Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique commune
G. Dewaele, A. Troesch

2015/2016

TP n° 13 : Systèmes linéaires

Correction de l'exercice 1 – Échelonnement d'une matrice et résolution d'un système

On prend le parti pris de faire toutes les opérations de façon élémentaire, coefficient par coefficient, afin d'avoir une meilleure appréciation de la complexité, non perturbée par d'éventuelles optimisations cachées dans l'utilisation d'opérations complexes.

1. On commence par définir un certain nombre de fonctions annexes : une fonction recherchant le pivot maximal en prenant comme point de départ le coefficient en position (i, j), et les opérations sur les lignes et colonnes. Dans toutes ces fonctions, on se sert du fait qu'un tableau est un objet mutable, donc que les opérations effectuées sur la copie passée en paramètre seront prises en considération sur la matrice initiale. Ainsi, les fonctions modifient les paramètres, mais ne retourne par la matrice résultat en sortie.

   Dans les opérations sur les lignes, le paramètre j_0 permet de ne faire les opérations qu'à partir des coefficients de la ligne j_0. cela évite de faire des opérations inutiles si la ligne utilisée commence par des 0.

```
import numpy as np

def recherche_pivot(A,i,j):
    """ recherche le pivot de valeur absolue maximale, sur la
    colonne j, à partir de la ligne i """
    m = abs(A[i,j])
    imax = i
    for k in range(i+1,np.shape(A)[0]):
        if abs(A[k,j]) > m:
            m = abs(A[k,j])
            imax = k
    if m < 1e-15:
        return -1
    else:
        return imax

def echange(A,B,i,j,j0):
    """échange les lignes i et j, à partir de l'indice de colonne j0
    sur A, les coefficients précédents étant supposés nuls"""
    for k in range(j0,np.shape(A)[1]):
        A[i,k],A[j,k] = A[j,k],A[i,k]
    for k in range(np.shape(B)[1]):
        B[i,k],B[j,k] = B[j,k],B[i,k]

def dilatation(A,B,i,l,j0):
    """dilatation sur la ligne i d'un facteur l, à partir de l'indice
    de colonne j0 pour la matrice A"""
    for k in range(j0,np.shape(A)[1]):
        A[i,k] *= l
    for k in range(np.shape(B)[1]):
        B[i,k] *= l
```

```
def transvection(A,B,i,j,l,j0):
    """transvection L_i <- L_i + l* L_j à partir de la colonne j0"""
    for k in range(j0,np.shape(A)[1]):
        A[i,k] += l* A[j,k]
    for k in range(np.shape(B)[1]):
        B[i,k] += l* B[j,k]
```

On effectue alors l'échelonnement de la matrice A, en effectuant les mêmes opérations sur une matrice B (ces opérations sont déjà effectuées dans les opérations élémentaires définies ci-dessus). La matice B est passée en argument optionnel. En plus de la liste des pivots, on retourne aussi le nombre d'échanges effectués, ceci afin de pouvoir calculer le déterminant.

```
def echelonne(A,B=[[]]):
    """echelonne la matrice A, et retourne la liste des positions des
    pivots ; il faut passer les matrices en type réel; si les
    coefficients sont typés entiers, les opérations sont effectuées
    en partie entière """
    pivots = []
    i = 0
    ech = 0
    for j in range(np.shape(A)[1]):
        if i< np.shape(A)[0]:
            ipiv = recherche_pivot(A,i,j)
        if ipiv != -1:
            pivots.append(j) # il y a un pivot sur la colonne j
            echange(A,B,i,ipiv,j)
            ech += 1
            for k in range(i+1,np.shape(A)[0]):
                l = -A[k,j] / A[i,j]
                transvection(A,B,k,i,l,j)
            i += 1 # le prochain pivot est sur la ligne suivante
    return pivots, ech
```

2. Puisqu'on dispose de la position des pivots (il y a un pivot sur chaque ligne, donc les pivots sont en position (i, a_i), où les a_i sont les termes de la liste `pivots`), il n'est pas dur d'adapter l'algorithme précédent pour la remontée.

```
def PivotRemontant(A,B,pivots):
    """ on effectue la remontée après le premier passage. On normalise
    les pivots"""
    i = len(pivots)-1 # indice de la dernière ligne non nulle
                      # après échelonnement
    pivots.reverse()
    for j in pivots:
        for k in range(i):
            transvection(A,B,k,i,-A[k,j]/A[i,j],j)
        dilatation(A,B,i,1/A[i,j],j)
        i -= 1
    pivots.reverse()
       # remet les pivots dans l'ordre afin de rendre la liste mutable
       # dans le même état qu'au départ
```

3. Le rang est égal au nombre de lignes non nulles de la matrice échelonnée, c'est-à-dire au nombe de pivots utilisés.

```
def rang(A):
    A = 1. * A
    B = [[]] * np.shape(A)[0]
    pivots = echelonne(A,B)[0]
    print(pivots)
    return len(pivots)
```

4. L'inverse d'une matrice s'effectue par un pivot descendant puis remontant, à condition d'avoir l'inversibilité, ce qu'on peut tester entre les deux étapes (suivant le nombre de pivots obtenus).

```
def inverse(A):
    A = 1.* A
    (m,n)= np.shape(A)
    if n != m:
        raise ValueError("Matrice␣non␣inversible␣(non␣carrée)")
    B = np.eye(n)
    pivots = echelonne(A,B)[0]
    if len(pivots) != n:
        raise ValueError("Matrice␣carrée␣non␣inversible")
    PivotRemontant(A,B,pivots)
    return(B)
```

Après avoir importé le module `numpy.random` sous l'alias `npr` et le module `time`, on définit une fonction créant aléatoirement une matrice d'ordre n, et une fonction mesurant le temps (qu'on itère jusqu'à ce qu'on ait effectivement une matrice aléatoire).

```
def matrice_alea(n):
    """ définit une matrice aléatoire de taille nxn"""
    A = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            A[i,j]= npr.randint(1000)
    return A

def mesure_temps(n):
    """temps d'exécution de l'inversion
    On itère jusqu'à  ce qu'une matrice aléatoire fournie soit
    effectivement inversible"""
    while True:
        A = matrice_alea(n)
        try:
            deb= time.time()
            inverse(A)
            fin = time.time()
            return fin - deb
        except:
            pass
```

L'instruction `print(mesure_temps(200)/ mesure_temps(100))` retourne environ 7.88, ce qui confirme le caratère quadratique de l'algorithme (multiplier la taille des données par 2 multiplie le temps de calcul par 8).

Pour le tracé on importe `matplotlib.pyplot` en tant que `plt`.

```
def trace_complexité_inverse(n):
    X = [i for i in range (n)]
    Y = [mesure_temps(i) for i in range(n)]
    plt.plot(X,Y)
    plt.xlabel("taille␣de␣A")
    plt.ylabel("temps␣d'inversion")
    plt.savefig('py065.eps')
    plt.show()
```

On obtient le graphe de la figure 1

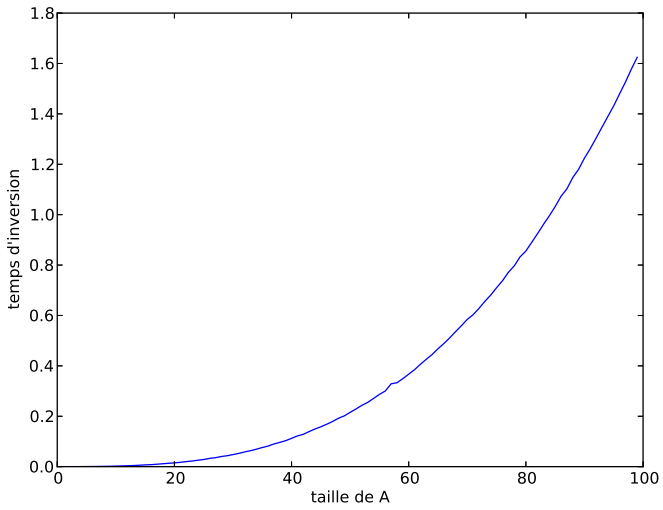

FIGURE 1 – Temps de réponse de l'inversion par pivot en fonction de la taille

5. Le déterminant se calcule par échelonnement, puis produit des coefficients diagonaux. Les opérations d'échange de ligne inversent le signe. Comme on effectue l'échelonnement sans dilatation, les autres opérations ne modifient pas le déterminant.

```
def determinant(A):
    n = echelonne(A)[1]
    d = 1
    for i in range(np.shape(A)[0]):
        d *= A[i,i]
    if n % 1 == 1:
        d *= -1
    return d
```

6. Après échelonnement, et pivot remontant, on teste l'existence d'une solution en vérifiant que toutes les coordonnées de B sous le dernier pivot sont toutes nulles. Ensuite, on donne une solution particulière en assignant aux coordonnées dont les indices correspondent aux indices de colonne des pivots, la valeur de la coordonnée de B située en face de ces pivots. Les autres coordonnées sont nulles. On trouve une base de l'espace homogène en assignant successivement à toute coordonnée d'indice non égal à l'indice de colonne d'un pivot la valeur 1 et 0 pour les autres ; La résolution de ce système fournit alors l'opposé des valeurs au-dessus du pivot correspondant pour les coordonnées correspondant aux autres colonnes, et 0 pour les autres coordonnées. On renvoie au cours de mathématiques pour une description plus claire.

```
def resout(A,B):
    pivots = echelonne(A,B)[0]
```

```
    PivotRemontant(A,B,pivots)
    n,m = np.shape(A)
    ## Cas où il n'y a pas de solution (système non compatible)
    for i in range(len(pivots),n):
        if abs(B[i,0])>1e-12:
            return ([],[])
    ## Solution particulière
    S0 = np.zeros((m,1))
    for i in range(len(pivots)):
        S0[pivots[i],0] = B[i,0]
    ## construction de la base
    base = []
    j = 0
    for i in range(m):
        if j < len(pivots) and i == pivots[j]:
            j+=1  ## indice associé à un pivot: on passe
        else:
            b = np.zeros((m,1)) ## construction d'un vecteur de base
            b[i,0]=1
            for k in range(len(pivots)):
                b[pivots[k],0] = - A[k,i]
            base.append(b)
    return (S0,base)
```

Testé sur l'exemple donné, on obtient bien le résultat escompté.

7. Un cas d'instabilité numérique.

   On ne suit pas tout à fait l'énoncé. On considère $B = H_{20}C_{20}$ puis on réout le système $H_{20}Y = B$. Normalement, on doit retrouver C_{20}.

```
def Hilbert(n):
    return(np.fromfunction(lambda i,j: 1/(i+j+1), (n,n)))

C1 = np.ones((20,1))
H = Hilbert(20)

B1 = np.dot(H,C1)
C2 = resout(H,B1)[0]

print(C1)
print(C2)
```

Le résultat obtenu est assez éloigné de C_{20} :

```
[[  1.00000014]
 [  0.99997507]
 [  1.00109398]
 [  0.97941607]
 [  1.20680651]
 [ -0.23692064]
 [  5.65599581]
 [-10.28126561]
 [ 18.60644517]
 [-16.97070976]
```

```
 [ 15.93709166]
 [-15.9956734 ]
 [ 20.74470627]
 [-10.49408031]
 [  0.        ]
 [  4.8661595 ]
 [  0.        ]
 [  1.61392825]
 [  0.        ]
 [  1.36703142]]
```

Ceci illustre le problème du mauvais conditionnement de la matrice H_n de Hilbert.

On pourra réutiliser les fonctions de l'exercice précédent, notamment les opérations sur les lignes et les colonnes, ainsi que les fonctions de résolution de système pour les comparaisons demandées en fin d'exercice.

Correction de l'exercice 2 – (Décomposition LU)

1. Les imports nécessaires et les opérations sur les lignes et les colonnes :

```python
import numpy as np
import numpy.random as npr
import time
import matplotlib.pyplot as plt

def echange(A,i,j,j0=0):
    """échange les lignes i et j, à partir de l'indice de colonne j0
    sur A, les coefficients précédents étant supposés nuls"""
    for k in range(j0,np.shape(A)[1]):
        A[i,k],A[j,k] = A[j,k],A[i,k]

def dilatation(A,i,l,j0=0):
    """dilatation sur la ligne i d'un facteur l, à partir de l'indice
    de colonne j0 pour la matrice A"""
    for k in range(j0,np.shape(A)[1]):
        A[i,k] *= l

def transvection(A,i,j,l,j0=0):
    """transvection L_i <- L_i + l* L_j à partir de la colonne j0"""
    for k in range(j0,np.shape(A)[1]):
        A[i,k] += l* A[j,k]

def transvectioncol(A,i,j,l,j0=0):
    """transvection C_i <- C_i + l* C_j à partir de la l j0"""
    for k in range(j0,np.shape(A)[0]):
        A[k,i] += l* A[k,j]


def echangecol(A,i,j,j0=0):
    """échange les col i et j, à partir de l'indice de colonne j0
    sur A, les coefficients précédents étant supposés nuls"""
    for k in range(j0,np.shape(A)[0]):
```

```
        A[k,i],A[k,j] = A[k,j],A[k,i]

def dilatationcol(A,i,l,j0=0):
    """dilatation sur la col i d'un facteur l, à partir de l'indice
    de li j0 pour la matrice A"""
    for k in range(j0,np.shape(A)[0]):
        A[k,i] *= l
```

On obtient alors la décomposition LU en échelonnant A sans échange de ligne, en effectuant à chaque fois les opérations inverses sur les colonnes de la matrice I_n. En particulier, pour toute transvection $L_i \leftarrow L_i + \lambda L_j$ effectuée sur A, on effectue $C_j \leftarrow C_j - \lambda C_i$ sur I_n.

```
def LU(A):
    """Retourne la décomposition LU"""
    (n,m)=np.shape(A)
    U=np.copy(A)
    if n != m:
        raise ValueError("La matrice n'est pas carrée")
    L = np.eye(n)
    for j in range(n):
        if abs(U[j,j])<1e-14:
            raise ValueError("Méthode non applicable")
        for k in range(j+1,n):
                l = -U[k,j] / U[j,j]
                transvection(U,k,j,l,j)
                transvectioncol(L,j,k,-l)
    return(L,U)
```

2. Pour résoudre un système triangulaire de façon minimale, on applique un pivot remontant, mais en ne faisant les calculs que sur le cevteur colonne B.

```
def syssup(A,B):
    """ Résolution du système AX=B, avec A triangulaire supérieure"""
    X = np.copy(B)
    for i in range(np.shape(A)[0]-1,-1,-1):
        for k in range(i):
            X[k,0] -= X[i,0] * A[k,i]/A[i,i]
        X[i,0] /= A[i,i]
    return X
```

3. De même pour les systèmes triangulaires inférieurs :

```
def sysinf(A,B):
    """ Résolution du système AX=B, avec A triangulaire inférieure"""
    X = np.copy(B)
    for i in range(np.shape(X)[0]):
        for k in range(i+1,np.shape(X)[0]):
            X[k,0] -= X[i,0] * A[k,i]/A[i,i]
        X[i,0] /= A[i,i]
    return X
```

4. On construit A carrée d'ordre n et une liste de k vecteurs colonnes B à n lignes.

```
def matrice_alea(n):
```

```
    """ définit une matrice aléatoire de taille nxn"""
    A = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            A[i,j]= npr.randint(1000)
    return A

def listeB(n,k):
    lb = []
    for a in range(k):
        B= np.zeros((n,1))
        for i in range(n):
            B[i,0]= npr.randint(1000)
        lb.append(B)
    return lb
```

On mesure ensuite les temps d'exécution pour chacune des trois méthodes proposées, en supposant définies deux fonctions `resout()` et `inverse()`, la première résolvant de bout en bout un système linéaire par la méthode du pivot, la deuxième calculant l'inverse d'une matrice par la méthode du pivot. Par ailleurs, afin d'effectuer les comparaisons de façon objective, sans avoir à tenir compte des optimisations liées aux implémentations Python de certaines opérations, on redéfinit manuellement le produit matriciel.

```
def produit(A,B):
    C = np.zeros((np.shape(A)[0],np.shape(B)[1]))
    for i in range(np.shape(A)[0]):
        for k in range(np.shape(B)[1]):
            for j in range(np.shape(A)[1]):
                C[i,k] += A[i,j]* B[j,k]
    return C

def tempsLU(A,lb):
    deb = time.time()
    L,U = LU(A)
    for B in lb:
        syssup(U,sysinf(L,B))
    fin = time.time()
    return fin - deb

def temps_gauss(A,lb):
    deb = time.time()
    for B in lb:
        Ap = np.copy(A)
        resout(Ap,B)
    fin = time.time()
    return fin - deb

def temps_inverse(A,lb):
    deb = time.time()
    Ainv = inverse(A)
    for B in lb:
        produit(Ainv,B)
    fin = time.time()
```

```
    return fin- deb
```

Et la comparaison :

```
def affiche_comparaison(n,k):
    A = matrice_alea(n)
    lb = listeB(n,k)
    print("par␣LU␣:␣{}".format(tempsLU(A,lb)))
    print("par␣pivot␣complet:␣{}".format(temps_gauss(A,lb)))
    print("par␣inversion␣:␣{}".format(temps_inverse(A,lb)))

affiche_comparaison(100,25)

### Résultat obtenu:

par LU : 1.315767765045166
par pivot complet: 14.398610353469849
par inversion : 1.9176409244537354
```

Un léger avantage semble se dessiner pour la méthode LU.

5. On effectue un graphe en faisant varier k (nombre de systèmes à résoudre) entre 1 et 25 :

```
def graphe(n,kmax):
    X = [k for k in range(1,kmax+1)]
    Y = []
    Z = []
    for k in range(1, kmax+1):
        A = matrice_alea(n)
        lb = listeB(n,k)
        Y.append(tempsLU(A,lb))
        Z.append(temps_inverse(A,lb))
    plt.plot(X,Y,label="LU")
    plt.plot(X,Z,label="inversion")
    plt.legend(loc = "best")
    plt.savefig('py046.eps')
    plt.show()

graphe(100,25)
```

La méthode par décomposition LU ne semble pas présenter un avantage considérable, à part un décalage initial. Peut-être y a-t-il des optimisations supplémentaires à faire ?

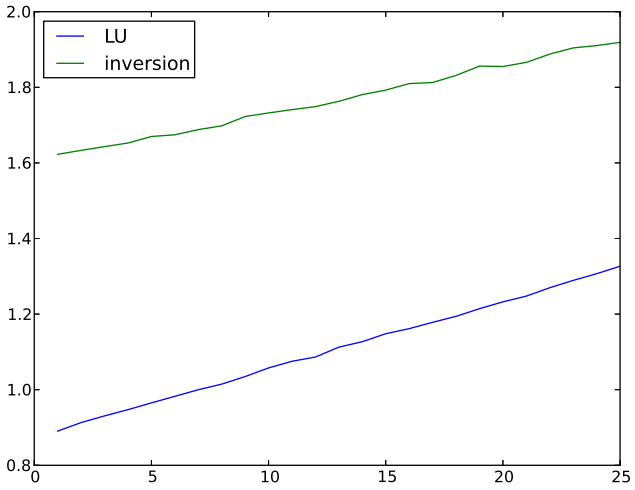

FIGURE 2 – Comparaison des temps de réponse pour k systèmes 100×100