

Réduction de variance

Exemples en C++

Vincent Lemaire
vincent.lemaire@upmc.fr

Functors - Objets fonctionnels

- Définition

- Composition

- ptr_fun

Monte-Carlo générique et réduction de variance

- Fonction Monte-Carlo

- Variables antithétiques

- Variable de contrôle

- Stratification

- Echantillonnage d'importance

Illustration : option Basket

- Utilisation de tvmet

- Option basket

std::function et Lambda functions en C++11

- std::function

- Un mot sur les variadic templates

- Lambda function

- Retour à Monte Carlo

Functors - Objets fonctionnels

- Définition

- Composition

- ptr_fun

Monte-Carlo générique et réduction de variance

- Fonction Monte-Carlo

- Variables antithétiques

- Variable de contrôle

- Stratification

- Echantillonnage d'importance

Illustration : option Basket

- Utilisation de tvmet

- Option basket

std::function et Lambda functions en C++11

- std::function

- Un mot sur les variadic templates

- Lambda function

- Retour à Monte Carlo

Functors - Objets fonctionnels (1)

Functor = objet fonctionnel, i.e. ayant un opérateur () défini.

Pour faciliter l'écriture et la manipulation de ces functors, 2 classes abstraites sont définies dans la STL :

► Unary Fonction

```
template <class _Arg, class _Result>
2 struct unary_function
{
4     typedef _Arg argument_type;
    typedef _Result result_type;
6 };
```

► Binary Fonction

```
template <class _Arg1, class _Arg2, class _Result>
2 struct binary_function
{
4     typedef _Arg1 first_argument_type;
    typedef _Arg2 second_argument_type;
6     typedef _Result result_type;
};
```

Functors - Objets fonctionnels (2)

De la même façon on définit la classe abstraite generator :

```
1 template <class _Result>
2 struct generator
3 {
4     typedef _Result result_type;
5 };

```

Dans les séances précédentes on aurait pu définir les classes abstraites `var_alea<T>` et `processus<T>` comme des classes dérivées de `generator<T>`.

Exemple : Pour trier un vecteur de **double** en valeur absolu :

```
1 struct less_mag : public binary_function<double, double, bool> {
2     bool operator()(double x, double y) { return fabs(x) < fabs(y); }
3 };
4
5 vector<double> V;
6 ...
7 sort(V.begin(), V.end(), less_mag());

```

Composition de Functors (1)

Il est possible de composer 2 functors pour en créer un nouveau, par exemple pour créer un nouveau generator à partir d'un unary_function et d'un generator.

$$X : \Omega \rightarrow E, \quad f : E \rightarrow F \quad \Rightarrow \quad f \circ X : \Omega \rightarrow F$$

Composition de Functors (1)

Il est possible de composer 2 functors pour en créer un nouveau, par exemple pour créer un nouveau generator à partir d'un unary_function et d'un generator.

$$X : \Omega \rightarrow E, \quad f : E \rightarrow F \quad \Rightarrow \quad f \circ X : \Omega \rightarrow F$$

```
2  template <typename Fct, typename Gen>
   struct compose_t : public generator< typename Fct::result_type >
   {
4     compose_t(Fct f, Gen X) : f(f), X(X) {};
       typename Fct::result_type operator()() {
6         return f(X());
       };
8     private:
       Fct f; Gen X;
10 };
```

Composition de Functors (1)

Il est possible de composer 2 functors pour en créer un nouveau, par exemple pour créer un nouveau generator à partir d'un unary_function et d'un generator.

$$X : \Omega \rightarrow E, \quad f : E \rightarrow F \quad \Rightarrow \quad f \circ X : \Omega \rightarrow F$$

```
2  template <typename Fct, typename Gen>
   struct compose_t : public generator< typename Fct::result_type >
   {
4     compose_t(Fct f, Gen X) : f(f), X(X) {};
       typename Fct::result_type operator()() {
6         return f(X());
       };
8     private:
       Fct f; Gen X;
10 };
```

Utilisation pas très commode car comme c'est une classe générique il faut préciser les types de f et de X qui peuvent être long et compliqués !

Composition de Functors (2)

Du coup, on utilise une fonction générique qui se charge de renvoyer un nouvel objet `compose_t` avec les bons types :

```
2 | template <typename Fct, typename Gen>  
  | inline compose_t<Fct, Gen> compose(Fct f, Gen X) {  
  |     return compose_t<Fct, Gen>(f, X);  
4 | };
```

- ▶ mot-clé **inline** : à la compilation le code de la fonction est directement inséré, il n'y a pas de mécanisme d'appel de la fonction : plus rapide (surtout lorsque le code est petit)
- ▶ rappel : les fonctions membres et opérateurs d'une classe définis au moment de la déclaration sont automatiquement **inline**

Exemple :

```
2 | Gaussian G(0,1);  
  | Black_Scholes BS(r, sigma, T);  
  | compose(BS, G());
```

Composition de Functors (2)

Du coup, on utilise une fonction générique qui se charge de renvoyer un nouvel objet `compose_t` avec les bons types :

```
2 | template <typename Fct, typename Gen>  
   | inline compose_t<Fct, Gen> compose(Fct f, Gen X) {  
   |     return compose_t<Fct, Gen>(f, X);  
4 | };
```

- ▶ mot-clé **inline** : à la compilation le code de la fonction est directement inséré, il n'y a pas de mécanisme d'appel de la fonction : plus rapide (surtout lorsque le code est petit)
- ▶ rappel : les fonctions membres et opérateurs d'une classe définis au moment de la déclaration sont automatiquement **inline**

Exemple :

```
2 | Gaussian G(0,1);  
   | Black_Scholes BS(r, sigma, T);  
   | compose(BS, G)();
```

Intérêt ?? à venir...

ptr_fun

La fonction `ptr_fun` permet d'encapsuler une fonction (classique) dans un functor.

Exemple :

```
double f(double x) {  
2     return 1+x+0.5*x*x;  
}  
4 ...  
vector<double> x(10);  
6 Gaussian G(0,1);  
generate(x.begin(), x.end(), compose(ptr_fun(f), G));
```

Remarque : Si une fonction est surchargée (comme celles de la librairie `math` : `pow`, `cos`, `exp`,...) il faut préciser les types :

```
ptr_fun<double, double>(exp);  
2 ptr_fun<double, int, double>(pow);
```

ptr_fun

La fonction `ptr_fun` permet d'encapsuler une fonction (classique) dans un functor.

Exemple :

```
double f(double x) {  
    return 1+x+0.5*x*x;  
}  
...  
vector<double> x(10);  
Gaussian G(0,1);  
generate(x.begin(), x.end(), compose(ptr_fun(f), G));
```

Remarque : Si une fonction est surchargée (comme celles de la librairie `math` : `pow`, `cos`, `exp`,...) il faut préciser les types :

```
ptr_fun<double, double>(exp);  
ptr_fun<double, int, double>(pow);
```

Autres adaptateurs :

- ▶ `bind1st` et `bind2nd`
- ▶ `not1` et `not2`
- ▶ `mem_fun` et `mem_fun_ref`

Functors - Objets fonctionnels

Définition

Composition

ptr_fun

Monte-Carlo générique et réduction de variance

Fonction Monte-Carlo

Variables antithétiques

Variable de contrôle

Stratification

Echantillonnage d'importance

Illustration : option Basket

Utilisation de tvmet

Option basket

std::function et Lambda functions en C++11

std::function

Un mot sur les variadic templates

Lambda function

Retour à Monte Carlo

Fonction Monte-Carlo générique

```
2  template <typename Gen>
   std::vector<double> monte_carlo(int n, Gen X)
   {
4     std::vector<double> result(3, 0.);
       double x;
6     for (int j = 0; j < n; j++) {
           x = X();           // restriction sur X...
8         result[0] += x;
           result[1] += x*x;
10    }
       result[0] /= (double) n;
12    result[1] = (result[1] - n*result[0]*result[0])/(double)(n-1);
       result[2] = 1.96*sqrt(result[1]/(double) n);
14    return result;
   }
```

Le vecteur result est composé de (m_n, σ_n^2, t_n) où

$$m_n = \frac{1}{n} \sum_{k=1}^n X_k, \quad \sigma_n^2 = \frac{1}{n-1} \sum_{k=1}^n (X_k - m_n)^2, \quad \text{et} \quad t_n = 1.96 \frac{\sigma_n}{\sqrt{n}}.$$

Exemple d'utilisation

```
1 #include <iostream>
2 #include <vector>
3 #include <cmath>
4 #include "var_alea.hpp"
5 #include "monte-carlo.hpp"
6
7 struct product : public std::unary_function<double, double> {
8     double operator()(double x) const {
9         return exp(x) > 1.2 ? (exp(x)-1.2) : 0;
10    };
11 };
12
13 using namespace std;
14 int main() {
15     init_alea();
16     gaussian G(0,1);
17     product phi;
18     vector<double> res = monte_carlo(1e5, compose(phi, G));
19     cout << res[0] << "\t" << res[1] << endl;
20     return 0;
21 };
```

Variables antithétiques

Soit X une v.a. et T une transformation qui laisse invariante la loi de X i.e.

$$T(X) \sim X$$

Si f est monotone et T décroissante alors $\text{cov}(f(X), f(T(X))) \leq 0$.

Dans ce cas, il est intéressant d'utiliser la représentation

$$\mathbf{E}[f(X)] = \mathbf{E}\left[\frac{f(X) + f(T(X))}{2}\right],$$

pour faire du Monte-Carlo avec $n/2$ itérations (pour garder la même complexité, i.e. le même nombre d'évaluations de f).

- ▶ Si $X \sim \mathcal{U}([0, 1])$ on peut prendre $T(x) = 1 - x$.
- ▶ Si $X \sim \mathcal{N}(0, \text{Id})$ on peut prendre $T(x) = -x$.

Code générique possible

```
1  template <typename Fct, typename Trans>
2  struct antithetic_t
3      : public std::unary_function<typename Fct::argument_type,
4                                     typename Fct::result_type>
5  {
6      antithetic_t(Fct f, Trans T) : f(f), T(T) {};
7      typename Fct::result_type
8      operator()(const typename Fct::argument_type &x) const {
9          return 0.5*(f(x) + f(T(x)));
10     }
11     private:
12         Fct f; Trans T;
13 };
14
15 template <typename Fct, typename Trans>
16 inline antithetic_t<Fct, Trans> antithet(Fct f, Trans T) {
17     return antithetic_t<Fct, Trans>(f, T);
18 };
```

Exemple d'utilisation

```
1 int main() {  
2     init_alea();  
3     gaussian G(0,1);  
4     product phi;  
5     int n = 1e5;  
6  
7     vector<double> res1, res2;  
8     res1 = monte_carlo(n, compose(phi, G));  
9     res2 = monte_carlo(n/2,  
10         compose(antithet(phi, negate<double>()), G));  
11  
12     cout << res1[0] << "\t" << res1[1] << endl;  
13     cout << res2[0] << "\t" << res2[1] << endl;  
14     return 0;  
15 };
```

Exemple d'utilisation

```
1 int main() {  
2     init_alea();  
3     gaussian G(0,1);  
4     product phi;  
5     int n = 1e5;  
6  
7     vector<double> res1, res2;  
8     res1 = monte_carlo(n, compose(phi, G));  
9     res2 = monte_carlo(n/2,  
10         compose(antithet(phi, negate<double>()), G));  
11  
12     cout << res1[0] << "\t" << res1[1] << endl;  
13     cout << res2[0] << "\t" << res2[1] << endl;  
14     return 0;  
};
```

Résultat :

0.786566	3.82878
0.793092	1.66383

Variable de contrôle

Soit X et Y 2 v.a. telles que

- ▶ $\mathbf{E}[Y]$ se calcule efficacement
- ▶ $\text{var}(X - Y) < \text{var}(X)$

alors Y est une variable de contrôle pour X et on utilise la représentation

$$\mathbf{E}[X] = \mathbf{E}[X - Y + \mathbf{E}[Y]],$$

pour faire du Monte-Carlo avec n réalisations de $X - Y$ (ou $X - Y + \mathbf{E}[Y]$).

Variable de contrôle

Soit X et Y 2 v.a. telles que

- ▶ $\mathbf{E}[Y]$ se calcule efficacement
- ▶ $\text{var}(X - Y) < \text{var}(X)$

alors Y est une variable de contrôle pour X et on utilise la représentation

$$\mathbf{E}[X] = \mathbf{E}[X - Y + \mathbf{E}[Y]],$$

pour faire du Monte-Carlo avec n réalisations de $X - Y$ (ou $X - Y + \mathbf{E}[Y]$).

Remarque : Il faut X et Y corrélées !

En général si $X = f(Z)$ alors on construit $Y - \mathbf{E}[Y] = g(Z)$ où g est bien choisie...

On implémente la variable de contrôle dans ce cadre.

On a $\mathbf{E}[g(Z)] = 0$.

Code générique possible

```
1  template <typename Fct1, typename Fct2>
2  struct var_control_t
3      : public std::unary_function<typename Fct1::argument_type,
4                                     typename Fct1::result_type>
5  {
6      var_control_t(Fct1 f, Fct2 g) : f(f), g(g) {};
7      typename Fct1::result_type
8      operator()(const typename Fct1::argument_type &x) const {
9          return (f(x) - g(x));
10     }
11     private:
12         Fct1 f; Fct2 g;
13 };
14
15 template <typename Fct1, typename Fct2>
16 inline var_control_t<Fct1, Fct2> var_control(Fct1 f, Fct2 g) {
17     return var_control_t<Fct1, Fct2>(f, g);
18 }
```

Exemple d'utilisation

```
2 struct product_vc : public std::unary_function<double, double> {  
    double operator()(double x) const { return exp(x) - exp(0.5); };  
};  
4 ...  
int main() {  
6     ...  
    vector<double> res1, res2, res3, res4;  
8     res1 = monte_carlo(n, compose(phi, G));  
    res2 = monte_carlo(n/2,  
10         compose(antithet(phi, negate<double>()), G));  
    res3 = monte_carlo(n,  
12         compose(var_control(phi, phi_vc), G));  
    res4 = monte_carlo(n/2,  
14         compose(antithet(var_control(phi, phi_vc),  
                    negate<double>()), G));
```

Exemple d'utilisation

```
2 struct product_vc : public std::unary_function<double, double> {  
    double operator()(double x) const { return exp(x) - exp(0.5); };  
};  
4 ...  
int main() {  
6     ...  
    vector<double> res1, res2, res3, res4;  
8     res1 = monte_carlo(n, compose(phi, G));  
    res2 = monte_carlo(n/2,  
10         compose(antithet(phi, negate<double>()), G));  
    res3 = monte_carlo(n,  
12         compose(var_control(phi, phi_vc), G));  
    res4 = monte_carlo(n/2,  
14         compose(antithet(var_control(phi, phi_vc),  
                    negate<double>()), G));
```

Résultat :

0.805241	4.17191
0.792391	1.73152
0.797565	0.143292

Exemple d'utilisation

```
1 struct product_vc : public std::unary_function<double, double> {  
2     double operator()(double x) const { return exp(x) - exp(0.5); };  
3 };  
4 ...  
5 int main() {  
6     ...  
7     vector<double> res1, res2, res3, res4;  
8     res1 = monte_carlo(n, compose(phi, G));  
9     res2 = monte_carlo(n/2,  
10         compose(antithet(phi, negate<double>()), G));  
11     res3 = monte_carlo(n,  
12         compose(var_control(phi, phi_vc), G));  
13     res4 = monte_carlo(n/2,  
14         compose(antithet(var_control(phi, phi_vc),  
15             negate<double>()), G));
```

Résultat :

0.805241	4.17191
0.792391	1.73152
0.797565	0.143292
0.794962	0.0130211

Variable de contrôle adaptative

Si Y est une variable de contrôle, alors λY est une variable de contrôle pour tout $\lambda \in \mathbf{R}$.

Quelle est la meilleure variable de contrôle ?

$$\lambda^* = \operatorname{argmin}_{\lambda} \operatorname{var}(X - \lambda Y) \quad \text{i.e.} \quad \lambda^* = \frac{\mathbf{E}[XY]}{\mathbf{E}[Y^2]} \quad (\text{si } \mathbf{E}[Y] = 0).$$

Il est très facile d'adapter le code précédent :

- ▶ on surcharge la fonction `var_control` : si elle prend un argument supplémentaire (la référence sur **double** `lambda`) alors l'objet renvoyé est le functor `adapt_var_control_t` dans lequel on modifie le `lambda` :

```
template <typename Fct1, typename Fct2>  
2 inline adapt_var_control_t<Fct1, Fct2>  
  var_control(Fct1 f, Fct2 g, double &lambda) {  
4     return adapt_var_control_t<Fct1, Fct2>(f, g, lambda);  
  }
```

- ▶ écriture de la classe `adapt_var_control_t`

Variable de contrôle adaptative (2)

```
2  template <typename Fct1, typename Fct2>
3  struct adapt_var_control_t
4      : public std::unary_function<typename Fct1::argument_type,
5                                     typename Fct1::result_type>
6  {
7      adapt_var_control_t(Fct1 f, Fct2 g, double &lambda)
8          : f(f), g(g), lambda(lambda), cov(0), var(0) {};
9      typename Fct1::result_type
10     operator()(const typename Fct1::argument_type &x) {
11         double f_x = f(x), g_x = g(x), result = f_x - lambda*g_x;
12         cov += f_x*g_x;
13         var += g_x*g_x;
14         lambda = var > 0 ? cov / var : lambda;
15         return result;
16     }
17     private:
18         Fct1 f; Fct2 g;
19         double cov, var;
20         double &lambda;
21 };
```

Variable de contrôle adaptative (3)

Remarques :

- ▶ **operator()** non constant...
- ▶ référence sur `lambda` très pratique...

Variable de contrôle adaptative (3)

Remarques :

- ▶ **operator()** non constant...
- ▶ référence sur `lambda` très pratique...

Code de test :

```
2      double lambda = 1.;
      res3 = monte_carlo(n,
          compose(var_control(phi, phi_vc, lambda), G));
4      cout << lambda << endl;
      cout << res3[0] << "\t" << res3[1] << "\t" << res3[2] << endl;
6
      res4 = monte_carlo(n/2,
          compose(antithet(var_control(phi, phi_vc, lambda),
              negate<double>()), G));
8      cout << res4[0] << "\t" << res4[1] << "\t" << res4[2] << endl;
10
```

Variable de contrôle adaptative (3)

Remarques :

- ▶ **operator()** non constant...
- ▶ référence sur `lambda` très pratique...

Code de test :

```
2   double lambda = 1.;
   res3 = monte_carlo(n,
                     compose(var_control(phi, phi_vc, lambda), G));
4   cout << lambda << endl;
   cout << res3[0] << "\t" << res3[1] << "\t" << res3[2] << endl;
6
   res4 = monte_carlo(n/2,
                     compose(antithet(var_control(phi, phi_vc, lambda),
                     negate<double>()), G));
8   cout << res4[0] << "\t" << res4[1] << "\t" << res4[2] << endl;
10
```

Résultat :

0.913989		
0.795478	0.106211	mieux!
0.795338	0.043257	moins bien...

Stratification

On découpe \mathbf{R}^d en une partition $(A_k)_{1 \leq k \leq K}$. Alors on a la représentation suivante

$$\mathbf{E}[F(X)] = \sum_{k=1}^K p_k \mathbf{E}[F(X) \mid X \in A_k]$$

où $p_k = \mathbf{P}[X \in A_k]$.

- ▶ Si on sait simuler $X \mid X \in A_k$ pour tout $k \in \{1, \dots, K\}$
- ▶ Si on connaît les p_k

Alors on a

$$\sum_{k=1}^K p_k \frac{1}{n_k} \sum_{i=1}^{n_k} F(X_i^{(k)}) \xrightarrow{n_1, \dots, n_K \rightarrow +\infty} \mathbf{E}[F(X)]$$

Stratification

On découpe \mathbf{R}^d en une partition $(A_k)_{1 \leq k \leq K}$. Alors on a la représentation suivante

$$\mathbf{E}[F(X)] = \sum_{k=1}^K p_k \mathbf{E}[F(X) \mid X \in A_k]$$

où $p_k = \mathbf{P}[X \in A_k]$.

- ▶ Si on sait simuler $X \mid X \in A_k$ pour tout $k \in \{1, \dots, K\}$
- ▶ Si on connaît les p_k

Alors on a

$$\sum_{k=1}^K p_k \frac{1}{n_k} \sum_{i=1}^{n_k} F(X_i^{(k)}) \xrightarrow{n_1, \dots, n_K \rightarrow +\infty} \mathbf{E}[F(X)]$$

Choix de n_k ? Soit $n = n_1 + \dots + n_K$.

- ▶ Choix optimal : $n_k = \frac{p_k \sigma_{F,k}}{\sum_j p_j \sigma_{F,j}} n$, où $\sigma_{F,k}^2 = \text{var}(F(X) \mid X \in A_k)$
- ▶ Choix proportionnel : $n_k = p_k n$

Simulation loi conditionnelle

- **Dimension 1.** Soit X de fonction de répartition F .

On définit $a_0 = -\infty$, $a_1 = F^{-1}(p_1)$, $a_K = F^{-1}(1)$ et les strates $A_k =]a_{k-1}, a_k]$. Alors $\mathbf{P}[X \in A_k] = p_k$ et

$$X|X \in A_k \sim F^{-1}\left(F(a_{k-1}) + U(F(a_k) - F(a_{k-1}))\right)$$

- $G \sim \mathcal{N}(0, \text{Id}_d)$. Soit v un vecteur unitaire.

Alors $\langle v, G \rangle \sim \mathcal{N}(0, 1)$ et on sait simuler $\langle v, G \rangle | \langle v, G \rangle \in A_k$.

De plus, si $Z \sim G$, $Z \perp\!\!\!\perp G$, alors

$$G \sim Z - \langle v, Z \rangle v + \langle v, G \rangle v$$

Donc

$$\mathbf{E}[F(G)] = \sum_{k=1}^K p_k \mathbf{E}\left[F(Z - \langle v, Z \rangle v + Y^{(k)})\right],$$

où $Y^{(k)} = \langle v, G \rangle | \langle v, G \rangle \in A_k$.

Implémentation

Attention : code fonctionnant uniquement dans le cas de la répartition proportionnelle : $n_k = p_k n$.

```
2  template <typename Fct, typename Gen>
   std::vector<double> stratification(int n, Fct f,
                                     std::list<double> pk,
4                                     std::list<Gen> Gk)
   {
6       std::vector<double> result(3, 0), res_tmp;
       std::list<double>::iterator it_pk = pk.begin();
8       typename std::list<Gen>::iterator it_Gk = Gk.begin();
       while (it_pk != pk.end()) {
10          res_tmp = monte_carlo(floor(n*(*it_pk)), f, *it_Gk);
          result[0] += (*it_pk) * res_tmp[0];
12          result[1] += (*it_pk) * res_tmp[1];
          it_pk++; it_Gk++;
14      }
       result[2] = 1.96*sqrt(result[1]/(double) n);
16      return result;
   }
```

Code de test

```
1 struct gauss_cond : public var_alea<double> {  
2     gauss_cond(double phi_a, double phi_b)  
        : phi_a(phi_a), phi_b(phi_b), U(0,1), s(0,1) {};  
4     double operator()() {  
        return quantile_normal(phi_a + U()*(phi_b - phi_a));  
6     };  
    private:  
8        double phi_a, phi_b;  
        uniform U;  
10 };
```

Exemple d'appel :

```
1     int K = 10;  
2     double p = 1./((double) K);  
        std::list<double> pk;  
4     std::list<gauss_cond> Gk;  
        for (int k = 0; k < K; k++) {  
6         pk.push_back(p);  
         Gk.push_back(gauss_cond(k*p, (k+1)*p));  
8     };  
        vect res = stratification(n, phi, pk, Gk);  
10     cout << res[0] << "\t" << res[1] << endl;
```

Exemple de fonction quantile_normal

```
2 static double a[]={ 2.50662823884, -18.61500062529,  
41.39119773534, -25.44106049637 };  
4 static double b[]={ -8.47351093090, 23.08336743743,  
-21.06224101826, 3.13082909833 };  
6 static double c[]={ 0.3374754822726147, 0.9761690190917186, 0.1607979714918209,  
0.0276438810333863, 0.0038405729373609, 0.0003951896511919,  
0.0000321767881768, 0.0000002888167364, 0.0000003960315187 };  
8  
double quantile_normal(double u) {  
10     double y = u - 0.5;  
12     if (fabs(y) < 0.42) {  
14         double r = y*y, nume = a[3], denom = b[3];  
16         for (int i = 2; i >= 0; i--) {  
18             nume *= r; nume += a[i];  
20             denom *= r; denom += b[i];  
22         }  
24         return y * nume / (denom * r + 1.);  
26     } else {  
28         double r = (u > 0.5) ? log(-log(1-u)) : log(-log(u));  
30         double x = c[8];  
32         for (int i = 7; i >= 0; i--) {  
34             x *= r; x += c[i]; }  
36         return (y < 0) ? -x : x;  
38     }  
39 }  
40 };
```

Voir aussi `boost::math::normal::normal_distribution`.

Echantillonnage d'importance

Changer de mesure pour favoriser l'apparition de réalisations qui ont un impact important sur l'espérance à calculer.

Soit X de densité g et Y de densité \tilde{g} . Alors on a la représentation

$$\mathbf{E}[F(X)] = \mathbf{E}\left[F(Y) \frac{g(Y)}{\tilde{g}(Y)}\right]$$

Echantillonnage d'importance

Changer de mesure pour favoriser l'apparition de réalisations qui ont un impact important sur l'espérance à calculer.

Soit X de densité g et Y de densité \tilde{g} . Alors on a la représentation

$$\mathbf{E}[F(X)] = \mathbf{E}\left[F(Y) \frac{g(Y)}{\tilde{g}(Y)}\right]$$

Choix de \tilde{g} ?

- ▶ Construction ad-hoc, il faut \tilde{g} (simulable) proche de Fg .
- ▶ Changement de mesure exponentielle :

$$\tilde{g}_\theta = e^{\langle \theta, x \rangle - \psi(\theta)} g(x)$$

Implémentation dans le cas de la Gaussienne $\mathcal{N}(0,1)$

```
1  template <typename Fct>
2  struct expo_tilting_gauss
3      : public std::unary_function<double, double> {
4      expo_tilting_gauss(Fct f, double theta) : f(f), theta(theta) {};
5      double operator()(double x) {
6          return f(x + theta) * exp(-theta*x - 0.5*theta*theta);
7      };
8      private:
9          Fct f;
10         double theta;
11 };
12
13 template <typename Fct>
14 expo_tilting_gauss<Fct> expo_tilting(Fct f, double theta) {
15     return expo_tilting_gauss<Fct>(f, theta);
16 };
```

Utilisation :

```
monte_carlo(n, expo_tilting(phi, 1.5), G);
```

Functors - Objets fonctionnels

Définition

Composition

ptr_fun

Monte-Carlo générique et réduction de variance

Fonction Monte-Carlo

Variables antithétiques

Variable de contrôle

Stratification

Echantillonnage d'importance

Illustration : option Basket

Utilisation de tvmet

Option basket

std::function et Lambda functions en C++11

std::function

Un mot sur les variadic templates

Lambda function

Retour à Monte Carlo

Utilisation de tvmet

En live.

Illustration : option Basket

En live.

Functors - Objets fonctionnels

Définition

Composition

ptr_fun

Monte-Carlo générique et réduction de variance

Fonction Monte-Carlo

Variables antithétiques

Variable de contrôle

Stratification

Echantillonnage d'importance

Illustration : option Basket

Utilisation de tvmet

Option basket

std::function et Lambda functions en C++11

std::function

Un mot sur les variadic templates

Lambda function

Retour à Monte Carlo

std::function

Voici le prototype définie dans le header functional

```
template< class R, class... Args >  
class function<R(Args...)>
```

Un objet function peut stocker n'importe quel contenu muni d'un opérateur () : une fonction ou un objet fonctionnel. L'utilisation est très souple.

Exemples

```
double mypow(double x, unsigned n) { // ... }  
std::function<double(double, unsigned)> f;  
f = mypow;  
  
std::function<double(double)> cube = std::bind(f, _1, 3);  
  
struct MyPow {  
    double operator()(double, unsigned) const;  
};  
f = MyPow();
```

Comment ça marche avec les méthodes ?

```
struct MaClasse {  
    int fct(int);  
};
```

std::function suite

La "vraie" signature de la méthode fct est `int fct(MaClasse *, int)` et son vrai nom est `MaClasse::fct`. Ainsi on peut écrire

```
1 struct MaClasse {  
2     int fct(int);  
3 };  
4 std::function<int(MaClasse *, int)> f = &MaClasse::fct;  
5  
6 MaClasse objet;  
7 f(&objet, 4);  
8  
9 std::function<int(int)> g = std::bind(&MaClasse::fct, &objet, _1);  
10 g(4);
```

La version "référence" marche aussi

```
1 struct MaClasse {  
2     int fct(int);  
3 };  
4 std::function<int(MaClasse &, int)> f = &MaClasse::fct;  
5  
6 MaClasse objet;  
7 f(objet, 4);  
8  
9 std::function<int(int)> g = std::bind(&MaClasse::fct, objet, _1);  
10 g(4);
```

Définition récursive des variadic templates

Les variadic templates (utilisant la syntaxe ...) permettent de passer un nombre quelconque d'arguments, de types quelconques. Le mécanisme est récursif et on peut l'illustrer sur l'exemple suivant :

```
template<typename T, typename ... Args>
2 void f(T head, Args ... args)
{
4     g(head);           // on execute g avec le premier argument
    f(args...);         // on rappelle f avec les arguments restants
6 }
void f() { }            // code de la fonction f sans argument
```

```
template<typename T>
2 void g(T x)
{
4     std::cout << x << " ";
}
```

Exemples d'appels de la fonction f

```
2 f("coucou", 0, 3.56);
f('c', 1e5, 1, 2, "stop");
```

Lambda function

On peut voir une Lambda function comme un objet fonctionnel dont l'écriture syntaxique est plus épurée. Prenons l'exemple suivant :

```
2 struct compare {  
    bool operator()(int a, int b) const { return abs(a) < abs(b); }  
}  
4 vector<int> v = {50, -10, 20, -30};  
    std::sort(v.begin(), v.end()); // the default sort  
6    std::sort(v.begin(), v.end(), compare());
```

On peut remplacer l'objet compare par une lambda function dont la syntaxe est la suivante :

```
| std::sort(v.begin(), v.end(), [](int a, int b) {return abs(a) < abs(b);});
```

On peut stocker une lambda function dans un objet `std::function`.

```
2 std::function<bool(int, int)> f2 = [](int a, int b) {return abs(a) < abs(b);};  
    auto f = [](int a, int b) { return abs(a) < abs(b); };
```

Capture and paramètres

Les crochets `[]` sont obligatoires et définissent une liste de capture indiquant si des variables locales doivent être des paramètres de la lambda function. Voici les options

- ▶ `[]` liste vide, aucun paramètre capturé
- ▶ `[=]` toutes les variables locales utilisées dans le code de la lambda sont capturées *par copie*
- ▶ `[&]` toutes les variables locales utilisées dans le code de la lambda sont capturées *par référence*
- ▶ `[x]` uniquement la variable `x` capturée par copie
- ▶ `[x, &y]` la variable `x` capturée par copie et la variable `y` capturée par référence

Exemple :

```
2 void print_modulo(const vector<int>& v, ostream& os, int m) {  
  for_each(v.begin(), v.end(),  
    [&os,m](int x) { if (x%m==0) os << x << '\n'; });}
```


Functor équivalent à la lambda function précédente

```
class Modulo_print {  
2     ostream& os; // members to hold the capture list  
    int m;  
4 public:  
    Modulo_print(ostream& s, int mm) :os(s), m(mm) {}  
6     void operator()(int x) const { if (x%m==0) os << x << '\n'; }  
};  
8  
void print_modulo(const vector<int>& v, ostream& os, int m) {  
10     for_each(begin(v),end(v),Modulo_print{os,m});  
}
```

Classe `linear_estimator`

```
1 class linear_estimator {
2 public:
3     linear_estimator() { reinit(); }
4     linear_estimator(unsigned size)
5         : _sum(size), _sum_of_squares(size), _sample_size(size) { reinit(); }
6     void reinit() {
7         _sum = 0; _sum_of_squares = 0; _sample_size = 0;
8         _time_start = std::chrono::steady_clock::now();
9         _time_end = _time_start;
10        _time_span = std::chrono::duration<double>(0);
11    }
12    double time() const { return _time_span.count(); }
13    linear_estimator & operator+=(linear_estimator const & other);
14 protected:
15     double _sum, _sum_of_squares;
16     unsigned _sample_size;
17     std::chrono::steady_clock::time_point _time_start;
18     std::chrono::steady_clock::time_point _time_end;
19     std::chrono::duration<double> _time_span;
20 };
```

Utilisation de `std::chrono` pour mesurer le temps de calcul.

Exemple d'une classe Monte Carlo

```
1 template <typename Generator>
2 class monte_carlo : public linear_estimator {
3 public:
4     typedef Generator TGenerator;
5     monte_carlo(std::function<double(Generator &)> X) : _random_variable(X) {}
6     double operator()(Generator & gen, unsigned M);
7     double mean() const { return this->_sum / this->_sample_size; }
8     double mean_of_squares() const { // ... }
9     double var() const { // ... }
10    double var_est() const { // ... }
11    double st_dev() const { return sqrt(var()); }
12    double ic() const { return 1.96*sqrt(var_est()); }
13    template <typename G2>
14    friend std::ostream & operator<<(std::ostream & o, monte_carlo<G2> const & M)
15 private:
16    std::function<double(Generator &)> _random_variable;
17 };
```

Exercice : compléter les fonctions `mean_of_squares`, `var` et `var_est`.

Operator () de la classe monte_carlo

```
2  template <typename Generator>
   double monte_carlo<Generator>::operator()(Generator & gen, unsigned M) {
3      _time_start = std::chrono::steady_clock::now();
4      for (unsigned m = 0; m < M; ++m) {
5          double x = _random_variable(gen);
6          _sum += x;
7          _sum_of_squares += x*x;
8      }
9      _sample_size += M;
10     _time_end = std::chrono::steady_clock::now();
11     _time_span = _time_end - _time_start;
12     return mean();
   };
```

Exercice : écrire un code d'exemple et recoder antithetic et variable de contrôle en C++11