Cómputo Científico para Probabilidad, Estadística y Ciencia de Datos

Ezau Faridh Torres Torres

TAREA 1: Descomposición LU y Cholesky.

Fecha de entrega: 04/Sep/2024.

NOTA: Los ejercicios se encuentran repartidos en los archivos:

- ejercicio1.py
- ejercicio2.py
- ejercicio3y4.py
- ejercicio5.py
- ejercicio6.py

---

1. Implementar los algoritmos de Backward y Forward substitution.

En el archivo ejercicio1.py se implementan las funciones *FORWARD_SUBST()* y *BACKWARD_SUBST()*, las cuales se usan para resolver sistemas de ecuaciones lineales para matrices triangulares inferiores y superiores, respectivamente. Ambas funciones reciben como argumentos dos arreglos de *numpy*: la matriz A y el vector b del sistema $Ax = b$ que se desea resolver (A es triangular superior o inferior, según sea el caso) y devuelven otro arreglo de *numpy* que es el vector solución x. Además, se incluye un par de ejemplos de uso de las funciones y se imprimen también las soluciones dadas por el resolvedor de *numpy*.

Dado que las matrices a tratar son triangulares y cuadradas de $n \times n$, las funciones operan de la siguiente forma iterativa: despejan el término del extremo (x_1 o x_n) y a partir de ahí, despejan de forma progresiva o regresiva a los términos faltantes en términos de los ya conocidos haciendo uso de las expresiones derivadas en clase:

Backward:

$$x_n = \frac{b_n}{u_{nn}} \implies x_i = \frac{1}{u_{ii}}\left(b_i - \sum_{j=i+1}^{n} u_{ij}x_j\right) \text{ para } i = n-1, \dots, 1. \tag{1}$$

Fordward

$$x_1 = \frac{b_1}{u_{11}} \implies x_i = \frac{1}{u_{ii}}\left(b_i - \sum_{j=2}^{i-1} u_{ij}x_j\right) \text{ para } i = 2, \dots, n. \tag{2}$$

---

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

En el archivo ejercicio2.py se da la función *LUP()*, la cual implementa el algoritmo de eliminación gaussiana con pivoteo parcial LUP. Tal función recibe como argumento un arreglo de numpy, el cual es la matriz A que nos interesa factorizar y devuelve tres arreglos de numpy:

- L es una matriz triangular inferior.

- U es una matriz triangular superior.
- P es una matriz de permutación

y cumplen que $PA = LU$.

También se incluyó un par de ejemplos de uso de la función *LUP()* y se verificó que estuviera correcto revisando que la norma de la matriz $PA - LU$ fuera muy cercana a cero gracias a la función *linalg.norm()* de *numpy*.

---

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

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{pmatrix} \tag{3}$$

En el archivo ejercicio3y4.py se usan las funciones creadas en los ejercicios anteriores. Se hace uso de la función *random.rand()* de *numpy* para generar la matriz $U(0,1)$ ya que de esta forma podemos usar una semilla para reproducir los datos dados en este reporte.

Se aplicó la función *LUP()* a $U(0,1)$ y a la matriz A, generando sus respectivas matrices L, U y P y se verificó que cumplieran $PA = LU$, esto se hizo como antes: que la norma de la matriz $PA - LU$ fuera muy cercana a cero en cada caso y se obtuvieron resultados satisfactorios.

Nota: En muchos casos, al generar de la matriz $U(0,1)$, existían errores al obtener las matrices L, U y P.

---

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

$$Dx = b$$

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

Del ejercicio anterior, se tienen matrices L, U y P tales que, para la matriz D se tiene la factorización $PD = LU$. Se quiere resolver el problema $Dx = b$, esto es equivalente a resolver $PDx = Pb$ ya que P es una matriz de permutación. A la vez, esto equivale a resolver $LUx = Pb$.

Haciendo $z = Ux$, se obtiene el problema $Lz = Pb$ donde ya sabemos que L es triangular inferior y se puede usar *forward substitution* para resolver para z gracias a la función *FORWARD_SUBST()* del ejercicio 1. Una vez resuelto, se obtiene el vector solución z y se puede resolver $Ux = z$ para x usando *BACKWARD_SUBST()* gracias a que U es triangular superior. Se repitió este procedimiento para 5 vectores b aleatorios para cada y se obtuvieron los siguientes resultados:

- A: para los 5 vectores aleatorios b se obtuvo solución, esto se verificó revisando que la solución obtenida coincidiera con la solución del solucionador de *numpy: linalg.solve()* y que la norma de los vectores $Ax - b$ fuera bastante pequeña.

- $U(0,1)$ Para ninguno de los 5 vectores aleatorios b se obtuvo solución.

---

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

En ejercicio5.py se implementa la función *LU_cholesky()*, la cual da la descomposición de Cholesky. La función toma como argumentos a un arreglo de *numpy*, el cual debe ser una matriz A simétrica y definida positiva y una variable booleana que indica si se desea agregar ceros debajo de la diagonal (default = True). Retorna un arreglo de *numpy*, el cual es una matriz R triangular superior que cumple que $A = R^T R$, es decir, es la matriz de la factorización de Cholesky de A.

Además, se inlcuyen dos ejemplos de uso de esta función en los cuales se encontró la matriz de Cholesky R para matrices A y se verificó la exactitud de los resultados revisando que la norma de la matriz $A - R^T R$ sea practicamente cero.

---

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.

En ejercicio6.py se comparó la complejidad de la 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.

Para esto, se tuvo que generar una matriz aleatoria simétrica y definida positiva. Una buena idea para esto, podría ser: generar una matriz aleatoria $B(0,1)$ de $n \times n$ y tomar $A = BB^T$ ya que, se sabe que esta matriz A siempre cumplirá con lo que se necesita, sin embargo, la multiplicación de matrices BB^T va a involucrar un costo computacional bastante alto a medida que n crece, así que se propone otro método:

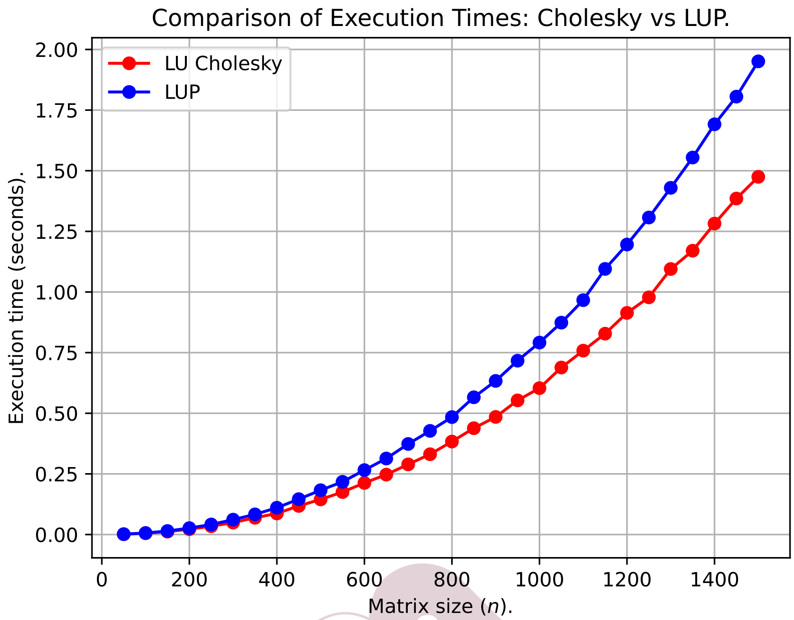
- Se genera una matriz aleatoria $C(0,1)$.
- Se toma $B = C + C^T$. Esto ya hace que B sea simétrica, pero no se está seguro de que sea definida positiva. Además, B tiene sus entradas en el intervalo $[0,2]$ ya que C las tiene en $[0,2]$.
- Se hace $A = B + nId_{n\times n}$. Esto obliga a que A sea una matriz simética (por B) y que sea definida positiva ya que, al sumar $nId_{n\times n}$, se vuelve una matriz diagonal dominante ya que una condición suficiente para que una matriz simétrica sea definida positiva es que todos los elementos diagonales sean positivos y la matriz sea diagonalmente dominante.

La generación de estas matrices A simétricas y definidas positivas está a cargo de la función *generar_matriz)()*. La cual toma como argumento a la dimensión n (un entero positivo) y devuelve la matriz aleatoria A de $n \times n$ como arreglo de *numpy*.

A continuación, se generan listas vacías para guardar el historial de tiempos para los algoritmos de Cholesky y LUP y se toman matrices de tamaño n, para n en el conjunto:

$$\{50, 100, 150, 200, \ldots, 1450, 1500\}.$$

Para cada una de estas matrices, se midió el tiempo de ejecución de los algoritmos dados por las funciones *LU_Cholesky()* y *LUP()* y se guardaron en cada iteración para finalmente graficar el historial de tiempos de ejecución según el tamaño de la matriz, generando el siguiente gráfico:



Se puede notar que a medida que n crece, el algoritmo LUP se vuelve más costoso que el de la factorización de Cholesky. A pesar de que ambos son de orden n^3, se puede notar que LUP es menos eficiente.