Informatique en CPGE (2018-2019) Résolution d'un système linéaire inversible: méthode de Gauss

S. B.

Lycée des EK

12 mars 2019

Nous pouvons utiliser des listes pour représenter des matrices. Une liste composée de n listes de longueurs p représente une matrice (n, p) (n lignes et p colonnes).

Par exemple la matrice $\begin{pmatrix} 2 & 2 & -4 \\ 5 & 13 & 7 \\ 4 & 8 & 1 \end{pmatrix}$ peut se définir en Python par le code suivant :

```
matrice=[[2,2,-4],[5,13,7],[4,8,1]]
a=matrice[1][2]
print(a) # affiche l'élément 7
```

On peut aussi créer une liste vide `matrice`, puis créer les listes `ligne` une par une en les ajoutant à la liste `matrice` :

```
matrice=[ ]
for i in range(n) : # n lignes
        ligne=[ . . . ] # une ligne de longueur p
        matrice.append(ligne)
```

On pourrait envisager une autre possibilité en créant une liste composée de *n* listes de longueurs *p* où chaque élément est initialisé avec la valeur **None** ou la valeur 0.

```
mat=2*[3*[None]] # initialisation de la matrice
for i in range(2) :
        for j in range(3) :
                mat[i][j]=i+2*j # par exemple
        print(mat[i])
print(mat)
```

Ce code affiche

```
[0, 2, 4]
[1, 3, 5]
[[1, 3, 5], [1, 3, 5]] # la première ligne a été modifiée
```

Donc cela ne fonctionne pas : la modification de la deuxième ligne s'est répercutée sur la première.

Un code qui fonctionne est :

```
mat=2*[None]
for i in range(2) :
    mat[i]=3*[None]

for i in range(2) :
    for j in range(3) :
        mat[i][j]=i+2*j
```

qui construit la matrice souhaitée :

```
[[0, 2, 4], [1, 3, 5]]
```

Pour la suite, nous allons définir une fonction `matrice` qui crée, avec le code précédent, une matrice nulle (n, p) dont on pourra modifier les coefficients à volonté.

```
def matrice(n,p) :
    mat=n*[None]
    for i in range(n) :
        mat[i]=p*[0]
    return mat
```

ou bien, avec une construction en compréhension :

```
def matrice(n,p) :
    return [ [0 for j in range(p)] for i in range(n)]
```

Attention, pour une matrice (n, p), les lignes sont numérotées de 0 à $n-1$ et les colonnes de 0 à $p-1$.

Somme de deux matrices

Pour faire la somme de deux matrices (n, p), on utilise deux boucles "for" imbriquées. On peut alors définir une fonction `somme` ainsi :

```
def somme(m1,m2) :
    n=len(m1) # on a besoin du nombre de lignes
    p=len(m1[0]) # et du nombre de colonnes
    mat=matrice(n,p) # une matrice nulle
    for i in range(n) : # boucle sur les lignes
        for j in range (p) : # boucle sur les colonnes
            mat[i][j]=m1[i][j]+m2[i][j]
    return mat
```

La matrice somme peut aussi se définir en compréhension en écrivant :

```
return [[m1[i][j]+m2[i][j] for j ...] for i...]
```

Multiplication d'une matrice par un réel

Le principe est le même que pour la somme :

```
def multiple(m,k) :
    n=len(m)
    p=len(m[0])
    mat=matrice(n,p)
    for i in range(n) : # boucle sur les lignes
        for j in range (p) : # boucle sur les colonnes
            mat[i][j]=k*m[i][j]
    return mat
```

Définition en compréhension :

```
def multiple(m,k) :
    n=len(m)
    p=len(m[0])
    return [ [k*u[j] for j in range(p)] for u in m]
```

Produit de deux matrices

Pour le produit de deux matrices, c'est un peu plus compliqué et il faut vérifier que le nombre de colonnes de la première matrice est égal au nombre de lignes de la deuxième matrice.

```
def produit(m1,m2) :
    n=len(m1)
    p=len(m1[0])
    q=len(m2)
    r=len(m2[0])
    if p !=q : return [None]
    mat=matrice(n,r)
    for i in range(n) : # boucle sur les lignes
        for j in range (r) : # boucle sur les colonnes
            for k in range(p) :
                mat[i][j]+=m1[i][k]*m2[k][j]
    return mat
```

Pour appliquer l'algorithme du pivot de Gauss, il est nécessaire de définir de nouvelles opérations. On se placera dans le cas où le système a une solution unique.

A chaque étape, on recherche le plus grand pivot (en valeur absolue).

$$\begin{array}{c} \\ L0 \\ L1 \\ L2 \\ \mathbf{Ls} \\ Ls{+}1 \\ Ls{+}2 \\ \dots \\ L\ n{-}1 \end{array} \begin{pmatrix} 2 & 2 & -4 & 6 & 5 & 4 & \dots & 3 & 2 \\ & 13 & 7 & 3 & -5 & 8 & \dots & -7 & 4 \\ & & 1 & -5 & 2 & 4 & \dots & -4 & 9 \\ & & & \mathbf{3} & -5 & 2 & \dots & 5 & 7 \\ & & & \mathbf{1} & 3 & 2 & \dots & 8 & 9 \\ & & & \mathbf{5} & 5 & 3 & \dots & 9 & 6 \\ & & & \mathbf{\dots} & \dots & \dots & \dots & \dots & \dots \\ & & & \mathbf{4} & -3 & 2 & \dots & 5 & -4 \end{pmatrix}$$

Le pivot provisoire sur l'exemple est m[s][s] = 3, et on cherche le maximum en valeur absolue des nombres m[i][s] pour i variant de s+1 à n-1. Dans le cas où le système a une solution unique, on démontre que ces nombres ne sont pas tous nuls.

La fonction `pivot` prend en argument une matrice et le numéro du pivot que l'on cherche, (0 pour la première étape), et renvoie le numéro de la ligne contenant le pivot qui va être utilisé.

```
def pivot(m,s) :
    n=len(m)
    np=s # numero du pivot provisoire
    for i in range(s+1,n) : # boucle sur les lignes restantes
        if abs(m[i][s])>abs(m[np][s]) :
            np=i
    return np
```

Pour échanger deux lignes d'une matrice, sans modifier la matrice d'origine du système, nous créons une nouvelle matrice, copie de la matrice passée en argument.
La fonction `permute` prend en argument une matrice et les numéros des deux lignes à échanger :

```
def permute(m,i,j) :
    n=len(m)
    p=len(m[0])
    mat=[ [u[j] for j in range(p)] for u in m] # copie
    for k in range (p) : # boucle sur les colonnes
        mat[i][k],mat[j][k]=mat[j][k],mat[i][k]
    return mat
```

Les transvections sont les transformations centrales dans l'algorithme du pivot de Gauss.
Si s est le numéro du pivot utilisé, on remplace chaque ligne m[i], pour i variant de s+1 à n-1, par m[i]- k*m[s], où

k=m[i][s]/m[s][s], soit $L_i \longleftarrow L_i - \frac{a_{i,s}}{a_{s,s}} L_s$.

Avec la notation matricielle habituelle, l'algorithme est le suivant :
Pour i variant de s+1 à n-1

$$k = \frac{a_{i,s}}{a_{s,s}}$$

Pour j variant de s à p-1

$$a_{i,j} = a_{i,j} - k \times a_{s,j}$$

```
def transvection(m,s) : # s numéro du pivot utilisé
    n=len(m)
    p=len(m[0])
    mat=[ [u[j] for j in range(p)] for u in m] # copie
    for i in range(s+1,n) : # boucle sur les lignes
        k=m[i][s]/m[s][s]
        for j in range (s,p) : # boucle sur les colonnes
            mat[i][j]=mat[i][j]-k*mat[s][j]
    return mat
```

Le principe de l'algorithme du pivot de Gauss est d'exécuter des tâches répétitives qui fournissent à chaque étape un système équivalent dans le but d'obtenir finalement un système triangulaire.

Voici un exemple de système triangulaire :

$$\begin{cases} 2x + y - 3z &= 4 \\ -2y + 2z &= 8 \\ 5z &= 15 \end{cases}$$

Algorithme avec la recherche du meilleur pivot :

Pour s variant de 0 à n-2
    Recherche du pivot : p=max$_{s \leq i \leq n-1} |a_{i,s}|$
    Si p différent de s
        Echange des lignes s et p
    Pour i variant de s+1 à n-1
        $k = \dfrac{a_{i,s}}{a_{s,s}}$
        Pour j variant de s à p-1
            $a_{i,j} = a_{i,j} - k \times a_{s,j}$

On résout un système triangulaire de bas en haut : on commence par la dernière équation puis à chaque étape, pour résoudre une équation, on substitue aux inconnues d'une ligne les valeurs trouvées dans les lignes inférieures.

La matrice associée au système précédent est

$$\begin{pmatrix} 2 & 1 & -3 & 4 \\ 0 & -2 & 2 & 8 \\ 0 & 0 & 5 & 15 \end{pmatrix}$$

Nous allons définir une fonction `solution` qui prend en argument une telle matrice et renvoie la solution du système associé. Si la solution est $(x_0, x_1, \ldots, x_{n-1})$, l'algorithme est le suivant :

Pour i variant de n-1 à 0

$$x_i = \frac{1}{a_{i,i}} \left(a_{i,p-1} - \sum_{j=i+1}^{p-2} a_{i,j} x_j \right)$$

```
def solution(m) :
    n=len(m)
    p=len(m[0])
    sol=n*[0] # création d'une solution
    for i in range(n-1,-1,-1) : # boucle sur les lignes
        sol[i]=m[i][p-1]
        for j in range(i+1,p-1) :
            sol[i]-=m[i][j]*sol[j]
        sol[i]=sol[i]/m[i][i]
    return sol
```

Pour résoudre un système linéaire, il n'y a plus qu'à assembler les fonctions qui viennent d'être étudiées.
On définit une fonction `gauss` qui prend en argument la matrice du système et renvoie la solution sous la forme d'une liste (que l'on peut considérer comme une matrice colonne).

```
def gauss(mat) :
    n=len(mat)
    for s in range(n-1) : # le dernier pivot est à l'avant dernière ligne
        piv=pivot(mat,s)
        if piv !=s :
            mat=permute(mat,s,piv)
        mat=transvection(mat,s)
    sol=solution(mat)
    return sol
```

Complexité

La résolution finale du système nécessite n divisions et $n(n-1)/2$ multiplications et soustractions.
Pour s donné, une transvection nécessite $n-1-s$ divisions pour le calcul de k,
puis $(n-1-s)(n-s+1)$ multiplications et soustractions pour les nouvelles lignes ;

s variant de 0 à $n-2$, on obtient donc $(n-1)n/2$ divisions, et

$$\sum_{1}^{n-1} u(u+2) = \sum_{1}^{n-1} u^2 + \sum_{1}^{n-1} 2u = (n-1)n(2n-1)/6 + (n-1)n$$

multiplications et soustractions, donc au total $n(n+1)/2$ divisions et $n^3/3 + n^2 - 4n/3$ multiplications et soustractions.

La complexité est en $\mathcal{O}(n^3)$.

La bibliothèque NumPy contient un module `linalg` pour l'algèbre linéaire.
Par exemple pour résoudre un système MX=C :

```
import numpy as np
M=[[1,1,1],[1,0,-1],[-1,1,0]]
C=[[6],[-2],[1]]
X=np.linalg.solve(M,C)
print(X) # solution : [1, 2, 3]

M=[[2,2,-3],[-2,-1,-3],[6,4,4]]
C=[[2],[-5],[16]]
X=np.linalg.solve(M,C)
print(X) # solution [-14, 21, 4]
```