gauss-lu

March 16, 2019

1 Ejercicio clase 13 de Marzo, 2019_

Equipo 9

```
In [1]: import numpy as np
        import scipy
        import scipy.linalg
```

1.1 Eliminación Gaussiana Simple

```
In [2]: A = np.array([
            [1,2,1,1],
            [2,2,3,1],
            [-1,-3,1,0],
            [-1,-3,0,1]
        ])
        B = np.array([0,3,2, 4])
        print("A = \n", A, "\n B =", B)
```

```
A =
 [[ 1  2  1  1]
 [ 2  2  3  1]
 [-1 -3  1  0]
 [-1 -3  0  1]]
 B = [0 3 2 4]
```

Resolución por Scipy

```
In [3]: A_ = scipy.linalg.lu_factor(A)
        A_
```

```
Out[3]: (array([[ 2.  ,  2.  ,  3.  ,  1.  ],
                [-0.5 , -2.  ,  2.5 ,  0.5 ],
                [-0.5 ,  1.  , -1.  ,  1.  ],
                [ 0.5 , -0.5 , -0.75,  1.5 ]]), array([1, 2, 3, 3], dtype=int32))
```

```
In [4]: scipy.linalg.lu_solve(A_, B)
```

```
Out[4]: array([ 4.66666667, -2.5       , -0.83333333,  1.16666667])
```

1.1.1 Factorización por Algoritmo

```
In [5]: A

Out[5]: array([[ 1,  2,  1,  1],
               [ 2,  2,  3,  1],
               [-1, -3,  1,  0],
               [-1, -3,  0,  1]])

In [6]: def factor_lu(A):
            N = A.shape[0]
            if N != A.shape[1]:
                raise Exception("Error: no es una matriz cuadrada")

            A_ = A.copy()
            A_ = A.astype(np.double)
            for k in range(0, N-1):
                A_[k+1:, k] = A_[k+1:, k] / A_[k, k]
                A_[k+1:,k+1:] = A_[k+1:,k+1:] - np.outer(A_[k+1:, k],A_[k, k+1:])
            return A_
        A_ = factor_lu(A)
        print("LU =\n", A_)

LU =
 [[ 1.          2.          1.          1.        ]
 [ 2.         -2.          1.         -1.        ]
 [-1.          0.5         1.5         1.5       ]
 [-1.          0.5         0.33333333  2.        ]]


In [7]: print("U =\n", np.triu(A_))

U =
 [[ 1.   2.   1.   1. ]
 [ 0.  -2.   1.  -1. ]
 [ 0.   0.   1.5  1.5]
 [ 0.   0.   0.   2. ]]


In [8]: print("L =\n", np.tril(A_, -1))

L =
 [[ 0.          0.          0.          0.        ]
 [ 2.          0.          0.          0.        ]
 [-1.          0.5         0.          0.        ]
 [-1.          0.5         0.33333333  0.        ]]
```

1.2 Resolución LU

```
In [9]: np.matrix([[1,2],[2,2]], ).shape

Out[9]: (2, 2)

In [10]: def solve_LU(A, B):
             N = A.shape[0]
             if N != A.shape[1]:
                 raise Exception("Error: no es una matriz cuadrada")
             if len(B.shape) == 1:
                 M = 1
             else:
                 M = B.shape[1]
             B_ = B.copy()
             L_ = np.tril(A, -1)
             U_ = np.triu(A)
             #Sustitución hacia adelante LD=B
             D_ = np.zeros((N,M), dtype=np.double)
             D_[0,] = B_[0]
             for i in range(1, N):
                 D_[i,] = B_[i,] - np.dot(A[i, 0:i], D_[0:i, :])
             #Sustitución hacia adelante UX=D
             X_ = np.zeros((N,M))
             X_[-1, :] = D_[-1,:]/U_[-1,-1]
             for i in range(N-2, -1, -1):
                 X_[i,] = (D_[i,] - U_[i,(i+1):N]@X_[(i+1):N,]) / U_[i,i]
             #Pivote (permutación)
             P_ = np.eye(N)
             return X_

         solve_LU(A_, B)

Out[10]: array([[ 4.66666667],
                [-2.5       ],
                [-0.83333333],
                [ 1.16666667]])
```

1.3 Cálculo de Tiempos

Para comprobar las velocidades utilizamos una matríz A de 1000x1000; B con un vector de 1000. Utilizamos la función `timeit` que saca promedios, con el fin de minimizar la variación de resultados por otros procesos (Sistema Operativo, otras aplicaciones, etc).

```
In [11]: np.random.seed(175904)
         A = np.random.randint(-5, 5, (1000, 1000))
         B = np.random.randint(-5, 5, (1000, 1))
         print("Tamaño de A:", A.shape,
               "Tamaño de B:", B.shape)
```

```
Tamaño de A: (1000, 1000) Tamaño de B: (1000, 1)
```

1.3.1 Usando Numpy

```
In [12]: %%timeit -n20
         np.linalg.solve(A, B)
         print
```

```
12.2 ms ± 1.22 ms per loop (mean ± std. dev. of 7 runs, 20 loops each)
```

1.3.2 LU - Scipy

Factorización

```
In [13]: %%timeit -n20
         L, U = scipy.linalg.lu_factor(A)
```

```
8.47 ms ± 143 µs per loop (mean ± std. dev. of 7 runs, 20 loops each)
```

Factorización y Resolución

```
In [14]: %%timeit -n20
         L, U = scipy.linalg.lu_factor(A)
         scipy.linalg.lu_solve((L, U), B)
         print
```

```
10.3 ms ± 153 µs per loop (mean ± std. dev. of 7 runs, 20 loops each)
```

1.3.3 LU - Nuestros Algoritmos

Factorización

```
In [15]: %%timeit -n10
         A_ = factor_lu(A)
```

```
1.35 s ± 64.3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Factorización y Resolución

```
In [16]: %%timeit -n10
         solve_LU(A_, B)
```

```
103 µs ± 30.8 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

1.4 Conclusiones

Nuestras funciones no son tan rápidas y ni optimizadas como las construidas por Scipy/Numpy e Intel, dado que usamos `Intel Distribution for Python`. Sin embargo, entregan llegan a los mismos resultados.