

Calcul matriciel et vectoriel sous Python

On utilisera la bibliothèque `numpy` pour les opérations matricielles et vectorielles, déjà importée dans le programme à compléter.

1. Résolution d'un système triangulaire - Pivot de Gauss

Soient $\mathbf{A}$ une matrice carrée inversible de taille n et $\mathbf{y}$ un vecteur de même taille. La *méthode du pivot de Gauss* vue en première année permet de résoudre le système matriciel $\mathbf{Ax} = \mathbf{y}$, c'est-à-dire de calculer le vecteur colonne $\mathbf{x}$ 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 $\mathbf{A}$ et le second membre $\mathbf{y}$, 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 2^e année, qui transforme $\mathbf{A}$ 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 $\mathbf{A}$ et où L_i désigne la i -ème ligne du système d'équations, c'est-à-dire de la matrice $\mathbf{A}$ et du vecteur colonne $\mathbf{y}$. Le code à remplir originalement est en annexe.

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

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

Réponse :

A l'issue de ces opérations, on dispose d'une matrice *triangulaire supérieure* $\mathbf{B}$ et d'un autre second membre $\mathbf{z}$ tels que le système triangulaire $\mathbf{Bx} = \mathbf{z}$ possède la même solution que le système initial $\mathbf{Ax} = \mathbf{y}$. 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 $\mathbf{x}$ du système triangulaire supérieur $\mathbf{Bx} = \mathbf{z}$.**

Réponse :

2. Orthonormalisation de Gram-Schmidt. Décomposition QR.

Soit $(\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1})$ 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 $(\mathbf{e}_0, \mathbf{e}_1, \dots, \mathbf{e}_{n-1})$ engendrant les mêmes sous-espaces vectoriels successifs que les $\mathbf{v}_i$: $\text{Vect}(\mathbf{e}_0, \dots, \mathbf{e}_{i-1}) = \text{Vect}(\mathbf{v}_0, \dots, \mathbf{v}_{i-1})$.

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 $\text{Vect}(\mathbf{e}_0, \dots, \mathbf{e}_{i-1})$ (ces vecteurs sont déjà construits), puis en normant le résultat. Cela s'écrit, pour tout i compris entre 0 et $n-1$:

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

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.

Réponse :

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 $(\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1})$ est la matrice carrée d'ordre n dont le coefficient d'indice (i,j) est le produit scalaire $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$.

- ✎ **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`.**

Réponse :

Annexe

```

import numpy as np
import time

def echange_lignes_m(A,i,j):
    A[i,:],A[j,:]=np.copy(A[j,:]),np.copy(A[i,:])
    return A
def echange_lignes_v(v,i,j):
    v[i],v[j]=np.copy(v[j]),np.copy(v[i])
    return v
def trans_mat(A,j,coeff,i):
    Li,Lj=np.copy(A[i,:]),np.copy(A[j,:])
    A[j,:]=Lj+(coeff*Li)
    return A
def trans_vec(v,j,coeff,i):
    vi,vj=np.copy(v[i]),np.copy(v[j])
    v[j]=vj+(coeff*vi)
    return v

# Triangulation de A, avec modification correspondante de
y
def triangul(A,y):
    B = np.copy(A)
    z = np.copy(y)
    n = len(z)
    for i in range(n): # Boucle sur toutes les lignes
        pivot=np.copy(B[i,i])
        ind_piv=i
        # DEBUT Recherche du plus grand pivot
        for k in range((i+1),n):
            if abs(B[k][i]) > abs(pivot):
                ind_piv = k
                pivot = np.copy(B[k,i])
        B = echange_lignes_m(B,ind_piv,i)
        z = echange_lignes_v(z,ind_piv,i)
        # FIN Recherche du plus grand pivot
        for k in range((i+1),n):
            coeff = -np.copy(B[k][i])/pivot
            trans_mat(B,k,coeff,i)
            trans_vec(z,k,coeff,i)
    return (B,z)

```