Cómputo científico para probabilidad y estadística. Tarea 1. Descomposición LU y Cholesky.

Juan Esaul González Rangel

Septiembre 2023

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

   En el script `LUyCholesky.py` que se entrega de tarea, las funciones están implementadas con los nombres `backsubs` y `forsubs`, respectivamente.

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

   En el script `LUyCholesky.py` que se entrega de tarea, las función está implementada con el nombre `LUP`.

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

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

   La matriz aleatoria que obtenemos mediante `np.random.random((5,5))` es

$$R = \begin{bmatrix} 0.08734964 & 0.2304771 & 0.41106107 & 0.3107827 & 0.56595589 \\ 0.5450637 & 0.80709944 & 0.91815511 & 0.52209075 & 0.42468726 \\ 0.07180395 & 0.89852885 & 0.42051383 & 0.58216979 & 0.21415354 \\ 0.44748568 & 0.4678638 & 0.10063716 & 0.92728789 & 0.16098684 \\ 0.87459418 & 0.05216006 & 0.99227957 & 0.10514732 & 0.40198985 \end{bmatrix}.$$

   Para A, las matrices L y U que obtenemos como salida de de la función `LUP(A)` son,

```
L
array([[ 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.]])

U
array([[ 1., 0., 0., 0., 1.],
       [-1., 1., 0., 0., 2.],
       [-1., -1., 1., 0., 4.],
       [-1., -1., -1., 1., 8.],
       [-1., -1., -1., -1., 16.]])
```

   La salida de la matriz P se omite porque es evidente, por la forma de A, que durante el algoritmo nunca es necesario realizar permutaciones (pues los valores en la diagonal de A son los máximos de su respectiva columna), así que $P = I_{n\times n}$.

   Notemos además, que en la salida, U no tiene explícitamente la forma de una matriz triangular superior y esto es porque la implementación del algoritmo, por eficiencia, no considera la escritura de las celdas correspondientes

a la parte inferior de U, que corresponde únicamente a ceros. Si tomamos esto en cuenta, la descomposición LU de A es,

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

Para R, tenemos que la salida de la función `LUP(R)` es

```
L_R
array[[ 1.          0.          0.          0.          0.        ]
      [ 0.08209974  1.          0.          0.          0.        ]
      [ 0.51164951  0.49334957  1.          0.          0.        ]
      [ 0.09987448  0.25190776 -0.39445671  1.          0.        ]
      [ 0.62321899  0.86619549 -0.01056158 -0.08745472  1.        ]]
U_R
array[[ 0.87459418  0.05216006  0.99227957  0.10514732  0.40198985]
      [ 0.5450637   0.89424652  0.33904793  0.57353722  0.18115027]
      [ 0.07180395  0.7745923  -0.57433135  0.59053498 -0.13406147]
      [ 0.44748568  0.44117614  0.00606584  0.38874317  0.42729275]
      [ 0.08734964  0.22526764  0.22654885 -0.03399742  0.05320087]]
 P_R
 array[[0. 0. 0. 0. 1.]
       [0. 0. 1. 0. 0.]
       [0. 0. 0. 1. 0.]
       [1. 0. 0. 0. 0.]
       [0. 1. 0. 0. 0.]]
```

Analógamente al caso anterior, podemos descartar las posiciones no sobreescritas de la matriz U_R para encontrar la descomposición:

$$\begin{bmatrix} 0. & 0. & 0. & 0. & 1. \\ 0. & 0. & 1. & 0. & 0. \\ 0. & 0. & 0. & 1. & 0. \\ 1. & 0. & 0. & 0. & 0. \\ 0. & 1. & 0. & 0. & 0. \end{bmatrix} R = \begin{bmatrix} 1. & 0. & 0. & 0. & 0. \\ 0.0820 & 1. & 0. & 0. & 0. \\ 0.5116 & 0.4933 & 1. & 0. & 0. \\ 0.0998 & 0.2519 & -0.3944 & 1. & 0. \\ 0.6232 & 0.8661 & -0.0105 & -0.0874 & 1. \end{bmatrix} \begin{bmatrix} 0.8745 & 0.0521 & 0.9922 & 0.1051 & 0.4019 \\ 0 & 0.8942 & 0.3390 & 0.5735 & 0.1811 \\ 0 & 0 & -0.5743 & 0.5905 & -0.1340 \\ 0 & 0 & 0 & 0.3887 & 0.4272 \\ 0 & 0 & 0 & 0 & 0.0532 \end{bmatrix}$$

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

$$Dx = b \tag{2}$$

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.

El primer paso es generar una matriz aleatoria de tamaño 5×5 cuyas entradas sean v. a. i. i. d. uniformes en $(0, 1)$. La j-ésima columna representa un vector de tamaño 5 que corresponde al j-ésimo vector b con el que se resuelve el sistema $Ax = b$. Con la semilla usada, la matriz aleatoria obtenida es:

```
b
array([[0.07183655 0.01919304 0.72131415 0.04146587 0.01865054]
       [0.08293774 0.28090455 0.42914171 0.01973034 0.4616679 ]
       [0.66958658 0.67304387 0.7969659  0.08372024 0.89487394]
       [0.78670726 0.66402411 0.85267895 0.54438794 0.19747141]
       [0.53870933 0.50728999 0.27735705 0.66848423 0.73135733]]),
```

que corresponde a

$$B = \begin{bmatrix} 0.07183655 & 0.01919304 & 0.72131415 & 0.04146587 & 0.01865054 \\ 0.08293774 & 0.28090455 & 0.42914171 & 0.01973034 & 0.4616679 \\ 0.66958658 & 0.67304387 & 0.7969659 & 0.08372024 & 0.89487394 \\ 0.78670726 & 0.66402411 & 0.85267895 & 0.54438794 & 0.19747141 \\ 0.53870933 & 0.50728999 & 0.27735705 & 0.66848423 & 0.73135733 \end{bmatrix}.$$

Como intentamos resolver $Ax = b$, y tomando $A = PA = LU$ (usando que $P = I_{n\times n}$), esto significa que $LUx = b$, por lo cual es posible resolver en dos pasos, primero resolvemos para y en $Ly = b$, y a continuación resolvemos para x en $Ux = y$, con lo que la solución que satisfaga esto será la que también satisface nuestro sistema original.

Con el objetivo de verificar si el programa corre sin complicaciones, se introdujo un bloque de excepcón `try-catch`, de la siguiente forma:

```
try:
    L, U, P = LUP(A,out="copy")
    L_R, U_R, P_R = LUP(R, out="copy")
except:
    print("Program didn't finish! Exception ocurred!")
```

Lo que conseguimos es que el programa intente encontrar las factorizaciones usando la función `LUP()`, y si se encuentra alguna excepción (error durante la ejecución), termina e imprime un mensaje de error en la pantalla. Si se termina la ejecución sin dicho error, significa que el sistema se puede resolver.

Después de verificar que el sistema se puede resolver, se usaron las funciones `backsubs` y `forsubs` con cada uno de los vectores b (columnas de B) para encontrar,

$$X = \begin{bmatrix} -0.15135302 & -0.21796723 & 0.08312366 & -0.07046919 & -0.27600275 \\ -0.29160485 & -0.17422295 & -0.12592512 & -0.16267391 & -0.10898813 \\ 0.00343914 & 0.04369341 & 0.11597395 & -0.26135792 & 0.21522979 \\ 0.12399896 & 0.07836706 & 0.28766095 & -0.06204815 & -0.26694296 \\ 0.22318957 & 0.23716027 & 0.63819049 & 0.11193506 & 0.29465328 \end{bmatrix},$$

donde esta matriz tiene una interpretación análoga a B, es decir, cada columna x de X representa un vector solución al sistema $Ax = b$, con B la columna correspondiente de B.

Para resolver los sistemas $Rx = b$, es necesario que tomemos en cuenta que en este caso la factorización nos indica que $PRx = LUx = Pb$, por lo tanto es necesario primero encontrar $z = Pb$ de forma que se satisfaga $LUx = z$. Una vez que se ha hecho esto, se encuentra y tal que $Ly = z$ y finalmente x tal que $Ux = y$. De manera análoga al caso $Ax = b$, guardamos la información de los 5 pares de vectores x y b en las matrices X_R y B_R para encontrar que la solución al sistema está dada por,

$$X_R = \begin{bmatrix} -16.2177413 & -11.74925034 & -8.83024329 & -8.07083815 & -15.49777936 \\ -14.29089476 & -10.21830868 & -7.40322026 & -7.92951685 & -12.83212444 \\ 19.93182075 & 14.6723009 & 10.2113118 & 10.45107116 & 19.36772763 \\ 16.32463072 & 11.88974378 & 8.9692724 & 8.70675197 & 14.56700148 \\ -14.99133424 & -11.17713672 & -6.68967675 & -7.82377868 & -14.41561819 \end{bmatrix}.$$

Una vez más, cada columna corresponde a la solución de $Rx = b$, para b la columna correspondiente en la matriz B.

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

   En el script `LUyCholesky.py` que se entrega de tarea, las función está implementada con el nombre `cholesky`.

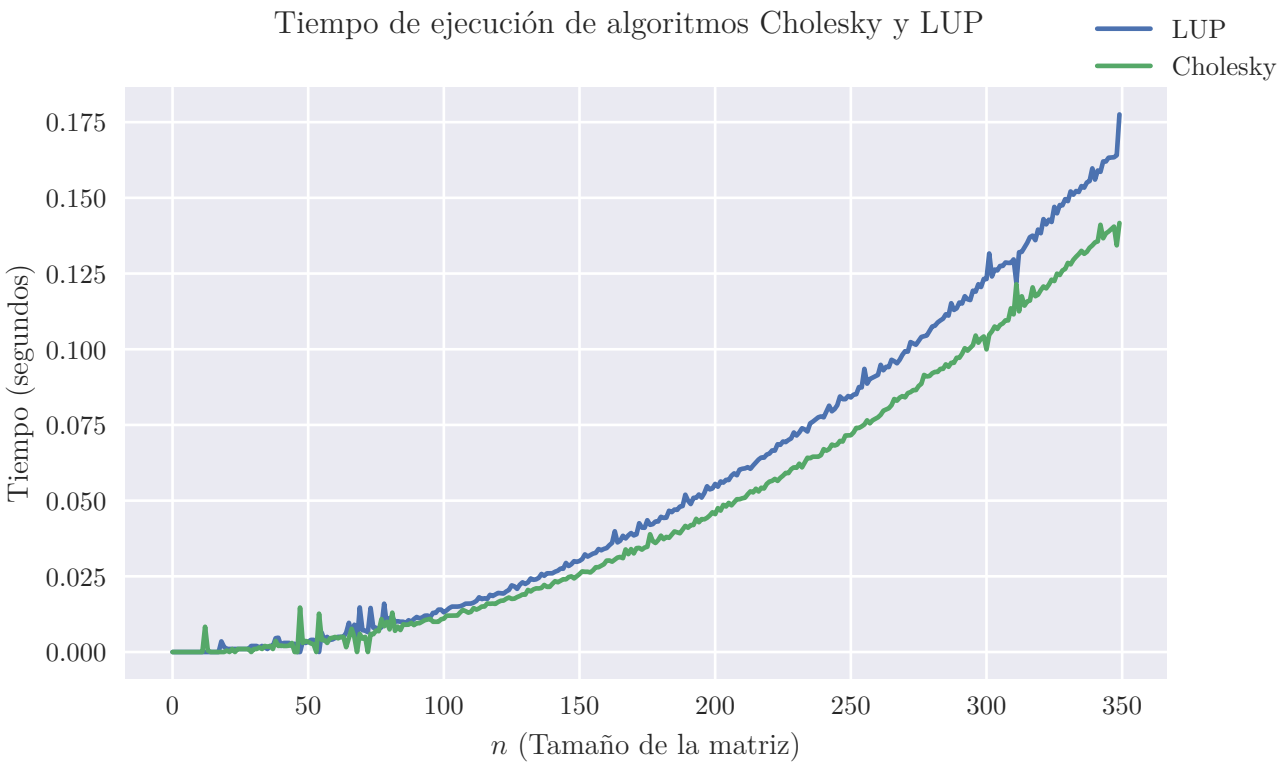
6. Comparar la complejidad de su implementación de los algoritmos de factorización de Cholesky y LUP mediante la medición de los tiempos que tardan con respecto a la descomposición de una matriz aleatoria hermitiana definida positiva. Graficar la comparación.

Haciendo uso del módulo `time`, podemos crear una función que nos indique el tiempo de inicio y final de ejecución de una determinada porción de código. Al repetir este procedimiento para ambos algoritos y distintos tamaños de matrices, encontramos un vector de valores en el que podemos observar como es que el tiempo de ejecución del algoritmo incrementa al variar el tamaño de la entrada.

Una comparación muy sencilla es contar la cantidad de matrices para las cuales el algoritmo LU tardó más tiempo que el algoritmo Cholesky. Con los valores obtenidos en esta ejecución en particular, tenemos que esto ocurrió un total de 310 veces.

Otra manera útil de comparar ls tiempos de ejecución es mediante el uso de una gráfica.

En la figura a continuación observamos que, para n pequeño, el tiempo de ejecución de ambos algoritmos es similar, por lo que no se puede decir que alguna de las funciones esté dominada por la otra, pero, a medida que el tamaño de la entrada crece, el tiempo que tarda en ejecutarse el algoritmo LUP es cada vez mayor comparado con el tiempo que tarda en ejecutarse el algoritmo Cholesky, y las curvas que tienden a formar para tamaños de matriz grandes son muy similares al resultado teórico de complejidad $\sim 2m^3/3$ para LU y $\sim m^3/3$ para Cholesky.



También notamos en la gráfica que el comportamiento del tiempo presenta varias irregularidades y esto se debe a que los datos son tiempos de ejecución en lugar de cantidad de operaciones y estos tiempos pueden haber sido mayores en algún paso debido a una tarea en segundo plano que realiza el procesador.

En conclusión, podemos decir que en el timepo de ejecución del algoritmo intervienen varios factores, por lo que estrictamente hablando, el que un algoritmo tenga mayor complejidad que otro, no implica que siempre tardará más, pero a medida que incrementa el tamaño de la entrada, el comportamiento del algoritmo se aproxima al de su término domimante, por lo que en tamaños grandes, una menor complejidad sí supone una ventaja.