TP n°15

La matrice échelonnée réduite équivalente par lignes à une matrice donnée

Toutes les fonctions construites seront écrites dans le même fichier.

A – Représentation d'une matrice en Python, pour ce TP

- Dans ce TP, une matrice sera représentée par un tableau bidimensionnel de type `numpy.ndarray`. Nous aurons donc besoin de la bibliothèque `numpy` que nous chargeons à l'aide de l'instruction suivante.

```
import numpy as np
```

Par exemple la matrice

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

sera représentée comme suit en python.

```
A=np.array([[1,2,3],[4,5,6]])
```

Avec ce choix, les opérations élémentaires sur les lignes seront particulièrement aisées à coder (codage « intuitif » et moindre utilisation des boucles).

- Pour récupérer le nombre de ligne d'une matrice `M`, nous pouvons utiliser la méthode `shape`.

```
M.shape[0] # nombre de lignes de M
M.shape[1] # nombre de colonnes de M
```

Voici un exemple d'utilisation, avec la matrice `A` précédemment définie.

```
>>> A.shape[0]
2
>>> A.shape[1]
3
```

- Tous les indices de lignes et de colonnes seront en accord avec la convention adoptée en Python (qui diffère de la convention adoptée en mathématiques à un décalage près). Ainsi le coefficient situé à l'intersection de la ligne n°1 et de la colonne n°2 de `A` sera 6 (et non 2).

- Pour récupérer le coefficient situé à l'intersection de la ligne n°`i` et de la colonne n°`j` d'une matrice `M`, nous disposons de la commande suivante.

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

- Pour récupérer la ligne n°`i` d'une matrice `M`, nous disposons de la commande suivante (cf. slicing).

```
M[i,:]
```

Cette expression est un tableau unidimensionnel de type `numpy.ndarray`.

- Pour récupérer la colonne n°`j` d'une matrice `M`, nous disposons de la commande suivante (cf. slicing).

```
M[:,j]
```

Cette expression est un tableau unidimensionnel de type `numpy.ndarray`.

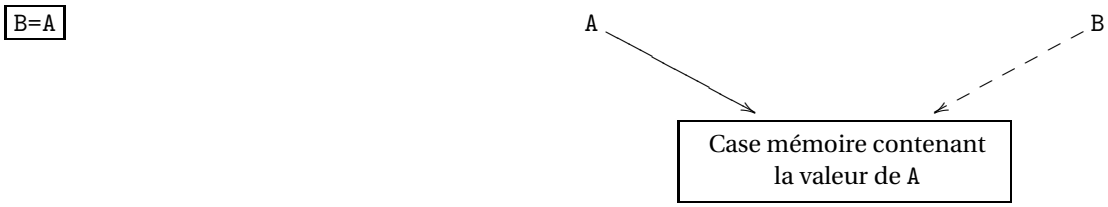
B – Adresse ou référence versus valeur

Considérons les exécutions d'instructions suivantes.

```
>>> A=np.array([[1,2,3],[4,5,6]])
>>> B=A
>>> A[0,0]=100
>>> print(A)
[[100   2   3]
 [  4   5   6]]
>>> print(B)
[[100   2   3]
[  4   5   6]]
```

Nous comprenons aisément les quatre premières instructions (du moins le croyons-nous). La dernière est en revanche surprenante. En effet la modification du premier coefficient de `A` intervenant après « l'affectation » `B=A`, il est curieux que le premier coefficient de `B` soit également modifié.

Expliquons ce qui se passe. En exécutant `B=A`, nous ne créons pas une nouvelle variable nommée `B` dans laquelle est stockée la valeur `A`, mais nous créons seulement une « copie » de la variable `A` (sans que la mémoire n'en soit affectée). Seule la référence ou l'adresse de `A` est copiée, i.e. `A` et `B` désignent la même case mémoire. Ainsi une modification de la valeur stockée dans `A` entraîne une modification de la valeur de celle strockée dans `B`, puisque `A` et `B` sont deux noms différents pour la même case mémoire.



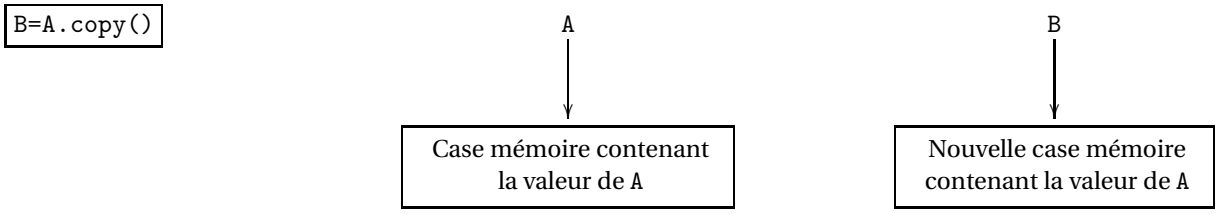
Pour connaître l'adresse d'une variable, nous pouvons utiliser la fonction `id`. Nous pouvons alors vérifier ce que nous avons affirmé plus haut.

```
>>> id(A)==id(B)
True
```

Il existe un moyen très simple de contourner ce qui pourrait être un problème ici (mais qui pourrait être un atout dans d'autres situations), grâce à la méthode `copy`. Voici comment nous pouvons modifier le code initial, afin d'avoir un comportement plus conforme à « l'habitude ».

```
>>> A=np.array([[1,2,3],[4,5,6]])
>>> B=A.copy() # application de la méthode copy
>>> A[0,0]=100
>>> print(A)
[[100   2   3]
 [  4   5   6]]
>>> print(B)
[[1 2 3]
 [  4   5   6]]
```

La modification du premier coefficient de `A` ne modifie plus la valeur de la variable `B`. Le processus en mémoire peut être schématisé par le diagramme suivant.



Conclusion : nous utiliserons systématiquement la méthode `copy` lors de l'affectation d'objets de type `numpy.ndarray`.

C – Opération élémentaire de type I sur les lignes (permutation)

Le code suivant est celui d'une fonction, nommée `permutation_faux`, d'arguments

- `M0` un tableau bidimensionnel de type `numpy.ndarray` ;
- `i1` et `i2` des indices de lignes de `M0`.

```
1  def permutation_faux(M0,i1,i2):
2      M=M0.copy() # pour ne pas modifier la matrice passée en argument
3
4      M[i1,:]=M[i2,:].copy() # bloc de 2 lignes à modifier
5      M[i2,:]=M[i1,:].copy()
6
7      return(M)
```

Nous voulions construire une fonction qui retourne la matrice obtenue en appliquant à `M0` l'opération élémentaire

$$L_{\texttt{i1}} \longleftrightarrow L_{\texttt{i2}}.$$

Exercice 1

1. Tester la fonction `permutation_faux` pour vérifier qu'elle ne répond pas à notre souhait.
2. Expliquer l'erreur.
3. Modifier le bloc de deux instructions indiqué pour construire une fonction nommée `permutation` d'arguments
   - `M0` un tableau bidimensionnel de type `numpy.ndarray` ;
   - `i1` et `i2` des indices de lignes de `M0`

   et qui retourne la matrice désirée.
4. Tester la fonction `permutation` construite.

D – Opération élémentaire de type II sur les lignes (dilatation)

Ayant des tableaux bidimensionnels de type `numpy.ndarray` pour représenter les matrices, il est aisé de multiplier une ligne d'une matrice par un scalaire.

```
>>> A=np.array([[1,2,3],[4,5,6]])
>>> A[1,:]=(-2)*A[1,:].copy() # l'application de la méthode copy pourrait être omise ici
>>> print(A)
[[  1   2   3]
 [ -8 -10 -12]]
```

Exercice 2

1. En complétant le code ci-dessous, construire une fonction nommée `dilatation` d'arguments
   - `M0` un tableau bidimensionnel de type `numpy.ndarray` ;
   - `i` un indice de ligne de `M0` ;
   - `k` un flottant (non nul...)

   et qui retourne la matrice obtenue en appliquant à `M0` l'opération élémentaire

$$L_{\texttt{i}} \longleftarrow \texttt{k}.L_{\texttt{i}}.$$

```
1  def dilatation(M0,i,k):
2      M=M0.copy() # pour ne pas modifier la matrice passée en argument
3
4      ... # à remplir
5
6      return(M)
```

2. Tester la fonction `dilatation` construite.

E – Opération élémentaire de type III sur les lignes (transvection)

Nous pouvons également très simplement ajouter à une ligne d'une matrice un multiple d'une autre ligne de cette matrice.

```
>>> A=np.array([[1,2,3],[4,5,6]])
>>> A[0,:]=A[0,:].copy()+3*A[1,:].copy() # à nouveau les .copy() sont optionnels ici
>>> print(A)
[[13 17 21]
 [ 4  5  6]]
```

Exercice 3

1. En complétant le code ci-dessous, construire une fonction nommée `transvection` d'arguments
   - `M0` un tableau bidimensionnel de type `numpy.ndarray`;
   - `i1` et `i2` des indices de lignes (distincts...) de `M0`;
   - `k` un flottant

   qui retourne la matrice obtenue en appliquant à `M0` l'opération élémentaire

$$L_{i1} \longleftarrow L_{i1} + k.L_{i2}.$$

```
1 def transvection(M0,i1,i2,k):
2     M=M0.copy() # pour ne pas modifier la matrice passée en argument
3
4     ... # à remplir
5
6     return(M)
```

2. Tester la fonction `transvection` construite.

F – Recherche du premier coefficient non nul dans un vecteur, après un indice donné

Exercice 4

1. En complétant le code ci-dessous, construire une fonction nommée `index_non_zero_coeff_after` d'arguments
   - `V0` un tableau unidimensionnel de type `numpy.ndarray`;
   - `i0` un entier supérieur ou égal à `-1`

   qui retourne
   - `-1` si tous les coefficients de `V0` d'indices strictement supérieurs à `i0` sont nuls;
   - l'indice du premier coefficient non nul de `V0` d'indice strictement supérieur à `i0` sinon.

```
1 def index_non_zero_coeff_after(V0,i0):
2     res=-1 # résultat qui sera retourné, initialisé à -1
3
4     ... # à remplir
5
6     return(res)
```

Nous donnons quelques exemples d'exécutions de la fonction `index_non_zero_coeff_after` afin de préciser notre propos.

```
>>> V=np.array([17,25,0,0,13,29,0,0])
>>> index_non_zero_coeff_after(V,-1)
0
>>> index_non_zero_coeff_after(V,1)
4
>>> index_non_zero_coeff_after(V,5)
-1
```

2. Tester la fonction `index_non_zero_coeff_after` construite.

G – La matrice échelonnée réduite équivalente par lignes à une matrice donnée

Soit M une matrice à n lignes et p colonnes. L'algorithme suivant, appelé algorithme de Gauß-Jordan [1], permet d'obtenir l'unique matrice échelonnée réduite équivalente par lignes à M.

index_last_pivot ← -1 # indice du dernier pivot trouvé
j ← 0 # indice de parcours des colonnes

Tant que (j < p et index_last_pivot < n) faire : # Traitement de la colonne n°j

  Si la colonne n°j possède un coefficient non nul d'indice de ligne > index_last_pivot, alors :

    i0 ← l'indice de ligne du 1er coefficient non nul de la colonne n°j d'indice de ligne > index_last_pivot

    index_last_pivot ← index_last_pivot+1 # un nouveau pivot apparaîtra dans la colonne n°j

    $L_{i0} \leftrightarrow L_{\text{index\_last\_pivot}}$ # on met le pivot de la colonne n°j au « bon endroit »

    $L_{\text{index\_last\_pivot}} \leftarrow 1/M[\text{index\_last\_pivot}, j].L_{\text{index\_last\_pivot}}$ # on met le pivot de la colonne n°j à 1

    Pour tout indice de ligne i différent de index_last_pivot faire :

      $L_i \leftarrow L_i - M[i,j].L_{\text{index\_last\_pivot}}$ # on « tue » les coefficients au dessus et en dessous du pivot de la colonne n°j

  j=j+1 # on passe à la colonne suivante

Exercice 5

1. En complétant le code ci-dessous, construire une fonction nommée `echelonnee_reduite` d'arguments
   - `M0` un tableau bidimensionnel de type `numpy.ndarray` ;

   qui retourne l'unique matrice échelonnée réduire équivalente par lignes à `M0`.

```
1  def echelonnee_reduite(M0):
2      M=M0.copy() # pour ne pas modifier la matrice passée en argument
3
4      ... # à remplir
5
6      return(M)
```

   La question posée revient à implémenter l'algorithme de Gauß-Jordan en Python. Les fonctions construites dans les exercices 1–4 peuvent, bien sûr, être réutilisées...

2. Tester la fonction `echelonnee_reduite` construite (cf. exemples listés ci-dessous).

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \underset{L}{\sim} \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 & 2 & 3 \\ 2 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix} \underset{L}{\sim} \begin{pmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 4 \\ 4 & 4 & 4 & 5 \end{pmatrix} \underset{L}{\sim} \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

1. L'algorithme naïf précédemment proposé est corrigé (cf. possibilité que certaines colonnes se terminent par une succession de 0), grâce à l'introduction d'une variable nommée index_last_pivot. De plus, nous n'avons pas séparé en trois phases (descente, stagnation, remontée) comme en mathématiques, afin d'optimiser le coût de l'algorithme.