cours - lineaire 1

September 8, 2023

1 Résolution de systèmes linéaires par routines et méthodes directes

Objectifs :

- Utilisation des routines de numpy pour résoudre des systèmes linéaires
- Méthode du pivot
- Décomposition LU
- Décomposition de Cholesky
- Complément : décomposition PLU

2 Les routines proposées par numpy

2.1 Inversion de matrices et résolution de systèmes linéaires

Pour résoudre

$$Ax = b$$

lorsque A est inversible, on peut écrire $x = A^{-1}b$ et faire appel à la fonction `linalg.inv` de numpy. L'exemple ci-dessous résout de cette manière l'équation

$$\begin{pmatrix} 10 & 1 & 2 \\ 3 & 6 & 1 \\ 0 & 2 & 7 \end{pmatrix} x_0 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \tag{1}$$

[54]:
```
import numpy as np
A0=np.array([[10,1,2],[3,6,1],[0,2,7]])
b0=np.array([[1],[2],[3]])
x0=np.linalg.inv(A0)@b0    # calcule A0^{-1} puis l'applique a b0
x0
```

[54]:
```
array([[0.00255754],
       [0.27365729],
       [0.35038363]])
```

Pour résoudre le système linéaire $Ax = b$, inverser la matrice A puis calculer $A^{-1}b$ peut être trop couteux en temps. Sans parler du cas où A n'est pas inversible, mais que le système $Ax = b$ admet

tout de même des solutions ! Nous verrons des méthodes plus efficaces et plus rapides. Une routine rapide directement intégrée à numpy pour une résolution lorsque A est inversible (mais sans calculer A^{-1}) est donnée par `np.linalg.solve(A,b)` qui renvoie la solution du système $Ax = b$. L'exemple précédent (1) peut être alors résolu par :

[55]:
```
x1=np.linalg.solve(A0,b0) #calcule la solution x4 de A4x4=b4 sans inverser A4
x1
```

[55]:
```
array([[0.00255754],
       [0.27365729],
       [0.35038363]])
```

La routine `np.linalg.solve(A,b)` utilise la méthode de la décomposition PLU décrite en complément plus loin dans ce cours.

2.2 Exercices

Exercice 1. Résoudre

$$\begin{cases} 17x + by - 15z + 3t &= 5, \\ ax - 12y + 8z + t &= -3, \\ 4x + 11y + cz - 5t &= 4, \\ x + 2y + 4z + 2t &= 1 \end{cases}$$

où a est votre jour de naissance, b votre mois de naissance, et c le chiffre des unités de votre année de naissance.

Exercice 2. 1. Utilisez `numpy.linalg.eig` (aidez-vous de `help(np.linalg.eig)` pour comprendre cette fonction et comment l'utiliser) pour calculer les valeurs propres λ_0 et λ_1 de la matrice

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

et les vecteurs colonnes v_0 et v_1 associés.

2. Utilisez `np.linalg.eig` pour diagonaliser la matrice A, c'est-à-dire pour calculer une matrice de changement de base P et une matrice diagonale D telle que $A = PDP^{-1}$.

3 Résolution de systèmes linéaires par la méthode du pivot

3.1 Le cas particulier des systèmes triangulaires

Pour résoudre un système triangulaire inférieur

$$\begin{cases} l_{0,0}x_0 &= b_0 \\ l_{1,0}x_0 + l_{1,1}x_1 &= b_1 \\ \dots &= . \\ l_{n-1,0}x_0 + l_{n-1,1}x_1 + \dots + l_{n-1,n-1}x_{n-1,n-1} &= b_{n-1} \end{cases}$$

lorsque les coefficients diagonaux sont non nuls, il suffit de calculer successivement x_0, puis x_1, etc. jusqu'à x_{n-1}. En effet, la première ligne du système nous donne $x_0 = \frac{b_0}{l_{0,0}}$. Une fois cette

valeur de x_0 connue, on peut alors déterminer la valeur de x_1 à l'aide de la deuxième ligne, car $x_1 = \frac{b_1 - l_{1,0}x_0}{l_{1,1}}$. Au début de l'étape k, x_0, x_1, … , x_{k-1} sont connus, et on détermine la valeur de x_k par la k-ième ligne du système qui donne

$$x_k = \frac{b_k - l_{k,0}x_0 - ... - l_{k,k-1}x_{k-1}}{l_{k,k}}.$$

La fonction `sol_tri_inf(L,b)` prend en entrée une matrice triangulaire inférieure $L = (l_{i,j})_{0 \leq i,j \leq n-1}$ et un vecteur $b = (b_0, ..., b_{n-1})$ et retourne la solution de $Lx = b$. Le code utilise la fonction `np.sum(v)` qui renvoie la somme des éléments $v_0 + v_1 + ... + v_{m-1}$ d'un vecteur v de taille m.

[89]:
```
def sol_tri_inf(L,b):
    n=len(b)                        # n est la dimension du vecteur b
    x=np.zeros(n)                   # on crée un vecteur x de longueur n
    for k in range(n):               # on calcule iterativement les valeurs x_k
 ↪pour k=0, k=1, .... jusqu'a k=n-1
        x[k]=(b[k]-np.sum(L[k,0:k]*x[0:k]))/L[k,k]     # L[k,0:k]*x[0:k] est le
 ↪vecteur (l_{k,0}x_0,...,l_{k,k-1}x_{k-1})
    return x
```

On le teste ci-dessous pour résoudre

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}$$

[98]:
```
L3=np.array([[1,0,0],[2,1,0],[1,2,3]])
b3=np.array([3,2,1])
x3=sol_tri_inf(L3,b3)
x3
```

[98]:
```
array([ 3., -4.,  2.])
```

Pour résoudre un système triangulaire supérieur

$$\begin{cases} u_{0,0}x_0 + u_{0,1}x_1 + ... + u_{0,n-1}x_{n-1} & = b_0 \\ u_{1,1}x_1 + ... + u_{1,n-1}x_{n-1} & = b_1 \\ ... & = . \\ u_{n-1,n-1}x_{n-1,n-1} & = b_{n-1} \end{cases}$$

lorsque les coefficients diagonaux $u_{0,0}$, …., $u_{n-1,n-1}$ sont non nuls, on calcule en premier x_{n-1}, puis x_{n-2}, … , jusqu'à x_0, de manière similaire. C'est ce qu'implémente la fonction `sol_tri_sup(U,b)` ci-dessous.

[109]:
```
def sol_tri_sup(U,b):
    n=len(b)
    x=np.zeros(n)
    for k in range(n):
```

```
        x[n-1-k]=(b[n-1-k]-np.sum(U[n-1-k,n-k:n]*x[n-k:n]))/U[n-1-k,n-1-k]
    return x
```

3.2 La méthode du Pivot

Pour résoudre un système

$$(2) \qquad Ax = b \quad \Leftrightarrow \quad \begin{cases} a_{0,0}x_0 & +a_{0,1}x_1 & +... & +a_{0,n-1}x_{n-1} & = b_0, \\ a_{1,0}x_0 & +a_{1,1}x_1 & +... & +a_{1,n-1}x_{n-1} & = b_1, \\ ... & & & & \\ a_{n-1,0}x_0 & +a_{n-1,1}x_1 & +... & +a_{n-1,n-1}x_{n-1} & = b_{n-1}. \end{cases}$$

on peut essayer de le transformer en un système triangulaire. Si $a_{0,0}$ est non nul, en ajoutant $-\frac{a_{1,0}}{a_{0,0}}$ fois la première ligne à la seconde, on obtient le système

$$\begin{cases} a_{0,0}x_0 & +a_{0,1}x_1 & +... & +a_{0,n-1}x_{n-1} & = b_0, \\ & (a_{1,1} - \frac{a_{1,0}}{a_{0,0}}a_{0,1})x_1 & +... & +(a_{1,n-1} - \frac{a_{1,0}}{a_{0,0}}a_{0,n-1})x_{n-1} & = b_1 - \frac{a_{1,0}}{a_{0,0}}b_0, \\ ... & & & & \\ a_{n-1,0}x_0 & +a_{n-1,1}x_1 & +... & +a_{n-1,n-1}x_{n-1} & = b_{n-1}. \end{cases}$$

où la présence de x_0 a disparu dans la deuxième ligne. De même, en ajoutant $-\frac{a_{2,0}}{a_{0,0}}$ fois la première ligne à la troisième, on efface la présence de x_0 dans la troisième ligne. En faisant cela pour toutes les lignes suivantes, on a transformé le système (2) en le système équivalent

$$\begin{cases} a_{0,0}x_0 & +a_{0,1}x_1 & +... & +a_{0,n-1}x_{n-1} & = b_0, \\ & (a_{1,1} - \frac{a_{1,0}}{a_{0,0}}a_{0,1})x_1 & +... & +(a_{1,n-1} - \frac{a_{1,0}}{a_{0,0}}a_{0,n-1})x_{n-1} & = b_1 - \frac{a_{1,0}}{a_{0,0}}b_0, \\ & ... & & & \\ & (a_{n-1,1} - \frac{a_{n-1,0}}{a_{0,0}}a_{0,1})x_1 & +... & +(a_{n-1,n-1} - \frac{a_{n-1,0}}{a_{0,0}}a_{0,n-1})x_{n-1} & = b_{n-1} - \frac{a_{n-1,0}}{a_{0,0}}b_0, \end{cases}$$

où x_0 n'apparaît plus que dans la première ligne. On a donc transformé (2) en

$$(3) \qquad A'x = b'$$

avec

$$A' = \begin{pmatrix} a_{0,0} & a_{0,1} & . & a_{0,n-1} \\ 0 & a_{1,1} - \frac{a_{1,0}}{a_{0,0}}a_{0,1} & . & a_{1,n-1} - \frac{a_{1,0}}{a_{0,0}}a_{0,n-1} \\ . & . & . & . \\ 0 & a_{n-1,1} - \frac{a_{n-1,0}}{a_{0,0}}a_{0,1} & . & a_{n-1,n-1} - \frac{a_{n-1,0}}{a_{0,0}}a_{0,n-1} \end{pmatrix} \quad \text{et} \quad b' = \begin{pmatrix} b_0 \\ b_1 - \frac{a_{1,0}}{a_{0,0}}b_0 \\ ... \\ b_{n-1} - \frac{a_{n-1,0}}{a_{0,0}}b_0 \end{pmatrix}.$$

On peut alors faire des opérations similaires pour éliminer la présence de x_1 dans les lignes 2 à n du système, ce qui revient à faire apparaître des 0 dans la deuxième colonne sous la diagonale de la matrice. Le système (3) est alors transformé en

$$A''x = b''$$

avec

$$(4) \qquad A'' = \begin{pmatrix} a'_{0,0} & a'_{0,1} & a'_{0,2} & . & a'_{0,n-1} \\ 0 & a'_{1,1} & a'_{1,2} & . & a'_{1,n-1} \\ 0 & 0 & a'_{2,2} - \frac{a'_{2,1}}{a'_{1,1}}a'_{1,2} & . & a'_{2,n-1} - \frac{a'_{2,1}}{a'_{1,1}}a'_{1,n-1} \\ . & . & . & . & . \\ 0 & 0 & a'_{n-1,2} - \frac{a'_{n-1,1}}{a'_{1,1}}a'_{1,2} & . & a'_{n-1,n-1} - \frac{a'_{n-1,1}}{a'_{1,1}}a'_{1,n-1} \end{pmatrix} \quad \text{et} \quad b'' = \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 - \frac{a'_{2,1}}{a'_{1,1}}b'_1 \\ ... \\ b_{n-1} - \frac{a_{n-1,0}}{a_{0,0}}b'_1 \end{pmatrix}.$$

En procédent ainsi de suite pour chaque colonne, le système (4) est alors transformé en $A^{(n-1)}x = b^{(n-1)}$ où $A^{(n-1)}$ n'a que des zeros sous la diagonale. $A^{(n-1)}$ est donc une matrice triangulaire supérieure que l'on note U. On a donc transformé le système initial (2) en le système triangulaire supérieur

$$(5) \qquad Ux = c$$

avec $c = b^{(n-1)}$. On peut alors résoudre ce système à l'aide de l'algorithme vu à la section précédente.

Il est à noter que pour que cet algorithme puisse être effectué, il faut que les coefficients rencontrés itérativement sur la diagonale $a_{0,0}$, $a'_{1,1} = a_{0,0} - \frac{a_{1,0}}{a_{0,0}} a_{0,1}$, ... , $a^{(n-2)}_{n-2,n-2}$ soient non nuls, pour qu'ils puissent servir de "pivot" pour éliminer la présence de la variable x_k dans les lignes $k+1$ jusqu'à $n-1$. Cette condition est équivalente au fait que la matrice A admette une décomposition $A = LU$ où les coefficients de la diagonale de U sont non nuls, sauf éventuellement le dernier. La décomposition LU est l'objet de la sous-section suivante. Si cette condition n'est pas satisfaite, on peut alors chercher d'autres pivots en effectuant des permutations entre les lignes du système, voir un des exercices plus bas.

Basée sur l'algorithme du pivot de Gauss que nous venons de voir, la fonction suivante `transfo_tri_sup(A,b)` prend en entrée une matrice A et un vecteur b est retourne le couple (U,c) associé à l'équation (5).

[148]:
```
def transfo_tri_sup(A,b):
    n=len(b)
    U=np.copy(A)
    c=np.copy(b)
    for j in range(n):
        for i in range(j+1,n):
            c[i]=c[i]-(U[i,j]/U[j,j])*c[j]
            U[i,:]=U[i,:]-(U[i,j]/U[j,j])*U[j,:]
    return(U,c)
```

On peut alors resoudre $A(x) = b$ en effectuant d'abord la transformation à l'aide de `transfo_tri_sup`, puis en résolvant $Ux = c$ à l'aide de `sol_tri_sup` vu dans la sous-section précédente. C'est ce qu'effectue la fonction `sol_pivot(A,b)` ci-dessous.

[149]:
```
def sol_pivot(A,b):
    (U,c)=transfo_tri_sup(A,b)
    return sol_tri_sup(U,c)
```

On la teste pour résoudre

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 0 & 3 & 1 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

[153]:
```
A4=np.array([[1,1,1],[2,1,1,],[0,3,1]])
b4=np.array([1,2,3])
x4=sol_pivot(A4,b4)
```

3.3 La décomposition LU

Supposons qu'une matrice A puisse s'écrire sous la forme

$$A = LU, \qquad L = \begin{pmatrix} 1 & 0 & & (0) \\ l_{1,0} & 1 & 0 & \\ . & . & . & 0 \\ l_{n-1,0} & l_{n-1,1} & . & 1 \end{pmatrix} \quad \text{et} \quad U = \begin{pmatrix} u_{0,0} & u_{0,1} & . & u_{0,n-1} \\ 0 & u_{1,1} & . & . \\ & 0 & . & . \\ (0) & & 0 & u_{n-1,n-1} \end{pmatrix},$$

où L est une matrice triangulaire inférieure avec des 1 sur la diagonale, et U est une matrice triangulaire supérieure dont les élements diagonaux sont non nuls. Alors, en calculant explicitement le coefficient en position (i, j) du produit LU, on observe que si $j \geq i$:

$$(6) \qquad a_{i,j} = u_{i,j} + l_{i,i-1}u_{i-1,j} + ... + l_{i,0}u_{0,j}$$

et si $j < i$:

$$(7) \qquad a_{i,j} = l_{i,j}u_{j,j} + l_{i,j-1}u_{j-1,j} + ... + l_{i,0}u_{0,j}.$$

On peut se servir de (6) et (7) pour calculer les matrices L et U. On va d'abord calculer la première ligne de U et la première colonne de L. Puis on va calculer la deuxième ligne de U et la deuxième colonne de L, et ainsi de suite. En effet, si les $k-1$ ièmes premières lignes de U et les $k-1$-ièmes premières colonnes de L sont connues, alors on calcule la k-ième ligne de U en utilisant (6):

$$u_{k,j} = a_{k,j} - l_{k,k-1}u_{k-1,j} - ... - l_{k,0}u_{0,j} \qquad \text{pour } j = k, ..., n-1,$$

et on calcule ensuite la k-ième colonne de L en utilisant (7):

$$l_{i,k} = \frac{a_{i,k} - l_{i,k-1}u_{k-1,k} - ... - l_{i,0}u_{0,k}}{u_{k,k}} \qquad \text{pour } i = k+1, ..., n-1.$$

Cet algorithme pour calculer la décomposition LU d'une matrice A (lorsque c'est possible), est implémenté par la fonction `decomp_LU` ci-dessous.

[179]:
```
def decomp_LU(A):
    n=len(A)
    L=np.identity(n)
    U=np.zeros(np.array([n,n]))
    for k in range(n):
        for j in range(k,n):
            U[k,j]=A[k,j]-np.sum(L[k,0:k]*U[0:k,j])
        for i in range(k,n):
            L[i,k]=(A[i,k]-np.sum(L[i,0:k]*U[0:k,k]))/U[k,k]
    return(L,U)
```

On teste cette fonction sur l'exemple de la matrice

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

ci-dessous.

[180]:
```
A5=np.array([[1,1,0],[0,2,3],[1,5,5]])
decomp_LU(A5)
```

[180]:
```
(array([[1., 0., 0.],
        [0., 1., 0.],
        [1., 2., 1.]]),
 array([[ 1.,  1.,  0.],
        [ 0.,  2.,  3.],
        [ 0.,  0., -1.]]))
```

Le système $Ax = b$ est maintenant équivalent à $LUx = b$. En posant $y = Ux$, on peut d'abord calculer y en résolvant $Ly = b$, puis calculer x en résolvant $Ux = y$. Ces deux systèmes sont triangulaires, et peuvent être résolus aisément avec l'algorithme vu précédemment dans ce cours.

La fonction `sol_LU(A,b)` résout le système $Ax = b$ de cette manière :

[169]:
```
def sol_LU(A,b):
    (L,U)=decomp_LU(A)
    y=sol_tri_inf(L,b)
    return sol_tri_sup(U,y)
```

On teste ci-dessous cette fonction pour résoudre

$$\begin{pmatrix} 1 & 10 & 2 \\ 0 & 20 & -1 \\ 1 & 5 & 10 \end{pmatrix} x = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}$$

[172]:
```
A6=np.array([[1,10,2],[0,22,-1],[1,5,10]])
b6=np.array([1,1,2])
x6=sol_LU(A6,b6)
```

[172]:
```
array([1., 1., 2.])
```

3.4 Exercices

Exercice 3. Dans l'algorithme de la fonction `sol_tri_inf`, le calcul de x[k] nécessite k multiplications, k additions et une division, soit un total de $2k + 1$ opérations. Le nombre total d'opérations effectuées par l'algorithme est donc $\sum_{k=0}^{n-1}(2k + 1) = n^2$, ce que l'on note $O(n^2)$. On appelle cet équivalent pour le nombre d'opérations effectuées par un algorithme sa complexité

1. Pouvez-vous estimer la complexité de l'algorithme de la fonction `transfo_tri_sup` ?
2. Pouvez-vous estimer la complexité de l'algorithme de la fonction `decomp_LU` ?
3. En déduire que cela prend un temps du même ordre de résoudre un système linéaire en appliquant la méthode du pivot directement (i.e. en utilisant la fonction `sol_pivot`) qu'en calculant d'abord la décomposition LU de la matrice puis en appliquant la méthode du pivot pour des systèmes triangulaires (i.e. en utilisant la fonction `sol_LU`)

Exercice 4. En utilisant la décomposition LU donnée par la fonction `decomp_LU(A)` ci dessus, écrire une fonction det(A) qui calcule le déterminant de A.

Le problème avec la fonction `transfo_tri_sup` est que son algorithme ne fonctionne que si les coefficients rencontrés itérativement sur la diagonale sont non nuls et peuvent servir de pivots. Il est en fait possible d'écrire un nouvel algorithme, utilisant en plus des permutations de lignes, qui permette d'éviter ce problème. C'est le but de l'exercice suivant.

Commençons par considérer le cas particulier où le premier coefficient du système est nul. Supposons donc que l'on cherche à résoudre $Ax = b$ avec une matrice A sous la forme

$$A = \begin{pmatrix} 0 & a_{0,1} & . & . & a_{0,n-1} \\ 0 & a_{1,1} & . & . & a_{1,n-1} \\ . & . & . & . & \\ 0 & a_{k-1,1} & . & . & a_{k-1,n-1} \\ a_{k,0} & a_{k,1} & . & . & a_{k,n-1} \\ a_{k+1,0} & a_{k+1,1} & . & . & a_{k+1,n-1} \\ . & . & . & . & \\ a_{n-1,0} & a_{n-1,1} & . & . & a_{n-1,n-1} \end{pmatrix}$$

c'est-à-dire que les k premiers coefficients de la première colonne de A sont nuls, et avec $a_{k,0} \neq 0$. Alors en permutant la première ligne du système $Ax = b$ avec la k-ième, on obtient le système équivalent

$$\begin{pmatrix} a_{k,0} & a_{k,1} & . & . & a_{k,n-1} \\ 0 & a_{1,1} & . & . & a_{1,n-1} \\ . & . & . & . & \\ 0 & a_{k-1,1} & . & . & a_{k-1,n-1} \\ 0 & a_{0,1} & . & . & a_{0,n-1} \\ a_{k+1,0} & a_{k+1,1} & . & . & a_{k+1,n-1} \\ . & . & . & . & \\ a_{n-1,0} & a_{n-1,1} & . & . & a_{n-1,n-1} \end{pmatrix} x = \begin{pmatrix} b_k \\ b_1 \\ . \\ b_{k-1} \\ b_0 \\ b_{k+1} \\ . \\ b_{n-1} \end{pmatrix}.$$

Puisque $a_{k,0}$ est non nul, on peut se servir de la méthode du pivot pour éliminer la variable x_0 de la seconde à la dernière ligne de ce nouveau système en suivant la méthode de pivot.

Exercice 5.

En reprenant la méthode du pivot implémentée dans `transfo_tri_sup`, et en vous aidant du cas particulier ci-dessus, écrivez un algorithme `transfo_tri_sup2(A,b)` qui pour n'importe quelle matrice inversible A, renvoie (U, c) tels que $Ax = b$ soit équivalent à $Ux = c$ avec U triangulaire supérieure.

4 La décomposition de Cholesky

Les matrices symmétriques $M = (m_{i,j})_{0 \leq i,j \leq n-1} \in \mathbb{R}^{n \times n}$ définies positives admettent une decomposition légèrement plus simple que LU et peuvent s'écrire sous la forme $M = LL^T$ (voir notes de cours) où les coefficients diagonaux de la matrice L sont non nuls.

4.1 L'algorithme

Un algorithme pour calculer une telle décomposition est donné dans la feuille d'exercices 1.

4.2 La routine de Numpy

La fonction `linalg.cholesky(M)` de numpy donne la décomposition de Cholesky d'une matrice symmétrique définie positive M. On la teste ci-dessous pour calculer la matrice L de la décomposition de Cholesky de la matrice

$$\begin{pmatrix} 10 & 1 & -1 \\ 1 & 8 & 0 \\ -1 & 0 & 12 \end{pmatrix}$$

[178]:
```
M7=np.array([[10,1,-1],[1,8,0],[-1,0,12]])
L7=np.linalg.cholesky(M0)
```

4.3 Exercices

Exercice 6 Une fois la décomposition de Cholesky connue, on peut résoudre $Mx = b$ en résolvant d'abord le système triangulaire inférieur $Ly = b$, puis le système triangulaire supérieur $L^T x = y$. Implémentez une fonction `sol_chol(M,b)` qui prend en entrée une matrice symmétrique définie positive M, et résout $Mx = b$ de cette manière. Vous pouvez utiliser les fonctions `sol_tri_inf` et `sol_tri_sup` de la section précédente,

5 Complément : La décomposition PLU

Toutes les matrices M n'admettent pas forcément de décomposition LU, mais toutes admettent une décomposition $M = PLU$ (cf notes de cours).

5.1 L'algorithme

Nous définissons d'abord la fonction `permutation(n,i,k)` qui pour des entiers i, k, n tels que $i \neq k$ and $i, k < n$, renvoie la matrice de permutation elementaire qui permutte les i-ieme et k-ieme elements de la base canonique dans $\mathbb{R}^n$.

[25]:
```
def permutation(n,i,k):
    P=np.identity(n)
    P[i,i]=0
    P[k,k]=0
    P[i,k]=1
    P[k,i]=1
    return(P)
```

[26]:
```
permutation(3,0,1)@permutation(3,0,1)
```

[26]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

La fonction `PLU(A)` ci-dessous renvoie la decomposition PLU pour la matrice A. Elle repose pour partie sur des calculs similaires à ceux de la fonction `decomp_LU` précédemment décrite, et intégre maintenant le choix du pivot à l'aide des matrices de permutation.

[181]:
```
def PLU(A):
    n=len(A)
    U=A
    L=np.identity(n)
    P=np.identity(n)
    for j in range(0,n-1):
        if U[j,j]!=0:
            for i in range(j+1,n):
                p=-U[i,j]/U[j,j]
                U[i,j:n]=U[i,j:n]+p*U[j,j:n]
                L[i,j]=-p
        else:
            k=j+1
            while k!=n and U[k,j]==0:
                k+=1
            if k!=n:
                U=permutation(n,j,k)@U
                L=permutation(n,j,k)@L@permutation(n,j,k)
                P=P@permutation(n,j,k)
                for i in range(j+1,n):
                    p=-U[i,j]/U[j,j]
                    U[i,j:n]=U[i,j:n]+p*U[j,j:n]
                    L[i,j]=-p
    return(P,L,U)
```

On la teste ci-dessous sur l'exemple des matrices

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

[28]:
```
PLU(np.array([[0,1],[1,1]]))
```

[28]:
```
(array([[0., 1.],
        [1., 0.]]),
 array([[1., 0.],
        [0., 1.]]),
 array([[1., 1.],
```

```
        [0., 1.]]),
 array([[0., 1.],
        [1., 1.]]))
```

[29]:
```
B=np.array([[1,1,0],[1,1,0],[1,2,1]])
```

[30]:
```
PLU(B)
```

[30]:
```
(array([[1., 0., 0.],
        [0., 0., 1.],
        [0., 1., 0.]]),
 array([[1., 0., 0.],
        [1., 1., 0.],
        [1., 0., 1.]]),
 array([[1., 1., 0.],
        [0., 1., 1.],
        [0., 0., 0.]]),
 array([[1., 1., 0.],
        [1., 1., 0.],
        [1., 2., 1.]]))
```

5.2 Exercices

Exercice 7. Le temps d'exécution de la fonction `PLU(A)` peut être amélioré. En effe,t les opérations U=permutation(n,j,k)@U, L=permutation(n,j,k)@L@permutation(n,j,k) et P=P@permutation(n,j,k) mettent en jeu des multiplications de matrices (complexité $O(n^2)$). En réalité, la multiplication à gauche par permutation(n,j,k) revient à inverser les lignes j et k d'une matrice, et la multiplication à droite par permutation(n,j,k) revient à inverser les colonnes j et k d'une matrice. Ces opérations sont de complexité $O(n)$. Modifier le code à cet endroit et écrire une fonction `PLU2(A)` qui effectue ces étapes en $O(n)$ opérations.

[ ]:
```

```