TP 15
Matrices

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

Nous avons déjà vu que la bibliothèque Numpy permettait de traiter des tableaux de toute sorte, et s'applique donc aux 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. 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 que nous allons manipuler en Python seront enregistrées comme des **listes de listes** et plus précisément comme la **liste de leurs lignes**. Par exemple la matrice

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

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 mathématique 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 « fraiches » (sans dépendance entre les éléments). On utilisera donc la syntaxe

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

qui crée une liste de quatre zéros (on dira « à l'intérieur »), puis répète cela trois fois. 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 en général pas le comportement voulu, donc nous commençons toujours par créer une nouvelle matrice nulle toute fraiche 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 un par un est la double boucle, une syntaxe telle que :

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

Plus précisément, ici elle parcourt les lignes et, pour chaque ligne, parcourt les colonnes. Si on échange l'ordre des deux boucles, alors on parcourt les colonnes et, pour chaque colonne, toutes les lignes. Dans la plupart des fonctions de la partie II cet ordre de parcours n'a pas d'importance car de toute façon il faut effectuer les opérations 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.

Par exemple, on veut que l'appel `diagonale([3, 5, 7])` produise la matrice $\begin{pmatrix} 3 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 7 \end{pmatrix}$.

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
1. Écrire la fonction `coeff_produit(A, B, i, j)` qui calcule le coefficient d'indice (i, j) du produit de matrices AB.
2. En déduire la fonction `produit(A, B)` qui calcule le produit de matrices AB.
3. Bonus : pouvez-vous écrire directement la fonction `produit`, sans fonction intermédiaire ?

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 $A^N = A \cdot A^{n-1}$. On pourra choisir entre une méthode itérative et une méthode récursive…

Remarque. 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 $A^8 = \left(\left(A^2\right)^2\right)^2$ 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 8
Écrire une fonction `est_diagonale(A)` qui renvoie `True` si la matrice A est diagonale, et `False` sinon.

Exercice 9
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 du moins dans un premier temps d'échelonner une matrice en appliquant l'algorithme du pivot de Gauss. On obtiendra le rang au passage.

Cela nécessite tout d'abord de programmer les opérations élémentaires. 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 nouvelle copie.

Exercice 10
Écrire les fonctions suivantes, prenant en argument une matrice `A` et **qui modifient directement la matrice** (sans en créer une nouvelle) :

1. `échange(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. En effet, quand on échelonne une matrice, la seule opération qui n'est pas complètement automatique est celle où on choisit une ligne qu'on va échanger avec la ligne sur laquelle on travaille pour éliminer une variable x_j. À la main, on choisit une ligne contenant devant x_j un coefficient sympathique (c'est-à-dire 1). En général, la seule contrainte importante est que ce coefficient soit non-nul, sinon on ne peut pas l'utiliser pour éliminer x_j dans les autres lignes. Et s'il n'y a aucun coefficient non-nul devant x_j, c'est que la variable x_j a déjà été éliminée !

On a donc besoin d'une fonction qui, en partant d'indices donnés (r, j), cherche un coefficient non-nul *dans la colonne et en dessous à partir* de ce coefficient, et renvoie l'indice de ligne où elle l'a trouvé, et `None` si elle n'en trouve pas ; et on a besoin que les fonctions qui utilisent celle-ci vérifient si le résultat est bien un indice ligne ou bien est `None`.

Exercice 11
Écrire une fonction `cherche_pivot(A, r, j)` qui cherche l'indice ligne d'un coefficient non-nul dans la colonne j et dans les lignes d'indice $i \geqslant r$ 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 12 (*)
Écrire une fonction `échelonne(A)` qui échelonne la matrice A.

Le programme peut se décrire ainsi :

- On démarre avec une boucle principale `for` portant sur l'indice `j` des colonne, et on initialise aussi un indice `r` à 0 pour les lignes. Ces deux indices ne jouent pas le même rôle : à chaque étape on va toujours passer à la colonne suivante, par contre, on va passer à la ligne d'en-dessous seulement si on a bel et bien trouvé un pivot.
- On appelle alors la fonction `cherche_pivot(A, r, j)` sur la case (r, j).
  - Si on trouve un pivot : alors on effectue les opérations « comme d'habitude », à l'aide des fonctions de l'exercice précédent. On échange la ligne du pivot avec la ligne r, puis on effectue des combinaisons pour amener tous les coefficients à 0 en-dessous de (r, j). À la fin, on incrémente `r`, autrement dit on passera à la colonne suivante et on « descend d'une ligne ».
  - Si on n'en trouve pas : alors on ne fait rien du tout. La boucle `for` va faire passer à la colonne suivante, et la variable `r` ne bouge pas car on reste sur la même ligne.
- À la fin si tout se passe bien, la matrice est échelonnée est la variable `r` est en fait le **rang** de la matrice (on pourra l'afficher avec `print` avant de renvoyer la forme échelonnée).

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

Exercice 13 (*)
Écrire une fonction `inverse(A)` qui renvoie l'inverse de la matrice A, dans le cas où A est carrée et inversible.

- Au départ, on a besoin d'une matrice B qui est une copie de A (pour ne pas modifier A) ainsi que d'une matrice I qui est l'identité.
- **Toutes** les opérations élémentaires seront 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.
- Dans un premier temps on échelonne B exactement comme dans l'exercice précédent, en effectuant les opérations simultanément sur I.
  - Si tout se passe bien, on trouve un pivot à chaque étape, et à la fin la matrice B est échelonnée et le dernier coefficient en bas à droite est non-nul, le rang est bien égal à la taille n. En même temps, la matrice I est triangulaire inférieure.
  - Si ce n'est pas le cas on peut éventuellement s'arrêter là et renvoyer une erreur : c'est que la matrice n'est pas inversible !
- Ensuite il faut remonter, en partant d'en bas. On a donc un deuxième morceau de la fonction avec une nouvelle (double) boucle, qui cette fois part de la fin. On utilise des opérations élémentaires $L_i \leftarrow aL_i + bL_j$ pour éliminer toute la colonne au-dessus du coefficient diagonal de B, partant d'en bas à droite, et ensuite on le met à 1 avec une dilatation. Encore une fois, ces opérations sont faites simultanément sur B et sur I. Ici il n'y a rien à « chercher », car les pivots sont déjà sur la diagonale et sont non-nuls.
- À la fin, B est d'indentité et la matrice I est l'inverse de A.