
TD 6

Variations sur les matrices

6.1 Objectif

Cet exercice prolonge le précédent (5) et nécessite d'en avoir réalisé la classe `Matrix`. Vous trouverez, à l'endroit habituel, dans le répertoire `Linear_Algebra`, la correction de la classe `MVector`, mais vous aurez à terminer la classe `Matrix`.

L'exercice illustre la dérivation de classes (héritage) en C++ et les problèmes de construction et de copie y afférents.

Cet exercice ne clôt pas la représentation des matrices en C++. On lira en particulier avec profit la section 6.4.3.

6.2 Diverses sortes de matrices

6.2.1 Matrices carrées

Une matrice carrée est une matrice, elle en a toutes les propriétés et tous les opérateurs. Elle a sûrement des propriétés supplémentaires (comme des valeurs propres par exemple) mais nous ne nous en préoccupons pas ici.

On vous demande donc d'écrire une classe représentant des matrices carrées, nommée `SquareMatrix`, dérivant de `Matrix` et utilisant, dans la mesure du possible, les propriétés de cette dernière.

Vous apporterez une attention toute particulière aux opérateurs arithmétiques (+, -, *) et à leur typage : en particulier remarquez que la somme (la différence, le produit) de deux matrices carrées est une matrice carrée ; ou encore que le produit d'une matrice carrée par une matrice rectangulaire (de nombre de lignes convenable) n'est qu'une matrice rectangulaire¹.

6.2.2 Matrices diagonales

Les matrices diagonales sont un cas particulier de matrices carrées. On vous demande donc d'écrire une classe `DiagonalMatrix` les représentant. Elles doivent pouvoir, entre autres, se construire à l'aide d'un `MVector` qui sera la valeur de leur diagonale :

```
// construit un vecteur de 10 composantes, toutes égales à 1
MVector mv(10, 1.0);
// construit la matrice identité 10x10
DiagonalMatrix dmat(mv);
```

Ici encore, vous apporterez une attention toute particulière aux opérateurs arithmétiques (+, -, *) et à leur typage : en particulier remarquez que la somme (la différence, le produit) de deux matrices diagonales est une matrice diagonale, mais que la somme d'une matrice diagonale et d'une matrice carrée n'est qu'une matrice carrée². Par ailleurs le calcul de la somme (différence, produit) de deux matrices diagonales peut être simplifié par rapport à celui des matrices ordinaires (rectangulaires ou carrées).

1. En tous cas, en ce qui concerne le *type*. Une matrice rectangulaire (`Matrix`) pouvant être carrée (« par hasard » à l'exécution) la matrice résultat peut elle-même être de fait carrée ; il n'empêche que pour le compilateur, qui ignore ce qui va se passer à l'exécution, son type ne peut être que celui d'une matrice ordinaire (`Matrix`).

2. Même remarque que précédemment.

6.2.3 Matrices scalaires

Une matrice scalaire est une matrice diagonale dont tous les éléments diagonaux sont égaux (elle est de la forme $\lambda \mathbf{I}_n$ où \mathbf{I}_n est la matrice identité de dimension n). Il s'agit donc d'un cas particulier de matrice diagonale. Une telle matrice doit pouvoir être construite, entre autres, à partir de sa dimension et de la valeur commune des éléments diagonaux :

```
int n = 10;
ScalarMatrix smat(n, lambda); // construit  $\lambda \mathbf{I}_n$ 
```

Ici encore la somme de deux matrices scalaires est scalaire, mais celle d'une matrice scalaire et d'une matrice diagonale n'est que diagonale, et celle d'une matrice scalaire et d'une matrice carrée n'est que carrée... Par ailleurs l'algorithme des opérations entre matrices scalaires est particulièrement simple par rapport à celui des autres types de matrices.

6.3 Question pour les meilleurs : métamorphoses des matrices

Si la hiérarchie que nous vous avez établie prend bien en compte un typage statique des matrices et de leurs opérations arithmétiques, elle ignore complètement le problème de la « métamorphose » des matrices. Cette métamorphose peut se présenter de deux manières :

1. Les opérateurs d'indexation (`operator[]`, `operator()` et fonctions-membres `at()`) peuvent changer la nature d'une matrice ; ainsi avec la matrice `dmat` précédente (qui était par ailleurs égale à la matrice identité de dimension 10), l'affectation

```
dmat(2, 3) = 5.0
```

lui fait perdre sa nature diagonale !
2. Par ailleurs, on pourrait souhaiter définir des conversions entre les différents types de matrices quand leur structure interne le permet ; ainsi si l'on a

```
Matrix mat(5, 5);
```

on pourrait souhaiter convertir `mat` en une matrice carrée (`SquareMatrix` de dimension 5) ou même diagonale (`DiagonalMatrix` de diagonale nulle) ou scalaire (`ScalarMatrix` de valeur 0.0) !

Le premier point est le plus crucial (si on ne fait rien, c'est un vrai bug !) mais aussi le plus délicat à résoudre. Une solution brutale serait d'interdire l'indexation pour *modifier* un élément (mais de conserver la possibilité d'indexation pour *lire* la valeur) : comment réaliseriez vous cela ? *Évidemment une telle solution serait un pis-aller* et, de toutes façons, si vous l'essayez vous constaterez que cela casse pas mal de choses ! On peut imaginer des solutions plus satisfaisantes : par exemple, interdire la modification d'une matrice diagonale en dehors de sa diagonale ; ou encore, changer dynamiquement le type d'une matrice diagonale lorsque l'on crée un élément non nul en dehors de la diagonale... Ces solutions sont, pour l'instant, hors de votre portée (voir cependant 6.4.3). Si vous avez une idée, exprimez la, mais n'essayez pas de la réaliser tout de suite.

En ce qui concerne le second point, il est un peu plus facile d'autoriser certaines conversions. Essayez en particulier d'écrire un constructeur de la classe `SquareMatrix` qui permette de convertir (implicitement) une matrice ordinaire en matrice carrée si la matrice ordinaire est effectivement carrée et qui, sinon, lève une exception. Que deviennent alors les expressions de la forme `sqmat + mat` où `sqmat` est une matrice carrée et `mat` une matrice ordinaire ? Est-ce normal ? Préfixez la déclaration de votre constructeur par le mot clé `explicit`, cela devrait régler le problème (mais cela vous forcera à expliciter la conversion par un `cast` (`static_cast`)).

6.4 Remarques et conseils

6.4.1 Code fourni

Le répertoire `Linear_Algebra_2`, à l'endroit habituel contient un exemple de programme de test (`main_SquareMatrix.cpp`) des différents types de matrices. Il n'est utilisable (ainsi que la `Makefile`)

- que si la hiérarchie de fichiers établie dans les séances de TD précédents est respectée : en particulier, vérifiez que vous avez bien le fichier `default.mk` et le répertoire `include` (avec son contenu !) deux niveaux de répertoires au-dessus de votre répertoire de travail (`../.. / default.mk` et `../.. / include`, donc) ;
- que si vous respectez les noms de classes indiqués dans ce sujet ainsi que l'organisation des fichiers indiquée plus bas (sinon vous devrez éditer la `Makefile` et le programme de test)..

Le fichiers `tst_SquareMatrix.out` contient le résultat de l'exécution du programme de test avec ma propre solution.

6.4.2 Conseils

Méthode de développement Les mêmes remarques que lors des TDs précédents s'appliquent. Essayer de travailler *incrémentalement*. Définissez les méthodes et les classes progressivement, dans un ordre compatible avec les programmes de test. Testez au fur et à mesure, en commentant les parties du programme de test correspondant à des fonctionnalités non encore implémentées.

Par ailleurs ne définissez pas les fonctions automatiquement synthétisées par C++ si cette définition automatique vous agréée et évitez les duplications de code.

Organisation des fichiers Prenez de bonnes habitudes ! Bien que cela ne soit en rien obligatoire en C++, on place en général une seule classe C++ par « module ». Un module est constitué (ici aussi, *en général*) de deux fichiers sources dont le nom de base est celui de la classe : un fichier d'entête (d'extension `.h`) qui contient la définition de la classe, un fichier `.cpp` qui contient la définition du corps des fonctions membres et amies de la classe. Bien entendu, le fichier `.cpp` n'est pas nécessaire si l'ensemble de la définition de la classe et de ses fonctions peut-être placé dans le fichier `.h` : les fonctions sont alors `inline` et ceci doit être réservé à des fonctions simples et courtes, pour lesquelles il n'apparaît pas souhaitable de payer le coût d'une séquence d'appel et de retour de fonction.

Inspirez-vous de la `Makefile` fournie pour avoir une idée des fichiers que vous avez à produire.

6.4.3 Limites de l'exercice

Ces différentes classes de matrices peuvent ne pas apparaître complètement satisfaisantes à un programmeur rigoureux et soucieux aussi bien d'abstraction que d'efficacité (c'est à dire un vrai programmeur, en C++ ou tout autre langage !).

Du côté positif, la hiérarchie de classes que nous avons constituée reflète bien la relation logique (abstraite) entre les différentes sortes de matrices et assure le typage correct des différentes opérations. Elle n'est évidemment pas complète, ni en ce qui concerne les types possibles (il y a bien d'autres sortes de matrices, creuses, bandes, symétriques, triangulaires, etc.), ni en ce qui concerne les opérations (calcul du rang, déterminant, inversion, calcul de valeurs propres, triangulation, diagonalisation). Il paraît cependant assez facile d'ajouter ces propriétés.

Le plus grand problème est certainement celui de l'efficacité, en temps (trop de copies ?) et surtout, en mémoire. En effet, le stockage des éléments de nos matrices ne dépend pas du type de ces dernières : elles héritent toutes leur implémentation de la classe `Matrix`, et donc allouent $m \times n$ éléments (ou n^2 si elles sont carrées) quelle que soit leur nature. Cela a l'avantage de faciliter les conversion en montant dans la hiérarchie, par exemple de `DiagonalMatrix` à `SquareMatrix`. Cependant, on pourrait souhaiter n'allouer que les éléments nécessaires (une demie matrice pour une matrice triangulaire ou symétrique, un vecteur pour une matrice diagonale, un simple `double` pour une matrice scalaire, etc.). Mais alors les conversions implicites d'héritage ne fonctionneraient plus aussi bien, ce qui veut dire qu'il nous faudrait réviser complètement notre hiérarchie de classes ! Et il ne faudrait pas que cette réorganisation nous fasse perdre la vision logique de la hiérarchie des types de matrices...

Nous reviendrons sur cette question (et sur l'exemple des matrices) dans la seconde partie du cours, lorsque nous aborderons l'étude des schémas de conception (*Design Patterns*). Nous essayerons alors de donner une solution plus satisfaisante tout en conservant les propriétés abstraites de nos objets. Il nous faudra alors apporter également une réponse aux questions importantes soulevées en 6.3.