Décomposition PLU et résolution de systèmes d'équations linéaires

Lucas Roig

Novembre 2015

1 Introduction

Ce projet est réalisé dans le cadre du cours " Méthodes Matricielle ", il consiste à implémenter en langage Python l'algorithme de décomposition PLU, ainsi qu'une fonction permettant de résoudre des systèmes d'équations linéaires de Cramer à partir de cette décomposition.

Le projet se divise en quatre parties : Permutations, Transvection, Décomposition, Applications. Les deux premières parties implémentent des fonctions permettant d'effectuer des opérations sur les matrices. Ces fonction sont utilisées dans la troisième partie afin d'implémenter l'algorithme de décomposition PLU. Enfin la dernière partie utilise cette décomposition afin de résoudre des systèmes de Cramer ou bien de calculer le determinant d'une matrice. Pour chacune de ces parties, chaque fonction demandée a été implémentéé et des tests unitaires ont été écrits.

Au cours de ce travail, la principale préocupation a été de réduire le temps d'éxécution de l'algorithme de décomposition. Ainsi certaines fonctions ont été rajoutées au projet, elles sont toutes des variantes des fonctions demandées conçues spécifiquement pour l'algorithme de décomposition, les utiliser dans un autre contexte poserait des problèmes, comme par exemple la modification en place des matrices données en entrée.

L'analyse des performances des fonction a été réalisée grâce à l'outil lineprofiler développé par Robert Kern.

La méthodologie de travail adoptée pour ce projet est la suivante :

- Ecriture des tests unitaires de la fonction
- Ecriture de la fonction
- Optimisation de la fonction d'un point de vue algorithmique et optimisation du code python. Ces deux étapes sont réalisées simultanément car parfois certains algorithmes permettent des optimisations rendant leur éxécution significativement plus rapide que celle d'autres algorithme étant pourtant légèrement plus moins complexes.

2 Décomposition PLU

Le coeur de ce projet est la fonction decomposition_plu. C'est l'algorithme le moins facile à comprendre du projet et aussi le plus complexe. C'est donc cette fonction qui a demandé le plus de temps non seulement à implémenter mais aussi à tester et à optimiser. La difficulté lors de l'écriture des tests est de trouver des matrices qui permettent de tester le comportement de la fonction lorsqu'elle doit effectuer de nombreuses permutations ou bien lorsqu'elle rencontre des colonnes sans pivot.

2.1 Optimisation

Au fur et à mesure de l'étape d'optimisation, l'algorithme a changé d'apparence, il me paraît donc nécessaire d'expliciter ces modifications afin de montrer que l'algorithme réalise toujours les mêmes opérations.

2.1.1 Instructions conditionnelles

Des tests sur des matrices générées aléatoirement ont montrés qu'il était rare de ne pas trouver de pivot dans une colonne ou même d'avoir besoin d'effectuer une permutation pour trouver le pivot. Dans l'algorithme de decomposition PLU, à chaque itération, on effectue une permutation sur les matrices P, L et U. Lorsqu'aucune permutation n'est nécessaire on multiplie donc ces matrices par la matrice Identité afin de ne pas les modifier. Puisqu'on ne les modifie pas on pourrait sauter cette étape. La fonction decomposition_plu teste donc si cette permutation est necessaire (en vérifiant la position du pivot) et ne l'effectue que si elle l'est. Certes, dans le pire des cas (si une permutation est necessaire à chaque itération) on réalise n tests supplémentaires (pour une matrice de taille n).Mais ce cas est très rare et le temps gagné en moyenne n'est pas négligeable.

2.1.2 Les fonctions permutation_ligne et permutation_colonne

Lorsque l'on effectue une permutation, l'algorithme de décomposition PLU calcule une matrice de permutation P_{aux} et utilise les fonctions produit_permut afin de modifier les matrices P, L et U. En réalité cette permutation ne fait qu'échanger deux lignes (lp et lig) ou deux colonnes des matrices. La compléxité des fonctions produit_permutation est due au fait qu'elles sont capables de permuter un nombre quelconque de lignes ou de colonnes dans la matrice donnée en entrée. Or nous avons simplement besoin de permuter deux lignes ou deux colonnes, on a donc écrit deux fonctions permutation_ligne et permutation_colonne prenant en paramètre l'indice des lignes et des colonnes à permutter et réalisant cette permutation. Ceci nous permet également d'éviter de générer la matrice P_{aux}. Grâce à cela, on économise dans le pire des cas environ 10 % du temps d'éxécution.

2.1.3 La fonction produit_transvection_gauche_iteratif

Cette idée est due au constat suivant : en python l'appel de fonction est lent, il faut donc en appeler le moins possible. Dans la boucle la plus interne de l'algorithme de décomposition PLU, on modifie la matrice A grace à la fonction produit_transvection_gauche. Cette fonction est appelée environ n^2 fois. L'idée est simplement de réaliser l'itération dans la fonction. Ainsi on appelle cette fonction seulement n fois. Pour cela, la fonction prend en paramètre la ligne à partir de laquelle elle doit effectuer les transvection, la colonne en court de traitement et le pivot. Ensuite à chaque itération elle calcule le coefficient de la transvection.

On remarque aussi que cette fonction n'utilise pas les opérateurs d'addition et de multiplication, mais les fonctions correspondantes. Ceci nous permet d'utiliser le built-in map et d'adopter une solution fonctionnelle au problème de la transvection. Cette approche est bien plus performante que l'approche itérative consistant à effectuer n affection à chaque ligne.

Une autre astuce prend en considération le fait que dans le cadre de notre décomposition PLU, pour chaque ligne, jusqu'à un certain indice tous les éléments sont égaux à 0 et le resteront après la transvection. On peut donc se permettre de calculer seulement les éléments à droite de cet indice.

Cette fonction est sans aucun doute celle qui permet de gagner le plus de temps à l'exécution.

2.1.4 La matrice L

En analysant le comportement de l'algorithme de décomposition PLU lors du traitement d'une matrice ne necessitant aucune permutation, on s'aperçoit que la ligne modifiant L grâce à la fonction produit_transvection_d pourrait être remplacée par :

$$L[i][col] = \frac{A[i][col]}{pivot}$$

Ce qui s'exécute en temps constant.

Cependant dès qu'une matrice nécessite une permutation, cela n'est plus aussi simple, il existe bien une cellule de L qui prend la valeur $\frac{A[i][col]}{pivot}$, mais on ne sait pas la trouver en temps constant. Il était donc important de trouver une solution pour utiliser l'affectation simple dans tous les cas.

En comparant la matrice obtenue en effectuant simplement l'affection (on l'appellera L') à celle obtenue avec l'algorithme décomposition PLU on se rend compte que L' contient les mêmes nombres que L mais ils ne sont pas disposés de la bonne manière (mis à part les un sur la diagonale). Pour les remettre dans le bon ordre, il faut permutter les nombres à gauche de la diagonale ligne à ligne et dans l'ordre dans lequel on aurait effectué ces permutations dans l'algorithme de Decomposition PLU. Par exemple :

Soit $A = \begin{pmatrix} 1 & 2 & 1 & 1 \\ 3 & 6 & 3 & 4 \\ 4 & 2 & 4 & 1 \\ 2 & 3 & 4 & 1 \end{pmatrix}$

On trouve $L' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 2 & \frac{1}{6} & 0 & 1 \end{pmatrix}$

Au cours de l'algorithme on a fait permuter les lignes 2 et 3 puis les lignes 3 et 4. Dans notre matrice L' on fait donc permuter les nombres des lignes 2 et 3 étant à gauche de la diagonale.

On obtient $L' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 2 & \frac{1}{6} & 0 & 1 \end{pmatrix}$

On répète l'opération avec les termes des lignes 3 et 4:

$L' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 \\ 2 & \frac{1}{6} & 1 & 0 \\ 3 & 0 & 0 & 1 \end{pmatrix}$

Ce qui est bien égal à L. Je n'ai pas trouvé de démonstration mathématique de cette méthode mais les tests unitaires ont montrés qu'elle fonctionnait.

Ici aussi l'économie de temps est significative, au lieu d'appeler n^2 fois une fonction effectuant n opérations, on effectue n^2 affections.

2.2 Performances

En résumé : on peut montrer l'évolution de la taille des matrices pouvant être traitées en une seconde sur notre machine. Avant toute les modifications apportées, la fonction decomposition_PLU s'exécutait en un peu plus d'une seconde pour une matrice de taille 50, dans le même temps, on peut maintenant traiter une matrice de taille 275, et ce peu importe le nombre de permutations nécessaires. En effet le traitement du pire des cas demande environ 5 % de temps en plus que le traitement du cas moyen.

2.3 Le problème des arrondis

Un problème récurrent relatif à la représentation des nombre en informatique est celui des arrondis. Par exemple pour python : $99 + (\frac{-91*99}{91}) = 1,4 \times 10^{-14}$

Cette opération est semblable à celle que réalise la fonction produit transvection gauche iteratif. Ceci pose de nombreux problèmes lors des tests. Par exemple : des matrices sensées être triangulaires ne le sont pas à cause de ces arrondis. La solution mise en place est simple : lors d'une transvection dans l'algorithme de décomposition PLU, on sait que le coefficient de la colonne en dessous du pivot

sera nul (puisqu'on calcule le coefficient de la transvection de manière à l'annuler). On choisit alors d'affecter la cellule correspondante à 0 et ce sans effectuer de calcul.

Ce problème se présente aussi dans les tests unitaire lorsque l'on compare le produit des trois matrices PLU avec la matrice dont on a calculé la décomposition. Il faut donc comparer les arrondis de chaque nombre.

3 Résolution de systèmes de Cramer

La dernière partie de ce projet consite donc à utiliser la décompositon PLU afin de résoudre des systèmes de Cramer. Dans les fonctions nécessaires à cela, aucune optimisation n'a été apportée, en effet leur temps d'exécution étant négligeable comparé au temps nécessaire au calcul de la décomposition, le gain de temps serait lui aussi négligeable.

Des tests ont été effectués afin de comparer les résultats obtenus en partant de la décomposition de Sage et en partant de notre décomposition. Ils sont bien entendu identiques, signe que l'intégralité du projet est fonctionnel.