Résolution d'un système linéaire : Pivot de Gauss

I-Introduction

On décrit ici un algorithme pour résoudre numériquement des systèmes linéaires.

Nous utiliserons la méthode du pivot de Gauss pour résoudre un système linéaire de la forme:

$$\{a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 + \ldots a_{1,n}x_n = b_1 \; a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 + \ldots a_{2,n}x_n = b_2 \quad . \quad . \quad . \quad .$$

Exemple

Prenons un système 3x3 que l'on veut résoudre:

$$\{x + y + z = 2 \qquad (L1)\; x - y + 2z = 9 \qquad (L2)\; 2x - y + z = 7 \qquad (L3)$$

Pour résoudre le système:

- On élimine la variable **x** dans la deuxième équations, pour cela on réalise l'opération: **L2 ←L2- L1**
  On obtient alors le système:

$$\{x + y + z = 2 \qquad (L1) \quad -2y + z = 7 \qquad (L2)\; 2x - y + z = 7 \qquad (L3)$$

- On élimine la variable *x* dans la deuxième équations, on réalise l'opération **: L3 ←L3 - 2xL1**
  On obtient alors le système:

$$\{x + y + z = 2 \qquad (L1) \quad -2y + z = 7 \qquad (L2) \quad -3y - z = 3 \qquad (L3)$$

- On peut ensuite éliminer y dans la dernière équation en réalisant l'opération : **L3 ←L3 - (3/2) x L2**
  On obtient

$$\{x + y + z = 2 \qquad (L1) \quad -2y + z = 7 \qquad (L2) \quad -\tfrac{5}{2}z = -\tfrac{15}{2} \qquad (L3)$$

Maintenant que le système est sous forme **triangulaire**, on va pouvoir effectuer une phase de **remontée** pour déterminer la solution :

$$\{x + y + z = 2 \quad -2y + z = 7 \quad \{x + y + z = 2 \quad y = -2 \quad \{\; x = 1 \quad y = -2 \;\; z = 3$$

II- Formalisme matriciel et opérations élémentaires

Le système précédent peut s'écrire sous forme matricielle : **AX = B** avec :

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \dots a_{1,n} \\ a_{2,1} & a_{2,2} \dots a_{2,n} \\ . & . \\ . & . \\ . & . \\ . & . \\ . & . \\ . & . \\ a_{n,1} & a_{n,2} \dots a_{n,n} \end{pmatrix} \quad X$$

Dans notre exemple:

A= $\begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & 2 \\ 2 & - \end{pmatrix}$ **X=** $(x\ y\ z)$ Et **B=** $(2\ 9\ 7)$

La résolution d'un système linéaire par **la méthode de Gauss** se fait en **deux** phases :

- **mise du système sous forme triangulaire**; elle comporte les opérations:
  - La **transvection** : consiste à ajouter à une ligne une autre ligne, multipliée par un certain facteur
  - L'**échange** de deux lignes
- La phase de **remontée**.

Dans l'exemple introductif, nous n'avons pas eu besoin d'échanger des lignes. Mais si nous étions par exemple partis du système :

$$\begin{cases} y + z = 2 & (L1) \\ x - y + 2z = 9 & (L2) \\ 2x - y + z = 7 & (L3) \end{cases}$$

Il aurait été impossible d'éliminer **x** dans la deuxième et troisième équation puisque le coefficient devant **x** est nul dans la première. Pour se faire, il suffit d'échanger la première ligne avec une des deux autres, ce qui se traduit matriciellement par l'échange de deux lignes de la matrice **A** et des lignes correspondantes du vecteur colonne **B**.

- **Implémentation d'une matrice en python:**

Une matrice est représentée par un tableau à deux dimensions (en python par une **liste double** (une liste de listes) ou par un tableau numpy).

Dans notre implémentation on représentera une matrice par une **liste** de listes.

Exemple:

la matrice $A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & 2 \\ 2 & -1 & 1 \end{pmatrix}$ sera représentée par la liste :
`A=[[1,1,1],[1,-1,2],[2,-1,1]]`.

Le vecteur colonne $\mathbf{B} = \begin{pmatrix} 2 \\ 9 \\ 7 \end{pmatrix}$ sera représenté par la liste : **`B=[[2],[9],[7]]`**

On accèdera à l'indice ***(i , j)*** de la matrice **A** grâce à l'instruction **A[i][j].** Attention cependant aux indices: en mathématiques, les indices des matrices commencent à **1**. En Python, les indices commencent à **0**!

On décide de résoudre les systèmes linéaires sous forme matricielle. Pour résoudre le système **AX = B** supposé de Cramer (la matrice **A** est inversible).

On effectue les opérations sur les lignes de la matrice **A** et le vecteur colonne **B** jusqu'à arriver à un **système triangulaire.** On remonte ensuite par substitutions pour obtenir la solution du système.

On aura besoin d'une fonction permettant de réaliser la copie d'une matrice. En effet, a priori on ne veut pas toucher à la matrice de départ **A**, ni au vecteur colonne **B**.

```
def copie_matrice(M):
        return [M[i][:] for i in range(len(M))]
```

ou

```
def copie_matrice(M):
      n=len(M)
      L=[]
      for i in range(n):
            L.append(M[i][:])
      return L
```

❖ Écrire une fonction **echange_lignes(M,i,j)** qui échange dans la matrice **M** la ligne **i** et la ligne **j**. On devra obtenir:

```
>>>A=[[1,2,3],[5,6,7],[9,10,11]]
>>>echange_lignes(A , 0, 2)
>>>A
[[9,10,11],[5,6,7],[1,2,3]
```

```
def echange_lignes(M,i,j):
      """ transforme sur place la matrice M avec Li <-> Lj """
      nc=len(M[0])
      for k in range(nc):
          M[i][k],M[j][k]=M[j][k],M[i][k]
```

Complexité : $O(n)$

❖ Écrire une fonction **transvection (M,i, j ,mu)** qui ajoute, à la ligne **i, mu** fois la ligne **j**. cette fonction ne renvoie rien : elle modifie en place la matrice M.

```
>>>A=[[1,2,3],[5,6,7],[9,10,11]]
>>>transvection (A , 0, 2, -2)
>>>A
[[-17,-18,-19],[5,6,7],[9,10,11]]
```

```
def transvection(M,i,j,mu):
     """ transforme sur place la matrice M avec Li <- Li + mu*Lj
     Les indices commencent à zéro """
     nc=len(M[0]) #nc est le nombre de colonnes de M.
     for k in range(nc):
           M[i][k]=M[i][k] + mu*M[j][k]
```

Complexité : $O(n)$

♦ **Choix du pivot:**

Le choix du pivot $a_{i,i}$ est important pour des raisons de précision numérique, il est judicieux de choisir le pivot de **valeur absolue maximale**. En effet, diviser par un pivot dont la valeur absolue faible par rapport aux autres coefficients du système conduit à des erreurs d'arrondi importantes

❖ Écrire une fonction **indice_pivot(A,i)** qui cherche dans les éléments $a_{i,i}$, $a_{(i+1),i}$, $a_{(i+2),i}$ ....,$a_{n,i}$ le pivot ayant la valeur maximum en valeur absolue et qui renvoie l'indice de la ligne correspondante. On doit obtenir le résultat suivant:

```
>>> A =[[1,3,2,4,1],[0,0,3,7,1],[0,1,5,1,0],[1,2,1,1,4]]
>>> indice_pivot(A ,1)
3
```

```
def indice_pivot(A,i):
    '''retourne le plus grand pivot en valeur absolue sous M[i][i]'''
    nc=len(A)
    imax=i # la ligne du maximum provisoire
    for k in range(i+1,nc):
        if abs(A[k][i])>abs(A[imax][i]):
            imax=k # un nouveau maximum provisoire
    return(imax)
```

Complexité : $O(n))$

♦ **Mise d'une matrice sous une forme triangulaire:**

Pour transformer le système en un système triangulaire on propose le **pseudo code** suivant:

Pour *i* allant de **0** à ***n-2*** **faire**
- Trouver *j* entre *i* et ***n-1*** tel que soit maximal *(indice du pivot max);*
- **Si** $i \neq j$ **alors**
  - Echanger ***Li*** et ***Lj*** (coefficients de la matrice et membres de droite)
- **Pour** *k* allant de ***i+1*** à ***n-1*** **faire**
  - (pour les deux matrices)

❖ Écrire une fonction **trianguler(A,B)** qui réalise cette opération. On devrait obtenir le résultat suivant

```
>>>A=[[1, 1, 1], [1, -1, 2], [2, -1, 1]]
>>>B=[[2],[9],[7]]
>>>trianguler(A,B)
>>> A
[[2, -1, 1], [0.0, 1.5, 0.5], [0.0, 0.0, 1.6666666666666667]]
>>> B
[[7], [-1.5], [5.0]]
```

```
def trianguler(A,B):
        """Mise sous forme triangulaire"""
        n=len(A)
        for i in range(n-1):
                j=indice_pivot(A,i)
                if j!=i:
                        echange_lignes(A,i,j)
                        echange_lignes(B,i,j)
                for k in range(i+1,n):
                        mu=-A[k][i]/A[i][i]
                        transvection(A,k,i,mu)
                        transvection(B,k,i,mu)
```

❖ Calculer la complexité de la fonction **trianguler(M)**

n: nombre de ligne la matrice **A** (paramètre de complexité)

Nombre approximative des opérations :

$$T(n) = 2 + 1 + \sum_{i=1}^{n-2} \left(n + 1 + 1 + 2n + \sum_{k=i+1}^{n-1} (1 + 3 + 2n) \right)$$

$$T(n) = n^3 + 2n^2 - 8n + 3$$

$$T(n) = O(n^3)$$

♦ Phase de remontée : résolution d'un système triangulaire

Comme son nom l'indique, La phase de remontée consiste à obtenir les composantes de X.

Le calcul des composantes de **X** se fait en commençant par la dernière ligne.

Posons $A = (a_{i,j})_{0 \le i\,,\, j \le n-1}$ *avec* $a_{i,j} = 0$ *pour* $i > j$

Posons ***X*** le vecteur colonne de composantes : $x_0, x_1, \ldots, x_{n-1}$, le produit ***A.X*** s'écrit:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \ldots & \ldots & a_{0,n-1} \\ 0 & a_{1,1} & a_{1,2} & \ldots & a_{1,n-1} \\ 0 & 0 & a_{2,2} & \ldots & a_{2,n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \end{pmatrix}$$

$$= \begin{pmatrix} a_{0,0}.x_0 + (a_{0,1}.x_1 + a_{0,2}.x_2 + \ldots a_{0,n-1}.x_{n-1}) \\ a_{1,1}.x_1 + (a_{1,2}.x_2 + a_{1,3}.x_3 + \ldots a_{1,n-1}.x_{n-1}) \\ a_{2,2}.x_2 + (a_{3,2}.x_3 + a_{3,4}.x_4 + \ldots a_{3,n-1} \\ \ldots \end{pmatrix}$$

Pour tout ***i*** entre ***0*** et ***n-1***, on tire l'égalité : $a_{i,i}.x_i + \sum_{j=i+1}^{n-1} a_{i,j}.x_j = b_i$

L'expression $x_i = \frac{1}{a_{i,i}}(b_i - \sum_{j=i+1}^{n-1} a_{i,j}.x_j)$ **,** ne fait intervenir que les x_j *avec* $j > i$.

$$X[i][0] = \frac{1}{A[i][i]}.(B[i][0] - \sum_{j=i+1}^{n-1} A[i][j].X[j][0])$$

Pour cela on vous propose le pseudo code suivant:

Pour *i* allant de **n-1** à ***0*** **faire**

    S 0

    Pour *j* allant de ***i+1*** à ***n-1*** **faire**

        S S +

❖ Écrire la fonction **`remonter(A,B)`** de remontée qui prend en argument la matrice **A** et le vecteur colonne **B** de type supposés triangulaires et renvoie le vecteur colonne solution **X** associé. On doit obtenir le résultat suivant:

```
>>>A=[[1, 1, 1], [1, -1, 2], [2, -1, 1]]
>>>B=[[2],[9],[7]]
>>>trianguler(A,B)
>>> A
[[2, -1, 1], [0.0, 1.5, 0.5], [0.0, 0.0, 1.6666666666666667]]
>>> B
[[7], [-1.5], [5.0]]
```

```
>>> remonter(A,B)
[[1.0], [-2.0], [3.0]]
```

```
def remonter(A,B):
        n=len(A)
        X=[[0]for i in range(n)]
        for i in range(n-1,-1,-1):
                s=0
                for j in range(i+1,n):
                        s=s+A[i][j]*X[j][0]
                X[i]=[(B[i][0]-s)/ A[i][i]]
        return X
```

❖ Calculer la complexité de la fonction **`remonter(A,B):`**

n: nombre de ligne la matrice A (paramètre de complexité)

Nombre approximative des opérations :

$$T(n) = 2 + n + 1 + \sum_{i=0}^{n-1} \left(1 + 1 + \left(\sum_{j=i+1}^{n-1} 3 \right) + 3 \right) + 1$$

$$T(n) = \frac{1}{2}\left(3n^2 + 7n + 2\right)$$

$$T(n) = O(n^2)$$

Écrire une fonction **Gauss(A,B)** qui prend en argument la matrice **A** carrée supposée inversible et la matrice colonne **B** et renvoie le vecteur solution **X** de l'équation **AX = B**. Par exemple, pour le système de l'exemple introductif : $\{x+y+z=2 \quad x-y+2z=9 \quad 2x-y+z=7$

On aura :

```
>>>PivotDeGauss([[1, 1, 1], [1, -1, 2], [2, -1, 1]],[[2],[9],[7]])
[[1.0], [-2.0], [3.0]]
```

```
def Gauss(A,B):
        A=copie_matrice(A)
        B=copie_matrice(B)
        trianguler(A,B)
        X=remonter(A,B)
        return (X)
```

❖ en déduire la complexité de la fonction **gauss(A,B)**:

n: nombre de ligne la matrice **A** (paramètre de complexité)

$$T(n) = 2n + n^3 + n^2$$

$$T(n) = O(n^3)$$

❖ Vérifier le résultat avec la fonction **solve** du module **numpy.linalg**, on doit obtenir :

```
>>>from numpy.linalg import solve
>>>solve([[1, 1, 1], [1, -1, 2], [2, -1, 1]],[[2],[9],[7]])
array([[ 1.],
       [-2.],
       [ 3.]])
```

❖ **Conclusion**

La méthode pivot de Gauss appelée **aussi Elimination de 'Gauss-Jordan'** peut être utilisée pour:

- Calculer le déterminant d'une matrice carrée inversible
- Calculer l'inverse d'une matrice carrée inversible

Voir(TP)