Développement d'une librairie matricielle

Objectif : *développer une librairie d'algèbre matricielle en C++. Étant donné que cette librairie effectue un usage systématique de la spécialisation de patrons, la totalité du code est implémentée dans des fichiers d'entêtes.*

Votre implémentation va inclure les éléments suivants :
- Spécialisation de patrons pour le stockage des matrices par lignes et par colonnes.
- Spécialisation de patrons pour les matrices à taille fixes et à taille dynamique.
- Surcharge des opérateurs arithmétiques pour les opérations matricielles.
- Optimisation des opérations matricielles en fonction du stockage (par lignes ou par colonnes) de la matrice.

Important :
- votre implémentation doit éviter les calculs non pertinents, les copies inutiles et les allocations de mémoire superflues.
- votre code doit compiler sous *Visual Studio 2019* en utilisant les versions *Release* et *Debug*[1].

Des tests unitaires sont fournis pour tester automatique votre code. Ces tests vous permettent d'évaluer votre progression, mais ils sont volontairement incomplets. À vous de les compléter.

La librairie `Google Test` est déjà incluse dans le projet fourni.
- Si vous travaillez sous Windows, avec *Visual Studio*
  1. Dans le dossier `GTI320-Labo1`, identifier le fichier `labos.sln` et l'ouvrir avec *Visual Studo 2017* ou *2019*.
  2. Dans *Visual Studio*, faire CTRL-B pour compiler et F5 pour exécuter le programme.
- Si vous travaillez avec un terminal et l'outil `cmake` :
  1. Ouvrir un terminal et se rendre dans le dossier `GTI320-Labo1`.
  2. Entrer la commande suivante :

     ```
     $ mkdir build && cd build && cmake ..
     ```

  3. Ensuite, faire `make` pour compiler et `./main` pour exécuter le programme.

1. Menu `Générer` → `Gestionnaire de configurations...`

Tests unitaires : vous ne devez pas modifier les tests fournis. Pour ajouter de nouveaux tests, il suffit d'ajouter une fonction respectant la syntaxe :

```
TEST(TestLabo1, Supplementaires)
{
  /* écrivez vos tests ici */
}
```

Présentation du projet

Le stockage de matrice

La classe `DenseStorage` est une classe permettant de stocker des données matricielles. Les patrons (*templates*) de la classe permettent de stocker les données dans un tableau de taille fixe ou alloué dynamiquement. Les tableaux à taille dynamique peuvent être redimensionnés en cours d'exécution. Un deuxième patron permet d'utiliser un objet `DenseStorage` pour stocker des éléments de type `float` ou `double`.

Complétez le fichier `DenseStorage.h` de manière à compléter les spécialisations de patrons pour :

- les matrices de taille fixe
- les matrice de taille dynamique

Important : les éléments des matrices à taille fixe doivent **obligatoirement** être situés dans la pile d'exécution alors que les éléments des matrices à taille dynamique doivent être stockés sur le tas.

Note : les matrices à taille dynamique peuvent être redimensionnées. Tout changement de taille entraîne une réallocation de mémoire. Il n'est pas pertinent de copier les données car le résultat serait de toute façon incohérent.

Les matrices et les vecteurs

La classe `MatrixBase` regroupe les éléments communs aux matrices et aux vecteurs. La classe possède une instance de `DenseStorage`.

La classe `Vector` implémente un vecteur de $\mathbb{R}^n$. Les vecteurs à taille fixe et à taille dynamique sont implémentés. Vous devez entre autres implémenter :

- L'opérateur de copie : `operator=(const Vector&)`
- L'opérateur d'accès aux éléments : `operator()(int)`
- Le produit scalaire : `dot(const Vector&)`

La classe `Matrix` implémente une matrice de $\mathbb{R}^{m \times n}$. Les matrices à taille fixe et à taille dynamique sont implémentées. Vous devez entre autres implémenter :

- L'opérateur de copie : `operator=(const Matrix&)`
- L'opérateur d'accès aux éléments : `operator()(int, int)`
- La transposée : `transpose()`

Important : Toutes les fonctions à implémenter ne sont pas listées ici. Dans les fichiers sources, les fonctions que vous devez compléter sont toutes identifiées par le commentaire : `// TODO`.

Important : Tous les `TODO` doivent être complétés et sont évalués.

Implémentation d'opérateurs arithmétiques

Le fichier `Operator.h` contient les implémentations des opérations arithmétiques vectorielles et matricielles. On y retrouve les opérations suivantes :

- Multiplication d'un vecteur par un scalaire : `operator*(Scalar, const Vector&)`
- Multiplication d'un vecteur par une matrice : `operator*(const Matrix& , const Vector&)` avec spécialisation selon le type de stockage de la matrice (par lignes ou par colonnes).
- Multiplication et addition de deux matrices : `operator*(const Matrix&, const Matrix&)` et `operator+(const Matrix&, const Matrix&)` avec des spécialisations de templates pour certaines combinaisons de stockage en lignes et en colonnes.
- Addition de vecteurs `operator+(const Matrix&, const Matrix&)`

Important : Encore une fois, vous devez compléter toutes les fonctions marquées d'un `TODO` dans le code fourni.

Les mathématiques 3D

Le fichier `Math3d.h` contient de types, des déclarations et des spécialisations pour les structures de données utilisées dans les applications 3D. On y retrouve entre autres :

— les matrices de transformation en coordonnées homogènes (`Matrix4d/4f`).
— les matrices de rotation (`Matrix3d/3f`).
— les vecteurs 3D (`Vector3d/3f`).

Certaines fonctions doivent être redéfinies à l'aide de spécialisations de patrons de manière à optimiser le calcul. Entre autres :

— la multiplication d'une matrice homogène avec un vecteur 3D (`operator*(const Matrix4&, const Vector3&)`
— le calcul de la matrice inverse d'une rotation.
— le calcul de la matrice inverse d'une rotation en coordonnées homogènes.

Important : vous devez compléter toutes les fonctions marquées d'un `TODO` dans le code fourni.

Tests unitaires additionnels

Implémentez vos propres tests unitaires pour vérifier ou valider les cas non considérés par les tests fournis. Vous pouvez également implémenter des fonctions supplémentaires ou des classes supplémentaires qui étendent les fonctionnalités de la librairie. Par contre, assurez-vous que vos modifications ne *cassent* pas les tests existants. De plus, ne testez pas une deuxième fois ce qui a déjà été testé.

—

Grille d'évaluation

Évaluation	Détails	Pondération
Stockage des matrices	Taille fixe vs taille dynamique, redimensionner, allocation sur le tas vs sur la pile	15
Opérations vectorielles	Produit scalaire, norme, `operator()(int)`, `operator=`	10
Opérations matricielles	`operator()(int,int)`, `operator=`, transposée.	10
Opérateurs arithmétiques	`operator*(Scalar, const Vector&)`, `operator*(const Matrix&, const Vector&)`, `operator*(Scalar, const Matrix&)`, `operator*(const Matrix&, const Matrix&)`, `operator+(const Matrix&, const Matrix&)`, `operator+(const Vector&, const Vector&)`	20
Sous-matrices	`operator()(int,int)`, `operator=(const Matrix&)`, transposée.	10
Mathématiques 3D	transformations homogènes, rotations, inverse.	15
Tests unitaires supplémentaires	Au moins trois, chaque test doit être adéquatement documenté.	10
Style de programmation	En entête de chaque fichier : nom, code permanent et courriel. Indentation du code et commentaires. Code clair et concis.	10
Total		**100**