**Travaux pratiques**
**1 Bibliothèque Matrix**

Notre objectif final est de créer un logiciel de visualisation des images en trois dimensions. Dans chaque TP de la discipline, nous allons créer une partie de ce logiciel.

Nous allons construire notre logiciel au fur et a mesure des TP. Donc, nous devons commencer le travail de forme organisé, de manière a pouvoir continuer sur les TP suivants. Vous n'êtes pas obligé à suivre à la lettre, mais voici un exemple d'organisation des vos fichier pour ce TP, et les prochains :

```
logiciel/
   |
   +- bin/      (exécutables)
   +- build/    (objets -- .o)
   +- include/  (entêtes -- .h)
   +- src/      (implémentations - .cpp)
   +- test/     (routines de test -- .cpp)
   +- makefile
```

Au fur et à mesure des TP, nous allons remplir cette structure pour créer le logiciel complet.

Pour ce premier TP, nous allons créer une bibliothèque de calculs vectoriels et matriciels. Cette bibliothèque sera fondamentale pour notre logiciel. La bibliothèque doit être crée en C++.

Pour bien organiser les choses, vous devez créer un fichier en-tête (ex. `libmatrix.h`) où vous allez définir l'interface de votre bibliothèque. L'implémentation doit être définie dans un autre fichier (ex. `libmatrix.cpp`). Toutes les classes et fonctions de la bibliothèque doivent être crées dans un espace de nom (ex. `libmatrix`). Vous devez aussi penser à créer des routines de test pour votre bibliothèque. La bibliothèque doit au moins définir les classes, méthodes, fonctions et constantes décrits ci-dessous.

**Classe Vector :** Définie une séquence de taille fixe de $n$ valeurs de type $T$, où $n$ et $T$ sont des paramètres de la classe. La classe doit au moins définir les méthodes suivantes :

- **Méthodes :**
  - `at()` : adresse le i-ème élément ($i$ passé en argument) du vecteur.
    Lève une exception en cas d'erreur d'indice.
  - `cross()` : produit vectoriel avec un autre vecteur (passé en argument). Utilise uniquement les 3 premières coordonnées. Lève une exception si le vecteur possède moins de 3 coordonnées.
  - `is_ortho()` : retourne `true` si le vecteur est orthogonal à un autre passé en argument et `false` sinon.
  - `is_null()` : retourne `true` si le vecteur contient une valeur invalide et `false` sinon.
    Notamment, si le vecteur possède des valeurs `nan`.
  - `is_unit()` : retourne `true` si le vecteur est unitaire et `false` sinon.
  - `norm()` : retourne la norme du vecteur.
  - `to_unit()` : retourne une copie du vecteur normalisée.
  - opérateur << : surcharge l'opérateur pour les outputs.
  - opérateur `[]` : adresse le i-ème élément ($i$ passé en argument) du vecteur.
    Doit également permettre l'affectation.
    Ne vérifie pas si $i$ est valide.
  - opérateur + : addition de vecteurs.
  - opérateur += : addition de vecteurs et affectation.
  - opérateur - : définie deux opérations (surcharge) :
    - inverse du vecteur.
    - différence de vecteurs.
  - opérateur -= : différence de vecteurs et affectation.
  - opérateur * : définie cinq opérations (surcharge) :
    - multiplication de scalaire et vecteur.

    - multiplication de vecteur et scalaire.
    - produit scalaire de vecteur et vecteur.
    - multiplication de vecteur et matrice (voir la classe Matrix en bas).
    - multiplication de matrice et vecteur (voir la classe Matrix en bas).
  - opérateur *= : produit (toutes les possibilités ci-dessus) et affectation.

**Classe Matrix :** Définie une matrice de dimension fixe $(n, m)$ de valeurs de type $T$, où $n$, $m$ et $T$ sont des paramètres de la classe. La classe doit au moins définir les fonctions suivantes :

- **Méthodes :**
  - `at()` : adresse l'élément $(i, j)$ (passés en arguments) de la matrice.
    Lève une exception en cas d'erreur d'indice.
  - `inverse()` : retourne la matrice inverse de la matrice. Retourne une matrice nulle si la matrice n'est pas inversible.
  - `is_null()` : retourne `true` si la matrice contient une valeur invalide et `false` sinon.
  - `is_ortho()` : retourne `true` si la matrice est orthogonale.
  - `transpose()` : retourne la transposée de la matrice.
  - opérateur << : surcharge l'opérateur pour les outputs.
  - opérateur `[][]` : adresse l'élément $(i, j)$ (passées en arguments) de la matrice.
    Doit également permettre l'affectation.
    Ne vérifie pas si l'indice est valide.
  - opérateur + : addition de matrices.
  - opérateur += : addition et affectation.
  - opérateur * : définie cinq opérations (surcharge).
    - multiplication de scalaire et matrice.
    - multiplication de matrice et scalaire.
    - multiplication de vecteur et matrice.
    - multiplication de matrice et vecteur.
    - multiplication de matrice et matrice.
  - opérateur *= : multiplication (toutes les possibilités ci-dessous) et affectation.

**Fonctions :**

- `dot()` : alias pour l'opération *. Les deux opérandes sont passés en arguments. Les opérandes incluent les scalaires, vecteurs et matrices.
- `cross()` : alias pour la méthode `cross`. Les deux opérandes sont passés en arguments.

**Classe Vec2i** Définie un vecteur de 2 entiers.

**Classe Vec3i** Définie un vecteur de 3 entiers.

**Classe Vec4i** Définie un vecteur de 4 entiers.

**Classe Vec2r** Définie un vecteur de 2 réels.

**Classe Vec3r** Définie un vecteur de 3 réels.

**Classe Vec4r** Définie un vecteur de 4 réels.

**Classe Mat44r** Définie une matrice de dimension $(4, 4)$ de valeurs réels.

**Constantes** La bibliothèque Matrix doit encore définir les constantes suivantes :

- `zerovector` : un vecteur de zéros.
- `zerovec2i`
- `zerovec3i`
- `zerovec4i`

— `zerovec2r`
— `zerovec3r`
— `zerovec4r`
— `IdentityMat` : la matrice identité.
— `Identidy44i`
— `Identity44r`

**Exercice bonus :** Concevez et réalisez une expérience pour déterminer laquelle des deux méthodes vues en cours pour trouver la matrice inverse est la plus efficace (élimination de Gauss-Jordan ou avec le déterminant). Cette expérience doit être implémentée comme une routine de test. Après avoir tiré vos conclusions, implémentez la routine la plus efficaces de deux dans votre classe Matrix.