1 Matrici e norme

Exercise 1.1. Si consideri la matrice A

$$A = \begin{pmatrix} 1 & 2 \\ 0.499 & 1.001 \end{pmatrix}$$

Calcolare la norma 1, la norma 2, la norma Frobenius e la norma infinito di A con `numpy.linalg.norm()`:

```
norma_uno = np.linalg.norm(A, ord=1)
print ("Norma uno:", norma_uno)
norma_due = np.linalg.norm(A, ord=2)
print ("Norma due:", norma_due)
norma_frob = np.linalg.norm(A, ord='fro')
print ("Norma di Frobenius:", norma_frob)
norma_inf = np.linalg.norm(A, ord=np.inf)
print ("Norma infinito:", norma_inf)
```

Output:

Norma uno: 3.001
Norma due: 2.500200104037774
Norma di Frobenius: 2.5002003919686118
Norma infinito: 3.0

Calcolare il numero di condizionamento di A con `numpy.linalg.cond()`:

```
cond1 = np.linalg.cond(A, 1)
print ('\nK(A)_1 = ', cond1)
cond2 = np.linalg.cond(A, 2)
print ('K(A)_2 = ', cond2)
condfro = np.linalg.cond(A, 'fro')
print ('K(A)_fro =', condfro)
condinf = np.linalg.cond(A, np.inf)
print ('K(A)_inf =', condinf, '\n')
```

Output:

K(A)_1 = 3001.0000000001082
K(A)_2 = 2083.6668534103555
K(A)_fro = 2083.6673333334084
K(A)_inf = 3001.0000000001082

Considerare il vettore colonna x = $(1, 1)^T$ e calcolare il corrispondente termine noto b per il sistema lineare Ax = b

```
x = np.ones((2,1))
b = A@x
print('Ax =', b)
```

Output:

Ax = [[3.] [1.5]]

Considerare ora il vettore my_btilde = $(3, 1.4985)^T$ e verifica che xtilde = $(2, 0.5)^T$ è soluzione del sistema A xtilde = my_btilde

```
btilde = np.array([[3], [1.4985]])
xtilde = np.array([[2, 0.5]]).T
my_btilde = A@xtilde
print ('A*xtilde =\n', btilde, '\n')
```

Output:

A*xtilde = [[3.   ] [1.4985]]

Calcolare la norma 2 della perturbazione sui termini noti $\Delta_b = \|b - my\_btilde\|_2$ e la norma 2 della perturbazione sulle soluzioni $\Delta_x = \|x - xtilde\|_2$. Confrontare Δ_b con Δ_x.

```
deltax = np.linalg.norm(x-xtilde, ord=2)
deltab = np.linalg.norm(b-btilde, ord=2)
print ('delta x = ', deltax)
print ('delta b = ', deltab)
```

Output:

```
delta x =  1.1180339887498895
delta b =  0.0015000000000000568
```

2 Metodi diretti

Exercise 2.1. Si consideri la matrice

$$A = \begin{pmatrix} 3 & -1 & 1 & -2 \\ 0 & 2 & 5 & -1 \\ 1 & 0 & -7 & 1 \\ 0 & 2 & 1 & 1 \end{pmatrix}$$

Creare il problema test in cui il vettore della soluzione esatta è x = (1, 1, 1, 1)^T e il vettore termine noto è b = Ax

```
A = np.array ([ [3,-1, 1,-2], [0, 2, 5, -1], [1, 0, -7, 1], [0, 2, 1, 1]  ])
x = np.ones((4,1))
b = A@x
condA = np.linalg.norm(A, 2)
print('x: \n', x , '\n')
print('x.shape: ', x.shape, '\n' )
print('b: \n', b , '\n')
print('b.shape: ', b.shape, '\n' )
print('A: \n', A, '\n')
print('A.shape: ', A.shape, '\n' )
print('K(A)=', condA, '\n')
```

Output:

```
x:
[[1.]
[1.]
[1.]
[1.]]
x.shape:  (4, 1)

b:
 [[ 1.]
 [ 6.]
 [-5.]
 [ 4.]]
b.shape:  (4, 1)

A:
[[ 3 -1  1 -2]
[ 0  2  5 -1]
 [ 1  0 -7  1]
[ 0  2  1  1]]
A.shape:  (4, 4)
K(A)= 8.949375864009538
```

Guardare l'help del modulo `scipy.linalg.decomp_lu` e usare una delle sue funzioni per calcolare la fattorizzazione LU di A con pivoting. Verificare la correttezza dell'output

```
lu, piv = LUdec(A)
print('lu\n',lu,'\n')
print('piv',piv,'\n')
```

Output:

```
lu
[[ 3.         -1.          1.         -2.        ]
 [ 0.          2.          5.         -1.        ]
 [ 0.33333333  0.16666667 -8.16666667  1.83333333]
 [ 0.          1.          0.48979592  1.10204082]]

piv [0 1 2 3]
```

Risolvere il sistema lineare con la funzione `lu_solve` del modulo decomp lu oppure con la funzione `scipy.linalg.solve_triangular;` Stampare la soluzione calcolata e valutarne la correttezza.

Tramite lu_solve:

```
my_x= scipy.linalg.lu_solve((lu, piv), b)
print('my_x = \n', my_x)
print('norm =', scipy.linalg.norm(x-my_x, 'fro'))
```

Output:

```
my_x =
 [[1.]
 [1.]
 [1.]
 [1.]]
norm = 3.1401849173675503e-16
```

Soluzione alternativa con LUfull e solve_triangular:

```
# IMPLEMENTAZIONE ALTERNATIVA - 1
P, L, U = LUfull(A)
print ('A = ', A)
print ('P = ', P)
print ('L = ', L)
print ('U = ', U)
print ('P*L*U = ', np.matmul(P , np.matmul(L, U)))

print ('diff = ',   np.linalg.norm(A - np.matmul(P , np.matmul(L, U)), 'fro'  ) )

invP = np.linalg.inv(P)
y = scipy.linalg.solve_triangular(np.matmul(L,invP), b, lower=True, unit_diagonal=True)
my_x = scipy.linalg.solve_triangular(U, y, lower=False)

print('\nSoluzione calcolata: ', my_x)
print('norm =', scipy.linalg.norm(x-my_x, 'fro'))
```

Output:

```
A =     [[ 3 -1  1 -2]
         [ 0  2  5 -1]
         [ 1  0 -7  1]
         [ 0  2  1  1]]

P =     [[1. 0. 0. 0.]
         [0. 1. 0. 0.]
         [0. 0. 1. 0.]
         [0. 0. 0. 1.]]

L =     [[1.         0.         0.         0.        ]
         [0.         1.         0.         0.        ]
         [0.33333333 0.16666667 1.         0.        ]
         [0.         1.         0.48979592 1.        ]]

U =     [[ 3.         -1.          1.         -2.        ]
         [ 0.          2.          5.         -1.        ]
         [ 0.          0.         -8.16666667  1.83333333]
         [ 0.          0.          0.          1.10204082]]

P*L*U =         [[ 3. -1.  1. -2.]
                 [ 0.  2.  5. -1.]
                 [ 1.  0. -7.  1.]
                 [ 0.  2.  1.  1.]]

diff =  1.5700924586837752e-16

Soluzione calcolata:  [[1.]
                       [1.]
                       [1.]
                       [1.]]
norm = 5.438959822042073e-16
```

La soluzione calcolata con entrambi i metodi coincide con la soluzione esatta calcolata all'inizio.

Dato un problema test di dimensioni variabili

Sia Ax = b un problema test di dimensioni variabili (in questo caso A avrà dimensioni da 2x2 a 10x10) la cui soluzione esatta è il vettore x_true= $(1, \ldots, 1)^T$ e b è il termine noto, mostro a video il numero di condizionamento della matrice A e la soluzione x del medesimo sistema lineare mediante la fattorizzazione LU con pivoting.

Nota: A è quadrata, e i suoi valori sono numeri casuali (compresi tra 10 e 1000) generati mediante la funzione rand di Numpy.

```
# Genera un numero casuale 'n' compreso tra 2 e 10
n = np.random.randint(2, 11)

# Matrice quadrata di dimensione 'n' con valori casuali compresi tra 10 e 1000
A = np.random.rand(n, n) * 990 + 10
x = np.ones((n,1))

print(f"A (size {n}x{n}):\n{A})")
print('K(A)=', np.linalg.norm(A, 2), '\n')

b = A@x

print('x: \n', x , '\n')
print('b: \n', b , '\n')

lu, piv = LUdec(A)
print('lu\n',lu,'\n')
print('piv',piv,'\n')

my_x= scipy.linalg.lu_solve((lu, piv), b)
print('my_x = \n', my_x)
print('norm =', scipy.linalg.norm(x-my_x, 'fro'))
```

Output:

A (size 5x5):

[[533.30619347 975.53999967 934.50176627 238.84678523 633.30799193]

[886.11518488 708.52032906 883.31348734 429.21007065 423.95564996]

[362.8379107 255.64375891 450.34947423 235.23326649 525.86647286]

[907.47648693 538.91290361 122.46107179 968.61743436 347.17123172]

[193.06785106 106.58149788 659.52659364 552.18804642 663.70295958]])

K(A)= 2790.0509823744064

x:

[[1.]

[1.]

[1.]

[1.]

[1.]]

b:

[[3315.50273658]

[3331.11472189]

[1829.9308832]

[2884.63912841]

[2175.06694857]]

lu

[[9.07476487e+02 5.38912904e+02 1.22461072e+02 9.68617434e+02
  3.47171232e+02]

[5.87680454e-01 6.58831420e+02 8.62533788e+02 -3.30390748e+02
  4.29282245e+02]

[2.12752456e-01 -1.22543423e-02 6.44042484e+02 3.42063587e+02
  5.95101999e+02]

[9.76460765e-01 2.76691453e-01 8.15286788e-01 -7.04070478e+02
 -5.19000962e+02]

[3.99831749e-01 6.09704835e-02 5.41573819e-01 4.50465324e-01
  2.72383120e+02]]

piv [3 3 4 3 4]

my_x =

[[1.]

[1.]

[1.]

[1.]

[1.]]

norm = 1.2609709600486848e-15

Altro Output:

A (size 2x2):

[[895.31337164 605.9904596]

 [791.84903555 78.27056328]])

K(A)= 1305.1372988991388

x:

 [[1.]

 [1.]]

b:

 [[1501.30383124]

 [870.11959883]]

lu

 [[8.95313372e+02 6.05990460e+02]

 [8.84437852e-01 -4.57690337e+02]]

piv [0 1]

my_x =

 [[1.]

 [1.]]

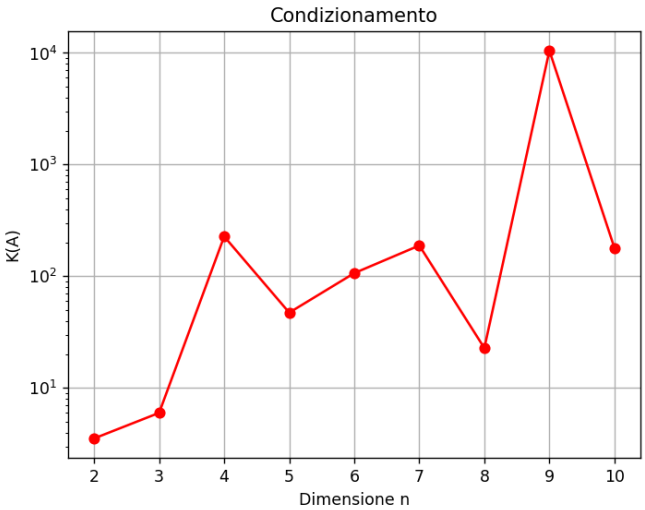
norm = 2.220446049250313e-16

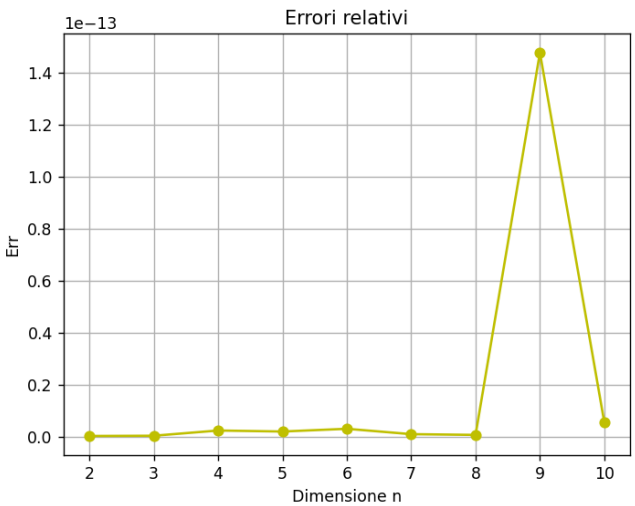
Generalizzando il codice dell’esercizio 2.2 (Matrici di Hilbert):

```
for n in range:
    A = np.random.rand(n, n) * 990 + 10
    cond_numbers.append(np.linalg.cond(A))
    x = np.ones((n, 1))
    b=A@x

    lu, piv = LUdec(A)
    print('lu\n',lu,'\n')
    print('piv',piv,'\n')

    my_x= scipy.linalg.lu_solve((lu, piv), b)
    errors.append(norm(x-my_x)/norm(x))
```





Sia Ax = b un problema test di dimensioni nxn variabili (in questo caso A avrà dimensioni da 2x2 a 10x10) la cui soluzione esatta è il vettore x_true= $(1, \ldots, 1)^T$ e b è il termine noto, mostro a video il numero di condizionamento della matrice A e la soluzione x mediante la fattorizzazione di Cholesky.

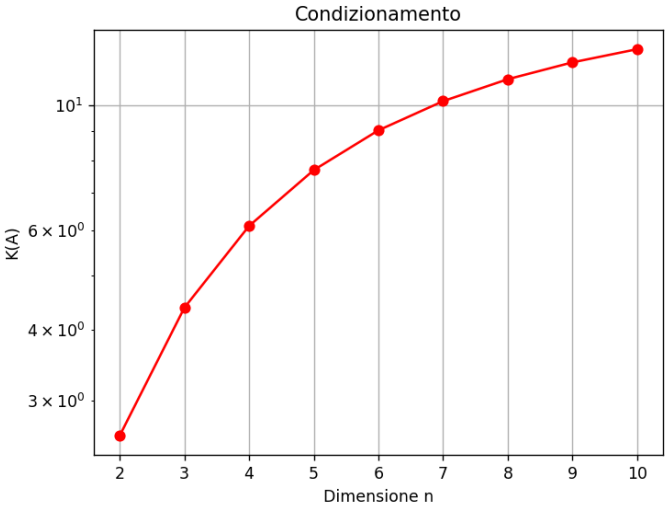
A è la matrice tridiagonale simmetrica definita positiva, avente sulla diagonale principale elementi uguali a 9 ed elementi nella sopra e sotto-diagonale uguali a -4, con n variabile. Per definirla uso la seguente funzione:

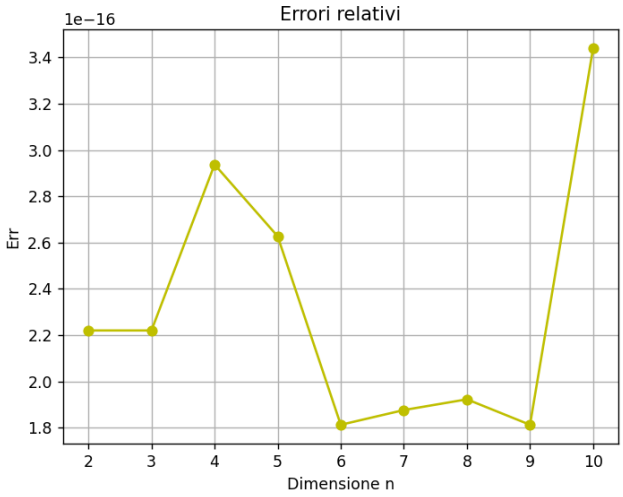
```
def init_A(n):
    diagonale_principale = 9 * np.ones(n)
    sottodiagonale = -4 * np.ones(n - 1)

    matrice_tridiagonale = np.diag(diagonale_principale) + np.diag(sottodiagonale,
    k=1) + np.diag(sottodiagonale, k=-1)

    return matrice_tridiagonale
```

Per lo svolgimento dell'esercizio sfrutto il codice dell'esercitazione 2.2 sostituendo la matrice di Hilbert con quella definita tramite la funzione sopra descritta





Exercise 2.2. Si ripeta l'esercizio precedente sulla matrice di Hilbert, che si può generare con la funzione `A = scipy.linalg.hilbert(n)` per n = 5, . . . , 10. In particolare:

Calcolare il numero di condizionamento di A e rappresentarlo in un grafico al variare di n.
Considerare il vettore colonna x = (1, . . . , 1)^T , calcola il corrispondente termine noto b per il sistema lineare Ax = b e la relativa soluzione $\tilde{x}$ usando la fattorizzazione di Cholesky come nel caso precedente.

Si rappresenti l'errore relativo al variare delle dimensioni della matrice.

```
START = 5
END = 11
cond_numbers = []
relative_errors = []
range = range(START, END)

for n in range:
    A = hilbert(n)
    cond_numbers.append(np.linalg.cond(A))
    x = np.ones((n, 1))
    L = cholesky(A, lower=True)
    b = np.dot(A, x)
    y = solve(L, b)
    x̃ = solve(L.T, y)
    relative_errors.append(norm(x-x̃)/norm(x))

# Rappresenta i numeri di condizionamento in un grafico al variare della dimensione
plt.semilogy(np.arange(START, END), cond_numbers, '-ro', marker='o', linestyle='-')
plt.xlabel('Dimensione n')
plt.ylabel('K(A)')
plt.title('Condizionamento delle matrici di Hilbert')
plt.grid(True)
plt.show()

# Rappresenta gli errori relativi in un grafico al variare della dimensione
plt.plot(np.arange(START, END), relative_errors, '-yo',  marker='o', linestyle='-')
plt.xlabel('Dimensione n')
plt.ylabel('Err')
plt.title('Errori relativi')
plt.grid(True)
plt.show()
```

Output: