

Cours 15 : Calcul scientifique. Matrices - Système d'équations linéaires

MPSI-PCSI - Lycée Thiers

Calcul matriciel en python

Tableaux bi-dimensionnel et matrices avec `numpy`

Méthode du pivot de Gauss

- L'algorithme du pivot de Gauss

- Limitations

- Amélioration de l'algorithme

- Complexité de l'algorithme

Calcul matriciel en python

- Rappel : Soit $M \in \mathcal{M}_{n,p}(\mathbb{R})$ une matrice à n lignes, p colonnes et coefficients réels :

$$M = (M_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq p}} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,p-1} & a_{1,p} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,p-1} & a_{2,p} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,p-1} & a_{n-1,p} \\ a_{n,1} & a_{n,2} & \cdots & a_{n,p-1} & a_{n,p} \end{pmatrix}$$

Si $N \in \mathcal{M}_{p,q}(\mathbb{R})$, $N = (N_{i,j})_{\substack{1 \leq i \leq p \\ 1 \leq j \leq q}}$ alors le produit matriciel $M \times N$ est la matrice de $\mathcal{M}_{n,q}(\mathbb{R})$ définie par :

$$\forall i, j \in [[1, n]] \times [[1, q]], \quad (M \times N)_{i,j} = \sum_{k=1}^p M_{i,k} \times N_{k,j}$$

- Le produit matriciel des matrices M et N est défini dès que :

Nombre de colonnes de M = nombre de lignes de N

- En particulier pour un vecteur $x \in \mathbb{R}^p$ et $X \in \mathcal{M}_{p,1}$ la matrice colonne de ses composantes dans une base $\mathcal{B}$ de $\mathbb{R}^p$:

$$M \times X = \begin{pmatrix} \sum_{k=1}^p a_{1,k} x_k \\ \sum_{k=1}^p a_{2,k} x_k \\ \vdots \\ \sum_{k=1}^p a_{n,k} x_k \end{pmatrix} \in \mathcal{M}_{n,1}(\mathbb{R})$$

représente un vecteur $Y = M \times X$ dans $\mathbb{R}^n$ dès qu'une base $\mathcal{B}'$ de $\mathbb{R}^n$ est fixée.

Calcul matriciel en python

- Pour représenter une matrice en python , on peut utiliser une liste dont chaque élément est une liste de nombres : une ligne de la matrice. Par exemple :

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

se représentera par la liste :

```
M = [[1,2,3], [4,5,6]]
```

- L'élément ligne $i \in [[1,2]]$ colonne $j \in [[1,2,3]]$ de M , s'obtient alors par :

```
M[i][j]
```

à ceci près que i est dans $[[0,1]]$ et j est dans $[[0,2]]$ (on débute en 0, non plus en 1) :

```
>>> M[0][0]
1
>>> M[0][1]
2
>>> M[1][1]
5
>>> M[1][2]
6
```

Calcul matriciel en python

- Une fois ces conventions prises il est facile d'écrire une fonction appliquant le produit matriciel de 2 listes-matrices :

```
def produitMatriciel(M,N):  
    assert len(M[0])==len(N)    # Nbre colonnes de M = Nbre lignes de N?  
    n, p, q = len(M), len(N), len(N[0])  
    MN = [[0] * q for k in range(n)]    # Réserve de l'espace nécessaire  
    for i in range(n):  
        for j in range(q):  
            for k in range(p):  
                MN[i][j] += M[i][k] * N[k][j]  
    return MN
```

- De complexité $\Theta(n \times p \times q)$.

Pour le produit de 2 matrices carrées $n \times n$ (dans $\mathcal{M}_n(\mathbb{K})$) de complexité $\Theta(n^3)$.

Tableaux bi-dimensionnels avec numpy

- L'implémentation des matrices est plus aisée avec le module `numpy`. Ce module gère très efficacement les tableaux bi-dimensionnels, qui pourront correspondre à des matrices, et dispose de fonctions prédéfinies pour leur appliquer toutes les opérations du calcul matriciel.
- **La fonction** `array()` permet aussi de créer un tableau bi-dimensionnel, à partir d'une liste de listes de nombres (de même longueur) :

```
>>> import numpy as np
>>> A = np.array([[1,1,1],[0,1,1],[0,0,1]])
>>> A
array([[1, 1, 1],
       [0, 1, 1],
       [0, 0, 1]])
>>> print(A[0][0], A[0,0])
1 1
```

- On accède aux éléments d'un tableau comme en python, grâce à deux indices entre crochets : `A[0][0]`, ou plus simplement, en séparant les deux indices d'une virgule : `A[0,0]`.
- On peut appliquer du slicing, notamment pour récupérer une ligne, une colonne, ou un bloc :

```
>>> A[0,:] # Première ligne
array([1, 1, 1])
>>> A[:,1] # Deuxième colonne
array([[1],
       [1],
       [0]])
```

```
>>> A[0:2,0:2]
array([[1,1],
       [0,1]])
```

Tableaux bi-dimensionnels avec numpy

- Le type d'un tableau s'obtient grâce à la fonction `shape()`, son nombre d'élément grâce à `size()`. La fonction `reshape()` permet de changer la forme (=type) d'une matrice :

```
>>> L=np.arange(0,10)
>>> np.reshape(L,(2,5))
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> L
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> M = np.reshape(L,(2,5)) ; np.shape(M)
(2, 5)
```

- La fonction `transpose()` permet d'obtenir la transposée :

```
>>> np.transpose(A)
array([[1, 0, 0],
       [1, 1, 0],
       [1, 1, 1]])
```

Le module numpy

- Attention la multiplication '*' entre 2 tableaux bi-dimensionnels de numpy n'est pas le produit matriciel : c'est la multiplication terme à terme.
- Le produit matriciel s'obtient à l'aide de la fonction dot().
- Un vecteur peut s'écrire à l'aide d'un tableau uni-dimensionnel de numpy. Par exemple : $v = \text{np.arange}(0,3)$ ou $v = \text{array}([0, 1, 2])$ (et aussi comme la matrice colonne $v = \text{array}([[0],[1],[2]])$).

Exemple : avec la matrice 3×3 : $A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ et le vecteur $v = (0, 1, 2)$.

Le produit $A \times v$ est défini et égal à $(3, 3, 2)$.

```
>>> v = array([0,1,2])
>>> np.dot(A,v)
array([3, 3, 2])
```

Pour effectuer le produit scalaire de 2 "vecteurs" :

$$\left\langle \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \right\rangle = x \cdot x' + y \cdot y' + z \cdot z'$$

utiliser la fonction vdot(), par exemple :

```
u, v = np.array([1,1,1]), np.array([1,2,3])
np.vdot(u,v) = 6
```

Le module numpy

- Attention la multiplication '*' entre tableaux s'effectue terme à terme. En particulier l'opération $A ** 2$ correspond à une élévation au carré terme à terme :

```
>>> A ** 2
array([[1 1 1]
       [0 1 1]
       [0 0 1]])
>>> (2*A) ** 2
array([[4 4 4]
       [0 4 4]
       [0 0 4]])
```

- Pour élever A au carré faire plutôt :

```
>>> np.dot(A,A)
array([[1, 2, 3],
       [0, 1, 2],
       [0, 0, 1]])
```

- Pour élever une matrice carrée A à une puissance n :

```
>>> B = A
>>> for i in range(1,n):
       B = np.dot(A,B)    # à la sortie de boucle B contient  $A^n$ 
```

- De même l'inversion $A ** -1$ se fait terme à terme. Elle produira ici une erreur (division par 0) alors même que la matrice A est inversible.

matrix()

- On peut aussi définir une matrice à partir d'une liste ou d'une chaîne de caractère qui comprenne la multiplication matricielle, grâce à `matrix()` :

```
>>> A = np.matrix([[1,1,1],[0,1,1],[0,0,1]])
>>> A
matrix([[1, 1, 1],
        [0, 1, 1],
        [0, 0, 1]])
>>> B = np.matrix('1 2 3; 2 3 4; 4 5 6')
>>> B
matrix([[1, 2, 3],
        [2, 3, 4],
        [4, 5, 6]])
>>> A * B
matrix([[ 7, 10, 13],
        [ 6,  8, 10],
        [ 4,  5,  6]])
```

Le produit matriciel est correct. L'inversion aussi :

```
>>> A ** -1
matrix([[ 1., -1.,  0.],
        [ 0.,  1., -1.],
        [ 0.,  0.,  1.]])
```

Attention, erreur lorsque la matrice n'est pas inversible.

Le module numpy

- Certaines fonctions plus spécifiques à l'algèbre linéaire sont disponibles dans le sous-module `linalg` de `numpy`. Toutes les fonctions de ce sous-module devront être saisies avec le préfixe `"np.linalg."`.
- Pour l'inversion de matrice : utiliser la fonction `inv()` du sous-module `linalg` :

```
>> np.linalg.inv(A)
array([[ 1., -1.,  0.],
       [ 0.,  1., -1.],
       [ 0.,  0.,  1.]])
```

- le sous-module `linalg` contient aussi, entre autres :
 1. la fonction `det()` qui retourne le déterminant d'une matrice.
 2. La fonction `solve(A,b)` qui résout le système linéaire de matrice A et de vecteur second membre b.

Exemple : résoudre le système linéaire :

$$\begin{cases} x & -y & +z & = & 1 \\ -x & +y & +z & = & 1 \\ 2x & -y & -z & = & 0 \end{cases}, \text{ sous forme matricielle : } \begin{pmatrix} 1 & -1 & 1 \\ -1 & 1 & 1 \\ 2 & -1 & -1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

```
>>> A=np.array([[1,-1,1],[-1,1,1],[2,-1,-1]])
>>> b= np.array([1,1,0])
>>> np.linalg.solve(A,b)      # avec solve()
array([ 1.,  1.,  1.] )
```

Le module numpy

<code>array(l)</code>	crée un tableau à partir d'une liste l
<code>arange(a,b,k)</code>	crée un vecteur dont les coefs sont les $a+k.N$ entre a (inclu) et b (exclu).
<code>linspace(a,b,n)</code>	crée un vecteur de n valeurs régulièrement espacées entre a et b (inclus)
<code>zeros(p)</code>	crée un tableau de taille p rempli de zéros
<code>ones(p)</code>	crée un tableau de taille p rempli de uns
<code>zeros((p,q))</code>	crée un tableau de taille (p,q) rempli de zéros
<code>ones((p,q))</code>	crée un tableau de taille (p,q) rempli de uns
<code>shape()</code>	pour obtenir la taille d'un tableau (= type d'une matrice)
<code>size()</code>	pour obtenir le nombre d'éléments d'un tableau
<code>reshape()</code>	pour redimensionner un tableau
<code>dot()</code>	pour effectuer un produit matriciel de 2 matrices
<code>vdot()</code>	pour effectuer un produit scalaire de 2 "vecteurs"
<code>transpose()</code>	pour transposer une matrice
<code>mean()</code>	valeur moyenne d'un tableau
<code>matrix()</code>	pour créer une matrice
Avec le sous-module linalg :	
<code>inv()</code>	inversion d'une matrice
<code>det()</code>	déterminant d'une matrice
<code>matrix_rank()</code>	rang d'une matrice
<code>solve(A,b)</code>	résolution du système linéaire $A.X = b$

numpy contient aussi constantes et fonctions mathématiques usuelles s'appliquant à un scalaire ou à un tableau de scalaires.

Algorithme du pivot de Gauss

- Le produit matriciel n'est pas une opération élémentaire en fonction du nombre de lignes et colonnes des matrices.

On l'a vu le produit matriciel d'une matrice M de type (n, p) par une matrice N de type (p, q) est de complexité $O(n \times p \times q)$.

Le produit de deux matrices carrées (à n lignes et n colonnes) est de complexité $O(n^3)$.

- la résolution d'un système linéaire est elle aussi coûteuse : pour la résoudre on dispose de l'algorithme du pivot de Gauss.
- C'est un algorithme exact (mathématiquement) qui permet, sous diverses variantes :
 1. La résolution d'un système linéaire de n équations à p inconnues, c'est à dire en trouver toutes les solutions.
 2. L'inversion d'une matrice carrée (invertible).
 3. le calcul du déterminant d'une matrice carrée.
 4. La calcul d'une rang d'une matrice quelconque.
- Nous allons le programmer pour la résolution d'un système de Cramer.

Un système linéaire est dit de Cramer lorsque :

C'est un système linéaire de n équations à n inconnues,
qui admet une unique solution.

Algorithme du pivot de Gauss : exemple

- Exemple : résoudre par la méthode du pivot de Gauss le système de Cramer :

$$(S): \begin{cases} x & +y & +z & = 1 \\ 2x & -y & +z & = 2 \\ x & +2y & -z & = 0 \end{cases} \quad \begin{array}{l} L_2 \leftarrow L_2 - 2L_1 \\ L_3 \leftarrow L_3 - L_1 \end{array}$$

– On choisit sur la 1ère colonne le pivot : par exemple, ici, sur la première ligne L_1 , c'est le coefficient 1 devant x .

– On applique les opérations élémentaires sur les deuxièmes et troisièmes lignes.

On obtient un système équivalent n'ayant plus de x que sur la première ligne :

$$(S) \iff \begin{cases} x & +y & +z & = 1 \\ & -3y & -z & = 0 \\ & y & -2z & = -1 \end{cases} \xLeftrightarrow{L_2 \leftrightarrow L_3} \begin{cases} x & +y & +z & = 1 \\ & y & -2z & = -1 \\ & -3y & -z & = 0 \end{cases} \quad L_3 \leftarrow L_3 + 3L_2$$

– On choisit un pivot sur la 2ème colonne : par exemple ici sur la troisième ligne, on échange alors les lignes L_2 et L_3 . On applique une opération élémentaire sur la 3ème ligne.

$$(S) \iff \begin{cases} x & +y & +z & = 1 \\ & y & -2z & = -1 \\ & & -7z & = -3 \end{cases} \iff \begin{cases} x & = 1 - y - z = 5/7 \\ y & = -1 + 2z = -1/7 \\ z & = 3/7 \end{cases}$$

– On a obtenu un système sous forme triangulaire. En "remontant" on en déduit son unique triplet solution : $\mathcal{S} = \{(5/7, -1/7, 3/7)\}$

Algorithme du pivot de Gauss : algorithme

- Il est plus pratique d'écrire le système sous forme matricielle ($\mathbb{K} = \mathbb{R}$ ou $\mathbb{C}$) :

Soient $A = (a_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} \in \mathcal{M}_n(\mathbb{K})$ et $B = (b_i)_{1 \leq i \leq n} \in \mathcal{M}_{n,1}(\mathbb{K})$. En notant $X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathcal{M}_{n,1}(\mathbb{K})$:

$$\boxed{A \cdot X = B} \iff \begin{cases} a_{1,1} \cdot x_1 + \dots + a_{1,i} \cdot x_i + \dots + a_{1,n} \cdot x_n = b_1 \\ \vdots \\ a_{i,1} \cdot x_1 + \dots + a_{i,i} \cdot x_i + \dots + a_{i,n} \cdot x_n = b_i \\ \vdots \\ a_{n,1} \cdot x_1 + \dots + a_{n,i} \cdot x_i + \dots + a_{n,n} \cdot x_n = b_n \end{cases}$$

Algorithme (pour un système de Cramer).

Etape 1 : Trigonalisation

Pour i variant de 1 à n :

Choisir un pivot sur la colonne i de A : c.à.d. un élément $a_{k,i} \neq 0$ où $k \geq i$

Echanger dans A les lignes L_i et L_k : Le pivot s'écrit alors $a_{i,i}$

Echanger dans B les éléments b_i avec b_k

Pour k variant de $i+1$ à n :

Changer dans A la ligne L_k par $L_k - a_{k,i} / a_{i,i} \cdot L_i$

Changer dans B l'élément b_k par $b_k - a_{k,i} / a_{i,i} \cdot b_i$

Algorithme du pivot de Gauss : algorithme

- Il est plus pratique d'écrire le système sous forme matricielle ($\mathbb{K} = \mathbb{R}$ ou $\mathbb{C}$) :

Soient $A = (a_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} \in \mathcal{M}_n(\mathbb{K})$ et $B = (b_i)_{1 \leq i \leq n} \in \mathcal{M}_{n,1}(\mathbb{K})$. En notant $X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathcal{M}_{n,1}(\mathbb{K})$:

$$\boxed{A \cdot X = B} \iff \begin{cases} \alpha_{1,1} \cdot x_1 + \dots + \alpha_{1,i} \cdot x_i + \dots + \alpha_{1,n} \cdot x_n = \beta_1 \\ \vdots \\ \alpha_{i,i} \cdot x_i + \dots + \alpha_{i,n} \cdot x_n = \beta_i \\ \vdots \\ \alpha_{n,n} \cdot x_n = \beta_n \end{cases}$$

Algorithme (pour un système de Cramer) :

Etape 2 : Diagonalisation

Pour i variant de n à 2 :

 Pour k variant de 1 à i-1:

 Changer dans A la ligne L_k par $L_k - \alpha_{k,i} / \alpha_{i,i} \cdot L_i$

 Changer dans B l'élément b_k par $\beta_k - \alpha_{k,i} / \alpha_{i,i} \cdot \beta_i$

Algorithme du pivot de Gauss : algorithme

- Il est plus pratique d'écrire le système sous forme matricielle ($\mathbb{K} = \mathbb{R}$ ou $\mathbb{C}$) :

Soient $A = (a_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} \in \mathcal{M}_n(\mathbb{K})$ et $B = (b_i)_{1 \leq i \leq n} \in \mathcal{M}_{n,1}(\mathbb{K})$. En notant $X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathcal{M}_{n,1}(\mathbb{K})$:

$$\boxed{A \cdot X = B} \iff \left\{ \begin{array}{ll} \alpha_{1,1} \cdot x_1 & = \gamma_1 \\ & \vdots \\ & \alpha_{i,j} \cdot x_j & = \gamma_i \\ & \vdots \\ & \alpha_{n,n} \cdot x_n & = \gamma_n \end{array} \right.$$

Algorithme (pour un système de Cramer) :

Etape 3 : Obtention de la solution

Pour i variant de 1 à n :

$$x_i = \gamma_i / \alpha_{i,i}$$

L'unique solution est : $\mathcal{S} = \{(\gamma_1 / \alpha_{1,1}, \gamma_2 / \alpha_{2,2}, \dots, \gamma_n / \alpha_{n,n})\}$ (Pour un système de Cramer.)

Algorithme du pivot de Gauss : algorithme

- Il ne faut pas travailler directement sur les matrices A et B passées en paramètres, mais sur des copies : pour ne pas modifier A et B (ce n'est pas ce qui serait attendu).
- Il est pratique de constituer une matrice M ayant n lignes et $n + 1$ colonnes, dont le bloc des n premières colonnes est A et la dernière colonne est B :

$$M = \left(\begin{array}{ccc|c} a_{1,1} & \cdots & a_{1,n} & b_1 \\ \vdots & & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,n} & b_n \end{array} \right)$$

• Algorithme du pivot de Gauss (pour un système de Cramer).

```
# Etape 1 : Trigonalisation
Pour i variant de 1 à n :
    Choisir un pivot sur la colonne i de M : c.à.d. un élément  $M_{k,i} \neq 0$  où  $k \geq i$ 
    Echanger dans M les lignes  $L_i$  et  $L_k$  : Le pivot s'écrit alors  $M_{i,i}$  # Échange
    Pour k variant de i+1 à n :
        Changer dans M la ligne  $L_k$  par  $L_k - M_{k,i} / M_{i,i} \cdot L_i$  # Transvection
# Etape 2 : Diagonalisation
Pour i variant de n à 2 :
    Pour k variant de 1 à i-1:
        Changer dans M la ligne  $L_k$  par  $L_k - M_{k,i} / M_{i,i} \cdot L_i$  # Transvection
# Etape 3 : Obtention de la solution
Pour i variant de 1 à n :
     $x_i = M_{i,n+1} / M_{i,i}$ 
retourner  $(x_1, \dots, x_n)$ 
```

Pivot de Gauss : implémentation de l'algorithme

```
import numpy as np

# Echange (Li <-> Lj) :
def echange(M,i,j):
    NbreCol = len(M[0])
    for k in range(NbreCol):
        M[i, k], M[j, k] = M[j, k], M[i, k]

# Transvection (Lk <- Lk - mu.Li) :
def transvection(M,k,i,mu):
    M[k,:] -= mu * M[i,:]

# Recherche du pivot colonne i :
def cherchePivot(M,i):
    indice = i
    while M[indice,i] == 0:
        indice += 1
    return indice

# Constitution de la matrice M :
def matriceM(A,B):
    n = len(A)
    M = np.empty((n,n+1))
    M[:, :n] = A
    M[:, n] = B
    return M
```

Pivot de Gauss : implémentation de l'algorithme

```
def PivotGauss(A,B):  
    M = matriceM(A,B)  
    n = len(M)  
  
    # Etape 1 : trigonalisation  
    for i in range(n):  
        k = cherchePivot(M,i)  
        echange(M,i,k)  
        for k in range(i+1,n):  
            mu = M[k,i] / M[i,i]  
            transvection(M,k,i,mu)  
  
    # Etape 2 : diagonalisation  
    for i in range(n-1,0,-1):  
        for k in range(0,i):  
            mu = M[k,i]/M[i,i]  
            transvection(M,k,i,mu)  
  
    # Etape 3 : solution  
    sol = np.empty(n)  
    for i in range(n):  
        sol[i] = M[i,n] / M[i,i]  
  
    return sol
```

Pivot de Gauss : implémentation de l'algorithme

- Exemples :

$$\begin{cases} x + y + z = 3 \\ x + 2y - z = 2 \\ -x + 2y + z = 2 \end{cases}$$

est un système de Cramer ayant pour unique solution : (1, 1, 1).

En effet :

$$\begin{array}{l} \xleftrightarrow[L_3 \leftarrow L_3 + L_1]{L_2 \leftarrow L_2 - L_1} \end{array} \begin{cases} x + y + z = 3 \\ y - 2z = -1 \\ 3y + 2z = 5 \end{cases} \xleftrightarrow{L_3 \leftarrow L_3 - 3L_2} \begin{cases} x + y + z = 3 \\ y - 2z = -1 \\ 8z = 8 \end{cases}$$

$$\begin{cases} x = 3 - y - z = 1 \\ y = -1 + 2z = 1 \\ z = 1 \end{cases}$$

- Résolution :

```
>>> A = np.array([[1,1,1],[1,2,-1],[-1,2,1]])
>>> B = np.array([3,2,2])
>>> np.linalg.solve(A,B)
array([1,1,1])
>>> PivotGauss(A,B)
array([1,1,1])
```

Pivot de Gauss : limitations de l'implémentation

- Jusqu'ici tout va bien... Mais à y regarder de plus près :
- Question 1 : quel est le résultat de : $12 \times \left(\frac{1}{3} - \frac{1}{4}\right) - 1$? réponse : 0.

Pourtant sous python :

```
>>> 12 * (1/3 - 1/4) - 1 == 0  
False
```

La représentation des nombres en virgules flottantes a causé une erreur d'arrondi sur $1/3$ qui s'est répercuté en une erreur sur l'évaluation de l'expression, provoquant un résultat erroné !

- Question 2 : quel est le déterminant de la matrice :

$$A = \begin{pmatrix} 1 & 1/4 & 1 \\ 1 & 1/3 & 2 \\ 0 & 1 & 12 \end{pmatrix}$$

Réponse (en décomposant le long de la dernière ligne) :

$$\begin{vmatrix} 1 & 1/4 & 1 \\ 0 & 1/3 & 2 \\ 0 & 1 & 12 \end{vmatrix} = 0 - 1 \times \begin{vmatrix} 1 & 1 \\ 1 & 2 \end{vmatrix} + 12 \times \begin{vmatrix} 1 & 1/4 \\ 1 & 1/3 \end{vmatrix} = -1 + 12 \times \left(\frac{1}{3} - \frac{1}{4}\right) = 0$$

Le déterminant est nul : la matrice est non-inversible, et de rang 2.

Pivot de Gauss : limitations de l'implémentation

- Maintenant considérons le système linéaire suivant dont la matrice associée est la matrice A précédente :

$$\left\{ \begin{array}{cccc} x & + & \frac{1}{4}y & + & z & = & 0 \\ x & + & \frac{1}{3}y & + & 2z & = & 0 \\ & & y & + & 12z & = & 1 \end{array} \right. \xLeftrightarrow{L_2 \leftarrow L_2 - L_1} \left\{ \begin{array}{cccc} x & + & \frac{1}{4}y & + & z & = & 0 \\ & & \frac{1}{12}y & + & z & = & 0 \\ & & y & + & 12z & = & 1 \end{array} \right.$$

puisque $\det(A) = 0$ le système a soit aucune soit une infinité de solutions.

$$\xLeftrightarrow{L_2 \leftarrow 12L_2} \left\{ \begin{array}{cccc} x & + & \frac{1}{4}y & + & z & = & 0 \\ & & y & + & 12z & = & 0 \\ & & y & + & 12z & = & 1 \end{array} \right. \xLeftrightarrow{L_3 \leftarrow L_3 - L_2} \left\{ \begin{array}{cccc} x & + & \frac{1}{4}y & + & z & = & 0 \\ & & y & + & 12z & = & 0 \\ & & & & 0 & = & 1 \end{array} \right.$$

Le système n'a aucune solution. Pourtant :

```
>>> A = np.array([[1,1/4,1],[1,1/3,2],[0,1,12]])
>>> B = np.array([0,0,1])
>>> PivotGauss(A,B)
[ -5.62949953e+14  3.37769972e+15 -2.81474977e+14]
>>> np.linalg.solve(A,B)
[ -7.50599938e+14  4.50359963e+15 -3.75299969e+14]
>>> np.linalg.det(A)
-2.22044604925e-16
```

Pivot de Gauss : limitations de l'implémentation

- Peut-on pallier à ce problème ?
 - Ici on pourrait utiliser le module `fraction` de python qui permet d'effectuer des calculs exacts sur les fractions (en se ramenant à des couples d'entiers).
 - Mais on retrouverait le même problème pour des coefficients non rationnels (comme $\sqrt{2}$, e , π , etc...)
 - Les arrondis inévitables dans les calculs sur les nombres à virgule flottante ont des conséquences néfastes sur les calculs informatique sur les réels :
la comparaison à zéro du résultat d'un calcul en virgule flottante n'a en général pas de sens.
comparer les résultats de deux calculs en virgule flottante n'a en général pas de sens.
(nous avons déjà rencontré cette limitation pour la recherche de racines d'un trinôme à l'aide du discriminant).

Pivot de Gauss : limitations de l'implémentation

- Par contre on peut améliorer l'algorithme pour limiter les erreurs d'arrondi dans le résultat.
- Considérons le système de Cramer suivant :

$$\begin{cases} x + y + z = 1 \\ 10^{-14}y + z = 1 \\ y + z = 1 \end{cases}$$

Il a une unique solution, qui est évidente : $\mathcal{S} = \{(0,0,1)\}$.

- Cependant :

```
>>> A = np.array([[1,1,1],[0,10**-14,1],[0,1,1]])  
>>> B = np.array([1,1,1])  
>>> PivotGauss(A,B)  
[-0.01110223  0.01110223  1. ]
```

Pivot de Gauss : Amélioration de l'algorithme

- Pour cela on améliore l'algorithme du pivot de Gauss de la manière suivante :

Dans le choix du pivot, on prend celui de plus grande valeur absolue

En effet il faut mieux diviser par un grand nombre que par un nombre proche de 0 : diviser par un petit nombre revient à multiplier par un très grand nombre, accentuant par la même les erreurs d'approximation.

- C'est la **méthode du pivot partiel**.
- Il suffit donc de changer la fonction de recherche du pivot, de la façon suivante, pour qu'elle choisisse le pivot de plus grande valeur absolue :

```
def cherchePivot(M,i):  
    indice = i  
    maximum = abs(M[i,i])  
    n = len(M) # Nombre de lignes de M  
    for k in range(i+1,n):  
        if abs(M[k,i]) > maximum:  
            indice = k  
            maximum = abs(M[k,i])  
    return indice
```

```
>>> PivotGauss(A,B)  
[ 0.  0.  1.]
```

ça fonctionne mieux. Même si ça ne met pas à l'abri d'erreurs d'arrondi.

Pivot de Gauss : Complexité de l'algorithme

- On calcule la complexité de l'algorithme du pivot de Gauss en fonction du nombre n de lignes.
 1. La **constitution de la matrice** $M = (A|B)$ se fait en temps $\Theta(n^2)$: il faut copier $n \times (n+1)$ élément. En complexité spatiale c'est aussi en $\Theta(n^2)$: il faut réserver l'espace en mémoire.

```
def matriceM(A,B):  
    n = len(A)  
    M = np.empty((n,n+1))  
    M[:, :n] = A  
    M[:, n] = B  
    return M
```

2. L'**échange de deux lignes** de M : se fait en temps $\Theta(n)$: il faut recopier $2(n+1)$ élément. En complexité spatiale c'est en $O(1)$: ne requiert pas d'espace supplémentaire.

```
def echange(M,i,j):  
    NbreCol = len(M[0])  
    for k in range(NbreCol):  
        M[i, k], M[j, k] = M[j, k], M[i, k]
```

3. La **transvection** se fait en temps $\Theta(n)$, et en espace borné $O(1)$.

```
def transvection(M,k,i,mu):  
    M[k, :] -= mu * M[i, :]
```

Pivot de Gauss : Complexité de l'algorithme

4. La recherche du pivot colonne i (de valeur absolue maximale, ou pas), se fait en temps $\Theta(n-i)$ et en espace borné :

```
def cherchePivot(M,i):
    indice = i
    maximum = abs(M[i,i])
    n = len(M) # Nombre de lignes de M
    for k in range(i+1,n):
        if abs(M[k,i]) > maximum:
            indice = k
            maximum = abs(M[k,i])
    return indice
```

- Complexité de l'étape 1, de trigonalisation :

Etape 1 : Trigonalisation

Pour i variant de 1 à n :

Choisir un pivot sur la colonne i de M : c.à.d. un élément $M_{k,i} \neq 0$ où $k \geq i$

Echanger dans M les lignes L_i et L_k : Le pivot s'écrit alors $M_{i,i}$ # Échange

Pour k variant de $i+1$ à n :

Changer dans M la ligne L_k par $L_k - M_{k,i} / M_{i,i} \cdot L_i$ # Transvection

$$\begin{aligned} \sum_{i=1}^n \left(\Theta(n-i) + \Theta(n) + \sum_{k=i+1}^n \Theta(n) \right) &= \sum_{i=1}^{n-1} \Theta(i) + \sum_{i=1}^n \Theta(n) + \sum_{i=1}^n (n-i) \Theta(n) = \Theta(n^2) + \Theta(n^2) + \Theta(n) \sum_{i=1}^{n-1} (i) \\ &= \Theta(n^2) + \Theta(n) \times \Theta(n^2) = \Theta(n^2) + \Theta(n^3) = \Theta(n^3) \end{aligned}$$

Pivot de Gauss : Complexité de l'algorithme

- Complexité de l'étape 2, de diagonalisation :

Pour i variant de n à 2 :

Pour k variant de 1 à $i-1$:

Changer dans M la ligne L_k par $L_k - M_{k,i} / M_{i,i} \cdot L_i$ # Transvection

$$\sum_{i=2}^n \sum_{k=1}^{i-1} \Theta(n) = \Theta(n) \sum_{i=2}^n \sum_{k=1}^{i-1} 1 = \Theta(n) \sum_{i=2}^n (i-1) = \Theta(n) \sum_{i=1}^{n-1} i = \Theta(n) \frac{n(n-1)}{2}$$

$$= \Theta(n) \times \Theta(n^2) = \Theta(n^3)$$

- Complexité de l'étape 3 :

Pour i variant de 1 à n :

$$x_i = M_{i,n+1} / M_{i,i}$$

retourner $(x_1, \dots, x_n)$

en $\Theta(n)$.

- Complexité de l'algorithme du pivot de Gauss : **la complexité temporelle est cubique**

	Complexité temporelle	Complexité spatiale
Constitution de la matrice M	$\Theta(n^2)$	$\Theta(n^2)$
Trigonalisation	$\Theta(n^3)$	$O(1)$
Diagonalisation	$\Theta(n^3)$	$O(1)$
Solutions	$\Theta(n)$	$O(1)$
TOTAL	$\Theta(n^3)$	$\Theta(n^2)$