HAX406X — TP4 : Remontée et Descente

Nous avons vu en cours que la factorisation LU peut être utilisée pour résoudre un système linéaire. Étant donnée une matrice $A \in \mathbb{R}^{n,n}$ inversible admettant une factorisation LU ainsi qu'un seconde membre $b \in \mathbb{R}^n$, on peut reformuler le problème consistant à trouver $x \in \mathbb{R}^n$ tel que

$$Ax = LUx = b$$

comme suit : trouver $x, y \in \mathbb{R}^n$ tels que

$$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

Il s'agit de deux systèmes linéaires associés respectivement à des matrices triangulaire inférieure (pour L) et supérieure (pour U). Pour les résoudre, nous disposons d'algorithmes efficaces : descente (pour les matrices inférieures) et remontée (pour les matrices supérieures). Vous allez par la suite mettre en œuvre l'algorithme de la remontée et vérifier numériquement sa complexité.

1 Algorithme de remontée.

Le but de cette section est d'écrire un script qui, pour une matrice triangulaire supérieure $A \in \mathbb{R}^{n \times n}$ et un second membre $b \in \mathbb{R}^{n \times n}$ donnés, calcule la solution x de $Ax = b$. Évidemment vous n'utiliserez pas la fonction `numpy.linalg.solve()` de résolution de système!). Pour bien comprendre le principe on commence par un exemple simple.

Exercice 1. *Soient* $A = \begin{pmatrix} 1 & 1 & 3 \\ 0 & 2 & 4 \\ 0 & 0 & 4 \end{pmatrix}$ *et* $b = \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix}$. *Calculer à la main le déterminant puis la solution* $x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ *de* $Ax = b$. *Écrire un script qui calcule numériquement la solution. Vérifier le résultat. Indication : vous remarquerez que la matrice* A *étant triangulaire supérieure, la composante* x_3 *se calcule facilement, on en déduit alors* x_2 *puis enfin* x_1 *; c'est le principe de la remontée !*

On va maintenant généraliser cet algorithme à des matrices triangulaires de taille quelconque.

Exercice 2.

1. *En se servant des commandes* `np.random.rand()` *et* `np.triu()` *(vues au TP 2) créer des matrices triangulaires supérieures de taille* $n \geq 1$ *quelconque avec des coefficients compris dans* $[0, 1]$.
2. *Écrire un test pour détecter si la matrice est inversible ou non et l'afficher.*
3. *Calculer la solution* x *de* $Ax = b$ *en procédant comme à l'exercice précédente, c'est-à-dire en remontant (on calcule* x_n *puis* x_{n-1} *etc.) Afficher également si l'un des coefficients diagonaux est trop petit (inférieur à* 10^{-12} *par exemple).*

4. *Calculer ensuite la norme du résidu, c'est-à-dire* $\|Ax - b\|_\infty$.

Exercice 3. *En utilisant le script (ou la fonction) de l'exercice précédent, faire varier* n *entre* 2 *et* 1000. *Pour chaque* n*, créer une matrice triangulaire supérieure* $A \in \mathbb{R}^{n,n}$ *et un vecteur* $b \in \mathbb{R}^n$ *avec des coefficients entre* 0 *et* 1. *Calculer la solution de* $Ax = b$ *puis le résidu* $\|Ax-b\|_\infty$. *Tracer le résidu en fonction de* n*, que remarquez-vous ? Comme* A *et* b *sont choisis aléatoirement, faites plusieurs essais ; vous pouvez aussi tracer sur une même figure les différentes courbes.*

2 Vérification de la complexité de l'algorithme de remontée.

Dans cette section, nous allons nous intéresser au temps nécessaire à la machine pour exécuter la résolution de systèmes $Ax = b$ à l'aide de l'algorithme de remontée, ce qui revient à étudier sa complexité. Pour cela on va utiliser le module `timeit` et la commande `timeit.default_timer()` pour estimer le temps d'exécution.

Exercice 4.

1. *Exécuter le programme précédent pour des matrices de taille* n *variant entre* 2 *et* 1000 *puis chronométrer le temps d'exécution de la remontée. Attention à ne mesurer que le temps d'exécution de l'algorithme de remontée et non la génération des matrices ni des seconds membres.*
2. *Tracer ensuite la courbe du temps d'exécution en fonction de* n*. Faire de même en échelle logarithmique. Quelle est approximativement la pente de la courbe ?*
3. *En déduire la complexité de l'algorithme.*

À noter que la complexité de l'algorithme va dépendre de la manière dont il a été codé. Dans tous les cas, une boucle qui agit sur les lignes est nécessaire, induisant une complexité au moins linéaire donc en $O(n)$. Cependant, on pourrait être tenté de créer une deuxième boucle (à l'intérieur de la première) qui agit sur les colonnes. On obtient ainsi dans ce cas une complexité quadratique (donc en $O(n^2)$). Mais il est plus intéressant d'utiliser à la place un produit matriciel, rendant la complexité linéaire.

3 Algorithme de descente

Pour ceux qui avancent vite, faire la même étude avec l'algorithme de descente. Celui-ci consiste calculer la solution de $Ax = b$ lorsque $A \in \mathbb{R}^{n,n}$ est une matrice triangulaire inférieure et $b \in \mathbb{R}^n$ un vecteur donnés. On remarquera que dans ce cas, c'est la première composante de x qui se calcule facilement, on en déduit alors la deuxième etc. c'est le principe de la descente !

Exercice 5.

1. *En se servant des commandes* `np.random.rand()` *et* `np.tril()` *créer des matrices triangulaires inférieures de taille* $n \geq 1$ *quelconque avec des coefficients compris dans* $[0, 1]$.
2. *Écrire un test pour détecter si la matrice est inversible ou non et l'afficher.*
3. *Calculer la solution* x *de* $Ax = b$ *en descendant. Afficher également si l'un des coefficients diagonaux est trop petit (inférieur à* 10^{-12} *par exemple).*
4. *Calculer ensuite la norme du résidu, c'est-à-dire* $\|Ax - b\|_\infty$.
5. *Faire ensuite varier* n *entre* 2 *et* 1000 *puis tracer sur une première figure le résidu en fonction de* n *; et sur une deuxième figure le temps d'exécution en fonction de* n*. L'algorithme de descente a-t-il la même complexité que l'algorithme de remontée ?*