Résolution numérique de systèmes linéaires

1 Introduction

Dans le chapitre précédent, nous avons étudié comment il était possible d'approcher une racine d'une fonction grâce à différentes méthodes numériques. Il est possible d'étendre ce genre de méthode à des systèmes d'équations, mais cela présente des difficultés importantes. Dans ce chapitre, nous nous intéresserons au cas, plus simple, de la résolution d'un système d'équations *linéaires*.

Ce problème n'en est pas moins assez général, car même si un problème scientifique est non-linéaire, il n'est pas rare que l'on puisse le « linéariser » au voisinage d'un point intéressant, et les méthodes que nous verrons ici sont donc très souvent utiles.

On cherchera donc, dans ce chapitre, à résoudre un système dit *de Cramer*, tel que :

$$\begin{cases} -4a+4b+8c-8d=12 \\ -a+b+2c+2d=11 \\ a-e=-6 \\ -3a-6c-3e=-12 \\ -a-c-2d+2e=5 \end{cases}$$

c'est-à-dire déterminer les a, b, c, d et e qui satisfassent aux cinq équations précédentes.

Un tel problème peut être réécrit sous une forme matricielle, de la forme $A \cdot X = B$, où A est une matrice carrée, X un vecteur contenant les différentes inconnues, et B un vecteur également. Par exemple, le problème suivant est équivalent à l'équation matricielle :

$$\begin{bmatrix} -4 & 4 & 8 & -8 & 0 \\ -1 & 1 & 2 & 2 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ -3 & 0 & -6 & 0 & -3 \\ -1 & 0 & -1 & -2 & 2 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} 12 \\ 11 \\ -6 \\ -12 \\ 5 \end{bmatrix}$$

On cherche donc à déterminer un X qui vérifie $A \cdot X = B$. Dans le cas où A est une matrice inversible, il existe un unique X, égal à $A^{-1}B$, qui convienne. Si A n'est pas inversible, il peut n'y avoir aucune solution, ou une infinité de solutions au problème.

Pour déterminer X (on s'attachera particulièrement au cas où A est inversible et la solution unique), nous allons utiliser des méthodes dites *réduction de Gauss-Jordan*, ou *méthode du « pivot de Gauss »*. Elles consistent à effectuer des opérations sur les lignes d'une matrice dans le but de lui donner une forme particulière.

Dans un premier temps, nous présenterons les différentes opérations que l'on peut effectuer, et une façon de les écrire en Python. Dans un second temps nous étudierons les réductions proprement dites, et leurs différents usages, qui ne sont pas limités à la résolution de systèmes linéaires.

2 Opérations sur les matrices carrées

2.1 Rappels sur la manipulation de matrices

Nous utiliserons, dans ce cours, principalement des `numpy.array` pour représenter vecteurs et matrices, car ils permettent des manipulations aisées.

On définira par exemple la matrice A de l'exemple de la façon suivante :

```
A = numpy.array( [ [ -4,  4,  8, -8,  0 ],
                   [ -1,  1,  2,  2,  0 ],
                   [  1,  0,  0,  0, -1 ],
                   [ -3,  0, -6,  0, -3 ],
                   [ -1,  0, -1, -2,  2 ] ] )
```

Rappelons que `A[i, j]` désigne alors l'élément situé à la i^e ligne et la j^e colonne de la matrice A,

Pour obtenir la sous-matrice constitué des lignes comprises entre i_1 (incluse) et i_2 (exclue) et des colonnes comprises entre j_1 (incluse) et j_2 (exclue), on peut utiliser un « slice » et écrire `A[i1:i2, j1:j2]`.

Notamment, `A[i,:]` et `A[:,j]` désignent respectivement la i^e ligne et la j^e colonne de cette même matrice A. Lorsque l'on souhaite désigner une ligne, on écrira plus simplement `A[i]`.

Enfin, `A.shape` permet d'obtenir les dimensions d'un `numpy.array` `A`, sous la forme d'un tuple contenant autant d'éléments qu'il y a de dimensions.

Pour ce qui est du vecteur colonne B, on écrira :

```
B = numpy.array( [ [  12 ],
                   [  11 ],
                   [  -6 ],
                   [ -12 ],
                   [   5 ] ] )
```

Ici, le vecteur colonne B est représenté par un `array` à deux dimensions (n lignes et une unique colonne). On pourrait tout aussi bien le définir comme un tableau à une seule dimension, mais ce serait laisser à `numpy` le soin de déterminer si on parle d'un vecteur-ligne ou vecteur-colonne, ce qui peut parfois poser des problèmes, comme on l'a vu.

2.2 Transvection

L'opération de *transvection* sur une matrice M consiste à ajouter à la i^e ligne α fois la j^e ligne, avec $i \neq j$ et $\alpha \neq 0$. On notera cette transformation

$$\mathscr{L}_i \leftarrow \mathscr{L}_i + \alpha\mathscr{L}_j$$

Cette opération revient à multiplier la matrice M à gauche par la matrice inversible suivante, où le coefficient α (seul coefficient non-nul situé en dehors de la diagonale) est situé sur la i^e ligne et la j^e colonne :

$$\mathscr{T}_{\mathscr{L}_i \leftarrow \mathscr{L}_i + \alpha\mathscr{L}_j} = \mathbb{1} + \alpha E_{i,j} = \begin{bmatrix} 1 & & & & & \\ & \ddots & \alpha & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix}$$

Le déterminant de cette matrice est égal à 1.

On peut écrire cette opération en Python en modifiant un à un tous les coefficients de la ligne i [1], de la sorte :

```
def Transvection(M, i, j, a) :
    nblgns, nbcols = M.shape

    for col in range(nbcols) :
        M[i, col] = M[i, col] + a * M[j, col]
```

Le module `numpy` nous permet cependant d'écrire cette même opération de façon bien plus succinte :

```
def Transvection(M, i, j, a) :
    M[i] = M[i] + a * M[j]
```

Ou bien encore :

```
def Transvection(M, i, j, a) :
    M[i] += a * M[j]
```

Rappelons qu'il n'est pas besoin de retourner quoi que ce soit ici : on effectue une mutation, et non une affectation, aussi les opérations que l'on effectue sur le `numpy.array` désigné par `M` persistent bien lorsque l'on a quitté la fonction.

1. On évitera d'écrire explicitement une multiplication matricielle, qui serait nettement plus coûteuse en terme de calculs !

2.3 Dilatation

L'opération de *dilatation* sur une matrice M consiste à multiplier la i^e ligne par un coefficient α non nul. On notera cette transformation

$$\mathscr{L}_i \leftarrow \alpha\mathscr{L}_i$$

Cette opération revient à multiplier la matrice M à gauche par la matrice inversible diagonale suivante, où le coefficient α est situé à l'intersection de la i^e ligne et la i^e colonne :

$$\mathscr{T}_{\mathscr{L}_i \leftarrow \alpha\mathscr{L}_i} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & \alpha & & \\ & & & & 1 & \\ & & & & & \ddots \\ & & & & & & 1 \end{bmatrix}$$

Le déterminant de cette matrice est égal à α.

On peut écrire cette opération en Python en itérant sur les colonnes de la matrice, de la façon suivante :

```
def Dilatation(M, i, a) :
    nblgns, nbcols = M.shape

    for col in range(nbcols) :
        M[i, col] = a * M[i, col]
```

Mais cette fois encore, `numpy` fournit des écritures plus courtes pour cette même opération de dilatation :

```
def Dilatation(M, i, a) :
    M[i] = a * M[i]
```

```
def Dilatation(M, i, a) :
    M[i] *= a
```

On pourrait par ailleurs définir la dilatation à partir de la fonction réalisant une transvection :

```
def Dilatation(M, i, a) :
    Transvection(M, i, i, a-1)
```

2.4 Permutation

L'opération de *permutation* sur une matrice M consiste à échanger les i^e et j^e lignes. On notera cette transformation

$$\mathscr{L}_i \leftrightarrow \mathscr{L}_j$$

Cette opération revient à multiplier la matrice M à gauche par la matrice inversible suivante, correspondant à une matrice identité où les i^e et j^e lignes ont été échangées :

$$\mathscr{T}_{\mathscr{L}_i \leftrightarrow \mathscr{L}_j} = \begin{bmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & & 1 & & & \\ & & & & 1 & & & & \\ & & & 1 & & 0 & & & \\ & & & & & & 1 & & \\ & & & & & & & \ddots & \\ & & & & & & & & 1 \end{bmatrix}$$

Le déterminant de cette matrice est égal à -1.

On peut écrire cette opération en Python de la façon suivante :

```
def Permutation(M, i, j) :
    nblgns, nbcols = M.shape

    for col in range(nbcols) :
        M[i, col], M[j, col] = M[j, col], M[i, col]
```

On pourrait penser que cette solution pourrait convenir

```
def Permutation(M, i, j) :
    M[i], M[j] = M[j], M[i]  # ATTENTION, INCORRECT
```

Seulement, ce n'est pas le cas, et on obtient une copie de la j^e ligne sur la i^e ligne à la place. En effet, lorsque l'on écrit `M[i]`, pour éviter de recopier inutilement des données en mémoire, `numpy` fournit ce qu'il appelle une « vue » de la ligne, qui n'est pas indépendante de `M`, ce qui est ici un problème.

Décomposons l'instruction `M[i], M[j] = M[j], M[i]` en trois temps afin de mieux comprendre ce qui se passe.

Python commence par évaluer les termes à droite du signe égal :

```
tmp1 = M[j]
tmp2 = M[i]
```

Puis il effectue la première mutation, qui se passe bien :

```
M[i] = tmp1
```

Et enfin la seconde mutation :

```
M[j] = tmp2
```

Et c'est lors de cette dernière étape que le problème survient : la première mutation a modifié le `numpy.array` désigné par `M`, *mais également ce que désigne* `tmp2` *qui n'est qu'une vue de* `M`. La ligne désignée par `tmp2` est, après la première mutation, une copie de la j^e ligne. La dernière instruction n'a donc pas d'effet !

Il faut donc ruser et provoquer une copie de `M[j]`, ce qui peut être réalisé de différentes façon, en demandant explicitement une copie [2] :

```
def Permutation(M, i, j) :
    M[i], M[j] = M[j], M[i].copy()
```

2.5 Recherche du plus grand coefficient

Afin d'implémenter la méthode du pivot de Gauss, nous aurons besoin d'une dernière opération. Parmi les termes de la i^e colonne, situés sur la diagonale ou en-dessous, lequel est le plus grand en valeur absolue ?

C'est une variante d'un algorithme que nous avons déjà rencontré. On parcourt le morceau de colonne qui nous intéresse, en mémorisant la plus grande valeur absolue rencontrée, et la ligne correspondante :

```
def LignePlusGrand(M, i) :
    nblgns, nbcols = M.shape

    plusGrand = abs( M[i,i] )
    lignePlusGrand = i

    for lgn in range(i+1, nblgns) :
        if abs( M[lgn, i] ) > plusGrand :
            plusGrand = abs( M[lgn, i] )
            lignePlusGrand = lgn

    return lignePlusGrand
```

On peut cependand parvenir exactement au même résultat en une seule instruction en

2. On peut aussi effectuer un quelconque opération sur `M[i]` pour ce même résultat, par exemple en écrivant `M[i], M[j] = M[j], 0+M[i]`.

utilisant les possibilités offerte par numpy :

```
def LignePlusGrand(M, i) :
    return i + numpy.argmax( numpy.abs( M[i:,i] ) )
```

2.6 Sans numpy

Il n'est pas indispensable d'utiliser numpy pour représenter des matrices, on peut très bien le faire avec des listes de listes. Dans cette situation, il faudra réécrire les quatre opérations élémentaires précédentes. Les compréhensions de listes permettent de simplifier l'écriture de ces fonctions [3] :

```
def Transvection(M, i, j, a) :
    M[i] = [ M[i][k] + a*M[j][k] for k in range(len(M[i])) ]

def Dilatation(M, i, a) :
    M[i] = [ a*M[i][k] for k in range(len(M[i])) ]

def Permutation(M, i, j) :
    M[i], M[j] = M[j], M[i]     # Pas de problème avec une liste !

def LignePlusGrand(M, i) :
    nblgns = len(M)

    # On initialise la recherche avec le terme sur la diagonale
    plusGrand = abs( M[i][i] )
    lignePlusGrand = i

    # Puis on cherche un éventuel plus grand terme en-dessous
    for lgn in range(i+1, nblgns) :
        if abs( M[lgn][i] ) > plusGrand :
            plusGrand = abs( M[lgn][i] )
            lignePlusGrand = lgn

    return lignePlusGrand
```

La dernière fonction peut être écrite de façon plus courte et plus efficace, quoique moins lisiblement :

```
def LignePlusGrand(M, i) :
    return i + min(enumerate(abs(l[i]) for l in M[i:]), key=lambda x:x[1])[0]
```

3. Ces fonctions ne sont données qu'à titre d'exemple. La dernière fonction notamment, un peu complexe à comprendre, peut être réécrite de façon similaire à la version « longue » de son pendant numpy.

3 Réduction de Gauss-Jordan

3.1 Objectif

On suppose dans un premier temps que la matrice que l'on manipule est inversible. La méthode de la réduction de Gauss-Jordan consiste à utiliser les trois opérations élémentaires décrites dans la section précédente pour transformer la matrice M de façon à lui donner une des trois formes particulières suivantes [4] :

• transformation en une matrice triangulaire supérieure

$$\begin{bmatrix} -4 & 4 & 8 & -8 & 0 \\ -1 & 1 & 2 & 2 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ -3 & 0 & -6 & 0 & -3 \\ -1 & 0 & -1 & -2 & 2 \end{bmatrix} \longrightarrow \begin{bmatrix} -4 & 4 & 8 & -8 & 0 \\ 0 & -3 & -12 & 6 & -3 \\ 0 & 0 & -2 & 0 & -2 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

• transformation en une matrice diagonale

$$\begin{bmatrix} -4 & 4 & 8 & -8 & 0 \\ -1 & 1 & 2 & 2 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ -3 & 0 & -6 & 0 & -3 \\ -1 & 0 & -1 & -2 & 2 \end{bmatrix} \longrightarrow \begin{bmatrix} -4 & 0 & 0 & 0 & 0 \\ 0 & -3 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

• transformation en la matrice identité

$$\begin{bmatrix} -4 & 4 & 8 & -8 & 0 \\ -1 & 1 & 2 & 2 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ -3 & 0 & -6 & 0 & -3 \\ -1 & 0 & -1 & -2 & 2 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

3.2 Obtenir une matrices triangulaire supérieure

On procède colonne par colonne, de la gauche vers la droite. Pour chaque colonne i :

- on s'assure que le terme sur la diagonale, qui nous servira de pivot, est non nul, sinon on échange la ligne i avec une ligne $j > i$ pour y remédier ;
- on utilise des transvections pour éliminer tous les coefficients de la colonne i situés sous la diagonale, en retirant autant de fois que nécessaire la i^e ligne.

En fait, pour des raisons de stabilité numérique, on ne prendra pas n'importe quel coefficient non-nul comme pivot, mais le plus grand coefficient (en valeur absolue). Cela n'est pas toujours le meilleur choix possible, mais c'est généralement un choix pertinent.

4. Dans les deux premières situations, il existe une infinité de matrices équivalentes que l'on pourrait obtenir et qui auraient la forme attendue, les matrices présentées ici sont celles obtenues avec les algorithmes présentés ci-après.

La fonction s'écrit donc de la sorte :

```
def Pivot(M) :
    nblgns, nbcols = M.shape

    for col in range(nbcols) :
        # S'assurer de disposer d'un pivot non-nul
        lgn = LignePlusGrand(M, col)

        if lgn != col :
            Permutation(M, lgn, col)

        # On désignera par pivot le terme sur la diagonale
        pivot = M[col,col]

        # Annuler les coefficients dans la colonne, sous la diagonale
        for lgn in range(col+1, nblgns) :
            if M[lgn, col] != 0 :
                Transvection(M, lgn, col, -M[lgn,col]/pivot)
```

3.3 Obtenir une matrice diagonale

Le principe est exactement le même, mais on annule les coefficients de chaque colonne à la fois en-dessous et au-dessus de la diagonale. Seule la dernière boucle est ici quelque peu modifiée :

```
def Pivot(M) :
    nblgns, nbcols = M.shape

    for col in range(nbcols) :
        # S'assurer de disposer d'un pivot non-nul
        lgn = LignePlusGrand(M, col)

        if lgn != col :
            Permutation(M, lgn, col)

        # On désignera par pivot le terme sur la diagonale
        pivot = M[col,col]

        # Annuler les coefficients dans la colonne, hors diagonale
        for lgn in range(nblgns) :
            if lgn != col and M[lgn, col] != 0 :
                Transvection(M, lgn, col, -M[lgn,col]/pivot)
```

3.4 Obtenir la matrice identité

On ajoute cette fois-ci une dilatation pour obtenir un 1 sur la diagonale, avant d'annuler les autres coefficients :

```
def Pivot(M) :
    nblgns, nbcols = M.shape

    for col in range(nbcols) :
        # S'assurer de disposer d'un pivot non-nul
        lgn = LignePlusGrand(M, col)

        if lgn != col :
            Permutation(M, lgn, col)

        # Placer un 1 sur la diagonale
        Dilatation(M, col, 1/M[col,col])

        # Annuler les coefficients dans la colonne, hors diagonale
        for lgn in range(nblgns) :
            if lgn != col and M[lgn, col] != 0 :
                Transvection(M, lgn, col, -M[lgn,col])
```

3.5 Suppression de calculs inutiles

Les programmes précédents ne sont pas tout à fait aussi efficaces que possible.

En effet, une fois les k premières colonnes traitées, on sait que les coefficients de ces colonnes ne changeront plus. Aussi, par exemple, une transvection

```
    M[i] += a * M[j]
```

effectue pas mal de calculs inutiles, car

```
    M[i, k+1:] += a * M[j, k+1:]
```

serait suffisant !

Cette amélioration divise environ par deux le nombre d'opérations à effectuer, et une implémentation correcte de la méthode devrait en profiter. Toutefois, nous ne nous intéressons ici qu'au principe de la méthode, et ne nous préoccuperons pas de gagner un « simple » facteur deux si cela rend la lecture de l'algorithme un peu moins aisée.

De toute façon, confrontés à un problème réel, on préférera employer les fonctions des bibliothèques telles que numpy, convenablement optimisées, plutôt que des fonctions que l'on écrirait soi-même.

4 Utilisation

4.1 Résoudre un système linéaire

Pour résoudre un système de la forme $A \cdot X = B$ où A est inversible, on utilise la méthode de Gauss-Jordan pour transformer la matrice A en matrice identité, et on effectue les mêmes transformations sur B. Le vecteur X solution est le vecteur obtenu par l'application des transformations sur le vecteur B. Le programme s'écrira de la sorte :

```
def Resoudre(A, B) :
    hA, lA = A.shape
    hB, lB = B.shape
    if hA != lA or hA != hB or lB != 1 :
        raise "Erreur: les dimensions ne sont pas correctes"

    for col in range(lA) :
        # S'assurer de disposer d'un pivot non-nul
        lgn = Largest(A, col)

        if lgn != col :
            Permutation(B, lgn, col)
            Permutation(A, lgn, col)

        # Placer un 1 sur la diagonale
        Dilatation(B, col, 1/A[col,col])
        Dilatation(A, col, 1/A[col,col])

        # Annuler les coefficients dans la colonne, hors diagonale
        for lgn in range(ha) :
            if lgn != col and A[lgn, col] != 0 :
                Transvection(B, lgn, col, -A[lgn,col])
                Transvection(A, lgn, col, -A[lgn,col])

    return B
```

On peut interpréter ceci en terme de multiplications matricielles. Supposons que l'on effectue une série de n transformations élémentaires. Appliquées à l'équation $A \cdot X = B$, cela revient à considérer l'égalité suivante :

$$\mathcal{T}_n \cdot \mathcal{T}_{n-1} \cdot \ldots \cdot \mathcal{T}_2 \cdot \mathcal{T}_1 \cdot A \cdot X = \mathcal{T}_n \cdot \mathcal{T}_{n-1} \cdot \ldots \cdot \mathcal{T}_2 \cdot \mathcal{T}_1 \cdot B$$

Mais si $\mathcal{T}_n \cdot \mathcal{T}_{n-1} \cdot \ldots \cdot \mathcal{T}_2 \cdot \mathcal{T}_1 \cdot A = \mathbb{1}$, alors on a bien

$$X = \mathcal{T}_n \cdot \mathcal{T}_{n-1} \cdot \ldots \cdot \mathcal{T}_2 \cdot \mathcal{T}_1 \cdot B$$

On a donc déterminé ici la solution de notre équation !

Plusieurs remarques sur la fonction précédente. Tout d'abord, elle modifie ses arguments `A` et `B`, ce que l'on préfère éviter en général (en tout cas, il faut clairement le signaler aux personnes qui utiliseront la fonction). Pour éviter cela, on peut effectuer une copie de A et B avant de commencer, mais cela représente un coût en terme de temps de calcul.

Ensuite, on remarquera que les opérations sur B sont effectuées avant celles sur A. Ce n'est pas anodin : pour la dilatation et la transvection, on voit apparaître des coefficients `A[col,col]` et `A[lgn,col]` qui seront modifiés par l'opération sur A. Il convient donc d'effectuer l'opération sur B d'abord (ou de mémoriser les valeurs en question) !

Notons enfin que si A n'est pas inversible, la fonction ne parviendra pas à trouver un pivot non nul pour toutes les colonnes, et s'arrêtera sur une division par zéro lors d'une dilatation.

4.2 Inverser une matrice

Il est bien évident que cela n'a de sens que si la matrice A que l'on cherche à inverser est bien inversible !

Pour obtenir A^{-1}, la méthode est tout aussi simple que pour résoudre un système linéaire : on transforme, par la méthode du pivot de Gauss, la matrice A en une matrice identité, et on effectue les mêmes opérations sur une matrice identité.

En effet, on a $A \cdot A^{-1} = \mathbb{1}$. En appliquant les transformations à cette égalité, on arrive à

$$\mathcal{T}_n \cdot \mathcal{T}_{n-1} \cdot \ldots \cdot \mathcal{T}_2 \cdot \mathcal{T}_1 \cdot A \cdot A^{-1} = \mathcal{T}_n \cdot \mathcal{T}_{n-1} \cdot \ldots \cdot \mathcal{T}_2 \cdot \mathcal{T}_1 \cdot \mathbb{1}$$

Mais si $\mathcal{T}_n \cdot \mathcal{T}_{n-1} \cdot \ldots \cdot \mathcal{T}_2 \cdot \mathcal{T}_1 \cdot A = \mathbb{1}$, alors on a bien

$$A^{-1} = \mathcal{T}_n \cdot \mathcal{T}_{n-1} \cdot \ldots \cdot \mathcal{T}_2 \cdot \mathcal{T}_1 \cdot \mathbb{1}$$

Le programme s'écrira de la sorte :

```
def Inverser(A) :
    h, l = A.shape
    if h != l :
        raise "Erreur: la matrice n'est pas carrée"

    B = numpy.identity(h)

    for col in range(l) :
        [...]

    return B
```

On a omis, ci-dessus, le contenu de la boucle, qui est en tout point identique à celle du programme précédent.

4.3 Calcul d'un déterminant

Pour calculer un déterminant, on rend la matrice A triangulaire supérieure en n'utilisant que des permutations et des transvections.

La matrice triangulaire supérieure obtenue est de la forme $\mathcal{T}_n \mathcal{T}_{n-1} ... \mathcal{T}_2 \mathcal{T}_1 A$.

Le déterminant de cette matrice triangulaire supérieure est aisé à calculer : c'est simplement le produit des termes de sa diagonale ! Mais compte tenu de l'expression ci-dessus, il s'agit aussi du produit du déterminant de A et de chacun des déterminants des matrices $\mathcal{T}_i$. Or, les matrices de transvection ont un déterminant égal à 1, et celles des permutations un déterminant égal à -1. Donc le déterminant de la matrice triangulaire supérieure est simplement $\text{Det}(A) \times (-1)^p$ où p est le nombre de permutations effectuées.

On peut donc calculer le déterminant d'une matrice avec un algorithme tel que :

```
def Det(A) :
    nblgns, nbcols = A.shape

    if nblgns != nbcols :
        raise "Erreur: la matrice n'est pas carrée"

    # det est initialisé à 1
    det = 1

    for col in range(nbcols) :
        # S'assurer de disposer d'un pivot non-nul
        lgn = Largest(A, col)

        if lgn != col :
            Permutation(A, lgn, col)
            # Une permutation de plus, donc det change de signe
            det = -det

        pivot = A[col,col]

        # Annuler les coefficients sous diagonale
        for lgn in range(col+1, nblgns) :
            if A[lgn, col] != 0 :
                Transvection(A, lgn, col, -A[lgn,col]/pivot)

    # On multiplie enfin det par le produit des termes de la diagonale
    for lgn in range(nblgns) :
        det = det * A[lgn, lgn]

    return det
```

4.4 Estimation du coût de ces algorithmes

L'étude de l'efficacité (en terme de temps de calcul, d'utilisation de la mémoire, etc.) d'un algorithme est un aspect important de l'algorithmique. Un formalisme spécifique lui est dédié, et nous y consacrerons ultérieurement un chapitre.

Cependant, profitons de l'occasion qui nous est donnée ici pour estimer le nombre d'opérations qu'aura à effectuer l'interpréteur Python lorsque l'on exécute un des algorithmes précédents. Commençons par nous intéresser au nombre d'opérations nécessaires pour exécuter l'une des quatres fonctions élémentaires sur lesquels ils sont basés :

- une dilatation d'une ligne d'une matrice M requiert autant de multiplications que d'éléments dans une ligne de M ;
- une permutation de deux lignes de M nécessite autant d'opérations d'échange qu'il y a d'éléments sur les lignes de M ;
- une transvection sur M requiert autant d'additions et de multiplications qu'il y a d'éléments sur une ligne de M.

De la même façon, le nombre d'opérations (comparaison, calculs de valeur absolue, affectations) requises pour la recherche du plus grand élément d'une colonne de M augmente, dans le pire des cas, proportionnellement au nombre de lignes de M.

Lorsque l'on traite une colonne d'une matrice carrée de taille n, on effectue une recherche du plus grand élément, éventuellement une permutation, éventuellement une dilatation, et au plus $n-1$ transvections. Pour les grandes valeurs de n, ce sont les transvections qui prendront l'essentiel du temps, et nécessiteront de l'ordre de n^2 opérations.

Pour traiter l'ensemble de la matrice, soit n colonnes, il faudra par conséquent effectuer de l'ordre de n^3 opérations. Traiter une matrice deux fois plus grande pourra donc nécessiter huit fois plus de temps de calcul.

Les trois algorithmes (résolution d'un système linéaire, inversion, calcul de déterminant) nécessite non seulement de diagonaliser ou trigonaliser une matrice, mais également d'effectuer les mêmes calculs sur un vecteur colonne ou sur une seconde matrice. Cela nécessitera de l'ordre de n^2 (dans le cas d'un vecteur colonne) ou n^3 (dans le cas d'une seconde matrice) calculs supplémentaires. Au final, les trois algorithmes effectuent donc un nombre d'opérations qui croît en n^3.

Cela est à comparer à ce qu'il faudrait faire pour déterminer par exemple le déterminant d'une matrice par la formule mathématique originale : cela nécessiterait de l'ordre de $n!$ opérations élémentaires, ce qui est très rapidement bien plus coûteux que le n^3 de la méthode proposée ici.

La méthode présentée ici n'est pas rigoureusement la plus efficace qui soit, mais il n'y a pas, à l'heure actuelle, de méthode fonctionnant en pratique nettement plus rapidement [5].

5. Une méthode proposée par Strassen permet d'inverser une matrice avec un nombre d'opérations élémentaires de l'ordre de $n^{\ln_2(7)} \simeq n^{2.807}$, et des méthodes récentes introduites par Don Coppersmith et Shmuel Winograd permettent même de descendre à une dépendance en n égale à $n^{2.373}$ environ. Toutefois, ces méthodes sont tellement complexes qu'elles ne sont guères utilisées, car il faut des matrices de *très* grande taille pour commencer à en voir les avantages pratiques. Il a même été conjecturé, sans que cela soit encore démontré ou

5 Et pour les matrices non-inversibles ?

5.1 Problèmes rencontrés avec les matrices non-inversibles

Dans la partie précédente, nous nous sommes uniquement intéressé au cas des matrices inversibles [6]. Nous allons à présent voir brièvement ce que cela change pour des matrices non-inversibles.

Lorsque l'on utilise les algorithmes précédents, si A n'est pas inversible, on s'en rend compte aisément : pour une des colonnes au moins, il est impossible de trouver un pivot non nul sous la diagonale, ce qui empêche de poursuivre l'algorithme.

5.2 Calcul du déterminant

L'algorithme le plus simple à modifier pour traiter ce cas est celui du calcul du déterminant : si la matrice n'est pas inversible, son déterminant est nul. Aussi, si l'on tombe sur un pivot nul, on peut interrompre l'algorithme et directement retourner 0.

Il suffit donc d'ajouter deux lignes à la fonction, juste sous la ligne `pivot = A[col,col]`, qui visent à retourner directement 0 si l'on tombe sur un pivot nul :

```
        [...]

        pivot = A[col,col]

        # Si le pivot est nul, la matrice n'est pas inversible, donc
        # son déterminant est nul
        if pivot == 0 :
            return 0

        [...]
```

5.3 Calcul du rang

On peut également vouloir connaître le rang d'une matrice. L'idée est de dire que si l'on ne trouve pas de pivot dans une colonne, on passe simplement à la suivante ! Mais attention, dans la colonne suivante, on n'essaiera pas de placer un pivot sur la diagonale, mais simplement sur la ligne juste en-dessous de celle du dernier pivot !

On introduira donc un nom `nbpivots` désignant le nombre de pivots trouvés à un quelconque instant de l'algorithme.

Il faudra changer un peu la fonction de recherche du plus grand élément, car il n'est plus nécessairement sous la diagonale :

```
def LignePlusGrandBis(M, i, j) :
    return j + numpy.argmax( numpy.abs( M[j:,i] ) )
```

Si l'on ne se préoccupe pas des valeurs qui seront placées au niveau des pivots, ni au-dessus de ces derniers, on peut donc écrire un algorithme tel que celui-ci :

```
def Rang(M) :
    nblgns, nbcols = M.shape

    # On mémorise le nombre de pivots trouvés
    nbpivots = 0

    for col in range(nbcols) :
        # S'assurer de disposer d'un pivot non-nul
        lgn = LignePlusGrandBis(M, col, nbpivots)

        if lgn != nbpivots :
            Permutation(M, lgn, nbpivots)

        pivot = M[col, nbpivots]

        # Si le pivot est nul, on passe à la colonne suivante
        if pivot == 0 :
            continue

        # Annuler les coefficients dans la colonne, sous le pivot
        for lgn in range(nbpivots+1, nblgns) :
            if M[lgn, col] != 0 :
                Transvection(M, lgn, col, -M[lgn,col]/pivot)

        nbpivots = nbpivots + 1

    return nbpivots
```

Seulement, il y a un souci :

```
In []: Rang(numpy.array([ [ 1., 2., 3. ],
                           [ 4., 5., 6. ],
                           [ 7., 8., 9. ] ]))
Out[]: 3
```

La matrice précédente est clairement de rang 2, et pourtant on obtient un rang égal à

qu'une méthode en soit dérivée, qu'il est possible de descendre à n^2.

6. C'est vraisemblablement le seul cas qu'il vous est demandé de connaître

3. Encore une fois, c'est un problème lié aux calculs sur des flottants. En effet, le premier pivot sera le 7 de la première colonne, et 1/7 n'est pas représentable exactement avec les flottants. Il y aura donc une minuscule erreur d'arrodi, ce qui fait que on trouvera un pivot qui n'est pas rigoureusement égal à zéro dans chacune des trois colonnes.

Le problème est particulièrement gênant lorsque l'on calcule le rang d'une matrice, car changer d'un ϵ inifiniment petit l'un des coefficients peut changer le rang, entier, d'une matrice. Le problème n'était pas aussi marqué dans le problème du calcul du déterminant, car si un arrondi donne un coefficient pas tout à fait nul sur la diagonal, le déterminant pourrait effectivement ne pas être nul, mais il restera très faible de par la présence de ce coefficient quasiment nul. L'erreur commise sur le déterminant sera donc modérée.

Encore une fois, il n'est pas souhaitable de tester l'égalité entre un flottant et zéro. On peut essayer d'améliorer un peu l'algorithme précédent en considérant qu'un pivot très proche de zéro est en fait nul, en remplaçant le test de nullité par exemple par

```
        [...]

        # Si le pivot est considéré nul, on passe à la colonne suivante
        if abs(pivot) < 1e-15 :
            continue

        [...]
```

Ce n'est pas une solution parfaite, le rang obtenu n'étant pas toujours le bon. Lorsque les coefficients sont des entiers relatifs, ou des rationnels, on préférera modifier l'algorithme précédent pour qu'il travaille spécifiquement sur de tels nombres, pouréviter les erreurs d'arrondi obtenir un résultat correct dans tous les cas.

On peut poursuivre la transformation de la matrice plus avant, pour obtenir une matrice diagonale, contenant autant de 1 sur la diagonale que le rang de la matrice, puis des zéros. Cela nécessite cependant d'opérer des transformations sur les colonnes, et non pas uniquement des opérations sur les lignes.

Si cette transformation peut avoir des intérêts théoriques en algèbre linéaire, en pratique, c'est souvent nettement moins utile.

5.4 Cas des systèmes linéaires

Revenons enfin un instant sur le cas d'un système linéaire, se mettant sous la forme $A \cdot X = B$, dans la situation où A n'est pas inversible (ce n'est plus, dans ce cas, un système de Cramer).

Il n'existe plus, dans ce cas, un unique X qui soit solution de l'équation $A \cdot X = B$. Il peut y en avoir une infinité, ou bien n'en exister aucun, selon que B appartienne ou non à l'image de A.

Le fait que A ne soit pas inversible ne nous permet pas d'utiliser la réduction présentée précédemment, car dans certaines colonnes, on ne trouvera pas de pivot non nul. On peut néanmoins utiliser une méthode semblable à celle de la détermination du rang (en passant à la colonne suivante lorsque l'on ne trouve pas de pivot) pour transformer la matrice A et lui donner une forme similaire à la suivante :

$$\begin{bmatrix} 1 & 0 & 0 & a_{0,3} & 0 & 0 & 0 & 0 & a_{0,8} \\ & 1 & 0 & a_{1,3} & 0 & 0 & 0 & 0 & a_{1,8} \\ & & 1 & a_{2,3} & 0 & 0 & 0 & 0 & a_{2,8} \\ & & & & 1 & 0 & 0 & 0 & a_{3,8} \\ & & & & & 1 & 0 & 0 & a_{4,8} \\ & & & & & & 1 & 0 & a_{5,8} \\ & & & 0 & & & & 1 & a_{6,8} \\ & & & & & & & & \\ & & & & & & & & \end{bmatrix}$$

C'est-à-dire que l'on tente de transformer la matrice A en une matrice A' telle que la précédente, en faisant apparaître des 1 échelonnés dans chaque colonne, mais lorsque l'on ne trouve pas de pivot dans une colonne, on passe simplement à la colonne suivante.

Si l'on effectue les mêmes opérations sur B que sur A, on obtient un vecteur B', et le système devient donc $A' \cdot X = B'$.

Dès lors, on peut distinguer deux situations particulières :

- si les éléments de B' correspondant aux lignes de A' ne contenant que des zéros ne sont pas nuls, alors le système n'a clairement pas de solution ;
- si ces éléments sont nuls, alors on a une infinité de solutions : en effet, on peut librement choisir chacun des x_i correspondant aux colonnes i de A' pour lesquelles on n'a pu placer un 1 seul dans la colonne, les x_j des autres colonnes étant ensuite déduits de ceux que l'on a choisi grâce à l'équation correspondant à la ligne contenant le pivot de la colonne j.

On ne s'étendra pas davantage sur résolution d'un système de type $A \cdot X = B$ lorsque A n'est pas inversible.

5.5 Cas des systèmes sur-contraints

Profitons de l'occasion de dire un mot sur les systèmes linéaires sur-contraints. Ce sont des systèmes de n équations linéaires à p inconnues, avec $n > p$. De tels systèmes se mettent également sous la forme $A \cdot X = B$ où A est une matrice à n lignes et p colonnes, et B un vecteur colonne à n composantes.

Si les différentes équations sont indépendantes, il n'existe évidemment pas de solution à ce problème. Toutefois, il est fréquent de vouloir chercher une « solution » au sens des

moindres carrés [7], c'est-à-dire le vecteur X pour lequel $\|A \cdot X - B\|^2$ est le plus petit possible.

On peut montrer que cela revient à chercher le X solution du système ${}^tA \cdot A \cdot X = {}^tA \cdot B$.

Comme ${}^tA \cdot A$ est une matrice carré de taille n, et ${}^tA \cdot B$ un vecteur colonne de taille n, résoudre au sens des moindres carrés un système linéaire sur-contraint revient au problème étudié dans ce chapitre ! Notamment, si ${}^tA \cdot A$ est inversible, on est revenu à un système de Cramer que l'on peut résoudre grâce à une réduction de Gauss-Jordan.

6 Pour aller un peu plus loin

6.1 Systèmes triangulaires

Supposons que l'on ait un sysème de la forme $U \cdot X = B$, où U se trouve être une matrice carrée, de taille n, triangulaire supérieure et dépourvue de zéro sur sa diagonale. C'est, de façon évidente, un système de Cramer, qui admet une unique solution X. Cette solution peut être déterminée très simplement par substitution.

Notons dans la suite $\upsilon_{i,j}$ le coefficient situé sur la i^e ligne et la j^e colonne de la matrice U, et x_i et β_i respectivement les i^e coefficients de X et B. On utilisera dans la suite la numérotation mathématique des lignes et des colonnes, pour laquelle les indices sont compris entre 1 et n.

D'après le produit matriciel, $\beta_i = \sum_k \upsilon_{i,k} x_k = \sum_{k \geqslant i} \upsilon_{i,k} x_k$ puisque $\upsilon_{i,k} = 0$ lorsque $i > k$.

De façon évidente, on a donc $x_n = \dfrac{1}{\upsilon_{n,n}} \times \beta_n$. Pour les autres coefficients, on peut écrire

$$x_i = \frac{1}{\upsilon_{i,i}} \left(\beta_i - \sum_{k=i+1}^{n} \upsilon_{i,k} x_k \right)$$

La détermination de x_i ne nécessite que la connaissance de U et des x_j avec $j > i$, aussi peut-on obtenir tous les coefficients x_i en commençant par x_n et en remontant de n jusque 1. C'est une méthode dite de *substitution* que vous avez sans aucun doute déjà utilisée plus d'une fois à la main sur des systèmes linéaires.

Par exemple, dans le cas de ce système,

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

la dernière équation donne immédiatement $z = 4$, puis la seconde conduit à $y = 6$, et enfin la première donne $x = -5$. C'est exactement ce principe qui est mis en œuvre ici.

7. Un exemple très simple est celui d'une régression linéaire : cela consiste à trouver les coefficients α et β tels que la droite $y = \alpha x + \beta$ passe aussi près que possible d'un ensemble de points (x_i, y_i).

Il n'est donc pas nécessaire de transformer A en la matrice identité par une réduction de Gauss-Jordan, la rendre triangulaire supérieure suffit. Il suffit de procéder par substitution ensuite pour obtenir le vecteur X recherché.

Le principe fonctionne également pour un système de la forme $L \cdot X = B$, où L est une matrice carrée, de taille n, triangulaire inférieure sans zéro sur la diagonale.

Notons λ_i, j les coefficients de L. La substitution se fait dans l'autre sens.

On commence par déterminer $x_1 = \dfrac{1}{\lambda_{1,1}} \beta_1$.

Puis on utilise la relation

$$x_i = \frac{1}{\lambda_{i,i}} \left(\beta_i - \sum_{k=1}^{i-1} \lambda_{i,k} x_k \right)$$

pour déterminer les autres coefficients x_i.

6.2 Décomposition LU

Il est possible de montrer que la plupart des matrices carrées peuvent être écrites comme le produit d'une matrice triangulaire inférieure L avec une matrice triangulaire supérieure U.

Supposons par exemple que l'on peut transformer A en une matrice triangulaire supérieure U en effectuant une série d'opérations élémentaires $\mathscr{T}_i$. Toutes les opérations élémentaires $\mathscr{T}_i$ étant inversibles, on peut écrire :

$$\mathscr{T}_1^{-1} \cdot \mathscr{T}_2^{-1} \cdot \ldots \cdot \mathscr{T}_{n-1}^{-1} \cdot \mathscr{T}_n^{-1} \cdot \mathscr{T}_n \cdot \mathscr{T}_{n-1} \cdot \ldots \cdot \mathscr{T}_2 \cdot \mathscr{T}_1 \cdot A = A$$

Posons $M = \mathscr{T}_1^{-1} \cdot \mathscr{T}_2^{-1} \cdot \ldots \cdot \mathscr{T}_{n-1}^{-1} \cdot \mathscr{T}_n^{-1}$. La matrice $\mathscr{T}_n \cdot \mathscr{T}_{n-1} \cdot \ldots \cdot \mathscr{T}_2 \cdot \mathscr{T}_1 \cdot A$ est, par construction, triangulaire supérieure. Si M est triangulaire inférieure, alors on a trouvé une décomposition de type LU pour notre matrice A.

On peut calculer la matrice M en observant que

$$M = \mathscr{T}_1^{-1} \cdot \mathscr{T}_2^{-1} \cdot \ldots \cdot \mathscr{T}_{n-1}^{-1} \cdot \mathscr{T}_n^{-1} = {}^t\left({}^t\mathscr{T}_n^{-1} \cdot {}^t\mathscr{T}_{n-1}^{-1} \cdot \ldots \cdot {}^t\mathscr{T}_2^{-1} \cdot {}^t\mathscr{T}_1^{-1} \right)$$

L'inverse d'une transvection $\mathscr{L}_i \leftarrow \mathscr{L}_i + \alpha \mathscr{L}_j$ est la transvection $\mathscr{L}_i \leftarrow \mathscr{L}_i - \alpha \mathscr{L}_j$. Les transvections $\mathscr{T}_i$ correspondant à des matrices triangulaires inférieures (on travaille toujours avec $j < i$ dans la réduction de Gauss-Jordan), leurs inverses le sont également, et les transposées sont des matrices triangulaires supérieures.

Si toutes les opérations $\mathscr{T}_i$ sont des transvections, alors la matrice M est la transposée d'un produit de matrices triangulaires supérieures, et est donc une matrice triangulaire inférieure. On a bien construit, ainsi, une décomposition LU de la matrice A.

Seulement, si les pivots ont nécessité des permutations de lignes, les choses sont plus complexes. De fait, toute matrice A n'est pas décomposable en un produit d'une matrice triangulaire inférieure avec une matrice triangulaire supérieure.

Toutefois, il existe nécessairement au moins une permutation des lignes de A pour laquelle cette décomposition est possible.

Nous n'entrerons pas dans les détails ici, mais il suffit d'amener les pivots sur les bonnes lignes avant de commencer la transformation de la matrice en matrice triangulaire supérieure avec une réduction de Gauss-Jordan.

Pour toute matrice A, il existe donc une matrice U triangulaire supérieure, une matrice L triangulaire inférieure, et une matrice P de permutations, telle que $P \cdot A = L \cdot U$, ou bien, de façon équivalente, $A = P \cdot L \cdot U$.

6.3 Algorithme de Crout

Les outils de la réduction de Gauss-Jordan ne sont toutefois pas la méthode privilégiée pour obtenir la décomposition LU. Il est plus simple d'utiliser l'algorithme de Crout.

Supposons que l'on cherche L une matrice triangulaire inférieure et U une matrice triangulaire supérieure telles que $A = L \cdot U$. Notons $\lambda_{i,j}$ et $\upsilon_{i,j}$ les coefficients respectifs de chacune de ces matrices, et $\alpha_{i,j}$ ceux de la matrice A.

D'après le produit matriciel, on a

$$\sum_k \lambda_{i,k} \times \upsilon_{k,j} = \alpha_{i,j}$$

Cela nous donne n^2 équations (où n est la taille de la matrice A), pour $2 \times \frac{n(n+1)}{2} = n^2 + n$ inconnues, car $\lambda_i, j = 0$ lorsque $i < j$ et $\upsilon_{i,j} = 0$ lorsque $i > j$.

On a davantage d'inconnues que d'équations, donc nous avons un peu de liberté. On imposera en plus que la matrice triangulaire inférieure n'ait que des 1 sur sa diagonale ($\lambda_{i,i} = 1$). Nous n'avons alors plus que n^2 inconnues pour n^2 équations.

En fait, en prenant dans le bon ordre ces n^2 équations, on peut directement obtenir tous les coefficients $\lambda_{i,j}$ et $\upsilon_{i,j}$!

Il suffit de procéder de la sorte :

- Pour tout j (dans l'ordre croissant) :
  - Pour tout i (dans l'ordre croissant) :

$$\upsilon_{i,j} = \alpha_{i,j} - \sum_{k<i} \lambda_{i,k} \upsilon_{k,j}$$

$$\lambda_{i,j} = \frac{1}{\upsilon_{j,j}} \left(\alpha_{i,j} - \sum_{k<j} \lambda_{i,k} \upsilon_{k,j} \right)$$

Lorsque l'on exécute cette double boucle, les coefficients qui apparaissent du côté droit de chacune des deux équations ont la bonne idée d'avoir tous déjà été calculés, et l'évaluation ne pose donc aucune difficulté.

La fonction réalisant la décomposition LU d'une matrice A par l'algorithme de Crout est donc toute simple :

```
def LU(A) :
    h, l = A.shape
    if h != l :
        raise "Erreur: la matrice n'est pas carrée"

    # On met des zéros au-dessus de la diagonale de L, sous celle de U
    # et des 1 sur la diagonale de L
    L, U = numpy.identity( h ), numpy.zeros( (h,h) )

    for j in range(h) :
        for i in range(h) :
            if i <= j :
                U[i,j] = A[i,j] - sum(L[i,k]*U[k,j] for k in range(i))
            else :
                L[i,j] = A[i,j] - sum(L[i,k]*U[k,j] for k in range(j))
                L[i,j] = L[i,j] / U[j,j]

    return L, U
```

```
In []: A = numpy.array([ [ 1.,  4.,  2.],
  ...:                   [ 3.,  6.,  5.],
  ...:                   [ 0.,  8.,  7.] ] )

In []: L
Out[]:
array([[ 1.        ,  0.        ,  0.        ],
       [ 3.        ,  1.        ,  0.        ],
       [ 0.        , -1.33333333,  1.        ]]),

In []: U
Out[]:
array([[ 1.        ,  4.        ,  2.        ],
       [ 0.        , -6.        , -1.        ],
       [ 0.        ,  0.        ,  5.66666667]])

In []: L.dot(U)
Out[]:
array([[ 1.,  4.,  2.],
       [ 3.,  6.,  5.],
       [ 0.,  8.,  7.]])
```

Cet algorithme ne fonctionne cependant que si la matrice A admet une décomposition

de la forme $L \cdot U$, ce qui n'est pas toujours le cas (dans l'algorithme précédent, la difficulté apparaît sous la forme d'un coefficient $\upsilon_{j,j}$ nul, conduisant à une division par zéro).

L'algorithme peut toutefois être modifié pour trouver une décomposition $P \cdot L \cdot U$, mais les choses étant un plus compliqués, nous ne détaillerons pas cet algorithme [8].

6.4 Décomposition LU et systèmes linéaires

La décomposition LU est d'une grande importance dans de nombreux domaines, notamment celui des systèmes linéaires $A \cdot X = B$. Si la méthode de réduction de Gauss-Jordan (qu'elle utilise une transformation de A en identité, ou en une matrice triangulaire supérieure suivi d'une étape de substitution) permet d'obtenir une bonne approximation de X, si l'on cherche ensuite un X' qui serait la solution d'un système $A \cdot X' = B'$, il faut recommencer la réduction du début.

En effet, la matrice inverse A^{-1} que l'on peut obtenir par une réduction de Gauss-Jordan n'est pas assez précise pour que $A^{-1} \cdot B'$ donne un résultat aussi bon que celui obtenu en appliquant explicitement les étapes de la réduction à B'.

La méthode de réduction de Guass-Jordan n'est donc pas celle privilégiée pour la résolution de systèmes de Cramer en pratique, même si elle garde un intérêt pédagogique important. On préfère trouver une décomposition de A de la forme $P \cdot L \cdot U$. En effet, le système linéaire $A \cdot X = B$ devient $P \cdot L \cdot U \cdot X = B$, soit $L \cdot U \cdot X = P \cdot B$.

On cherche donc dans un premier temps un vecteur Y satisfaisant à $L \cdot Y = P \cdot B$, ce qui peut se faire directement par substitution, puisque L est triangulaire inférieure. Puis on en déduit X en écrivant $U \cdot X = Y$, un système qui là encore peut être résolu directement par substitution, puisque U est triangulaire supérieure !

Une fois la matrice A décomposée sous la forme $P \cdot L \cdot U$ permet donc de résoudre simplement un système quelconque $A \cdot X = B$. Cette méthode est très stable numériquement, et tend même à être un peu plus rapide (d'un coefficient constant) que la réduction de Gauss-Jordan. C'est donc la méthode utilisée dans les algorithmes de résolution numérique, dont ceux fournis par `scipy`.

7 Avec Scipy

Encore une fois, les algorithmes présentés ici le sont dans un cadre pédagogique, afin d'illustrer le principe de la résolution, par exemple, d'un système de Cramer. Dans la pratique, on évitera de reprogrammer des choses qui sont déjà disponible dans un module.

Le module `linalg` de `numpy` (dont nous avons déjà dit qu'il fournissait les fonctions `det` et `inv` donnant respectivement le déterminant et l'inverse de la matrice fournie en argument [9]) contient ce qu'il faut pour résoudre les systèmes linéaires d'équations.

Par exemple, la résolution d'un système de Cramer $A \cdot X = B$ peut être résolu grâce à la fonction `solve` du module `numpy.linalg`, qui prend en argument la matrice A et le vecteur colonne [10] B :

```
In []: A = numpy.array( [ [ 3., 1. ],
  ...:                   [ 1., 2. ] ] )

In []: B = numpy.array( [ [ 9. ],
  ...:                   [ 8. ] ] )

In []: numpy.linalg.solve(A, B)
Out[]:
array([[ 2.],
       [ 3.]])
```

Cette fonction suppose A carrée et inversible. Sinon (notamment si le système est sous ou sur-déterminé), on se reportera à la fonction `lstsq` de ce même module.

Curieusement, même si `numpy` utilise une décomposition LU pour résoudre un système linéaire, il n'y a pas dans `numpy` de fonction fournissant cette décomposition, mais il en existe une dans le module `scipy.linalg`, appelée `scipy.linalg.lu` (il s'agit en fait d'une décomposition PLU).

```
In []: A = numpy.array( [ [ 1., 4., 2. ],
  ...:                   [ 3., 6., 5. ],
  ...:                   [ 0., 8., 7. ] ] )

In []: scipy.linalg.lu(A)
Out[]:
(array([[ 0.,  0.,  1.],
        [ 1.,  0.,  0.],
        [ 0.,  1.,  0.]]),
 array([[ 1.        ,  0.        ,  0.        ],
        [ 0.        ,  1.        ,  0.        ],
        [ 0.33333333,  0.25      ,  1.        ]]),
 array([[ 3.        ,  6.        ,  5.        ],
        [ 0.        ,  8.        ,  7.        ],
        [ 0.        ,  0.        , -1.41666667]]))
```

8. Les plus curieux pourront se reporter aux ouvrages *Numerical Recipies* pour les algorithmes en questions, et se reporter sur les références qui y sont citées pour les détails théoriques.

9. On y trouve également une fonction `matrix_rank` qui retourne le rang d'une matrice avec, là aussi, d'éventuels problèmes de précision, avec un choix d'un ϵ à faire, même si l'algorithme utilisé pour déterminer le rang de la matrice est différent de celui présenté dans ce cours.

10. Il est possible de fournir à la fonction plusieurs vecteurs colonnes B d'un seul coup, sous la forme d'une matrice regroupant les différents vecteurs B_i, le résultat étant alors une matrice regroupant les vecteurs colonnes résultats X_i de chacune des équations $A \cdot X_i = B_i$.