Chapitre 7 : Calcul matriciel et pivot de Gauss

Instructions introduites dans ce cours : `numpy.linalg, dot, inv, solve`.

1 Calcul matriciel en Python

Dans la suite, on supposera qu'on a effectué la commande `import numpy as np`

1 A) Rappels sur la création de tableaux

On identifie, en Python, la matrice

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 1.5 & 4 & 0 \end{pmatrix}$$

avec un tableau qu'on crée avec la commande :

```
A = np.array([[1,2,3],[1.5,4,0]])
```

On rappelle que les lignes et les colonnes sont numérotées à partir de 0 dans un tableau `numpy` contrairement aux matrices en mathématiques qui sont numérotées à partir de 1 généralement.

Il existe des commandes Python du module `numpy` pour créer la matrice nulle et la matrice identité. Les commandes : `np.zeros((2,4))` et `np.eye(3)` créent respectivement la matrice nulle à 2 lignes et 4 colonnes et la matrice identité à 3 lignes et 3 colonnes.

Pour créer une matrice diagonale, on peut effectuer : `np.diag([2,4,5])` qui renvoie une matrice diagonale à 3 lignes et 3 colonnes et qui comporte 2, 4, 5 sur la diagonale.

Exercice 1.1. Écrire une fonctions Python `base(i,j,n,p)` qui renvoie une matrice à n lignes et p colonnes composée uniquement de 0 sauf à la ligne i et à la colonne j où elle comporte un 1 (ces matrices sont notées $E_{i,j}$ dans le cours sur les matrices).

1 B) Opérations sur les matrices

Pour additionner deux matrices, on utilisera l'opérateur `+` et pour multiplier par un scalaire, on utilise `s*A` où `s` est le scalaire et `A` la matrice. Pour effectuer le produit de deux matrices, on utilise la commande `np.dot(A,B)`, pour calculer AB. Pour calculer des inverses de matrices, on a besoin de la bibliothèque `numpy.linalg`.

```
>>> import numpy.linalg as npl
>>> A = np.array([[1,2],[2,1]])
>>> npl.inv(A)
array([[-0.33333333, 0.66666667],
       [0.66666667, -0.33333333])
```

On donne deux autres commandes pour obtenir la transposée, son rang : `np.transpose(A), np.rank(A)`. Pour résoudre numériquement un système linéaire en Python, par exemple le système, d'inconnues $x, y, z \in \mathbb{R}$:

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

on peut écrire

```
npl.solve([[0,-1,1],[0,1,1],[2,0,2]],[1,1,3])
```

Remarque 1.2.

1 C) Sémantique de pointeur

Comme pour une liste, la syntaxe `A=B` où `B` est objet de type `array` lie les deux matrices et toute modification de `A` induit une modification de `B`.

Pour cela, pour ne copier que le contenu d'une matrice, on utilise la syntaxe `A = np.copy(B)`

Pour agir sur une ligne d'une matrice de type `array`, on utilise une syntaxe du type `M[i,:]` où i est le numéro de ligne (pour agir sur les colonnes, on utilsierait une syntaxe du type `M[:,j]`.

Exercice 1.3. Écrivons une fonction Python `echange(M,i,j)` qui retourne, à partir d'une matrice M, la matrice M sur laquelle on a effectué l'opération élémentaire : $L_i \leftrightarrow L_j$.

2 Pivot de Gauss

2 A) Quelques rappels

On rappelle que pour résoudre un système linéaire (cette méthode a été vue en cours de mathématiques) qu'on procède de la façon suivante :

1. on écrit la matrice associée au système linéaire avec le second membre sous la forme d'une matrice « augmentée » ;
2. on échelonne la matrice pour la transformer en échelonnée réduite ;
3. on obtient ensuite l'ensemble des solutions du système en identifiant les inconnues principales et secondaires.

Exercice 2.1. Résolvons le système, d'inconnues $x, y, z \in \mathbb{R}$:

$$(S) \left\{ \begin{array}{rcl} x + 2y + 2z & = & 1 \\ 2x + 4y + z & = & 2 \\ 4x + 7y + 5z & = & 1 \end{array} \right.$$

en écrivant une matrice augmentée et en échelonnant de façon réduite

2 B) Programmation des opérations élémentaires

Le but de la suite de ce cours est de programmer la méthode du pivot de Gauss qui donne la matrice augmentée de la fin de l'exemple précédent. Pour cela, il faut programmer d'abord des fonctions qui donneront les opérations élémentaires puis une autre fonction qui utilisera les opérations élémentaires pour calculer la matrice échelonnée réduite. La fonction qui échange eux lignes a déjà été programmée pus haut, il reste les fonctions `dilatation(A,i,Lambda)` et `transvection(A,i,j,Lambda)` qui retournent, à partir d'une matrice M, la matrice A sur laquelle on a effectué respectivement les opérations : $L_i \leftarrow \lambda L_i$, $L_i \leftarrow L_i + \lambda L_j$.

```
def dilatation(A,i,Lambda):
    M = np.copy(A)
    p = M.shape[1]
    for j in range(p):
        M[i,j] = Lambda*M[i,j]
    return M

def transvection(A,i,j,Lambda):
    M = np.copy(A)
    p = M.shape[1]
    for k in range(p):
        M[i,k] = M[i,k] + Lambda*M[j,k]
    return M
```

Remarque 2.2.

2 C) Mise en place de l'algorithme d'échelonnement

On rappelle que toute matrice est équivalente en lignes à une unique matrice échelonnée réduite. L'existence se base sur l'algorithme du pivot de Gauss-Jordan (l'unicité est plus compliquée). On rappelle les étapes de cet algorithme en pseudo-code pour une matrice avec n lignes en p colonnes.

Échelonnement de la matrice M :

1. on commence par la ligne numéro 0 : $i_0 = 0$;
2. pour chaque colonne j : pour j allant de 0 à $p-1$

   on détermine s'il existe $k \in [\![i_0, n]\!]$ tel que $m_{k,j} \neq 0$ et si un tel k existe

i. on permute L_{i_0} et L_k ;

ii. on divise L_{i_0} par $m_{i_0,j}$ pour avoir un 1 en première position ;

iii. pour i allant de 0 à $n-1$, lorsque $i \neq i_0$, on fait l'opération élémentaire : $L_i \leftarrow L_i - m_{i,j}L_{i_0}$;

iv. on augmente i_0 de 1.

En Python, en suivant le pseudo-code, on obtient (on numérote les lignes pour commenter en dessous) :

```
def pivot_ech_reduite(A):
    M,n,p,i0 = np.copy(A),A.shape[0],A.shape[1],0 #1
    for j in range(p): #2
        k = i0 #3
        while k < n and M[k][j] == 0: #4
            k = k+1 #5
        if k < n: #6
            M = echange(M,i0,k) #7
            M = dilatation(M,i0,1/M[i0][j]) #8
            for i in range(0,n): #9
                if i !=i0: #10
                    Lambda = -M[i,j] #11
                    M = transvection(M,i,i0,Lambda) #12
            i0 = i0+1 #13
    return(M) #14
```

Commentaires :

On donne un autre code qui permet de résoudre le système complètement en effectuant simultanément les opérations élémentaires sur la matrice qu'on réduite ainsi que sur le second membre :

```
def resol_systeme(A,B):
    M,N,n,p,i0 = np.copy(A),np.copy(B),A.shape[0],A.shape[1],0
    for j in range(p):
        k = i0
        while k < n and M[k][j] == 0:
            k = k+1
            print(k)
        if k < n:
            M,N = echange(M,i0,k),echange(N,i0,k)
            M,N = dilatation(M,i0,1/M[i0][j]),dilatation(N,i0,1/M[i0][j])
            for i in range(0,n):
                if i !=i0:
                    Lambda = -M[i][j]
                    M,N = transvection(M,i,i0,Lambda),transvection(N,i,i0,Lambda)
            i0 = i0+1
    return M,N
```

Cela donne donc :

```
>>> A,B = np.array([[1,2,3],[4,5,6],[7,8,9]]),np.array([[1],[2],[3]])

>>> resol_systeme(A,B)
(array([[ 1.,  0., -1.],
       [-0.,  1.,  2.],
       [ 0.,  0.,  0.]]), array([[-0.33333333],
       [ 0.66666667],
       [ 0.        ]]))
```

et

```
>>> A,B = np.array([[1,2,3],[4,5,6],[7,8,10]]),np.array([[1],[2],[3]])

>>> resol_systeme(A,B)
(array([[ 1.,  0.,  0.],
       [-0.,  1.,  0.],
       [ 0.,  0.,  1.]]), array([[-0.33333333],
       [ 0.66666667],
       [ 0.        ]]))
```

Commentaires :