PROG2: Inversion de matrices

Rémi Hutin Rémy Sun

26 février 2016

Résumé

Introduction

1 Implémentation de matrices

1.1 Définition

On définit une classe *Matix* possédant trois champs privés : *size_i* contenant le nombre de lignes de la matrice, *size_j* donnant le nombre de colonnes et *contents* correspondant à la matrice des valeurs de la matrice à proprement dit.

Le champ *contents* contient un double vecteur de *scalar_t* pour représenter la matrice des valeurs.

1.2 Méthodes et fonctions

On définit les méthodes publiques suivantes :

— *get_size_i* et *get_size_j* permettent de consulter la valeur des *size_i* et *size_j*
— $set(j, j, x)$ permet de remplacer la valeur d'indice (i, j) dans la matrice par x.
— $get(i, j)$ permet de consulter la valeur d'indice (i, j) dans la matrice
— $print()$ affiche la matrice dans le terminal en formattant correctement l'affichage.

De plus, on définit en plus les fonctions suivantes :

— Des opérateurs $+, -$ qui effectuent la somme et la différence de deux matrices en effectuant un simple parcours de la matrice
— Un opérateur $*$ qui posséde deux sens différents : si le premier argument est un *scalar_t* alors on effectue juste la mutltiplication scalaire de la matrice comme pour les opérateurs précédents. Si c'est une matrice, on effectue une multiplication matricielle.
— Une fonction *transpose* qui crée la matrice transposée par simple parcours des valeurs de la matrice.
— Une fonction *Id* qui crée une matrice identité.
— Une fonction *submatrix* qui crée explicitement la sous-matrice obtenue en retirant une ligne et colonne spécifiée. (voir section 1.3)

— Une fonction *determinant* récursive qui calcule le déterminant par utilisation du développement par ligne.
— Une fonction *inverse* qui calcule la matrice inverse.

1.3 Extraction de sous-matrice

Il est nécessaire d'extraire une sous-matrice en retirant les lignes d'indice a et les colonnes d'indice b.

Ceci est fait en créant une matrice carrée de taille $(size_i - 1, size_j - 1)$, qu'on remplit par parcours de cette matrice en tirant parti du fait que l'expression $(i \leq a)$ renvoie 1 si $i \leq a$ ce qui permet d'engendrer un décalage de ligne/colonne quand nécessaire.

1.4 Erreurs de calcul d'inverse

Le calcul de l'inverse est sujet à des erreurs d'arrondis. De fait, il est nécessaire de mesurer l'erreur commise.

La mesure de l'erreur a été défini par la norme de $M \bullet M' - I_n$ où M' est la matrice inverse calculée par le programme proposé.

Pour $n \in \{10; 20; \ldots; 100\}$ on génère 10 matrices aléatoires de taille n et on récupère l'erreur moyenne.

Les résultats sont consignés dans le graphe suivant.

2 Optimisation : implémentation des sous-matrices

Jusqu'à maintenant, chaque matrice possédait un champ contents correspondant à un double vecteur de valeurs.

Cela veut notament dire que lorsqu'on veut extraire une sous matrice, il fallait pour créer la nouvelle sous-matrice recréer presque l'integralité de ce champ contents.

Ainsi, le coût en mémoire devenait très vite très important comme montré à la section précédente.

Nous allons maintenant présenter une façon de ne pas recréer un champ de valeurs.

2.1 Définition de matrices par vecteurs de bits d'activité

Nous passons contents en attribut static de la classe Matrix : toutes les matrices partagent le même champ contents.

L'idée est de faire de contents un triple vecteur telle que la première coordonnée repère l'index de la matrice considérée.

Ainsi, à chaque fois qu'une nouvelle matrice (à part entière) est crée, on ajoute son double vecteur de valeurs au vecteur contents et on passe en champ son index dans contents.

De plus, pour chaque matrice, on crée deux vecteurs lines et rows qui repèrent quelles lignes et colonnes de la matrice repérée par index dans contents sont "actives".

A l'origine, toutes les lignes et colonnes sont actives.

2.2 Redéfinition des méthodes de base

La plus grosse difficulté posée par cette construction est que l'accès à la valeur (i,j) de la matrice n'est plus direct : certaines lignes et colonnes sont "désactivées". Pour remédier à cela, on crée une méthode $find\_index$ qui fait un parcours du vecteur de bits de maniére à trouver le "vrai" i/j.

Cela se fait en déclarant un compteur count, puis en faisant une boucle sur une variable k telle qu'à chaque itération count augmente ssi le bit d'index k est positif. Quand count > index, on arrête la boucle et on récupére le k courant qui correspond à $real\_index + 1$. A partir de là, il devient très simple de redéfinir toutes le méthodes de base de Matrix en prenant soin de se rappeler que contents est maintenant un triple vecteur.

2.3 Création de sous-matrices

La création de sous-matrice devient très simple : on crée une copie de la matrice en question, puis on modifie les vecteurs rows et lines pour refléter le changement dans la matrice considérée.

Cela présente le grand avantage de ne pas explicitement créer un nouveau double vecteur de valeurs pour la sous-matrice. Sachant que la fonction *determinant* fait n ! appels à la fonction *submatrix* les économies en temps (et mémoire) sont substantielles

2.4 Comparaisons avec l'ancienne méthode

Nous avons annoncé à la partie précédente que cette nouvelle façon de procéder permet de faire des économies en temps de calcul.

Pour vérifier cette affirmation, on a effectuée l'expérience suivante :

Pour $n \in \{10; 20; \ldots; 100\}$ on génère 10 matrices aléatoires de taille n et on compare les temps d'exécution de l'ancienne méthode et de la nouvelle.

Les résultats sont consignés dans le graphe suivant.

2.5 Fuites de mémoire

Il est important de remarquer que notre implémentation à l'aide d'un vecteur statique n'est pas sans poser de problémes de fuites mémoires. En effet, la déclaration de nouvelles matrices dans une boucle peut causer une augmentation définitive du champ static *contents*.

```
for (i=0; i < 1000; i++)
    Matrix M(10,10)
```

Avec l'ancienne implémentation, M est déclarée puis oubliée à chaque tour de boucle.

Avec notre nouvelle implémentation à l'aide du type *static*, la matrice de valeur stockée demeure dans *contents*.

Il est donc nécessaire de définir un moyen d'effacer des matrices de *contents* pour au moins minimiser les coûts en mémoire.

Il a donc été crée un vecteur *count* qui repére pour chaque matrice de valeurs stockée le nombre de matrices s'y référant. A chaque fois qu'une matrice ou sous-matrice est crée, on incrémente le compteur correspondant. On surcharge le destructeur pour qu'il décrémente ce compteur.

2.6 Insuffisances

Cette méthode présente cependant un gros inconvénient : les sous-matrices ainsi déclarées ont un contenu directement liée à celui des matrices mères. Si on modifie leur contenu, on modifie aussi celui de la matrice mère. et vice-versa.

Cela ne pose pas de problème dans le projet proposé puisque l'application qui en est faite agit en place sur le problème et ne modifie pas les valeurs des matrices.

En réalité, cela vient du fait que ce qu'on a voulu faire à la base est d'éviter d'expliciter des sous-matrices qu'il n'était pas nécessaire d'expliciter puisqu'on ne faisait que consulter leur valeur. Si on veut pouvoir travailler sur ces sous-matries, il faudra définir une nouvelle méthode d'extraction de sous-matrices et on ne gagnera rien par rapport à la première façon de procéder.

Conclusion