Assignment 1

Machine Learning Course
Linear Algebra

Alliance Group

Nima Goodarzi, Amin Haratian, Mobina Sedaghat

18-June-2021

Exercise 1

Ax = b, unique solution $\Longrightarrow$ A is non-singular

To solve this problem, we use proof by contradiction, so we assume that A is singular hence, A is non-invertible (the columns are dependent) . Then, there would be a non-zero vector in its null space, call this vector z

$Az = 0 \to A(x - y) = 0$

because A is not invertible $\to x \neq y \diamond$

Also $Ax - Ay = 0 \to Ax = Ay \diamond\diamond$

According to the hypothesis of the problem $Ax = b$, so by looking at $\diamond\diamond$ therefore $Ay = b$. This is implying that $x = y$ which is in contradiction with $\diamond$. accordingly, the proposition has been proven.

Now, we need to look at the problem the other way around:

A is non-singular $\Longrightarrow Ax = b$, unique solution

$$Ax = b$$

$$A^{-1}(Ax) = A^{-1}b$$

$$(A^{-1}A)x = A^{-1}b$$

$$Ix = A^{-1}b$$

$$x = A^{-1}b$$

Exercise 2

We know that $E = e(I_m)$. So we just need to find the value of e_1 as if $E_1E = e_1(e(I))$ which means $e_1(e(A)) = A$.

First scenario: we assume e is a elementary row operation which multiples the r row of A by the non-zero scalar c. Now we define e_1 as a type I elementary row operation which multiples the r row of A by the non-zero scalar $\frac{1}{c}$. So:

$$a)r \neq i : e_1(e(A))_{ij} = e_1(A_{ij}) = A_{ij}$$
$$b)r = i : e_1(e(A))_{rj} = \frac{1}{c}e(A)_{rj} = \frac{1}{c}.c(A_{rj}) = A_{rj}$$
$$\rightarrow e_1(e(A)) = A$$

Second scenario: we assume that e is a elementary row operation which replaces the r^{th} row of A with $r^{\text{th}} + c.s^{\text{th}}$. Now we let e_1 be a type II elementary row operation which replaces the r^{th} row of A with $r^{\text{th}} + (-c).s^{\text{th}}$. for each i , $1 \leq i \leq m$, and each j, $1 \leq j \leq n$:

$$a)r \neq i : e_1(e(A))_{ij} = e_1(A_{ij}) = A_{ij}$$
$$b)r = i : e_1(e(A))_{ry} = e_1(e(A))_{ry} = e(A)_{ry} + (-c)e(A_{sj}) = A_{rj} + cA_(sj) - cA_{sj} = A_{ij}$$
$$\rightarrow e_1(e(A)) = A$$

Third scenario: we assume e is a elementary row operation which replaces the r^{th} row of A with the s^{th} row. Now we define that e_1 is a Type III row operation which replaces the s^{th} row with the r^{th} row. So:

$$a)i \neq r, s : e_1(e(A_{ij})), e_1(A_{ij}) = A_{ij}$$
$$b)i = r : e_1(e(A_{ij})) = e_1(A_{sj}) = A_{rj}$$
$$c)i = s : e_1(e(A_{sj})) = e_1(A_{rj}) = A_{sj}$$
$$\rightarrow e_1(e(A)) = A$$

Exercise 3

Part 1

$$\begin{bmatrix} a_1 & c_1 & 0 \\ e_2 & a_2 & c_2 \\ 0 & e_3 & a_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} a_1v_1 + c_1v_2 \\ e_2v_1 + a_2v_2 + c_2v_3 \\ e_3v_2 + a_3v_3 \end{bmatrix}$$

By looking at the mentioned equation, we can deduct the following algorithm:

$$v_n = e_n v_{n-1} + a_n v_n + c_n v_{n+1}$$

```
import numpy as np
# import time as time


#Since the array's range in the algorithms differs from the ones in the code, employed equations are different as well.



def TridMult(a, c, e, v):
    answer=np.zeros(len(a))
    c=np.append(c,0)
    v=np.append(v,0)
    e=np.append(e,0)
    e=np.insert(e,0,0,0)
#     tic=time.time()
    for i in range(len(a)):
        answer[i]=(a[i]*v[i]+c[i]*v[i+1]+e[i]*v[i-1])
#     toc=time.time()
    print('answer= ', answer)
#    print('time= ', toc-tic)




while True:
    a=np.array([int(j) for j in input("Enter your a array, for instance: 1,2,3,4,...: \n").split(',')])
    e=np.array([int(j) for j in input("Enter your e array: \n").split(',')])
    c=np.array([int(j) for j in input("Enter your c array: \n").split(',')])
    v=np.array([int(j) for j in input("Enter your v array: \n").split(',')])
#     a=np.ones(4000000)
#     c=np.ones(3999999)
#     v=np.ones(4000000)
#     e=np.ones(3999999)
```

```
    if len(a) == len(v) and len(a) == len(e)+1 and len(a) == len(c)+1:
        break
    else:
        print("Your given arrays are not valid. Check your input arrays
             again.")

TridMult(a,c,e,v)
```

time complexity : $\mathcal{O}(5n)$

Part 2

A=LU

$$\begin{bmatrix} a_1 & c_1 & 0 \\ e_2 & a_2 & c_2 \\ 0 & e_3 & a_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \beta_2 & 1 & 0 \\ 0 & \beta_3 & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 & c_1 & 0 \\ 0 & \alpha_2 & c_2 \\ 0 & 0 & \alpha_3 \end{bmatrix}$$

$$\begin{bmatrix} a_1 & c_1 & 0 \\ e_2 & a_2 & c_2 \\ 0 & e_3 & a_3 \end{bmatrix} = \begin{bmatrix} \alpha_1 & c_1 & 0 \\ \alpha_1\beta_2 & \beta_2 c_1 + \alpha_2 & c_2 \\ 0 & \alpha_2\beta_3 & \beta_3 c_2 + \alpha_3 \end{bmatrix}$$

$a_1 = \alpha_1$

$\beta_2 = \frac{e_2}{\alpha_1}, \alpha_2 = a_2 - \beta_2 c_1$

$\beta_3 = \frac{e_2}{\alpha_2}, \alpha_3 = a_3 - \beta_3 c_2$

By looking at the values of α and β which are calculated above, we can see that such pattern has been formed:

$\beta_n = \frac{e_n}{\alpha_{n-1}}$

$\alpha_n = a_n - \beta_n c_{n-1}$

Part 3

```
import numpy as np
# import time as time



def LUdecomp(a,c,e):
    beta=np.zeros(len(c))
    alpha=np.zeros(len(a))
    alpha[0]=a[0]
#     tic=time.time()
    for i in range (len(e)):
        beta[i]=e[i]/alpha[i]
        alpha[i+1]=a[i+1]-beta[i]*c[i]
#     toc=time.time()
    print('alpha= ', alpha, '\n', 'beta= ', beta)
#    print('time= ', toc-tic)



```

```
while True:
    a=np.array([int(j) for j in input("Enter your a array, for instance:
        1,2,3,4,...: \n").split(',')])
    e=np.array([int(j) for j in input("Enter your e array:
        \n").split(',')])
    c=np.array([int(j) for j in input("Enter your c array:
        \n").split(',')])
#     a=np.ones(800000)*5
#     c=np.ones(799999)*3
#     e=np.ones(799999)*4
    if len(a) == len(e)+1 and len(a) == len(c)+1:
        break
    else:
        print("Your given arrays are not valid. Check your input arrays
            again.")


LUdecomp(a,c,e)
```

Part 4

time complexity : $\mathcal{O}(3n)$

Part 5

Ax = B → A = LU
LUx = B → Ly = B, Ux = y

$$\begin{bmatrix} 1 & 0 & 0 \\ \beta_2 & 1 & 0 \\ 0 & \beta_3 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

$$B_1 = y_1$$

$$B_2 = B_2 y_1 + y_2 \rightarrow y_2 = B_2 - B_2 y_1$$

$$B_3 = B_3 y_2 + y_3 \rightarrow y_3 = B_3 - B_3 y_2$$

Once again we see that a pattern has emerged:

$$y_n = B_n - B_n y_{n-1}, n \neq 0$$

Also:

$$\alpha_3 x_3 = y_3 \rightarrow x_3 = \frac{y_3}{\alpha_3}$$

$$\alpha_2 x_2 = y_2 \rightarrow x_2 = \frac{y_2 - c_2 x_3}{\alpha_2}$$

$$\alpha_1 x_1 = y_1 \rightarrow x_1 = \frac{y_1 - c_1 x_2}{\alpha_1}$$

Now if we reverse a,y,c, and x:

$$x_1 = \frac{y_1}{\alpha_1}$$

$$x_2 = \frac{y_2 - c_1 x_1}{\alpha_2}$$

$$x_3 = \frac{y_3 - c_2 x_2}{\alpha_3}$$

According to the calculations done above the following pattern would be the right algorithm:

$$x_n = \frac{y_n - c_{n-1} x_{n-1}}{\alpha_n}, n \neq 0$$

Each calculated answer needs to be reversed once again.

Part 6

```python
import numpy as np
# import time as time


def Linsolver(alpha, beta, c, B):
    y=np.ones(len(B))
    y[0]=B[0]
#     tic=time.time()
    for i in range(len(B)-1):
        y[i+1]=B[i+1]-beta[i]*y[i]
#     print('y=', y)
    y=y[::-1]
    c=c[::-1]
    alpha=alpha[::-1]
    x=np.ones(len(B))
    x[0]=y[0]/alpha[0]
    for i in range(len(y)-1):
        x[i+1]=(y[i+1]-c[i]*x[i])/alpha[i+1]
    x=x[::-1]
#     toc=time.time()
    print('x=', x)
#    print('time= ', toc-tic)

while True:

    alpha=np.array([int(j) for j in input("Enter your alpha array, for instance: 1,2,3,4,...: \n").split(',')])
    beta=np.array([int(j) for j in input("Enter your beta array: \n").split(',')])
    c=np.array([int(j) for j in input("Enter your c array: \n").split(',')])
    B=np.array([int(j) for j in input("Enter your right-hand-side array, B: \n").split(',')])
#     alpha=np.ones(10000000)
#     beta=np.ones(9999999)
#     B=np.ones(10000000)
#     c=np.ones(9999999)
    if len(alpha) == len(beta)+1 and len(alpha) == len(c)+1 and len(B) == len(alpha):
        break
    else:
        print("Your given arrays are not valid. Check your input arrays again.")
Linsolver(alpha, beta, c, B)
```

time complexity : $\mathcal{O}(5n)$

Part 7

$A = RR^T$, R is a lower triangular matrix.

$$R = \begin{bmatrix} \alpha_1 & 0 & 0 \\ \beta_1 & \alpha_2 & 0 \\ \epsilon_1 & \beta_2 & \alpha_3 \end{bmatrix}, R^T = \begin{bmatrix} \alpha_1 & \beta_1 & \epsilon_1 \\ 0 & \alpha_2 & \beta_2 \\ 0 & 0 & \alpha_3 \end{bmatrix}$$

$$RR^T = \begin{bmatrix} \alpha_1^2 & \alpha_1\beta_1 & \alpha_1\epsilon_1 \\ \alpha_1\beta_1 & \alpha_2^2 + \beta_1^2 & \alpha_2\beta_2 \\ \alpha_1\epsilon_1 & \alpha_2\beta_2 & \alpha_3^2 + \beta_2^2 \end{bmatrix} = \begin{bmatrix} a_1 & c_1 & 0 \\ c_1 & a_2 & c_2 \\ 0 & c_2 & a_3 \end{bmatrix}$$

$$\alpha_1\epsilon_1 = 0 \rightarrow \epsilon_1 = 0$$
$$\alpha_1^2 = a_1 \rightarrow \alpha_1 = \sqrt{a_1}$$
$$\alpha_1\beta_1 = c_1 \rightarrow \beta_1 = \frac{c_1}{\alpha_1}$$
$$\alpha_2^2 + \beta_1^2 = a_2 \rightarrow \alpha_2 = \sqrt{a_2 - \beta_1^2}$$
$$\alpha_2\beta_2 = c_2 \rightarrow \beta_2 = \frac{c_2}{\alpha_2}$$
$$\alpha_3 = \sqrt{a_3 - \beta_2^2}$$

According to the calculations above, we can extract the following equations:

$$\alpha_1 = \sqrt{a_1}$$
$$\beta_n = \frac{c_n}{\alpha_n}$$
$$\alpha_n = \sqrt{a_n - \beta_{n-1}^2}, n \neq 1$$

Part 8

```python

import numpy as np
# import time as time




def Rdecomp (MD, OD):
    alpha=np.ones(len(MD))
    beta=np.ones(len(OD))
    alpha[0]=(MD[0])**0.5
#     tic=time.time()
    for i in range(len(OD)):
        beta[i]= OD[i]/alpha[i]
        alpha[i+1] = (MD[i+1]-beta[i]**2)**0.5
#     toc=time.time()
    print('alpha= ', alpha, '\n', 'beta= ', beta)
#     print('time= ', toc-tic)



while True:
#     MD=np.array([int(j) for j in input("Enter your positive symetirc definite main diagonal, for instance: 1,2,3,4,...: \n").split(',')])
#     OD=np.array([int(j) for j in input("Enter your positive symetirc definite outer diagonal: \n").split(',')])
    MD=np.ones(32000000)*32
    OD=np.ones(31999999)*8
    if len(MD) == len(OD)+1:
        break
    else:
        print("Your given arrays are not valid. Check your input arrays again.")

try:
    Rdecomp(MD, OD)
except:
    print("Given matrix is not posibble to be decomposed to R and R transposed")
```

time complexity : $\mathcal{O}(4n)$