Sorbonne Université
2022–2023

M1 Mathématiques
MU4MA056 : Programmation en C++

Examen de première chance : mardi 9 mai 2023

— Durée : 2h. Le sujet comporte 6 pages.

— **Aucun document n'est autorisé.**

— **Tout système muni d'un processeur et de mémoire (téléphone, ordinateur, montre connectée,...) est strictement interdit.**

— Le sujet comporte deux sections indépendantes : tout d'abord des questions générales indépendantes sur les différentes parties du cours, ensuite un problème autour du procédé d'orthogonalisation de Gram-Schmidt. Les questions **ne** sont **pas** classées par ordre de difficulté.

1 Questions générales

Q1.1. **Syntaxe élémentaire.** Écrire un programme *complet* en C++, le plus simple possible, qui affiche, ligne par ligne, chaque nombre de 1 à 100 suivi de l'inverse de son carré, et, à la fin, affiche la somme de ces inverses de carrés, selon le modèle :

```
1   1.
2   0.25
3   ...
4   ...
100 0.0001
6   valeur_de_la_somme_a_afficher
```

Q1.2. **λ-fonctions, bibliothèque `algorithm` de la STL et entrées-sorties.**

(a) Qu'est-ce qu'une λ-fonction ? Quel est l'intérêt par rapport aux fonctions usuelles ?

(b) Un fichier `input.txt` contient 3000 mots en minuscules séparés par des espaces. On souhaite réaliser les tâches suivantes :
— lire le fichier dans une variable de type `std::vector<std::string>` ,
— calculer et afficher la longueur moyenne des mots,
— créer un conteneur qui contienne seulement les mots de longueur supérieure à la moyenne,
— passer l'initiale de chaque mot sélectionner en majuscule (Pour cela, nous vous mettons à disposition la fonction `std::toupper(minus)` qui renvoie le caractère `minus` passé en majuscule),
— les trier par ordre alphabétique,
— produire un fichier `output.txt` qui contient ces mots triés.

Écrire un programme aussi simple que possible qui réalise cela. Pour chaque tâche en dehors de la lecture et l'écriture, un bonus de points sera octroyé si vous utilisez `algorithm` et/ou `numeric` .

NB : un objet de type `std::string` fonctionne comme un objet de type `std::vector<char>` en pratique pour cette question (presque les mêmes propriétés et les mêmes outils).

(c) Écrire la commande de compilation avec `g++` et celle d'exécution.

(d) Quels autres conteneurs de la STL décrivent des n-uplets d'éléments ? Pourquoi y en a-t-il plusieurs ?

Q1.3. **Classes.** Dans un fichier `gamer.hpp`, nous définissons la classe

```
class Game {
        private:
                std::string Player1;//nom du joueur 1
                std::string Player2;//nom du joueur 2
                double score1;//score du joueur 1
                double score2;//score du joueur 2
                int nb_of_turns;
        public:
                ...
};
```

(a) Écrire la classe complète pour que le code suivant fonctionne :

```
Game G("Alex","Jean");//noms initialisés, scores à zéro, zéro tours de jeu
G.wins(1,3.);//le joueur 1 marque 3 points,le nombre de tours augmente de 1
G.wins(2,4.);//le joueur 2 marque 4 points,le nombre de tours augmente de 1
auto winner_name = G.the_winner_is();
        //renvoie le nom de celui qui a plus de points.
std::cout << "Number of turns already played" << G.turns() << "\n";
G.write("jeu.dat");/*crée un fichier "jeu.dat" et y écrit
Winner: Jean 4.
Loser: Alex 3.
after 2 turns.
*/
```

Attention à ne pas oublier les const et les éventuelles références. Dans le constructeur, vous écrirez le moins de code possible dans le bloc `{...}`.

(b) Dans le constructeur, à quoi correspondent les `:` après le prototype et avant le code entre `{ }` ?

(c) Comment s'appelle la méthode `turns()` ? Pourquoi vaut-il mieux écrire son code directement dans la classe ?

(d) Qu'est-ce qu'un destructeur ? À quel moment et comment est-il appelé ?

(e) Que faut-il écrire d'autre dans les fichiers `gamer.hpp` pour qu'il soit utilisable partout autant de fois que possible ?

Q1.4. **Compilation avec `g++`.**

(a) À quoi correspond l'option `-c` ?

(b) Quelles sont les étapes de compilations d'un template de fonction ?

(c) Comment obtenir de meilleures performances à partir d'une bibliothèque de calcul numérique ?

(d) Quels sont les avantages d'un programme compilé plutôt qu'interprété ?

2 Autour de l'algorithme d'orthogonalisation de Gram-Schmidt

Soit $\mathcal{V}$ un espace vectoriel réel muni d'un produit scalaire $\langle \bullet, \bullet \rangle : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$. Pour toute suite *libre* de m vecteurs $(v_0, \ldots, v_{m-1})$ (qui ne sont pas nécessairement orthogonaux ni orthonormés) de $\mathcal{V}$, il existe une suite *libre* de m vecteurs $(\widehat{v}_0, \widehat{v}_1, \ldots, \widehat{v}_{m-1})$ de $\mathcal{V}$ telle que :

— les $\widehat{v}_i$ sont deux à deux orthogonaux,

— pour tout $1 \leq k \leq m$, l'espace engendré par $v_0, \ldots, v_{k-1}$ est égal à l'espace engendré par $\widehat{v}_0, \ldots, \widehat{v}_{k-1}$.

La preuve est constructive et réside sur l'algorithme de Gram-Schmidt que nous décrivons à présent. Pour $k = 0$, on pose $\widehat{v}_0 = v_0$ puis, pour tout k variant de 1 à $m-1$, on définit récursivement :

$$\widehat{v}_k := v_k + \sum_{l=0}^{k-1} \lambda_{k,l} \widehat{v}_l \qquad \text{avec} \qquad \lambda_{k,l} = -\frac{\langle \widehat{v}_l, v_k \rangle}{\langle \widehat{v}_l, \widehat{v}_l \rangle}$$

Un calcul rapide montre que la famille ainsi construite satisfait les exigences requises. Les vecteurs obtenus ne sont a priori pas de norme 1. On peut ensuite, si on le souhaite, les orthonormer ou non en définissant les vecteurs :

$$\widehat{v}'_k = \frac{1}{\sqrt{\langle \widehat{v}_k, \widehat{v}_k \rangle}} \widehat{v}_k$$

Le but de ce problème est d'écrire un template de fonction qui effectue cette orthogonalisation sous différentes formes selon le type d'objets `C++` utilisés.

2.1 Écriture de différents *templates* de fonction

Nous supposerons dans la suite que les codes de cette section sont écrits dans un fichier `gram_schmidt.hpp` .

Q2.5. On souhaite prendre pour décrire les vecteurs de $\mathcal{V}$ (`Vect` dans le code) toute classe déjà existante qui permet des combinaisons linéaires de vecteurs sous la forme `u3=3*u1+u2` mais n'a pas nécessairement de produit scalaire prédéfini. On peut fournir pour cela un deuxième argument `s` de type générique `ScalProd` dont on suppose que `s(u1,u2)` renvoie bien le produit scalaire de `u1` et `u2` , tous deux de type `Vect` , sans les copier ni les modifier.

Écrire un *template* de fonction

```
template <class Vect, class ScalProd>
void gram_schmidt_in_place( std::vector<Vect> &  v ,const ScalProd & s);
```

qui *modifie* la famille `v` (qui contient les v_i) pour qu'à la fin `v` contienne la famille des $\widehat{v}_i$ à la place des vecteurs de départ.

Q2.6. Si la classe `Vect` possède déjà une méthode `double scal(const Vect &) const` telle que `u.scal(v)` calcule $\langle u, v \rangle$, on souhaiterait ne pas avoir à fournir de produit scalaire `s` . Pour cela, on modifie la définition précédente en ajoutant des valeurs par défaut

```
template <class Vect, class ScalProd = DefaultScal >
void gram_schmidt_in_place(
   std::vector<Vect> &  v ,const ScalProd & s = DefaultScal() );
```

où `DefaultScal` est une classe vide (sans champ privé) dont le seul but est de fournir via son template d'opérateur `()` un appel à la méthode `scal` de `Vect` de telle sorte que `s(a,b)`

renvoie `a.scal(b)` si `s` est de type `DefaultScal`. Compléter la ligne ci-dessus pour que tout fonctionne (sans rien ajouter d'autre) :

```
class DefaultScal {
public:   ..... (const Vect & v1, const Vect & v2) .....
};
```

Q2.7. **Plus général :** on suppose à présent que la famille de vecteurs `v` n'est plus décrite par un objet de type `std::vector<Vect>` mais par n'importe quelle sous-partie d'un conteneur ordonné d'objets de type `Vect` muni d'itérateurs de type STL et borné par deux itérateurs de début et de fin `b` et `e`. Écrire le *template* de fonction

```
template <class Iter,class ScalProd>
void gram_schmidt_in_place(Iter b, Iter e, const Scalprod & s);
```

pour qu'à la fin les vecteurs v_i du conteneur soient transformés en les $\widehat{v}_i$.

Q2.8. **Plus général encore :** plutôt que d'effacer les vecteurs d'origine v_i, on souhaite écrire les vecteurs $\widehat{v}_i$ dans un autre conteneur-cible dont le début est donné par l'itérateur `b_out` et dont on est certain qu'il contient assez de cases. Écrire le *template* de fonction

```
template <class Iter, class Iter_out, class ScalProd>
void gram_schmidt(Iter b, Iter e, Iter_out b_out, const ScalProd & s );
```

2.2 Un premier exemple dans $\mathbb{R}^d$ basé sur la STL

Nous proposons d'écrire, dans deux fichiers `vec.hpp` et `vec.cpp`, une classe aussi simple que possible pour décrire les espaces vectoriels de dimension finie via :

```
class V {
        private:
                std::vector<double> coeffs;
        public:
                ....
};
```

Dans toute cette partie, puisqu'il s'agira d'une bibliothèque de calcul, il sera **très important** d'étiqueter `const` ce qui doit l'être et d'éviter toute copie inutile.

Pour chaque question de cette section, vous préciserez si votre code est écrit dans `vec.hpp` ou `vec.cpp` et adapterez votre syntaxe selon la situation.

Q2.9. Ajouter un constructeur qui prenne comme argument un entier d un réel a et construit le vecteur $(a, a, \ldots, a) \in \mathbb{R}^d$.

Q2.10. Ajouter un template de constructeur qui prenne comme argument deux itérateurs quelconques et crée le vecteur dont les coefficients sont lus dans le conteneur (en supposant que les tailles correspondent)

Q2.11. Ajouter une méthode `scal` qui calcule le produit scalaire de deux vecteurs.

Q2.12. Ajouter, *sous la forme de fonctions*, un opérateur `+` (entre deux vecteurs) et un opérateur `*` (entre un réel et un vecteur) de telle sorte que

```
w = a*u +v;// u,v,w de type V, a de type double
```

calcule $au+v$ et le stocke dans w. Que faut-il faire pour ces fonctions accèdent aux champs privés `coeffs` ?

Q2.13. Écrire un accesseur à la i-ème coordonnée à être utilisé sous la forme `u(i)` .

Q2.14. Écrire un mutateur à la i-ème coordonnée à être utilisé sous la forme `u(i)` .

Q2.15. Quels sont les avantages et inconvénients à tester si l'indice `i` est bien compris entre 0 et $d-1$ (dimension) ?

Q2.16. Ajouter un opérateur `<<` qui affiche un vecteur comme $(3.1, 2., 4.5, 7.)$ sous la forme

```
4   3.1   2.   4.5   7.
```

où le premier 4 indique la dimension du vecteur et les nombres suivants les coordonnées.

Q2.17. Ajouter un opérateur `>>` qui lit un vecteur sous le format précédent. *Attention*, un changement de dimension reste possible.

Q2.18. Écrire un programme *complet* dans un fichier `prog.cpp` qui lit, via la classe `V` ci-dessus, 24 vecteurs (tous de même dimension, supposés libres) dont un fichier `data.txt` , chacun écrit sur un ligne, applique le procédé de Gram-Schmidt par l'une des fonctions précédentes et écrit les 24 vecteurs dont un fichier `ortho.txt` .

Q2.19. Deux inconvénients de la classe précédente sont qu'aucune optimisation n'est possible en petite dimension et qu'il reste possible d'ajouter par inadvertance deux vecteurs de dimension différente. Que proposez-vous pour résoudre cela efficacement ? Réécrire brièvement la classe (que la définition de la classe, pas le code des méthode) sous une nouvelle forme adaptée.

2.3 Génération d'une matrice orthogonale uniforme dans $O(N)$.

Le groupe $O(N)$ des matrices réelles orthogonales de dimension N telles que $M^{-1} = M^t$ peut être muni d'une mesure de probabilité uniforme. On peut voir une telle matrice M comme N vecteurs-colonnes de dimension N. Pour générer une telle matrice aléatoire uniforme sur $O(N)$, on dispose de l'algorithme suivant :

- on génère N vecteurs $(u_i)_{0\leq i<N}$ aléatoires de $\mathbb{R}^N$ indépendants dont chaque coefficient est indépendant des autres et suit une loi normale centrée réduite,
- on applique le procédé de Gram-Schmidt à ces vecteurs pour obtenir une famille orthogonale de vecteurs $(\widehat{u}_i)$,
- on normalise chaque vecteur en posant $u'_i = \widehat{u}_i / \sqrt{\langle \widehat{u}_i, \widehat{u}_i \rangle}$.

La famille de vecteurs $(u'_i)_{0\leq i<N}$ ainsi obtenue, si on les met dans une matrice, définit une matrice orthogonale uniforme dans $O(N)$.

Q2.20. Écrire un template de fonction

```
template <class Iterator , class ScalProd = DefaultScalProd>
void normalize(Iterator b, Iterator e, const ScalProd & s= DefaultScalProd());
```

qui normalise tous les vecteurs d'un conteneur de vecteur (par exemple `V` de la section précédente ou équivalent) compris les deux itérateurs de début et de fin `b` et `e`. Il est **interdit** d'utiliser une boucle `for` ou `while` mais vous utiliserez des fonctions des bibliothèques `algorithm` ou `numeric` de la STL.

Q2.21. Écrire un template de fonction

```
template <class RandomNumberGen>
std::vector<V> generate_random_ortho(int N, .... G);
//N: taille du vecteur et dimension de l'espace
```

où `G` est un générateur de nombre pseudo-aléatoire (de la STL). Vous utiliserez la bibliothèque `random` de la STL, la classe `V` ci-dessus, l'une des fonctions de Gram-Schmidt précédente et la fonction `normalize` précédente.

Q2.22. Justifier précisément le type de l'argument `G` dans la fonction `generate_random_ortho`. Vous rappellerez brièvement comment fonctionne un tel générateur de nombre pseudo-aléatoire.

Q2.23. Écrire un programme complet `prog_ortho.cpp` qui écrit dans un fichier `example.txt` une réalisation de ces vecteurs en prenant soin que chaque appel au programme produise une réalisation différente sans avoir besoin d'interagir avec l'utilisateur.