César Isaí García Cornejo
cesar.cornejo@cimat.mx
Computo Científico

Tarea 1

Semestre 3

CIMAT

Ejercicio 1
Implementar los algoritmos de *Backward* y *Forward substitution*.

Solución:

Para poder obtener un algoritmo que ejecute una **substitución *forward*** es necesario un sistema de ecuaciones lineales cuya matriz es triangular inferior. Es decir, el sistema se puede expresar de la forma

$$Ly = b \tag{1}$$

donde L es una matriz triangular inferior. Otra forma de verlo es por medio de la expansión

$$\begin{pmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{m1} & l_{m2} & l_{m3} & \cdots & l_{mm} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_m \end{pmatrix}$$

Así tenemos,

$$\begin{aligned} y_1 &= \frac{b_1}{l_{11}}, \\ y_2 &= \frac{b_2 - l_{21}y_2}{l_{22}}, \\ &\vdots \\ y_j &= \frac{(b_j - l_{j1}y_1 - l_{j2}y_2 - \cdots - l_{j,j-1}y_{j-1})}{l_{jj}}, \qquad 2 \leq j \leq m. \end{aligned} \tag{2}$$

Para la implementación de dicho algoritmo es necesario anidar dos ciclos for. Se crea una variable auxiliar llamada aux, la cual el ciclo for interno suma los terminos $l_{j1}y_1 + l_{j2}y_2 + \cdots + l_{j,j-1}y_{j-1}$ para posteriormente usar (2)

```
for i in range(len(b)):
    aux = 0
    for j in range(i):
        aux += L[i,j]*y[j]    #Auxiliar variable
    y[i] = (b[i] - aux)/L[i,i]    # 'y' is the output and solution to the system of
        equations.
```

Por otro lado, la implementación de la **substitución *backward*** es análogo al algoritmo previo. Consideremos el sistema de ecuaciones lineales

$$Ux = b \tag{3}$$

donde U es una matriz triangular superior. La expansión es de la forma

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & u_{m-2,m-2} & u_{m-2,m-1} & u_{m-2,m} \\ 0 & 0 & 0 & u_{m-1,m-1} & u_{m-1,m} \\ 0 & 0 & 0 & 0 & u_{mm} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{m-2} \\ x_{m-1} \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_{m-2} \\ b_{m-1} \\ b_m \end{pmatrix}$$

De esta forma

$$\begin{aligned} x_{mm} &= \frac{b_m}{u_{mm}}, \\ x_{m-1,m-1} &= \frac{b_{m-1} - u_{m-1,m}x_m}{u_{m-1,m-1}}, \\ &\vdots \\ x_{i,i} &= \frac{b_{m-1} - u_{i,m}x_m - u_{i,m-1}x_{m-1} - \cdots - u_{i,m-i-1}x_{m-i-1}}{u_{i,i}} \end{aligned} \tag{4}$$

La implementación del algoritmo para la substitución *backward* es análago. La diferencia sustancial entre ambos se encuetra en que los indices de los vectores deben de correr al reves. Dicha implementación se fundamenta en el siguiente código.

```
for i in range(m):
        aux = 0
        for j in range(i):
            aux += U[m-i-1,m-j-1] * x[m-j-1]       #Auxiliar variable
        x[m-i-1] = (b[m-i-1] - aux)/U[m-i-1,m-i-1]     # Form obtained by 'despeje' of the
             variable of interest.
```

Lo que completa la implementación de ambos métodos.

Ejercicio 2
Implementar el algoritmo de eliminación Gaussiana con pivoteo parcial LUP, 21.1 del Trefethen (p. 160).

Solución:
La implementación es directa del pseudocódigo. Sea A una matriz numpy. Se crean tres matrices vacias (nulas) donde se almacenarán a L,U y P.

```
U = A.copy()
L = np.identity(m,dtype= float)
P = np.identity(m,dtype= float)
```

Es importante resaltar que la matriz U requiere ser una copia de A. Sin el metodo copy() el lenguage solamente haría referencia a la dirección de memoria y esta se modificaría.

La generalización de la factorización LU con pivoteo se sigue de una serie de modificaciones iterativas (cambios de renglos) dentro del primer for anidado. Es entonces que se pide encontrar el indice de la fila que hace que el valor k-ésimo de la columna se maximice. Esto se implementó con el siguiente código.

```
    for k in range(m-1):

    Index_max = k
    maxi = U[k,k]

        for i in range(k, m):  #This cicle for stan for check the maximum of the k-column
            .
            if abs(U[i,k]) > abs(maxi):
                maxi = U[i,k]
                Index_max = i
```

Lo siguiente a realizar es un cambio de renglones en las matrices L,U y P. Los cambios de L y P son directos ya que se cambia todo el renglon. Sin embargo, los cambios de L son más delicados pues solo se cambia una parte de cada renglon. Esto con el código siguiente:

```
    AuxRowL = L[k,: k].copy()
    L[k,: k] = L[Index_max,: k]
    L[Index_max,: k] = AuxRowL
```

Posteriormente se finaliza el método al anidar el segundo ciclo for mismo que en la factorización LU convencional. Los detalles se encuentra en el script adjunto.

Ejercicio 3
Dar la descomposición LUP para una matriz aleatoria de entradas U(0,1) de tamaño 5×5, y para la matriz

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{pmatrix} \tag{5}$$

Solución:

Para la matriz dada, se manda a llamar la función $LUP(A)$ donde la matriz A es la función a factorizar.

El resultado de la descomposición $PA = LU$ es

$$A = \begin{bmatrix} 1. & 0. & 0. & 0. & 0. \\ -1. & 1. & 0. & 0. & 0. \\ -1. & -1. & 1. & 0. & 0. \\ -1. & -1. & -1. & 1. & 0. \\ -1. & -1. & -1. & -1. & 1. \end{bmatrix} \begin{bmatrix} 1. & 0. & 0. & 0. & 1. \\ 0. & 1. & 0. & 0. & 2. \\ 0. & 0. & 1. & 0. & 4. \\ 0. & 0. & 0. & 1. & 8. \\ 0. & 0. & 0. & 0. & 16. \end{bmatrix} \tag{6}$$

donde para este caso particular P es la matriz identidad.

Usando la paqueteria de Scipy, se implementó el generador de matrices y vectores con entradas aleatorios con distribución uniforme (0,1). Además, se ajusto una *semilla* para mayor control de la matriz aleatoria. La matriz generada es:

$$D = \begin{bmatrix} 0{,}15416284 & 0{,}7400497 & 0{,}26331502 & 0{,}53373939 & 0{,}01457496 \\ 0{,}91874701 & 0{,}90071485 & 0{,}03342143 & 0{,}95694934 & 0{,}13720932 \\ 0{,}28382835 & 0{,}60608318 & 0{,}94422514 & 0{,}85273554 & 0{,}00225923 \\ 0{,}52122603 & 0{,}55203763 & 0{,}48537741 & 0{,}76813415 & 0{,}16071675 \\ 0{,}76456045 & 0{,}0208098 & 0{,}13521018 & 0{,}11627302 & 0{,}30989758 \end{bmatrix}$$

Su descomposición LU es tal que

$$P = \begin{bmatrix} 0. & 1. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 1. & 0. & 0. \\ 1. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 1. & 0. \end{bmatrix},$$

$$L = \begin{bmatrix} 1. & 0. & 0. & 0. & 0. \\ 0{,}83217735 & 1. & 0. & 0. & 0. \\ 0{,}30892983 & -0{,}44984958 & 1. & 0. & 0. \\ 0{,}16779684 & -0{,}80811921 & 0{,}35073561 & 1. & 0. \\ 0{,}56732269 & -0{,}05631829 & 0{,}48102103 & -0{,}24994986 & 1. \end{bmatrix},$$

$$U = \begin{bmatrix} 0{,}91874701 & 0{,}90071485 & 0{,}03342143 & 0{,}95694934 & 0{,}13720932 \\ 0. & -0{,}7287447 & 0{,}10739762 & -0{,}68007855 & 0{,}1957151 \\ 0. & 0. & 0{,}98221304 & 0{,}2511723 & 0{,}04791354 \\ 0. & 0. & 0. & -0{,}26451328 & 0{,}13290782 \\ 0. & 0. & 0. & 0. & 0{,}10407 \end{bmatrix}$$

lo que concluye el ejercicio.

Ejercicio 4
Usando la descomposición LUP anterior, resolver el sistema de la forma

$$Dx = b \tag{7}$$

donde D son las matrices del problema 3, para 5 diferentes b aleatorios con entradas $U(0,1)$. Verificando si es posible o no resolver el sistema.

Solución:
Para resolver el sisteme $Ax = b$ con el método de descomposición LU con pivoteo necesitamos hacer manipulaciones algebraicas. Notemos que

$$\begin{aligned} Ax &= b, \\ PAx &= Pb, \\ LUx &= Pb, \\ Ly &= Pb, \end{aligned}$$

donde

$$Ux = y$$

Por tanto, es necesario resolver con *backward substitution* para obtener y, luego con *forward substitution* se puede resolver para x. Esto siempre y cuando el sistema tenga solución única.

La función definida en el script, a manera de sumario es

```
def LinearSystem(A,b):
    try:
        L,U,P = LUP(A)

        y = forward(L,Pb)x = backward(U,y)except:print("The system of equations do not have a unique
            solution")return x
```

que intenta resolver el sistema por medio de la factorización LU con pivoteo. Si se da el caso de que la matriz es degenerada o cercana a una degenerada, entonces saltará un error e imprimirá que no hay solución.

Debido a que son 5 vectores b aleatorios, solo presentaremos uno y el resto se podrán obtener tras compilar el código correspondiente.

Para el sistema $Ax = b$

$$\begin{bmatrix} 1. & 0. & 0. & 0. & 1. \\ -1. & 1. & 0. & 0. & 1. \\ -1. & -1. & 1. & 0. & 1. \\ -1. & -1. & -1. & 1. & 1. \\ -1. & -1. & -1. & -1. & 1. \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0{,}15416284 \\ 0{,}7400497 \\ 0{,}26331502 \\ 0{,}53373939 \\ 0{,}01457496 \end{bmatrix}$$

donde la solución es dada por

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} -0{,}17511503 \\ 0{,}2356568 \\ -0{,}00542108 \\ 0{,}25958222 \\ 0{,}32927787 \end{bmatrix}$$

Para matrices aleatorias funciona igual.

Ejercicio 5
Implementar el algoritmo de descomposición de Cholesky 23.1 del Trefethen (p. 175).

Solución:

La descomposición de Cholesky funciona exclusivamente para matrices hermitianas (simétricas) y positivas definidas. Consideremos la matriz A y construyamos una matriz con las carácteristicas anteriormente mencionadas. Tomemos $D = A^*A$. Es claro que $D* = A^*A$, es decir $D^* = D$ una matriz hermitiana. Luego tomemos un vector $x \in \mathbb{C}^m$ e investiguemos

$$\begin{aligned} x^*Dx &= x^*A^*Ax, \\ &= (Ax)^*Ax, \\ &= ||Ax||^2 \geq 0 \end{aligned}$$

para $x \neq 0$ define una matrix definida positiva.

La implementación es directa del pseudocódigo. A manera de sumario:

```
def cholesky(D):

    import numpy as np
    R = D.copy()
    m = np.shape(D)[0]

    for k in range(m):
        for j in range(k+1,m):

            R[j,j:] = R[j,j:] - R[k,j:] * (R[k,j]/R[k,k])

        R[k,k:] = R[k,k:]/np.sqrt(R[k,k])
```

debido a que el algoritmo reescribe sobre la matriz R, entonces la diagonal inferor debería anularse por lo que es necesario construir un algoritmo para la limpia de dicho R. Esto simpliemente con el comando

```
    for i in range(m):
        for j in range(i):
            R[i,j] = 0
```

que anula cada elemento debajo de la identidad.

Usando la matrix (5) podemos construir de la forma ya mencionada la matriz de entrada a Cholesky. La matriz D es

$$D = \begin{bmatrix} 5. & 2. & 1. & 0. & -3. \\ 2. & 4. & 1. & 0. & -2. \\ 1. & 1. & 3. & 0. & -1. \\ 0. & 0. & 0. & 2. & 0. \\ -3. & -2. & -1. & 0. & 5. \end{bmatrix}$$

cuya factorización es $D = R^*R$ con

$$R = \begin{bmatrix} 2{,}23606798 & 0{,}89442719 & 0{,}4472136 & 0. & -1{,}34164079 \\ 0. & 1{,}78885438 & 0{,}3354102 & 0. & -0{,}4472136 \\ 0. & 0. & 1{,}63935963 & 0. & -0{,}15249857 \\ 0. & 0. & 0. & 1{,}41421356 & 0. \\ 0. & 0. & 0. & 0. & 1{,}72532437 \end{bmatrix}$$

lo que concluye la implementación.

Ejercicio 6
Comparar la complejidad de su implementación de los algoritmos de factorización de Cholesky y LUP mediante la medición de los tiempos que tardan con respecto a la descomposición de una matriz aleatoria hermitiana definida positiva. Graficar la comparación.

Solución:

Mediante la paquetería time podemos medir la *hora* en nanoseundos. Ahora buscamos medir el tiempo de ejecución de un programa en función del tamaño de la matriz de entrada para cada uno de los dos métodos de factorización vistos.

Con el código podemos guardar el resultado del tiempo de ejecución

```
inicio = time.time()
LUP(A)
fin = time.time()

tiempoLU[i] = fin - inicio
```

para el caso de LUP. El caso Cholesky es similar. Luego graficamos dichos tiempos para matrices que van desde $\mathbb{R}^{1\times 1}$ hasta $\mathbb{R}^{600\times 600}$

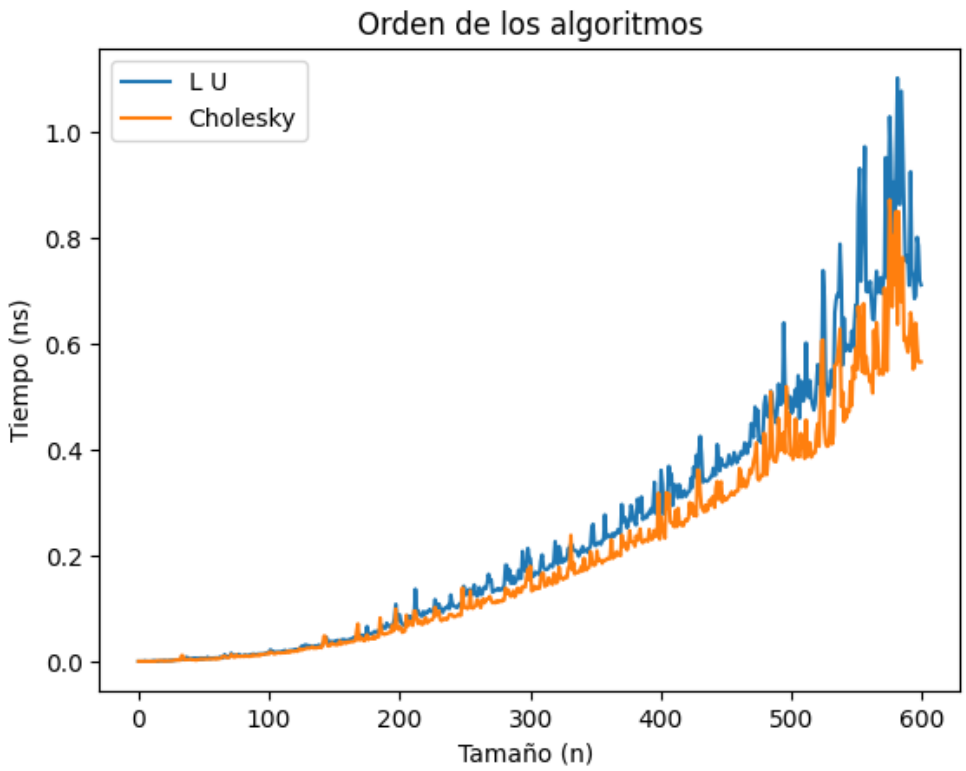

Figura 1: Tiempo de ejecución de los métodos de factorización LU y Cholesky

Observemos que ambos procesos parecen tener el tiempo creciente de orden polinomial. También podemos notar que el método LU es mas complejo pues usa mas operaciones que el método de Cholesky.