Simulation de variables aléatoires

Exemples en C++

Vincent Lemaire
vincent.lemaire@upmc.fr

Loi uniforme

- Dans la librairie standard du C et du C++

Inversion de la fonction de répartition

Méthode du rejet -1-(von Neumann)

- Densité log-concave

Méthode du rejet -2-(Forsythe-von Neumann)

- Loi exponentielle (von Neumann)
- Généralisation

Méthodes adhoc

- Gaussienne
- IG - Gaussienne inverse
- NIG - Normal inverse gaussian

Dimension supérieure

- Vecteurs Gaussiens

En C++11

Brique de base : l'uniforme !

- Générateurs congruentiels :

$$
\begin{aligned}
x_n &= a_1 x_{n-1} + \cdots + a_k x_{n-k} \mod m \\
u_n &= x_n / m
\end{aligned}
$$

où $m, k \in \mathbb{N}^*$, $a_i \in \mathbb{Z}_m = \{0, \ldots, m-1\}$.
- L'état du générateur à l'itération n est $(x_{n-k+1}, \ldots, x_n)$.
- L'état initial est appelé graine (*seed*).
- Pour $k = 1$ on parle de *Linear congruential generator* (LCG) :

- Linear feedback shift register (LFSR) :

$$
\begin{aligned}
x_n &= a_1 x_{n-1} + \cdots + a_k x_{n-k} \mod 2 \\
u_n &= \sum_{j=1}^{L} x_{ns+j-1} 2^{-j}.
\end{aligned}
$$

avec s et $L \leqslant k$.

- MT19937

A propos de rand()...

- Informations obtenues en tapant : `man 3 rand`

```
NAME
     rand, srand, sranddev, rand_r -- bad random number generator
LIBRARY
     Standard C Library (libc, -lc)
SYNOPSIS
     #include <stdlib.h>

     void     srand(unsigned seed);

     void     sranddev(void);

     int     rand(void);

     int     rand_r(unsigned *ctx);
DESCRIPTION
     These interfaces are obsoleted by random(3).
```

A propos de rand48()...

```
NAME
     drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48,
     lcong48 -- pseudo random number generators and initialization routines
LIBRARY
     Standard C Library (libc, -lc)
SYNOPSIS
     #include <stdlib.h>

     double     drand48(void);
     void      srand48(long seed);
     (...)
DESCRIPTION
     The rand48() family of functions generates pseudo-random numbers using a
     linear congruential algorithm working on integers 48 bits in size.  The
     particular formula employed is r(n+1) = (a * r(n) + c) mod m where the
     default values are for the multiplicand a = 0xfdeece66d = 25214903917 and
     the addend c = 0xb = 11.  The modulo is always fixed at m = 2 ** 48.
     r(n) is called the seed of the random number generator.

     (...)
```

A propos de random()...

```
NAME
     random, srandom, srandomdev, initstate, setstate -- better random number
     generator; routines for changing generators
LIBRARY
     Standard C Library (libc, -lc)
SYNOPSIS
     #include <stdlib.h>

     long     random(void);
     void     srandom(unsigned long seed);
DESCRIPTION
     The random() function uses a non-linear additive feedback random number
     generator employing a default table of size 31 long integers to return
     successive pseudo-random numbers in the range from 0 to (2**31)-1.  The
     period of this random number generator is very large, approximately
     16*((2**31)-1).

     The random() and srandom() functions have (almost) the same calling
     sequence and initialization properties as the rand(3) and srand(3) func-
     tions.  The difference is that rand(3) produces a much less random
     sequence -- in fact, the low dozen bits generated by rand go through a
     cyclic pattern.  All the bits generated by random() are usable.
     (...)
```

Exemple d'intégration d'un générateur en C++

- Code C très optimisé :
  `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html`

- Quelques fonctions codées dans le fichier `mt19937ar.c` :

```
init_genrand(seed) initializes the state vector by using one
unsigned 32-bit integer "seed", which may be zero.

genrand_int32() generates unsigned 32-bit integers.
genrand_int31() generates unsigned 31-bit integers.
genrand_real1() generates uniform real in [0,1] (32-bit resolution).
genrand_real2() generates uniform real in [0,1) (32-bit resolution).
genrand_real3() generates uniform real in (0,1) (32-bit resolution).
genrand_res53() generates uniform real in [0,1) with 53-bit resolution.
```

Encapsulation C++-1-

Classe générique var_alea qui constitue une coquille vide dont les autres variables aléatoires vont héritées.

```
template <class T>
struct var_alea
{
    typedef T result_type;
    var_alea() : value(0) {};
    var_alea(T value) : value(value) {};
    virtual ~var_alea() {};
    virtual T operator()() = 0;
    T current() const { return value; };
    protected:
        T value;
};
```

Encapsulation C++-2-

```
struct uniform : public var_alea<double>
{
    uniform(double left = 0, double right = 1)
        : left(left), size(right-left) {
        genrand = genrand_real3;
    };
    double operator()() {
        return value = left + size * genrand();
    };
    private:
        double left, size;
        double (*genrand)(void);
};
```

Rq : ne pas oublier d'appeler la fonction `init_genrand()` en début de programme. On pourrait aussi l'encapsuler.

Avec BOOST...

```
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/normal_distribution.hpp>
#include <boost/random/variate_generator.hpp>

typedef boost::mt19937 generator;
typedef boost::normal_distribution<double> normal_dist;
typedef boost::variate_generator< generator&,
                                  normal_dist > normal_rv;

int main()
{
    generator gen;
    normal_rv G(gen, normal_dist(0,1));

    gen.seed(static_cast<unsigned int>(std::time(0)));

    std::cout << G() << std::endl;
    return 0;
}
```

Incorporé maintenant dans le standard C++11, on en reparle en fin de séance.

Inversion de la fonction de répartition

- Soit une loi μ de fonction de répartition F continue et strictement croissante. Alors F a une réciproque F^{-1} définie sur $\mathscr{D}_{F^{-1}} \subset [0,1]$. Si $U \sim \mathcal{U}(\mathscr{D}_{F^{-1}})$, alors

$$X = F^{-1}(U) \sim \mu$$

- Si F est discontinue en certains points (lois discrètes) ou seulement croissante le résultat reste vrai en remplaçant la réciproque F^{-1} par l'inverse à gauche F_l^{-1}

$$\forall u \in]0,1[, \quad F_l^{-1}(u) = \inf \{x \in \mathscr{D}_F / F(x) \geqslant u\}$$

- Exemple : loi Exponentielle, $\lambda > 0$

$$X = -\frac{\log(U)}{\lambda} \sim \mathscr{E}(\lambda)\,.$$

Méthode du rejet

- On sait simuler $Y \sim g(x)\mathrm{d}x$.
- Il existe $C > 1$ tel que $\forall x, f(x) \leqslant Cg(x)$.
- Simulation de $X \sim f(x)\mathrm{d}x$:

```
do
    Yn = realisation selon g;
    Un = uniforme([0,1]);
while (C Un g(Yn) >= f(Yn))
return Yn
```

- Si
$$\tau = \inf\left\{n \geqslant 1, CU_n g(Y_n) < f(Y_n)\right\},$$
alors τ suit une loi géométrique de paramètre $p = \frac{1}{C}$ et
$$Y_\tau \sim Y_1 | \left\{CUg(Y_1) < f(Y_1)\right\} \sim f(x)\mathrm{d}x.$$

Densité log-concave

- Soit f log-concave sur $[0, +\infty[$ avec un mode en 0.
  On suppose que $f(0) = 1$.
- On a la majoration suivante :

$$\forall x \geqslant 0, \quad f(x) \leqslant \min(1, e^{1-x}).$$

- Simulation de $X \sim f$:

```
do
    Un = uniforme([0,2]);
    Vn = uniforme([0,1]);
    if (Un <= 1)
        then (Xn, Zn) = (Un, Vn)
        else (Xn, Zn) = (1-log(Un-1), Vn(Un-1))
while (Zn > f(Xn))
return Xn
```

Loi exponentielle (von Neumann)

Soit $X \sim \mathscr{E}(1)$.

- $X = (N-1) + Y$ où
  - N et X sont indépendantes,
  - N est une loi géométrique de paramètre e^{-1},
  - Y est une loi exponentielle conditionnée à $]0,1[$ *i.e.*

$$\mathbf{P}\left[Y \in \mathrm{d}y\right] = \frac{e^{-y}}{1-e^{-1}} \mathbf{1}_{]0,1[}(y)\mathrm{d}y$$

- Simulation de Y

```
do
    U1 = uniforme([0,1]);  Unm1 = U1;  n = 2;
    Un = uniforme([0,1]);
    while (Unm1 >= Un)
        Unm1 = Un;  Un = uniforme([0,1]);  n++;
while (n impair)
return U1
```

- Ligne 6 : sortie de la boucle **`do...while`** lorsque n est pair
- n est une réalisation de la v.a. N définie par

$$N = \inf \{n \geqslant 2, U_1 \geqslant \cdots \geqslant U_{n-1} < U_n\}$$

  de loi $\mathbf{P}[N > n] = \mathbf{P}[U_1 \geqslant \cdots \geqslant U_n] = \frac{1}{n!}$.
- Ligne 7 : on retourne $U_1(\omega)$ lorsque $N(\omega)$ est pair *i.e.*

$$\forall y \in]0,1[, \quad \mathbf{P}[Y \leqslant y] = \mathbf{P}[U_1 \leqslant y \mid N \text{ pair}]$$

- Loi de Y ? On a pour $y \in]0,1[$

$$\mathbf{P}[U_1 \leqslant y, N > n] = \mathbf{P}[y \geqslant U_1 \geqslant \cdots \geqslant U_n] = \frac{1}{n!}\mathbf{P}\left[\max_i U_i \leqslant y\right] = \frac{y^n}{n!}$$

  donc

$$\mathbf{P}[U_1 \leqslant y, N = n] = \frac{y^{n-1}}{(n-1)!} - \frac{y^n}{n!},$$

  d'où

$$\mathbf{P}[U_1 \leqslant y, N\text{pair}] = \sum_{k \geqslant 1} \mathbf{P}[U_1 \leqslant y, N = 2k] = 1 - e^{-y}$$

Généralisations

- $X \sim \mathscr{E}(\lambda)$ facile ! (exercice)
- (Forsythe, von Neumann) X de densité

$$f(x) = Ce^{-F(x)}g(x),$$

où F est une fonction continue $0 \leqslant F(x) \leqslant 1$ et g une densité de proba.

```
do
    X = real.de g;  U1 = F(X);  Unm1 = U1;  n = 2;
    Un = uniforme([0,1]);
    while (Unm1 >= Un)
        Unm1 = Un;  Un = uniforme([0,1]);  n++;
while (n impair)
return X
```

Exemple : loi gamma (tronquée) de paramètre $\alpha \in]0,1[$.

- Cas où $F \geqslant 0$ (mais pas $\leqslant 1$) : on partitionne $]0,+\infty[$ en $q_0, \ldots, q_R$ où $q_0 = 0$ et $q_k = \sup_x F(x) - F(q_{k-1}) \leqslant 1$ (avec R t.q. $\int_0^{q_R} f(x)\mathrm{d}x = 1$). Puis on sélectionne l'intervalle $]q_{k-1}, q_k[$ en tirant une uniforme sur $]0,1[$...

Simulation d'une Gaussienne

- **Box-Muller** : Si R^2 et Θ indépendantes telles que $R^2 \sim \mathscr{E}(1/2)$ et $\Theta \sim \mathscr{U}([0, 2\pi])$, alors

$$X = (R\cos\Theta, R\sin\Theta) \sim \mathscr{N}(0, \mathrm{Id}_2)\,.$$

- **Méthode polaire (Marsaglia)** : Si $(U, V) \sim \mathscr{U}(\mathscr{B}(0,1))$ et $R^2 = U^2 + V^2$, alors

$$X = \left(U\sqrt{-2\frac{\log(R^2)}{R^2}}, V\sqrt{-2\frac{\log(R^2)}{R^2}}\right) \sim \mathscr{N}(0, \mathrm{Id}_2)\,.$$

- **Ziggurat** : Méthode de rejet se basant sur des tables précalculées liées à la distribution gaussienne.
  Très rapide (mais nécessite des tables).

Méthode polaire

Exemple d'implémentation de la méthode de Marsaglia

```
struct gaussian : public var_alea<double>
{
    gaussian(double mean = 0, double std = 1)
            : mean(mean), std(std), flag(true), unif(-1,1) {};
    double operator()() {
        flag = !flag;
        if (!flag) {
            do {
                U = unif(); V = unif();
                R2 = U*U + V*V;
            } while (R2 > 1);
            rac = sqrt(-2 * log(R2) / R2);
            return value = mean + std * U * rac;
        } else
            return value = mean + std * V * rac;
    };
    private:
        double mean, std, U, V, R2, rac;
        uniform unif;
        bool flag;
};
```

IG - Gaussienne inverse

- Soit $(X_t)_{t \geqslant 0}$ un Brownien avec drift (tendance) $\nu > 0$ *i.e.*

  $$X_0 = 0, \quad X_t = \nu t + \sigma B_t$$

  Le premier temps de passage par $(X_t)_{t \geqslant 0}$ d'un niveau $\alpha > 0$ vérifie une loi Gaussienne inverse

  $$\tau_\alpha = \inf\{t > 0, X_t = \alpha\} \sim \mathcal{IG}\left(\frac{\alpha}{\nu}, \frac{\alpha^2}{\sigma^2}\right)$$

- La densité de $X \sim \mathcal{IG}(\mu, \lambda)$ est donnée par

  $$\forall x > 0, \quad f(x) = \sqrt{\frac{\lambda}{2\pi x^3}} e^{-\frac{\lambda(x-\mu)^2}{2\mu^2 x}}.$$

- Pour la simulation, on utilise

  $$Y = \frac{\lambda(X-\mu)^2}{\mu^2 X} \sim \chi^2$$

- 2 racines positives $X^{(+)}$ et $X^{(-)}$ (pour tout $\mu, \lambda > 0$)

$$
\begin{aligned}
X^{(\pm)} &= \frac{\mu}{2\lambda}\left(2\lambda + \mu Y \pm \sqrt{4\lambda\mu Y + \mu^2 Y^2}\right), \\
&= Z \pm \sqrt{Z^2 - \mu^2}, \quad \text{avec} \quad Z = \mu + \frac{\mu^2}{2\lambda} Y.
\end{aligned}
$$

- Michael, Schucany, Haas (76) proposent l'algorithme suivant :

```
struct inverse_gaussian : public var_alea<double>
{
    inverse_gaussian(double lambda, double mu) : (...);
    double operator()() {
        double Z = mu + 0.5*mu*mu/lambda*Y();
        double rac = sqrt(Z*Z - mu*mu);
        return value = (U() < mu/(mu+Z+rac)) ? Z+rac : Z-rac;
    };
    private:
        double lambda, mu;
        chi_deux Y;
        uniform U;
};
```

Choix de la racine (d'après leur article)

Soit $Y = \psi(X)$, où X de densité f_X et $\psi \in \mathscr{C}^1$ (dimension 1).

- Soit y fixé et $I_y^h =]y-h, y+h[$, $h > 0$.
- Il existe $\psi_1^{-1}, \dots, \psi_n^{-1}$ fonctions continues t.q. $x_i = \psi_i^{-1}(y)$, $(x_i \neq x_j)$.
- Par définition (on suppose que les zéros x_i sont isolés et h petit)

$$\begin{aligned}\mathbf{P}\left[X \in \psi_i^{-1}(I_y^h) \,\middle|\, Y \in I_y^h\right] &= \frac{\mathbf{P}\left[X \in \psi_i^{-1}(I_y^h)\right]}{\sum_j \mathbf{P}\left[X \in \psi_j^{-1}(I_y^h)\right]}, \\ &= \left(1 + \sum_{j \neq i} \frac{\mathbf{P}\left[X \in \psi_j^{-1}(I_y^h)\right]}{\mathbf{P}\left[X \in \psi_i^{-1}(I_y^h)\right]}\right)^{-1}\end{aligned}$$

- De plus,

$$\mathbf{P}\left[X \in \psi_i^{-1}(I_y^h)\right] = \int_{y-h}^{y+h} \frac{f_X \circ \psi_i^{-1}(z)}{\left|\psi_i' \circ \psi_i^{-1}(z)\right|} \mathrm{d}z \xrightarrow{h \to 0} \frac{f_X(x_i)}{|\psi'(x_i)|}$$

- Conclusion :

$$\mathbf{P}\left[\{\text{on choisit } x_i\} \mid Y = y\right] = \left(1 + \sum_{j \neq i} \frac{|\psi'(x_i)|}{|\psi'(x_j)|} \frac{f_X(x_j)}{f_X(x_i)}\right)^{-1}$$

NIG - Normal inverse gaussian

- X suit une NIG de paramètres α, β, μ et δ si $0 \leqslant |\beta| \leqslant \alpha$, $\delta > 0$

$$
\begin{aligned}
Y \sim \mathcal{IG}(\delta/\gamma, \delta^2), \quad \gamma = \sqrt{\alpha^2 - \beta^2}, \\
X|Y = y \sim \mathcal{N}(\mu + \beta y, y)
\end{aligned}
$$

- La densité de X est donnée par

$$
\forall x \in \mathbf{R}, \quad f(x) = \frac{\alpha\delta K_1\left(\alpha\sqrt{\delta^2 + (x-\mu)^2}\right)}{\pi\sqrt{\delta^2 + (x-\mu)^2}} e^{\delta\gamma + \beta(x-\mu)},
$$

```
struct normal_inverse_gaussian : public var_alea<double>
{
    normal_inverse_gaussian(double alpha, double beta, (...)
    double operator()() {
        double y_ = Y();
        return value = mu + beta*y_ + sqrt(y_) * G();
    };
    private:
        double alpha, beta, mu, delta;
        gaussian G;
        inverse_gaussian Y;
};
```

Vecteurs Gaussiens

Soit $X \sim \mathcal{N}(0, \Sigma)$ où Σ est une matrice de variance-covariance (symétrique définie positive).

- Par la transformation de Cholesky, il existe L matrice triangulaire inférieure telle que
$$\Sigma = LL^t.$$
- Si $G \sim \mathcal{N}(0, \mathrm{Id}_d)$ alors
$$X = LG \sim \mathcal{N}(0, \Sigma).$$

Exemple en dimension 2 :
Soit $X_1 \sim \mathcal{N}(0, \sigma_1)$ et $X_2 \sim \mathcal{N}(0, \sigma_2)$ t.q. corr $(X_1, X_2) = \rho$. Alors

$$\begin{cases} X_1 & = \sigma_1 G_1, \\ X_2 & = \rho \sigma_2 G_1 + \sqrt{1-\rho^2}\sigma_2 G_2, \end{cases}$$

où $G = (G_1, G_2) \sim \mathcal{N}(0, \mathrm{Id}_2)$.

Rappel sur Cholesky

$$
\begin{pmatrix}
a_{11} & a_{12} & \dots & a_{1d} \\
a_{21} & a_{22} & \dots & a_{2d} \\
\vdots & \vdots & \vdots & \vdots \\
a_{d1} & a_{d2} & \dots & a_{dd}
\end{pmatrix}
=
\begin{pmatrix}
l_{11} & 0 & \dots & 0 \\
l_{21} & l_{22} & \dots & 0 \\
\vdots & \vdots & \vdots & \vdots \\
l_{d1} & l_{d2} & \dots & l_{dd}
\end{pmatrix}
\begin{pmatrix}
l_{11} & l_{21} & \dots & l_{d1} \\
0 & l_{22} & \dots & l_{d2} \\
\vdots & \vdots & \vdots & \vdots \\
0 & 0 & \dots & l_{dd}
\end{pmatrix}
$$

où

$$
l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}, \quad i = 1, \dots, d
$$

$$
l_{ji} = \frac{a_{ji} - \sum_{k=1}^{i-1} l_{jk} l_{ik}}{l_{ii}}, \quad j = i+1, \dots, d.
$$

Rq : Complexité $\mathcal{O}(d^2)$.

Qu'est ce que uniform_real_distribution en C++11

```
template<typename RealType = double>
class uniform_real_distribution
{
    static_assert(std::is_floating_point<RealType>::value,
                  "template argument not a floating point type");
public:
    typedef RealType result_type;
    struct param_type
    {
        typedef uniform_real_distribution<RealType> distribution_type;
        explicit param_type(RealType a = RealType(0),RealType b = RealType(1))
            : _a(a), _b(b) { }
        result_type a() const { return _a; }
        result_type b() const { return _b; }
        friend bool operator==(const param_type& p1, const param_type& p2)
        { return p1._a == p2._a && p1._b == p2._b; }
    private:
        RealType _a;
        RealType _b;
    };
    // suite sur le prochain slide...
```

Qu'est ce que `uniform_real_distribution` en C++11

```
public:
    explicit uniform_real_distribution(RealType a = RealType(0), RealType b = RealType(1))
        : _param(a, b) { }
    explicit uniform_real_distribution(const param_type& p)
        : _param(p) { }
    void reset() { }
    result_type a() const { return _param.a(); }
    result_type b() const { return _param.b(); }
    param_type param() const { return _param; }
    void param(const param_type& param) { _param = param; }
    result_type min() const { return this->a(); }
    result_type max() const { return this->b(); }

    template<typename Generator>
    result_type operator()(Generator& gen) { return this->operator()(gen, _param); }

    template<typename Generator>
    result_type operator()(Generator& gen, const param_type& p) {
    result_type u = // en fait c'est presque gen() sauf que...
        std::generate_canonical<result_type>(gen);
        return (u * (p.b() - p.a()) + p.a());
    }

    friend bool
    operator==(const uniform_real_distribution& d1, const uniform_real_distribution& d2)
    { return d1._param == d2._param; }
private:
    param_type _param;
};
```

Exemple d'appel

- 3 types de générateurs d'entiers pseudo-aléatoires (classes génériques)
  - `linear_congruential_engine`
  - `mersenne_twister_engine`
  - `subtract_with_carry_engine`

  et des instances particulières comme `minstd_rand`, `mt19937` (mersenne twiser 32 bits) et `mt19937_64` (version 64 bits).

```cpp
#include <random>
#include <iostream>

int main() {
auto seed = std::chrono::system_clock::now().time_since_epoch().count();
    std::mt19937 gen(seed);
    std::uniform_real_distribution<> U(-1, 1);
    for (int n = 0; n < 10; ++n) {
        std::cout << U(gen) << std::endl;
    }
}
```

Exemple de composition entre "distribution" et "engine"

Première version :

```
template <typename Distribution, typename Generator>
class random_variable {
public:
    typedef typename Distribution::result_type result_type;
    random_variable(Distribution const & X, Generator const & gen)
        : _X(X), _gen(gen), _value(0) {}
    result_type operator()() { return _value = _X(_gen); }
    result_type value() const { return _value; }
private:
    Distribution _X;
    Generator _gen;
    result_type _value;
};
```

Fonction générique "chapeau" qui permet de trouver (implicitement par le compilateur) les types `Distribution` et `Generator` à partir des arguments X et g.

```
template <typename Distribution, typename Generator>
random_variable<Distribution, Generator>
make_random_variable(Distribution const & X, Generator const & g) {
    return random_variable<Distribution, Generator>(X, g);
};
```

Exemple de composition entre "distribution" et "engine"

```
int main()
{
    auto seed = std::chrono::system_clock::now().time_since_epoch().count();
    std::mt19937_64 mt64bits(seed);
    std::uniform_real_distribution<double> Udistrib(0,1);
    std::uniform_real_distribution<double> Vdistrib(-1,1);

    auto U = make_random_variable(Udistrib, mt64bits);
    auto V = make_random_variable(Vdistrib, mt64bits);
    double sum = 0;
    uint n = 1e6;
    for (uint i = 0; i < n; ++i)
        sum += U() * V();
    sum /= n;
    std::cout << sum << std::endl;
    return 0;
}
```

Résultat de ce programme : `0.166467`

Problème ?

Exemple de composition entre "distribution" et "engine"

Seconde version :

```
template <typename Distribution, typename Generator>
class random_variable {
public:
    typedef typename Distribution::result_type result_type;
    random_variable(Distribution const & X, Generator & gen)
        : _X(X), _gen(gen), _value(0) {}
    result_type operator()() { return _value = _X(_gen); }
    result_type value() const { return _value; }
private:
    Distribution _X;
Generator & _gen;
    result_type _value;
};

template <typename Distribution, typename Generator>
random_variable<Distribution, Generator>
make_random_variable(Distribution const & X, Generator & g) {
    return random_variable<Distribution, Generator>(X, g);
};
```

Résultat du programme précédent : `0.00028241`

Fonction générique `std::bind`

Dans le header `functional` il existe une fonction générique `srd::bind` qui permet de fixer les arguments d'une fonction ou d'un objet fonctionnel. Par exemple l'appel

```
auto U = std::bind(Udistrib, mt64bits);
```

Crée un objet fonctionnel `U` qui ne prend pas d'arguments et dont un appel `U()` correspond à `Udistrib(mt64bits)`.

Voici le propotype de la fonction générique `std::bind`

```
template< class F, class... Args >
/*unspecified*/ bind( F&& f, Args&&... args );
```

Deux nouveautés au coeur du `C++11` :

- && référence sur une `rvalue`
- **`class`** `... Args` les `variadic` **`template`** qui permettent un nombre générique de types génériques

Pour l'instant on laisse ça de côté et on utilise `std::bind`

Exemples d'utilistation de `std::bind`

```cpp
#include <iostream>
#include <functional>
void f(int a, int b, double c) {
    std::cout << "f(" << a << "," << b << "," << c << ")\n";
};
int main() {
    auto g = std::bind(f, 2, 3, 4);
    g();

    using namespace std::placeholders;
    auto h = std::bind(f, _1, 2, 3);
    h(10);

    using namespace std::placeholders;
    auto k = std::bind(f, _2, 2, _1);
    k(3.5, 8);
}
```

Le namespace `std::placeholders` contient les objets _1, _2, ... qui permettent de récupérer des arguments lors de l'appel fonctionnel. Le numéro _n correspond à l'ordre d'appel du nouvel argument.

Résultat du programme ?

Remplacer make_var_alea par std::bind

- la première version (mauvaise...) correspond à

```cpp
auto U = std::bind(Udistrib, mt64bits);
```

- un appel équivalent à la seconde version nécessite d'utiliser la fonction générique `std::ref` pour encapsuler l'objet `mt64bits` dans un objet `std::reference_wrapper` qui se comporte comme une référence

```cpp
auto U = std::bind(Udistrib, std::ref(mt64bits));
```

  Il existe aussi la fonction générique `std::cref` qui permet d'encapsuler un objet dans une (pseudo) référence constante.