1 Algorithme du *solver*

Le but de ce devoir était d'implémenter une factorisation QR, et de résoudre un système linéaire donné sous forme matricielle utilisant celle-ci. L'implémentation est divisée en deux parties :

- La fonction `QR` qui calcule, pour une matrice $A \in \mathbb{R}^{m\times n}$, la matrice triangulaire supérieure $R \in \mathbb{R}^{n\times n}$, ainsi qu'une matrice V reprenant les différents réflecteurs de Householder.
- La fonction `QRsolve`, qui résout le système $Ax = b$ en utilisant la fonction `QR` pour factoriser A, et qui utilise ensuite une substitution arrière pour trouver x, sans devoir calculer explicitement Q.

2 Complexité

`QR` La complexité $f(n)$ de la fonction `QR` est majoritairement due à la boucle la plus imbriquée. Cette boucle prend 4 flops pour chaque entrée. À chaque itération, la boucle extérieure s'effectue sur une ligne de moins. En plus, le nombre de colonnes diminue également de un à chaque itération. On trouve donc par un raisonnement géométrique que

$$f(n) \sim 4\left(\frac{1}{2}mn^2 - \frac{1}{6}n^3\right) \sim 2mn^2 - \frac{2}{3}n^3\,.$$

`QRsolve` Dans la fonction `QRsolve`, on fait appel à `QR`, cependant dans cette section on détaille uniquement le coût algorithmique propre à la fonction. Celui-ci est dominé par la substitution arrière, dont la complexité est

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

Totale Comme la complexité $t(m,n)$ qui nous intéresse ici est asymptotique, et que `QRsolve` et `QR` se font séquentiellement, on peut laisser tomber les termes d'ordre inférieur dus à la substitution arrière. On obtient donc une complexité totale

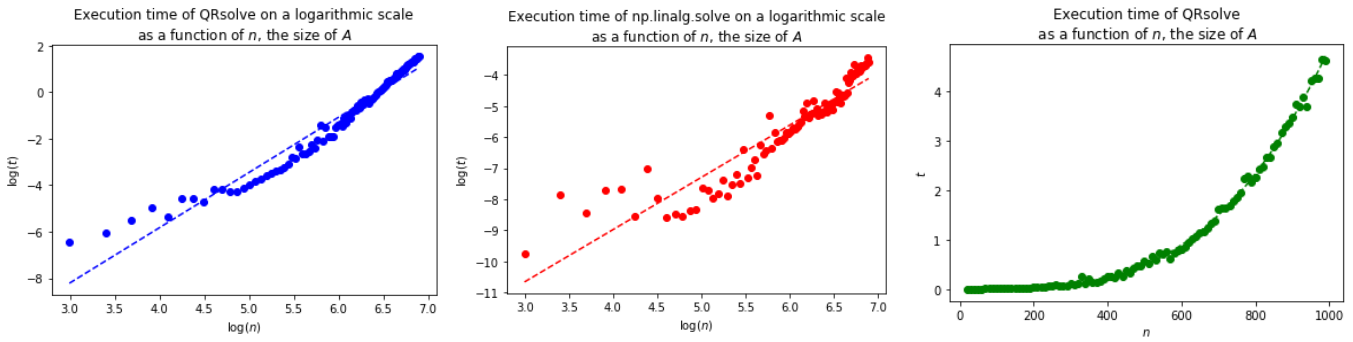
$$t(m,n) \sim 2mn^2 - \frac{2}{3}n^3 + n^2 \sim 2mn^2 - \frac{2}{3}n^3 \overset{m=n}{\Longrightarrow} t(n) \sim \frac{4}{3}n^3\,.$$

Si on construit le graphe donnant le temps d'exécution en fonction de la taille de l'entrée en échelle logarithmique, on s'attend donc à avoir une pente de 3, car $\log n^d = d\log n$.

3 Résultats

Afin de vérifier la bonne implémentation de l'algorithme, celui-ci a été testé en comparant les résultats avec ceux de `np.linalg.solve`. On remarque que sur les graphes log-log (figures 1a et 1b), la pente pour les deux solvers est très proche de 3, ce qui correspond à la prédiction faite à la section 2. Notons cependant que `np.linalg.solve` est plus rapide d'un facteur $\approx 10\,000$, car il s'agit d'une implémentation basée sur l'algorithme de décomposition LU plus efficace (d'un facteur 2, environ) et car cet algorithme est implémenté en Fortran (LAPACK), un langage beaucoup plus bas niveau que Python (car Python est un langage interpreté alors que Fortran est compilé).

À cause de cette différence d'ordre de grandeur, le solver de `NumPy` est plus sensible aux autres tâches effectuées par le processeur. On remarque en effet que les points sont moins regroupés (surtout pour les petites valeurs) que pour le graphe de `QRsolve`.



(a) `QRsolve`, échelle logarithmique. (b) `NumPy`, échelle logarithmique. (c) `QRsolve`, échelle linéaire.

FIGURE 1 – Graphes du temps d'exécution en fonction de la taille d'entrée sur différentes échelles.