Sorbonne Université **M1 Mathématiques**
2022–2023 **MU4M056 : Programmation en C++**

TP noté 2 : Mesures atomiques finies

L'objectif de ce TP est d'implémenter en C++ la notion de mesure atomique finie sur un espace quelconque avec les constructions associées : masse totale, mesure d'un ensemble intégrale d'une fonction, mesure-image.

Les fichiers fournis doivent conserver les mêmes noms et les fichiers rendues doivent être compilables sans erreur, quitte à commenter les parties de code qui ne marchent pas.

Remarque : dans le texte ci-dessous, le symbole **(T)** *indique un fichier à télécharger sur Moodle avant le début du partiel.*

1 Introduction mathématique et implémentation

1.1 Description mathématique

Soit $(E, \mathcal{P}(E))$ un espace mesurable muni de la tribu complète. Une mesure μ est une mesure atomique finie si et seulement s'il existe un sous-ensemble *fini* X de E et une fonction $\alpha : X \to \mathbb{R}_+$ telle que

$$\mu = \sum_{x \in X} \alpha(x) \delta_x$$

où δ_x est la mesure de Dirac en x. L'ensemble X, appelé l'ensemble des *atomes*, et la fonction α caractérisent complètement la mesure μ. Les nombres $\alpha(x)$ sont appelés les *masses* des atomes.

On a alors les propriétés suivantes :

— pour toute partie A de E, on a

$$\mu(A) = \sum_{x \in X \cap A} \alpha(x) \tag{1}$$

— pour toute fonction $f : E \to F$, la mesure-image ν sur F de μ par f est encore une mesure atomique finie donnée par avec un ensemble d'atomes X' de masses $\alpha' : X' \to \mathbb{R}_+$ donnés par :

$$X' = f(X) \qquad \forall y \in X', \quad \alpha'(y) = \sum_{x \in X ; f(x) = y} \alpha(x) \tag{2}$$

Du point de vue calculatoire, on peut calculer la mesure-image en prenant l'image $y = f(x)$ de chaque atome $x \in X$ et en ajoutant une masse $\alpha(x)$ au point y dans la mesure-image.

— pour toute fonction $f : E \to \mathbb{R}$, on a l'évaluation suivante de l'intégrale :

$$\int f d\mu = \sum_{x \in X} \alpha(x) f(x) \tag{3}$$

1.2 Implémentation par des templates

Nous choisissons d'implémenter une mesure atomique de la façon suivante dans un fichier `fam.hpp` ***(T)*** :

```
template <class E>
class FiniteAtomicMeasure{
   private:
      std::map< E , double > mass;
```

```
    public:
6       //cf fichier fourni
};
```

avec les spécifications suivantes :

- l'objet `mass` de type `std::map< E, double >` décrit l'ensemble des paires $(x, \alpha(x))$ avec x atome variant dans X (cf. ci-dessous pour la description de `std::map`) et $\alpha(x)$ sa masse associée.
- le constructeur par défaut construit la mesure nulle avec $X = \emptyset$;
- la méthode `nb_of_atoms()` renvoie le nombre d'atomes, i.e. le cardinal de X ;
- la méthode `total_mass()` renvoie la somme des masses de tous les atomes, i.e. la mesure de E tout entier ;
- la méthode `add_mass(x,a)` ajoute une masse `a` au point `x` : si `x` était déjà un atome de masse $\alpha(x)$, sa masse devient $\alpha(x) + a$; sinon un atome est ajouté en `x` avec masse `a` .
- les autres méthodes sont documentées plus bas dans les questions correspondantes.

1.3 Documentations sur les objets de type `std::map<E,V>`

Les objets de type `std::map<E,V>` correspondent aux dictionnaires en Python. Un objet de ce type représente un ensemble fini de paires (x_i, v_i) avec x_i de type `E` et v_i de type `V` avec la contrainte que les x_i sont distincts. Cela correspond exactement à une fonction $f : X \to V$ où $X \subset E$ est l'ensemble des x_i et où on a $f(x_i) = v_i$ pour chaque paire.

Pour tout objet `M` de type `std::map<E,V>` , nous avons à disposition :

- La classe `std::map<E,V>` possède un constructeur par défaut qui crée un objet vide.
- une méthode `clear()` qui vide le contenu de l'objet
- un accesseur `size()` au nombre d'éléments présents
- un opérateur `M[x]` avec `x` de type `E` qui permet de
  - de lire la valeur de type `V` associée à `x`
  - de changer la valeur associée à `x` par `M[x]=v` si `x` est déjà dans `M`
  - d'ajouter un objet `x` avec la valeur `v` par la même syntaxe `M[x]=v` si `x` n'est pas déjà dans `M`
- on peut parcourir `M` de deux manières équivalents :
  - soit par

```
for( const std::pair<E, V> & p : M) {
2       //p.first accède à x_i et p.second donne  v_i
}
```

    où `p` prend successivement toutes les valeurs (x_i, v_i)
  - soit par des itérateurs de `M.begin()` à `M.end()` avec `*iterateur` qui est un objet de type `std::pair<E, V>` et correspont à chaque (x_i, v_i)

2 Implémentation pas à pas

2.1 Fonctionnalités élémentaires

1. Dans le fichier `fma.hpp` ***(T)***, compléter les prototypes des méthodes manquantes (type d'arguments complet, `const` éventuels, etc.). Ajouter le constructeur par défaut.

2. Écrire le code de la méthode `nb_of_atoms` .

3. Écrire le code de la méthode `total_mass` .

4. Écrire le code de la méthode `add_mass(x,a)` . *Indication 1 : on pourra utiliser l'opérateur `[]` des `std::map` décrit ci-dessus. Le code tient alors en une ligne.*

5. Écrire le code l'opérateur `<<` de telle sorte que la mesure atomique finie $\sum_{i=1}^{n} \alpha_i \delta_{x_i}$ s'affiche sous la forme suivante :

```
n
x1 a1
x2 a2
...
xn an
```

6. Ajouter les en-têtes nécessaires et vérifier que l'intégralité du code suivant présent dans `test_fam.hpp` ***(T)*** compile et fonctionne.

```
        Finite_atomic_measure<int> mu;
        mu.add_mass( 3, 1.);// un atome de masse 1. en 3
        mu.add_mass( 5, 2.);// un atome de masse 2. en 5
        mu.add_mass( 8, 0.5);// un atome de masse 0.5 en 8
        cout << "*** Masse totale: " << mu.total_mass()<< "\n";//attendu: 3.5
        cout << "*** Nb points: " << mu.nb_of_atoms() << "\n";//attendu: 3
        cout << "*** Mesure mu:\n" << mu << "\n";//attendu: cf. fichier
```

2.2 Mesure d'un ensemble

Nous souhaitons implémenter et tester le template de méthode suivante :

```
template <class E>
template <class Domain>
double Finite_atomic_measure<E>::measure(const Domain & D) const;
```

qui calcule la mesure d'un sous-ensemble ("domaine") `D` de `E` . Nous supposerons que les classes `Domain` acceptables dans le template sont celles qui possède une méthode

```
bool Domain::contains(const E & x) const
```

telle que `D.contains(x)` avec `x` de type `E` renvoie `true` si `x` appartient au sous-ensemble `D` et `false` sinon.

7. Écrire le code du template de méthode `measure` . Pour tester ce code, nous introduisons la classe suivante dans le fichier `geometrie.hpp` ***(T)***

```
template <class K>
class Segment {
        private:
                K left;
                K right;
        public:
                Segment(K l,K r);//l va dans left, r va dans right
                bool contains(K x) const;
                //teste si x est plus grand que left et plus petit que right
};
```

pour décrire les sous-ensembles de $\mathbb{Z}$ d'entiers consécutifs du type $\{l, l+1, \ldots, r-1, r\}$ ou de $\mathbb{R}$ de segments $[l, r]$.

8. Compléter dans `geometrie.hpp` le code du constructeur et de la méthode `contains` .

9. Vérifier que les lignes suivantes de `test_fam.cpp` compilent et donnent le résultat attendu :

```
Segment<int> S1(4,9), S2(-3,6), S3(-3,0);
cout << "Mesure mu([4,9]): " << mu.measure(S1) << "\n";//attendu: 2.5
cout << "Mesure mu([-3,6]): " << mu.measure(S2) << "\n";//attendu: 3.
cout << "Mesure mu([-3,0]): " << mu.measure(S3) << "\n";//attendu: 0.
```

10. Écrire le code du template de méthode d'intégration :

```
template <class E >
template < class RealFunction_on_E >
double Finite_atomic_measure<E>::integral(const RealFunction_on_E & f) const;
```

et vérifier que la ligne suivante de `test_fam.cpp` s'exécute correctement :

```
cout << "Intégrale de sqrt: "
     << mu.integral([](int n) { return sqrt(n); })<< "\n";
```

(valeur attendue : $7,6184$).

2.3 Lecture d'une mesure dans un fichier.

Le but de cette section est de surcharger l'opérateur `>>` .

11. Surcharger l'opérateur `>>` de telle sorte qu'un code du style

```
        Finite_atomic_measure< ... > mu;
        ...
        std::ifstream Input( ... );
        Input >> mu;
```

remplisse `mu` à partir des données écrites dans `Input` selon le même format que dans la question **5.** pour l'opérateur `<<` . *Attention. On prendra garde à bien nettoyer* `mu` *avant de la remplir à nouveau.*

12. Le fichier `"atomic_data.txt"` ***(T)*** contient des données pour une mesure atomique sur $\mathbb{R}$. Écrire un programme complet dans `test_input.cpp` ***(T)*** qui ouvre ce fichier, remplisse un objet d'une classe `Finite_atomic_measure` à partir des données, calcule la masse totale (attendu : 42.2503), ainsi que les mesures des ensembles $[-3., 1.5]$ (attendu : 34.0778), $[0.5, 4.2]$ (attendu : 13.4099). *On pourra réutiliser le template de classe* `Segment` .

2.4 Mesure-image d'une mesure par une fonction

Nous souhaitons à présent implémenter une fonction (pas une méthode !)

```
template < class E1 , class Function  >
Finite_atomic_measure< std::invoke_result_t< Function, E1> >
image(const Finite_atomic_measure<E1> & mu, Function & f);
```

qui calcule la mesure-image d'une mesure `mu` par une fonction `f` selon (2). Le résultat est une mesure sur un ensemble `F` qui n'est pas un paramètre du template car il est déductible à partir du type de retour de `f` : c'est précisément ce qui est fait par l'instruction `std::invoke_result_t` qui donne le type de retour d'un objet de type `Function` appelé sur un argument de type `E1` . C'est du C++17 donc vous devez compiler avec l'option `-std=c++17` .

13. Écrire le code du template de fonction `image` . *Indication : vous pourrez pour cela définir une mesure-image nulle et ajouter progressivement de la masse aux différents points par* `add_mass` .

14. Compléter le code dans `test_fam.cpp` (cf. question 6) pour calculer la mesure-image `mu` par la fonction $f : \mathbb{Z} \to \mathbb{R}$, $x \mapsto f(x) = (x-4)^2 + \pi$ et afficher le résultat.

15. Compléter le code dans `test_input.cpp` (cf. question 11) pour obtenir la mesure-image sur $\mathbb{Z}$ de la mesure lue dans le fichier par la fonction partie entière (rappel : `std::floor` dans `<cmath>` mais elle renvoie un réel, pas un entier). Afficher alors la mesure des ensembles $\{0, 1, 2\}$ et $\{-10, 10\}$ (valeurs attendues : 16.4031 et 42.2503) et la mesure-image toute entière.

2.5 Fonctionnalités additionnelles (bonus)

16. Écrire un template de constructeur qui prend en argument deux itérateur, l'un de début, l'autre de fin sur un conteneur arbitraire (dont on supposera que les objets sont de type `E`) qui construise une mesure atomique finie dont les atomes sont définis par les objets du conteneur et leurs masses sont toutes égales à 1. Le tester dans `test_fam.cpp` .

17. Construire un accesseur `atomic_masses` au champ privé `mass` qui *ne copie pas* cet objet.

18. Définir un opérateur `+` qui fasse la somme de deux mesures atomiques finies sur le même espace.