Sorbonne Université **M1 Mathématiques**
2020–2021 **MU4MA056 : Programmation en C++**

Examen du lundi 3 mai : Autour de la méthode du rejet.

— Durée : 2h. Le sujet comporte 8 pages.
— **Aucun document n'est autorisé.**
— **Tout système muni d'un processeur et de mémoire (téléphone, ordinateur, montre connectée,...) est strictement interdit.**

Informations importantes :
— les questions ne sont pas classées par ordre de difficulté et couvrent l'ensemble des thématiques du cours ;
— nulle connaissance mathématique n'est requise : tous les algorithmes sont fournis en pseudo-code.
— le sujet est long car plusieurs codes sont fournis et documentés. Ne soyez pas impressionnés !
— la section 4 est *en bonus*. Ne l'abordez pas si vous n'avez pas fini le reste.

1 Introduction

La méthode du rejet permet de simuler des réalisations d'une variable aléatoire Y, *a priori* difficile à simuler directement, à partir d'une autre variable aléatoire X, *a priori* facile à simuler directement, sur le même espace d'états E.

Nous supposerons que Y et X ont des densités, notées f_Y et f_X par rapport à une mesure commune. Nous supposerons de plus qu'il existe une constante $C \geq 1$ telle que, pour tout $z \in E$, $f_Y(z) \leq C f_X(z)$. L'algorithme, en pseudo-code, est le suivant :

```
------- Algo Rejet --------
    rejet:= vrai
    compteur:=0;
    tant que (rejet)
        x = réalisation de la v.a. X
        u = réalisation d'une uniforme sur [0,1]
        compteur = compteur +1
        si(  u*C*f_X(x) < f_Y(x) )                  [*]
             rejet=faux
        fin si
    fin tant que
    renvoyer x
---------------
```

Le but de ce problème est de créer un modèle de classe qui puisse implémenter cet algorithme de manière aussi générale que possible. Dans la section 2, nous formulerons une première approche. Dans la section 3, nous implémenterons une classe particulière de densités constantes par morceaux et testerons la méthode du rejet. Dans la section 4, nous verrons comment il est possible d'optimiser la méthode du rejet.

Remarque. Nous souhaitons que le modèle de classe que nous allons construire possède les mêmes méthodes que les distributions de la classe `<random>` de C++11. Pour cela, nous rappelons les faits suivants :
— il existe un type `std::mt19937_64` de générateurs de nombres aléatoires de bonne qualité dont le constructeur prend une graine en argument,
— chaque classe `std::DISTRIBUTION` possède une méthode `operator()` , qui prend un générateur de nombres aléatoires en argument et renvoie une réalisation de la variable, et également un constructeur qui prend les paramètres de la distribution en argument.
— il existe une classe `std::uniform_real_distribution<double>` qui permet de générer par défaut des nombres uniformes sur $[0,1]$,
— il existe une classe `std::exponential_distribution<double>` pour des nombres uniformes de loi exponentielle $ae^{-ax}dx$ dont le paramètre a est passé au constructeur.
— la bibliothèque `<cmath>` contient le nombre π dans la variable `M_PI` .

2 Une première implémentation très générale

Nous proposons le modèle de classe suivant écrit dans un fichier `rejection.hpp` :

```
   template <class RandomVar, class DensityY, class DensityX, class E>
2  class Rejection_distribution {
   private:
4     RandomVar X;
      DensityY fY;
6     DensityX fX;
      double C;
8     std::uniform_real_distribution<double> U;
   public:
10    Rejection_distribution(
          const RandomVar & X0, const DensityY & fY0, const DensityX & fX0,double C0);
12    template <class RNG>  E operator()(RNG & G);
      double density_wanted(const E & x) const;
14    double density_simulated(const E & x) const;
   };/* requires:
16  *        1/  X(G), where G is a generator of type RNG, have values of type E
    *        2/  fY(z) and fX(z), where z has type E, have values in R_+
18  *        3/  fY(z) < C fX(z) for all z
    *  4/ U is always uniform on [0,1]    */
```

Q1. Quel est le principe général des générateurs de nombres pseudo-aléatoires ?

Q2. Écrire le code du constructeur, comme s'il était écrit directement dans la classe.

Q3. Écrire les codes des méthodes `density_wanted` et `density_simulated` qui évaluent les densités `fY` et `fX` au point $x \in E$ comme si elles étaient écrites directement dans la classe.

Q4. Que signifient les `const` à la fin des lignes 13 et 14 ?

Q5. Quels codes faut-il écrire préférentiellement à l'intérieur de la classe ? Où faut-il écrire les autres codes de templates ?

Q6. Écrire un accesseur et un mutateur (de même nom `constante()`) associés à `C` , comme s'ils étaient écrits directement dans la classe.

Q7. Écrire le code du template de méthode `operator()` en dehors de la classe, qui implémente le pseudo-code fourni en introduction.

Q8. Expliquer en détail pourquoi l'argument `G` de `operator()` n'est pas étiqueté `const` et pourquoi il est passé par référence.

Q9. Que faudrait-il écrire en tête et en fin du fichier `rejection.hpp` pour éviter d'éventuels problèmes d'inclusions multiples ?

Q10. Pourquoi faut-il écrire le code du modèle de classe dans un fichier `.hpp` et non dans un fichier `.cpp` ?

Q11. Nous souhaitons simuler une variable Y de densité $\sqrt{2/\pi}\exp(-x^2)dx$ sur $\mathbb{R}_+$ (i.e. la valeur absolue d'une gaussienne) à partir d'une variable de loi exponentielle de paramètre 1 par la méthode du rejet. Une rapide étude de fonction donne l'identité

$$\forall x \in \mathbb{R}_+, \qquad \sqrt{\frac{2}{\pi}}e^{-x^2/2} \leq Ke^{-x} \qquad \text{avec} \qquad K = \left(\sqrt{\frac{2e}{\pi}}\right) \tag{1}$$

1. Écrire des fonctions (ou des lambda-fonctions) `density_exp` et `density_y` qui codent les densités de ces deux lois.
2. En supposant que le code de la classe est écrit dans un fichier `rejection.hpp` , écrire un programme **complet** de nom `prog.cpp` qui permet d'estimer la moyenne de Y en calculant la moyenne empirique d'un grand nombre (disons 10^6) de réalisations de Y par la méthode précédente. *On veillera à bien instancier complètement les templates utilisés. Si vous avez besoin de récupérer le type d'une v.a.* `a` *que vous ne connaissez pas (par exemple une lambda-fonction), la syntaxe* `decltype(a)` *introduite en C++11 vous fournit ce type.*
3. Quelle suite de commandes faut-il écrire pour obtenir un programme exécutable et l'exécuter ?
4. Quelles options d'optimisation peut-on fournir au compilateur et à quoi servent-elles ?

Q12. L'un des problèmes de la méthode du rejet est précisément le rejet d'un grand nombre de réalisations de la variable aléatoire X lorsque la constante C et la loi de X sont mal choisies. Pour cela, il est utile de mesurer numériquement le nombre de rejets avant acceptation d'une valeur. Pour cela, nous proposons d'ajouter un champ privé `unsigned int steps` initialisé à zéro au début du code de l'opérateur `operator()` et incrémenté à chaque pas de la boucle `tant que` du pseudo-code. Modifier le code de `operator()` et ajouter un accesseur `get_steps()` à cette nouvelle variable.

3 Variables aléatoires à densités constantes par morceaux sur les réels

Remarque culturelle : cette section consiste en une réécriture adaptée à l'examen de la classe `std::piecewise_constant_distribution<double>` *de la bibliothèque* `<random>` *.*

3.1 Plusieurs classes utiles

Nous allons étudier dans cette section le cas simple $E = \mathbb{R}$ avec des densités pour les variables Y de type :

$$f_Y(x) = \sum_{k=0}^{n-1} h_k \mathbf{1}_{[a_k,b_k[} \tag{2}$$

avec les hypothèses suivantes :
- les intervalles $[a_k, b_k[$ sont disjoints et supposés ordonnés, i.e. $a_0 < b_0 < a_1 < b_1 < \ldots < a_{n-1} < b_{n-1}$
- les réels h_k sont strictement positifs
- les densités sont normalisées de telle sorte que

$$\int_{\mathbb{R}} f_Y(x)dx = \sum_{k=0}^{n-1} h_k(b_k - a_k) = 1 \tag{3}$$

Nous appellerons *boîte* un triplet (a_k, b_k, h_k) et *aire de la boîte* la quantité $A_k = h_k(b_k - a_k)$. On vérifie en particulier que $\sum_{k=0}^{n-1} A_k = 1$. Nous introduisons également les aires cumulées $CA_k = \sum_{l=0}^{k} A_l$ pour $0 \leq k \leq n-1$ (on a en particulier $CA_{n-1} = 1$).

La simulation d'une v.a. Y de densité f_Y donnée par (2) est alors facile et se fait via l'algorithme suivant :

```
--------- Algo simu piecewise constant densities ---
k := entier aléatoire entre 0 et n-1 choisi avec probabilité A_k
u := réel aléatoire uniforme sur [0,1]
renvoyer a_k +u * (b_k-a_k)
----------------------------------------------
```

Nous définissons alors les classes suivantes :

```
struct Box {
    public:
        double left; // a_k
        double right; // b_k
        double height; // h_k
};

class PiecewiseConstant_function {
   private:
      std::vector< Box > boxes;// sorted by ascending b_k's
   public:
      template <class Iter> PiecewiseConstant_function(
        Iter leftbegin,Iter leftend, Iter rightbegin, Iter heightbegin);//  (A)
      double operator() (double x) const;
      int nb_of_boxes() const;
      bool check_ranges() const;
      bool check_disjoint() const;
};

class PiecewiseConstant_distribution {
    private:
        PiecewiseConstant_function f;
        std::uniform_real_distribution<double> U;
        std::vector<double> cumul_area; // aires cumulées CA_k
    public:
        template <class Iter> PiecewiseConstant_distribution(
         Iter leftbegin,Iter leftend,Iter rightbegin, Iter heightbegin);//  (A')
        int random_box(double u) const;
        template <class RNG> double operator() (RNG & G);
};
```

Remarque. **Pour simplifier l'écriture de vos codes sur papier**, vous pourrez écrire

`PCF` à la place de `PiecewiseConstant_function` et
`PCD` à la place de `PiecewiseConstant_distribution`
`EWPCF` à la place de `EqualWidth_PiecewiseConstant_function` .

3.1.1 La classe `PiecewiseConstant_function`

Q13. Écrire dans la structure `Box` le constructeur qui prend comme argument (a_k, b_k, h_k) et construit les champs publics comme indiqués (a_k est copié dans `left` , etc...).

Q14. Écrire un constructeur par défaut de `PiecewiseConstant_function` qui correspond à la fonction nulle.

Q15. Écrire le code du template de constructeur (A) dans lequel `leftbegin` et `leftend` sont des itérateurs de début et fin sur un conteneur qui contient les éléments (a_k), `rightbegin` (resp. `heightbegin`) est un itérateur de début sur un conteneur qui contient les (b_k) (resp. (h_k)). On supposera que les trois conteneurs ont bien des tailles compatibles.

Ce constructeur remplit le conteneur `boxes` comme attendu et **trie ce tableau par ordre croissant des** b_k.

Remarque importante : **nous supposerons dans toute la suite que le vecteur `boxes` est toujours classé par ordre croissant des b_k et on pourra se servir de cette information dans les algorithmes. Si une méthode ou une fonction doit modifier le vecteur `boxes` , on veillera à ce que cet ordre ne soit pas rompu.**

Q16. Écrire le code des surcharges des opérateurs `<<` et `>>` pour `Box` de telle sorte qu'une suite de caractères telle que

```
a b h
```

soit écrite pour ou lue comme une boîte de paramètre (a, b, h).

Q17. Écrire le code des surcharges des opérateurs `<<` et `>>` pour la classe `PiecewiseConstant_function`
afin qu'un fichier formaté de la manière suivante :

```
n
a0 b0 h0
a1 b1 h1
...
a_{n-1} b_{n-1} h_{n-1}
```

corresponde à l'objet de la classe de manière idoine.

Que faut-il faire de plus dans la classe pour que cela soit possible ?

Q18. Que faut-il faire pour que toutes les méthodes de `PiecewiseConstant_distribution` aient accès au champ privé `boxes` de `f` ?

Q19. Écrire un court programme *complet*, de nom `checkio.cpp` (avec en-têtes, etc) qui lit avec `>>` une fonction constante par morceaux dans un fichier `source.dat` et la réécrit avec `<<` à l'identique dans un fichier `target.dat`

Q20. Écrire le code de la méthode `check_ranges()` qui vérifie bien si toutes les variables `height` sont strictement positives et si chaque champ `right` est bien plus grand que chaque `left` . **Vous n'utiliserez que des algorithmes de la bibliothèque `algorithm` .**

Q21. Écrire le code de la méthode `operator()` qui renvoie $f(x)$ selon la formule (2). Vous utiliserez pour cela la fonction `std::lower_bound` de <algorithm> en cherchant la première boîte de `boxes` telle que le champ `right` dépasse `x` .

*L'algorithme `std::lower_bound` fonctionne de la manière suivante : si `C` est un conteneur **ordonné** (i.e. les éléments rencontrés lors du parcours du début vers la fin sont **croissants**) d'objets de type `T1` et si `x` est un objet de type `T2` et `comparator` est une fonction `bool comparator(const T1 & a, const T2 & b)` qui indique si `a` est considéré plus petit que `b` , alors*

```
auto iterator = std::lower_bound(C.begin(), C.end(), x, comparator)
```

renvoie un itérateur égal à `C.end()` si tous les éléments de `C` sont considérés plus petits que `x` , ou bien un itérateur vers le premier élément de `C` qui n'est pas plus petit que `x` .

3.1.2 La classe `PiecewiseConstant_distribution`

Q22. Écrire le code du template de constructeur (A'), qui initialise `f` avec le constructeur précédent, initialise `U` pour avoir des uniformes sur $[0, 1]$ et remplit `cumul_area` de telle sorte que `cumul_area[k]` contienne la somme partielle $\sum_{l=0}^{k} A_l$.

Q23. Écrire la méthode `random_box(u)` qui, à partir d'un réel $u \in (0, 1)$ (qui sera vu ensuite comme une réalisation d'une v.a.), renvoie l'unique indice k tel que : $\sum_{l=0}^{k-1} A_l < u \leq \sum_{l=0}^{k} A_l$. On utilisera pour cela le tableau `cumul_area` et à nouveau l'algorithme `std::lower_bound` sus-mentionné. Si vous n'y arrivez pas, vous pourrez aussi opter pour la solution de votre choix !

Q24. [bonus] Avez-vous une idée de la complexité de `random_box` en fonction du nombre n de boîtes ? Autrement dit, avez-vous une idée de comment pourrait fonctionner l'algorithme `std::lower_bound` sur un `std::vector` ?

Q25. Écrire à présent le template de `operator()` qui renvoie une réalisation d'une v.a. Y de densité f_Y. Cette méthode utilisera en particulier `random_box` pour le choix de la boîte.

Q26. Toutes les lois de la partie `<random>` de la STL sont définies de telle sorte que, pour une loi `L` [1] :
- `L::value_type` donne le type de valeurs de la v.a. de loi `L` (par exemple `int` pour `std::poisson_distribution<in` et `float` pour `std::normal_distribution<float>`) ;
- `L::param_type` donne le type des paramètres de la loi, ici la fonction `f` ;
- il existe un accesseur et un mutateur `param()` à l'ensemble des paramètres de la loi, ici le champ privé `f` ;
- il existe deux méthodes `min() const` et `max() const` qui renvoient les valeurs minimale et maximale des réalisations de la v.a. (pour une uniforme sur $(0,1)$, par exemple, ce sera 0 et 1).

Que faut-il écrire dans la classe précédente pour que `PCD::value_type` et `PCD::param_type` soient définis respectivement à `double` et à `PCF` ? (indice : syntaxe avec `using`)

Compléter également les méthodes manquantes.

3.2 Test comparatif de performance

Nous souhaitons comparer les performances de différentes méthodes du rejet afin de simuler une v.a. X de densité $(p+1)x^p \mathbf{1}_{0\le x\le 1}$ (que l'on peut d'ailleurs simuler par inversion de la fonction de répartition).

Q27. Écrire un modèle de fonction :

```
template <class RealDistri, class RandomNumberGen>
std::tuple<double,double,double> eval_perf(
        RealDistri & X, RandomNumberGen & G, uint nb_samples);
```

qui renvoie un 3-uplet (t, m, σ) tel que :
- `RealDistri` est la loi d'une v.a. réelle (à valeurs de type `double`) ;
- la fonction réalise `nb_samples` réalisations indépendantes de la v.a. `X` stockées dans un vecteur de `nb_samples` objets de type `E` ;
- t est le temps total pour cette génération ;
- m (resp. σ) est la moyenne (resp. écart-type) empirique de cette v.a. sur ces réalisations.

Q28. **Test comparatif de performances.** Pour tout entier $p \ge 2$, on s'intéresse à la v.a. Y de densité que l'on souhaite simuler par la méthode du rejet à partir de différentes v.a. X_i :
- X_1 uniforme sur $[0,1]$ avec $C_1 = p+1$
- X_2 de densité
$$f_2(x) = \frac{1}{a^{p+1}+1-a}\left(a^p \mathbf{1}_{0\le x\le a} + \mathbf{1}_{a<x\le 1}\right)$$
avec $C_2 = (p+1)(a^{p+1}+(1-a))$ et $a = (1/(p+1))^{1/p}$ (choix optimal pour une fonction à deux boîtes)
- X_3 de densité (on choisira $M = 128$)
$$f_3(x) = \frac{1}{Z_M}\sum_{k=0}^{M-1}\left(\frac{k+1}{M}\right)^p \mathbf{1}_{k/M\le x<(k+1)/M}, \qquad Z_M = \frac{1}{M}\sum_{k=1}^{M}(k/M)^p$$
avec $C_3 = (p+1)Z_M$ (on remarque $C_3 \to 1$ lorsque $M \to \infty$!).

Écrire un programme complet (en-tête compris) qui :

1. Vous pourrez consulter la liste complète des prérequis à l'adresse https://en.cppreference.com/w/cpp/named_req/RandomNumberDistribution.

— demande à l'utilisateur un entier p ;
— pour chaque variable X_i, $1 \leq i \leq 3$, mesure les performances avec la fonction `eval_perf` avec 10000 échantillons à chaque fois ;
— affiche pour chaque variable les trois informations mesurées.

Ce programme est assez long. Nous vous conseillons de le faire à la fin. Toute avancée dans la bonne direction rapportera des points.

4 Optimisation supplémentaire [BONUS]

4.1 Optimisation des fonctions constantes par morceaux

La méthode du rejet fait énormément d'appels à l'évaluation des densités : il faut donc optimiser autant que possible cette étape.

Q29. Dans le cas où toutes les boîtes ont la même largeur δ, l'algorithme de `PCF::operator()(double) const` peut être optimisé. Pour cela, nous décidons de définir la classe fille (EWPCF en abrégé)

```
class EqualWidth_PiecewiseConstant_function : ???1 {
   private:
             double delta;
   public:
      template<class Iter> EqualArea_PiecewiseConstant_function(
          double Min, double Max, int n, Iter heightbegin);// (C)
               ???2
};
```

1. Que faut-il écrire à la place de `???1` pour que cette classe hérite publiquement de `PiecewiseConstant_function` ?
2. Que faut-il changer dans la classe mère pour que les méthodes de la classe fille puissent accéder aux champs privés `boxes` , etc. ?
3. Que faut-il écrire à la place de `???2` pour définir une méthode `random_box(u)` qui renvoie, avec une complexité $O(1)$, l'entier k tel que $k \leq u < k+1$?
4. Faut-il faire quelque chose (et si oui, quoi ?) dans les classes mère et/ou fille pour que, lors de tout appel à `operator()` , ce soit toujours l'appel à la méthode `random_box` de la classe fille qui soit fait.
5. Écrire le constructeur `(C)` de la classe fille de telle sorte que `n` soit le nombre de boîte, a_0 est donné par `Min` et b_{n-1} donné par max, avec $a_k = b_{k+1}$. L'itérateur `heightbegin` désigne le début du conteneur où sont stockées les hauteurs.

4.2 Optimisation du rejet

Une implémentation directe de la ligne `[*]` du pseudo-code qui correspond à l'acceptation ou non de la valeur `x` n'est pas toujours optimale. En effet, dans l'inégalité (1), qui n'est autre que `[*]` sans le `u` , l'évaluation des deux termes de l'inégalité nécessite le calcul de deux exponentielles. Ce calcul est assez coûteux et on souhaiterait faire des économies de temps de calcul. De même, si on s'y prend mal, les calculs de racines carrées répétés peuvent nuire au temps de calcul. Pour éviter cela, on constate alors que le test

$$\sqrt{\frac{2}{\pi}} e^{-x^2/2} \leq uKe^{-x}$$

peut se réécrire avec profit sous la forme

$$e^{-(x-1)^2/2} \leq u \tag{4}$$

où il n'y a plus qu'une seule exponentielle à calculer et où les constantes ont disparu.

Nous souhaitons également avoir un code unique qui reprenne le code de la section 2 et accepte aussi ce type de test simplifié. Pour cela, nous remplaçons le template de départ par :

```
struct NoTest {};

template <class RV, class DensityY, class DensityX,
          class E=RV::value_type, class OptimizedTest = NoTest>
class Rejection_distribution {
private:
   RV X;
   DensityY fY;
   DensityX fX;
   double C;
   std::uniform_real_distribution<double> U;
   OptimizedTest acception_test;
public:
   Rejection_distribution(
       const RV & X0, const Dens1 & fY0, const Dens2 & fX0,
       double C0, OptimizedTest acception0 = {} );
   ...
};
//requires: bool acception_test(const E & x,double u) to replace test [*]
```

Les arguments par défaut dans le template et le constructeur permettent de gérer le cas où l'on n'a pas à fournir de test simplifié.

Q30. Réécrire le code de `operator()` qui permette d'utiliser soit le test classique si `NoTest` est le type utilisé, soit le test optimisé `acception_test` sinon. Pour cela, on utilisera avec profit :

- le template `std::is_same<T,U>::value` dans la bibliothèque `<type_traits>` qui renvoie *dès la compilation* `true` ou `false` selon que les types `T` et `U` sont identiques ou non.
- la nouveauté de C++17 `if constexpr` au lieu de `if` pour le test de type qui permet de faire le choix d'algorithmes dès la compilation. Il fonctionne de la manière suivante : lorsque `test` est connu dès la compilation, le code

```
if constexpr (test) {
        [bloc1]
} else {
        [bloc2]
}
```

  produit un code exécutable contenant *seulement* `bloc1` ou `bloc2` selon la valeur de `test` et permet ainsi que le code `test` ne soit plus effectué à chaque exécution.

Q31. Reprendre la question ***Q11*** (2) en utilisant un test d'acceptation qui correspond à (4).

Q32. Que proposeriez-vous comme test simplifié pour les densités $f_Y(x) = (2/\pi)\sqrt{1-x^2}\mathbf{1}_{-1\leq x\leq 1}$ et $f_X(x) = (1/2)\mathbf{1}_{-1\leq x\leq 1}$ avec $C = 4/\pi$ afin d'économiser au mieux le temps de calcul ?