TP 15
Matrices

L'objectif de ce TP est assez simple à résumer : nous apprenons à représenter des matrices et à programmer les opérations usuelles dessus.
Nous avons déjà vu que la bibliothèque `numpy` permettait de traiter des tableaux de toute sorte et en particulier des matrices. En fait, toutes les fonctions que nous allons écrire ici se trouvent déjà intégrées dans `numpy` ainsi que dans son sous-module `numpy.linalg`, et il faudra continuer à apprendre à les utiliser, voir par exemple le Mémo Agro-véto. Cependant pour notre apprentissage actuel nous allons prendre une autre approche et nous allons programmer toutes ces fonctions sans autre pré-requis que les listes Python.

I Représentation des matrices

I.1 Définir des matrices

Les matrices en Python seront enregistrées comme des **listes de listes** et plus précisément comme la **liste de leurs lignes**. Par exemple la matrice

$$A = \begin{pmatrix} 3 & 5 & -7 \\ 0 & -1 & 2 \end{pmatrix} \tag{1}$$

sera représentée en Python par

```
A = [[3, 5, -7], [0, -1, 2]]
```

Cela est en quelque sorte une convention : on pourrait aussi décider de travailler en donnant la liste des colonnes. Mais cela est bien pratique ! En effet dans notre exemple `A[0]` désigne en fait le premier élément de la liste, donc la liste `[3, 5, -7]`, ainsi `A[0][0]` (c'est la même chose que s'il y avait des parenthèses : `(A[0])[0]`) est bien l'élément `3`, et `A[0][1]` est donc l'élément `5` etc. De même `A[1]` désigne en fait la deuxième ligne `[0, -1, 2]` et donc `A[1][0]` est `0`, `A[1][1]` est `-1`, `A[1][2]` est `2`. Ainsi le coefficient d'indice (i, j) est `A[i][j]` mais attention, contrairement à la convention mathématique, les indices partent de 0 !
On rappelle que `len(A)` donne la longueur de `A` vue comme une liste simple, ce qui correspond ici au nombre de lignes ; et donc `len(A[0])` donnera la longueur de la première liste à l'intérieur de `A` et c'est le nombre de colonnes (on suppose que les matrices sont bien formées : toutes les colonnes ont la même longueur). On **donne** donc la fonction suivante qui sera pratique pour la suite pour la lisibilité :

```
def taille(A):
    n = len(A)
    p = len(A[0])
    return (n, p)
```

On pourra donc écrire des fonctions qui commencent par `(n, p) = taille(A)` pour récupérer le tuple des dimensions de `A` et donc le premier indice, qu'on appellera souvent i, sera l'indice des lignes et variera de 0 à $n-1$ (attention, cela diffère de la convention mathématiques ! mais pose peu de problèmes en pratique) et le deuxième indice, qu'on appellera souvent j, sera celui des colonnes et variera entre 0 et $p-1$.
Il faut aussi savoir faire sans cette fonction ; celle du fichier contient en plus une petite vérification rapide que toutes les colonnes sont bien de même taille, ce qui permettra surtout de détecter des bugs et des incohérences.

I.2 Créer des nouvelles matrices

Nous aurons aussi besoin de pouvoir créer des nouvelles matrices de taille donnée. Pour créer une nouvelle matrice remplie de zéros, par exemple à 2 lignes et 3 colonnes, nous pourrions être tentés d'écrire

```
A = [[0] * 3] * 2
```

mais cela pose un petit problème (testez)...

```
>>> A[0][0] = 1
>>> print(A)
```

En fait, la syntaxe ci-dessus crée bien une ligne `[0] * 3` de zéros, puis ne recopie pas la ligne mais uniquement la **référence** à cette ligne (les deux lignes de `A` deviennent des références à la **même** liste `[0, 0, 0]`) ainsi toute modification sur une ligne provoque une modification sur l'autre ligne. L'auteur de ce présent TP s'est lui-même fait piéger lors de son apprentissage de Python.
Pour créer une nouvelle matrice toute fraiche (sans copie de références) on utilisera plutôt la syntaxe suivante :

```
A = [[0 for _ in range(3)] for _ in range(2)]
```

Ici les 6 coefficients sont bien indépendants, essayez par exemple de modifier `A[0][0]`. En fait, on **donne** la fonction suivante qui crée une matrice nulle de taille (n, p) :

```
def matrice_nulle(n, p):
    return [[0 for _ in range(p)] for _ in range(n)]
```

De même, on fera attention à ce qu'écrire `B = A` ne crée pas une copie de `A` mais copie seulement le référence, et toutes les modifications de `A` vont alors affecter `B`. Pour écrire nos fonctions ce n'est pas le comportement voulu, donc nous commençons toujours par créer une nouvelle matrice toute fraiche (c'est à dire, sans références) que nous remplissons au fur et à mesure.
L'outil essentiel pour parcourir une matrice et effectuer une opération sur chaque coefficient est, bien entendu la double boucle. Une syntaxe telle que

```
for i in range(n):
    for j in range(p):
        ... A[i][j] ...
```

va parcourir uns par uns tous les coefficients en parcourant, pour chaque ligne, toutes les colonnes. Si on échange les deux boucles on parcourt, pour chaque colonne, toutes les lignes. Cet ordre aura peu d'importance dans les fonctions de la partie II car les opérations doivent de toute façon être effectuées sur tous les coefficients.

I.3 Autres fonctions utiles

Pour faire des tests, on a besoin de matrices, mais il ne faut pas donner d'exemples trop faciles. On donne donc une fonction `matrice_hasard(n, p)` qui génère une matrice avec des coefficients aléatoires de taille (n, p).
Pour afficher le résultat de façon plus conviviale qu'une liste de lignes, on pourra regarder et tester les trois fonctions `matrice_print` qui sont données aussi.

II Exercices

Les fonctions sont à compléter dans le fichier ci-joint. Elles commencent par récupérer la taille des matrices données en argument, éventuellement vérifier la compatibilité des tailles pour les opérations à effectuer. Puis elles génèrent une **nouvelle** matrice pour contenir le résultat.

II.1 Créer des matrices

Exercice 1. Écrire la fonction `identite(n)` qui crée la matrice identité de taille n.

Exercice 2. Écrire la fonction `diagonale(L)` qui prend en argument une liste (simple) de coefficients et qui crée une matrice diagonale, avec ces coefficients sur la diagonale.

II.2 Opérations sur les matrices

Exercice 3. Écrire la fonction `somme(A, B)` qui calcule la somme des matrices `A` et `B`.

Exercice 4. Écrire la fonction `produit_constante(A, x)` qui calcule la matrice xA (où x est un nombre réel et A est une matrice).

Exercice 5. Écrire la fonction `coeff_produit(A, B, i, j)` qui calcule le coefficient d'indice (i, j) du produit de matrices AB.

Exercice 6. Écrire la fonction `produit(A, B)` qui calcule le produit de matrices AB.

Exercice 7. Écrire la fonction `puissance(A, N)` qui calcule la puissance A^N (où N est un entier positif). On rappelle que A^0 est la matrice identité (de la même taille que A) et que $A^1 = A$. Bien entendu, on pourra choisir entre une méthode itérative et une méthode récursive...

Exercice 8. Écrire la fonction `transpose(A)` qui retourne la transposée de la matrice A.

Définition 1. Soit $A \in \mathcal{M}_{n,p}(\mathbb{R})$. La **transposée** de A est la matrice notée ${}^tA$ obtenue à partir de A par symétrie par rapport à la diagonale. C'est donc la matrice ${}^tA \in \mathcal{M}_{p,n}(\mathbb{R})$ avec $({}^tA)_{i,j} = A_{j,i}$.

Exemple 1.

$$A = \begin{pmatrix} 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix} \qquad {}^tA = \begin{pmatrix} 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{pmatrix} \tag{2}$$

De plus la transposée d'une matrice triangulaire supérieure est triangulaire inférieure, et réciproquement...

Remarque 1. Toutes les fonctions ci-dessus peuvent être écrites sans boucles mais avec des listes en compréhension et un peu de magie Python ! Voir la correction.

II.3 Quelques tests

Exercice 9. Écrire une fonction `est_diagonale(A)` qui teste si la matrice A est diagonale.

Exercice 10. Écrire une fonction `est_triangulaire_sup(A)` qui teste si la matrice A est triangulaire supérieure.

III Pour aller plus loin

Le but ultime bien entendu est de pouvoir programmer l'inversion de matrices, et donc l'algorithme du pivot de Gauss. Pour cela il faut d'abord s'occuper des opérations élémentaires. Calculer l'inverse d'une matrice A revient à effectuer simultanément les opérations élémentaires sur A et sur la matrice identité I_n : d'abord échelonner, puis remonter le système en utilisant encore des opérations élémentaires.
Les deux fonctions suivantes ne sont en fait pas absolument nécessaires : on écrira une fonction pour échelonner, et une fonction pour inverser qui échouera si la matrice n'est pas inversible.

Exercice 11. Écrire une fonction `est_echelonnee(A)` qui teste si la matrice A est échelonnée.
Une proposition est la suivante : il faut bien entendu une boucle (simple) et garder trace de deux indices i et j, au départ initialisés à 0. On pose x le coefficient $A_{i,j}$. Si x est non-nul, alors ce doit être un pivot : il ne faut pas qu'en dessous de lui il y ait des coefficients non-nuls (cela signifie « comme d'habitude » qu'on passe par la condition inverse : c'est dans le cas contraire qu'on sait qu'on peut retourner `False`), et ensuite il faudra passer à la colonne et à la ligne suivante car c'est à partir de là qu'on continuera à chercher les pivots. Mais si x est nul alors il faut, encore, que toute la colonne en dessous de x soit nulle puis passer à la colonne suivante mais en gardant la même ligne.

Exercice 12. Écrire une fonction `rang_echelonnee(A)` qui calcule le rang de A quand A est échelonnée.
Si la fonction précédent est écrite correctement, elle contient une variable i qui compte le numéro de ligne actuel où on cherche les pivots et donc qui à la fin est égal au rang, donc il suffit de reprendre exactement la fonction précédente mais de lui faire renvoyer i.

On s'intéresse maintenant à la procédure d'échelonnage et pour cela on a besoin des opérations élémentaires.

Exercice 13. Écrire des fonctions correspondant aux 3 opérations élémentaires sur les matrices. Attention, cette fois on veut que les opérations modifient directement la matrice A passée en argument, il faut donc faire attention en échangeant des lignes.

1. `echange(A, i, j)` qui échange les lignes i et j de A ($L_i \leftrightarrow L_j$),
2. `dilate(A, i, x)` qui dilate la ligne i d'un facteur x ($L_i \leftarrow xL_i$),
3. `transvection(A, i, j, x)` qui correspond à l'opération $L_i \leftarrow L_i + xL_j$.
4. Pour éviter de faire apparaitre des flottants dans la première étape, on pourra trouver plus pratique d'écrire la fonction `transvection2(A, i, j, x, y)` qui correspond à l'opération $L_i \leftarrow xL_i - yL_j$.

Exercice 14. Écrire une fonction `cherche_pivot(A, i, j)` qui cherche l'indice ligne d'un pivot (c'est à dire, cherche un coefficient non nul) dans la colonne j et dans les lignes d'indice $\geqslant i$. Si elle en trouve un, elle renvoie l'indice de la ligne du pivot trouvé. Sinon, elle renvoie -1. En utilisant cette fonction, il sera alors important de tester si elle a bien trouvé un pivot ou non.

La fonction suivante ne sera pas utile directement dans le calcul de l'inverse, mais est tout de même une étape importante.

Exercice 15. Écrire une fonction `echelonne(A)` qui échelonne la matrice A. Plus précisément, il y a plusieurs étapes :

1. D'abord on ne veut pas que la fonction modifie A, il faut donc faire une copie de A. Pour cela on pourra écrire une ligne

```
B = [[A[i][j] for j in range(p)] for i in range(n)]
```

2. On itère sur l'indice j (la colonne), on cherche un pivot...
3. ... cela nécessitait donc d'initialiser une variable i pour les lignes, cependant contrairement à j la variable i n'augmente pas à chaque passage de boucle mais seulement quand un pivot a été trouvé.
4. Si le pivot est trouvé, on effectue les opérations élémentaires « comme d'habitude ». Dans la fonction pour échelonner on peut se passer entièrement de divisions (et donc de nombres à virgule flottante) en utilisant la fonction `transvection2`.
5. Sinon, on passe simplement à la suite. Là, l'indice i ne doit pas augmenter.
6. La fonction renvoie la matrice B qui, si tout s'est bien passé, est échelonnée.

Si vous avez correctement écrit la fonction `rang_echelonnee` de l'exercice 12, il est alors rapide d'écrire une fonction `rang(A)` qui calcule le rang de A quelque soit la matrice A.
Enfin on en vient à la fonction finale pour inverser une matrice.

Exercice 16. Écrire une fonction `inverse(A)` qui renvoie l'inverse de la matrice A, dans le cas où A est carrée et inversible. On s'y prend de la façon suivante :

1. Comme précédemment, on initialise une matrice B qui est une copie de A sur laquelle on va travailler, mais aussi une matrice I qui au départ est l'identité.
2. **Toutes** les opérations élémentaires sont effectuées en même temps sur B et sur I. À la fin B doit être transformée en identité et I sera la matrice inverse de A.
3. Dans un premier temps on échelonne exactement comme tout à l'heure. Si tout se passe bien, à la fin, la matrice B est échelonnée et le dernier coefficient en bas à droite est non nul. Sinon on arrête là, la matrice n'est pas inversible. En fait même en cours de route, si on ne trouve pas de pivot à chaque étape, c'est que la matrice n'est pas inversible. En même temps à la fin, la matrice I est triangulaire inférieure.
4. Ensuite il faut remonter, en partant d'en bas. On a donc un deuxième morceau de la fonction, avec une boucle qui s'exécute après la toute première partie. Il suffit de boucler sur l'indice ligne, et de faire des opérations élémentaires.
5. À la fin il faut renvoyer la matrice I.