Calcul matriciel et vectoriel sous Python

L'objectif de ce DM est de s'exercer à la manipulation des vecteurs et matrices sous Python et de l'appliquer à quelques problématiques issues du programme de mathématiques.

On utilisera la bibliothèque numpy pour les opérations matricielles et vectorielles.

1. Résolution d'un système triangulaire - Pivot de Gauss (révisions 1re année)

Soient **A** une matrice carrée inversible de taille *n* et **y** un vecteur de même taille. La *méthode du pivot de Gauss* vue en première année permet de résoudre le système matriciel **Ax = y**, c'est-à-dire de calculer le vecteur colonne **x** solution de cette équation.

Rappelons que cette méthode consiste dans un premier temps à transformer le système d'équations, c'est-à-dire la matrice **A** et le second membre **y**, de sorte à rendre la matrice *triangulaire*. On utilise pour cela l'algorithme décrit par le pseudo-code suivant, appelé *triangulation* (ne pas confondre avec la « trigonalisation » du cours de math de 2e année, qui transforme A en une matrice qui lui est « semblable », alors qu'on obtient ici par le pivot de Gauss une matrice « équivalente par lignes » !):

pour j de 0 à *n*-1 **faire:**
  trouver i entre j et *n*-1 tel que $|a_{i,j}|$ soit maximal *# (le "pivot partiel")*
  échanger les lignes L_i et L_j *# (de la matrice et du second membre)*
  pour *k* de j+1 à *n*-1 **faire:**
    $L_k \leftarrow L_k - \frac{a_{k,j}}{a_{j,j}} L_j$ *# (sur la matrice et le second membre)*

où $a_{i,j}$ désigne le coefficient d'indices (*i*,*j*) de la matrice **A** et où L_i désigne la *i*-ème ligne du système d'équations, c'est-à-dire de la matrice **A** *et* du vecteur colonne **y**. Ces opérations sont programmées en Python dans le fichier **`gauss.py`** que vous trouverez dans le répertoire du TP.

Ce programme utilise le type `ndarray` du module `numpy` pour représenter les matrices et les vecteurs. Un vecteur est ainsi représenté par un tableau à une seule dimension (commande `array([1,2])` tandis qu'une matrice est représentée par un tableau à deux dimensions contenant ses *lignes* (commande `array([[1,2],[3,4]])`).

Attention, les contenus des tableaux `array` *sont typés : s'ils sont initialement entiers, ils le resteront, quitte à ce que Python effectue des arrondis indésirables. Pour éviter ce problème, on peut les définir comme des tableaux de flottants en utilisant la syntaxe suivante :* `A = array([[1,2],[3,4]],float)`.

- **Après avoir pris connaissance du code source et identifié les différentes fonctions, définir les systèmes suivants et leur appliquer la transformation ci-dessus à l'aide de la fonction `triangul(A,y)` :**

$$\begin{cases} x - y + 2z = 5 \\ 3x + 2y + z = 10 \\ 2x - 3y - 2z = -10 \end{cases} \quad (1) \qquad \text{et} \qquad \begin{cases} 2x - y = 0 \\ -x + 2y - z = 0 \\ 0 - y + z = 1 \end{cases} \quad (2)$$

A l'issue de ces opérations, on dispose d'une matrice *triangulaire supérieure* **B** et d'un autre second membre **z** tels que le système triangulaire **Bx = z** possède la *même* solution que le système initial **Ax = y**. Il reste alors à résoudre ce système triangulaire. La résolution s'effectue de proche en proche

- **En réfléchissant bien à l'ordre des différentes opérations, écrire la fonction `resolTriang(B,z)` de sorte à retourner la solution x du système triangulaire supérieur Bx = z.**
- **Ecrire alors la fonction `resolGauss(A,y)` de sorte à retourner la solution du système Ax = y.**
- **Vérifier le bon fonctionnement de votre programme en le testant sur les systèmes (1) et (2) ci-dessus et en comparant le résultat à celui obtenu par le solveur linéaire du module `numpy`, appelé par la commande `x = solve(A,y)` sous Spyder ou `x = np.linalg.solve(A,y)` dans le cas général** (en supposant que **`numpy`** a été importé par la commande **`import numpy as np`**).

Pour finir, nous allons comparer l'efficacité des deux méthodes (pivot de Gauss et solveur de Python) sur des systèmes linéaires de taille variable, générés aléatoirement. On utilisera pour cela la fonction `time()` du module `time`, dont on rappelle l'utilisation :

```
import time
debut = time.time()
# insérer ici les instructions à chronométrer
fin = time.time()
print('Temps de calcul : ',fin-debut)
```

- **Compléter le programme de sorte à générer des matrices et vecteurs aléatoires de tailles 100, 200 et 400** (on pourra utiliser la fonction **`rand`**), **et à chronométrer les résultats obtenus par les deux méthodes. Commenter le résultat obtenu. Pour le pivot de Gauss, est-il en accord avec la complexité théorique de la méthode ?**

2. Orthonormalisation de Gram-Schmidt. Décomposition QR.

Soit $(\mathbf{v_0}, \mathbf{v_1}\ldots\ \mathbf{v_{n-1}})$ une famille libre de *n* vecteurs. Le *procédé de Gram-Schmidt* est une méthode permettant d'orthonormaliser cette famille, c'est-à-dire de construire une famille orthonormale $(\mathbf{e_0}, \mathbf{e_1}, \ldots\ \mathbf{e_{n-1}})$ engendrant les mêmes sous-espaces vectoriels successifs que les $\mathbf{v_i}$: $\mathbf{Vect(e_0, \ldots, e_{i-1}) = Vect(v_0, \ldots, v_{i-1})}$.

Cette méthode consiste à construire successivement chacun des vecteurs $\mathbf{e_i}$ en partant du vecteur $\mathbf{v_i}$ correspondant, en lui soustrayant son projeté orthogonal sur $\mathbf{Vect(e_0, \ldots, e_{i-1})}$ (*ces vecteurs sont déjà construits*), puis en normant le résultat. Cela s'écrit, pour tout *i* compris entre 0 et *n*-1 :

$$\boldsymbol{e}_i \leftarrow \frac{\boldsymbol{v}_i - \sum_{j=0}^{i-1}\langle \boldsymbol{v}_i, \boldsymbol{e}_j\rangle\, \boldsymbol{e}_j}{\left\|\boldsymbol{v}_i - \sum_{j=0}^{i-1}\langle \boldsymbol{v}_i, \boldsymbol{e}_j\rangle \boldsymbol{e}_j\right\|}$$

Nous allons programmer cette méthode en utilisant le produit scalaire canonique (qui s'obtient par la syntaxe `np.dot(u,v)` où `u` et `v` sont des vecteurs) et la norme associée (qui s'obtient par la syntaxe `np.linalg.norm(u)`).

- **Écrire une fonction `gramschmidt(V)` prenant comme argument une liste de vecteurs contenant la famille libre et retournant une autre liste de vecteurs contenant la famille orthonormalisée.**

- **Tester cette fonction sur la famille suivante en vérifiant que la famille obtenue est bien orthonormale :**

$$\boldsymbol{v}_1 = \begin{pmatrix}2\\1\\2\end{pmatrix} \qquad \boldsymbol{v}_2 = \begin{pmatrix}12\\-6\\0\end{pmatrix} \qquad \boldsymbol{v}_3 = \begin{pmatrix}-3\\-3\\18\end{pmatrix}$$

  Une façon commode de tester le résultat est de construire la **matrice de Gram** de la famille obtenue par le procédé : la matrice de Gram d'une famille de vecteurs $\mathbf{(v_0, v_1, \ldots, v_{n-1})}$ est la matrice carrée d'ordre n dont le coefficient d'indice (i,j) est le produit scalaire $\langle v_i, v_j\rangle$.

- **Écrire une fonction `gram(V)` prenant comme argument une liste de vecteurs et retournant la matrice de Gram de cette famille de vecteurs. L'utiliser pour vérifier les résultats de la fonction `gramschmidt`.**

Pour les plus courageux : Enfin, si **A** est une matrice carrée inversible d'ordre **n**, elle se décompose de façon unique sous forme d'un produit : **A=QR**, où **Q** est une matrice orthogonale, et **R** une matrice triangulaire supérieure dont les coefficients diagonaux sont strictement positifs. Si on considère la famille $\mathbf{C=(c_0, c_1, \ldots, c_{n-1})}$ des vecteurs-colonnes de la matrice **A**, et si on note $\mathbf{E=(e_0, e_1\ldots\ e_{n-1})}$ son orthonormalisée, alors **Q** est la matrice de passage de la base canonique $\mathbf{B_0}$ vers la base **E** (elle contient donc les vecteurs de **E** disposés en colonne), et **R** est la matrice de passage de la base **E** vers la base **C** (ses coefficients sont donc des produits scalaires $\langle e_i, c_j\rangle$).

- **Écrire une fonction `QR(A)` prenant comme argument une matrice A supposée inversible et retournant les matrices Q et R définies ci-dessus. La tester sur des exemples et vérifier l'égalité QR=A.**