Université Pierre et Marie Curie — **Master M1 "Mathématiques et Applications"**
Période 2 - 2011/2012 — **Programmation en C/C++**

TD 7 : Objets fonctionnels et modèles de fonctions (templates)

Dans ce TD, on utilise la notion d'*objet fonctionnel* et de modèle de fonction très utilisés par la bibliothèque standard du C++. Les objets fonctionnels ont pour but de remplacer les fonctions. Par convention, le mot-clé `struct` sera utilisé pour déclarer des *objets fonctionnels* (déclaration publique par défaut). Par exemple, on définira la classe fonctionnelle `Uniforme` de cette façon :

```
struct Uniforme {
    double operator()() const { return rand()/(double) RAND_MAX; }
};
```

Un objet `Uniforme` s'utilisera comme une fonction et renverra un nombre aléatoire sur $[0,1]$.

1. *Écrire un objet fonctionnel* `ExpoAffine` *qui permet de coder des fonctions paramétriques de la forme* $f(x) = e^{ax+b}$ *où* a *et* b *sont deux réels. Les paramètres* a *et* b *doivent être des champs privés initialisés par le constructeur de la classe* `ExpoAffine`*.*

On considère le problème numérique suivant : l'estimation par une méthode de Monte Carlo d'intégrales de la forme suivante

$$I_f = \int f(x)g(x)dx = \mathbb{E}[f(X)]$$

où X est une variable aléatoire de densité g (par rapport à la mesure de Lebesgue) et f une fonction à valeurs réelles telle que $\mathbb{E}[f^2(X)] < +\infty$. La méthode de Monte-Carlo repose sur la loi des grands nombres et le théorème de la limite centrale. Si on considère une suite $(X_n)_{n\geq 1}$ i.i.d. de même loi que X, alors

$$\bar{I}_n := \frac{1}{n}\sum_{k=1}^{n} f(X_k) \xrightarrow{p.s.} I_f \quad \text{et} \quad \sqrt{n}\left(\frac{1}{n}\sum_{k=1}^{n} f(X_k) - I_f\right) \xrightarrow{\mathcal{L}} \mathcal{N}(0, \sigma_f^2),$$

où $\sigma_f^2 = \mathrm{var}(f(X)) = \lim_n \bar{\sigma}_n^2$ avec $\bar{\sigma}_n^2 = \dfrac{1}{n-1}\sum_{k=1}^{n}(f(X_k) - \bar{I}_n)^2$.

2. *Implémenter cette méthode (pour* $g(x) = \mathbf{1}_{0<x<1}$*, loi d'une variable uniforme) dans une fonction* `monte-carlo1` *de prototype :*

```
void monte_carlo1(ExpoAffine const & f, Uniforme const & U, unsigned n,
                  double & mean, double & var);
```

Tester cette fonction en calculant I_f *avec* $f(x) = e^{\frac{x}{2}+1}$ *et donner l'intervalle de confiance asymptotique à* 95% *défini par*

$$\left]\bar{I}_n - 1,96\frac{\bar{\sigma}_n}{\sqrt{n}}; \bar{I}_n + 1,96\frac{\bar{\sigma}_n}{\sqrt{n}}\right[$$

On rappelle maintenant que pour générer un couple de 2 gaussiennes indépendantes à partir d'un tirage de 2 uniformes, on peut utiliser l'algorithme de Box-Müller : soit U_0 et U_1 deux variables aléatoires indépendantes uniformément distribuées sur $]0,1]$ et Z_0 et Z_1 telles que

$$\begin{cases} Z_0 = \sqrt{-2\ln(U_0)}\cos(2\pi U_1), \\ Z_1 = \sqrt{-2\ln(U_0)}\sin(2\pi U_1). \end{cases}$$

Alors Z_0 et Z_1 sont des variables aléatoires indépendantes suivant la loi normale centrée réduite.

3. *Écrire une classe fonctionnelle* `Gaussienne` *qui renvoie une réalisation d'une variable aléatoire gaussienne de moyenne* $m \in \mathbb{R}$ *et de variance* $s > 0$*. Le constructeur doit se charger d'initialiser les champs codants* m *et* s *avec des valeurs par défauts de* `0` *et* `1`*. L'opérateur fonctionnel doit renvoyer une réalisation.*

4. *En utilisant la classe* `Polynome` *de la feuille précédente, écrire une fonction* `monte_carlo2` *(avec* $g(x)$ *la densité de la loi normale qui justifie l'usage de variables gaussiennes) de prototype :*

```
void monte_carlo2(Polynome const & P, Gaussienne const & G, unsigned n,
                  double & mean, double & var);
```

Tester cette fonction en calculant I_P *avec* $P(x) = 4x^3 + \frac{1}{2}x^2 - 1$ *et* $G \sim \mathcal{N}(0,1)$*. Donner l'intervalle de confiance asymptotique à* 95%*.*

Noter la grande similitude entre les fonctions `monte_carlo1` et `monte_carlo2`. Il est possible de définir une fonction générique ou un modèle de fonction qui sera traduit (à la compilation) par exemple en la fonction `monte_carlo1` ou en `monte_carlo2` (en fonction des appels).

5. *En utilisant les templates, écrire une fonction* `monte_carlo` *générique qui permet d'estimer* $\mathbb{E}(f(X))$ *pour toute variable aléatoire (réelle, vectorielle, complexe...) et toute fonction* f *(prenant ses valeurs dans* $\mathbb{R}$, $\mathbb{R}^d$, $\mathbb{C}$,*...).*

6. *Tester cette fonction avec différents appels.*

Dimension supérieure

La méthode de Monte-Carlo est surtout intéressante en dimension grande. On va donc créer une petite classe `Array` qui permet de manipuler simplement des tableaux (en pratique on utilisera le conteneur `vector` de la `STL` que l'on verra un peu plus tard). Voici un extrait du fichier `array.hpp`

```
#ifndef ARRAY_H
//------
//------
class Array
{
    public:
        Array(unsigned n = 0) //------
        Array(const Array &a) //------
        ~Array() { //------ }
        Array& operator=(const Array &);
        double operator[](unsigned) const;
        double& operator[](unsigned);
        unsigned size() const { return n; }
        friend std::ostream& operator<<(std::ostream &, const Array &);
        friend std::istream& operator>>(std::istream &, Array &);
    private:
        unsigned n;
        double  *data;
};
Array& Array::operator=(const Array &a) {
//------
...
```

7. *Compléter ce fichier* `array.hpp` *en définissant les contructeurs, le destructeur et les opérateurs associés.*

8. *Créer un objet fonctionnel de classe* `UniformeDim2` *qui renvoie un objet* `Array` *de taille* 2 *dont chaque composante est de loi* $\mathcal{U}([0;1])$.

9. *On considère la fonction suivante* $g(u,v) = 4 \times \mathbf{1}_{\{u^2+v^2>1\}}$ *définie pour* $(u,v) \in [0;1]^2$. *Écrire un objet fonctionnel associé à cette fonction. Et tester la fonction* `monte_carlo` *générique pour estimer* $\mathbb{E}(g(U,V))$ *où* $U, V \sim \mathcal{U}([0;1])$.