TP6 : Polymorphisme sur des matrices

1 Présentation du problème

On souhaite implémenter plusieurs classes de matrices qui garderont leurs coefficients en mémoire de manière adaptée à chaque classe. De même, ces matrices calculeront les produits matrice/vecteur de manière adaptée. Le but est de réduire l'espace mémoire utilisé pour stocker les coefficients et de diminuer le nombre d'opérations nécessaires pour le produit matrice/vecteur. On considérera les types de matrices suivants :

— un type "matrice pleine",
— un type "matrice identité".

Par exemple, on connaît les coefficients de la matrice identité. Il n'y a donc pas besoin d'utiliser de l'espace mémoire pour les stocker. De même, le produit d'une matrice identité avec un vecteur ne nécessite aucun calcul.

Les matrices pleines utiliseront le même stockage et les mêmes opérations que celles mises en place dans le DM. Les matrices identité ne stockeront aucun coefficient et renverront leur produit avec un vecteur comme suit : $\mathbb{I} \times X = X$.

2 Mise en œuvre du polymorphisme

Pour permettre à toutes ces matrices de partager une même interface, on définit une classe abstraite *virtMat*. Les matrices présentées ci-dessus seront représentées par les classes *fullMat* et *idMat* qui dériveront de *virtMat*.

Le code correspondant à *fullMat* vous est fourni. **Vous devrez coder** *idMat* et *virtMat* dans les fichiers *matrices.hpp* et *matrices.cpp*. Pour simplifier, on ne vous demande pas de coder l'opérateur de copie de matrices et le constructeur par copie.

3 Application : la méthode du gradient conjugué

Plusieurs algorithmes existent pour résoudre le système linéaire

$$AX = B, \tag{1}$$

où l'on cherche $X \in \mathbb{R}^n$ étant donnés une matrice inversible $A \in \mathrm{GL}_n(\mathbb{R})$ et un vecteur $B \in \mathbb{R}^n$ ($n \in \mathbb{N}^*$). Parmi eux, nous citons l'algorithme du gradient conjugué (cf Algorithme 1). Il s'agit d'une méthode itérative qui fournit la solution du problème à $\varepsilon > 0$ près (que l'on choisit) sous condition que la matrice soit symétrique définie positive.

Algorithm 1 La méthode du gradient conjugué

1. Choisir un premier vecteur $X_0 \in \mathbb{R}^n$ et poser $k := 0$. On calcule également le résidu $r_0 := AX_0 - B$ et la direction de descente $p_0 := -r_0$.

while $\|r_k\|_{\ell^2}^2 > \varepsilon$ et $k < N$ **do**

2. On calcule l'"amplitude" de descente $\alpha_k := -\dfrac{r_k^T p_k}{p_k^T A p_k}$.
3. Le nouveau vecteur est $X_{k+1} := X_k + \alpha_k p_k$.
4. On calcule le résidu $r_{k+1} := r_k + \alpha_k A p_k$.
5. La nouvelle direction de descente est $p_{k+1} := -r_{k+1} + \dfrac{r_{k+1}^T r_{k+1}}{r_k^T r_k} p_k$.
6. On pose $k := k + 1$ et on teste la condition de la boucle "while".

end while

Dans l'algorithme 1, $\varepsilon > 0$ est l'erreur en-dessous de laquelle on arrête les itérations. De même, $N > 0$ est le nombre d'itérations maximales qu'on s'autorise de faire.

La fonction *GC* fournie s'applique à des matrices pleines. Il faudra faire en sorte qu'elle puisse prendre en argument une *virtMat*. On comparera le temps de résolution d'un système linéaire faisant intervenir la matrice identité en fonction de si cette matrice est une *fullMat* ou une *idMat*.

4 Instructions

1. Construire une classe abstraite *virtMat* de manière à ce que la fonction *GC* puisse prendre une *virtMat* en argument au lieu d'une *fullMat*. Il faudra

faire en sorte que *fullMat* dérive de *virtMat*. Vérifier que l'on obtient le bon résultat sur l'exemple suivant :

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}, \; B = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \text{ et } X = \frac{1}{4} \begin{pmatrix} 3a - b - c \\ -a + 3b - c \\ -a - b + 3c \end{pmatrix}, \qquad (2)$$

avec a, b, $c \in \mathbb{R}$.

2. Écrire une classe *idMat* qui dérive de *virtMat* et représente une matrice identité.
3. Résoudre un système linéaire dont la matrice est l'identité. On comparera l'utilisation de *idMat* et de *fullMat*. Utiliser une taille de matrice qui permette de voir une différence en terme de temps de résolution. Voyez-vous l'utilité des classes abstraites ? Bonus : comparer l'espace mémoire utilisé.

5 Autre bonus : une classe de matrices creuses

Lorsque la plupart des coefficients d'une matrice K sont nuls, il est avantageux d'utiliser une classe adaptée pour les stocker. Cela permet de réduire l'espace mémoire utilisé ainsi que le temps des opérations liées à cette matrice et en particulier le temps de résolution des systèmes linéaires.

Nous détaillons maintenant le format "Yale Sparse Matrix". On note N^* le nombre de coefficients non nuls dans la matrice. Un premier tableau *A* contient les valeurs des coefficients non nuls de la matrice (de gauche à droite puis de haut en bas). Un deuxième tableau noté *IA* de taille "le nombre de lignes plus 1" est défini comme *IA[0] = 0* et *IA[i+1] = IA[i] + NB_i* avec *NB_i* le nombre de coefficients non nuls sur la i^e ligne. Un troisième tableau noté *JA* de longueur N^* contient le numéro de colonne de chaque coefficient non nul.

Par exemple la matrice

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 6 & 0 & 0 \end{pmatrix},$$

peut être stockée sous la forme

$$\begin{aligned} A &= \begin{pmatrix} 1 & 2 & 3 & 9 & 1 & 4 & 1 & 3 & 5 & 6 \end{pmatrix}, \\ IA &= \begin{pmatrix} 0 & 2 & 4 & 4 & 6 & 6 & 7 & 8 & 10 \end{pmatrix}, \\ JA &= \begin{pmatrix} 0 & 1 & 1 & 6 & 2 & 7 & 5 & 4 & 2 & 5 \end{pmatrix}. \end{aligned}$$

1. Coder ce format dans une classe *sparseMat*.
2. Comparer le temps de résolution d'un système linéaire en fonction de si la matrice considérée est une *sparseMat* ou une *fullMat*. On pourra par exemple considérer

$$\begin{pmatrix} 2 & 1 & & & \\ 1 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ \vdots \\ \vdots \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ \vdots \\ 4 \\ 3 \end{pmatrix}. \tag{3}$$