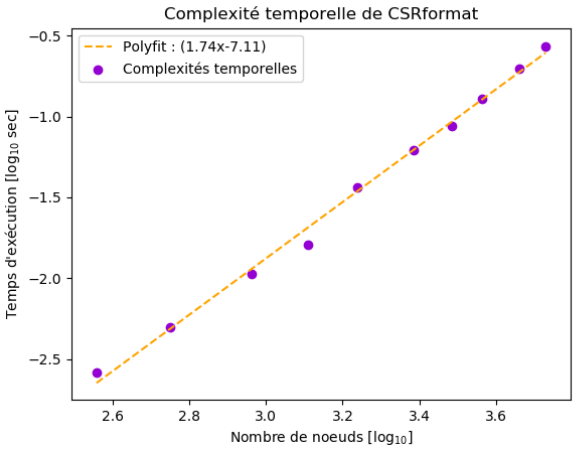
Analyse Numérique : Devoir 3
LU creux

- Abstract -

L'entierté des algorithmes commentés peuvent être retrouvés dans le fichier *LUcsr.py*. Tous les plots peuvent être retrouvés dans le fichier *plot.py*. Pour éviter de recalculer toutes les solutions à chaque fois, toutes ces données ont été sauvegardées, cependant la taille maximale du zip est trop grande pour Moodle, ces données ne sont donc pas présentes dand le zip mais bien sur le GitHub `https://github.com/RomainGrx/LINMA1170-Homeworks`.

1 CSR [Compressed Sparse Row]



Pour réaliser le passage en format creux (CSR) de la matrice, la fonction *CSRformat* doit localiser tous les élements non-nuls de la matrice, pour se faire elle doit parcourir toute la matrice pleine, ce qui nous donne une complexité en $\Theta(n^2)$. Ensuite cette fonction fait la somme d'éléments non-nuls par ligne et fait une somme cumulative par nombre d'éléments non-nuls par ligne pour trouver le vecteur *iA*. Cette complexité est menée par le parcours de la matrice, comme nous pouvons le remarquer ci à gauche, nous obtenons une pente d'environ 2 pour la régression linéaire en logarithme. L'algorithme est décrit ci-dessous.

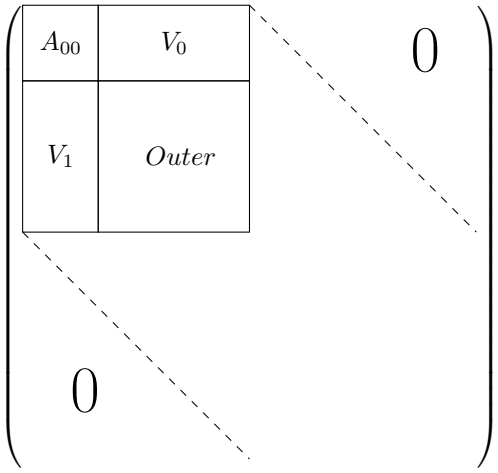
```
1  def CSRformat(A):
2      m, n  = A.shape
3      iA = np.zeros(m+1, dtype=int) # Contiendra l'index de la colonne du premier element
       non-nul d'une ligne
4      (row, col)  = np.nonzero(A) # Liste toutes les valeures non-nulles
5      jA = col # Simplement tous les indexs des colonnes des elements non-nuls
6      ax, count = np.unique(row, return_counts=True) # Somme le nombre d'elements non-nulles
       par ligne
7      iA[ax+1] = count
8      iA = np.cumsum(iA) # Renvoie le vecteur de la somme cumulative par ligne
9      sA = A[row,col] # Renvoie un vecteur avec toutes les valeurs des elements non-nuls
10     return sA, iA, jA
```

2 LU creuse

2.1 Factorisation

2.1.1 Explication

Pour faire cette décomposition, il est tout d'abord nécessaire d'avoir une matrice creuse sous format CSR. Ensuite, il nous faut connaître la largeur de la bande pour connaître la future taille maximale de la décompostion, en effet, lors de la décomposition LU, seul les éléments contenu dans la bande (même nuls) peuvent être modifiés dû au *fill-in* du *outer product*. Une fois cette taille connue, nous pouvons allouer des vecteurs pouvant contenir tous les élements de la bande. Pour connaître l'élément *(i,j)* correspondant au bon élément dans *sLU*, une fonction *get(i,j)* a été créée, elle peut prendre en argument soit deux entiers, soit deux vecteurs d'entiers, ce qui aura pour but de se passer de boucle lors du outer product et ainsi rendre la décomposition plus rapide. Ensuite on peut donc faire un mapping entre les éléments initiaux et les vecteurs représentant la matrice *LU* (copier/coller).



Et enfin, on peut réaliser la décomposition composée d'une itération sur les n éléments diagonaux A_{ii} (pivots) et qui à chaque itération vient diviser la colonne sous le pivot (V_1) par ce même pivot et ensuite vient soustraire toute la matrice carrée contenue à droite et sous l'élément diagonale par le *outer product* de la colonne contenue sous le pivot avec la ligne à droite du pivot (V_0). La force avec les matrices bande creuses est que le *outer product* va se faire seulement jusqu'à l'index maximal entre la borne droite et gauche de la bande et ainsi agir que sur une petite sous-matrice contenue dans les bornes de la bande. L'algorithme de la décomposition se trouvant ci dessous :

```
for k in range(n):
    k_ranger = np.arange(k+1, min(k+left, n)) # Range de la colonne sous le pivot max
    ind = get(k_ranger, [k]*len(k_ranger)) # Les indices dans sLU de cette colonne
    sLU[ind] /= sLU[get(k,k)] # Division par le pivot
    max = min(k+left, k+right, n) # L'indice le plus petit contenant un elem non nul
    outer_ranger = np.arange(k+1,max) # Range du futur outer product
    len_outer = len(outer_ranger)
    outer = np.outer(sLU[get(outer_ranger, [k]*len_outer)], sLU[get([k]*len_outer,
    outer_ranger)]) # Outer product du carre de longueur 'len_outer' centre sur la diagonale
    sLU[get(np.repeat(outer_ranger, len_outer), np.tile(outer_ranger, len_outer))] -=
    outer.flatten() # Reduction du carre d'elements concerne
```

2.1.2 Complexité temporelle

Sous format CSR, la largeur de la bande a de l'importance pour la complexité, elle dépend de la borne à gauche (notée g) par rapport à la diagonale et de la borne à droite (notée d) toujours par rapport a cette même diagonale avec $l = max(g, d) + 1$. Cette complexité temporelle dépend de différentes étapes: tout d'abord le calcul de la largeur de la bande qui est simplement le parcours par ligne : $\Theta(n)$, ensuite l'allocation des vecteurs sLU, iLU et jLU de complexité maximale $O(n(2l+1)+1) \subseteq O(nl)$ et enfin la décomposition de complexité maximale (décrite au point 2.1.1)

$$O\left(\sum_{i=1}^{n}\left(\sum_{j=i+1}^{min(i+g,n)} + \left(\sum_{k=i+1}^{min(i+g,n)}\right)\left(\sum_{m=i+1}^{min(i+d,n)}\right)\right)\right) \subseteq O(n(g+gd)) \subseteq O(nl^2)$$

2.2 Résolution

2.2.1 Explication

Pour cette résolution, nous devons d'abord factoriser en LU comme décrit ci-dessus et ensuite faire une substitution avant pour $Lu = b$ et enfin une substitution arrière pour $Ux = y$.

2.2.2 Complexité temporelle

Comme sus-mentionné, nous avons deux substitutions, ainsi la complexité de ces deux substitutions à la suite donne:

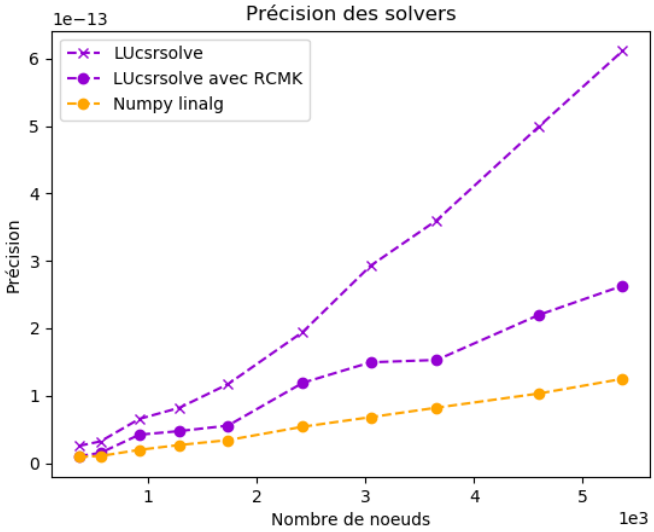
$$O(\text{Sub. avant} + \text{Sub. arrière}) = O\left(\sum_{i=1}^{n}\sum_{j=max(0,i-g)}^{i-1} + \sum_{i=1}^{n}\sum_{j=i+1}^{min(i+d,n)}\right) \subseteq O(ng+nd) \subseteq O(nl)$$

Étant donné qu'il est nécessaire de d'abord factoriser, la complexité totale de la résolution nous donne:

$$O(nl^2 + nl) \subseteq O(nl^2)$$

2.2.3 Précision

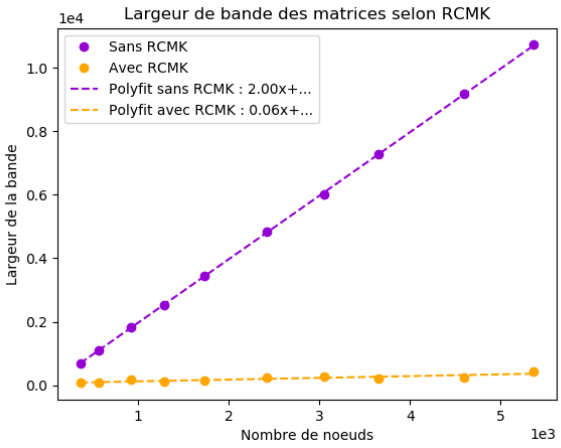
La fonction sous format creux et avec les noeuds renumérotés a une précision proche de celle donnée par *Numpy linalg solve*, cependant lorsqu'il n'y a pas de renumérotation, plus de calculs sont faits et les erreurs d'arrondis s'accumulent, amenant à une précision moindre.



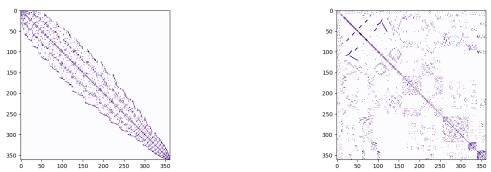
3 RCMK [Reverse Cuthill-McKee]

Étant donné que la complexité temporelle de *LUcsrsolve* dépend de la largeur de la bande, il serait intéressant de réduire la largeur de la bande. L'algorithme *RCMK* vient réaranger les noeuds du problème pour densifier la bande la matrice et ainsi avoir une largeur de bande moindre.

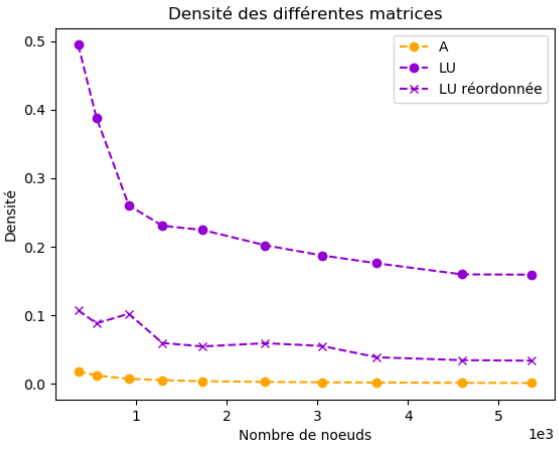
3.0.1 Influence sur la densité de la factorisation



Comme nous pouvons le voir ci à gauche, sans renumérotation, la largeur de la bande augmente linéairement avec le nombre de noeuds du modèle et avec une pente de 2, ce qui veut dire que la borne à gauche (resp. à droite) sus-mentionnée de la bande augmente au même taux que le nombre de noeuds augmente. Contrairement à la largeur de la bande lorsqu'il y a une renumérotation, la bande augmente avec un taux d'environ 6% par rapport au nombre de noeuds.

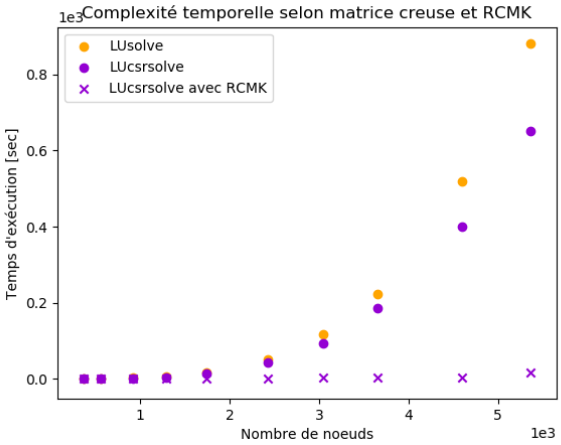


(a) Avec renumérotation (b) Sans renumérotation



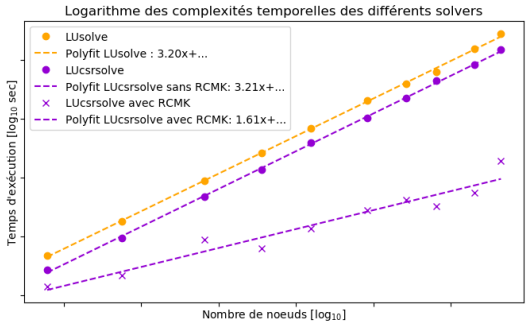
La densité de la matrice A est moindre que les matrices LU avec ou sans RCMK, ce qui est normal dû au *fill-in*. Ensuite, on peut remarquer que la décomposition LU sans RCMK est plus dense, dû au phénomène de *fill-in* qui est plus présent dû à la largeur de bande plus grande. Enfin, on voit que cette densité diminue avec la taille du système ce qui est une priorité du modèle d'éléments finis.

3.0.2 Influence sur la complexité temporelle de la résolution

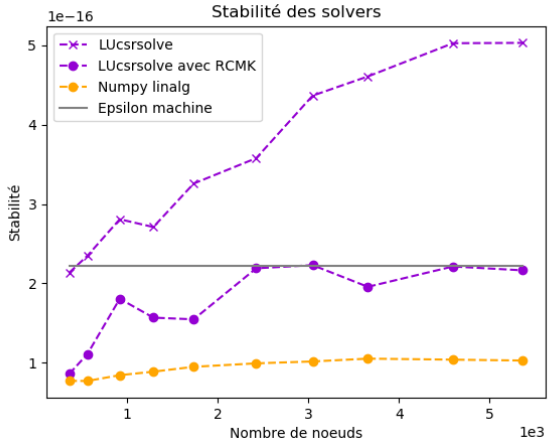


On peut donc bien remarquer que dans le cas de notre modèle d'élements finis qui est creux, la fonction de résolution par *factorisation LU* sous format creux et plus performant en terme de temps de calculs. Étant donné que la bande couvre quasiment toute la largeur de la matrice lorsqu'il n'y a pas de renumérotation, elle a donc une complexité temporelle proche de la factorisation avec matrice pleine car les calculs sont quasiement identiques. Cependant lors de l'utilisation de la renumérotation, la complexité est très faible car comme indiqué au point 3.0.1 et 2.2.2, la largeur de la bande n'augmente que de 6% par rapport aux noeuds et la complexité dépend de la largeur de bande.

De manière qualitative, on peut voir que la complexité du solver LU en format creux est d'environ $O(n^{1.6})$. Cette complexité n'est pas objective dû à toutes les fonctions numpy vectorisées.



4 Nécessité du pivotage



Lors de l'algorithme sans renumérotation RCMK, la stabilité est semblable à l'algorithme LU dense sans pivotage, ainsi, la stabilité n'est pas optimale dû aux possibles divisions par 0 de l'élimination de Gauss. Cependant, lors de l'utilisation de la renumérotation RCMK, la bande de la matrice est bien plus dense, les éléments sont réarangés autour de la diagonale, ainsi, les divisions par 0 ne sont pas possibles et les phénomènes de *fill-in* se font moins que dans la version avec les éléments disparates. La *backward stability* est donc bien meilleure comme peut le montrer le graphe ci à gauche, elle est de l'ordre de $O(\epsilon_{\text{machine}})$.