Autour du pivot de Gauss

18 avril 2015

Introduction

Il existe deux types de méthodes de résolution d'un système linéaire $Ax = b$:

- résolution dite directe à l'aide du pivot de Gauss, que nous allons étudier
- les méthodes itératives (ou indirectes) : on part d'un vecteur x_0 et on considère une suite récurrente du type
$$x_{k+1} = Nx_k + c.$$
On approximera la solution par une valeur de la suite (x_k) qui converge vers la solution.

1 Description de l'algorithme du pivot de Gauss

Dans ce texte, on suppose que les systèmes linéaires $AX = b$ sont de Cramer, c'est-à-dire admettent une unique solution. La matrice A est donc inversible. On suppose que A est de taille n de coefficients $a_{i,j}$ et que **attention** comme en Python, les indices commencent à 0.

L'algorithme du pivot de Gauss, consiste à trianguler la matrice A à l'aide d'opérations élémentaires.

1.1 Le cas triangulaire

Si la matrice A est triangulaire, on résoud facilement le système $AX = b$ par «substitutions remontantes». Le système s'écrit
$$\forall i \in [\![0, n-1]\!], \quad \sum_{j=0}^{n-1} a_{i,j}x_j = b_i.$$

On a ainsi :
$$x_{n-1} = \frac{b_{n-1}}{a_{n-1,n-1}} \quad \text{et} \quad \forall i \in [\![0, n-2]\!],\ x_i = \frac{b_i - \sum_{j=i+1}^{n-1} a_{i,j}x_j}{a_{i,i}}.$$

On en déduit le pseudo-code suivant pour résoudre un système triangulaire :

```
Algo: triangle
Données: A matrice triangulaire supérieure inversible et b un vecteur
Résultat: solution de Ax =b
n = longueur de b
x = un tableau de longueur n
x[n-1] = b[n-1]/ a[n-1][n-1]
Pour i de n-2 à 0 (pas de -1)
    s = 0
    Pour j de i+1 à n-1
        s = s + a[i][j]*x[j]
    Fin pour
    x[i] = (b[i] -s)/a[i][i]
Fin Pour
```

1.2 Mise sous forme triangulaire, phase d'élimination

A chaque étape, on obtient un sous-système carré qui a une inconnue de moins.

Un exemple modèle tracé en Python.

Le pseudo-code :

```
Algo: triangulation
Données: A matrice inversible et b un vecteur
Résultat: mise sous forme triangulaire du système Ax =b
n = longueur de b
Pour i de 0 à n-2
    chercher premier indice i0 entre i et n-1 tel que A[i0][i] non nul # pivot
    échanger les lignes d'indice i0 et i pour A et b
    Pour k de i+1 à n-1
        L_k <- L_k - L_i0 *A[k][i0]/A[i0][i0] # pour A et b
     Fin Pour
Fin Pour
```

Quelques justifications :

- comme A est inversible, on est sûr de trouver un pivot non nul à chaque itération
- à chaque étape k, pour tout i compris entre 1 et k, la i-ème variable a disparu de toutes les équations du système à partir de la $(i+1)$-ième. C'est l'invariant de boucle.

Remarque :

- cette méthode d'élimination est extrêmement importante et s'adapte à des matrices rectangulaires, on peut par exemple alors récupérer le rang.
- certaines matrices ne nécessitent pas de recherche de pivot et donc pas d'échanges de lignes, c'est le cas par exemple des matrices à diagonales strictement dominante ou des matrices définies positives. Ces matrices auront donc une décomposition LU (voir plus loin).

1.3 La problématique supplémentaire des flottants

Quel type de données doit-on choisir pour représenter les coefficients du système ? On ne peut pas choisir le type entier puisque des divisions interviennent, on peut ainsi prendre le type flottant. Deux problématiques apparaissent alors :

- la comparaison à zéro, lorsque que l'on recherche un pivot non nul. On rappelle qu'on ne teste jamais l'égalité à zéro d'un flottant.
- les erreurs arrondis, un exemple pour comprendre :

  supposons que les nombres flottants soient codés en base 10 avec une mantisse de 3 chiffres et considérons le système suivant :

$$\left\{\begin{array}{rcrcl} 10^{-4}x & + & y & = & 1 \\ x & + & y & = & 2 \end{array}\right.$$

  Son unique solution est le couple $(x, y) = (1.0001, 0.9999)$.

  En choisissant le nombre 10^{-4} comme pivot, en effectuant la transvection $L_2 \leftarrow L_2 - \frac{L_1}{10^{-4}}$, le système est équivalent à

$$\left\{\begin{array}{rcrcl} 10^{-4}x & + & y & = & 1 \\ & & y(1-10000) & = & 2-10000 \end{array}\right.$$

  Mais $10000 - 1 = 9999$ et comme la mantisse ne possède que 3 chiffres, ce nombre est arrondi à 9990. De même $10000 - 2 = 9998$ est arrondi à 9990. Ainsi $y = \frac{9998}{9999}$ est arrondi à 1 et par suite $10^{-4}x = 1 - y = 0$ donc $x = 0$. On a ici une perte de précision importante.

En divisant par un petit pivot, on a obtenu de grands nombres.

Résolvons maintenant d'une autre façon : on permute les lignes 1 et 2 et on choisit 1 comme pivot. On effectue ensuite la transvection $L_2 \leftarrow L_2 - \frac{10^{-4}}{1} L_1$ et le système est équivalent à

$$\begin{cases} x & + & y & = & 2 \\ & & y(1-0.0001) & = & 1-2\times 0.0001 \end{cases}$$

Mais $1-0.0001 = 0.9999$ est arrondi à 0.9990 et $1-2\times 0.0001 = 0.9998$ est aussi arrondi à 0.9990. Donc y vaut 1 puis en remontant $x = 2-y = 1$. Cette fois-ci le résultat est satisfaisant.

Morale : il vaut mieux perdre des chiffres significatifs pour un nombre petit que pour un nombre grand. C'est pourquoi, on préférera chosir comme pivot «le plus grand des pivots». On utilisera donc l'amélioration suivante où l'on va chercher un pivot maximal. On parle de *pivot partiel* (on parle de pivot total lorsque le pivot est maximal en ligne et en colonne, mais on ne le fera pas ici, car on ne raisonne que sur les lignes).

Remarque : on peut éviter les problématiques de nombres flottants si les coefficients des matrices sont des rationnels ou des éléménts de corps finis. On peut alors représenter les coefficients de manière exacte en utiliser le type `fraction` de Python par exemple ou un logiciel de calcul formel. Mais alors, pourquoi ne choisit-on pas toujours les rationnels au lieu des flottants ? Et bien cela dépend des besoins. Avec des rationnels, on calcule de manière exacte mais lentement et avec des flottants, on calcule très vite mais de manière approchée.

2 Implémentation en Python

2.1 Découpage du travail

Codons tout d'abord les deux opérations élémentaires suivantes :

```
def echange_ligne(A,i,j):
    """Applique l'opération élémentaire Li <-> Lj à la matrice A"""
    n = len(A[0]) # nbre de colonnes de A
    for k in range(n):
        temp = A[i][k]
        A[i][k] = A[j][k]
        A[j][k] = temp
```

Attention cette version ne fonctionne pas pour une matrice colonne B car `B[0]` n'est plus une liste mais un entier. Cela fonctionne si on écrit `B` comme une liste de liste `B = [ [1],[2],[3]]` au lieu de `[1,2,3]`.

```
def echange_ligne_bis(A,i,j):
    """Applique l'opération élémentaire Li <-> Lj à la matrice A"""
    temp = A[i]
    A[i] = A[j]
    A[j] = temp

def transvection(A,i,j,mu):
    """Applique l'opération élémentaire Li <-Li + mu*Lj à la matrice A"""
    # cas où A est une matrice colonne
    if type(A[0]) != list:
        A[i] = A[i] + mu*A[j]
    else:
        n = len(A[0]) # nbre de colonnes de A
        for k in range(n):
            A[i][k] = A[i][k] + mu*A[j][k]
```

Remarque : ces deux procédures suivantes ne renvoient rien, elles modifient la matrice passée en paramètre.

Codons maintenant la fonction qui recherche le pivot maximal.

```
def pivot_partiel(A, j0):
    n = len(A)  # nbre de lignes de A
    imax = j0 # indice de ligne avec pivot max
    for i in range(j0+1, n):
        if abs(A[i][j0]) > abs(A[imax][j0]):
            imax = i
    return imax
```

Écrivons enfin une fonction `triangle` qui résoud les systèmes triangulaires supérieurs.

```
def triangle(A,b):
    """ Renvoie la solution du système Ax =b
    lorsque A est tringulaire supérieure inversible"""
    n =len(b)
    x = [0 for i in range(n)]
    x[n-1] = b[n-1]/A[n-1][n-1]
    for i in range(n-2,-1,-1):
        s = 0
        for j in range(i+1,n):
            s = s + A[i][j]*x[j]
        x[i] = (b[i] - s)/ A[i][i]
    return x
```

2.2 Recoller les morceaux

Puisque nous allons appliquer des échanges et des transvections à la matrice de départ, celle-ci va être modifiée. Nous allons donc faire en premier lieu une copie de notre donnée. Nous allons pour cela utiliser la fonction `copy.deepcopy`. Rappelons que si on pense réaliser une copie de `A0` en effectuant l'affectation `A =A0`, il n'en est rien puisque A et $A0$ sont deux noms correspondant à un même objet (voir exercices).

```
import copy

def solution_systeme(A0,b0):
    """ Renvoie la solution X du système de Cramer A0 X= b0"""
    A,b = copy.deepcopy(A0), copy.deepcopy(b0) # on ne détruit pas les données initiales
    n = len(A)
    # Mise sous forme triangulaire
    for i in range(n-1): # on itère n-1 fois
        i0 = pivot_partiel(A,i)
        echange_ligne_bis(A,i,i0)
        echange_ligne_bis(b,i,i0)
        for k in range(i+1,n):
            x = A[k][i]/A[i][i]
            transvection(A,k,i,-x)
            transvection(b,k,i,-x) # attention b matrice colonne
    # phase de remontée
    return triangle(A,b)
```

Testons avec

$$A = \begin{pmatrix} 2 & 2 & -3 \\ -2 & -1 & -3 \\ 6 & 4 & 4 \end{pmatrix} \quad \text{et} \quad b = \begin{pmatrix} 2 \\ -5 \\ 16 \end{pmatrix}$$

```
>>>A = [ [2,2,-3], [-2,-1,-3], [6,4,4] ]
>>> b = [2,-5,16]
>>> solution_systeme(A,b)
[-14.000000000000036, 21.000000000000046, 4.000000000000007]
```

2.3 Comparaison avec Numpy et un logiciel de calcul formel

La fonction `numpy.linalg.solve` permet de résoudre les systèmes linéaires. Elle utilise des tableaux de type `array`.

```
>>> x = numpy.linalg.solve(A,b)
array([-14.,  21.,   4.])
>>> x[0]
-14.000000000000036
```

3 Complexité temporelle cubique

On essaye d'estimer le nombre d'opérations significatives qui peuvent être des affectations, des comparaisons, des opérations arithmétiques sur les flottants (addition, multiplication, division).

Côut temporel de l'élimination :

On effectue $(n-1)+(n-2)+\cdots+2+1=\frac{(n-1)n}{2}$ transvections sur la matrice A donc de l'ordre de n^2. Chaque transvection côute n opérations flottantes du type $x \leftarrow ax+b$. Il y a donc environ n^3 opérations flottantes du type $x \leftarrow ax+b$.

Pour chercher les pivots, pour i compris entre 0 entre $n-2$, on effectue $n-i$ comparaisons du type $|a|<|b|$, soit un total de $n+(n-1)+\cdots+2$ comparaisons qui est de l'ordre de n^2.

Coût de la remontée :

Il y a deux boucles imbriquées, le coût est seulement quadratique.

Ainsi le coût global est en en $O(n^3)$, on parle de complexité cubique. Il est important de remarquer que c'est la phase d'élimination qui est coûteuse. C'est d'ailleurs sur cette remarque que repose l'intérêt de la décomposition LU d'une matrice, pour résoudre plusieurs fois un système avec même une matrice mais des seconds membres différents.

4 Branching out

1. Le pivot de Gauss s'adapte facilement pour résoudre les problèmes suivants :
   (a) calcul de l'inverse d'une matrice
   (b) calcul de rang d'une matrice à l'aide d'une mise sous-forme échelonnée
   (c) décomposition LU (L comme «lower» et «U») d'une matrice
   (d) calcul de déterminant
2. Mesure de la propagation des erreurs d'arrondi, notion de conditionnement d'une matrice
   Cas des matrices de Hilbert ou de Virginie