Devoir d'informatique à rendre Lundi 11 Mars 2019

Matrices tridiagonales

Une matrice tridiagonale est une matrice carrée de la forme

$$M = \begin{pmatrix} x_0 & y_0 & 0 & \cdots & \cdots & 0 \\ z_0 & x_1 & y_1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \cdots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & z_{n-2} & x_{n-1} & y_{n-1} \\ 0 & \cdots & \cdots & \cdots & z_{n-1} & x_n \end{pmatrix} \in M_{n+1}(\mathbb{R})$$

De telles matrices apparaissent dans divers problèmes concrets : problèmes d'interpolation (splines cubiques), convertisseur analogique-numérique, équations aux dérivées partielles, ... avec des n d'autant plus grands qu'on exige plus de précision. Il est donc intéressant d'avoir une méthode efficace de résolution d'un système linéaire dont la matrice est tridiagonale, efficace dans le sens où la complexité sera meilleure que celle en $O(n^3)$ pour une matrice quelconque.

On cherche donc à résoudre un système linéaire du type MX = Y. Pour cela on cherche une factorisation M = LU où L et U sont triangulaires de la forme

$$L = \begin{pmatrix} 1 & 0 & 0 & \cdots & \cdots & 0 \\ c_0 & 1 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \cdots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & c_{n-2} & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & c_{n-1} & 1 \end{pmatrix} \in M_{n+1}(\mathbb{R}) \quad \text{et} \quad U = \begin{pmatrix} a_0 & b_0 & 0 & \cdots & \cdots & 0 \\ 0 & a_1 & b_1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \cdots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{n-1} & b_{n-1} \\ 0 & \cdots & \cdots & \cdots & 0 & a_n \end{pmatrix} \in M_{n+1}(\mathbb{R})$$

les coefficients $a_0, \dots, a_n, b_0, \dots, b_{n-1}, c_0, \dots, c_{n-1}$ étant à déterminer, une fois donnés $x_0, \dots, x_n, y_0, \dots, y_{n-1}, z_0, \dots, z_{n-1}$.

Une telle factorisation est possible ssi tous les réels δ_k de la suite récurrente suivante sont non nuls :

$$\boldsymbol{\delta_0 = 1 \ , \ \delta_1 = x_0 \ , \ \delta_{k+1} = x_k \delta_k - z_{k-1} y_{k-1} \delta_{k-1} \ , \ 1 \le k \le n}$$

1) Ecrire une fonction **matrice_tridiagonale(X,Y,Z)** qui prend en arguments trois listes de la forme X = $[x_0, \dots, x_n]$, Y = $[y_0, \dots, y_{n-1}]$ et Z = $[z_0, \dots, z_{n-1}]$ et qui retourne la matrice tridiagonale avec les x_i sur la diagonale, les y_i juste au-dessus de la diagonale et les z_i juste en-dessous de la diagonale comme la matrice M ci-dessus.
On en donnera deux versions : l'une où la matrice sera codée comme liste de listes (ligne par ligne) et une seconde où la matrice sera un tableau dimensionnel de type numpy.
Remarque : si *M* est une liste de liste, la commande np.array(M) transforme *M* en tableau numpy, mais il y a des commandes numpy qui permettent de construire très facilement les matrices tridiagonales (voir le mode d'emploi simplifié de numpy).

2) Ecrire une fonction **est_factorisable(M)** qui prend en argument une matrice tridiagonale (codage libre), calcule la liste des δ_k et retourne cette liste et True (respectivement False) si tous les δ_k sont non nuls (resp. si au moins des δ_k est nul).
Les coefficients étant des nombres flottants, la comparaison exacte à 0 sera remplacée par un test du type

$$|\delta_k| < 1E{-}precision$$

Recherche des relations de récurrence

On cherche donc *L* et *U* tels que *M* = *LU*.
On doit avoir

$$L_0(U) = L_0(M) \quad \text{et} \quad L_i(M) = c_{i-1} L_{i-1}(U) + L_i(U) \ , \ 1 \le i \le n$$

où $L_i(M)$ désigne la ligne d'indice *i* de la matrice *M* . L'idée est donc de calculer $L_i(U)$ à partir de $L_{i-1}(U)$.
On a donc $\boldsymbol{L_0(U) = [x_0, y_0, 0, \dots, 0]}$ ce qui donne a_0 et b_0.
et (en soulignant les coefficients sur la colonne d'indice i),

$L_i(U) = [0, \dots, 0, \underline{a_i}, b_i, 0, \dots, 0] = L_i(M) - c_{i-1} L_{i-1}(U)$
$= [0, \dots, z_{i-1}, \underline{x_i}, y_i, 0, \dots, 0] - c_{i-1} [0, \dots, 0, a_{i-1}, \underline{b_{i-1}}, 0, \dots, 0]$

avec $\delta_i = \prod_{k=0}^{i-1} a_k$ qui doit être non nul, donc $a_{i-1} \neq 0$, ce qui donne

$z_{i-1} - c_{i-1} a_{i-1} = 0 \Rightarrow \boldsymbol{c_{i-1} = \frac{z_{i-1}}{a_{i-1}} = z_{i-1} \frac{\delta_{i-1}}{\delta_i}}$ pour $1 \leq i \leq n$

$\boldsymbol{b_i = y_i}$ pour $0 \leq i \leq n - 1$

$\boldsymbol{a_i = x_i - c_{i-1} b_{i-1} = x_i - c_{i-1} b_{i-1}}$ pour $1 \leq i \leq n$

$= x_i - z_{i-1} \frac{\delta_{i-1}}{\delta_i} y_{i-1} = \boldsymbol{\frac{\delta_{i+1}}{\delta_i}}$ pour $0 \leq i \leq n$ vu la formule de récurrence des δ_i .

Cela permet donc de calculer $L_i(U)$ et $L_i(L)$ à partir de $L_i(M)$.

3) Ecrire une fonction **factorisation_LU(M)** qui prend en argument une matrice tridiagonale M. La fonction commence par appeler est_factorisable(M) pour s'assurer que la factorisation est bien possible et pour récupérer la liste des δ_k.
La fonction calcule ensuite à l'aide d'une boucle les listes $[a_0, \ldots, a_n]$, $[b_0, \ldots, b_{n-1}]$ et $[c_0, \ldots, c_{n-1}]$, la i-ième itération calculant c_{i-1}, b_{i-1} et a_i.
On appelle ensuite la fonction tridiagonale pour construire les matrices L et U sous forme de tableaux numpy et on retourne L et U dans cet ordre.

Vérification
Pour vérifier que le programme de factorisation fonctionne correctement, on pourra utiliser la fonction **np.dot(A,B)** qui calcule le produit matriciel ordinaire si les dimensions de A et B autorisent ce produit. On vérifiera donc que np.dot(L,U) = M aux erreurs d'arrondi près.

4) Estimer la complexité de la fonction factorisation en fonction de la taille de M : estimer le nombre de multiplications, de divisions et prouver que la complexité est un $O(n)$.

5) La résolution d'un système linéaire $MX = Z$ se ramène à $LUX = Z$.
On résout d'abord $LY = Z$ puis une fois Y connu, on résout $UX = Y$. Les deux systèmes linéaires sont triangulaires
On donne ci-dessous une fonction qui prend en argument une matrice M triangulaire supérieure (liste de listes) et un vecteur Y (liste) et qui résout le système MX = Y d'inconnue X.

```
1  def resol_triang_sup(M, Y) :
2  |       n = len(M) # .....
3  |       X = [0.]*n  # initialisation du vecteur solution .....
4  |       X[n − 1] = Y[n − 1]/M[n − 1][n − 1]  # .....
5  |       # "remontée" : on calcule les inconnues à partir de X[n − 1]
6  |       for i in range(n − 2, − 1, − 1):  .....
7  |       |       s = 0  # initialisation du calcul de ∑ M_{i,j} X_j , j ≥ i + 1
8  |       |       for j in range(i + 1, n):.....
9  |       |       |       s = s + X[j]*M[i][j].....
10 |       |       X[i] = (Y[i] - s)/M[i][i] .....
11 |       return X
```

a) Justifier le choix des indices, en particuliers lignes 6,8.

b) Justifier que le programme est correct à l'aide de l'invariant de boucle ligne 6,

" les composantes de X[i + 1:n] sont celles de la solution MX = Y "

où X[i + 1:n] est composées des termes de X de l'indice i + 1 inclus à l'indice n exclu.

c) Prouver, en estimant le nombre de multiplications et de divisions flottantes que la complexité d'une résolution d'un système triangulaire est en $O(n^2)$ où n est la taille de M.

6) Ecrire une fonction **resolution_tridiagonale(M,Z)** qui prend en argument une matrice tridiagonale M et un second membre Y sous forme de liste et qui calcule la solution X du système $MX = Z$. La fonction commence par obtenir la factorisation $M = LU$ et appelle ensuite une fonction resol_triang_inf (à écrire) pour $LY = Z$ puis appelle la fonction resol_triang_sup pour $UX = Y$. Si la factorisation $M = LU$ n'est pas possible, on retournera un message d'erreur.

7) Comparaison des temps d'exécution
La commande **np.linalg.solve(M,Y)** résout le système linéaire $MX = Y$, M étant une matrice carrée inversible et X un vecteur de même taille que M.
Pour M tridiagonale, on souhaite comparer les temps d'exécution pour la résolution de $MX = Y$ avec d'une part la fonction resolution_tridiagonale et d'autre part avec np.linalg.solve.

Ecrire une fonction **comparaison_temps(n)** : la fonction génère une matrice tridiagonale de taille $n + 1$ avec des x_i, y_i et des z_i qui sont des nombres aléatoires entre 0 et 1 (utiliser la commande **np.random.rand(p)** pour un vecteur à p composantes ou **np.random.rand(n,p)** pour un tableau bidimensionnel de taille (n,p)). Générer de même un second membre Y. Variante : coefficients dans un segment [a,b].
Pour mesurer le temps, on importe le module time contenant la commande time.clock() . En voici le principe :

```
import time
t0 = time.clock()
resolution_tridiagonale(M, Y)
t1 = time.clock()
np.linalg.solve(M, Y)
t2 = time.clock()
print(t1 − t0, t2 − t1)
```

La fonction retourne la liste [t1 − t0, t2 − t1].

Tester alors la fonction avec des valeurs de n de plus en plus grandes et faire un bilan des appels, éventuellement sous forme d'un graphique.