Équations linéaires : méthodes directes

1. Introduction

On considère la résolution d'un système d'équations linéaires, que l'on met sous la forme matricielle suivante (4 équations) :

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_3 \\ b_3 \end{pmatrix} \tag{1}$$

soit sous forme générale :

$$Ax = b \tag{2}$$

où A est une matrice carrée de taille (N, N), x et b deux matrices colonnes de taille $(N, 1)$.

Dans l'hypothèse d'une matrice A inversible, on recherche l'unique solution (ou une valeur approchée) :

$$x = A^{-1}b \tag{3}$$

Dans certains cas, seule la solution x est recherchée. Dans d'autres cas, on recherche la matrice inverse de A. Celle-ci est obtenue en résolvant l'équation suivante :

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4}$$

Chaque colonne de la matrice $Y = A^{-1}$ est solution d'un système de la forme (7). En conséquence, la matrice inverse s'obtient par la résolution conjointe de N systèmes linéaires dont les colonnes b sont les vecteurs de la base canonique.

Plus généralement, on cherche à résoudre :

$$AX = B \tag{5}$$

où X et B sont deux matrices de taille (N, M).

Nous allons implémenter des méthodes directes de résolution de ce problème (par opposition aux méthodes itératives). Ces méthodes sont adaptées aux systèmes comportant un faible nombre d'équations (moins d'une centaine). Si les matrices A et b sont numériques, il s'agira d'un calcul numérique approché, soumis aux erreurs d'arrondis. Les algorithmes pourront aussi s'appliquer à des calculs formels opérant sur des fonctions (par exemple des fractions rationnelles), à condition de disposer d'une implémentation des opérations élémentaires pour ces fonctions (addition, multiplication).

2. Méthode d'élimination de Gauss-Jordan

2.a. Algorithme

Une méthode d'élimination consiste à éliminer des inconnues des équations. Elle procède en remplaçant chaque équation par une combinaison linéaire d'elle-même et d'une autre équation. Autrement dit, une étape élémentaire consiste à remplacer une ligne de la matrice A par une combinaison linéaire d'elle-même et d'une autre ligne, en effectuant la même combinaison pour la matrice B.

La méthode de Gauss-Jordan transforme la matrice A de manière à obtenir finalement la matrice identité. La matrice B finale est donc la solution cherchée. Par exemple, si B est initialement la matrice identité, sa version finale est la matrice A^{-1}.

La méthode de Gauss-Jordan procède colonne par colonne, en annulant à chaque fois les termes non diagonaux de la matrice A. Soit a_{ij} l'élément de la ligne i et de la colonne j (indices variant de 0 à N-1). Considérons le traitement de la colonne j (les $j-1$ colonnes à sa gauche étant déjà traitées). Supposons pour l'instant que a_{jj} soit non nul. La ligne j (des matrices A et B) peut être divisée par cet élément. Les autres lignes sont transformées en retranchant à chacune d'elle la ligne j multipliée par a_{ij} (transformation qui n'affecte pas les colonnes précédentes). Si $a_{jj} = 0$, il faut tout d'abord rechercher une ligne dont l'élément de la colonne j est non nul puis l'échanger avec la ligne j. L'échange de deux lignes (dans les matrices A et B) revient à changer l'ordre des équations, et n'a donc aucun effet sur l'ordre des inconnues. Si tous les éléments de la colonne sont nuls, la matrice est singulière.

Pour chaque colonne j, on doit donc choisir un élément non nul, appelé le pivot, dont la ligne correspondante est permutée avec la ligne j. Dans le cas d'un calcul matriciel numérique, les erreurs d'arrondis ont tendance à croître à chaque étape de l'élimination. Pour minimiser ce problème, ont doit choisir comme pivot l'élément de la colonne dont la valeur absolue est la plus grande.

2.b. Implémentation

Les calculs sont faits avec des `ndarray` (ou tableaux `numpy`). On remarquera qu'une combinaison linéaire de deux lignes d'une matrice peut être écrite directement, sans boucle sur les indices des colonnes. Pour échanger deux lignes, il faut procéder avec des copies de ces deux lignes.

La fonction `elimination_gauss_jordan(A,B)` effectue la réduction pour une matrice A carrée et un matrice B contenant éventuellement plusieurs colonnes. La matrice inverse est aussi calculée. La fonction renvoit la solution et la matrice inverse.

```
import numpy

def elimination_gauss_jordan(A,B):
    A=A.copy()
    B=B.copy()
    (n1,n2)=A.shape
    if n1!=n2:
        raise Exception(u'Matrice A non carrée')
    N=n1
    s=B.shape
```

```
    if len(s)==1:
        n1=s[0]
        n2 = 1
    else:
        (n1,n2)=s
    if N!=n1:
        raise Exception(u'Matrice B : nombre de lignes incorrect')
    inverse = numpy.identity(N)
    for j in range(N):
        i_pivot = 0
        maximum = abs(A[0,j])
        for i in range(1,N):
            if abs(A[i,j]) > maximum:
                maximum = A[i,j]
                i_pivot = i
        if maximum==0:
            raise Exception(u'Matrice A singulière')
        if i_pivot!=j:
            temp = A[j].copy()
            A[j] = A[i_pivot].copy()
            A[i_pivot] = temp
            temp = B[j].copy()
            B[j] = B[i_pivot].copy()
            B[i_pivot] = temp
            temp = inverse[j].copy()
            inverse[j] = inverse[i_pivot].copy()
            inverse[i_pivot] = temp
        f = 1.0/A[j,j]
        A[j] *= f
        B[j] *= f
        inverse[j] *= f
        for i in range(N):
            if i!=j:
                g = A[i,j]
                A[i] -= g*A[j]
                B[i] -= g*B[j]
                inverse[i] -= g*inverse[j]
    return (B,inverse)
```

Voici un exemple :

```
A=numpy.array([[5,4,-2,1],[-3,2,0,-5],[3,-5,2,0],[2,-3,0,1]],dtype=numpy.float64)
B=numpy.array([1,-2,3,0],dtype=numpy.float64)
(X,iA) = elimination_gauss_jordan(A,B)


print(X)
```

```
--> array([ 0.52173913,  0.43478261,  1.80434783,  0.26086957])

print(iA)
--> array([[ 0.14130435,  0.02173913,  0.14130435, -0.0326087 ],
       [ 0.07608696, -0.06521739,  0.07608696, -0.40217391],
       [-0.02173913, -0.19565217,  0.47826087, -0.95652174],
       [-0.05434783, -0.23913043, -0.05434783, -0.14130435]])
```

Pour vérifier, on multiplie à gauche par la matrice A initiale :

```
print(numpy.dot(A,X))
--> array([  1.00000000e+00,  -2.00000000e+00,   3.00000000e+00,
        -1.11022302e-16])

print(numpy.dot(A,iA))
--> array([[  1.00000000e+00,   0.00000000e+00,  -1.11022302e-16,
         -2.49800181e-16],
       [  0.00000000e+00,   1.00000000e+00,   1.11022302e-16,
         -2.22044605e-16],
       [  7.63278329e-17,   0.00000000e+00,   1.00000000e+00,
          0.00000000e+00],
       [  6.93889390e-18,   2.77555756e-17,  -5.55111512e-17,
          1.00000000e+00]])
```

Les erreurs d'arrondis sont bien visibles sur les éléments initialement nuls. Pour réduire ces erreurs, il exise une méthode simple consistant à évaluer l'erreur δX en résolvant l'équation :

$$A\delta X = A(X + \delta X) - B \tag{6}$$

où $X + \delta X$ désigne la solution approchée obtenue précédemment.

```
(dX,diA) = elimination_gauss_jordan(A,numpy.dot(A,X)-B)
X -= dX


print(numpy.dot(A,X))
--> array([ 1., -2.,  3.,  0.])
```

2.c. Complexité

Considérons le cas général de l'élimination pour le système $AX = B$. Pour simplifier, négligeons le temps nécessaire à la recherche du pivot et à l'échange des deux lignes. Pour chaque colonne traitée, la division de la ligne du pivot par a_{jj} comporte $N + M$ divisions. Le traitement de chacune des $N - 1$ lignes nécessite $2(N + M)$ opérations (une multiplication et une soustraction pour chaque élément). On a ainsi pour chaque colonne $N + M + 2(N + M)(N - 1) = (N + M)(2N - 1)$ opérations, soit $2(N + M)N$ opérations lorsque N est grand. Finalement, le nombre d'opérations pour l'élimination complète est $2N^2(N + M)$. Par exemple, pour la résolution d'un seul système avec $M = 1$, il y a $2N^3$ opérations. L'inversion de la matrice nécessite $4N^3$ opérations.

3. Méthode d'élimination de Gauss

3.a. Algorithme

La méthode d'élimination de Gauss consiste à transformer la matrice A de manière à obtenir une matrice triangulaire supérieure :

$$\begin{pmatrix} a'_{00} & a'_{01} & a'_{02} & a'_{03} \\ 0 & a'_{11} & a'_{12} & a'_{13} \\ 0 & 0 & a'_{22} & a'_{23} \\ 0 & 0 & 0 & a'_{33} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b'_0 \\ b'_1 \\ b'_3 \\ b'_3 \end{pmatrix} \tag{7}$$

Pour ce faire, on procède comme dans la méthode de Gauss-Jordan mais en transformant, pour chaque colonne, seulement les lignes situées sous la diagonale, ce qui réduit par deux le nombre d'opérations à effectuer pour chaque colonne. Le système obtenu est résolu par substitution en partant de la dernière équation puis en remontant (opération aussi appelée substitution rétrograde) :

$$x_3 = \frac{b'_3}{a'_{33}} \tag{8}$$

$$x_i = \frac{1}{a'_{ii}} \left(b'_i - \sum_{j=i+1}^{N-1} a'_{ij} x_j \right) \tag{9}$$

3.b. Implémentation

La fonction `elimination_gauss(A,B)` effectue la réduction pour une matrice A carrée et un matrice B contenant éventuellement plusieurs colonnes.

```
def elimination_gauss(A,B):
    A=A.copy()
    B=B.copy()
    (n1,n2)=A.shape
    if n1!=n2:
        raise Exception('Matrice A non carrée')
    N=n1
    s=B.shape
    if len(s)==1:
        n1=s[0]
        n2 = 1
    else:
        (n1,n2)=s
    if N!=n1:
        raise Exception('Matrice B : nombre de lignes incorrect')
    for j in range(N):
        i_pivot = j
        maximum = abs(A[i_pivot,j])
        for i in range(j+1,N):
            if abs(A[i,j]) > maximum:
                maximum = A[i,j]
                i_pivot = i
```

```
        if maximum==0:
            raise Exception('Matrice A singulière')
        if i_pivot!=j:
            temp = A[j].copy()
            A[j] = A[i_pivot].copy()
            A[i_pivot] = temp
            temp = B[j].copy()
            B[j] = B[i_pivot].copy()
            B[i_pivot] = temp
        f = 1.0/A[j,j]
        A[j] *= f
        B[j] *= f
        for i in range(j+1,N):
            g = A[i,j]
            A[i] -= g*A[j]
            B[i] -= g*B[j]
    B[N-1] = B[N-1]/A[N-1,N-1]
    for i in range(N-2,-1,-1):
        j=i+1
        somme = A[i,j]*B[j]
        for j in range(i+2,N):
            somme += A[i,j]*B[j]
        B[i] = (B[i]-somme)/A[i,i]
    return (B)
```

Exemple :

```
A=numpy.array([[5,4,-2,1],[-3,2,0,-5],[3,-5,2,0],[2,-3,0,1]],dtype=numpy.float64)
B=numpy.array([1,-2,3,0],dtype=numpy.float64)
X = elimination_gauss(A,B)


print(X)
--> array([ 0.52173913,  0.43478261,  1.80434783,  0.26086957])

print(numpy.dot(A,X))
--> array([  1.00000000e+00,  -2.00000000e+00,   3.00000000e+00,
         1.11022302e-16])
```

Pour améliorer la précision, on pourra bien sûr utiliser la méthode montrée plus haut.

3.c. Complexité

4. Décomposition LU

4.a. Algorithme

La décomposition LU consiste à décomposer la matrice A en produit d'une matrice triangulaire inférieure L (lower) et d'une matrice triangulaire supérieure U (upper) :

$$A = LU \tag{10}$$

Pour une matrice $4x4$, on a :

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} \alpha_{00} & 0 & 0 & 0 \\ \alpha_{10} & \alpha_{11} & 0 & 0 \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & 0 \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix} \begin{pmatrix} \beta_{00} & \beta_{01} & \beta_{02} & \beta_{03} \\ 0 & \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & 0 & \beta_{33} \end{pmatrix} \tag{11}$$

Une fois cette décomposition effectuée, on peut efficacement obtenir la solution du système $AX = b$ pour un second membre b quelconque en procédant en deux étapes. On résout tout d'abord $Ly = b$ par substitution directe :

$$y_0 = \frac{b_0}{\alpha_{00}} \tag{12}$$

$$y_i = \frac{1}{\alpha_{ii}} \left(y_i - \sum_{j=0}^{i-1} \alpha_{ij} y_j \right) \text{ pour } i = 1, 2, \ldots, N-1 \tag{13}$$

La seconde étape est la résolution de $Ux = y$ par substitution rétrograde :

$$x_{N-1} = \frac{y_{N-1}}{\beta_{N-1,N-1}} \tag{14}$$

$$x_i = \frac{1}{\beta_{ii}} \left(y_i - \sum_{j=i+1}^{N-1} \beta_{ij} x_j \right) \text{ pour } i = N-2, N-3, \ldots, 0 \tag{15}$$

La décomposition LU est obtenue par l'algorithme de Crout, qui procède colonne par colonne en transformant la matrice A (comme les algorithmes d'élimination) de manière à obtenir une matrice finale dont la partie triangulaire inférieure est L, et la partie triangulaire supérieure est U.

Lorsqu'on traite la colonne j (les colonnes 0 à $j-1$ étant déjà transformées), on commence par calculer les éléments au dessus de la diagonale (c'est-à-dire les β_{ij}) avec la relation suivante :

$$\beta_{ij} = a_{ij} - \sum_{k=0}^{i-1} \alpha_{ik} \beta_{kj} \text{ pour } i = 0, 1, \ldots, j \tag{16}$$

Le calcul de β_{ij} fait appel aux éléments α_{ik} de la ligne i et aux éléments β_{kj} de la colonne j déjà calculés (et stockés dans le même tableau) :

$$\begin{array}{ccccc} & & & & \beta_{0j} \\ & & & & \ldots \\ & & & & \beta_{i-1,j} \\ \alpha_{i0} & \ldots & \alpha_{i,i-1} & \ldots & \beta_{ij} \end{array} \tag{17}$$

Le dernier élément calculé est l'élément diagonal :

$$\beta_{jj} = a_{jj} - \sum_{k=0}^{j-1} \alpha_{ik} \beta_{kj} \tag{18}$$

La seconde étape consiste à calculer des éléments (temporaires) situés sous la diagonale, avec la relation :

$$\beta'_{ij} = a_{ij} - \sum_{k=0}^{j-1} \alpha_{ik}\beta_{kj} \text{ pour } i = j+1, j+2, \ldots, N-1 \tag{19}$$

Cette relation est similaire à la précédente, mais fait appel à tout la colonne au dessus de la diagonale déjà calculée. Le pivot est l'élément parmi β_{jj} et les β'_{ij} dont la valeur absolue est la plus grande. Soit j_p l'indice de ligne du pivot. Cette ligne est échangée avec la ligne j. On procède finalement au calcul des éléments α_{ij} situés dans la colonne j sous la diagonale de la manière suivante :

$$\alpha_{ij} = \frac{\beta'_{ij}}{\beta_{jj}} \text{ pour } i = j+1, j+2, \ldots, N-1 \tag{20}$$

où l'élément diagonal β_{jj} correspond au pivot sélectionné précédemment. La division se fait ainsi avec l'élément dont la valeur absolue est la plus grande, ce qui assure la stabilité de l'algorithme.

Les échanges de ligne doivent être mémorisés pour être pris en compte lors de la résolution de $Ly = b$. En effet, la décomposition LU obtenue est celle d'une matrice A dont certaines lignes ont été permutées, et les mêmes permutations doivent être effectuées sur la matrice b.

Le déterminant de la matrice A est le produit des éléments diagonaux β_{ij}, ou son opposé si le nombre de permutations de lignes est impair.

4.b. Implémentation

Nous implémentons la décomposition LU sous forme d'une classe, qui stocke la décomposition afin d'effectuer ultérieurement des résolutions pour différents second membres. Le constructeur `__init__(A)`, auquel ont fournit la matrice A, effectue sa décomposition LU en opérant sur une copie de la matrice A d'origine. Tous les calculs se font dans cette matrice, notée `self.A`. Cette matrice contient à la fin les éléments de L et de U. Pour mémoriser les permutations, on utilise un tableau à N éléments nommé `self.index` qui contient initialement les valeurs $0, 1, .., N-1$. À chaque permutation de ligne, on effectue la même permutation dans ce tableau. On met à jour aussi une variable `self.d` valant 1 si le nombre de permutations est pair, -1 s'il est impair, qui servira pour le calcul du déterminant.

La fonction de classe `resoudre(b)` effectue la résolution du système pour une matrice colonne b donnée. Elle commence par ordonner les éléments de cette matrice en utilisant le tableau `self.index`, avant d'effectuer une substitution directe pour obtenir la matrice colonne y, suivie d'une substitution rétrograde permettant d'obtenir la solution x.

La fonction `ameliorer(b,x)` effectue une amélioration de la précision en résolvant à nouveau l'équation pour un second membre égal au résidu :

La fonction `inverse()` calcule l'inverse de la matrice A. Il s'agit d'une transcription de fonction de résolution, avec une matrice B égale à l'identité.

La fonction `det()` calcule de déterminant.

```
class DecompositionLU:
    def __init__(self,A):
        self.Ai = A
```

```
        self.A = A.copy()
        (n1,n2) = A.shape
        if n1!=n2:
            raise Exception('Matrice A non carrée')
        self.N=N=n1
        self.index = numpy.arange(N)
        self.d = 1
        for j in range(N):
            for i in range(j+1):
                somme = self.A[i,j]
                for k in range(i):
                    somme -= self.A[i,k]*self.A[k,j]
                self.A[i,j] = somme
            maximum = abs(somme)
            i_max = i
            for i in range(j+1,N):
                somme = self.A[i,j]
                for k in range(j):
                    somme -= self.A[i,k]*self.A[k,j]
                    self.A[i,j] = somme
                if abs(somme)>maximum:
                    maximum = somme
                    i_max = i
            if j!=i_max:
                (self.index[j],self.index[i_max])=(self.index[i_max],self.index[j])
                temp = self.A[j].copy()
                self.A[j] = self.A[i_max].copy()
                self.A[i_max] = temp
                self.d = -self.d
            for i in range(j+1,N):
                self.A[i,j] = self.A[i,j]/self.A[j,j]

    def resoudre(self,b):
        N=self.N
        self.b = b.copy()
        for i in range(N):
            self.b[i] = b[int(self.index[i])]
        y = self.b.copy()
        for i in range(1,N):
            somme = self.b[i]
            for j in range(i):
                somme -= self.A[i,j]*y[j]
            y[i] = somme
        x = numpy.zeros(self.N)
        x[N-1] = y[N-1]/self.A[N-1,N-1]
        for i in range(N-2,-1,-1):
            somme = y[i]
            for j in range(i+1,N):
```

```
                somme -= self.A[i,j]*x[j]
            x[i] = somme/self.A[i,i]
        return x

    def ameliorer(self,b,x):
        return x+self.resoudre(numpy.dot(self.Ai,x)-b)

    def inverse(self):
        N=self.N
        I = numpy.identity(N,dtype=numpy.float64)
        B = numpy.zeros((N,N),dtype=numpy.float64)
        for i in range(N):
            B[i] = I[int(self.index[i])]
        y = B.copy()
        for i in range(1,N):
            somme = B[i]
            for j in range(i):
                somme -= self.A[i,j]*y[j]
            y[i] = somme
        x = numpy.zeros((N,N))
        x[N-1] = y[N-1]/self.A[N-1,N-1]
        for i in range(N-2,-1,-1):
            somme = y[i]
            for j in range(i+1,N):
                somme -= self.A[i,j]*x[j]
            x[i] = somme/self.A[i,i]
        return x

    def det(self):
        d = self.d
        for i in range(self.N):
            d *= self.A[i,i]
        return d
```

On remprend l'exemple précédent :

```
A=numpy.array([[5,4,-2,1],[-3,2,0,-5],[3,-5,2,0],[2,-3,0,1]],dtype=numpy.float64)
B=numpy.array([1,-2,3,0],dtype=numpy.float64)
LU = DecompositionLU(A)
X=LU.resoudre(B)



print(X)
--> array([ 0.52173913,  0.43478261,  1.80434783,  0.26086957])

print(numpy.dot(A,X))
--> array([  1.00000000e+00,  -2.00000000e+00,   3.00000000e+00,
        -4.99600361e-16])
```

```
iA=LU.inverse()
det=LU.det()
import scipy.linalg


print(numpy.dot(iA,A))
--> array([[  1.00000000e+00,  -1.38777878e-17,   0.00000000e+00,
          4.16333634e-17],
       [  0.00000000e+00,   1.00000000e+00,  -2.77555756e-17,
          0.00000000e+00],
       [  4.44089210e-16,   4.44089210e-16,   1.00000000e+00,
          0.00000000e+00],
       [  5.55111512e-17,   5.55111512e-17,   1.38777878e-17,
          1.00000000e+00]])

print(det)
--> -183.99999999999997

print(scipy.linalg.det(A))
--> -184.0
```