TP 8 : matrices

Matrices avec Numpy

Pour manipuler des matrices en Python, nous aurons besoin d'outils supplémentaires afin de gagner du temps. Une première possibilité serait de créer nous-mêmes des nouveaux objets Python. Une matrice serait codée sous la forme d'une liste de listes :

```
[[1,2,3,4],[-6,1,0,2]]
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ -6 & 1 & 0 & 2 \end{pmatrix}$$

L'objet Python ci-dessus à gauche est une liste contenant deux éléments : chaque élément est lui-même une liste contenant quatre éléments. Cet objets représenterait donc une matrice de taille 2×4, plus précisément la matrice représentée au dessus à droite. Afin d'éviter les problèmes de recopie de listes, nous allons utiliser un **module** Python qui gère directement les matrices, avec la syntaxe décrite ci-dessus. Dans tous vos scripts par la suite (ou bien dans l'interpréteur), tapez la commande suivante

```
import numpy as np
```

Pour créer une nouvelle matrice, on utilise la commande `np.array(.....)` et on donne en argument une liste de listes de tailles raisonnables pour pouvoir en faire une matrice. Voici quelques exemples à taper dans l'interpréteur pour observer et comprendre les affichages obtenus.

```
>>> A = np.array([[5, -1, 2], [1, 2, 3]])
>>> A
>>> B = np.array([[1, -1], [1, 3]])
>>> B
>>> A.shape
>>> B.shape
>>> A.size
>>> B.size
>>> p,n = A.shape
>>> A.transpose()
>>> (n,p) == A.transpose().shape
>>> C = np.array([[-1, 0, 1, 3]])
>>> C
>>> C.transpose()
>>> B * A  # tentative de produit
>>> np.dot(B, A)  # prod. matriciel
>>> np.dot(A, B) # produit impossible
```

```
>>> np.dot(A, C)
>>> np.dot(C.transpose(), C)
>>> np.dot(C, C.transpose())
>>> A + 3  # ajout terme a terme
>>> 2 + A
>>> 2 * A  # multipl. terme a terme
>>> A * 4
>>> 1 / A   # inverse terme a terme
>>> A[1,2]  # lecture d'un element
>>> A[0,1]
>>> A[2,3]  # attention tailles !
>>> A[1,1] = 7  # modif. 1 element
>>> A
>>> np.zeros(10)
>>> np.zeros([10, 15])
>>> np.ones([6, 3])
>>> np.eye(7)
```

Exercice 1.

1. Codez une fonction Python qui prend en argument une matrice A et qui renvoie la trace de A, c'est-à-dire la somme des éléments diagonaux de A. Testez votre fonction sur les matrices A, B et C précédentes. **Vérifiez votre résultat.**
2. Codez une fonction Python qui prend en argument une matrice 2×2 et qui renvoie son déterminant c'est-à-dire le réel $ad - bc$ si la matrice est de la forme $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

   Optionnel : codez la fonction précédente à l'aide uniquement de deux lignes de code (dont la déclaration et le return).
3. À l'aide du module `random` et de la fonction `random()`, remplissez aléatoirement une matrice de taille 100×100. Créez une fonction Python nommée `val_moy` sans argument (son code commencera donc par `def val_moy():`) qui créée une telle matrice aléatoire A puis renvoie la moyenne des éléments de $A^t A$. Exécutez plusieurs fois cette fonction.

Algèbre linéaire avec linalg

Le module `numpy` contient des fonctions d'algèbre linéaire qui permettent par exemple de calculer l'inverse d'une matrice (lorsque c'est possible). Essayez par exemple avec votre matrice B de la page précédente

```
>>> invB = np.linalg.inv(B)
>>> invB
>>> np.dot(B,invB)
```

Exercice 2.

1. En fait pour les matrices de taille 2×2 on dispose directement d'une formule pour trouver l'inverse.

   La matrice $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ est inversible si et seulement si $ad - bc$ est non nul son inverse est alors
   $$\frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}.$$
   Comparez avec le calcul de l'ordinateur dans le cas de la matrice B de la page précédente.
2. Générez une matrice aléatoire D de taille 100×100 et calculez son inverse D^{-1} à l'aide de numpy. Calculez alors la valeur moyenne des éléments de $D^{-1}\,{}^t(D^{-1})$.

Le module `numpy.linalg` permet aussi de résoudre des systèmes linéaires. Par exemple pour résoudre le système linéaire $BX = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$, où X est une matrice inconnue de $\mathcal{M}_{3,1}(\mathbb{R})$ et B la matrice de la page précédente, on utilise

```
>>> np.linalg.solve(B,np.array([1,2]))
```

Attention cette méthode ne fonctionne que pour des systèmes de Cramer !
Essayez en remplaçant B par la matrice A.
Essayez aussi en remplaçant B par la matrice $\begin{pmatrix} 1 & 2 \\ -2 & -4 \end{pmatrix}$.

Pivot de Gauss en Python

On étudie les matrices
$$A = \begin{pmatrix} 0 & 2 & 3 \\ -1 & 0 & 3 \\ 1 & -2 & 4 \end{pmatrix} \qquad \text{et} \qquad B = \begin{pmatrix} 0 & 2 & -7 \\ -1 & 0 & 3 \\ 1 & -2 & 4 \end{pmatrix}$$
Faisons une succession d'opérations sur les lignes, comme dans le pivot de Gauss. Dans un cas on trouve
$$A \underset{L_3 \leftrightarrow L_1}{\sim} \begin{pmatrix} 1 & -2 & 4 \\ -1 & 0 & 3 \\ 0 & 2 & 3 \end{pmatrix} \underset{L_2 \leftarrow L_2 + L_1}{\sim} \begin{pmatrix} 1 & -2 & 4 \\ 0 & -2 & 7 \\ 0 & 2 & 3 \end{pmatrix} \underset{L_3 \leftarrow L_3 + L_2}{\sim} \begin{pmatrix} 1 & -2 & 4 \\ 0 & -2 & 7 \\ 0 & 0 & 10 \end{pmatrix}$$
Dans l'autre cas on trouve avec la même suite d'opérations
$$B \underset{L_3 \leftrightarrow L_1}{\sim} \begin{pmatrix} 1 & -2 & 4 \\ -1 & 0 & 3 \\ 0 & 2 & -7 \end{pmatrix} \underset{L_2 \leftarrow L_2 + L_1}{\sim} \begin{pmatrix} 1 & -2 & 4 \\ 0 & -2 & 7 \\ 0 & 2 & -7 \end{pmatrix} \underset{L_3 \leftarrow L_3 + L_2}{\sim} \begin{pmatrix} 1 & -2 & 4 \\ 0 & -2 & 7 \\ 0 & 0 & 0 \end{pmatrix}$$
Vous aurez (probablement mais pas obligatoirement) besoin des commandes suivantes :

```
import numpy as np, A.shape, np.dot(A,B), np.array([[...],...,[...]]), np.zeros([p,n]),
np.eye(n), A[i,j], A[1,2], A[:,2], A[1:,2:], A[1:3,2:4],

et bien sur les desormais classiques :

len(liste),  if ... : , [.... for k in .... if .....], for k in range(....): , while .... :
```

Matrices de transvection, dilatation, permutation

On considère une matrice A de taille $p \times n$.
On cherche à coder un algorithme de pivot de Gauss qui met la matrice A sous forme échelonnée.

— On note $D_i(\lambda)$ la matrice $p \times p$ diagonale qui n'a que des 1 sur sa diagonale sauf en position i, où le coefficient est λ.
— On note $T_{i,j}(\lambda)$ la matrice de taille $p \times p$ qui ne contient que des 1 sur sa diagonale, des 0 ailleurs sauf en position i, j où le coefficient vaut λ.
— On note $P_{i,j}$ la matrice de taille $p \times p$ qui ne contient que des 1 sur sa diagonale et des 0 ailleurs, sauf aux positions i, i et j, j où le coefficient est alors 0, et aux positions i, j et j, i où le coefficient est 1.

Exercice 3 (sur papier)**.**

1. On peut écrire $P_{i,j}$ comme produit des matrices $T_{i,j}(1)$, $T_{i,j}(-1)$ et $D_i(-1)$. Comment ?
2. Donner une condition nécessaire et suffisante pour que $D_i(\lambda)$ soit inversible.
   Donner l'inverse de $D_i(\lambda)$ dans ce cas.
3. Donner l'inverse de $T_{i,j}(\lambda)$.

Exercice 4.

1. Codez une fonction Python qui prend en argument un entier p, un entier i et un réel a et qui renvoie la matrice $D_i(a)$ de taille $p \times p$.
2. Codez une fonction Python qui prend en argument un entier p, deux entiers i et j et qui renvoie la matrice $P_{i,j}$ de taille $p \times p$.
3. Codez une fonction Python qui prend en argument un entier p, deux entiers i et j et un réel a et qui renvoie la matrice $T_{i,j}(a)$ de taille $p \times p$.

Exercice 5 (Échauffement)**.**

On considère la matrice $A = \begin{pmatrix} 0 & -1 & 1 & 2 \\ 1 & 2 & 3 & 0 \\ -1 & 1 & 2 & 0 \end{pmatrix}$.

En utilisant vos fonctions codées à l'exercice précédent, faites subir à A une succession de multiplications à gauche de sorte qu'elle devienne échelonnée.

Algorithme du pivot de Gauss

Exercice 6 (Colonnes nulles)**.**

Écrire une fonction Python `colonne_nulle` qui prend argument une matrice A et un entier j et qui renvoie `True` si la colonne j de A est nulle, et `False` sinon.

Exercice 7 (placement du pivot)**.**

On va coder une fonction Python qui prend en argument une matrice A dont la première colonne est supposée non nulle, et qui met en place un pivot non nul en position $1, 1$ (donc $0, 0$ pour Python !) afin de démarrer les éliminations du pivot de Gauss.

L'algorithme est le suivant :

Entrée : matrice A de taille $p \times n$.
Algorithme :
— On cherche dans la colonne 1 de A le premier coefficient non nul (en partant du haut), on note k le numéro de la ligne correspondante
— On permute la ligne 1 et la ligne k de A à l'aide d'une matrice de permutation $P_{1,k}$.
— **On renvoie** A.

Codez une fonction Python nommée `pivot_en_place` qui réalise ces opérations.

Exercice 8 (première étape du pivot)**.**

Entrée : matrice A de taille $p \times n$.
Algorithme :
Pour k allant de 2 à p :
— On créée D une matrice de dilatation et T une matrice de transvection qui permettront de faire l'opération éliminant le coefficient en position $k, 1$ sans faire apparaître de fractions non nécessaires.
— On remplace A par TDA.
Sortie : la matrice A.

Codez une fonction Python nommée `elimination` qui réalise ces opérations.

Exercice 9 (le pivot de Gauss)**.**

Pour finalement coder le pivot de Gauss, nous allons utiliser un algorithme récursif :
— **Cas de base de la récursion :** si $p = 1$ on se contente de renvoyer A.
— **Cas général :** On part de la matrice A de taille $p \times n$, on cherche la première colonne non-nulle.
— On retire les colonnes nulles pour se concentrer sur les autres colonnes. On note B la matrice obtenue.
— On place le pivot sur la matrice B
— On fait les éliminations sur la matrice B.
— On créée une sous-matrice C obtenue en retirant la première ligne et la première colonne de B.
— On applique récursivement le pivot sur C.
— On complète C en lui ajoutant la première colonne de B et la première ligne de B.
— On complète encore C en ajoutant les colonnes nulles de A qu'on avait retiré.
— On renvoie C qui est maintenant de taille $p \times n$ et échelonnée.