Name: Kunal Ghosh

Course: M.Tech (Aerospace Engineering)

Subject: AE 291 (Matrix Computations)

SAP No.: 6000007645

S.R. No.: 05-01-00-10-42-22-1-21061

Importing the necessary libraries

[1]:
```
import numpy as np
```

[2]:
```
import pandas as pd
```

[3]:
```
import sys
```

[4]:
```
import warnings
```

[5]:
```
warnings.filterwarnings(action='ignore', category=UserWarning)
```

Problem (1):

Write a function that implements Cholesky decomposition to get the Cholesky factor R for a (symmetric) positive definite matrix A. The function should read any matrix, and it should output an error message when the matrix is not positive definite.

Answer (1):

Formula to calculate the Cholesky factor, R

Let,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Also,

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} & \cdots & r_{1n} \\ 0 & r_{22} & r_{23} & \cdots & r_{2n} \\ 0 & 0 & r_{33} & \cdots & r_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & r_{nn} \end{bmatrix}$$

So,

$$A = R^T R$$

So,

$$r_{11} = \sqrt{a_{11}}$$

Also,

$$r_{1j} = \frac{a_{1j}}{r_{11}}$$

where, $j \neq 1$ and $j = \{2, 3, ..., n\}$

So, if i > 1 and $i = \{2, 3, ..., n\}$

$$r_{ii} = \sqrt{a_{ii} - \left(\sum_{k=1}^{i-1} r_{ki}^2\right)}$$

So, if i > 1 and $i = \{2, 3, ..., n\}$ and also, $j = i + 1, i + 2, ..., n$

$$r_{ij} = \frac{a_{ij} - \left(\sum_{k=1}^{i-1} r_{ki} r_{kj}\right)}{r_{ii}}$$

Pseudocode to calculate the Cholesky factor, R

for i = 1,2,. . . ,n

  for k = 1,2,. . . ,i-1 (NOT executed when i = 1)

    $a_{ii} \leftarrow a_{ii} - a_{ki}^2$

  $a_{ii} \leq 0$,set error flag, exit (A is NOT positive definite)

  $a_{ii} \leftarrow \sqrt{a_{ii}}$ (This is r_{ii})

  for j = i+1,i+2,. . . ,n (NOT executed when i = n)

    for k = 1,2,. . . ,i-1 (NOT executed when i = 1)

      $a_{ij} \leftarrow a_{ij} - a_{ki} a_{kj}$

  $a_{ij} = \frac{a_{ij}}{a_{ii}}$ (This is r_{ij})

Cholesky(A) function would take symmetric positive definite matix (A) as input and return Cholesky factor (R) as output

[6]:
```
def Cholesky(A):

    # NOTE: Matrix A will be modified after the execution of this function

    # Error flag if the input matrix is NOT a square matrix
    # Checking if the number of rows and columns of the matrix A are equal or NOT

    if A.shape[0] != A.shape[1]:
        print(f"Cholesky(A): A matrix is NOT a square matrix")
        sys.exit()
    else:
        # Dimension of the matrix is stored in the variable n
        n = A.shape[0]

    # Calculating the Cholesky factor (R)
    for i in range(n):
        if i > 0:
            for k in range(i):
                A[i][i] = A[i][i] - (A[k][i]**2)

        # Error flag if the input matrix is NOT a positive definite matrix

        if A[i][i] <= 0:
            print(f"Cholesky(A): A matrix is NOT a positive definite matrix")
            sys.exit()
        else:
            A[i][i] = A[i][i]**0.5

        for j in range(i+1,n):
            for k in range(i):
                A[i][j] = A[i][j] - (A[k][i]*A[k][j])
            A[i][j] = A[i][j]/A[i][i]

    # Declaring a new matrix to store the Cholesky factor(R)
    R = np.zeros(A.shape)

    # Storing the Cholesky factor to R
    for i in range(n):
        for j in range(i,n):
            R[i,j] = A[i,j]

    # Returning the Cholesky factor R
    return R
```

Problem (2):

Write functions that implement forward and backward substitutions for linear systems whose coefficient matrices are lower and upper triangular, respectively.

Answer (2):

Let L be a lower triangular matrix.

$$L = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix}$$

And b be a vector.

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$$

So, Ly = b

$$\begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$$

Formula for forward substitution

$$y_1 = \frac{b_1}{l_{11}}$$

For i > 1,

$$y_i = \frac{b_i - \left(\sum_{j=1}^{i-1} l_{ij} y_j\right)}{l_{ii}}$$

Pseudocode for forward substitution

for i = 1,2,. . . ,n

    for j = 1,2,. . . ,i-1 (NOT executed when i = 1)

        $b_i \leftarrow b_i - (l_{ij} b_j)$

    if $l_{ii} = 0$, set error flag exit

    $b_i \leftarrow \frac{b_i}{l_{ii}}$

Forward_Lower_Triangular(L,b) would take a lower triangular matrix (L) and the corresponding RHS (b) of the system of equations as inputs. This function would solve them using forward substitution and return the solution. The solution will be returned in b. So, the vector b will be modified after the execution of this function.

[7]:
```python
def Forward_Lower_Triangular(L,b):

    # NOTE: Vector b will be modified after the execution of this function

    # Error flag if the input matrix is NOT a square matrix
    # Checking if the number of rows and columns of the matrix L are equal or NOT

    if L.shape[0] != L.shape[1]:
        print(f"Forward_Lower_Triangular(L,b): L matrix is NOT a square matrix")
        sys.exit()
    else:
        # Dimension of the matrix is stored in the variable n
        n = A.shape[0]

    # Calculatig the solution
    for i in range(n):
        if i > 0:
            for j in range(i):
                b[i] = b[i] - L[i][j]*b[j]

        # Error flag if the diagonal element of the input matrix, L is 0
        if L[i][i] == 0:
            print(f"Forward_Lower_Triangular(L,b): Diagonal element zero")
            sys.exit()
        else:
            b[i] = b[i]/L[i][i]

    # Returning the solution
    return b
```

Let U be an upper triangular matrix.

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

And b be a vector.

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$$

So, Uz = b

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$$

Formula for backward substitution

$$z_n = \frac{b_n}{u_{nn}}$$

For i $<$ n,

$$z_i = \frac{b_i - \left(\sum_{j=i+1}^{n} u_{ij} z_j\right)}{u_{ii}}$$

Pseudocode for backward substitution

for i = n,n-1,. . . ,1

    for j = i+1,i+2,. . . ,n (NOT executed when i = n)

        $b_i \leftarrow b_i - (u_{ij} b_j)$

    if $u_{ii} = 0$, set error flag exit

    $b_i \leftarrow \frac{b_i}{u_{ii}}$

Backward_Upper_Triangular(U,b) would take a upper triangular matrix (U) and the corresponding RHS (b) of the system of equations as inputs. This function would solve them using backward substitution and return the solution. The solution will be returned in b. So, the vector b will be modified after the execution of this function.

[8]:
```
def Backward_Upper_Triangular(U,b):

    # NOTE: Vector b will be modified after the execution of this function

    # Error flag if the input matrix is NOT a square matrix
    # Checking if the number of rows and columns of the matrix U are equal or NOT

    if U.shape[0] != U.shape[1]:
        print(f"Backward_Upper_Triangular(U,b): U matrix is NOT a square matrix")
        sys.exit()
    else:
        # Dimension of the matrix is stored in the variable n
        n = A.shape[0]

    # Calculatig the solution
    for i in range(n-1,-1,-1):
        if i < (n-1):
            for j in range(i+1,n):
                b[i] = b[i] - U[i][j]*b[j]

        # Error flag if the diagonal element of the input matrix, U is 0
        if U[i][i] == 0:
            print(f"Backward_Upper_Triangular(U,b): Diagonal element zero")
            sys.exit()
        else:
            b[i] = b[i]/U[i][i]

    # Returning the solution
    return b
```

Problem (3):

Write a function that solves the linear system Ax=b using the above functions, taking a matrix A (positive definite) and a right hand side vector b.

Answer (3):

Matrix_Transpose(A) would take a matrix A as input and return its transpose as the output

(NOTE: This function will be used to generate positive definite matrix from any random matrix)

[9]:
```
def Matrix_Transpose(A):

    # Declaring a new matrix (A_T) to store the transpose of A matrix
    A_T = np.zeros((A.shape[1],A.shape[0]))

    # Calculating the transpose of matrix A
    for i in range(A.shape[0]):
        for j in range(A.shape[1]):
            A_T[i][j] = A[j][i]

    # Returning the transpose of matrix A
    return A_T
```

As, A is a positive definite matrix, so,

$$A = R^T R$$

where, R = Cholesky Factor of A

Also, R is a upper triangular matrix.

So, R^T is lower triangular matrix. Let,

$$L = R^T$$

So,

$$A = LR$$

As, we have

$$Ax = b$$

So,

$$LRx = b$$

Let,

$$y = Rx$$

So,

$$Ly = b$$

We can solve this for y using forward substitution, as L is a lower triangular matrix.

Once, we get the y, we don't need b after that. So, b will be modified.

So,

$$Rx = y$$

We can solve this for x using backward substitution, as R is an upper triangular matrix.

So, we can obtain x for the linear system of equation $Ax = b$, provided A is a positive definte matrix.

linear_solver(A,b) would solve a system of equation Ax = b. Here, matrix A is a positive definte matrix. Also, the vector b will be modified after the execution of this function

[10]:
```
def linear_solver(A,b):
    # NOTE: Vector b will be modified after the execution of this function

    # Checking the compatibility of the dimensions of A and b
    if A.shape[0] != b.shape[0]:
        print(f"linear_solver(A,b): Dimensions of A and b are NOT compatible.")
        sys.exit()

    # Calculating the Cholesky factor of A
    R = Cholesky(A)

    # Calculating the transpose of the Cholesky factor of R
    L = Matrix_Transpose(R)

    # Solving the linear system of equation using forward substitution with␣
↪lower triangular coefficient matrix, L
    b = Forward_Lower_Triangular(L,b)

    # Solving the linear system of equation using backward substitution with␣
↪upper triangular coefficient matrix, U
    b = Backward_Upper_Triangular(R,b)

    # Returning the solution
    return b
```

NOTE: User may choose to directly modify "A.csv" and "b.csv" to input the coefficient matrix A and RHS b, rather than generating random A and b. Then the user should skip execution until * snippet below.

NOTE: Do NOT execute the snippets immediately below † if "A.csv" and "b.csv" is given by the user.

†Generating a random matrix, M using numpy

†Dimension of matrix M can be modified by changing M_n

$$M \in \mathbb{R}^{M_n \times M_n}$$

[11]:
```
M_n = 6
```

[12]:
```
M = np.random.rand(M_n,M_n)
```

†As M matrix must be non-singular

[13]:
```
if np.linalg.det(M) == 0:
    print("Please re-generate the M matrix. A singular matrix, M was genenrated")
    sys.exit()
```

†Displaying the matrix M

[14]:
```
M
```

[14]:
```
array([[0.24464033, 0.32890386, 0.67296616, 0.11117826, 0.01400661,
        0.4332117 ],
       [0.35383414, 0.65582155, 0.10782543, 0.9968977 , 0.66727074,
        0.34887289],
       [0.21904896, 0.82670995, 0.66667357, 0.72782795, 0.69605662,
        0.16114649],
       [0.34257744, 0.13813318, 0.47888485, 0.76742922, 0.39759538,
        0.22864822],
       [0.37246241, 0.09495505, 0.65927381, 0.8381055 , 0.01560623,
        0.83400147],
       [0.71334666, 0.55518648, 0.93109618, 0.27340465, 0.23606823,
        0.18951789]])
```

†Generating a positive definite matix, A

$$A \in \mathbb{R}^{M_n \times M_n}$$

$$A = M^T M$$

[15]:
```
A = (M.T)@M
```

†Storing the number of rows of matrix A

[16]:
```
n = A.shape[0]
```

†Generating a random RHS (b) for the system of equation Ax = b

$$b \in \mathbb{R}^{M_n}$$

[17]:
```
b = np.random.rand(n)
```

†Converting A to a pandas dataframe

[18]:
```
A_file = pd.DataFrame(A)
```

†Converting b to a pandas dataframe

[19]:
```
b_file = pd.DataFrame(b)
```

†Writing A to a csv file

[20]:
```
A_file.to_csv("A.csv",index=None,header=None)
```

†Writing b to a csv file

[21]:
```
b_file.to_csv("b.csv",index=None,header=None)
```

Reading from the input files*

Reading input file "A.csv" for the coefficient matrix A

(This can be modified by the user)

[22]:
```
A = pd.read_csv("A.csv",header=None)
```

Converting the A matrix from pandas dataframe to numpy array

[23]:
```
A = A.to_numpy(dtype = np.float64)
```

Reading input file "b.csv" for the RHS b

(This can be modified by the user)

[24]:
```
b = pd.read_csv("b.csv",header=None)
```

Converting the b vector from pandas dataframe to numpy array

[25]:
```
b = b.to_numpy(dtype = np.float64)
```

Calculating the solution of Ax = b, using inbuilt numpy functions

[26]:
```
np.linalg.inv(A)@b
```

[26]:
```
array([[  3.1645799 ],
       [ 10.68135715],
       [ -2.81420873],
       [ 11.71101009],
       [-21.99811914],
       [-10.87303593]])
```

Solving the linear system of equation Ax = b, using linear_solver(A,b)

[27]:
```
x = linear_solver(A,b)
```

Converting the solution, x to a pandas dataframe

[28]:
```
x_file = pd.DataFrame(x)
```

Storing the solution to a csv file, "x.csv"

[29]:
```
x_file.to_csv("x.csv",index=None,header=None)
```

Displaying the solution, using linear_solver(A,b)

[30]:
```
x
```

[30]:
```
array([[  3.1645799 ],
       [ 10.68135715],
       [ -2.81420873],
       [ 11.71101009],
       [-21.99811914],
       [-10.87303593]])
```

Example:

If A is NOT positive definite:

Reading input file "A_NOT_Positive_Definite.csv" for the coefficient matrix A

(This can be modified by the user)

[31]:
```
A = pd.read_csv("A_NOT_Positive_Definite.csv",header=None)
```

Converting the A matrix from pandas dataframe to numpy array

[32]:
```
A = A.to_numpy(dtype = np.float64)
```

Reading input file "b_NOT_Positive_Definite.csv" for the RHS b

(This can be modified by the user)

[33]:
```
b = pd.read_csv("b_NOT_Positive_Definite.csv",header=None)
```

Converting the b vector from pandas dataframe to numpy array

[34]:
```
b = b.to_numpy(dtype = np.float64)
```

Solving the linear system of equation Ax = b, using linear_solver(A,b)

[35]:
```
x = linear_solver(A,b)
```

```
Cholesky(A): A matrix is NOT a positive definite matrix
```

```
An exception has occurred, use %tb to see the full traceback.

SystemExit
```

Example:

If Ax = b is having non-unique solution:

Reading input file "A_NON_UNIQUE_SOLUTION.csv" for the coefficient matrix A

(This can be modified by the user)

[36]:
```
A = pd.read_csv("A_NON_UNIQUE_SOLUTION.csv",header=None)
```

Converting the A matrix from pandas dataframe to numpy array

[37]:
```
A = A.to_numpy(dtype = np.float64)
```

Reading input file "b_NON_UNIQUE_SOLUTION.csv" for the RHS b

(This can be modified by the user)

[38]:
```
b = pd.read_csv("b_NON_UNIQUE_SOLUTION.csv",header=None)
```

Converting the b vector from pandas dataframe to numpy array

[39]:
```
b = b.to_numpy(dtype = np.float64)
```

Solving the linear system of equation Ax = b, using linear_solver(A,b)

[40]:
```
x = linear_solver(A,b)
```

Converting the solution, x to a pandas dataframe

[41]:
```
x_file = pd.DataFrame(x)
```

Storing the solution to a csv file, "x_NON_UNIQUE_SOLUTION.csv"

[42]:
```
x_file.to_csv("x_NON_UNIQUE_SOLUTION.csv",index=None,header=None)
```

Displaying the solution, using linear_solver(A,b)

[43]:
```
x
```

[43]:
```
array([[1.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.]])
```

Example:

If A is NOT a matrix square

Reading input file "A_NON_SQUARE.csv" for the coefficient matrix A

(This can be modified by the user)

[44]:
```
A = pd.read_csv("A_NON_SQUARE.csv",header=None)
```

Converting the A matrix from pandas dataframe to numpy array

[45]:
```
A = A.to_numpy(dtype = np.float64)
```

Reading input file "b_NON_SQUARE.csv" for the RHS b

(This can be modified by the user)

[46]:
```
b = pd.read_csv("b_NON_SQUARE.csv",header=None)
```

Converting the b vector from pandas dataframe to numpy array

[47]:
```
b = b.to_numpy(dtype = np.float64)
```

Solving the linear system of equation Ax = b, using linear_solver(A,b)

[48]:
```
x = linear_solver(A,b)
```

```
Cholesky(A): A matrix is NOT a square matrix
```

```
An exception has occurred, use %tb to see the full traceback.

SystemExit
```

Example:

If A and b are of incompatible dimensions

Reading input file "A_INCOMPATIBLE.csv" for the coefficient matrix A

(This can be modified by the user)

[49]:
```
A = pd.read_csv("A_INCOMPATIBLE.csv",header=None)
```

Converting the A matrix from pandas dataframe to numpy array

[50]:
```
A = A.to_numpy(dtype = np.float64)
```

Reading input file "b_INCOMPATIBLE.csv" for the RHS b

(This can be modified by the user)

[51]:
```
b = pd.read_csv("b_INCOMPATIBLE.csv",header=None)
```

Converting the b vector from pandas dataframe to numpy array

[52]:
```
b = b.to_numpy(dtype = np.float64)
```

Solving the linear system of equation Ax = b, using linear_solver(A,b)

[53]:
```
x = linear_solver(A,b)
```

```
linear_solver(A,b): Dimensions of A and b are NOT compatible.
```

```
An exception has occurred, use %tb to see the full traceback.

SystemExit
```
