Analyse Numérique : Devoir 2
Factorisations matricielles

1 LU

1.1 Factorisation

Cette décomposition a pour but de factoriser la matrice initiale A en une matrice *triangulaire inférieure* L à diagonale unité et une matrice *triangulaire supérieure* U ainsi, A = LU. Cette factorisation cherche d'abord a faire un *pivotage partiel*, ainsi les lignes sont échangées pour mettre les éléments maximums contenus sous la diagonale sur l'élément de la diagonale, de cette manière on évitera par la suite de diviser par zéro lorsque l'on divisera par le pivot. Ensuite, on applique les *éliminiations de Gauss* de manière itérative sur chaque éléments de L et de U. Comme nous sommes en Python, j'ai cherché à vectoriser un maximum de boucles, ce qui nous donne l'algorithme de 3 lignes suivant une fois la matrice pivotée:

```python
def LU(A):
    LU, P = pivot(A) # Retourne la matrice PA pivotee ainsi que le vecteur de pivotage
    m,n = LU.shape
    for k in range(m):
        LU[k+1:,k]     /= LU[k,k] # Division par le pivot
        LU[k+1:,k+1:] -= np.einsum('i,j->ij',LU[k+1:,k], LU[k,k+1:]) # Outer product
    return LU, P
```

1.2 Résolution

Nous appliquons d'abord la *matrice de pivotage* (P) de chaque côté de l'équation $Ax = b$ ainsi, $PAx = Pb$. Ensuite, ayant LU comme étant deux *matrices triangulaires*, nous pouvons résoudre de la manière suivante en appliquant pour la résolution avec L, une substitution avant et avec U, une substitiution arrière, de la manière suivante:

$$\begin{aligned}
Ax &= b & & \\
PAx &= Pb & & \text{Multiplication par la matrice de permutation } P \\
LUx &= Pb & & \text{Factorisation } LU \text{ de la matrice } PA \\
Ly &= Pb & & \text{Résolution de } y \text{ par substitution avant} \\
Ux &= y & & \text{Résolution de } x \text{ par substitution arrière}
\end{aligned}$$

1.3 Propriétés

Déterminant: Le déterminant d'un produit est le produit des déterminants, ainsi : $det(A) = det(PLU) = det(P)det(L)det(U)$ hors :

- le déterminant d'une *matrice de permutation* vaut -1^p avec p le nombre de permutations.
- le déterminant d'une *matrice triangulaire* est la multiplication des éléments de sa diagonale, de plus, L a une diagonale unitaire, son déterminant vaut donc toujours 1.

→ Le déterminant de A vaut donc simplement le produit des éléments diagonaux de U multiplié par -1 si il y a un nombre impair de permuations ainsi : $det(A) = (-1)^p \prod_{i=1}^{m} U_{ii}$

Inverse: Il y a deux techniques pour calculer l'inverse de A:

1. L'inverse d'un produit est le produit des inverses en permutant le sens du produit, ainsi : $A^{-1} = (PLU)^{-1} = U^{-1}L^{-1}P^{-1}$ hors :
   - l'inverse d'une *matrice de permutation* est sa *transposée*.
   - l'inverse d'une *matrice triangulaire supérieure* (resp. *inférieure*) renvoie une *matrice triangulaire supérieure* (resp. *inférieure*), ainsi on peut se concentrer que sur une moitié de la matrice.

2. L'inverse de A peut être obtenue en resolvant le système $PLUA^{-1} = I_m$ hors :

$$\begin{aligned}
PLUA^{-1} &= I_m \\
LUA^{-1} &= P^T && \text{Comme sus-mentionné : } P^T = P^{-1} \\
Ly &= P^T && \text{Résolution par substitution avant} \\
UA^{-1} &= y && \text{Résolution par substitution arrière}
\end{aligned}$$

2 QR

2.1 Factorisation

Cette factorisation a été réalisée autour d'une décomposition *Gram–Schmidt modifié* car elle est plus stable que la classique dû aux erreurs d'arrondis. Cette décomposition a pour but de factoriser la matrice initiale A en une matrice Q *unitaire* et en une matrice R *triangulaire supérieure* ainsi, A = QR.

```
def QR(A):
    R = np.zeros_like(A, shape=(n,n))
    V = np.array(A)
    m,n = V.shape
    for i in range(n):
        R[i,i]       = np.linalg.norm(V[:,i], axis=0)
        Qi           = np.divide(V[:,i], R[i,i])
        R[i,i+1:n]   = Qi @ V[:,i+1:n]
        V[:,i+1:n] -= np.einsum('i,j->ij', Qi, R[i,i+1:n])
    return V, R
```

2.2 Résolution

Ayant Q comme étant une matrice unitaire ($Q^TQ = I$), nous pouvons simplement obtenir la sous solution en multipliant par sa transposée. Ensuite R étant une matrice *triangulaire supérieure*, nous pouvons obtenir la solution par *substitution arrière*, de la manière suivante:

$$\begin{aligned}
Ax &= b \\
QRx &= b && \text{Factorisation } QR \text{ de la matrice } A \\
y &= Q^T b && \text{Résolution de } y \text{ par multiplication de } Q^T \\
Rx &= y && \text{Résolution de } x \text{ par substitution arrière}
\end{aligned}$$

3 Analyse

3.1 Complexité temporelle sur maillages différents

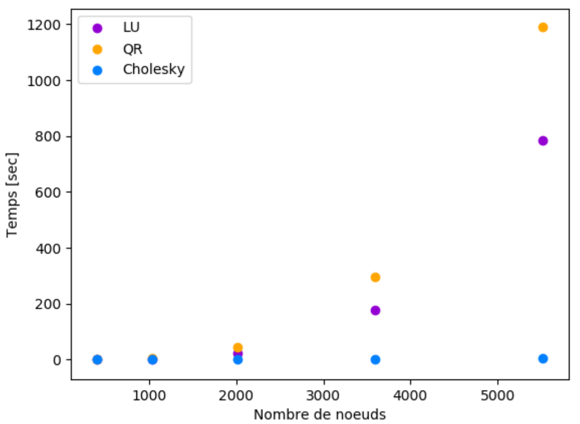
LU: La complexité de *LUsolve* peut-être décomposée en deux parties:

- *LU*: Cette décomposition LU a une complexité temporelle de $\frac{2m^3}{3}$
- *forward* & *backward*: Les deux ont le même nombre d'opérations, prenons donc *forward*, elle va exécuter : $\sum_{i=1}^{m} \frac{b_i \sum_{j=1}^{i-1} A_{ij}x_i}{A_{ii}}$ ce qui est équivalent à une complexité pour les deux à la suite d'environ $2m^2$

  $\rightarrow$ Complexité totale $\sim \frac{2m^3}{3} + 2m^2 \sim \frac{2m^3}{3}$

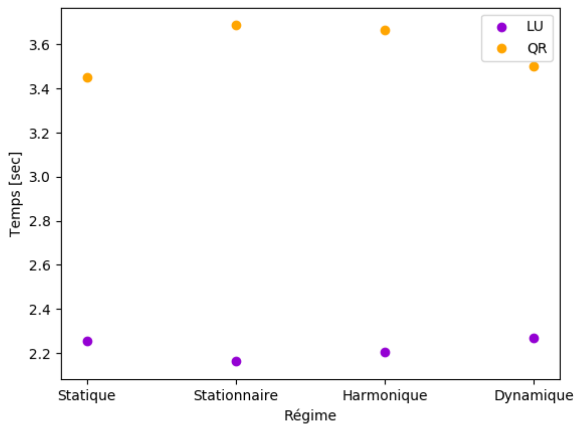
QR: La complexité de *QRsolve* peut également être décomposée en deux parties, la première est liée à la décomposition et vaut environ $\frac{4m^3}{3}$ et l'autre est liée à la résolution où la complexité est dominée par la substituion arrière elle même dominée par la décomposotion.

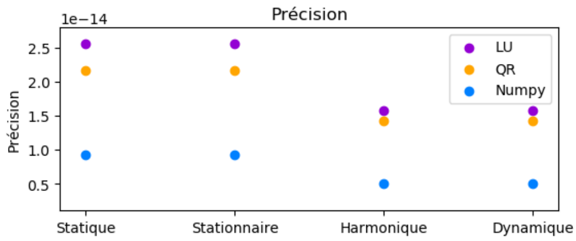
$\rightarrow$ Complexité totale $\sim \frac{4m^3}{3}$



Étant donné que la complexité de la *factorisation LU* est de $\frac{2m^3}{3}$ et la complexité de la *factorisation QR* est de $\frac{4m^3}{3}$ (m étant la taille de la matrice A), il est normal d'observé que la résolution avec la *factorisation LU* est plus rapide et les courbes suivent bien les complexités attendues. Piste d'amélioration: Étant donné que nous sommes en régime statique, la matrice A est symétrique et définie positive, il est possible d'utiliser une décomposition de *Cholesky* qui a une complexitée deux fois moindre qu'une simple *factorisation LU* puisqu'elle agit sur un triangle de la matrice symétrique. (Fonction *Cholesky* de *numpy*)

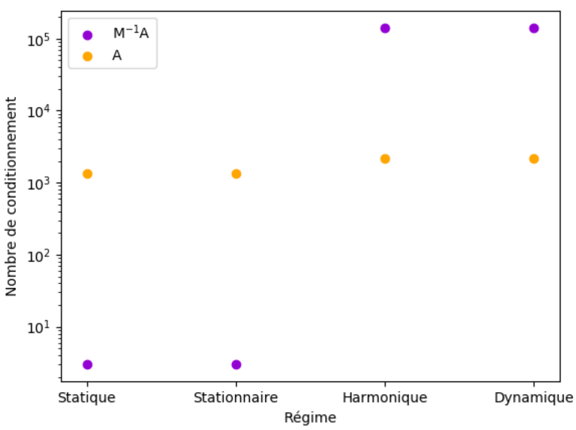
3.2 Complexité temporelle sur régimes différents





Comme expliqué au point [3.1], on peut observer que la complexité temporelle de QR est environ deux fois supérieure à celle de LU. Ensuite, ci-dessus nous pouvons observer que les deux décompositions ont une précision ($\frac{\|Ax-b\|}{\|b\|}$) proche de 10^{-14}.

3.3 Conditionnement ILU



Comme nous pouvons l'observer, lorsque la *vélocité* est égale de 0, la matrice $M^{-1}A$ est bien préconditionnée par rapport à la matrice initiale, cependant lorsque la *vélocité* rentre en jeu, le conditionnement devient extrêmement plus grand dû aux termes dans A se rapprochant de 0 qui vont être divisé dans ILU.