Tarea 4

Iván Vega Gutiérrez

16 de Octubre de 2021

1 Ejercicio 1

El objetivo de este ejercicio es comparar el desempeño de los métodos LU y Cholesky.

a) Escriba una función LUfacto que regrese las matrices L y U mediante el algoritmo 6.1. Si el algoritmo no se puede ejecutar (división entre 0), regrese un mensaje de error.

1.1 Solución

[1]:
```
import numpy as np

def LUfacto(A):
    """
    Esta función devuelve las matrices L, U, las cuales
    son la facotrización LU de A (A = LU). Donde L es una
    matriz triangular inferior y U es triangular superior.

    Argumentos:
    A(matriz cuadrada): Matriz que se desea factorizar.

    Salida:
    L(matriz cuadrada): Matriz triangular inferior con unos en su diagonal.
    U(matriz cuadrada): Matriz triangular superior.

    Excepciones:
    El algoritmo funciona si las submatrices de A son invertibles.
    "División entre cero": si algún pivote es cero.
    """
    # Hallamos la dimension de la matriz
    n = A.shape[0]
    #Creamos la matriz U compuesta por ceros
    U = np.zeros((n,n))
    #La matriz L tendrá unos en su diagonal
    L = np.identity(n)
    #La primera fila de U será igual a la primera fila de A
    U[0] = A[0]
    #Verificamos que no intentemos una división entre cero
    if A[0][0] == 0:
```

```
        print("Error de ejecución: División entre cero")
        return
    else:
        #Hallamos la primera columna de L
        L[:,0] = A[:,0]/A[0][0]
        #Este ciclo recorrerá las filas i
        for i in range(1,n):
            #El siguiente ciclo recorrerá las columnas j
            for j in range(1,n):
                #Si la siguiente condición se cumple entonces llenaremos la␣
↪matriz L
                if i>j:
                    suma = 0
                    for k in range(j):
                        suma += L[i][k]*U[k][j]
                    #Verificamos que no exista una división entre cero
                    if U[j][j] == 0:
                        print("Error de ejecución: Divisi{on entre cero")
                        return
                    else:
                        L[i][j] = (A[i][j] - suma)/U[j][j]
                #Si i >= j entonces se llenará la matriz U
                else:
                    suma = 0
                    for k in range(i):
                        suma += L[i][k]*U[k][j]
                    U[i][j] = A[i][j] - suma
    return (L,U)
```

b) Escriba una función llamada Cholesky que regrese la matriz B calculada por el algoritmo 6.2. Si el algoritmo no se puede ejecutar(matriz no simetrica, división entre 0, raíz cuadrada negativa), regrese un mensaje de error. Compare la función con la función chol de Matlab.

1.2 Solución

[2]:
```
import numpy as np
import math #Para usar la función sqrt

def Cholesky(A):
    """
    Esta funcion regresa una matriz triangular inferior B
    tal que la matriz A se puede expresar como el producto
    de B y la transpuesta de B (factorizacion de Cholesky)

    Argumentos:
    A(matriz cuadrada): A es una matriz simetrica
```

```
    Salida:
    B(matriz): B es una matriz triangular inferior.

    Excepciones:

    El algoritmo solo funciona si la matriz es simetrica positiva definida.

    "Division entre cero": si A no es positiva definida
    "Raiz cuadrada de un numero negativo": si A no es positiva definida
    "Resultado erroneo":  si A no es simetrica
    """
    #Creamos la matriz B
    n = A.shape[0]
    B = np.zeros((n,n))
    #Verificamos si la matriz es simetrica
    #NOTA: Esta parte está comentada para que exista solución al siguiente␣
↪ejercicio.
    #for i in range(n):
    #    for j in range(i+1, n):
    #        print(A[i][j])
    #        print(A[j][i])
    #        if A[i][j] != A[j][i]:
    #            return "La matriz no es simetrica"
    #Hallamos los valores de la j-esima columna de B
    for j in range(n):
        suma = 0
        #El siguiente ciclo es para hallar los valores de la diagonal
        for k in range(j):
            suma += B[j][k]*B[j][k]
        #Verificamos que no se calculen raíces cuadradas negativas
        if A[j][j] - suma < 0:
            print("Error de ejecución: Raíz cuadrada negativa")
            return
        else:
            #Asignamos el valor de B en la diagonal
            B[j][j] = math.sqrt(A[j][j] - suma)
        #Hallamos los valores de la iésima fila de B (por debajo del pivote)
        for i in range(j+1,n):
            suma = 0
            for k in range(j):
                suma += B[j][k]*B[i][k]
            #Verificamos que no existan divisiones entre cero
            if B[j][j] == 0:
                print("Error de ejecución: División entre cero")
                return
            else:
                #Asigamos el valor de B en la i-esima fila
```

```
                B[i][j] = (A[i][j] - suma)/B[j][j]
    return B
```

c) Para n = 10, 20, . . . , 100, definimos una matriz A=PdSMat(n) y un vector b=ones(n,1). Compare:

- Por un lado, el tiempo de ejecución para calclular las matrices L y U dadas por la función LUfacto, entonces la solución x del sistema Ax=b.Use las funciones BackSub y ForwSub.
- Por otro lado, el tiempo de ejecución para calcular la matriz B dada por la función Cholesky, entonces la solución x del sistema Ax=b. Use la función BackSub y ForwSub. Grafique sobre la misma gráfica las curvas que representan los tiempos de ejecución en términos de n. Comente.

1.3 Solución

Primero, para construir la función PdSMat(n) construimos la función SymmetricMat, la cual nos devuelve una matriz aleatoria simétrica de dimensión n. Para esta función se utiliza la lbrería random para generar números de manera aleatoria.

[3]:
```
import numpy as np
import random

def SymmetricMat(n):
    """
    Esta función devuelve una matriz simetrica de tamaño
    nxn.

    Argumentos:
    n (entero positivo): Dimensión de la matriz que deseamos crear

    Observaciones:
    Los elementos de la matriz están entre 0 y 1
    """
    #Creamos la matriz que deseamos hallar
    A = np.empty((n,n)) #Creamos una matriz de tamaño n
    #Rellenamos la matriz con valores aleatorios
    for i in range(n):
        A[i][i] = random.random()
        #El siguiente ciclo es para forzar que la matriz sea simétrica
        for j in range(i+1,n):
            A[i][j] = random.random()
            A[j][i] = A[i][j]
    return A
```

A continuación se muestra la función PdSMat(n), la cual nos regresa una matriz positiva definida de dimension n.

[4]:
```
import numpy as np

def PdSMat(n):
    """
    Esta función regresa una matriz positiva definida de
    dimensión n.

    Argumentos:
    n(entero) : dimensión de la matriz que desamos construir

    Observaciones:
    Se necesita la función SymmetricMat(n)
    """
    #Creamos una matriz simetrica A
    A = SymmetricMat(n)
    #Hallamos los valores y vectories propios de A, respectivamente
    D,P = np.linalg.eig(A)
    #Sacamos el valor absoluto de cada valor propio
    D = abs(D)
    #Pasamos el vector D a una matriz cuya diagonal tiene los valores propios.
    D = D*(np.identity(n))
    #A la diagonal de la matriz D sumamos la norma (euclidiana(2)) de D
    D = D + (np.linalg.norm(D,2))*(np.identity(n))
    #Obtenemos la matriz buscada qu es el producto de las matrices P, D y la␣
 ↪inversa de P.
    A = P.dot((D.dot(np.linalg.inv(P))))
    return A
```

Las funciones ForwSub y BackSub, sirven para hallar la solución de sistemas lineales cuando las matrices son triangulares inferior y superior, respectivamente.

[5]:
```
def ForwSub(A,b):
    """
    Esta función calcula la solución de un sistema Ax=b,
    mediante sustitución hacia adelante, cuando A es una
    matriz triangular inferior.

    Argumentos:
    A (matriz cuadrada): Matriz triangular inferior.
    b (vector): Vector de términos independientes.
    """
    #Creamos el vector que contendrá los valores de la solución.
    n = A.shape[0]
    x = np.empty(shape=n)
    #Hallamos el valor de cada x_i
    for i in range(n):
        suma = 0
```

```
        #Este ciclo utiliza el valor de x_i, para hallar x_i+1
        for j in range(i):
            suma += A[i][j]*x[j]
        x[i] = (b[i] - suma)/ A[i][i]
    return x
```

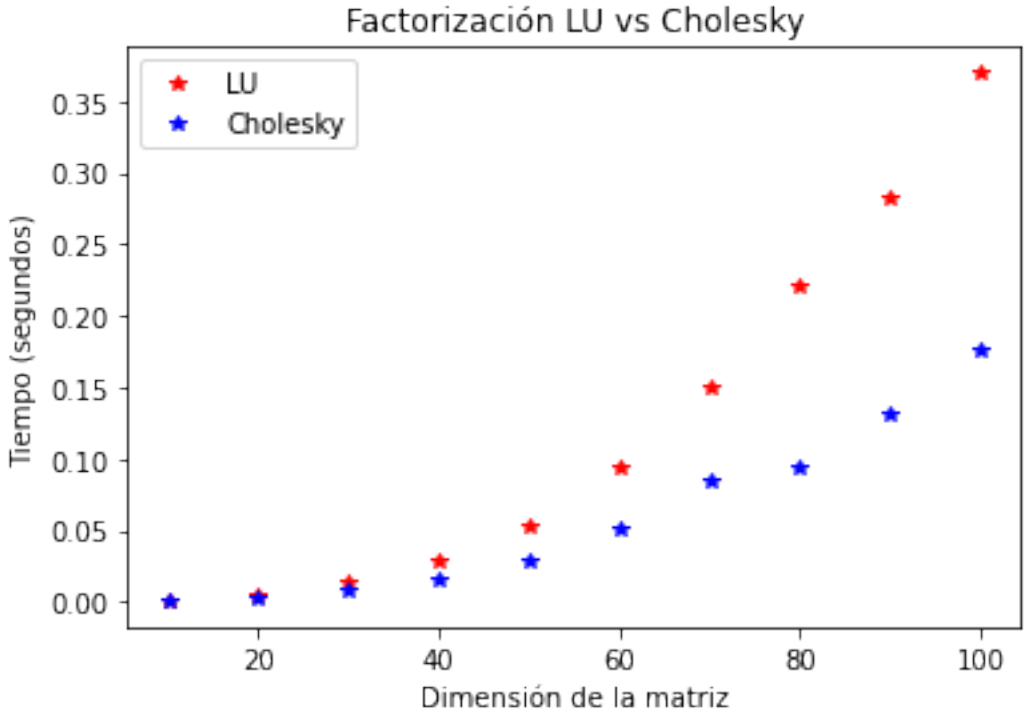
[6]:
```
def BackSub(A,b):
    """
    Esta función calcula la solución de un sistema Ax=b,
    mediante sustitución hacia atrás, cuando A es una matriz
    triangular superior.

    Argumentos:
    A (matriz cuadrada): Matriz triangular superior.
    b (vector): Vector de términos independientes.
    """
    #Creamos el vector que contendrá los valores de la solución.
    n =  A.shape[0]
    x = np.empty(shape=n)
    #Hallamos el valor de la variable x_i (empezando por x_n)
    for i in range(n-1, -1, -1):
        suma = 0
        #A partir del valor x_i hallamos el valor de x_i-1
        for j in range(n-1, i, -1):
            suma += A[i][j]*x[j]
        x[i] = (b[i] - suma) / A[i][i]
    return x
```

Comparación entre los métodos LU y Cholesky

[7]:
```
import numpy as np
import matplotlib.pyplot as plt
from timeit import default_timer


#Tiempo de ejecución y solución del método LU
tiempo1 = []
X1 = []
#Tiempo de ejeción y solución del método Cholesky
tiempo2 = []
X2 = []
#Dimensión de las matrices
NumDatos = []
#Errores del método LU y Cholesky
ErrorLU = []
ErrorChol = []
```

```
for n in range(10, 101, 10):

    A = PdSMat(n)
    b = np.ones(n)
    NumDatos.append(n)

    #Método Solve
    SOL = np.linalg.solve(A,b)


    #Método LU
    inicio_1 = default_timer()
    #Factorizacion LU
    L,U = LUfacto(A)
    #Solución por LU
    X1 = BackSub(U, ForwSub(L,b))
    fin_1 = default_timer()
    #Acumulamos los tiempos de ejecución
    tiempo1.append(fin_1 - inicio_1)

    #Método de Cholesky
    inicio_2 = default_timer()
    #Factorizacion de cholesky
    B = Cholesky(A)
    #Solución por Cholesky
    X2 = BackSub(np.transpose(B), ForwSub(B,b))
    fin_2 = default_timer()
    #Acumulamos los tiempos de ejecución
    tiempo2.append(fin_2 - inicio_2)

    #Calculamos el error con respecto al método solve
    error1 = 0
    error2 = 0
    for i in range(n):

        error1 += abs(SOL[i] - X1[i])
        error2 += abs(SOL[i] - X2[i])

    #Vectores que contienen los errores en cada iteración
    ErrorLU.append(error1)
    ErrorChol.append(error2)


#Graficamos los tiempos de ejecución
plt.plot(NumDatos, tiempo1,'r*', label='LU')
plt.plot(NumDatos, tiempo2, 'b*', label='Cholesky')
plt.title("Factorización LU vs Cholesky")
```

```
plt.xlabel("Dimensión de la matriz")
plt.ylabel("Tiempo (segundos)")
plt.legend()
plt.show()

print("El tiempo de ejecución del método LU es {} segundos".format(sum(tiempo1)))
print("El tiempo de ejecución del método Cholesky es {} segundos \n".
 ↪format(sum(tiempo2)))

#Comparamos las soluciones
print("El error del método LU es {} ".format(sum(ErrorLU)/10))
print("El error del método Cholesky es {} ".format(sum(ErrorChol)/10))
contLU = 0
contChol = 0
for i in range(10):
    if ErrorLU[i] < ErrorChol[i]:
        contLU += 1
    else:
        contChol+= 1
print("El método LU de los 10 casos tuvo una mejor aproximación en {} casos ".
 ↪format(contLU))
```




```
El tiempo de ejecución del método LU es 1.2190294580000227 segundos
El tiempo de ejecución del método Cholesky es 0.5915909569999371 segundos
```

```
El error del método LU es 3.1866870253693944e-16
El error del método Cholesky es 1.7690710008011478e-15
El método LU de los 10 casos tuvo una mejor aproximación en 10 casos
```

A partir de la gráfica, se puede observar que el método de Cholesky es más rápido que el método LU (prácticamente el doble de rápido), sin embargo al comparar los errores con respecto al método solve que implementa Python, observamos que el método LU tiene una mejor aproximación a la solución, más aún, en los 10 casos de prueba el método LU tuvo un mejor aproximación, pero el método de Cholesky fue más rápido.

2 Ejercicio 2

El objetivo de este ejercicio es evaluar la influencia de las filas de permutación en la eliminación Gausiana. Sean A y b definidas por

e=1.E-15;

A=[e 1 1: 1 -1 1; 1 0 1];

b=[2 0 1]

[8]:
```
e = .000000000000001
A = np.array([[e, 1, 1], [1, -1, 1], [1, 0, 1]])
b= np.array([2, 0, 1])
print(A)
```

```
[[ 1.e-15  1.e+00  1.e+00]
 [ 1.e+00 -1.e+00  1.e+00]
 [ 1.e+00  0.e+00  1.e+00]]
```

a) Calcule las matrices L y U usando la función LUfacto.

2.1 Solución

[9]:
```
L, U = LUfacto(A)
print("La matriz L es:\n {} \n".format(L))
print("La matriz U es:\n {}".format(U))
```

```
La matriz L es:
 [[1.e+00 0.e+00 0.e+00]
 [1.e+15 1.e+00 0.e+00]
 [1.e+15 1.e+00 1.e+00]]

La matriz U es:
 [[ 1.e-15  1.e+00  1.e+00]
 [ 0.e+00 -1.e+15 -1.e+15]
 [ 0.e+00  0.e+00 -1.e+00]]
```

b) Definimos dos matrices l y u por [l u]= LUfacto(P*A), donde P es la matriz de permutaciones definida por la instrucción [w z p]=lu(A).

Muestre las matrices l y u ¿Qué puede observar?

2.2 Solución

[10]:
```
import scipy.linalg as la
#Hallamos la matriz de permutaciones P
P, L, U = la.lu(A)
l, u = LUfacto(P.dot(A))
print("La matriz de permutaciones P es:\n {} \n".format(P))
print("La matriz l es:\n {} \n".format(l))
print("La matriz u es:\n {} \n".format(u))
```

```
La matriz de permutaciones P es:
 [[0. 1. 0.]
 [1. 0. 0.]
 [0. 0. 1.]]

La matriz l es:
 [[1.e+00 0.e+00 0.e+00]
 [1.e-15 1.e+00 0.e+00]
 [1.e+00 1.e+00 1.e+00]]

La matriz u es:
 [[ 1. -1.  1.]
 [ 0.  1.  1.]
 [ 0.  0. -1.]]
```

En primer instancia, tenemos que la matriz de permutaciones lo que hace es intercambiar las filas 1 y 2 de la matriz A, por lo tanto tendríamos como primer pivote al 1 en lugar del valor e, el cual es muy pequeño. Es justamente este cambio el que hace que la factorización lu sea más estable que la factorización LU, esto se observa claramente en las matrices U y u. Por un lado U maneja cantidades que nos pueden llevar a un error de redondeo, mientras que u tiene valores enteros, todo esto por tomar como pivote al número e.

c) Determine la solución del sistema Ax=b calculado por la instrucción BackSub(U,ForwSub(L,b)), entonces la solución calculada por la instrucción BackSub(u, ForwSub(1, P*b)). Compare con la solución exacta= (0,1,1)t. Concluya

2.3 Solución

[11]:
```
sol1 = BackSub(U, ForwSub(L,b))
sol2 = BackSub(U, ForwSub(l, P.dot(b)))
print("La solución tomando en cuenta las matrices L y U es:\n {}\n".format(sol1))
print("La solución tomando en cuenta las matrices l y u es:\n {}\n".format(sol2))
```

```
La solución tomando en cuenta las matrices L y U es:
 [-4.4408921e-16 -1.0000000e+00  1.0000000e+00]
```

```
La solución tomando en cuenta las matrices l y u es:
 [-2.22044605e-16  1.00000000e+00  1.00000000e+00]
```

Observamos que la solución tomando en cuenta la factorización lu es mejor que utilizar LU, lo cual era de suponerse, pues en el inciso anterior notamos que la estabilidad del sistema considerando la matriz de permutaciones nos evitaría errores de redondeo. Aunque, realmente la diferencia de las soluciones es mínima, ambas soluciones se aproximan muy bien a la solución exacta.

3 Ejercicio 3

Calcular el determinante de una matriz triangular.

3.1 Solución

Sea $A = (a_{i,j})_{1 \leq i,j \leq n}$, una matriz triangular. Sin pérdida de generalidad, supongamos que A es una matriz triangular superior. Por definición sabemos que,

$$\begin{aligned} \det(A) &= \sum_{\sigma \in S_n} \epsilon(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)} \\ &= \sum_{\sigma \in S_n} \left(a_{1,\sigma(1)} a_{2,\sigma(2)} a_{3,\sigma(3)}, \cdots, a_{n,\sigma(n)} \right) \end{aligned} \tag{1}$$

Sea $\sigma' \in S_n$, tal que $\sigma'(i) = j$ con $i \neq j$, para todo $i, j \in \{1, 2, \cdots, n\}$. Notemos que si $i < j$, entonces

$$a_{i,\sigma'(i)} = a_{i,j} = 0.$$

Por lo tanto,

$$\prod_{i=1}^{n} a_{i,\sigma'(i)} = 0.$$

Por otro lado, si $j > i$, se tendría que,

$$a_{j,\sigma'(j)} = a_{j,i} = 0.$$

Por lo tanto,

$$\prod_{i=1}^{n} a_{i,\sigma'(i)} = 0.$$

De lo anterior, se concluye que la única permutación que no nos conduce a tener un producto igual a cero es la permutación $\sigma(i) = i$, para $1 \leq i \leq n$. Así de (1), se sigue que

$$\det(A) = a_{1,1} a_{2,2}, \cdots, a_{n,n}.$$