
   -0.7071    0.4082         0
         0    0.8165         0


S2 =

    1.0000         0
         0    1.7321
         0         0


V2 =

    0.7071   -0.7071
    0.7071    0.7071
```

```
U22 =

   -0.4082    0.7071   -0.5774
   -0.4082   -0.7071   -0.5774
   -0.8165   -0.0000    0.5774


S22 =

    1.7321         0
         0    1.0000
         0         0


V22 =

   -0.7071    0.7071
   -0.7071   -0.7071
```

Figure 7: Q3 _ Ex 2

```
A3 =

    1.0000    8.0000
    2.0000    1.0000
    5.0000    1.4142
    2.0000    6.0000


U3 =

   -0.3732    0.7372         0         0
    0.3176    0.1525         0         0
    0.8709    0.2855         0         0
   -0.0371    0.5931         0         0


S3 =

    4.8358         0
         0   10.6590
         0         0
         0         0


V3 =

    0.9393   -0.3430
    0.3430    0.9393
```

```
U33 =

    0.7372   -0.3732   -0.0574   -0.5604
    0.1525    0.3176   -0.9321    0.0846
    0.2855    0.8709    0.3219   -0.2374
    0.5931   -0.0371    0.1560    0.7890


S33 =

   10.6590         0
         0    4.8358
         0         0
         0         0


V33 =

    0.3430    0.9393
    0.9393   -0.3430
```

Figure 8: Q3 _ Ex 3