TP 15
Matrices

L'objectif de ce TP est tout simplement d'apprendre à manipuler des matrices en Python, à la fois les représenter et programmer quelques 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 Préliminaires

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

$$A = \begin{pmatrix} 8 & -1 & 7 & 4 \\ 6 & -2 & 5 & -3 \\ 7 & 2 & 0 & 7 \end{pmatrix} \tag{1}$$

sera représentée en Python par

```
A = [[8, -1, 7, 4], [6, -2, 5, -3], [7, 2, 0, 7]]
```

Cela est en quelque sorte une convention : on pourrait très bien 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 (de listes), donc la liste `[8, -1, 7, 4]`, et ainsi `A[0][0]` (c'est la même chose que s'il y avait des parenthèses : `(A[0])[0]`) est l'élément `8`, et `A[0][1]` est donc l'élément `-1` etc. De même `A[1]` est toute la deuxième ligne `[6, -2, 5, -3]` et donc `A[1][0] = 6`, `A[1][1] = -2`, `A[1][2] = 5`. Ainsi le coefficient d'indice (i, j) est `A[i][j]` à condition de, contrairement à la convention mathématique, numéroter les indices à partir de 0 !

Exercice 1. À partir de la fonction `len()`, comment obtient-on le nombre de lignes et de colonnes de la matrice `A` ? Écrire la fonction `taille(A)` qui renvoie un couple formé du nombre de lignes (toujours noté n) et du nombre de colonnes (toujours p).

Pour utiliser la fonction précédente, on pourra écrire des fonctions qui commencent par `(n, p) = taille(A)` ce qui récupère le tuple des dimensions de `A`. Le premier indice, qu'on appellera souvent i, sera l'indice des lignes et variera de 0 à $n-1$ (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 $p-1$.

I.2 Créer des nouvelles matrices

Nous aurons besoin de pouvoir créer des nouvelles matrices de taille donnée, et en particulier d'avoir une fonction `matrice_nulle(n, p)` qui crée une nouvelle matrice à n lignes et p colonnes remplie de zéros.
L'idée la plus simple pour créer par exemple une nouvelle matrice à 3 lignes et 4 colonnes serait d'écrire

```
>>> A = [[0] * 4] * 3
>>> print(A)
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

ainsi les listes à l'intérieur sont de taille 4 remplies de zéros, et on les répète 3 fois.
Malheureusement, cela pose un petit problème...

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

En fait, la syntaxe ci-dessus crée bien une ligne `[0] * 4` de zéros, puis ne recopie pas la ligne mais uniquement la **référence** à cette ligne. Les trois lignes de `A` deviennent des références à la **même** liste `[0, 0, 0, 0]`, ainsi toute modification sur une ligne provoque une modification sur l'autre ligne. L'auteur de ce TP s'est lui-même fait piéger lors de son apprentissage de Python.
La syntaxe des listes en compréhension, elle, permet toujours de créer des nouvelles listes. On utilisera donc la syntaxe

```
>>> A = [[0 for _ in range(4)] for _ in range(3)]
```

Ici, on crée une liste de quatre zéros (on dira « à l'intérieur »), et on répète cela trois fois. Les coefficients obtenus sont tous indépendants. Plus généralement, on **donne** la fonction suivante qui crée une matrice nulle de dimensions (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 la référence, et toutes les modifications de `B` vont alors affecter `A`. Pour écrire nos fonctions ce n'est pas le comportement voulu, donc nous commençons toujours par créer une nouvelle matrice nulle toute fraiche (c'est à dire, sans références) que nous remplissons au fur et à mesure.
Enfin, rappelons que l'outil essentiel pour parcourir une matrice et effectuer une opération sur chaque coefficient est la double boucle. Une syntaxe telle que

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

va parcourir les lignes et, pour chaque ligne, parcourir les colonnes. Si on échange les deux boucles on parcourt, pour chaque colonne, toutes les lignes. Dans la plupart des fonctions de la partie II cet ordre n'aura pas d'importance car les opérations doivent de toute façon être effectuées sur tous les coefficients.

II Exercices

Toutes 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 créent une **nouvelle** matrice pour contenir le résultat, et la remplissent peu à peu.

II.1 Créer des matrices

Exercice 2. Écrire la fonction `identité(n)` qui crée la matrice identité de taille n.

Exercice 3. Écrire la fonction `diagonale(L)` qui prend en argument une liste (simple) de coefficients et qui crée une matrice diagonale, en plaçant les coefficients donnés dans `L` sur la diagonale.

II.2 Opérations sur les matrices

Exercice 4. Écrire la fonction `somme(A, B)` qui calcule la somme des matrices `A` et `B`.

Exercice 5. Écrire la fonction `produit_constante(A, a)` qui calcule la matrice aA (où a est un nombre réel et A est une matrice).

Exercice 6. Écrire la fonction `coeff_produit(A, B, i, j)` qui calcule le coefficient d'indice (i, j) du produit de matrices AB.

Exercice 7. Écrire la fonction `produit(A, B)` qui calcule le produit de matrices AB.

Remarque 1. Pour le produit de matrices il n'est pas absolument nécessaire d'écrire la fonction annexe `coeff_produit`. On peut s'en sortir avec une *triple* boucle, en exprimant que pour un triplet (i, k, j) le produit $A_{i,k} \times B_{k,j}$ doit se sommer dans le coefficient (i, j) du produit.

Exercice 8. 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). On pourra choisir entre une méthode itérative et une méthode récursive...

Remarque 2. Ici plus encore, l'algorithme des puissances rapides (TP 10 : récursivité, exercice 5) est particulièrement important ; on rappelle que celui-ci consiste à écrire $A^N = \left(A^{N/2}\right)^2$ si N est pair et $A^N = AA^{N-1}$ sinon, par exemple $A^8 = ((A^2)^2)^2$ se calcule avec seulement 3 multiplications de matrices. En effet un produit de matrices est toujours une opération lourde, nécessitant beaucoup de calculs de produits de coefficients entre eux puis de sommes, et les matrices utilisées pour modéliser finement des phénomènes physiques peuvent avoir des milliers de coefficients. Il est donc crucial de calculer des puissances en minimisant le nombre d'opérations de produits de matrices qu'on va effectuer.

II.3 Quelques tests

Exercice 9. Écrire une fonction `est_diagonale(A)` qui renvoie `True` si la matrice A est diagonale, et `False` sinon.

Exercice 10. Écrire une fonction `est_triangulaire_supérieure(A)` qui renvoie `True` si la matrice A est triangulaire supérieure, et `False` sinon.

III Le pivot de Gauss

L'objectif ultime serait d'écrire un programme capable de calculer l'inverse d'une matrice, ou au moins au départ d'échelonner une matrice en appliquant l'algorithme du pivot de Gauss.
Cela nécessite tout d'abord de programmer les opérations élémentaires. Nous allons écrire directement le minimum dont nous avons besoin. Pour éviter de faire apparaitre des nombres à virgule flottante, on essaie au maximum de travailler avec des nombres entiers et sans division pour l'instant (rappelons que le calcul `6 / 2` donne `3.0` et fait donc passer en virgule flottante, avec les erreurs d'arrondis qui peuvent se cumuler). Pour pouvoir ré-utiliser facilement les fonctions, cette fois nous avons besoin qu'elles modifient la matrice passée en argument au lieu de créer une copie nouvelle.

Exercice 11. Écrire les fonctions suivantes, prenant en argument une matrice `A` et **qui modifient directement la matrice** (sans en créer une nouvelle) :

1. `echange(A, i, j)` : échange les lignes i et j de A (opération $L_i \leftrightarrow L_j$).
2. `combinaison(A, a, i, b, j)` : opération $L_i \leftarrow aL_i + bL_j$.
3. `dilate(A, a, i)` : opération $L_i \leftarrow aL_i$.

Avant de passer à l'échelonnage, il reste une fonction manquante qu'on écrira à part : celle pour l'étape de recherche d'un pivot. On doit lui fixer des indices (i, j) pour qu'elle cherche un coefficient non-nul *dans la colonne et en dessous à partir* de ce coefficient. Si elle en trouve, alors on échangera toute la ligne pour la placer à l'indice i ; si elle n'en trouve pas, il faudra seulement passer à la colonne suivante. Il est donc important que la fonction renvoie l'objet spécial `None` si elle ne trouve pas de pivot **et** que celui qui utilise la fonction teste si le résultat est `None` ou sinon est un indice de ligne.

Exercice 12. Écrire une fonction `cherche_pivot(A, i, j)` qui cherche l'indice ligne d'un coefficient non-nul dans la colonne j et dans les lignes d'indice $\geqslant i$ de A. Si elle en trouve un, elle renvoie l'indice de la ligne du pivot trouvé. Sinon, elle renvoie `None`.

Tout est prêt pour appliquer le pivot de Gauss !

Exercice 13. Écrire une fonction `echelonne(A)` qui échelonne la matrice A.
Pour que la fonction ne modifie pas la matrice A, on pourra commencer par travailler une copie, en copiant tous les coefficients uns par uns. De plus on ne veut pas utiliser l'opération élémentaire de dilatation, et on veut utiliser la combinaison $L_i \leftarrow aL_i + bL_j$ pour annuler un coefficient sans utiliser de division.
Il faut bien sûr une boucle en maintenant des indices lignes et colonnes, mais qui ne jouent pas le même rôle rôle car après avoir éventuellement trouvé un pivot on passera toujours à la colonne suivante, par contre on ne

descend pas toujours d'une ligne (seulement si on a bien pu trouver le pivot). On utilise donc une variable j pour les colonnes dans une boucle principale `for`, mais une variable `r` pour les lignes qui augmente de 1 seulement dans le cas où on a trouvé un pivot avec la fonction précédente. Une fois le pivot trouvé, on effectue les opérations élémentaires « comme d'habitude » pour échelonner.
À la fin si tout se passe bien la variable `r` contient en fait le rang de la matrice. On pourra, en plus de renvoyer la matrice échelonnée, utiliser `print` pour afficher le rang.

Enfin on en vient à la fonction finale pour inverser une matrice.

Exercice 14. 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. Au départ on initialise deux matrices : B qui sera une copie de A (pour ne pas modifier A) 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 (on copie-colle le code de la fonction précédente, mais les opérations se font sur B et sur I). Si tout se passe bien, à la fin, la matrice B est échelonnée et le dernier coefficient en bas à droite est non-nul, le rang est égal au nombre de lignes et de colonnes. Si ce n'est pas le cas on peut éventuellement s'arrêter là et renvoyer une erreur, 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. On utilise des transvections pour éliminer toute la colonne au-dessus du coefficient diagonal de B, partant d'en bas à droite, puis on le met à 1 avec une dilatation (simultanément sur B et I, toujours). Cela est en fait plus simple que l'échelonnage car on boucle sur un seul indice et sans avoir à chercher de pivot.
5. À la fin il faut renvoyer la matrice I.