RESOLUTION DE SYSTEMES D'EQUATIONS LINEAIRES

De nombreux problèmes scientifiques se ramènent à la résolution de systèmes d'équations linéaires, au moins après une phase de linéarisation : pour des petits angles, des petits instants etc….
L'algorithme du pivot de Gauss permet de résoudre de tels systèmes (ou montrer qu'ils n'ont pas de solution) et ce de manière automatique et sûre sans intervention de quelconques astuces de calcul et quelles que soient les équations obtenues.
Il est essentiellement basé sur les transvections, même si des transpositions sont parfois nécessaires ainsi que des dilatations selon le probléme soumis.
En effet, l'algorithme permet de répondre à plusieurs questions :

- Résoudre un système de Cramer
- Trouver un paramétrage des solutions d'un systême
- Déterminer l'inverse d'une matrice inversible
- Déterminer le rang d'une matrice
- Calculer le déterminant d'une matrice carrée
- etc….

Nous serons confrontés principalement à deux problèmes :

- **La précision du résultat** : elle dépend de celle des données mais, même avec des données exactes, les erreurs d'arrondis peuvent induire des erreurs plus ou moins importantes dans le résultat
- **La comparaison d'un réel à zéro** : les calculs avec les flottants induisent des erreurs qui peuvent faire apparaitre ou au contraire disparaitre le réel nul. Comparer un coefficient à zéro n'a donc pas grand sens, alors que dans l'algorithme du pivot de Gauss, il est crucial de s'assurer que le pivot en est bien un, c'est-à dire qu'il est non nul.

I) Résolution de AX = Y : principe du pivot

L'algorithme du pivot de Gauss sait résoudre des systèmes généraux à n équations et p inconnues. Cependant, le programme officiel d'informatique de première année se limite à la résolution de systèmes de Cramer c'est-à-dire de systèmes à n équations et n inconnues inversibles.

Le cas des systèmes triangulaires

On a vu en cours de mathématiques que la résolution des systèmes de Cramer triangulaires est très simple : il suffit de reprendre les équations de la dernière à la première en exprimant progressivement chaque nouvelle inconnue rencontrée.

Exemple : $\begin{cases} x + y = 4 \\ 2y + z = 1 \\ z = 3 \end{cases} \Leftrightarrow \begin{cases} x + y = 4 \\ y = -1 \\ z = 3 \end{cases} \Leftrightarrow \begin{cases} x = 5 \\ y = -1 \\ z = 3 \end{cases}$ (Noter les équivalences)

Le but de l'algorithme de résolution d'un système linéaire sera donc de le transformer (de façon équivalente) à un système triangulaire (ou dans le cas général d'un système non nécessairement de Cramer, de le transformer en un systême échelonné).

Méthode du pivot de Gauss

On se donne un système AX = Y.
But: par manipulations élémentaires sur les lignes ou les colonnes, on veut transformer A pour obtenir une matrice échelonnée
On part de A. On considère la première colonne de A:

- Si un des éléments est non nul, par exemple a_{j1} , on permute les lignes L_j et L_1 . Puis sur les lignes L_i , $i \geqslant 2$, de la nouvelle matrice $\begin{pmatrix} a'_{11} & a'_{12} & \cdots \\ a'_{21} & a'_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$, on effectue les transvections : $L_i \leftarrow L_i - \frac{a'_{i1}}{a'_{11}} L_1$. On obtient une matrice du type $\begin{pmatrix} a'_{11} & a'_{12} & \cdots & a'_{1p} \\ 0 & a''_{22} & \cdots & a''_{2p} \\ \vdots & \cdots & \cdots & \vdots \\ 0 & \vdots & \cdots & \cdots \end{pmatrix} = \begin{pmatrix} a'_{11} & \cdots & a'_{1p} \\ (0) & & (A_1) \end{pmatrix}$
- Si tous les a_{j1} étaient nuls : si sur les colonnes il n'y a également que des 0, on avait déjà une matrice du type voulu, sinon, on échange la colonne C_1 avec une colonne qui n'a pas que des 0
- Puis on réitère le procédé avec la seconde colonne en ne touchant plus à la première ligne (ce qui revient à travailler sur A_1).

Après ces opérations sur les lignes ou les colonnes, on obtient bien une matrice échelonnée.

Pour la résolution du système AX = Y, il faut noter que les opérations faites à partir de A seront faites sur Y pour les opérations sur les lignes de A et sur X pour les opérations sur les colonnes de A (traduites sur les lignes de X…)

La comparaison à zéro

Lorsqu'on travaille avec des systèmes à coefficients entiers (voire rationnels en étant prudent sur les opérations), on travaille avec des objets informatiques qui représentent de façon exacte les objets mathématiques correspondants. Il n'en va pas de même lorsque l'on travaille avec des flottants. Par exemple : 12 * (1/3 - 1/4) - 1 donne dans Python, la réponse -2.220446049250313e-16

En pratique :

- les calculs approchés peuvent faire apparaitre des termes petits mais non nuls alors qu'ils devraient pourtant représenter le réel nul. Ceci peut alors transformer en système de Cramer un système qui ne l'était pas ou prendre comme pivot une quantité qui ne devrait pas l'être…
- au contraire, on peut voir apparaitre des coefficients nuls alors que les objets mathématiques qu'ils représentent ne le sont pas.

Ainsi le systéme : $\begin{cases} x + \frac{1}{4}y + z = 0 \\ x + \frac{1}{3}y + 2z = 0 \\ y + 12z = 1 \end{cases}$ n'a pas de solution alors que sa résolution numérique fournira un résultat : (-750599937895082.8 , 4503599627370496.0 , -375299968947541.25)

Inversement, le système suivant $\begin{cases} x + (1 - 10^{15})y + z = 1 \\ x + (1 + 10^{-15})y + 2z = 0 \\ 10^{15}y + z = 0 \end{cases}$ qui, lui, est de Cramer, n'aura pas de résolution numérique après les transvections $L_2 \leftarrow L_2 - L_1$ puis $L_3 \leftarrow L_3 - L_2$ donnera une troisième équation incompatible (car Python assimile $(10^{15} + 10^{-15}) - 10^{15}$ à 0….

On est donc confronté à la difficulté pour Python de tester l'égalité sur les flottants : a == 0 (et de manière générale a == b avec a et b deux flottants…)

On peut toutefois pallier à de grandes familles de soucis en choisissant pour pivot dans une colonne l'élément qui aura la plus grande valeur absolue (ou module), même si des singularités peuvent apparaitre…

Formalisation de l'algorithme

On considère que l'on travaillera toujours avec un système de Cramer. La trigonalisation consiste donc à éliminer des variables dans les équations successives. On va donc faire en sorte qu'après k étapes, pour tout i entre 1 et k, la i-ième variable ait disparu de toutes les équations du système à partir de la (i+1)-ième : ce sera l'invariant de boucle. Ainsi après la (n-1)-ième étape, le système est sous forme triangulaire.

Dans l'algorithme qui suit, on résout le système AX = Y. La ligne L_i désigne à la fois les coefficients de A (qui est un tableau bidimensionnel) et les seconds membres (qui sont dans la matrice colonne Y, unidimensionnelle).

Données : A, Y
n ← taille de A (ou de Y)
pour i de 0 jusqu'à n – 2 **faire**
  Trouver j entre i et n – 1 tel que $|a_{i,j}|$ soit maximal
  Echanger L_i et L_j (dans la matrice et le second membre)
  pour k de i + 1 jusqu'à n – 1 **faire**
    $L_k \leftarrow L_k - \frac{a_{k\,i}}{a_{i\,i}} L_i$

Rechercher j tel que $|a_{i,j}|$ soit maximal a deux objectifs :: s'assurer que le pivot choisi sera non nul et minimiser les erreurs numériques (d'arrondis) dans la suite du calcul.

Arrivé ici, le système est sous forme triangulaire et il n'y a plus qu'à effectuer la phase remontée pour trouver la solution

pour i de n – 1 à 0 par pas de – 1 **faire**
  pour k de i + 1 jusqu'à n – 1 **faire**
    $y_i \leftarrow y_i - a_{i\,k}\, x_k$
  $x_i \leftarrow \frac{y_i}{a_{i\,i}}$

II) Mise en œuvre de l'algorithme du pivot

Découper le travail

On a vu que pour la résolution du système on avait besoin de 3 outils principaux : recherche du pivot, échange de deux lignes (transposition) et tranvections.

Recherche du pivot

```
def recherche_pivot (A, i) :
    n = len(A)
    j = i                                   premier candidat potentiel
    for k in range(i+1, n) :
        if abs(A[k][i]) > abs(A[j][i]) :
            j = k                           nouveau candidat potentiel
    return (j)
```

Echanges de lignes

```
def echange_lignes (A, i, j) :
    nc = len(A[0])
    for k in range(nc) :
        A[i][k] , A[j][k] = A[j][k] , A[i][k]
```

Remarquer que l'appel à la fonction ne fournit pas de réponse mais agit sur la matrice A.

Transvections

```
def transvection_lignes (A, i, j , mu) :
    """ L_i <- L_i + mu * L_j """
    nc = len(A[0])
    for k in range(nc) :
        A[i][k] += mu * A[j][k]) :
```

Remarquer que l'appel à la fonction ne fournit pas de réponse mais agit sur la matrice A.

Comme on utilisera des fonctions qui vont transformer la matrice entrée en argument, on doit d'abord faire une copie de la matrice de départ

Copie de matrice

```
def copie_matrice (A) :
    """ Crée une copie de A """
    n, p = len(A) , len(A[0])
    return ( [ [ A[i][j] for j in range(p) ]  for i in range(n) ] )
```

Algorithme complet

On regroupe dans un programme l'algorithme du pivot en faisant appel aux différentes procédures intermédiaires écrites et en faisant l'hypothèse que l'on n'utilisera le programme que pour des sytèmes de Cramer.

Résolution d'un système de Cramer

```
def resolution (A0, Y0) :                                                    1
    """ Résolution de A0 . X = Y0, avec A0 inversible """                    2
    A , Y  = copie_matrice(A0) , copie_matrice(Y0)                           3
    n = len(A)                                                               4
    assert (len(A[0]) == n)                                                  5
    # Mise sous forme triangulaire                                           6
    for i in range(n) :                                                      7
        j = chercher_pivot(A, i)                                             8
        if  j > i :                                                          9
            echange_lignes (A, i, j)                                         10
            echange_lignes (Y, i, j)                                         11
        for k in range(i+1, n) :                                             12
            x = A[k][i] / float(A[i][i])                                     13
            transvection_lignes (A, k, i, -x)                                14
            transvection_lignes (Y, k, i, -x)                                15
    # Phase de remontée                                                      16
    X = [0.] * n                                                             17
    for i in range(n-1, -1, -1) :                                            18
        X[i] = (Y[i][0] – sum(A[i][j] * X[j]  for j in range(i+1, n) ) ) / A[i][i]   19
    return (X)                                                               20
```

III) Complexité

Au niveau de la boucle : " for i in range(n) : " de la ligne 7, pour chaque i, on a besoin de

- n – i comparaisons pour trouver le pivot (ligne 8)
- éventuellement 2(n + 2) affectations (pour les échanges des lignes 10 et 11)
- (2n + 2) * (n – i) affectations, divisions, multiplications et soustractions (pour les transvections lignes 14 et 15).

Au total pour la triangularisation, on a besoin de $O(n^3)$ opérations élémentaires (et en fait on a de l'ordre de n^3 opérations élémentaires)

Concernant la phase de remontée, elle nécessite de l'ordre de n^2 opérations élémentaires.

IV) Comparaison avec numpy

La bibliothéque numpy possède une fonction résolvant les systèmes : numpy.linalg.solve qui est très performante. Dans le tableau suivant, on compare les temps de résolution de systèmes n × n

n	40	80	160	320	640
résolution	0.014 s	0.096 s	0.701 s	5.411 s	42.378 s
numpy.linalg.solve	0.00065 s	0.0021 s	0.0067 s	0.0342 s	0.214 s