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
- Echantillonage 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 facilier l'écriture et la manipulation de ces functors, 2 classes abstraites sont définies dans la STL :

- Unary Fonction

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

- Binary Fonction

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

Functors - Objets fonctionnels (2)

De la même façon on définit la classe abstraite `generator` :

```
template <class _Result>
struct generator
{
    typedef _Result result_type;
};
```

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 :

```
struct less_mag : public binary_function<double, double, bool> {
   bool operator()(double x, double y) { return fabs(x) < fabs(y); }
};

vector<double> V;
...
sort(V.begin(), V.end(), less_mag());
```

Composition de Functors (1)

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

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

Composition de Functors (1)

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

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

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

Composition de Functors (1)

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

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

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

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 :

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

- 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 :

```
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 :

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

- 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 :

```
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) {
    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 le librairie math : pow, cos, exp,...) il faut préciser les types :

```
ptr_fun<double, double>(exp);
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 le 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

Fonction Monte-Carlo générique

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

```
#include <iostream>
#include <vector>
#include <cmath>
#include "var_alea.hpp"
#include "monte-carlo.hpp"

struct product : public std::unary_function<double, double> {
    double operator()(double x) const {
        return exp(x) > 1.2 ? (exp(x)-1.2) : 0;
    };
};

using namespace std;
int main() {
    init_alea();
    gaussian G(0,1);
    product phi;
    vector<double> res = monte_carlo(1e5, compose(phi, G));
    cout << res[0] << "\t" << res[1] <<  endl;
    return 0;
};
```

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 $\operatorname{cov}(f(X), f(T(X))) \leqslant 0$.

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 $X \sim \mathcal{N}(0, \mathrm{Id})$ on peut prendre $T(x) = -x$.

Code générique possible

```
template <typename Fct, typename Trans>
struct antithetic_t
    : public std::unary_function<typename Fct::argument_type,
                                 typename Fct::result_type>
{
    antithetic_t(Fct f, Trans T) : f(f), T(T) {};
    typename Fct::result_type
    operator()(const typename Fct::argument_type &x) const {
        return 0.5*(f(x) + f(T(x)));
    }
    private:
        Fct f; Trans T;
};

template <typename Fct, typename Trans>
inline antithetic_t<Fct, Trans> antithet(Fct f, Trans T) {
    return antithetic_t<Fct, Trans>(f, T);
};
```

Exemple d'utilisation

```
int main() {
    init_alea();
    gaussian G(0,1);
    product phi;
    int n = 1e5;

    vector<double> res1, res2;
    res1 = monte_carlo(n, compose(phi, G));
    res2 = monte_carlo(n/2,
                compose(antithet(phi, negate<double>()), G));

    cout << res1[0] << "\t" << res1[1] <<  endl;
    cout << res2[0] << "\t" << res2[1] <<  endl;
    return 0;
};
```

Exemple d'utilisation

```
int main() {
    init_alea();
    gaussian G(0,1);
    product phi;
    int n = 1e5;

    vector<double> res1, res2;
    res1 = monte_carlo(n, compose(phi, G));
    res2 = monte_carlo(n/2,
                compose(antithet(phi, negate<double>()), G));

    cout << res1[0] << "\t" << res1[1] <<  endl;
    cout << res2[0] << "\t" << res2[1] <<  endl;
    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
- $\mathrm{var}(X - Y) < \mathrm{var}(X)$

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

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

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
- $\mathrm{var}(X - Y) < \mathrm{var}(X)$

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

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

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

```
template <typename Fct1, typename Fct2>
struct var_control_t
    : public std::unary_function<typename Fct1::argument_type,
                                 typename Fct1::result_type>
{
    var_control_t(Fct1 f, Fct2 g) : f(f), g(g) {};
    typename Fct1::result_type
    operator()(const typename Fct1::argument_type &x) const {
        return (f(x) - g(x));
    }
    private:
        Fct1 f; Fct2 g;
};

template <typename Fct1, typename Fct2>
inline var_control_t<Fct1, Fct2> var_control(Fct1 f, Fct2 g) {
    return var_control_t<Fct1, Fct2>(f, g);
}
```

Exemple d'utilisation

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

Exemple d'utilisation

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

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

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

Il est très facile d'adapter le code précédent :

- on surcharge la fonction `var_control` : si elle prend un argument suplémentaire (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>
inline adapt_var_control_t<Fct1, Fct2>
var_control(Fct1 f, Fct2 g, double &lambda) {
    return adapt_var_control_t<Fct1, Fct2>(f, g, lambda);
}
```

- écriture de la classe `adapt_var_control_t`

Variable de contrôle adaptative (2)

```
template <typename Fct1, typename Fct2>
struct adapt_var_control_t
    : public std::unary_function<typename Fct1::argument_type,
                                 typename Fct1::result_type>
{
    adapt_var_control_t(Fct1 f, Fct2 g, double &lambda)
        : f(f), g(g), lambda(lambda), cov(0), var(0) {};
    typename Fct1::result_type
    operator()(const typename Fct1::argument_type &x) {
        double f_x = f(x), g_x = g(x), result = f_x - lambda*g_x;
        cov += f_x*g_x;
        var += g_x*g_x;
        lambda = var > 0 ? cov / var : lambda;
        return result;
    }
    private:
        Fct1 f; Fct2 g;
        double cov, var;
        double &lambda;
};
```

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 :

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

Variable de contrôle adaptative (3)

Remarques :

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

Code de test :

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

Résultat :

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

Stratification

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

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

où $p_k = \mathbf{P}\left[X \in A_k\right]$.

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

Alors on a

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

Stratification

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

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

où $p_k = \mathbf{P}\left[X \in A_k\right]$.

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

Alors on a

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

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

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

Simulation loi conditionnelle

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

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

- $G \sim \mathcal{N}(0, \mathrm{Id}_d)$. 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 : $n_k = p_k n$.

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

Code de test

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

Exemple d'appel :

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

Exemple de fonction quantile_normal

```
static double a[]={ 2.50662823884, -18.61500062529,
                    41.39119773534, -25.44106049637 };
static double b[]={ -8.47351093090, 23.08336743743,
                    -21.06224101826, 3.13082909833 };
static double c[]={ 0.3374754822726147, 0.9761690190917186, 0.1607979714918209,
    0.0276438810333863, 0.0038405729373609, 0.0003951896511919,
    0.0000321767881768, 0.0000002888167364, 0.0000003960315187 };

double quantile_normal(double u) {
    double y = u - 0.5;
    if (fabs(y) < 0.42) {
        double r = y*y, nume = a[3], denom = b[3];
        for (int i = 2; i >= 0; i--) {
            nume *= r; nume += a[i];
            denom *= r; denom += b[i];
        }
        return y * nume / (denom * r  + 1.);
    } else {
        double r = (u > 0.5) ? log(-log(1-u)) : log(-log(u));
        double x = c[8];
        for (int i = 7; i >= 0; i--) {
            x *= r; x += c[i]; }
        return (y < 0) ? -x : x;
    }
};
```

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

Echantillonage 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

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

Echantillonage 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

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

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 $\mathcal{N}(0,1)$

```
template <typename Fct>
struct expo_tilting_gauss
        : public std::unary_function<double, double> {
    expo_tilting_gauss(Fct f, double theta) : f(f), theta(theta) {};
    double operator()(double x) {
        return f(x + theta) * exp(-theta*x - 0.5*theta*theta);
    };
    private:
        Fct f;
        double theta;
};

template <typename Fct>
expo_tilting_gauss<Fct> expo_tilting(Fct f, double theta) {
    return expo_tilting_gauss<Fct>(f, theta);
};
```

Utilisation :

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

Utilisation de tvmet

En live.

Illustration : option Basket

En live.

std::function

Voici le protoype 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

```
struct MaClasse {
    int fct(int);
};
std::function<int(MaClasse *, int)> f = &MaClasse::fct;

MaClasse objet;
f(&objet, 4);

std::function<int(int)> g = std::bind(&MaClasse::fct, &objet, _1);
g(4);
```

La version "référence" marche aussi

```
struct MaClasse {
    int fct(int);
};
std::function<int(MaClasse &, int)> f = &MaClasse::fct;

MaClasse objet;
f(objet, 4);

std::function<int(int)> g = std::bind(&MaClasse::fct, objet, _1);
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>
void f(T head, Args ... args)
{
    g(head);          // on execute g avec le premier argument
    f(args...);       // on rappelle f avec les arguments restants
}
void f() { }          // code de la fonction f sans argument
```

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

Exemples d'appels de la fonction f

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

Lambda function

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

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

```
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 :

```
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 {
    ostream& os; // members to hold the capture list
    int m;
public:
    Modulo_print(ostream& s, int mm) :os(s), m(mm) {}
    void operator()(int x) const { if (x%m==0) os << x << '\n'; }
};

void print_modulo(const vector<int>& v, ostream& os, int m) {
   for_each(begin(v),end(v),Modulo_print{os,m});
}
```

Classe linear_estimator

```
class linear_estimator {
public:
    linear_estimator() { reinit(); }
    linear_estimator(unsigned size)
        : _sum(size), _sum_of_squares(size), _sample_size(size) { reinit(); }
    void reinit() {
        _sum = 0; _sum_of_squares = 0; _sample_size = 0;
        _time_start = std::chrono::steady_clock::now();
        _time_end = _time_start;
        _time_span = std::chrono::duration<double>(0);
    }
    double time() const { return _time_span.count(); }
    linear_estimator & operator+=(linear_estimator const & other);
protected:
    double _sum, _sum_of_squares;
unsigned _sample_size;
    std::chrono::steady_clock::time_point _time_start;
    std::chrono::steady_clock::time_point _time_end;
    std::chrono::duration<double> _time_span;
};
```

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

Exemple d'une classe Monte Carlo

```
template <typename Generator>
class monte_carlo : public linear_estimator {
public:
    typedef Generator TGenerator;
    monte_carlo(std::function<double(Generator &)> X) : _random_variable(X) {}
    double operator()(Generator & gen, unsigned M);
    double mean() const { return this->_sum / this->_sample_size; }
    double mean_of_squares() const { // ... }
    double var() const { // ... }
    double var_est() const { // ... }
    double st_dev() const { return sqrt(var()); }
    double ic() const { return 1.96*sqrt(var_est()); }
    template <typename G2>
    friend std::ostream & operator<<(std::ostream & o, monte_carlo<G2> const & MC
private:
    std::function<double(Generator &)> _random_variable;
};
```

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

Operator ( ) de la classe monte_carlo

```
template <typename Generator>
double monte_carlo<Generator>::operator()(Generator & gen, unsigned M) {
    _time_start = std::chrono::steady_clock::now();
    for (unsigned m = 0; m < M; ++m) {
        double x = _random_variable(gen);
        _sum += x;
        _sum_of_squares += x*x;
    }
    _sample_size += M;
    _time_end = std::chrono::steady_clock::now();
    _time_span = _time_end - _time_start;
    return mean();
};
```

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