Calcul matriciel

I. Présentation du TD

L'objet de ce TD est double :

- dans un premier temps, nous allons écrire des procédures permettant de créer et de manipuler des matrices ; bien sûr, nous n'utiliserons pas la bibliothèque `numpy`, puisqu'il s'agit avant tout de s'entraîner à la programmation ;
- dans un second temps, nous utiliserons ces procédures pour résoudre des systèmes linéaires, inverser une matrice, calculer un déterminant etc...

II. Calcul matriciel de base

II.1. Représentation des matrices

Une matrice sera représentée par une liste de listes, chacune d'entre elles représentant les lignes. Ainsi, la matrice $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ sera représentée par :

```
>>> m = [ [1, 2, 3], [4, 5, 6]]
```

Pour obtenir le terme situé ligne i, colonne j, on écrira : $m[i][j]$. L'appel à $m[i]$ renvoie alors la i-ème ligne.
Attention : le premier élément d'une liste est indexé par 0 !
Exemple :

```
>>> m = [ [1, 2, 3], [4, 5, 6]]
>>> print(m[0][2])
3
>>> print(m[1])
[4, 5, 6]
>>> print(len(m))
2
>>> print(len(m[0]))
3
```

Puisque les listes en Python sont des objets *mutables*, il faut se méfier lorsque l'on affecte à une matrice une autre matrice, ou à une ligne d'une matrice une autre ligne... Il est indispensable d'utiliser la fonction `copy` ou la fonction `deepcopy`. Voyez les exemples ci-dessous :

```
>>> from copy import *
>>> m = [ [1, 2, 3], [4, 5, 6]]
>>> v = [7, 8, 9]
>>> m[1] = v
>>> print(m)
[[1, 2, 3], [7, 8, 9]]
>>> v[0] = 10
>>> print(m)
[[1, 2, 3], [10, 8, 9]]
>>> m[1] = copy(v)
>>> v[0] = 11
>>> print(m)
[[1, 2, 3], [10, 8, 9]]
>>> n = m
>>> n[1][1] = 14
>>> print(m)
[[1, 2, 3], [10, 14, 9]]
>>> n = deepcopy(m)
>>> n[1][1] = 15
>>> print(m)
```

```
[[1, 2, 3], [10, 14, 9]]
```

II.2. Calculs de base

Les procédures ci-dessous (qui sont fournies dans le fichier `matrices.py`, dans le répertoire de la classe, permettent de faire les calculs matriciels élémentaires.

```python
from copy import *

# on crée ici une nouvelle exception pour gérer les erreurs sur les matrices
class ExceptionMatrices(Exception):
    def __init__(self, raison):
        self.raison = raison
    def __str__(self):
        return self.raison

def cree_matrice(p, q, valeur=0):
    # crée une matrice de type (p,q) où chaque élément vaut valeur
    m = [None] * p
    for i in range(p):
        m[i] = [valeur] * q
    return m

def identite(n):
    # matrice identité d'ordre n
    m = cree_matrice(n, n, 0)
    for i in range(n):
        m[i][i] = 1
    return m

def nblignes(m):
    # renvoie le nombre de lignes de la matrice
    return(len(m))

def nbcolonnes(m):
    # renvoie le nombre de colonnes de la matrice
    return(len(m[0]))

def dimensions(m):
    # taille de la matrice
    return (nblignes(m), nbcolonnes(m))

def transpose(m):
    # retourne la transposée de m en utilisant
    # la définition d'une lsite par compréhension
    p, q = dimensions(m)
    return [[m[i][j] for i in range(p)] for j in range(q)]

def transpose2(m):
    # retourne la transposée, version plus élémentaire
    p, q = dimensions(m)
    t = cree_matrice(q, p, None)
    for i in range(p):
        for j in range(q):
            t[j][i] = m[i][j]

def prod_scal(k, m):
    # produit de la matrice m par le scalaire k
    p, q = dimensions(m)
    return [[k * m[i][j] for j in range(q)] for i in range(p)]

```

```python
def somme(A, B):
    # somme de deux matrices
    p, q = dimensions(A)
    p1, q1 = dimensions(B)
    if p != p1 or q != q1:
        raise ExceptionMatrices('Tailles incompatibles dans la somme.')
    return [[A[i][j] + B[i][j] for j in range(q)] for i in range(p)]

def prod(A, B):
    # produit de deux matrices, version très basique
    p, q = dimensions(A)
    q1, r = dimensions(A)
    if q != q1:
        raise ExceptionMatrices('Tailles incompatibles dans le produit.')
    C = cree_matrice(p, r, 0)
    for i in range(p):
        for k in range(r):
            for j in range(q):
                C[i][k] += A[i, j] * B[j, k]
    return C
```

III. Résolution d'un système linéaire par la méthode du pivot

III.1. Cas d'un système de Cramer

Nous considérons ici un système linéaire de la forme $AX = B$ où A est une matrice carrée inversible d'ordre n. On appelle matrice complète du système la matrice par blocs $\begin{bmatrix} A & B \end{bmatrix}$ obtenue en ajoutant la colonne B à droite de A.
Rappelons brièvement la méthode du pivot partiel, qui permet de transformer le système en un système triangulaire équivalent :

- La première colonne n'est pas nulle, sinon la matrice ne serait pas inversible. Il existe donc dans la première colonne un élément non nul, disons sur la ligne numéro i.
- On va alors échanger (éventuellement) les lignes d'indices 1 (0 en Python !) et i, pour mettre ce pivot en haut à gauche.
- Puis pour les lignes de numéro 2 à n, on fait les opérations $L_k \longleftarrow L_k - \dfrac{a_{k1}}{a_{11}} L_1$, qui ont pour effet de mettre un zéro en dessous du pivot.
  Cependant, compte tenu des erreurs d'arrondi qui vont s'accumuler :
  - il faut considérer qu'un coefficient inférieur à ε en valeur absolue (ε choisi par l'utilisateur, par exemple $\varepsilon = 1\text{e}-10$) est en fait nul ;
  - pour éviter de diviser par des nombres petits (car par exemple diviser par $1\text{e}-9$ revient à multiplier par 10^9 !), le pivot choisi dans la colonne sera le terme le plus grand en valeur absolue.
- On passe ensuite à la colonne suivante. À la colonne numéro j (avec $j \leqslant n$, on ne fait pas d'opération sur le second membre !) :
  - on choisit un pivot non nul dans la colonne j dans une ligne d'indice supérieur ou égal à j (il en existe sinon la matrice ne serait pas inversible ; comme précédemment on choisit le plus grand en valeur absolue).
  - on place ce pivot ligne j en faisant (éventuellement) un échange de lignes.
  - on effectue sur les lignes de numéros $j+1$ à n les opérations : $L_k \longleftarrow L_k - \dfrac{a_{kj}}{a_{jj}} L_j$.
- On s'arrête lorsque $j = n$. Le système est alors devenu triangulaire et il suffit de le résoudre en commençant par la dernière ligne et en remontant (je vous laisse écrire les formules).

Le programme final sera écrit en le « découpant » en procédures plus simples, décrites ci-dessous :

```python
# ici j'importe les procédures précédentes,
# stockées dans le fichier Matrices1.py
from Matrices1 import *

eps = 1e-12 # valeur à partir de laquelle un terme sera considéré nul

```

```python
def entre_systeme():
    # entrée du système par l'utilisateur, présentation rudimentaire mais suffisante
    # cette procédure renvoie la matrice complète du système, avec second membre
    n = int(input('Taille du système ? '))
    A = cree_matrice(n, n+1, 0)
    for i in range(n):
        if i == 0:
            print('Entrée de la 1ère ligne de la matrice du système:')
        else:
            print('Entrée de la ',i+1, 'ème ligne de la matrice du système:')
        for j in range(n):
            print('A[{:d},{:d}]= '.format(i+1,j+1), end='')
            A[i][j] = float(input())
    print('Entrée du second membre:')
    for i in range(n):
        print('B[{:d}]= '.format(i+1), end='')
        A[i][n] = float(input())
    return A

def ecrit_systeme(A):
    # affichage du système, présentation rudimentaire
    p, q = dimensions(A)
    for i in range(p):
        for j in range(q-1):
            print('{:.3f}*x[{:d}]'.format(A[i][j],i+1),end='')
            if j == q-2:
                print('=',A[i][q-1])
            else:
                if A[i][j+1]>0:
                    print('+',end='')

def ligne_pivot(A, i):
    # renvoie le numéro de ligne où figure le pivot à la ième étape
    # renvoie -1 si pas trouvé: erreur, matrice non inversible
    # ......

def echange_lignes(A, i, j):
    # échange dans A les lignes de numéros i et j
    # cette procédure ne renvoie rien, elle modifie A en place
    # .....

def transvection(A, k, j, mu):
    # fait dans A l'opération Lk <-- Lk - mu*Lj
    # à partir de la colonne j seulement car les termes à gauche sont déjà nuls
    # .....

def trigonalise(A):
    # applique la méthode du pivot en utilisant les procédures précédentes
    # pour rendre le système triangulaire
    # .....

def resolution(A):
    # le système de matrice complète A étant supposé triangulaire,
    # le résout et renvoie le vecteur solution
    n = nblignes(A)
    X = [None] * n
    #
    return X

# Et enfin le programme principal
A0 = entre_systeme()
```

```
68  A = deepcopy(A0)
69  # on fait une copie pour ne pas modifier le système initial
70  trigonalise(A)
71  X = resolution(A)
72  print('Les solutions du système:')
73  ecrit_systeme(A0)
74  print('sont:')
75  for i in range (nblignes(X)):
76      print('x[{:d}]={:.3f}'.format(i+1,X[i]))
```

Évaluation de la complexité de la méthode :

Pour évaluer cette complexité, il suffit de dénombrer le nombre d'opérations calculatoires ($+$, $-$, $*$, $/$) faites (on ne comptabilise pas les échanges, ni les affectations, ni les comparaisons).

- Pour la trigonalisation du système : considérons la k-ème étape. Pour chacune des lignes de numéros $k+1$ à n :
  - on calcule le coefficient $\dfrac{a_{ik}}{a_{kk}}$;
  - on retranche à la i-ème ligne la k-ième multipliée par ce coefficient soit, comme les opérations ne commencent qu'au k-ème élément, $n+2-k$ opérations.

  En tout, cette k-ième étape nécessite

  $$(n-k)\big(1+2(n+2-k)\big) = (n-k)(2n-2k+5)$$

  opérations, et comme k varie de 1 à n, cela donne en tout

  $$\sum_{k=1}^{n}(n-k)(2n-2k+5) = \sum_{k=0}^{n-1}k(2k+5) = \frac{n(n-1)(2n-1}{3} + \frac{5n(n-1)}{2}$$

  opérations, c'est-à-dire un nombre équivalent à $\dfrac{2n^3}{3}$.
- Pour la résolution du système une fois celui-ci trigonalisé, chaque x_i s'obtient par la formule :

  $$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=i+1}^{n} a_{ij}x_j\right)$$

  ce qui nécessite $n-i$ additions, $n-i$ multiplications et 1 division soit $2n-2i+1$ opérations en tout, et donc au total la résolution a nécessité :

  $$\sum_{i=1}^{n}(2n-2i+1) = \sum_{i=0}^{n-1}(2i+1) = n^2$$

  opérations.

Au final, l'ordre de grandeur de la complexité de la méthode est un $\mathrm{O}(n^3)$.

III.2. Cas d'un système quelconque

On considère ici un système linéaire à p lignes et q colonnes, de rang $r \geqslant 1$. On lui applique alors la méthode du pivot de Gauss-Jordan. Cette méthode consiste à mettre le système sous forme échelonnée. Pour cela :

- Chaque fois que l'on a trouvé un pivot, on met aussi des zéros au dessus ;
- lorsque l'on ne trouve pas de pivot, on se contente de passer à la colonne suivante, sans changer de ligne.

On obtiendra donc au final une matrice complète qui peut être de la forme suivante :

$$\begin{pmatrix}
\otimes & 0 & 0 & \bullet & 0 & \bullet & \dots & \dots & b_1 \\
0 & \otimes & 0 & \bullet & 0 & \bullet & \dots & \dots & b_2 \\
\vdots & 0 & \otimes & \bullet & 0 & \bullet & \dots & \dots & b_3 \\
\vdots & \vdots & 0 & 0 & \otimes & \bullet & \dots & \dots & b_4 \\
\vdots & \vdots & \vdots & \vdots & 0 & 0 & \otimes & \dots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \ddots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & b_n
\end{pmatrix}$$

où les $\otimes$ représentent des pivots (non nuls) et les $\bullet$ des termes éventuellement non nuls.

Dans ce cas, le rang r du système est le nombre de pivots (non nuls). Si les b_i pour $i \in [\![r+1\,;n]\!]$ sont non nuls, le système est incompatible. Sinon, les inconnues dont les numéros de colonne sont ceux où il y a un pivot sont les inconnues principales, et on peut alors les exprimer en fonction des inconnues secondaires.

En modifiant le programme précédent, écrire un programme qui permet de résoudre complètement un système linéaire.

IV. Autres applications de la méthode du pivot

IV.1. Calcul du déterminant d'une matrice carrée

Pour calculer le déterminant d'une matrice carrée, il suffit de la trigonaliser par la méthode exposée au **III.1**. Le déterminant sera alors le produit des éléments diagonaux, mais il faudra en plus le multiplier par $(-1)^\nu$ où ν est le nombre d'échanges de lignes qui ont été faits.
Le plus simple pour ce faire est de modifier la procédure `trigonalise`, en créant une variable `det`, initialement fixée à -1 et en faisant `det = -det` à chaque échange de ligne.

IV.2. Calcul de l'inverse d'une matrice carrée

Par l'algorithme de Gauss-Jordan décrit en **III.2**, et en divisant en plus chaque ligne par le pivot, on peut rendre une matrice carrée inversible d'ordre n équivalente par lignes à la matrice I_n.
On sait alors que les mêmes opérations élémentaires faites à partir de la matrice I_n permettent d'obtenir A^{-1}.