Introduction

Pour ce devoir, il était demandé d'écrire des fonctions en langage Python permettant de résoudre éfficacement un système d'équations linéaires issu d'un modèle d'éléments appelé `ccore`. Le devoir se divise en deux tâches :

— la résolution d'un système en format plein et
— la résolution d'un système en format creux.

Pour la première partie, deux fonctions ont dû être écrites :

— `LUfactorize`, qui factorise en place la matrice $A \in \mathbb{C}^{n\times n}$ et
— `LUsolve`, qui résout un système linéaire $Ax = b$.

La seconde partie fait la même chose pour une matrice dite « creuse ». Une heuristique de renumérotation est également implémentée pour profiter le plus possible de l'aspect creux de la matrice.

1 Format plein

Dans la partie en format plein du devoir, le but était d'étudier la densité de la factorisation LU en place d'une matrice de faible densité. Pour faire cela, il a fallu écrire une fonction capable de faire cette factorisation : `LUfactorize`. Elle prend en arguments la matrice A et décompose celle-ci en un produit d'une triangulaire supérieure U et d'une matrice triangulaire inférieure à diagonale unité L de sorte à ce que $A = LU$.

Pour résoudre le système, il suffit alors de d'abord calculer $Ly = b$ par substitution avant, puis de résoudre $Ux = y$ par substitution arrière pour trouver la solution x. La fonction `LUsolve` fait ceci.

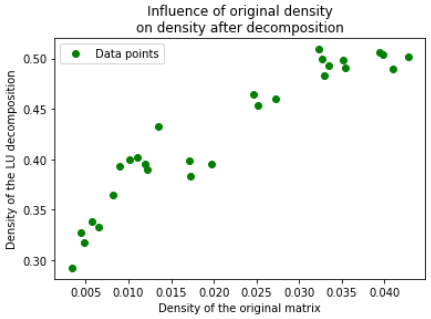
1.1 Étude de la densité de la décomposition LU

Comme la matrice A est issue d'un modèle d'éléments finis, elle contient beaucoup d'entrées nulles. La densité d'une matrice est définie comme

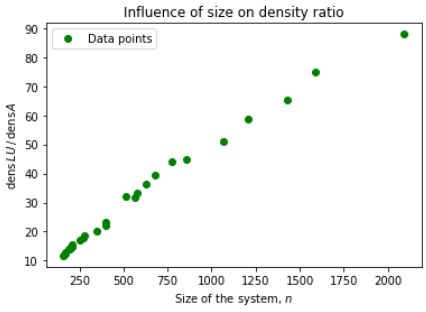
$$\operatorname{dens} A = \frac{\operatorname{nnz}}{n^2}, \quad A \in \mathbb{C}^{n\times n},$$

où nnz est défini comme le nombre d'éléments non nuls de A.

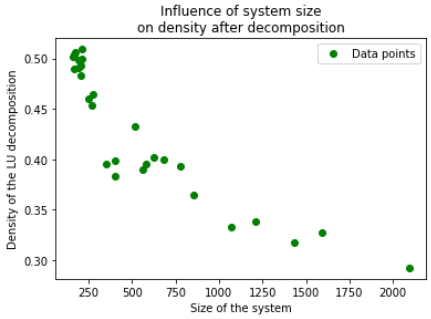
Afin de faire une étude correcte de l'évolution de la densité de la matrice A après sa décomposition LU en place, plusieurs graphes ont été réalisés. Ils sont représentés à la Figure 1.



(a) Valeur de la densité après factorisation en fonction de la densité originelle.



(b) Évolution du rapport entre la densité de la factorisation en place et la densité de la matrice originelle en fonction de la taille du système.



(c) Influence de la taille du système sur la densité de la matrice après factorisation en place.

FIGURE 1 – Différents graphes pertinents pour l'analyse de la section 1.1.

On remarque que plus la matrice devient grande, plus elle devient creuse ; c'est une propriété du modèle d'éléments finis. Sur le graphe de la Figure 1a , on remarque que la densité de la factorisation augmente lorsque la densité de A augmente. Cependant, comme expliqué à la section 2.3, ceci n'est qu'un rapport qualitatif.

En combinant le fait que la densité de A diminue plus le système devient grand, et que la densité de la factorisation diminue lorsque la densité de A devient plus petite, on peut prédire par transitivité que plus le système devient grand, plus la densité de sa factorisation LU va diminuer. C'est ce qu'on observe sur le graphe de la Figure 1c. Finalement, le rapport de la densité de la matrice après factorisation sur sa densité avant factorisation est représenté comme une fonction de la taille du système à la Figure 1b.

1.2 Complexité temporelle de `LUsolve`

Le solveur `LUsolve` fait trois choses séquentiellement, et la complexité totale est donc la somme des trois complexités partielles.

1.2.1 Complexité temporelle de `LUfactorize`

Il s'agit ici de calculer le nombre d'opérations pour une décomposition LU en place. Pour cela, on se réfère au livre de référence p.151. Le nombre d'opérations de l'Algorithme 20.1 est dominé par l'opération vectorielle $u_{j,k:n} = u_{j,k:n} - \ell_{jk} u_{k,k:n}$. On voit que celle-ci effectue une multiplication scalaire-vecteur et une soustraction entre vecteurs. Soit $l = n - k - 1$ la longueur des vecteurs-lignes étant manipulés. Le nombre de *flops* est alors de $2l$. Pour chaque valeur de k, l'indice de la boucle extérieure, la boucle intérieure est répétée pour toutes les lignes de $k+1$ jusque n. La complexité de `LUfactorize`, $d(n)$, est alors

$$d(n) \sim \sum_{k=1}^{n} 2(n-k)(n-k+1) = 2\sum_{j=0}^{n-1} j(j+1) = 2\sum_{j=0}^{n-1} j^2 + 2\sum_{j=0}^{n-1} j = \frac{2n^3}{3} - n^2 + \frac{n}{3} + n^2 - n \sim \frac{2n^3}{3}\,.$$

1.2.2 Complexité temporelle de la substitution arrière

La complexité temporelle de la substitution arrière, $b(n)$, est due à $\frac{n^2-n}{2}$ soustractions, $\frac{n^2-n}{2}$ multiplications et n divisions. Le calcul complet est

$$b(n) \sim \sum_{j=1}^{n} \big(2(n-j)+1\big) \sim 2\sum_{k=0}^{n-1} k + n = n(n-1) + n = n^2\,.$$

1.2.3 Complexité temporelle de la substitution avant

Comme pour la substitution arrière, la complexité temporelle pour la substitution avant, $f(n)$, est due à $\frac{n^2-n}{2}$ soustractions et $\frac{n^2-n}{2}$ multiplications. Le calcul complet est alors

$$f(n) \sim \sum_{j=1}^{n} 2(n-j) = 2\sum_{k=0}^{n-1} k = n(n-1) \sim n^2\,.$$

1.2.4 Complexité totale

Comme dit plus haut, pour trouver $s(n)$, la complexité temporelle totale, il suffit de sommer les 3 complexités (comme les opérations se font en série). On trouve alors

$$s(n) = d(n) + b(n) + f(n) \sim \frac{2n^3}{3} + n^2 + n^2 \sim \frac{2n^3}{3}\,.$$

Comme on peut le voir, l'étape de factorisation est dominante pour la complexité temporelle. On voit sur le graphe de la Figure 2 que cette prédiction théorique est confirmée expérimentalement. Mentionnons finalement que comme les matrices sont issues d'un modèle d'éléments finis, elles sont symétriques (hermitiennes) et définies positives ; il n'est alors pas nécessaire de rajouter une vérification pour éviter les divisions par zéro dans `LUfactorize`. Ceci peut se démontrer par le critère de Sylvester.

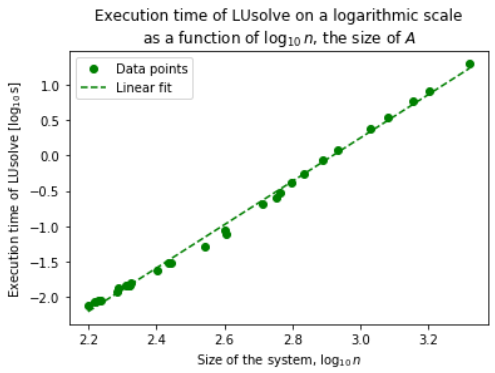


FIGURE 2 – Temps d'exécution de la fonction `LUsolve` en fonction de la taille du système, en échelle logarithmique.

2 Format creux

Comme expliqué dans l'introduction, les matrices issues de `ccore` sont très creuses. Il serait donc intéressant de trouver une manière de résoudre le système sans avoir à faire autant de calculs. Pour cela, il serait utile d'avoir une représentation de la matrice qui n'utilises pas autant de mémoire. Pour faire cela, nous utilisons le format CSR et la fonction `CSRformat`. Une fois cette représentation obtenue, il est possible d'améliorer la complexité de la résolution du système par factorisation LU. Comme expliqué à la section 2.2, cette complexité va dépendre de la largeur de bande de la matrice A. La section 2.3 détaille comment profiter plus de cette nouvelle complexité.

2.1 Performances de `CSRformat`

La fonction `CSRformat` parcourt toute la matrice, ligne par ligne, en copiant toutes les entrées non nulles dans un vecteur `sA`, leurs indices de colonne dans `jA` et l'indice de leur premier élément non nul dans `iA`. On voit donc facilement que cette fonction doit avoir une complexité temporelle de $\Theta(n^2)$. Les opérations les plus coûteuses effectuées sont :

— parcours de la matrice pour trouver la valeur de nnz en $\Theta(n^2)$;
— n recherches d'indices et de valeurs d'éléments non nuls en $\Theta(n^2)$ au total ;

On voit donc bien que la complexité totale est également en $\Theta(n^2)$, ce qui se voit sur le graphe de la Figure 3.

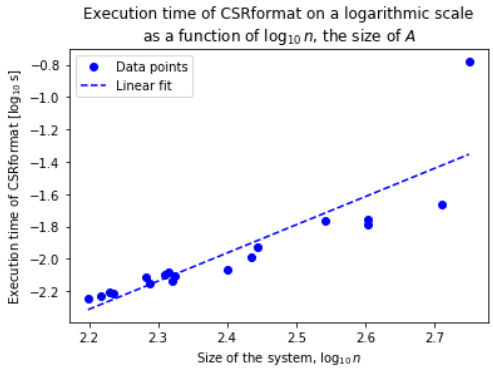


FIGURE 3 – Temps d'exécution de la fonction `CSRformat` en fonction de la taille du système, en échelle logarithmique.

2.2 Complexité temporelle de `LUcsrsolve`

La fonction `LUcsrsolve` est une version optimisée de `LUsolve` spécifiquement pour les matrices creuses. Définissons la largeur de bande k de A comme étant

$$k = \max(k_1, k_2)\,, \quad \text{où} \quad a_{ij} = 0 \quad \text{si} \quad j < i - k_1 \quad \text{ou} \quad j > i + k_2\,; \quad k_1, k_2 \geq 0\,.$$

En effet, une des propriétés de la factorisation LU est que lors de la décomposition, des nouvelles entrées non nulles peuvent apparaître mais uniquement à l'intérieur de la bande de la matrice (phénomène de *fill-in*). On peut donc préallouer facilement cette bande, la remplir au fur et à mesure de la factorisation, et n'avoir qu'à considérer une partie de cette bande (et donc de la matrice A) lors des substitutions.

Pour calculer la complexité de la fonction `LUcsrsolve`, il faut observer qu'elle sera égale à la somme des complexités de `CSRformat` ($\Theta(n^2)$, voir section 2.1), `LUcsr`, et de deux substitutions modifiées.

2.2.1 Complexité temporelle de `LUcsr`

La fonction `LUcsr` a une complexité qui dépend de plusieurs étapes :

— calcul de la largeur de bande en $\Theta(n)$ grâce au format CSR ;
— préallocation de vecteurs en $\mathcal{O}(nk)$;
— décomposition LU creuse en $\mathcal{O}(nk^2)$.

Pour trouver $\lambda(n, k)$, la complexité de la décomposition LU creuse, on observe que la boucle intérieure fait une multiplication et une soustraction, et on calcule (en idéalisant un peu)

$$\lambda(n,k) \in \mathcal{O}\left(\sum_{i=0}^{n-1} \sum_{j=i+1}^{\min(i+k_1+1,n)} \sum_{m=i+1}^{\min(i+k_2+1,n)} 2\right) \in \mathcal{O}\left(2\sum_{i=0}^{n-1} k_1 k_2\right) \in \mathcal{O}(nk^2)\,.$$

Cette complexité-ci domine le temps d'exécution de `LUcsr`.

2.2.2 Complexité des substitutions modifiées

Dans les substitutions modifiées, on peut utiliser le fait que seule la partie non nulle de la matrice nous intéresse. On écrit alors la complexité $\sigma(n,k)$ des deux substitutions en série comme

$$\sigma(n,k) \sim \sum_{i=0}^{n} k_1 + \sum_{i=0}^{n} k_2 \sim 2nk\,.$$

2.2.3 Complexité totale de `LUcsrsolve`

Comme dit plus haut, ces opérations se font toutes en série, et la complexité totale $t(n,k)$ s'écrit alors

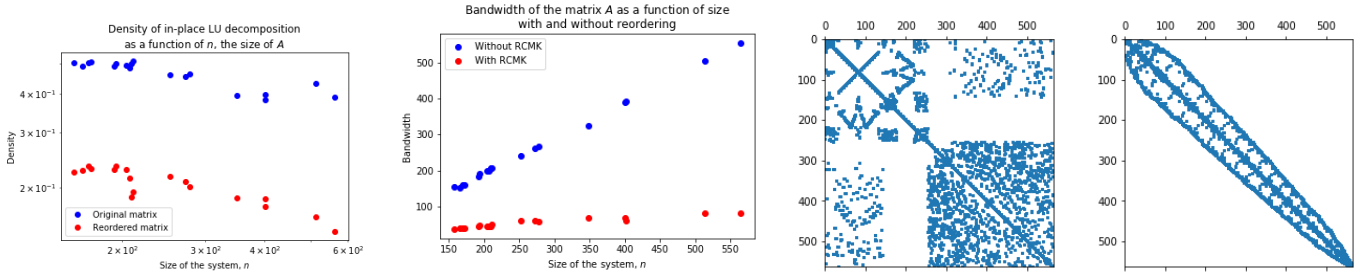
$$t(n,k) \in \mathcal{O}(n^2 + nk^2 + nk) \in \mathcal{O}(nk^2)\,.$$

La factorisation est donc l'étape contribuant le plus à la complexité temporelle.

2.3 Numérotation « reverse Cuthill-McKee »

Comme expliqué à la section 2.2, la complexité de la fonction `LUcsrsolve` dépend du carré de la largeur de bande. Il serait donc intéressant d'utiliser un algorithme de renumérotation des nœuds tel que RCMK pour diminuer cette largeur de bande.

Cet algorithme n'influence pas uniquement la complexité en temps mais aussi en mémoire : lors du *fill-in* de la décomposition LU, plus la bande est petite, plus on peut borner la densité de la matrice résultant de la factorisation en place.



(a) Comparaison entre la densité de la factorisation avec et sans RCMK.

(b) Influence de RCMK sur la largeur de bande de A.

(c) Matrice A sans renumérotation RCMK.

(d) Matrice A avec renumérotation RCMK.

FIGURE 4 – Influence de la renumérotation RCMK sur la densité de A.

Sur la Figure 4, on peut voir que la densité de la factorisation LU diminue lorsque la matrice est permutée (Figure 4a) et que la largeur de bande de la matrice A diminue (et augmente beaucoup moins rapidement) lorsque la matrice est permutée (Figure 4b).

2.4 Complexité de `LUcsrsolve` avec renumérotation RCMK

Afin d'évaluer l'utilité de cette renumérotation, il est utile de regarder l'évolution de la complexité lorsqu'on applique la permutation. Il faut alors rajouter deux étapes à la fonction : le calcul de la permutation optimale (fonction `RCMK`) et son application.

2.4.1 Complexité du calcul de la permutation

L'étape dominante pour la complexité temporelle est le fait de devoir trier à chaque itération la liste des nœds adjacents au nœud venant de sortir de la file. Or, nous pouvons borner la taille de cette liste de nœuds adjacents par le degré maximal dans le graphe, $\Delta(G) \leq k_1 + k_2 \leq 2k$. En assumant qu'un algorithme de tri avec une complexité dans le pire cas inéarithmique, tel que *mergesort*, soit utilisé, on peut alors dire que la fonction `RCMK` appliquée au graphe $G(E,V)$ dont A est la matrice d'adjacence a une complexité $r(G)$en $\mathcal{O}(|V(G)|\Delta(G)\log\Delta(G)$, ou avec les paramètres de notre système n et k,

$$r(n,k) \in \mathcal{O}(n(2k)\log(2k) \in \mathcal{O}(nk\log k)\,.$$

2.4.2 Complexité de l'application de la permutation

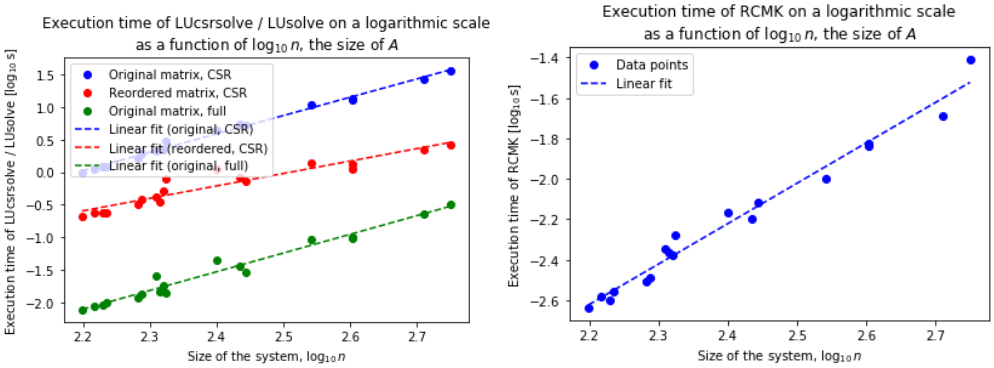
Pour permuter la matrice, on peut imaginer qu'il faut permuter d'abord les lignes puis les colonnes. Pour les lignes, il faut donc déplacer au plus $2nk$ éléments (comme une ligne a au plus de l'ordre de $2k$ entrées), et pour les colonnes la même chose. Il est alors possible d'appliquer la permutation en $\mathcal{O}(nk)$.

2.4.3 Complexité de `LUcsrsolve` avec renumérotation RCMK

La complexité totale $\tau(n,k)$ pour `LUcsrsolve` avec renumération est alors

$$\tau(n,k) \in \mathcal{O}(nk^2 + nk\log k + nk) \in \mathcal{O}(nk^2)\,, \quad \text{car} \quad \lim_{(n,k)\to(+\infty,+\infty)} (nk^2 + nk\log k) = \lim_{(n,k)\to(+\infty,+\infty)} nk^2\,.$$

C'est la même complexité que pour `LUcsrsolve` sans renumérotation, mais on remarque que comme la largeur de bande a fortement diminué, la résolution du système sera beaucoup plus rapide pour les matrices creuses. C'est aussi ce qu'on observe sur la Figure 5a. Le temps d'exécution de la fonction `RCMK` est montré sur la Figure 5b. On observe que le solveur plein est plus rapide pour les matrices de ces tailles (grâce à la vectorisation et aux opérations optimisées de `BLAS`), mais la pente pour le solveur avec RCMK est plus petite, et asymptotiquement, cette méthode sera plus rapide.



(a) Temps d'exécution pour les différents solveurs.

(b) Temps d'exécution pour le calcul de la permutation RCMK.

FIGURE 5 – Influence de la renumérotation RCMK sur le temps d'exécution du solveur.

Un autre point intéressant est que dans le cas de matrices très mal numérotées ($k = n$), l'algorithme creux devient un simple algorithme de factorisation en $\mathcal{O}(n^3)$ et sa pente est égale à la pente pour le solveur plein.