

RESOLUTION DE SYSTEME LINEAIRES SOUS PYTHON

I. introduction

Pour faire du calcul matriciel ou la résolution de systèmes sous Python, le module **numpy** offre les éléments indispensables à la modélisation des tableaux de dimensions arbitraires (vecteurs, matrices et plus généralement des tableaux multidimensionnels).

Ce module doit être systématiquement importé à chaque début de programme ou de script à l'aide de : **import numpy** ou encore **import numpy as np**

Le sous module **numpy.linalg** du module **numpy** propose les fonctionnalités relatives à l'algèbre linéaire. Il doit être également importé de la même manière avant utilisation.

Le module numpy fournit le type **ndarray**, qui, bien que très proche sur le plan syntaxique du type **list**, diffère de ce dernier sur plusieurs points importants :

- la taille des tableaux numpy est fixée au moment de la création et ne peut plus être modifiée par la suite (sauf le redimensionnement à voir plus tard) ;
- les tableaux numpy sont homogènes (tous les éléments sont de même type : int, float, complex, ...).
- l'accès aux éléments d'un tableau numpy est incomparablement plus rapide, ce qui justifie pleinement leur usage pour manipuler des matrices de grandes tailles.

Dans ce chapitre, on présente en premier un rappel de quelques outils nécessaires du module numpy suivi de divers algorithmes de résolution de systèmes linéaires à implémenter en python dont on évalue leurs complexités algorithmiques et temporelles.

II. Rappels :

1. Création de vecteurs/matrices

Le tableau suivant présente les mots-clefs et syntaxes pour définir une matrice (vecteur):

Syntaxe / Rôle	Exemples
np.array(l) # l une liste, un tuple ou un intervalle → retourne le vecteur formé par les éléments de l <i>N.B: np.array peut prendre des paramètres optionnels tels que le type des éléments ou le nombre de dimensions du tableau</i>	<pre> >>> a=np.array([3,6, 2,-3]);a;type(a) array([3, 6, 2,-3]) <class 'numpy.ndarray'> >>> b=np.array(range(5));b;type(b) array([0, 1, 2, 3, 4]) <class 'numpy.ndarray'> >>>v=np.array((1,-2),dtype=complex) >>> v; type(v) array([1.+0.j, -2.+0.j]) <class 'numpy.ndarray'> >>> np.array([1, 2, 3], ndmin=2) array([[1, 2, 3]]) </pre>
np.array(ll) # ll une liste de listes ou tuple de tuples → retourne la matrice ligne formée par les éléments de ll	<pre> >>> a=np.array([[3,6],[2,-3]]); a array([[3, 6], [2, -3]]) >>>np.array(((3,6),(2,0)),dtype=float) array([[3., 6.], [2., 0.]]) </pre>

2. Création de vecteurs/matrices spécifiques

Syntaxe / Rôle	Exemples
np.zeros(N) # N un entier non nul → retourne le vecteur formé N éléments égaux à 0	<pre>>>> np.zeros(3) array ([0. 0. 0.])</pre>
np.zeros(L) # L est une liste ou tuple de 2 entiers non nuls → retourne la matrice à $L[0]$ lignes et à $L[1]$ colonnes formée de 0	<pre>>>> np.zeros((2,3)) array ([[0., 0., 0.], [0., 0., 0.]])</pre>
np.ones(N) # N un entier non nul → retourne le vecteur formé de N éléments égaux à 1	<pre>>>> np.ones(4) array ([1. 1. 1. 1.])</pre>
np.ones(L) # L est une liste ou tuple de 2 entiers non nuls → retourne la matrice à $L[0]$ lignes et $L[1]$ colonnes formée de 1	<pre>>>> np.ones([2,4]) array ([[1. 1. 1. 1.] [1. 1. 1. 1.]])</pre>
np.diag(M) # M est une matrice → retourne un vecteur contenant les éléments de la diagonale de M	<pre>>>> a=np.array([[3,6],[2,-3]]) >>> np.diag(a) array([3, -3])</pre>
np.diag(L) # L est une liste → retourne la matrice diagonale avec les éléments de L sur la diagonale	<pre>>>> a=[3,2,-3] ; np.diag(a) array([[3, 0, 0, 0], [0, 0, 2, 0], [0, 0, 0, -3]])</pre>
np.diag(L,i) # L une liste et i un entier > 0 → retourne la matrice avec les éléments de L sur la $i^{\text{ème}}$ diagonale supérieure	<pre>>>> a=[3,2,-3] ; np.diag(a,2) array([[0, 0, 3, 0, 0], [0, 0, 0, 2, 0], [0, 0, 0, 0, -3], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]])</pre>
np.diag(L,-i) # L une liste et i un entier > 0 → retourne la matrice avec les éléments de L sur la $i^{\text{ème}}$ diagonale inférieure	<pre>>>> a=[3,2,-3] ; np.diag(a,-2) array([[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [3, 0, 0, 0, 0], [0, 2, 0, 0, 0], [0, 0, -3, 0, 0]])</pre>
np.identity(N) #↔ np.eye(N) # N un entier non nul → retourne la matrice identité d'ordre N	<pre>>>> np.identity(3) # np.eye(3) array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])</pre>
np.arange(M,N,P) # où M , N et P des entiers ou des réels. Par défaut $P=1$ et $N=0$ ($P=1$. et $N=0$.) → retourne le vecteur dont les éléments sont entre M (inclus) à N (exclus) avec un pas P	<pre>>>> np.arange(5) array([0, 1, 2, 3, 4]) >>> np.arange(2,10,3) array([2, 5, 8]) >>> np.arange(1.2,5.,1.2) array([1.2, 2.4, 3.6, 4.8])</pre>
np.linspace(M,N,P) #où M , N et P des entiers ou des réels → retourne le vecteur formé de P éléments <u>uniformément répartis</u> entre M et N	<pre>>>> np.linspace(2,10,3) array([2., 6., 10.]) >>> np.linspace(2,3,5) array([2. , 2.25, 2.5 , 2.75, 3.]) >>> np.linspace(1.2,5.8,3) array([1.2, 3.5, 5.8])</pre>

3. Dimensions de matrices :

Syntaxe / Rôle	Exemples
A.shape # A est soit un vecteur soit une matrice → retourne les dimensions de A <i>N.B: Si A est une matrice, le résultat est un tuple dont le premier élément est la dimension des lignes et le second est celle des colonnes.</i>	<pre>>>>A =np.array([[3,6,2],[11,3,5], [0,6,1]],dtype=float); A array([[3., 6., 2.], [11., 3., 5.], [0., 6., 1.]]) >>> A.shape (3, 3) >>> B =np.array([3,6,2], dtype=float); >>> B.shape (3,)</pre>
A.ndim #A est une matrice → retourne le nombre de dimensions de A	<pre>>>> A.ndim 2 >>> B.ndim 1</pre>
A.size #A est une matrice → retourne le nombre d'éléments de A	<pre>>>> A.size 9 >>> B.size 3</pre>

4. Redimensionnement de matrices:

On peut passer d'un vecteur de taille n à une matrice de taille (l,c) et inversement ou passer d'une matrice de taille (l, c) à une matrice de taille (p, q) à condition que $n=lc=pq$

Syntaxe / Rôle	Exemples
A.reshape(n) # A est une matrice et n un entier → renvoie une <u>copie</u> redimensionnée de A	<pre>>>> A =np.array([[3,6,2], [11,6,0]],dtype=float);A.shape (2, 3) >>> A.reshape(6) array([3., 6., 2., 11., 6., 0.])</pre>
A.reshape(l,c) # A est une matrice, l et c deux entiers → renvoie une copie redimensionnée de A	<pre>>>> A ; A.reshape(3,2) array([[3., 6., 2.], [11., 6., 0.]]) array([[3., 6.], [2., 11.], [6., 0.]])</pre>

5. Extraction de sous-matrices

Le tableau suivant présente quelques mots-clefs et syntaxes permettant d'extraire des éléments ou des sous matrices à partir d'une matrice donnée illustrés par des exemples:

Sachant que :

A est la matrice

```
array([[ 3.,  6.,  2.],
       [11.,  3.,  5.],
       [ 0.,  6.,  1.]])
```

et i, j deux entiers

Syntaxe / Rôle	Exemples
A[i][j] ou A[i,j] → extrait le $(i+1, j+1)^{\text{ème}}$ élément de A	<pre>>>> A[2][1]; A[0,0]; A[-2][-1] 6.0 3.0 5.0</pre>

A[:,i] → extrait le vecteur correspondant à la $(i+1)^{\text{ème}}$ colonne de A	<pre>>>> A[:,2]; A[:,0];A[:,1];A[:,-2] array([2., 5., 1.]) array([3., 11., 0.]) array([6., 3., 6.]) array([6., 3., 6.])</pre>
A[:, :i] → extrait la sous matrice formée par les i premières colonnes de A	<pre>>>> A[:, :2]; A[:, :-1];A[:, :1]; A[:, :-2] array([[3., 6.], [11., 3.], [0., 6.]]) array([[3., 6.], [11., 3.], [0., 6.]]) array([[3.], [11.], [0.]]) array([[3.], [11.], [0.]])</pre>
A[:, i:] → extrait la sous matrice formée à partir de la $(i+1)$ colonne de A jusqu'à la dernière.	<pre>>>> A[:,2:]; A[:, -1:]; A[:, -2:];A[:,0:] array([[2.], [5.], [1.]]) array([[2.], [5.], [1.]]) array([[6., 2.], [3., 5.], [6., 1.]]) array([[3., 6., 2.], [11., 3., 5.], [0., 6., 1.]])</pre>
A[:, i:j+1] → extrait la sous matrice à partir de la $i^{\text{ème}}$ à la $j^{\text{ème}}$ colonne de A	<pre>>>> A[:,2:2];A[:,1:2];A[:,0:2] array([], shape=(3, 0), dtype=float64) array([[6.], [3.], [6.]]) array([[3., 6.], [11., 3.], [0., 6.]])</pre>
A[i, :] → extrait le vecteur ligne formé par la $(i+1)^{\text{ème}}$ ligne de A	<pre>>>> A[1,:]; A[2,:]; array([11., 3., 5.]) array([0., 6., 1.])</pre>
A[:i, :] → extrait la sous matrice formée par les i premières lignes de A	<pre>>>> A[:1,:];A[:2,:];A[:3,:] array([[3., 6., 2.]]) array([[3., 6., 2.], [11., 3., 5.]]) array([[3., 6., 2.], [11., 3., 5.], [0., 6., 1.]])</pre>
A[i:, :] → extrait la sous matrice formée par la $(i+1)^{\text{ème}}$ ligne de A jusqu'à la dernière.	<pre>>>> A[1:,:];A[-1:,:]; array([[11., 3., 5.], [0., 6., 1.]]) array([[0., 6., 1.]])</pre>

A[i:j+1, :] → extrait la sous matrice à partir de la $i^{\text{ème}}$ à la $j^{\text{ème}}$ ligne de A	<pre>>>> A[1:2, :] ; A[1:3, :] ; A[-1:-3:-1, :]; array([[11., 3., 5.]]) array([[11., 3., 5.], [0., 6., 1.]]) array([[0., 6., 1.], [11., 3., 5.]])</pre>
--	---

6. Accès non contigu aux données (fancy indexing):

On peut accéder au contenu d'un tableau numpy tant en lecture qu'en écriture en utilisant le *fancy indexing*.

Dans ce cas, les positions dans un tableau ne procèdent plus nécessairement par des coupes mais peuvent être données dans un ordre quelconque : si A est un tableau et L une liste (ou un tuple) d'indices, alors A[L] renvoie le tableau formé des éléments A[i] où i est dans L.

Il est donc possible de :

- échanger les lignes i et j d'un tableau A par : A[[i, j]] = A[[j, i]]
- échanger les colonnes i et j d'un tableau A par : A[:, [i, j]] = A[:, [j, i]]

Sachant que : A est la matrice `array([[ 3., 6., 2.],`
`[ 11., 3., 5.],`
`[ 0., 6., 1.]])`

et i, j deux entiers

Syntaxe / Rôle	Exemples
A[[i,j]] → donne la matrice formée par la ligne d'indice i et la ligne d'indice j de A.	<pre>>>> A[[2,0]] array([[0., 6., 1.], [3., 6., 2.]]) >>> A[[2,1],[1,0]] array([6., 11.]])</pre>
A[[i, j]] = A[[j, i]] → échange la ligne i et la ligne j de A	<pre>>>> A[[2, 0]] = A[[0, 2]] ; A array([[0., 6., 1.], [11., 3., 5.], [3., 6., 2.]])</pre>
A[:,[i, j]] = A[:,[j, i]] → échange la colonne i et la colonne j de A	<pre>>>> A[:, [2, 0]] = A[:, [0, 2]] ; A array([[2., 6., 3.], [5., 3., 11.], [1., 6., 0.]])</pre>

7. Outils mathématiques pour les matrices sous python

Les outils mathématiques à notre disposition sous Python sont les suivants.

Sachant que A est la matrice : `array([[ 3., 6., 2.],`
`[ 11., 3., 5.],`
`[ 0., 6., 1.]])`

Syntaxe / Rôle	Exemples
A.max() → retourne le maximum global des éléments de A	<pre>A=np.array([[3,6,2], [11,3,5], [0,6,1]], dtype=float) ; A array([[3., 6., 2.], [11., 3., 5.], [0., 6., 1.]]) A.max() 11.</pre>
A.max(0) → retourne un vecteur où chaque	<pre>>>> A.max(0) array([11., 6., 5.])</pre>

élément est le maximum de chaque colonne	
A.max(1) → retourne un vecteur où chaque composante est le maximum de chaque ligne	<pre>>>> A.max(1) array([6., 11., 6.]</pre>
A.min() → retourne le minimum global des éléments	<pre>>>> A.min() 0.0</pre>
A.min(0) → retourne un vecteur où chaque composante est le minimum de chaque colonne	<pre>>>> A.min(0) array([0., 3., 1.]</pre>
A.min(1) → retourne un vecteur où chaque composante est le minimum de chaque ligne	<pre>>>> A.min(1) array([2., 3., 0.]</pre>
A.sum() → retourne la somme totale des éléments	<pre>>>> A.sum() 37.0</pre>
A.sum(0) → retourne un vecteur où chaque composante est la somme de chaque colonne	<pre>>>> A.sum(0) array([14., 15., 8.]</pre>
A.sum(1) → retourne un vecteur où chaque composante est la somme de chaque ligne	<pre>>>> A.sum(1) array([11., 19., 7.]</pre>
A.prod() → retourne le produit de tous les éléments	<pre>>>> A.prod() 0.0</pre>
A.prod(0) → retourne un vecteur où chaque composante est le produit de chaque colonne	<pre>>>> A.prod(0) array([0., 108., 10.]</pre>
A.prod(1) → retourne un vecteur où chaque composante est le produit de chaque ligne	<pre>>>> A.prod(1) array([36., 165., 0.]</pre>
A.mean() → retourne la moyenne de tous les éléments	<pre>>>> A.mean() 4.1111111111111107</pre>
A.mean(0) → retourne un vecteur où chaque composante est la moyenne de chaque colonne	<pre>>>> A.mean(0) array([4.66666667, 5. , 2.66666667])</pre>
A.mean(1) → retourne un vecteur où chaque composante est la moyenne de chaque ligne	<pre>>>> A.mean(1) array([3.66666667, 6.33333333, 2.33333333])</pre>

A + B # B est une matrice → retourne la matrice somme	<pre> >>> B=np.array([[-1,7,2],[13,1,6],[0,-6,-8]],dtype=float); >>> B array([[-1., 7., 2.], [13., 1., 6.], [0., -6., -8.]]) >>> A + B array([[2., 13., 4.], [24., 4., 11.], [0., 0., -7.]]) </pre>
t * A # t un scalaire → retourne la matrice où tous les éléments sont multipliés par t	<pre> >>> 5*A array([[15., 30., 10.], [55., 15., 25.], [0., 30., 5.]]) </pre>
np.dot(A,B) #B une matrice → retourne la matrice produit <i>Attention: A * B donne le produit terme à terme de chacun des éléments de A et B et non pas le produit matriciel</i>	<pre> >>> np.dot(A,B) array([[75., 15., 26.], [28., 50., 0.], [78., 0., 28.]]) </pre>
f(A) # f une fonction → retourne la matrice dont les éléments sont les éléments images de A par f	<pre> >>> f=lambda x: x**2+1 >>> f(A) array([[10., 37., 5.], [122., 10., 26.], [1., 37., 2.]]) </pre>

8. Calcul matriciels:

Syntaxe / Rôle	Exemples
A.transpose() → retourne la matrice transposée de A	<pre> A=np.array([[3,6,2],[11,3,5],[0,6,1]]) >>> A.transpose() array([[3, 11, 0], [6, 3, 6], [2, 5, 1]]) </pre>
A.conj() → retourne la matrice conjuguée de A	<pre> >>>B=np.array([[3+6j,2],[11,-3-1j]]);B array([[3.+6.j, 2.+0.j], [11.+0.j, -3.-1.j]]) >>> B.conj() array([[3.-6.j, 2.-0.j], [11.-0.j, -3.+1.j]]) </pre>
A.conj().transpose() → retourne la matrice adjointe de A	<pre> >>> B.conj().transpose() array([[3.-6.j, 11.-0.j], [2.-0.j, -3.+1.j]]) </pre>
np.linalg.inv(A) → retourne la matrice inverse de A	<pre> >>>A=np.array([[11,3],[5,1]]);A array([[11, 3], [5, 1]]) >>> np.linalg.inv(A) array([[-0.25, 0.75], [1.25, -2.75]]) </pre>
np.linalg.det(A) → retourne le déterminant de A	<pre> >>> np.linalg.det(A) -4.0000000000000018 </pre>

np.linalg.solve(A,b) # b un vecteur → retourne le vecteur solution de $Ax=b$	<pre>>>> b=np.array([1,1]); b array([1, 1]) >>> np.linalg.solve(A,b) array([0.5, -1.5])</pre>
np.linalg.eigvals(A) → retourne le vecteur des valeurs propres de A	<pre>>>> np.linalg.eigvals(A) array([12.32455532, - 0.32455532])</pre>
np.linalg.eig(A) → retourne un tuple formé par le vecteur des valeurs propres de A ainsi que la matrice des vecteurs propres de A	<pre>>>> np.linalg.eig(A) (array([12.32455532, - 0.32455532]), array([[0.91480224, -0.25607791], [0.40390206, 0.9665615]]))</pre>
np.inner(x,y) # x et y sont deux vecteurs → retourne la valeur du produit scalaire de x et y	<pre>>>> x=np.array([2,5]) >>> y=np.array([3,4]) >>> np.inner(x,y) 26</pre>

9. Méthodes sur les lignes et les colonnes d'une matrice :

Syntaxe / Rôle	Exemples
np.insert(A,i,V,0) # A est une matrice, i un entier et V un vecteur ou une valeur (vecteur contenant la même valeur V). → insère V dans A juste avant la ligne d'indice i	<pre>>>> np.insert(A,0,0,0) array([[0., 0., 0.], [3., 6., 2.], [11., 3., 5.], [0., 6., 1.]]) >>> b=np.array([-1,-2,-3]); np.insert(A,0,b,0) array([[-1., -2., -3.], [3., 6., 2.], [11., 3., 5.], [0., 6., 1.]])</pre>
np.insert(A,i,V,1) # A est une matrice, i un entier et V une valeur (vecteur contenant la même valeur V) ou un vecteur. → insère V dans A juste avant la colonne d'indice i	<pre>>>> np.insert(A,0,0,1) array([[0., 3., 6., 2.], [0., 11., 3., 5.], [0., 0., 6., 1.]]) >>> b=np.array([-1,-2,-3]); np.insert(A,0,b,1) array([[-1., 3., 6., 2.], [-2., 11., 3., 5.], [-3., 0., 6., 1.]])</pre>
np.delete(A,i,0) # A est une matrice, i un entier → supprime la ligne d'indice i de A	<pre>>>> np.delete(A,2,0) array([[3., 6., 2.], [11., 3., 5.]])</pre>
np.delete(A,i,1) # A est une matrice, i un entier. → supprime la colonne d'indice i de A	<pre>>>> np.delete(A,2,1) array([[3., 6.], [11., 3.], [0., 6.]])</pre>
np.delete(A,L,0) # A est une matrice, L une liste d'indices. → supprime les lignes de A dont les indices sont dans L	<pre>>>> np.delete(A,[2,0],0) array([[11., 3., 5.]])</pre>

np.delete(A,L,1) # A est une matrice, L une liste d'indices. → supprime les colonnes de A dont les indices sont dans L	<pre>>>> np.delete(A, [2,0], 1) array([[6.], [3.], [6.]])</pre>
np.append(A,M,0) # A et M sont 2 matrices et i un entier. → ajoute M dans A après la dernière ligne	<pre>>>> M=np.array([[-1, -2, -3], [3,2,1]]); >>> np.append(A,M,0) array([[3., 6., 2.], [11., 3., 5.], [0., 6., 1.], [-1., -2., -3.], [3., 2., 1.]]) >>> V=np.array([[-1, -2, -3]]); >>> np.append(A,V,0) array([[3., 6., 2.], [11., 3., 5.], [0., 6., 1.], [-1., -2., -3.]])</pre>
np.append(A,V,1) # A et M sont des matrices et i un entier. → ajoute M dans A après la dernière colonne	<pre>>>> M=np.array([[-1], [-2], [-3]]); >>> np.append(A,M,1) array([[3., 6., 2., -1.], [11., 3., 5., -2.], [0., 6., 1., -3.]]) >>> V=np.array([[-1, -2, -3]]); >>> np.append(A,M,1) array([[3., 6., 2., -1.], [11., 3., 5., -2.], [0., 6., 1., -3.]])</pre>

III. Résolution des Système Linéaires

On se propose de programmer quelques méthodes permettant de résoudre des systèmes de n équations à n inconnues de la forme :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases}$$

Nous noterons ce système sous la forme matricielle

$$Ax = b$$

Où A , la matrice d'ordre n à coefficients a_{ij} avec $1 \leq (i, j) \leq n$; x le vecteur à déterminer ayant les coefficients x_i et b le vecteur à coefficients b_i avec $1 \leq i \leq n$.

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

Un système linéaire $Ax = b$ admet une unique solution s'il est un système de Cramer. La matrice A est donc inversible.

Hypothèses de travail:

- On peut résoudre toutes les parties en utilisant des listes et des listes de listes mais dans le cadre de ce travail on vous demande d'utiliser les tableaux numpy.
- Pour chacune des méthodes, penser à travailler sur une copie de la matrice et du vecteur pour ne pas altérer les données d'origine.

Afin de créer la matrice A et le vecteur b, on vous demande d'écrire les fonctions suivantes communes à toutes les méthodes:

- Ecrire une fonction python sans paramètres, appelée **taille**, qui permet de saisir et retourner un entier ≥ 2 .
- Ecrire une fonction python, nommée **saisie_Vect** qui permet la saisie des n coefficients réels d'un vecteur.
- Ecrire une fonction python, nommée **saisie_Mat** qui permet la saisie des coefficients réels d'une matrice carrée d'ordre n .

1. Résolution du système par la méthode du pivot de Gauss

1.1. Description de la méthode

Le pivot de Gauss est une méthode qui peut s'appliquer sur des matrices ou sur des systèmes d'équations. Le but de cette méthode est de transformer une matrice ou un système d'équations en une matrice ou en un système triangulaire.

L'algorithme du pivot de Gauss se déroule en deux étapes :

- La première étape consiste à rendre triangulaire supérieure la matrice A à l'aide d'opérations élémentaires appliquées aux lignes. Si on désigne par L_i la $i^{\text{ème}}$ ligne du système linéaire, les opérations élémentaires sont :

- permuter les lignes i et j de A ainsi que celle de b : $L_i \leftrightarrow L_j$
- multiplier la ligne i par λ (avec λ une valeur complexe non nulle) : $L_i \leftarrow \lambda L_i$
- remplacer la ligne i par $L_i + \lambda L_j$: $L_i \leftarrow L_i + \lambda L_j$

Ces opérations effectuées dans n'importe quel ordre et en nombre quelconque, laissent invariant l'ensemble des solutions du système linéaire.

- La deuxième étape est une étape de remontée qui consiste à résoudre, de bas en haut, le système triangulaire obtenu à l'étape 1. On commence par la dernière équation pour déterminer le x_n puis, on substitue x_n dans l'équation de la ligne $n-1$ pour déterminer x_{n-1} et ainsi de suite. A chaque étape, on substitue aux inconnues d'une ligne les valeurs trouvées dans les lignes inférieures pour résoudre l'équation courante jusqu'à trouver toutes les solutions.

1.2. Etape de Triangularisation :

Cette étape consiste à transformer progressivement la matrice A par opérations élémentaires sur les lignes en une matrice triangulaire supérieure par des itérations successives. Le système devient alors triangulaire:

$$\left\{ \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \vdots \\ a_{nn}x_n = b_n \end{array} \right.$$

Le principe est le suivant :

- **Itération 1:**
 - On cherche un coefficient a_{11} devant x_1 qui soit non nul, appelé premier pivot. Si ce coefficient apparaît en ligne i , on permute les lignes 1 et la ligne i .
 - on élimine par la suite tous les coefficients devant x_1 à partir de la ligne 2 jusqu'à la ligne n en appliquant la relation $L_i \leftarrow L_i - \frac{a_{i1}}{a_{11}} L_1$ aux coefficients des lignes entre 2 à n . L_i représente les coefficients de ligne i de A .

A la fin de cette itération la matrice apparaît comme suit :

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ 0 & a_{22} & \dots & a_{2j} & \dots & a_{2n} \\ \vdots & a_{32} & \ddots & & & \vdots \\ 0 & \vdots & \vdots & a_{jj} & \dots & a_{jn} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & a_{n2} & & a_{nj} & \dots & a_{nn} \end{pmatrix}$$

A l'itération suivante, la ligne 1 reste inchangée, et on réapplique le même processus à la sous matrice à partir de la ligne 2.

- **Itération j :** la matrice correspondant à A à cette étape se présente comme suit :

$$\begin{pmatrix} a_{11} & a_{12} & \dots & \dots & \dots & a_{1n} \\ 0 & a_{22} & \dots & \dots & \dots & a_{2n} \\ \vdots & \ddots & \ddots & & & \vdots \\ 0 & \dots & 0 & a_{jj} & \dots & a_{jn} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & a_{nj} & \dots & a_{nn} \end{pmatrix}$$

On procède de nouveau en deux temps :

- on recherche le plus grand pivot non nul en module a_{ij} parmi les $a_{jj}, a_{j+1j}, \dots, a_{nj}$ puis on permute les lignes L_i et L_j .
- comme $a_{jj} \neq 0$, on l'utilise comme pivot pour remplacer tous les termes a_{ij} pour $i > j$ par des zéros à l'aide de l'opération élémentaire :

$$L_i \leftarrow L_i - \frac{a_{ij}}{a_{jj}} L_j \quad \forall i \in [j+1, n]$$

Remarque : Choix du pivot

Mathématiquement, tous les pivots non nuls se valent. Il n'en est pas de même du point de vue numérique : diviser par un pivot dont la valeur absolue est trop faible par rapport aux autres coefficients du système conduit à des erreurs d'arrondi importantes. Pour cela nous choisirons le plus grand en module des coefficients $a_{jj}, a_{j+1j}, \dots, a_{nj}$ comme pivot de Gauss.

En tenant compte du décalage d'indice pythonien, le pseudo algorithme de triangularisation de la matrice A est le suivant :

```

Pour j de 0 à n - 2 faire
    Trouver i entre j et n - 1 tel que  $|a_{ij}|$  soit maximal non nul
    Échanger  $L_i$  et  $L_j$ 
    Pour i de j + 1 à n - 1 faire
        
$$L_i \leftarrow L_i - \frac{a_{ij}}{a_{jj}} L_j$$

    Fin Pour
Fin Pour
    
```

1.3. Etape de remontée:

A la fin de l'étape précédente, le système triangulaire s'écrit :

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & & a_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & a_{nn} \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Il n'y a donc plus qu'à remonter le système linéaire via des substitutions.

On calcule donc les valeurs de $x_1, \dots, x_n$ par récurrence descendante. Nous avons :

$$\begin{cases} x_n = \frac{b_n}{a_{nn}} \\ x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^n a_{ij} x_j \right) \quad \forall i \in \{n-1, \dots, 1\} \end{cases}$$

En tenant compte du décalage d'indice pythonien, le pseudo algorithme de cette étape est le suivant :

```

Pour i de n - 1 à 0 faire
    Pour j de i + 1 à n - 1 faire
         $y_i \leftarrow y_i - a_{ij} x_j$ 
    Fin Pour
     $x_i \leftarrow \frac{y_i}{a_{ii}}$ 
Fin Pour
    
```

1.4. Implémentation de la méthode du pivot de Gauss en python

- 1.a) Ecrire une fonction python, nommée **recherche_pivot**, qui prend en paramètres une matrice M d'ordre n ainsi qu'un indice de ligne j , et détermine et retourne le pivot non nul associé à cette ligne.
- 1.b) Ecrire une fonction python, nommée **echange_ligne**, qui prend en paramètres une matrice M d'ordre n , un vecteur V de taille n et deux indices de lignes p et j , permute les lignes d'indices p et j de M et V .
- 1.c) Ecrire une fonction python, nommée **transvection**, qui prend en paramètres une matrice M d'ordre n , un vecteur V de taille n et un indice de ligne j , positionne à zéros les coefficients de la colonne j à partir de la ligne $j+1$ jusqu'à la dernière.

- 1.d) Ecrire une fonction python, nommée **triangulaire**, qui prend en paramètres une matrice M d'ordre n et un vecteur V de taille n et qui rend triangulaire la matrice M en appelant les fonctions adéquates.
- 1.e) Ecrire une fonction python, nommée **remontee**, qui prend en paramètres une matrice T d'ordre n triangulaire supérieure et un vecteur V de taille n et retourne un vecteur X solution du système triangulaire.
- 1.f) Ecrire une fonction python, nommée **methode1**, qui prend en paramètres une matrice M et un vecteur V et retourne le vecteur X , solution du système.

2. Résolution du système par la méthode de Cramer

2.1. Description de la méthode

Dans cette méthode, et pour chaque composante x_i du vecteur X , on construit une nouvelle matrice obtenue en remplaçant les composantes de la colonne j de la matrice A par les composantes du vecteur b . La composante x_j est calculée par division du déterminant de cette nouvelle matrice par celui de A . Il est nécessaire que le déterminant de A soit non nul.

$$x_j = \frac{\begin{vmatrix} a_{11} & \dots & a_{1j-1} & b_1 & a_{1j+1} & \dots & a_{1n} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & \dots & a_{nj-1} & b_n & a_{nj+1} & \dots & a_{nn} \end{vmatrix}}{\det(A)}$$

2.2. Implémentation en python de la méthode de Cramer

- 2.a) Ecrire une fonction python, nommée **minus**, qui permet à partir d'une matrice M d'ordre n de construire et retourner une matrice d'ordre $(n-1)$ contenant les éléments de M privée de la $i^{\text{ème}}$ ligne et la $j^{\text{ème}}$ colonne.
N.B: L'utilisation de la méthode `delete` du module `numpy` est interdite.
- 2.b) Ecrire une fonction python, nommée **replace**, qui permet à partir d'une matrice M d'ordre n et d'un vecteur V à n composantes de construire et retourner une matrice d'ordre n dans laquelle la $j^{\text{ème}}$ colonne de M est remplacée par les éléments du vecteur V .
- 2.c) Ecrire une fonction python, nommée **methode2**, qui permet de résoudre (si possible) le système $Ax = b$ par utilisation de la méthode de CRAMER à partir d'une matrice M d'ordre n et d'un vecteur V à n composantes.

3. Résolution du système par calcul de l'inverse

3.1. Description de la méthode

A partir de l'équation $Ax = b$ et en supposant que la matrice A est inversible, on peut calculer le vecteur solution x par la relation :

$$x = A^{-1}b$$

On rappelle que chaque coefficient x_i du vecteur résultat x est obtenu par la relation :

$$x_i = \sum_{j=1}^n a_{ij} b_j$$

avec a_{ij} le coefficient de A^{-1} d'indices ij et b_j le coefficient d'indice j du vecteur b .

On note A_{ij} la matrice obtenue en supprimant la $i^{\text{ème}}$ ligne et la $j^{\text{ème}}$ colonne de A .

On pose AD une matrice de même ordre que A , dont chaque coefficient ad_{ij} est calculé par la relation :

$$ad_{ij} = (-1)^{i+j} \det(A_{ij})$$

Le calcul de l'inverse est donné par la relation :

$$A^{-1} = \frac{(AD)^t}{\det(A)}$$

où $(AD)^t$ représente la transposée de AD . On rappelle que la transposée est obtenue en inversant les lignes et les colonnes.

3.2. Implémentation en python de la méthode de résolution par calcul de l'inverse

N.B: ne pas utiliser les fonctions de calcul d'inverse et de transposé proposées par le module numpy.

- 3.a) Ecrire une fonction python, nommée **calcul_AD**, qui permet à partir d'une matrice M d'ordre n de construire et retourner une matrice d'ordre n dont les coefficients sont les ad_{ij} de la matrice AD données par la relation.
- 3.b) Ecrire une fonction python, nommée **transposer**, qui permet à partir d'une matrice M d'ordre n de construire et retourner la matrice transposée de M .
- 3.c) Ecrire une fonction python, nommée **inverser**, qui permet à partir d'une matrice M d'ordre n de construire et retourner la matrice inverse de M .
- 3.d) Ecrire une fonction python, nommée **methode3**, qui permet de résoudre (si possible) le système $Ax = b$ par calcul de l'inverse à partir d'une matrice M d'ordre n et d'un vecteur V à n composantes réelles.

4. Application et comparaison des différentes méthodes :

Nous voulons comparer les différentes méthodes de résolution du système présentées dans les parties 1, 2 et 3. Le temps d'exécution pourrait être obtenu en faisant appel à la méthode `time()` du module `time` avant et après l'appel de chaque fonction dont on désire calculer le temps d'exécution. La méthode `time()` retourne le temps CPU. Il suffit donc de calculer la différence entre les deux valeurs retournées par `time()` pour connaître la durée d'exécution d'une fonction donnée.

On prendra comme exemple d'application le système suivant:

$$S = \begin{cases} \frac{1}{2}x_1 + \frac{1}{3}x_2 + \frac{1}{4}x_3 = 1 \\ \frac{1}{3}x_1 + \frac{1}{4}x_2 + \frac{1}{5}x_3 = 2 \\ \frac{1}{4}x_1 + \frac{1}{5}x_2 + \frac{1}{6}x_3 = 3 \end{cases}$$

Travail demandé

- 4.a) Saisir la taille de la matrice.
- 4.b) Définir la matrice A constituée par les coefficients en x_i du système S .
- 4.c) Définir le vecteur b constitué par les coefficients du vecteur résultat du système S .
- 4.d) Ecrire une fonction python, appelée **comparaison**, qui permet la résolution du système S par les différentes méthodes de résolution présentées en 1, 2, et 3. Indiquer la méthode la plus rapide pour la résolution de ce système et comparer le résultat de la meilleure méthode avec celui donné par le sous module `linalg` du module `numpy`.