Scripts de test

Un premier script de test vérifie le bon fonctionnement des fonctions de base et auxiliaires définies dans `matrix.h`. Pour le lancer, il suffit de taper la commande `make test_mat` dans le terminal (MakeFile a été joint aux fichiers afin d'automatiser et de simplifier la compilation).
L'ensemble des scripts de test repose sur l'exploitation de la librairie `assert.h` qui fournit l'appel système `void assert(scalar expression)`. Ce dernier entraîne la terminaison du programme en cas d'échec d'un test, i.e. si l'expression fournie en argument n'est pas vérifiée. Ainsi, la réussite des tests proposés est garantie si les scripts de test se terminent sans interruption externe.
Le second script de test est plus intéressant. Il est responsable de contrôler la correction des fonctions `lu`, `solve` et `cholesky` définies dans `lu.h`. Pour l'exécuter, il faut à nouveau entrer la commande `make test_lu` dans le terminal. Il teste d'abord la fonction `lu` sur des matrices de petites dimension ($\mathbb{R}^{2\times2}$ et $\mathbb{R}^{3\times3}$) dont la décomposition $L * U$ est connue et vérifie la correction de l'algorithme par comparaison directe. Le test s'assure aussi assuré que la multiplication des matrices L et U obtenues ont pour résultat la matrice de départ. Ensuite, le script vérifie que la fonction détecte les pivots nuls en lui fournissant des matrices de rang non plein ou qui contiennent un pivot nul se trouvait en dernière position de la diagonale. Enfin, il génère des matrices pseudo-aléatoires de tailles croissantes afin de les décomposer et de vérifier que la multiplication des composantes produit bien la matrice de départ.
De manière analogue, le script vérifie l'exactitude de la fonction `solve` par comparaison directe des solutions obtenues pour des systèmes linéaires de 2 à 3 équations/inconnues. De nouveau, un test produit des matrices A et des vecteurs b de taille croissante pseudo-aléatoirement afin de s'assurer que la multiplication matricielle de A et du vecteur solution calculé par la fonction `solve` fournit comme résultat le vecteur b initial.
A titre comparatif, la fonction `cholesky` calculant la factorisation de Cholesky d'une matrice symétrique et définie positive a aussi été implémentée dans `lu.c`. Cette dernière a été testée de manière similaire à la fonction `lu`.

Complexités

Nous avons vu au cours que la complexité théorique de la fonction `lu` peut être trouvée assez facilement. La k-ième itération lors de la décomposition d'une matrice $A^{n\times n}$ requiert $2 \times (n-k-1)^2$ opérations en virgule flottante. Dès lors,

$$2\sum_{k=0}^{n-1}(n-k-1)^2 = 2\left(\frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}\right) \sim \frac{2}{3}n^3 \tag{1}$$

En divisant ce résultat par le nombre de FLOPS calculés selon les spécifications du processeur exécutant, on obtient ainsi le temps d'exécution théorique de la fonction `lu` (figure 1) pour des matrices inversibles de taille croissante [1]. En pratique, les temps d'exécution mesurés sont supérieurs de plusieurs ordres de grandeur à cette estimation théorique. Ce n'est guère surprenant puisque plusieurs hypothèses importantes n'ont pas été prises en compte :

1. L'ensemble des coeurs (et des threads) du CPU ne sont pas exclusivement alloués à l'exécution du programme ;
2. Le fil d'instructions n'est pas parfaitement parallélisable et n'est pas réparti équitablement entre les coeurs ;
3. La compilation et l'exécution du code par le système d'exploitation induit une borne inférieure physique sur le temps d'exécution ;
4. Les accès mémoire prolongent aussi le temps d'exécution.

1. Ces matrices ont aussi été générées pseudo-aléatoirement. Afin de garantir leur non-singularité qui interromprait prématurément l'exécution de la fonction `lu`, des matrices à diagonale strictement dominante sont privilégiées.

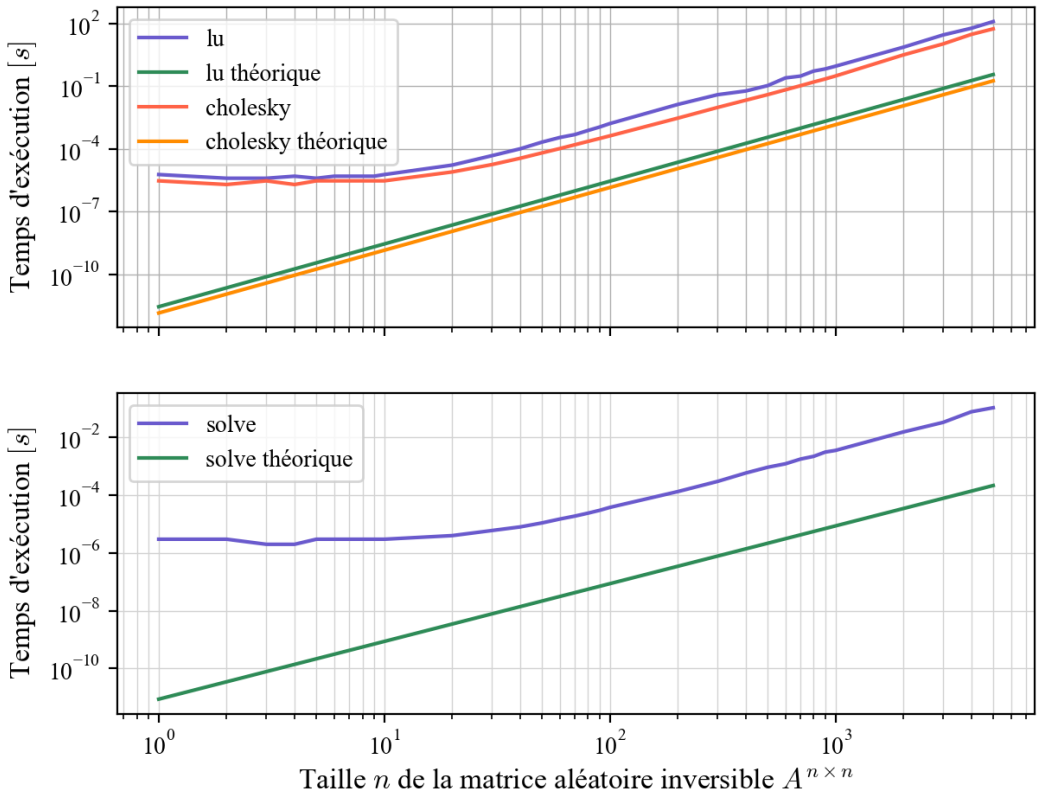

FIGURE 1

Toutefois, le temps d'exécution asymptotique possède le comportement prédit (une droite de pente 3).
En comparaison, la factorisation de Cholesky nécessite seulement 1 division, $n-k+1$ multiplications et $n-k+1$ divisions lors de la k-ième itération de la boucle principale. Le nombre d'opérations est alors $\sim \frac{1}{3}n^3$. La fonction `cholesky` est donc théoriquement deux fois plus rapide que la fonction `lu`. En pratique, ceci a bien été observé comme le montre la figure [2]. On observe aussi un gain de performance au niveau de la complexité spatiale. En effet, l'algorithme laisse la partie triangulaire inférieure sous la diagonale de la matrice à décomposer inchangée. On pourrait donc allouer une structures de données plus compact ne stockant en mémoire que les éléments diagonaux et supérieurs. Cependant, ce gain n'a pas été exploité en pratique puisque les tests réutilisent la structure `Matrix` définie dans `matrix.h`.
Pareillement, on peut estimer le temps d'exécution théorique de la fonction `solve`. Pour un système linaire inférieurement [3] triangulaire, la résolution de la k-ième équation requiert 1 division, k soustraction et k multiplications. Dès lors,

$$2\sum_{k=0}^{n-1} 2k+1 = 2n^2 \sim 2n^2 \tag{2}$$

Les remarques précédents restent valides et à nouveau le comportement asymptotique est conforme à nos espérances (une droite de pente 2).

2. Le processus de génération de matrices inversibles est identique à celui utilisé pour la fonction `lu` auquel on ajoute la contrainte que les coefficients diagonaux soient strictement positifs (et que la matrice soit symétrique évidemment). Ainsi, les matrices obtenues sont définies positives.
3. Pour un système supérieurement triangulaire, on obtient le même résultat en commençant par la dernière équation.