Simulation (exacte) de processus aléatoires

Exemples en C++

Vincent Lemaire
vincent.lemaire@upmc.fr

Rappels sur la programmation générique

Mouvement Brownien et dérivés...

- Mouvement Brownien
- Pont Brownien et Brownien biasé
- Black-Scholes

Ornstein-Uhlenbeck

- CIR

Processus de Poisson

Processus de Lévy

- Brownien + Poisson composé
- Brownien subordonné

Processus gaussien stationnaire

- One dimensional SDE

Rappels sur la programmation générique

Evolution du C au C++ :
- POO (objets, héritages, polymorphisme dynamique)
- Prog. générique (template, polymorphisme statique)
- Nouvelle librairie standard : la STL (Standard Template Library).

Pourquoi utiliser la STL ?

- C'est standard...
- Structures de données pratiques : **conteneurs**
- Algorithmes préprogrammés : **algorithmes** (tri, recherche...)
- Nouveau concept (pour ne plus utiliser de pointeurs) : **itérateurs**
- Existe des versions optimisées / parallélisées
- Utilisée par de nombreuses bibliothèques

Le futur ? Surement B00ST qui reprend et développe de nombreux concepts de la STL.

`typedef`, `typename`, `template`

- **`typedef`** : existe aussi en C, permet de définir un nouveau type à partir d'un autre type, en gros : copier/coller à la compilation.
  Permet d'écrire un code plus court, plus lisible ! **A utiliser !**
  Syntaxe : **`typedef`** *ancientypelongetcomplique* *NouveauType*

`typedef`, `typename`, `template`

- **`typedef`** : existe aussi en C, permet de définir un nouveau type à partir d'un autre type, en gros : copier/coller à la compilation.
  Permet d'écrire un code plus court, plus lisible ! **A utiliser !**
  Syntaxe : **`typedef`** *ancientypelongetcomplique* *NouveauType*

- **`typename`** : mot-clé indiquant que le « terme » suivant est un type.
  Deux usages :
  - Lorsqu'il y a un litige possible sur le « terme » qui suit.
    Exemple : dans la classe abstraite `processus`.
  - Dans l'argument d'un **`template`** pour définir le ou les noms des types génériques (peut-être remplacé par le mot-clé **`class`**).
    Exemple : `template <typename T1, typename T2 = T1>`

`typedef`, `typename`, `template`

- **`typedef`** : existe aussi en C, permet de définir un nouveau type à partir d'un autre type, en gros : copier/coller à la compilation.
  Permet d'écrire un code plus court, plus lisible ! **A utiliser !**
  Syntaxe : **`typedef`** *ancientypelongetcomplique* *NouveauType*

- **`typename`** : mot-clé indiquant que le « terme » suivant est un type.
  Deux usages :
  - Lorsqu'il y a un litige possible sur le « terme » qui suit.
    Exemple : dans la classe abstraite `processus`.
  - Dans l'argument d'un **`template`** pour définir le ou les noms des types génériques (peut-être remplacé par le mot-clé **`class`**).
    Exemple : `template <typename T1, typename T2 = T1>`

- **`template`** : mot-clé permettant de définir une fonction ou une classe *générique* i.e. qui ne dépend pas du type de certains éléments.
  A la compilation, autant de versions que nécessaires sont codées par le compilateur.
  Syntaxe : précède la déclaration **et** la définition de la fonction ou de la classe.

Classe abstraite : processus

```
template <typename T>
struct processus
{
    typedef std::pair<double, T> state;
    typedef std::list<state> result_type;
    typedef typename result_type::iterator iter;
    typedef typename result_type::const_iterator cst_iter;
    processus(int size = 0) : value(size) {};
    virtual result_type operator()() = 0;
    result_type current() const { return value; };
    template <typename S>
    friend std::ostream& operator<<(std::ostream &o,
                                    const processus<S> &p);
    protected:
        result_type value;
};
```

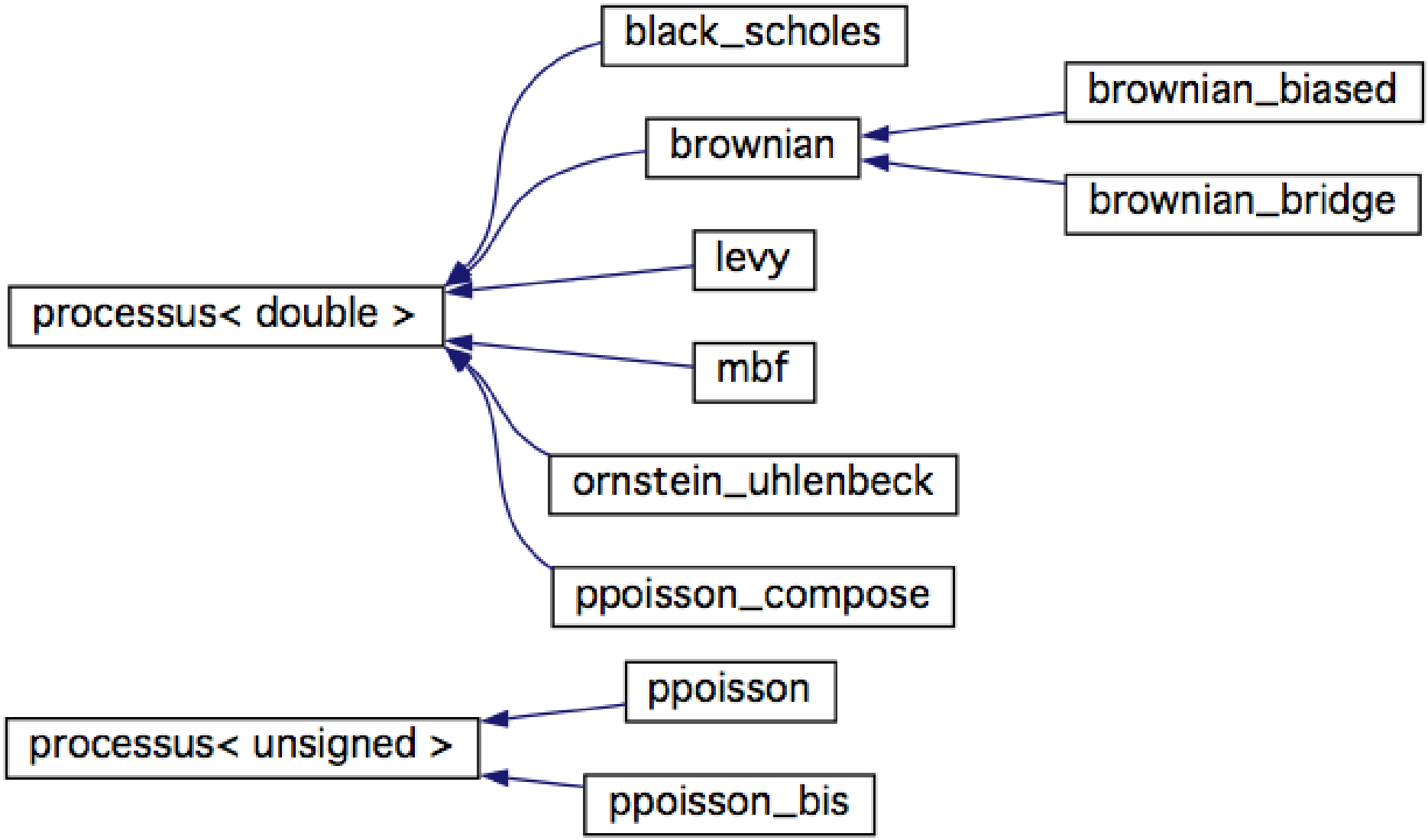
Classe abstraite : processus - 2 -

- Utilise les conteneurs `pair` et `list` de la STL.
- Surcharge l'opérateur `()` par une fonction *virtuelle pure* : il ne peut pas exister d'objet de cette classe, c'est une classe abstraite.
- Surcharge de l'opérateur << (injection) par une *fonction amie*, voici une définition possible de cette fonction :

```
template <typename T>
std::ostream& operator<<(std::ostream &o, const processus<T> &p)
{
    typename processus<T>::cst_iter i;
    for(i = p.value.begin(); i != p.value.end(); ++i)
        o << (*i).first << "\t" << (*i).second << std::endl;
    return o;
}
```

- *headers* nécessaires : `iostream`, `list`, `utility`.
- Attention aux **const**... On ne peut pas utiliser un iterateur non constant car l'objet p est passé (par référence) comme étant constant (non modifié par la fonction).

Hierarchie des classes à venir



Mouvement Brownien

- Approximation (exacte aux instants de discrétisation) par une marche aléatoire (implémentée dans **`operator()`**) :
  Par définition, on a

$$\forall 0 \leqslant s < t, \quad B_t = B_s + \sqrt{t-s}G,$$

  où $G \sim \mathcal{N}(0;1)$.

- Construction récursive (implémentée dans la méthode `affine`) :

$$\forall h > 0, t \geqslant 0, \quad \mathcal{L}\left(B_t | B_{t-h} = a, B_{t+h} = b\right) \sim \mathcal{N}\left(\frac{a+b}{2}; \frac{h}{2}\right).$$

  (construction de Paul Lévy du mouvement Brownien).

Classe brownian

```
struct brownian : public processus<double>
{
    brownian(int n, double T=1)
        : processus<double>(pow(2,n)+1), n(n), T(T),
          h(T/pow(2., n)), G(0,sqrt(h)) {};
    result_type operator()();
    result_type affine();
    friend struct black_scholes;
    protected:
        int n;
        double h, T;
        gaussian G;
};
```

Classe brownian, définition de l'opérateur ()

```
result_type brownian::operator()() {
    value.clear();
    state val_k(0,0);
    value.push_back(val_k);
    do {
        val_k.first += h;
        val_k.second += G();
        value.push_back(val_k);
    } while (val_k.first < T);
    return value;
};
```

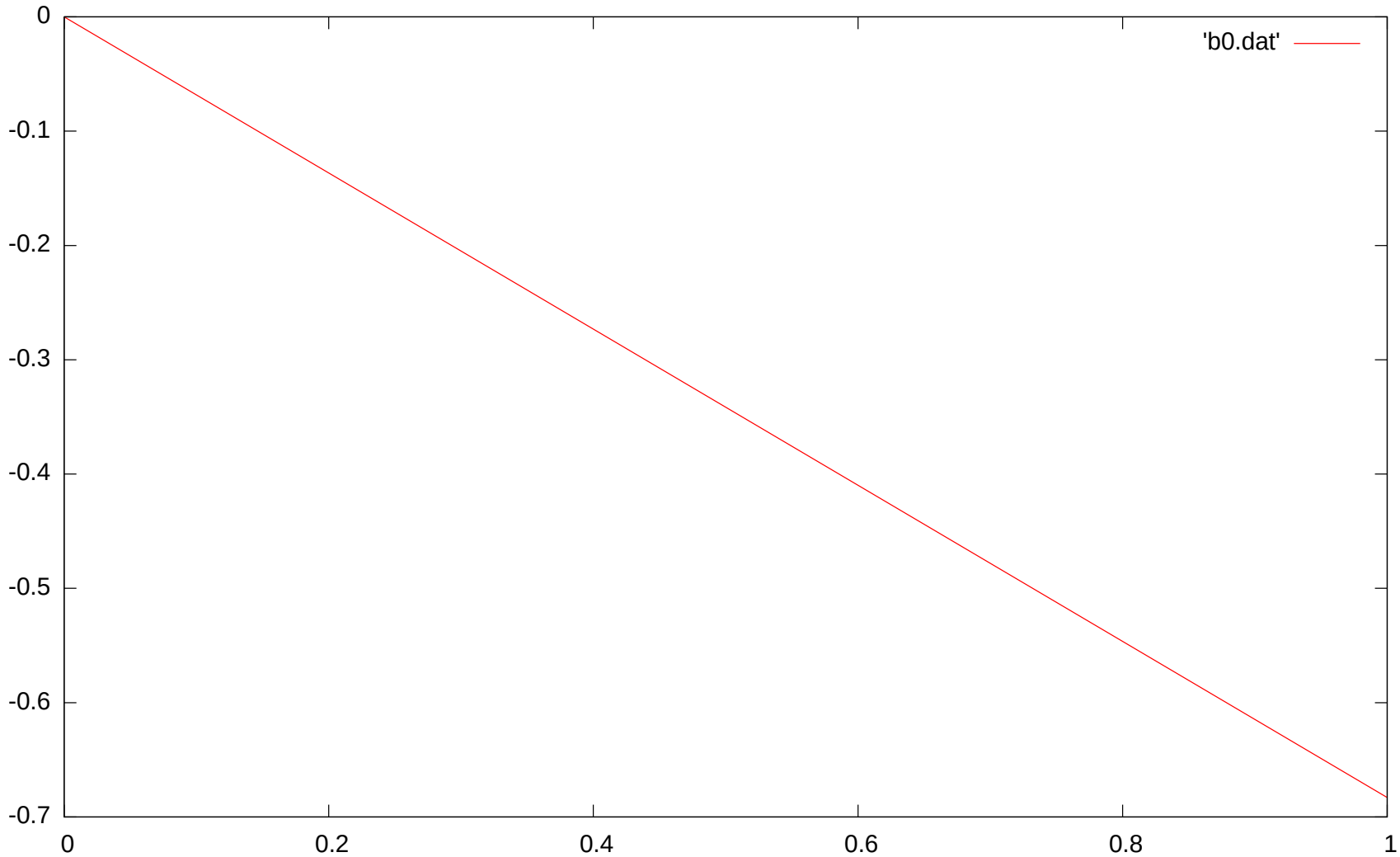
Exercice : Ecriture en 5 lignes sans utiliser la variable temporaire val_k.

Classe brownian, définition de la méthode affine

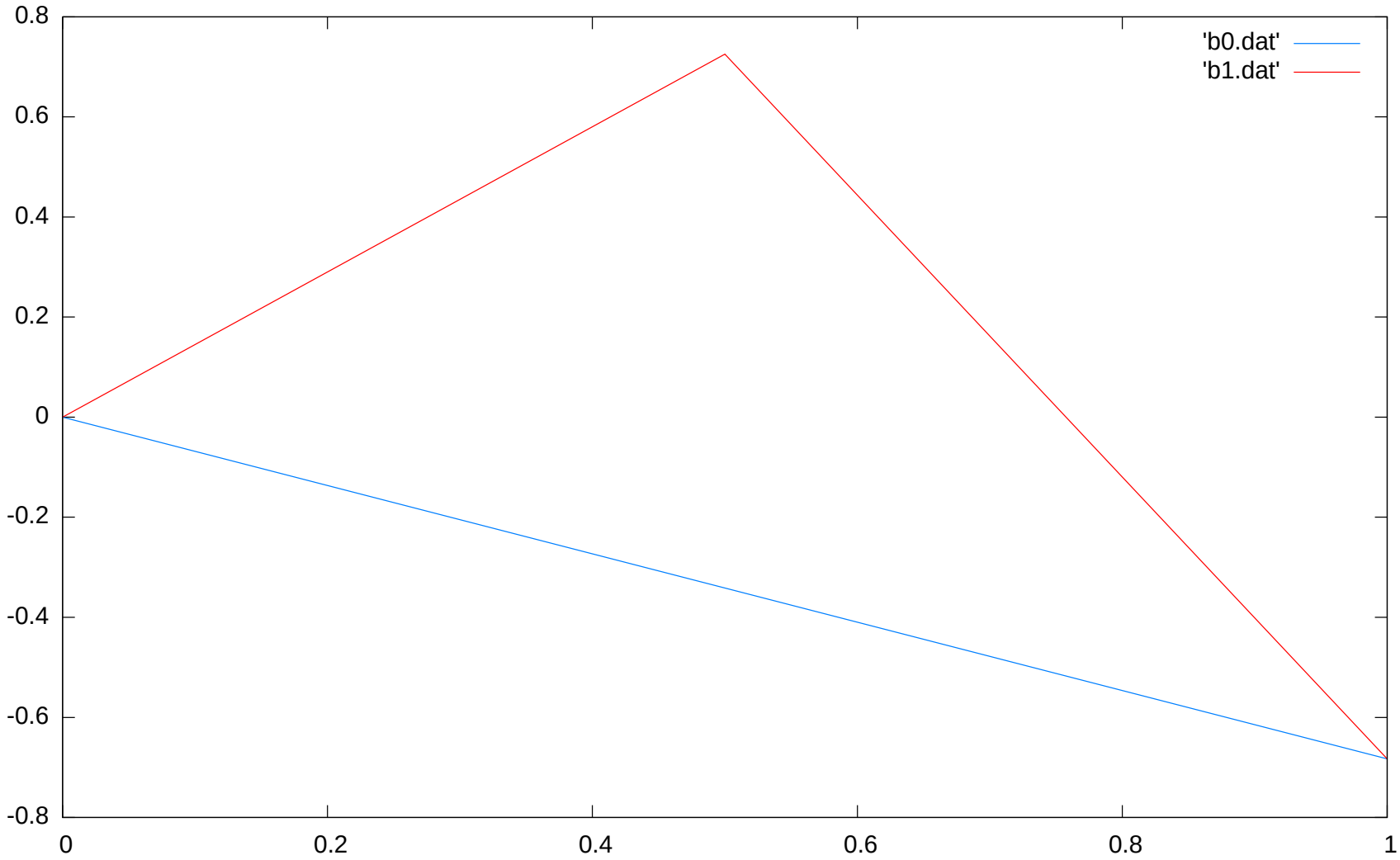
```
result_type brownian::affine() {
    n++; h *= 0.5;
    G = gaussian(0, sqrt(0.5*h));
    iter precedent = value.begin(), current = ++value.begin();
    while (current != value.end()) {
        value.insert(current,
          state(0.5*((*current).first+(*precedent).first),
            0.5*((*current).second+(*precedent).second)+G()));
        precedent = current;
        current++;
    }
    return value;
};
```

Remarque : Intérêt d'avoir choisi le conteneur `list` pour stocker la trajectoire `value`.

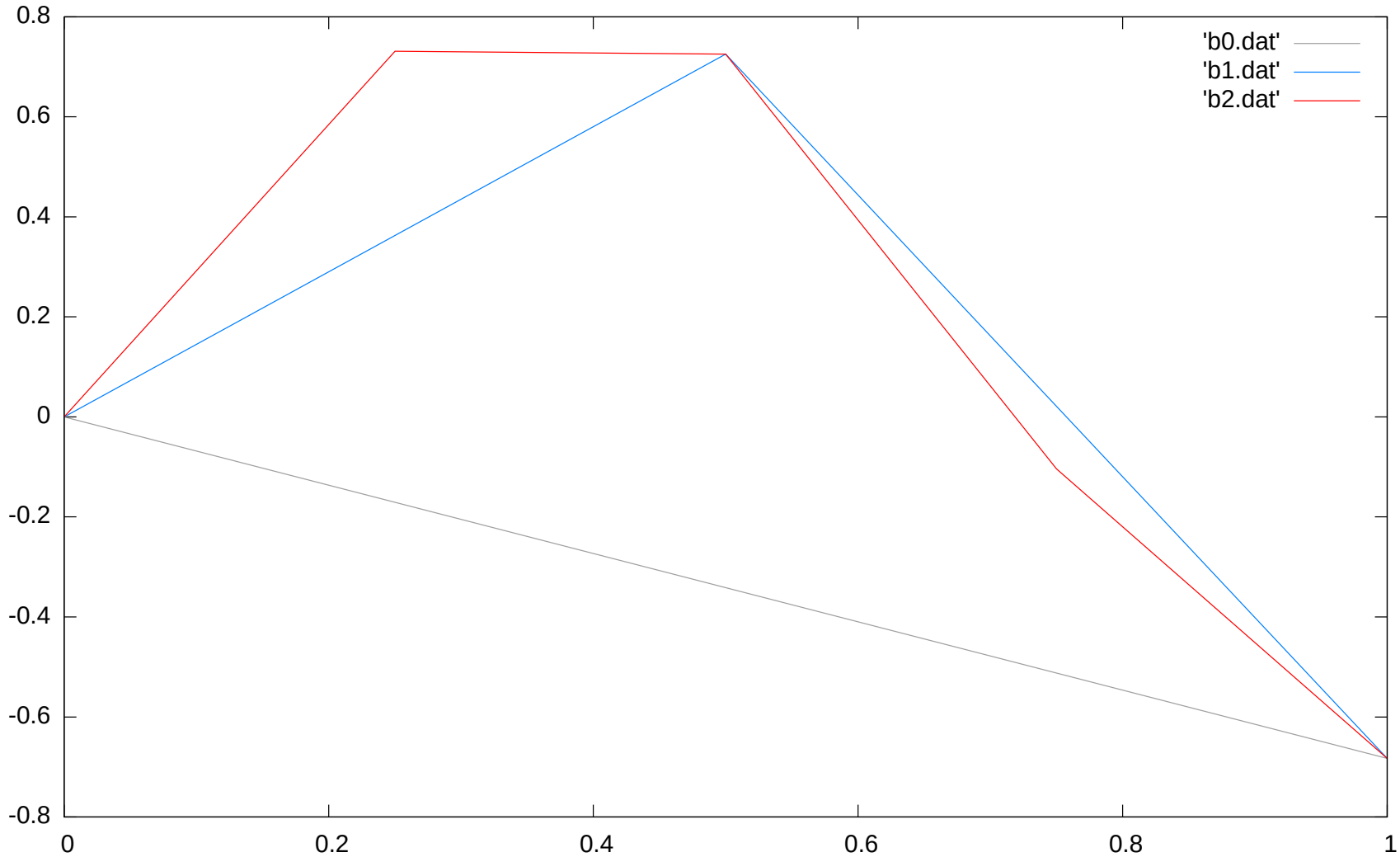
Construction récursive (trajectorielle)



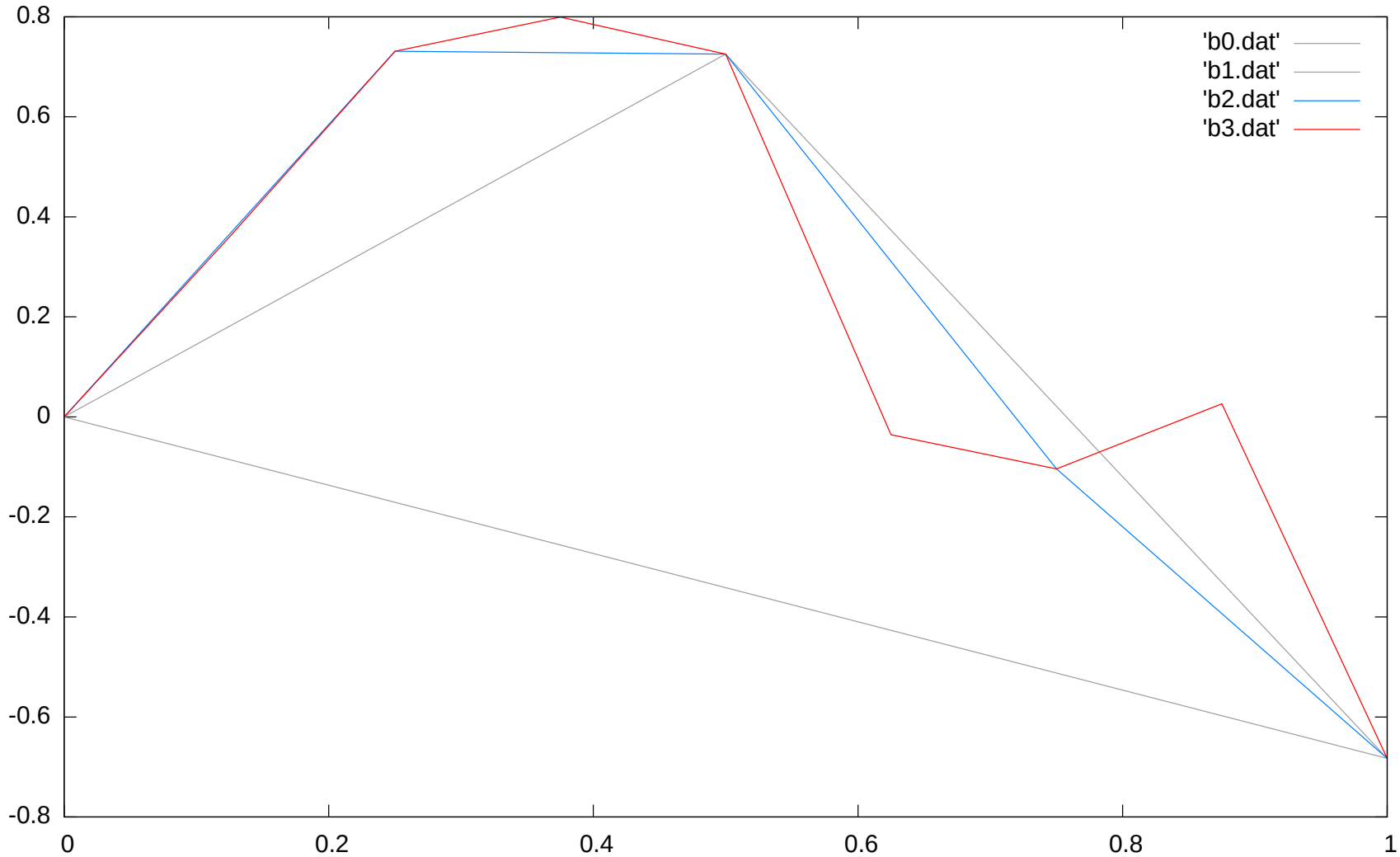
Construction récursive (trajectorielle)



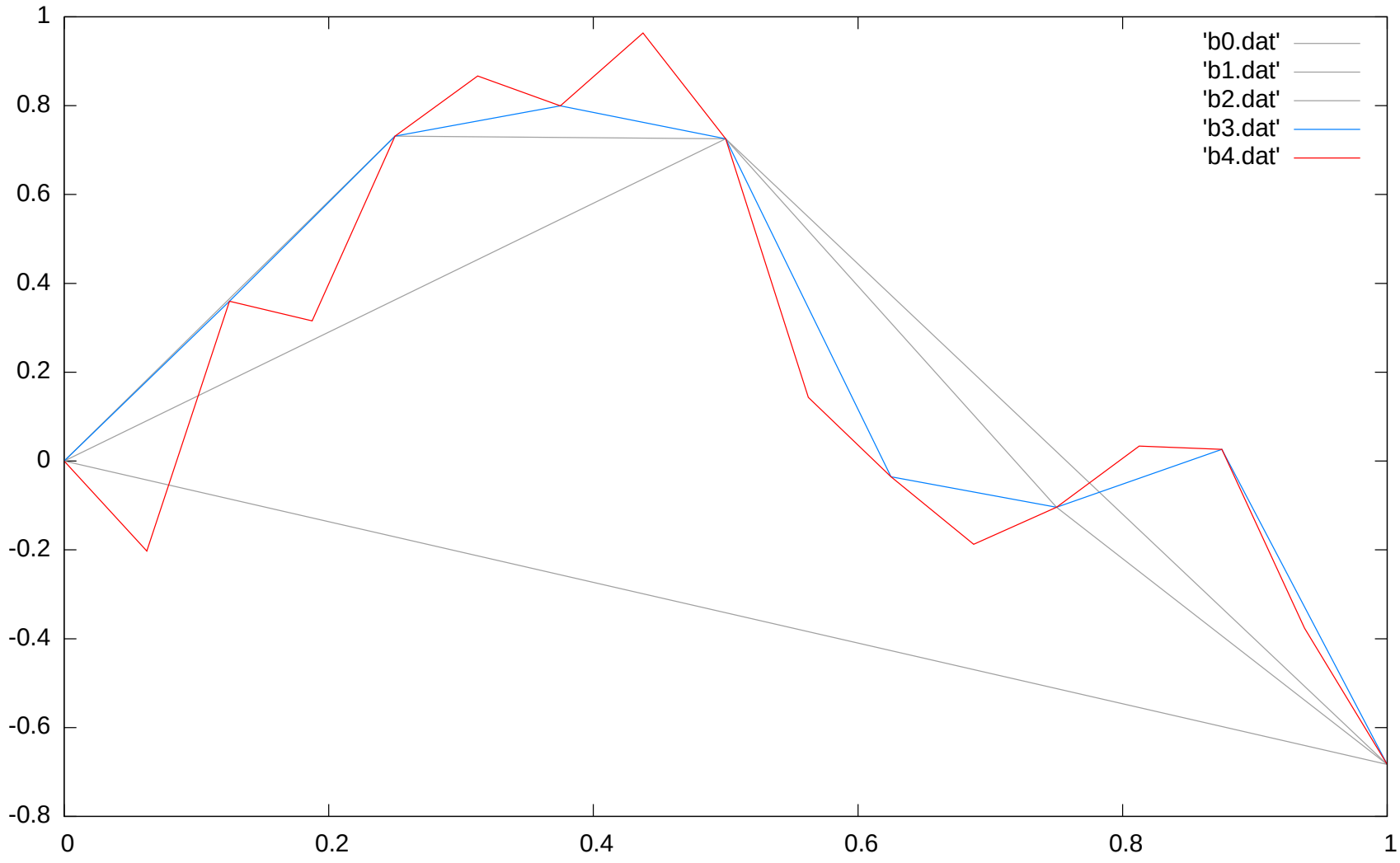
Construction récursive (trajectorielle)



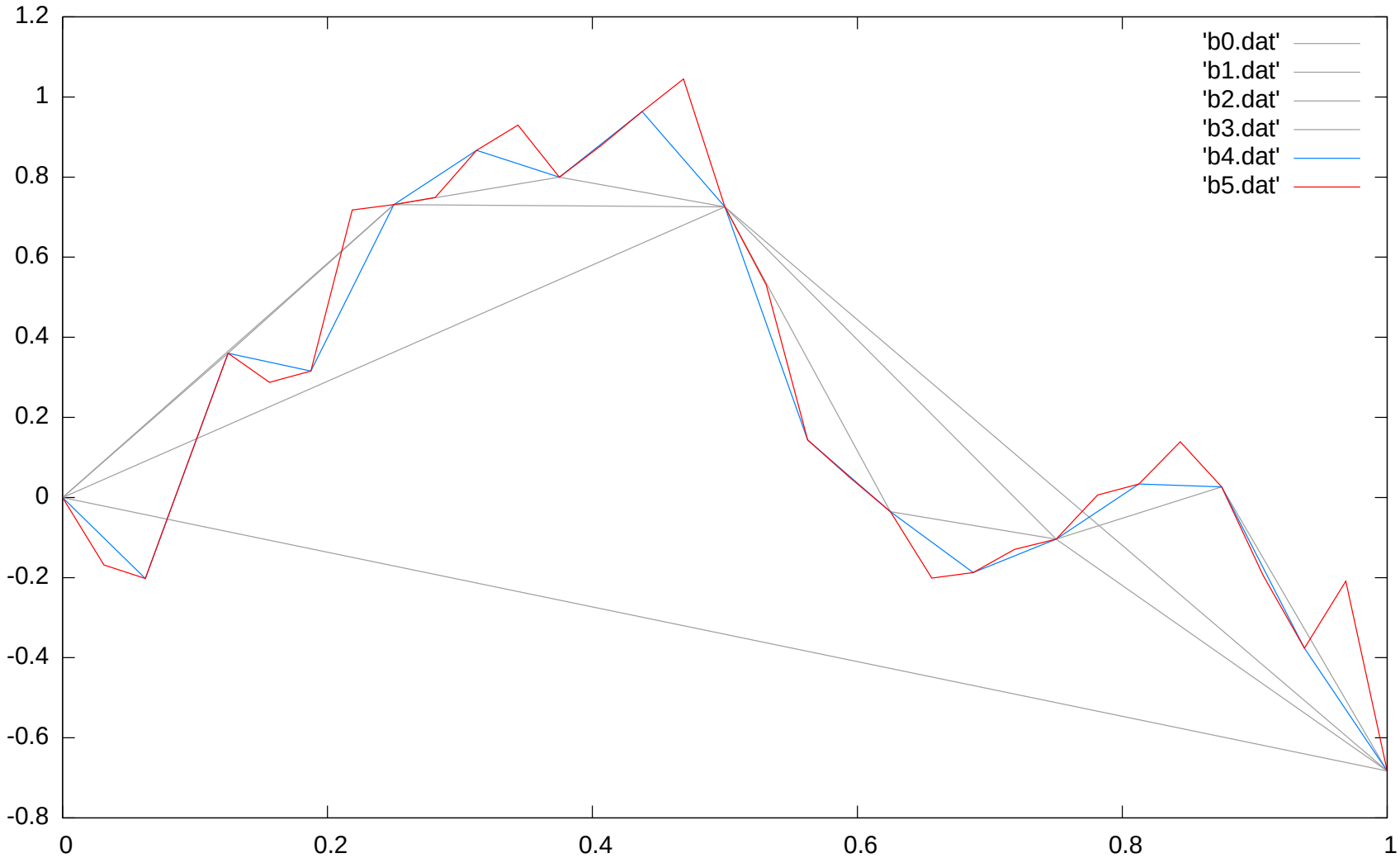
Construction récursive (trajectorielle)



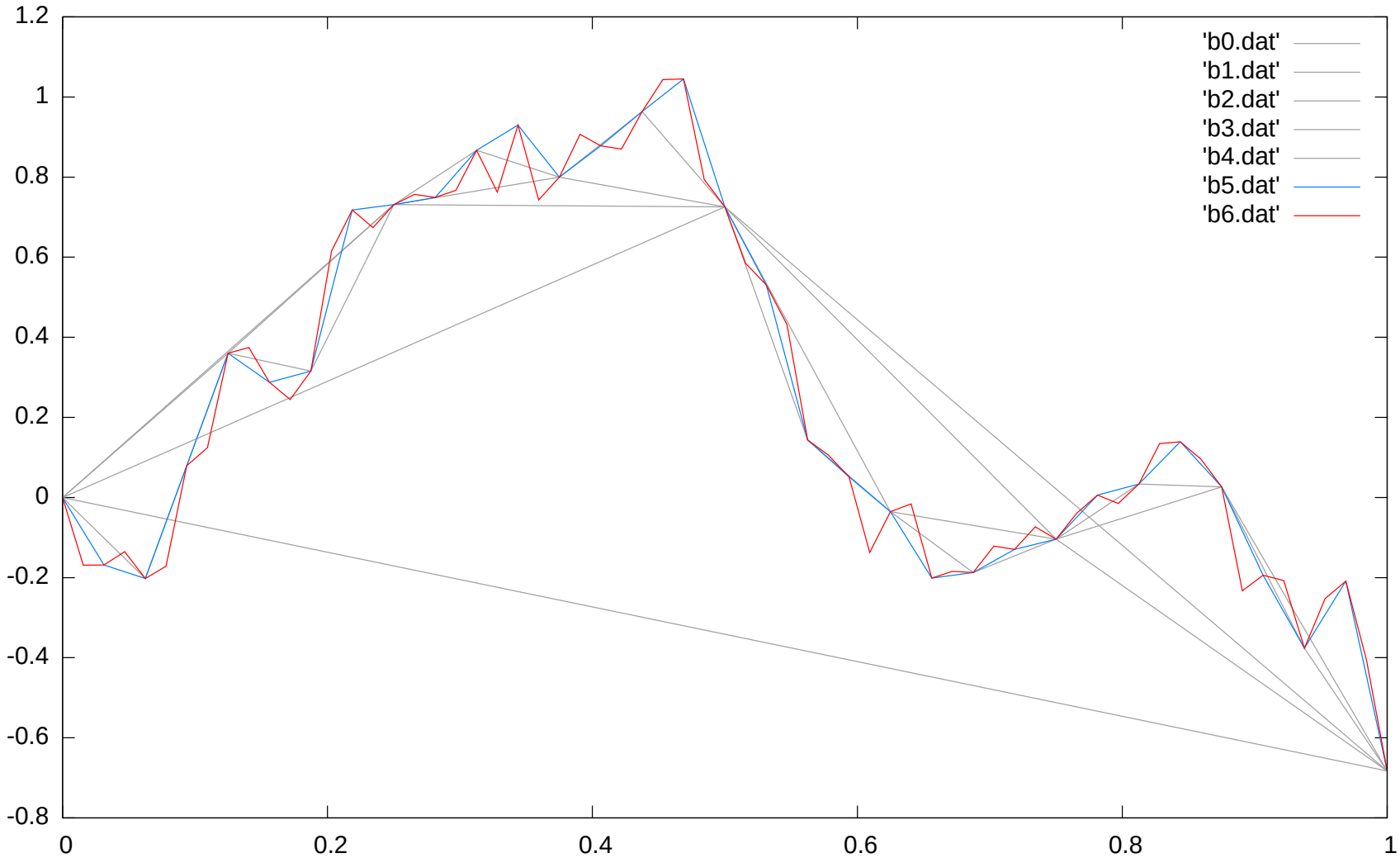
Construction récursive (trajectorielle)



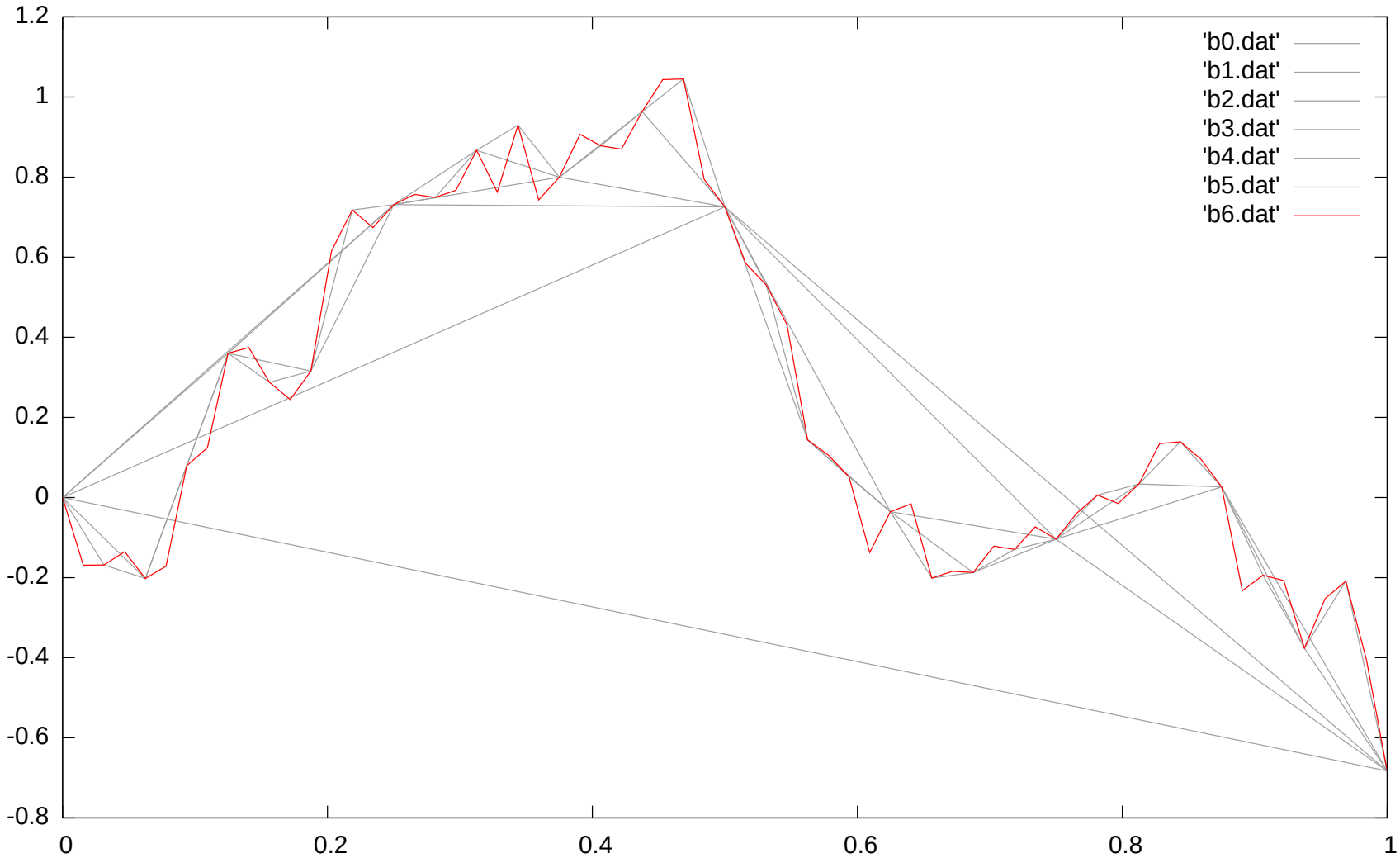
Construction récursive (trajectorielle)



Construction récursive (trajectorielle)



Construction récursive (trajectorielle)

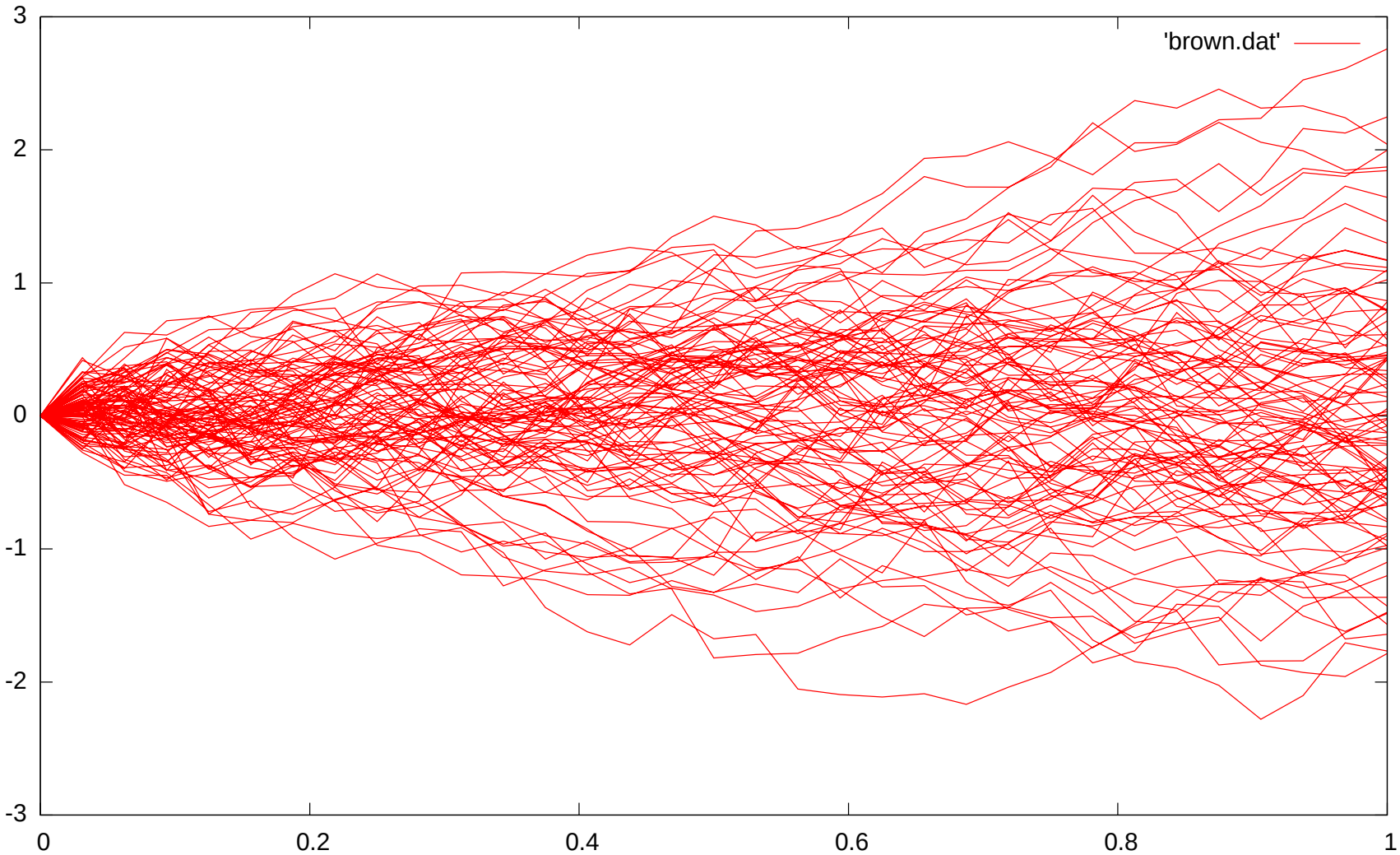


Exemple d'un programme de test

```
#include <iostream>
#include <fstream>
#include "process.hpp"
using namespace std;

template <typename T>
void affiche(processus<T> &p, char name[]) {
    ofstream file;
    file.open(name);
    for (int k = 0; k < 100; k++) {
        p();
        file << p << endl;
    }
    file.close();
};
int main() {
    init_alea();
    brownian B(5);
    affiche(B, "brown.dat");
    return 0;
}
```

100 trajectoires Browniennes



Pont Brownien et Brownien biasé

- Pont Brownien : on conditionne un Brownien sur $[0,T]$ à l'évènement $\{B_T = y\}$ où y est un point fixé de $\mathbf{R}$,

$$\mathcal{L}\left((X_t^y)_{t\in[0,T]}\right) \sim \mathcal{L}\left((B_t)_{t\in[0,T]} | B_T = y\right).$$

  Simulation ? Presque déjà faite...

- Brownien biasé : on conditionne un Brownien sur $[0,T]$ à l'évènement $\{B_T \sim \mu\}$ où μ est une mesure de probabilité sur $\mathbf{R}$,

$$\mathcal{L}\left((X_t^\mu)_{t\in[0,T]}\right) \sim \mathcal{L}\left((B_t)_{t\in[0,T]} | B_T \sim \mu\right).$$

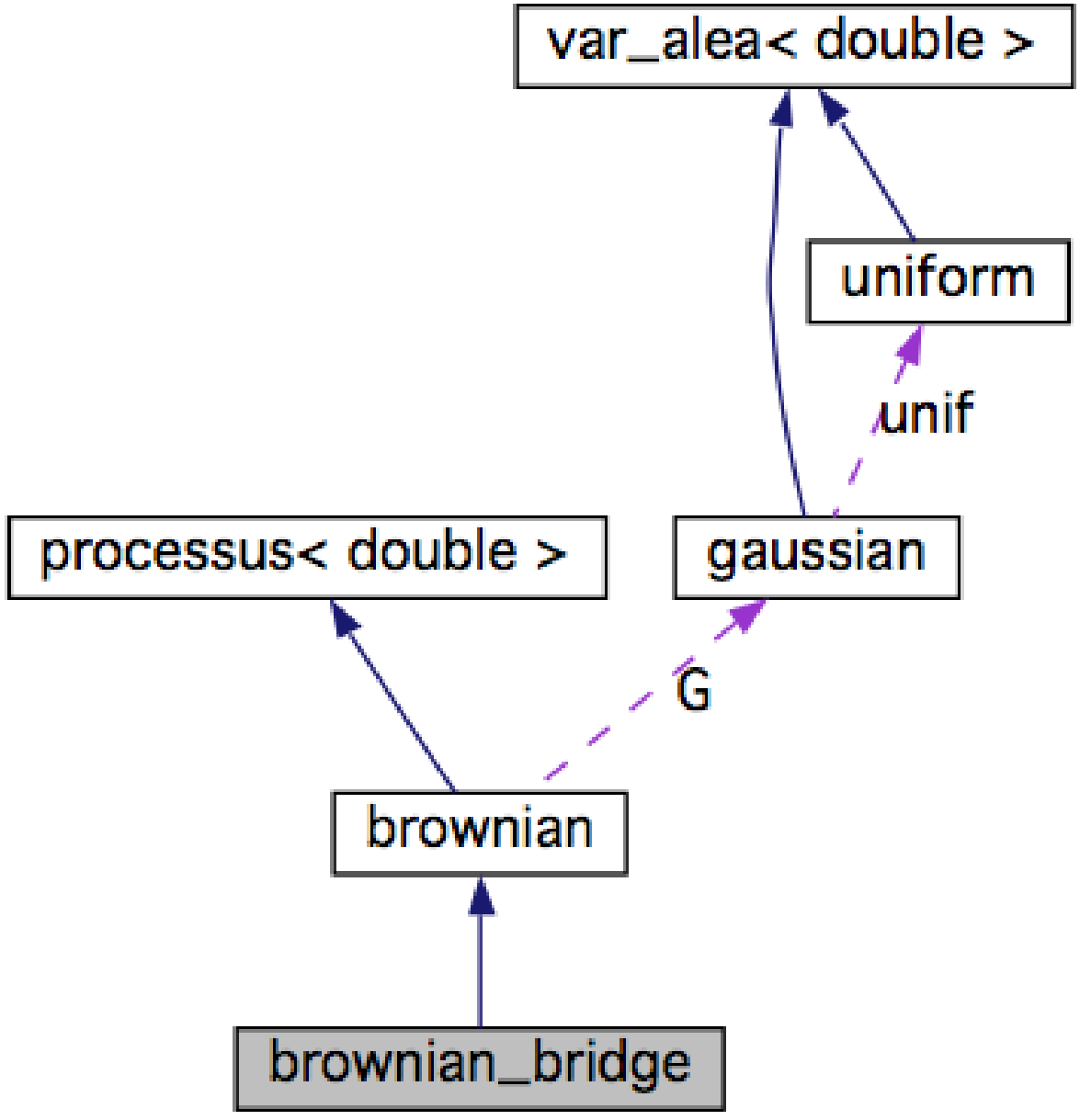
  Utilisé récemment pour la simulation exacte de diffusions.

Remarque : On pourrait « dériver » le pont Brownien du Brownien biaisé en considérant la loi $\mu(\mathrm{d}x) = \delta_y(\mathrm{d}x)$.
Dans la suite on écrit 2 classes distinctes.

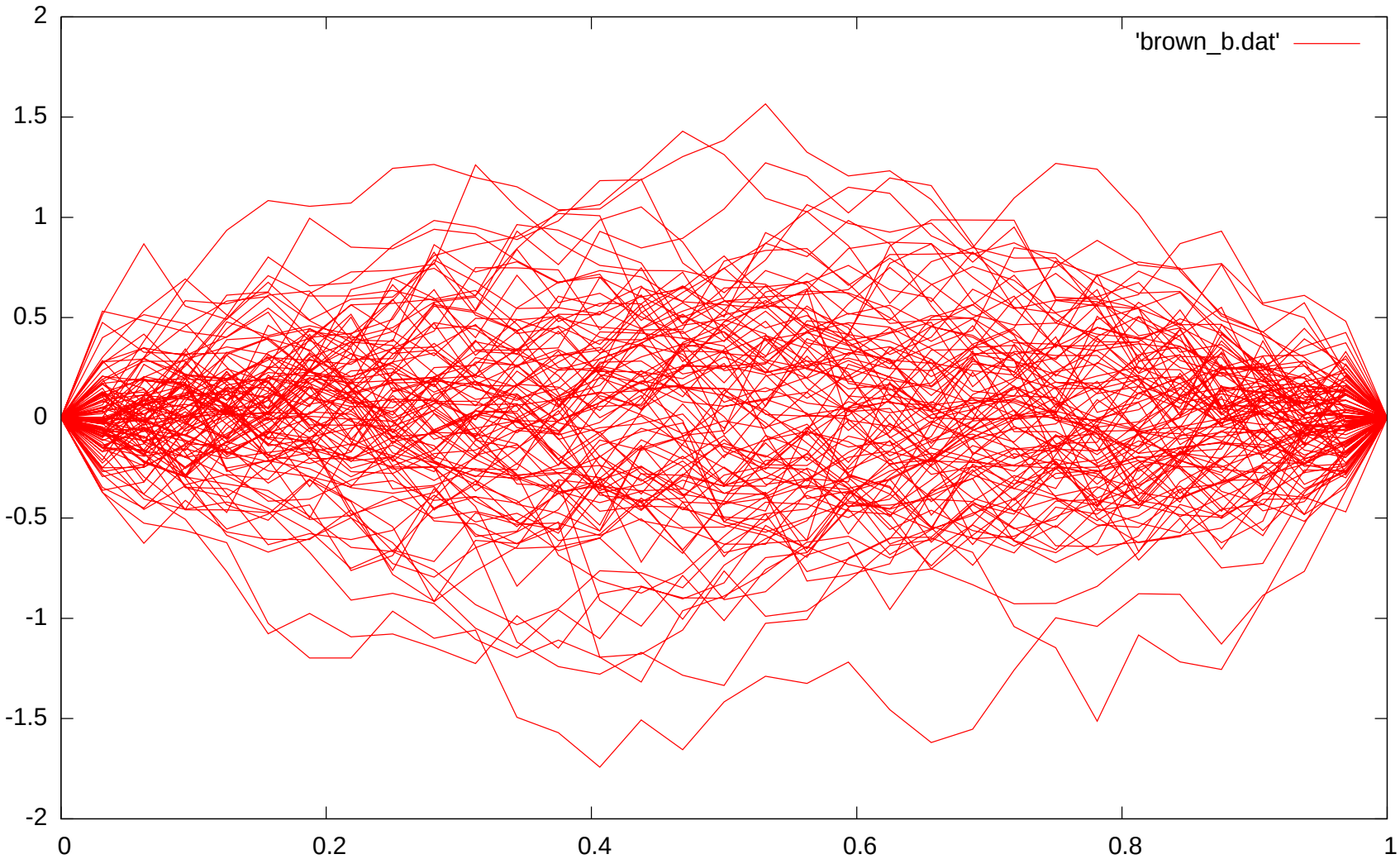
Classe brownian_bridge

```
struct brownian_bridge : public brownian
{
    brownian_bridge(int n, double T = 1, double B_T = 0)
        : brownian(n, T), B_T(B_T) {};
    result_type operator()() {
        value.clear();
        value.push_back(state(0,0));
        value.push_back(state(T, B_T));
        int n_tmp = n; n = 0; h = T;
        for (int j = 0; j < n_tmp; j++) affine();
        n = n_tmp;
        return value;
    };
    private:
        double B_T;
};
```

Arbre d'héritage de la classe



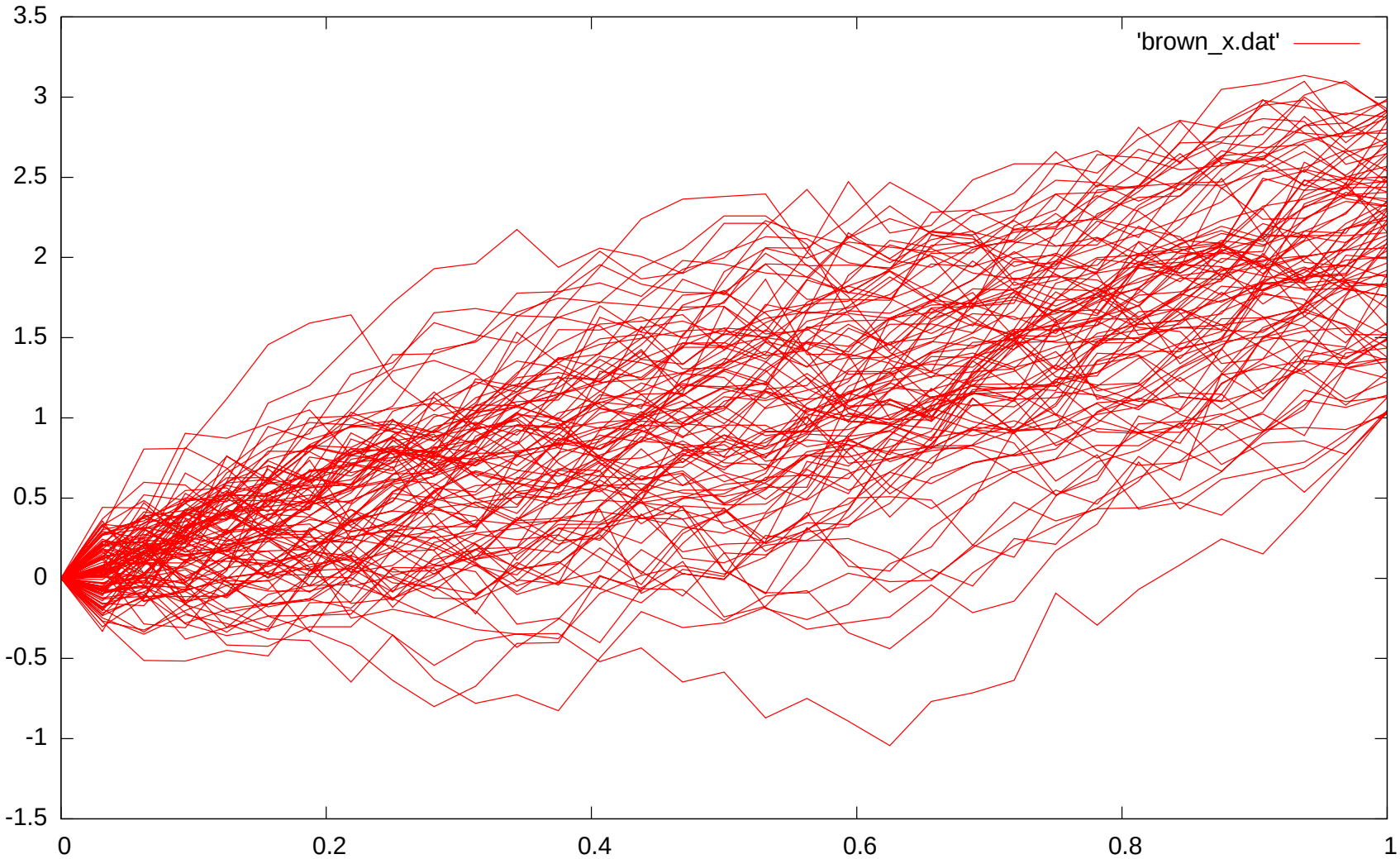
100 trajectoires du pont Brownien



Classe brownian_biased

```
struct brownian_biased : public brownian
{
    brownian_biased(int n, double T = 1, var_alea<double> &X)
        : brownian(n, T), X(X) {};
    result_type operator()() {
        value.clear();
        value.push_back(state(0,0));
        value.push_back(state(T, X()));
        int n_tmp = n; n = 0; h = T;
        for (int j = 0; j < n_tmp; j++) affine();
        n = n_tmp;
        return value;
    };
    private:
        var_alea<double> &X;
};
```

100 trajectoires du Brownien biasé



Black-Scholes (dimension 1)

Soit $\left(X_t\right)_{t\geqslant 0}$ solution de

$$\mathrm{d}X_t = rX_t\mathrm{d}t + \sigma X_t \mathrm{d}W_t, \quad X_0 = x_0 \in \mathbf{R}^+,$$

alors pour tout $t \geqslant 0$, on a

$$X_t = x_0 e^{\left(r-\frac{\sigma^2}{2}\right)t+\sigma W_t}.$$

- X est un Brownien géométrique (transformation exponentielle d'un Brownien)
- Définition de la classe `black_scholes` :
  - par une classe dérivée
  - par une classe amie
- Avantages, inconvénients ?

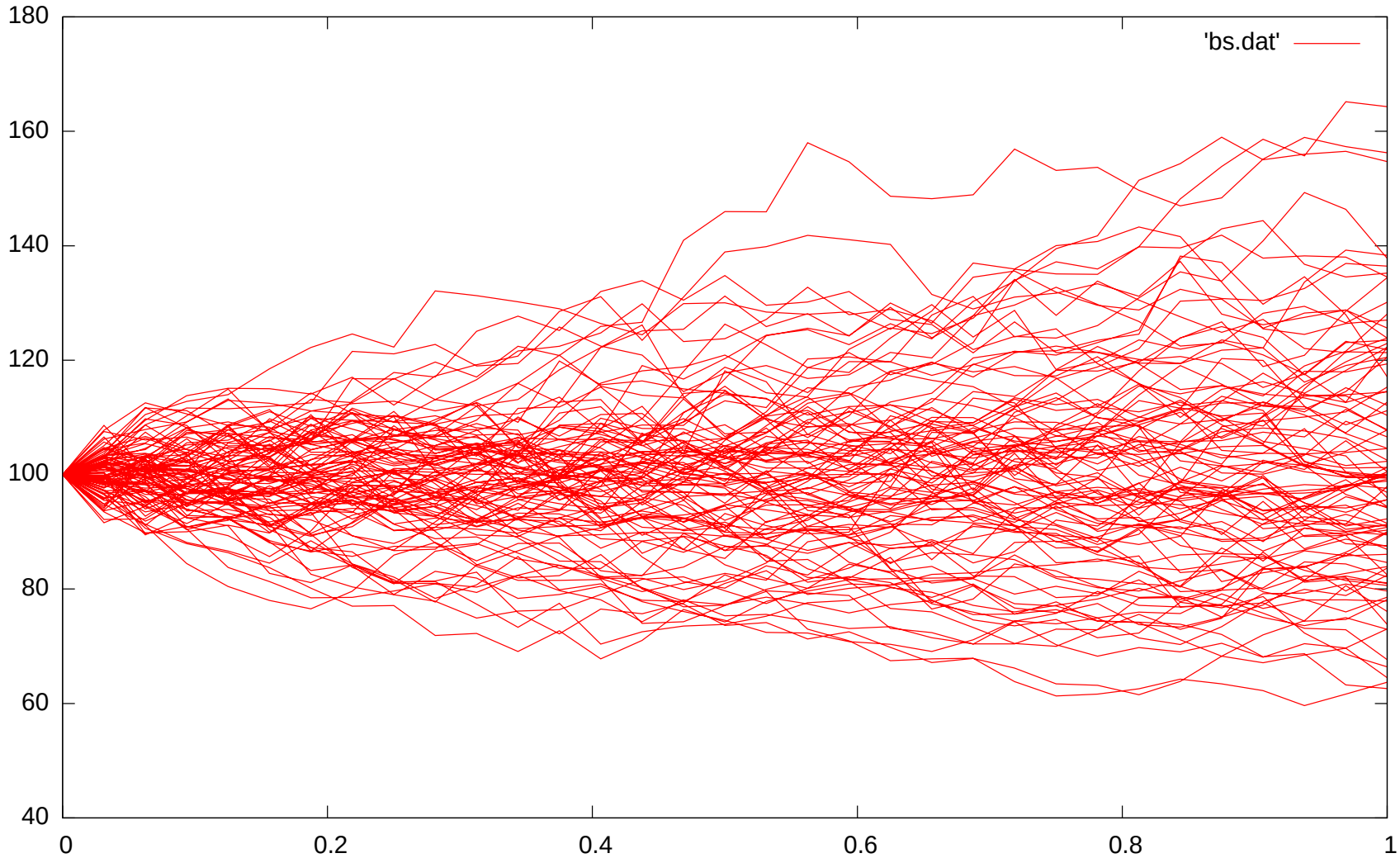
Black-Scholes version classe dérivée

```
struct black_scholes : public brownian
{
    black_scholes(int n, double x0, double r, double s, double T=1)
        : brownian(n, T), bs(x0, r, s) {};
    result_type operator()() {
      brownian::operator()();
      std::transform(value.begin(), value.end(), value.begin(), bs);
      return value;
    };
    private:
      struct fun_bs : public std::unary_function<state, state> {
        fun_bs(double x0, double r, double s)
            : x0(x0), s(s), mu(r-0.5*s*s) {};
        state operator()(const state &x) {
            return state(x.first, x0*exp(mu*x.first + s*x.second));
        };
        private:
            double x0, s, mu;
      } bs;
};
```

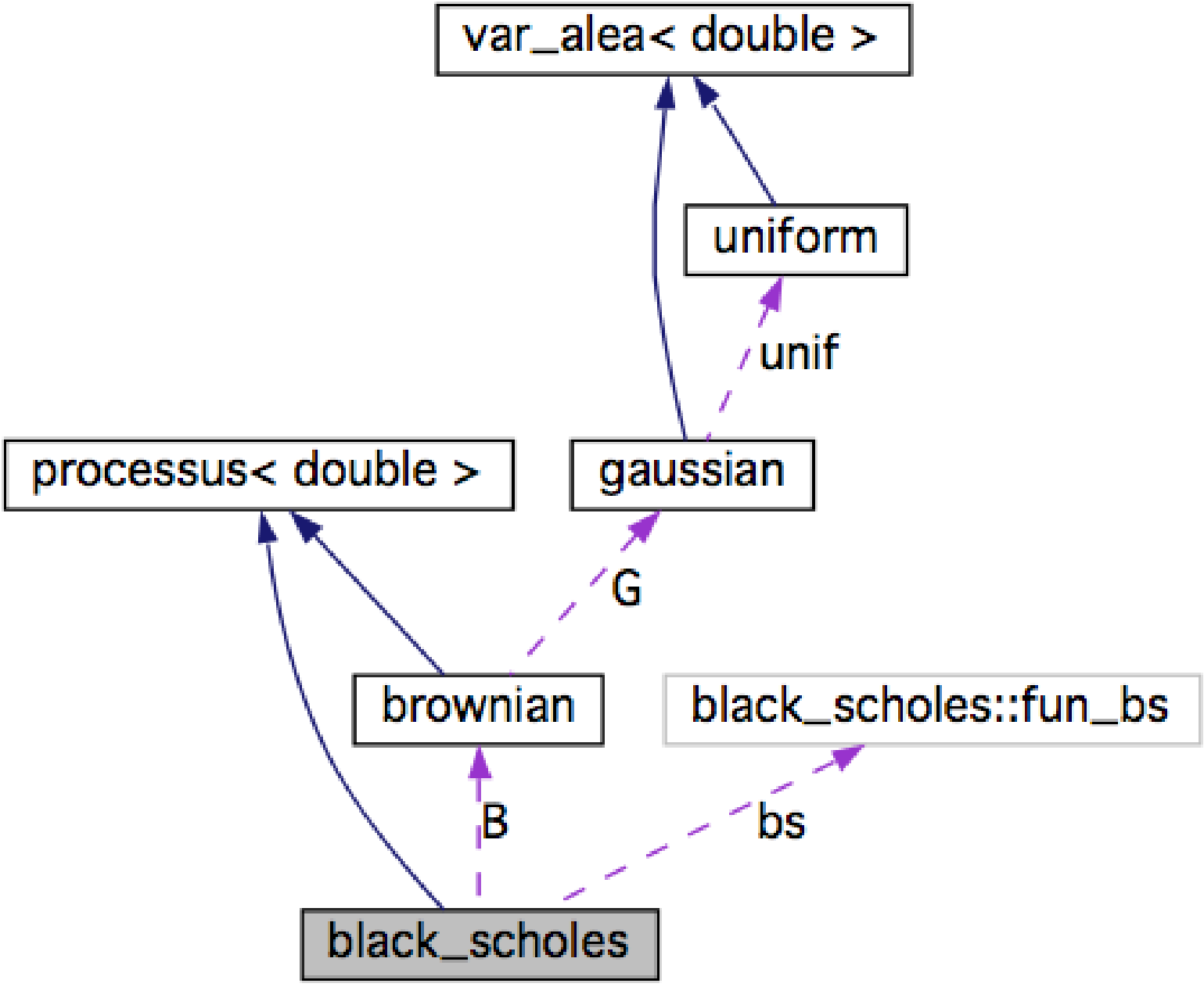
Black-Scholes version classe amie

```
struct black_scholes : public processus<double>
{
    black_scholes(int n, double x0, double r, double s, double T=1)
        : processus<double>(pow(2,n)+1), B(n, T), bs(x0, r, s) {};
    result_type operator()() {
      B();
      std::transform(B.value.begin(),B.value.end(),value.begin(),bs);
      return value;
    };
    private:
      brownian B;
      struct fun_bs : public std::unary_function<state, state> {
        fun_bs(double x0, double r, double s)
            : x0(x0), s(s), mu(r-0.5*s*s) {};
        state operator()(const state &x) {
            return state(x.first, x0*exp(mu*x.first + s*x.second));
        };
        private:
            double x0, s, mu;
      } bs;
};
```

100 trajectoires de Black-Scholes



Arbre d'héritage de la classe (version amie)



Ornstein-Uhlenbeck

Soit $\left(X_t\right)_{t\geqslant 0}$ solution de

$$\mathrm{d}X_t = \lambda\left(\mu - X_t\right)\mathrm{d}t + \sigma \mathrm{d}W_t, \quad X_0 = x_0,$$

alors pour tout $t \geqslant 0$ on a

$$X_t = x_0 e^{-\lambda t} + \mu(1 - e^{-\lambda t}) + \sigma\sqrt{\frac{1 - e^{-2\lambda t}}{2\lambda}}G,$$

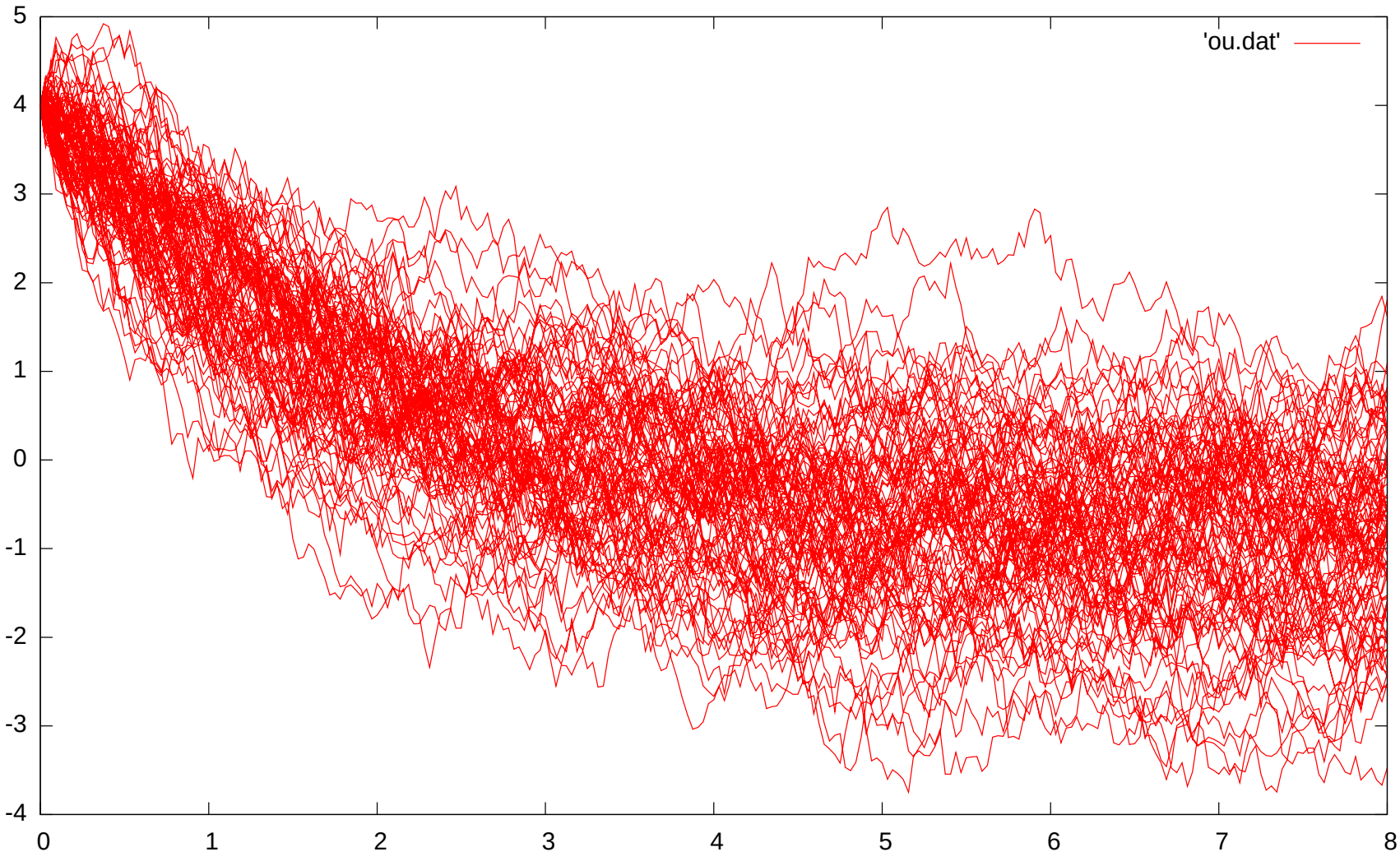
où $G \sim \mathcal{N}(0; 1)$.

- Processus ergodique, de mesure invariante $\mathcal{N}\left(\mu; \frac{\sigma^2}{2\lambda}\right)$ (récurrent positif).
- Simulation exacte par une marche aléatoire.
- Dans le cas d'un Ornstein plus général (avec coefficient de diffusion non constant ou dirigé par un Lévy), il est important de discrétiser/simuler $e^{\lambda t}X_t$ plutôt que X_t...

Ornstein-Uhlenbeck (simulation exacte)

```
struct ornstein_uhlenbeck : public processus<double>
{
    ornstein_uhlenbeck(int n, double x0, double lambda, double mu,
                       double s, double T = 1)
        : T(T), h(pow(2.,-n)), x0(x0), ret(exp(-lambda*h)),
          G(mu*(1-ret), s*s*(1-exp(-2*lambda*h))/(2*lambda)) {};
    result_type operator()() {
        value.clear();
        state val_k(0, x0);
        value.push_back(val_k);
        do {
            val_k.first += h;
            val_k.second = val_k.second*ret + G();
            value.push_back(val_k);
        } while (val_k.first < T);
        return value;
    };
    private:
        double x0, h, T, ret;
        gaussian G;
};
```

100 trajectoires d'Ornstein-Uhlenbeck



Square root process (CIR)

Let $\left(X_t\right)_{t\geqslant 0}$ solution of the SDE

$$\mathrm{d}X_t = \alpha\left(b - X_t\right)\mathrm{d}t + \sigma\sqrt{X_t}\,\mathrm{d}W_t, \quad X_0 = x_0 > 0,$$

with $\alpha, b > 0$ and $2\alpha b \geqslant \sigma^2$. By Feller classification under theses conditions $X(t) > 0$ for all t, almost surely.

Proposition

The distribution of X_t given X_u $(u < t)$ is, up to a scale factor, a noncentral chi-square distribution

$$X_t \sim \frac{\sigma^2(1 - e^{-\alpha(t-u)})}{4\alpha}\chi'^2_d\left(\frac{4\alpha e^{-\alpha(t-u)}}{\sigma^2(1 - e^{-\alpha(t-u)})}X_u\right),$$

where $d = \frac{4b\alpha}{\sigma^2}$.

Noncentral chi-square distribution

A noncentral chi-square random variable $\chi'^2_d(\lambda)$ with d degrees of freedom and noncentrality parameter λ has distribution

$$\mathbf{P}\left[\chi'^2_d(\lambda) \leqslant y\right] = \sum_{j=0}^{+\infty} \left(e^{-\frac{\lambda}{2}} \frac{(\lambda/2)^j}{j!}\right) \frac{1}{2^{d/2+j}\Gamma(d/2+j)} \int_0^y e^{-z/2} z^{d/2+j-1}\,\mathrm{d}z,$$

and a central chi-square random variable $\chi^2_\nu(\lambda)$ has distribution

$$\mathbf{P}\left[\chi^2_\nu(\lambda) \leqslant y\right] = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} \int_0^y e^{-z/2} z^{\nu/2-1}\,\mathrm{d}z,$$

(special case of the gamma distribution with scale parameter $\beta = 2$ and shape parameter $a = \nu/2$).

A general method to simulate $X \sim \chi'^2_d(\lambda)$ is to

1. simulate N a Poisson random variable with mean $\frac{\lambda}{2}$
2. simulate X as a chi-square random variable with $d + 2N$ degrees of freedom.

Processus de Poisson homogène

Soit $\left(N_t\right)_{t\geqslant 0}$ processus de Poisson d'intensité $\lambda > 0$ (homogène) *i.e.* issu de 0 à valeurs dans $\mathbf{N}$ tel que

- $\forall t > 0,\ N_t \sim \mathcal{P}(\lambda t)$
- $\forall s, t > 0,\ N_{t+s} - N_t \perp\!\!\!\perp N_t$

Caractérisation : Soit $\left(S_n\right)_{n\geqslant 0}$ une suite de v.a. *i.i.d.* $\sim \mathcal{E}(\lambda)$.
Alors $\left(N_t\right)_{t\geqslant 0}$ défini par

$$N_t = \sum_{n=1}^{+\infty} \mathbf{1}_{\{T_n \leqslant t\}}, \quad T_n = S_1 + \cdots + S_n,$$

est un processus de Poisson d'intensité λ.

Propriété : (utile pour la simulation)

$$\mathcal{L}\left((T_1, \ldots, T_k) | N_T = k\right) \sim \mathcal{L}\left((TU_{(1)}, \ldots, TU_{(k)})\right),$$

où $(u_{(1)}, \ldots, u_{(k)})$ est le vecteur ordonné.

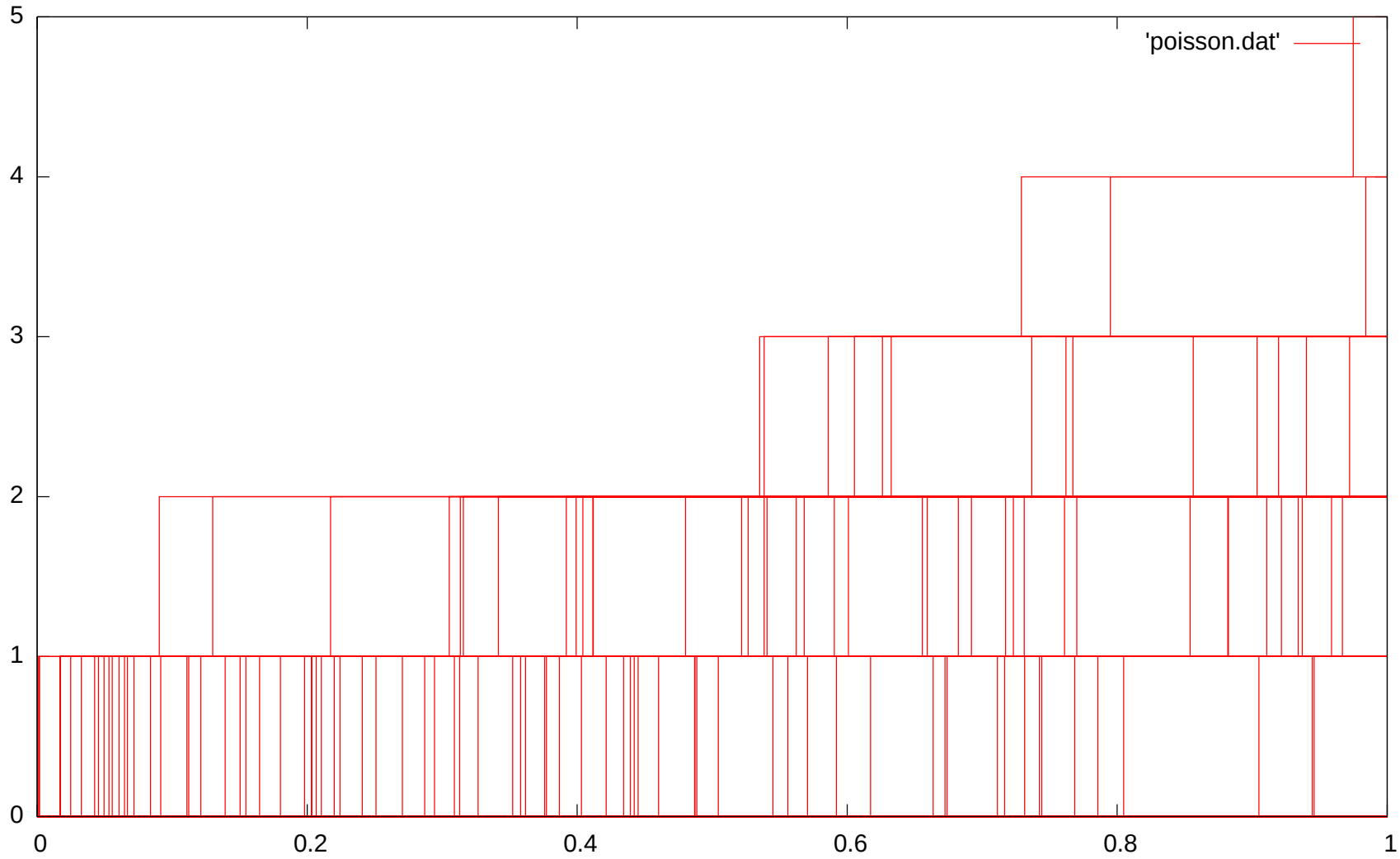
Classe ppoisson par marche aléatoire

```
struct ppoisson : public processus<unsigned>
{
    ppoisson(double lambda, double T = 1.) : E(lambda), T(T) {};
    result_type operator()() {
        value.clear();
        state val_k(0, 0);
        value.push_back(val_k);
        val_k.first += E();
        while (val_k.first < T) {
            val_k.second += 1;
            value.push_back(val_k);
            val_k.first += E();
        };
        val_k.first = T;
        value.push_back(val_k);
        return value;
    };
    protected:
        double T;
        expo E;
};
```

Classe ppoisson par marche aléatoire

```
struct ppoisson_bis : public processus<unsigned>
{
    ppoisson_bis(double lambda, double T = 1.)
        : U(0,1), P(lambda*T), T(T) {};
    result_type operator()() {
        value.clear();
        value.push_back(state(0,0));
        std::vector<double> vect_Uk(P());
        std::generate(vect_Uk.begin(), vect_Uk.end(), U);
        std::sort(vect_Uk.begin(), vect_Uk.end());
        for (int k = 0; k < vect_Uk.size(); k++) {
            value.push_back(state(T*vect_Uk[k], k+1));
        }
        value.push_back(state(T, P.current()));
        return value;
    };
    protected:
        double T;
        uniform U;
        poisson P;
};
```

100 trajectoires d'un Poisson homogène



Poisson composé

Soit $\left(X_t\right)_{t\geqslant 0}$ défini par

$$X_t = \sum_{n\geqslant 1} Y_n \mathbf{1}_{\{T_n \geqslant t\}},$$

où

- $\left(Y_n\right)_{n\geqslant 0}$ suite de v.a. *i.i.d.* $\sim \nu \perp\!\!\!\perp \left(T_n\right)_{n\geqslant 0}$.
- $\left(T_n\right)_{n\geqslant 0}$ est la suite des instants de sauts d'un processus de Poisson $\left(N_t\right)_{t\geqslant 0}$ d'intensité λ.

Alors $\left(X_t\right)_{t\geqslant 0}$ est un processus de Poisson composé d'intensité λ et de loi de saut μ.

On le note aussi

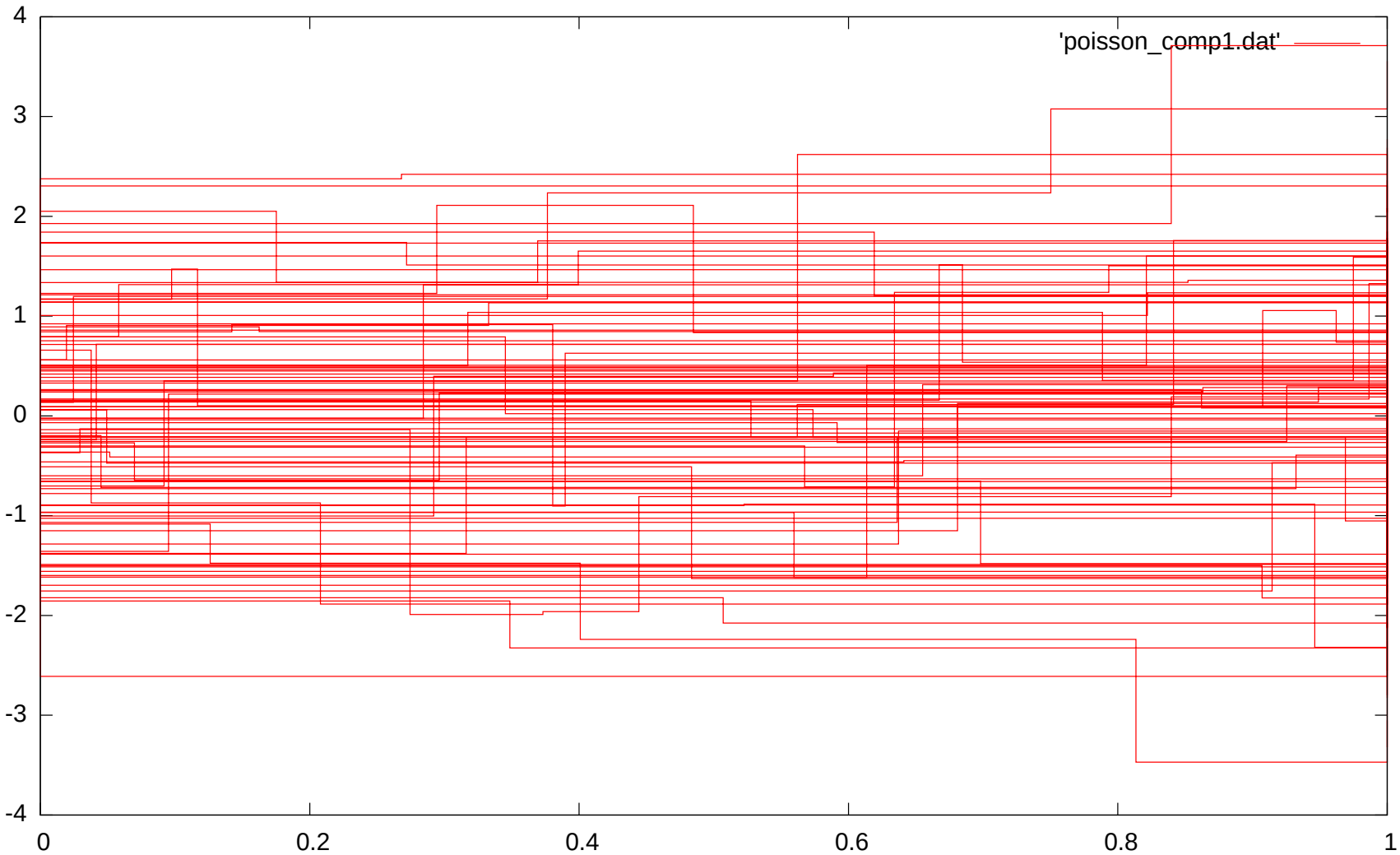
$$X_t = \sum_{n=1}^{N_t} Y_n.$$

Simulation facile ! (quand on utilise ce qui est déjà fait...)

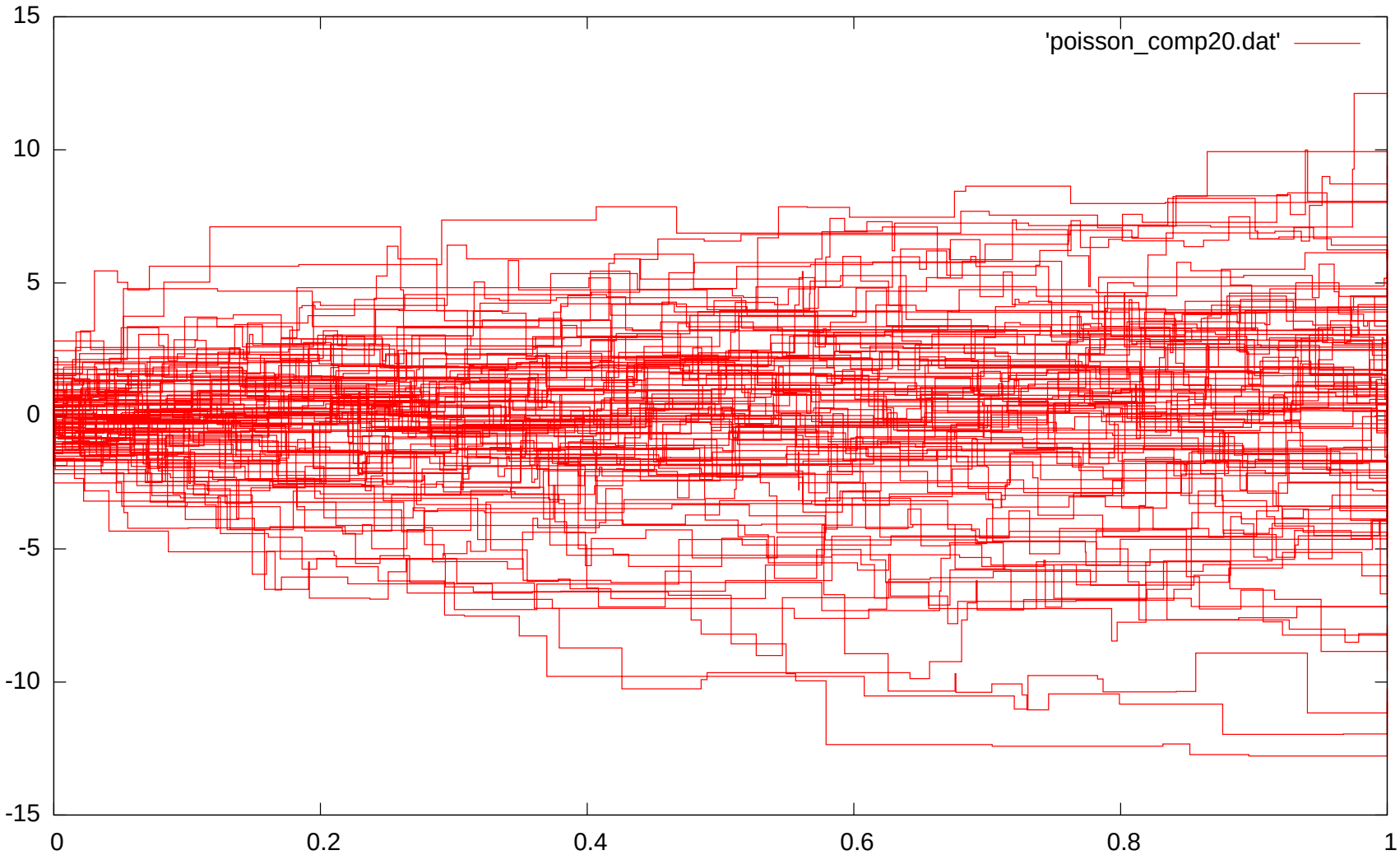
Classe ppoisson par marche aléatoire

```
struct ppoisson_compose : public processus<double>
{
    ppoisson_compose(double lambda, var_alea<double> &X, double T=1)
        : N(lambda, T), X(X) {};
    result_type operator()() {
      value.clear();
      state val_k = state(0,0);
      value.push_back(val_k);
      N();
      for (ppoisson::cst_iter i=N.value.begin();i!=N.value.end();i++)
      {   val_k.first = (*i).first;
          val_k.second += X();
          value.push_back(val_k);
      }
      value.push_back(state(N.T, val_k.second));
      return value;
    };
    protected:
        ppoisson N;
        var_alea<double> &X;
};
```

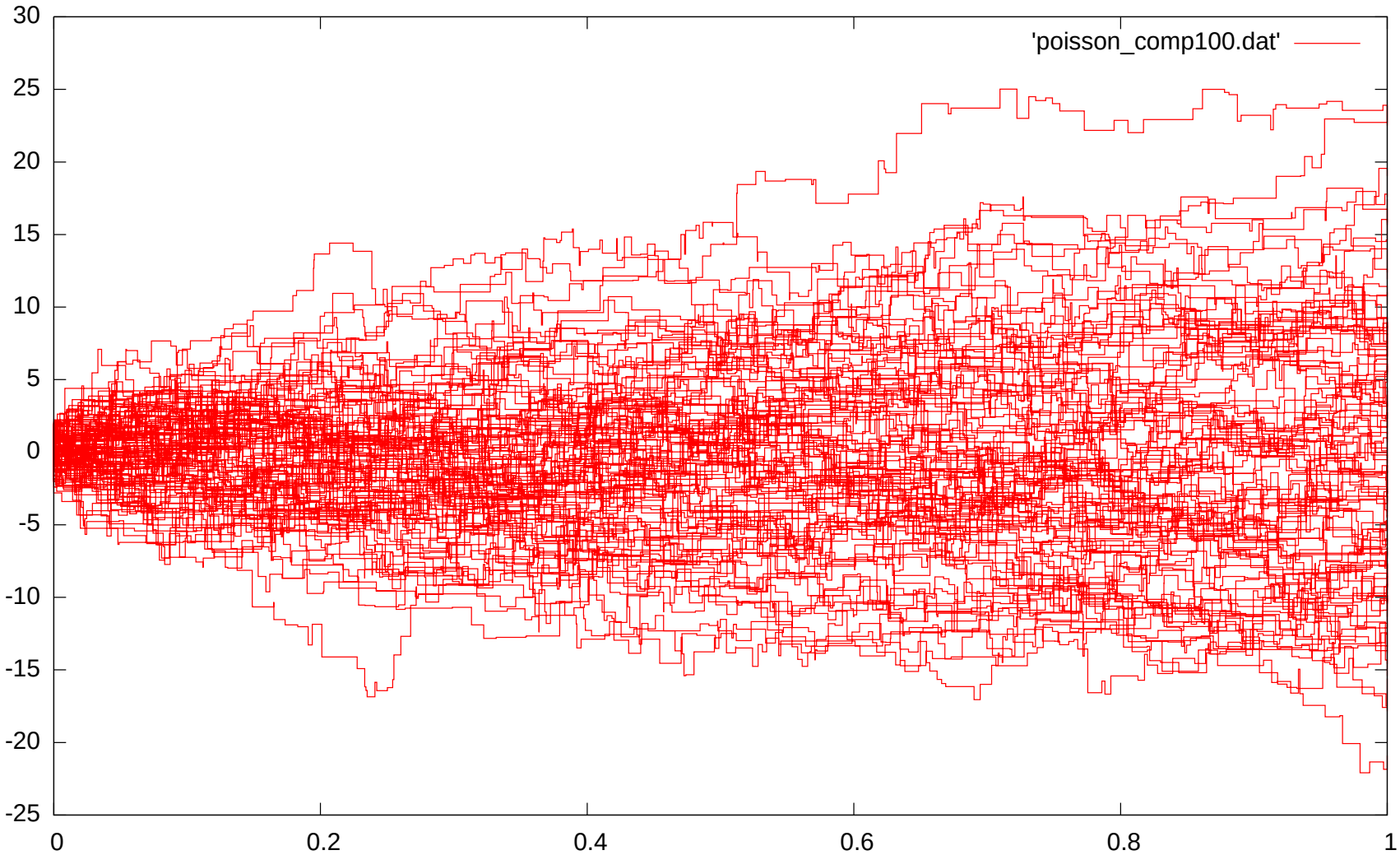
100 trajectoires d'un Poisson composé $\lambda = 1$



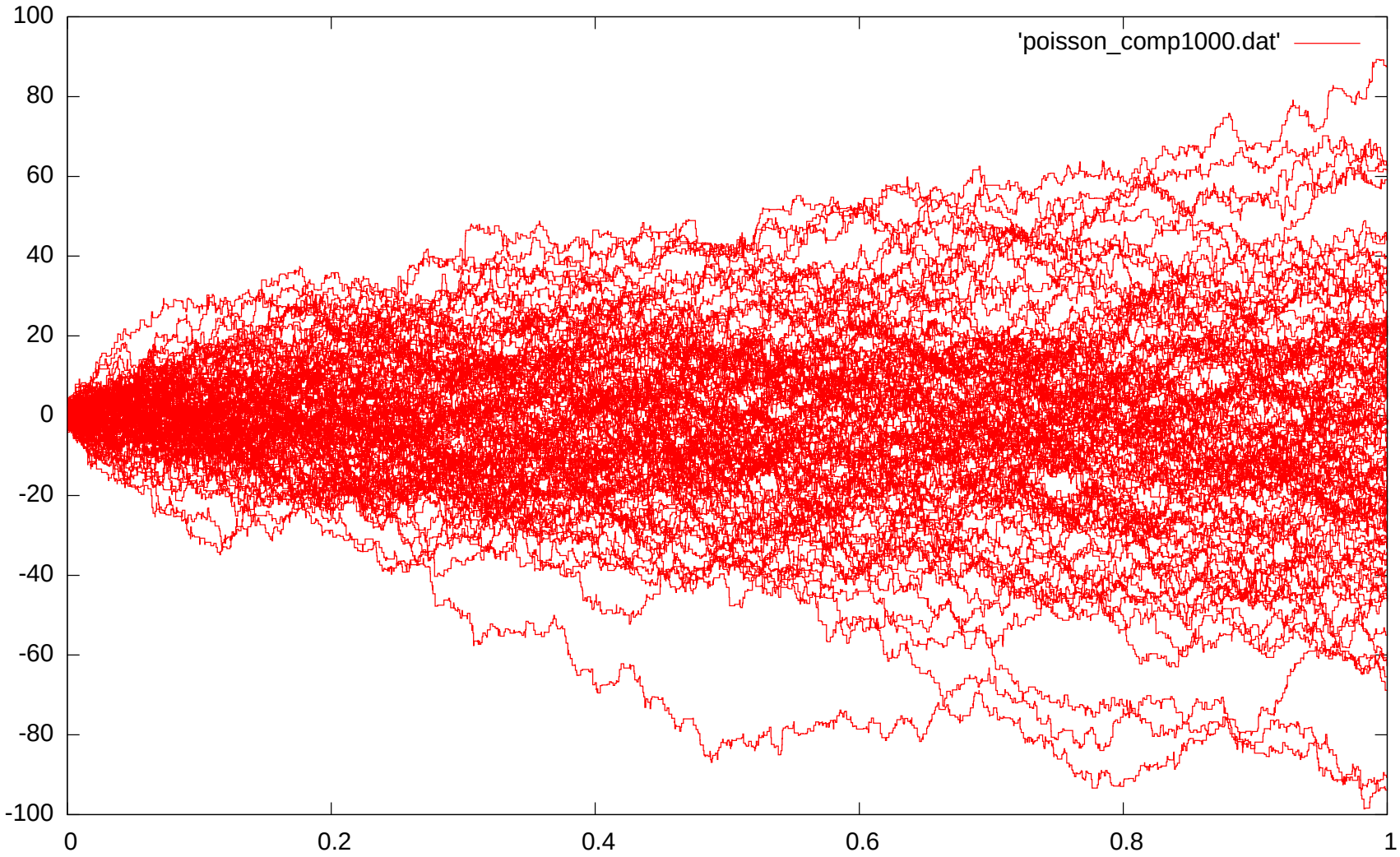
100 trajectoires d'un Poisson composé $\lambda = 20$



100 trajectoires d'un Poisson composé $\lambda = 100$



100 trajectoires d'un Poisson composé $\lambda = 1000$



Brownien + Poisson composé

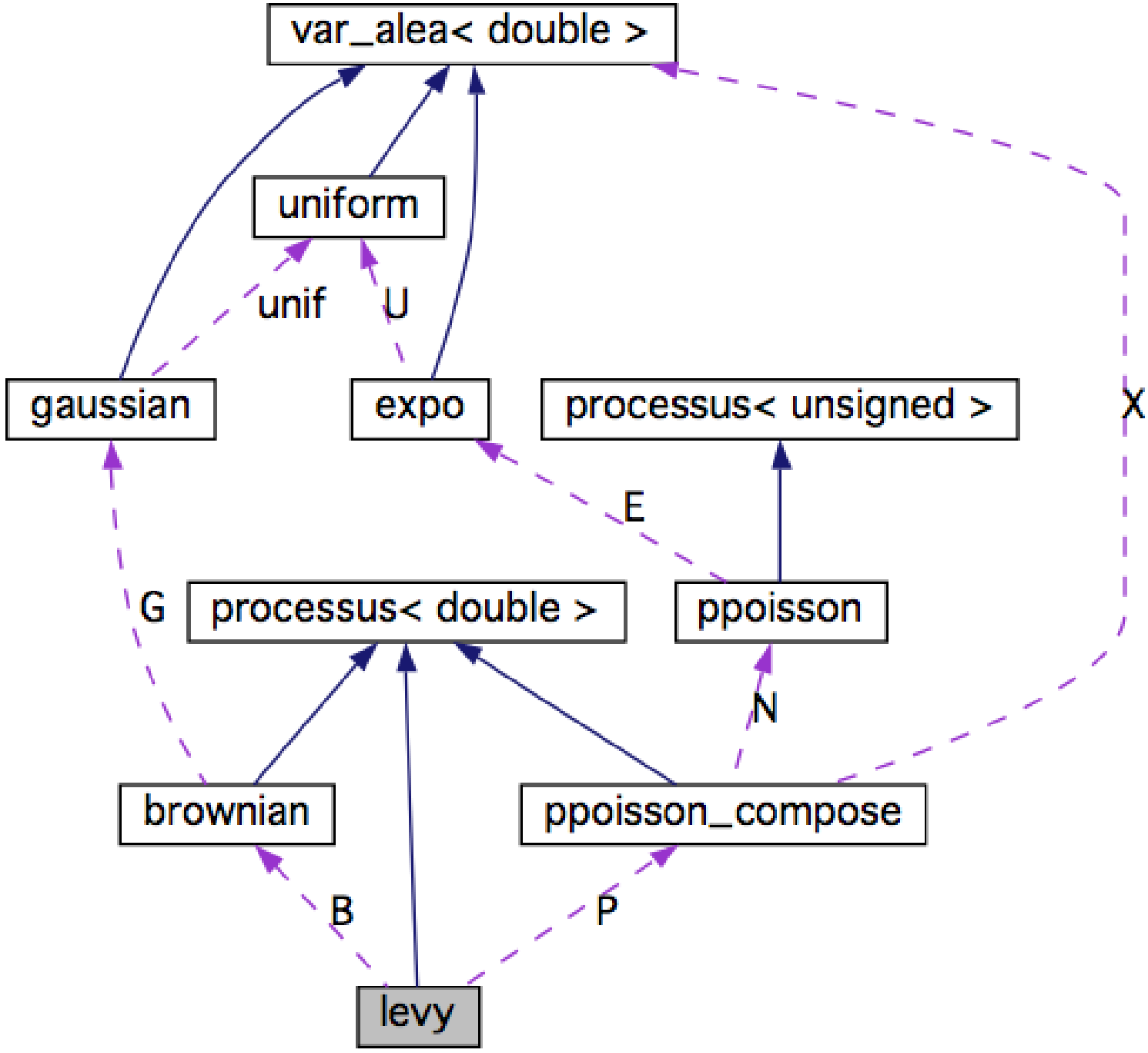
Soit $\left(X_t\right)_{t\geqslant 0}$ défini par

$$X_t = \nu t + B_t + \sum_{k=1}^{N_t} Y_k,$$

avec $\nu \in \mathbf{R}$.

- simulation d'un Brownien de pas 2^{-n}
- simulation d'un Poisson composé (d'instants de sauts T_k)
- algorithme « d'addition » :
  - parcourir les instants $t_j = j2^{-n}$
  - si $t_j < T_k < t_{j+1}$ on affine la trajectoire brownienne en T_k et on ajoute le saut puis on incrément k,
  - sinon facile !

A faire en exercice ! Très bon exercice pour manipuler les itérateurs. (correction par mail si nécessaire).
Alternative : simulation en simultané du Brownien et du Poisson...
Très utilisé pour approcher des Lévy qu'on ne sait pas simuler de façon exacte.


var_alea< double >
uniform
unif
U
gaussian
expo
processus< unsigned >
X
E
G
processus< double >
ppoisson
N
brownian
ppoisson_compose
B
P
levy

Brownien subordonné

Changement de temps d'un Brownien par un processus de Lévy croissant (positif, appelé subordinateur) indépendant du Brownien.

$$X_t = B_{Y_t},$$

où $\left(Y_t\right)_{t\geqslant 0}$ est le subordinateur du processus de Lévy $\left(X_t\right)_{t\geqslant 0}$.

Si on sait simuler le subordinateur, aucune difficulté pour simuler le Lévy.

Exemples :

- Variance-Gamma : le subordinateur est un PAIS dont la loi des accroissements est la loi Gamma.
- Normal Inverse Gaussian : le subordinateur est un PAIS dont la loi des accroissements est la loi Inverse Gaussian.

Design possible en C++ :

- une classe `pais` qui simule un PAIS par une marche aléatoire (de loi donnée), on pourra alors faire deriver `brownian`, `ornstein_uhlenbeck`, `gamma`, `inverse_gaussian` de cette classe
- une classe `brownien_subordonne` qui se construit avec un subordinateur en argument

By Cholesky

Generate a random vector $X = (X_0, X_{\frac{1}{n}}, \quad , X_{\frac{n-1}{n}})$ from a zero mean stationary Gaussian process $(X_t)_{t\in[0,T]}$ with prescribed covariance function

$$\forall t \in [0,T], \quad \gamma(t) = \gamma(0,t) = \mathrm{cov}(X_0, X_t).$$

Then $X \sim \mathcal{N}(0, \Sigma)$ where

$$\Sigma = \begin{pmatrix} \gamma(0) & \gamma(\frac{1}{n}) & \dots & \gamma(\frac{n-1}{n}) \\ \gamma(\frac{1}{n}) & \gamma(0) & \dots & \gamma(\frac{n-2}{n}) \\ \vdots & \vdots & & \vdots \\ \gamma(\frac{n-1}{n}) & \gamma(\frac{n-2}{n}) & \dots & \gamma(0) \end{pmatrix}$$

- Σ is a symmetric Toeplitz matrix, nonnegative definite.
- Cholesky transform provides a lower triangular matrix A $(n \times n)$ satisfying

$$AA^t = \Sigma \quad \text{and} \quad X = AG$$

where $G \sim \mathcal{N}(0, \mathrm{Id}_n)$. Complexity is n^2.

By Fast Fourier Transform

The essence of the simulation by FFT is the two steps :

1. Embed Σ in a circulant covariance matrix C $(m \times m)$ where $m \geqslant 2(n-1)$ (and $m = 2^l$)

$$C = \begin{pmatrix} c_0 & c_1 & \dots & c_{m-1} \\ c_{m-1} & c_0 & \dots & c_{m-2} \\ \vdots & \vdots & & \vdots \\ c_1 & c_2 & \dots & c_0 \end{pmatrix}$$

2. By FFT (twice) generate a random vector $Y \sim \mathcal{N}(0, C)$ and extract $X = (Y_0, \dots, Y_{n-1}) \sim \mathcal{N}(0, \Sigma)$.

For the first step we can consider

$$c_j = \begin{cases} \gamma(\frac{j}{n}) & \text{if } 0 \leqslant j \leqslant \frac{m}{2} \\ \gamma(\frac{m-j}{n}) & \text{if } \frac{m}{2} \leqslant j < m-1. \end{cases}$$

Warning : It may happen that C fails to be nonnegative definite !

By Fast Fourier Transform

Proposition

The eigenvalues of a circulant matrix are

$$\lambda_k = \sum_{j=0}^{m-1} c_j e^{-\frac{2\pi i j k}{m}}, \quad k = 0, \ldots, m-1.$$

And $C = Q\Lambda Q^*$ *where* $\Lambda = \mathrm{diag}\{\lambda_0, \ldots, \lambda_{m-1}\}$ *is the diagonal matrix with the eigenvalues of* C *and* $Q = (q_{jk})$,

$$q_{jk} = \frac{1}{\sqrt{m}} \exp\left(-\frac{2\pi i j k}{m}\right).$$

- simulate $Q^* Z$ directly and computation of $W_j = \sqrt{\lambda_j}(Q^* Z)_j$,
- computation (using FFT) of Y

$$Y_k = \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} W_j e^{-\frac{2\pi i j k}{m}}, \quad k = 0, \ldots, m-1.$$

By Fast Fourier Transform

Proposition

The random vector Q^*Z can be simulated directly using m independent random variables $\mathcal{N}(0,1)$, $(S_j, 0 \leqslant j \leqslant \frac{m}{2})$ and $(T_j, 1 \leqslant j \leqslant \frac{m}{2} - 1)$

$$
\begin{aligned}
(Q^*Z)_0 &= S_0, \\
(Q^*Z)_{\frac{m}{2}} &= S_{\frac{m}{2}}, \\
(Q^*Z)_j &= \frac{1}{\sqrt{2}}(S_j + iT_j) \quad \forall j = 1, \ldots, \frac{m}{2} - 1 \\
(Q^*Z)_j &= \frac{1}{\sqrt{2}}(S_j - iT_j) \quad \forall j = \frac{m}{2} + 1, \ldots, m - 1
\end{aligned}
$$

Fractional Brownian motion

Let $\left(B_t^H\right)_{t\geqslant 0}$ a fractional Brownian motion *i.e.* a zero mean Gaussian process with covariance function

$$r_H(t,s) = \frac{\sigma^2}{2}\left(|t|^{2H} + |s|^{2H} - |t-s|^{2H}\right)$$

where $H \in (0,1)$ is the *Hurst parameter* or self-similarity parameter).

For any fixed $h > 0$ the increments $B_h^H, B_{2h}^H - B_h^H, \ldots$ form a stationary Gaussian sequence with covariance function

$$\gamma(n) = \frac{\sigma^2 h^{2H}}{2}\left((n+1)^{2H} + (n-1)^{2H} - 2n^{2H}\right)$$

- if $H = \frac{1}{2}$ it is the standard Brownian motion
- if $H > \frac{1}{2}$ the increments are positively correlated
- if $H < \frac{1}{2}$ the increments are negatively correlated

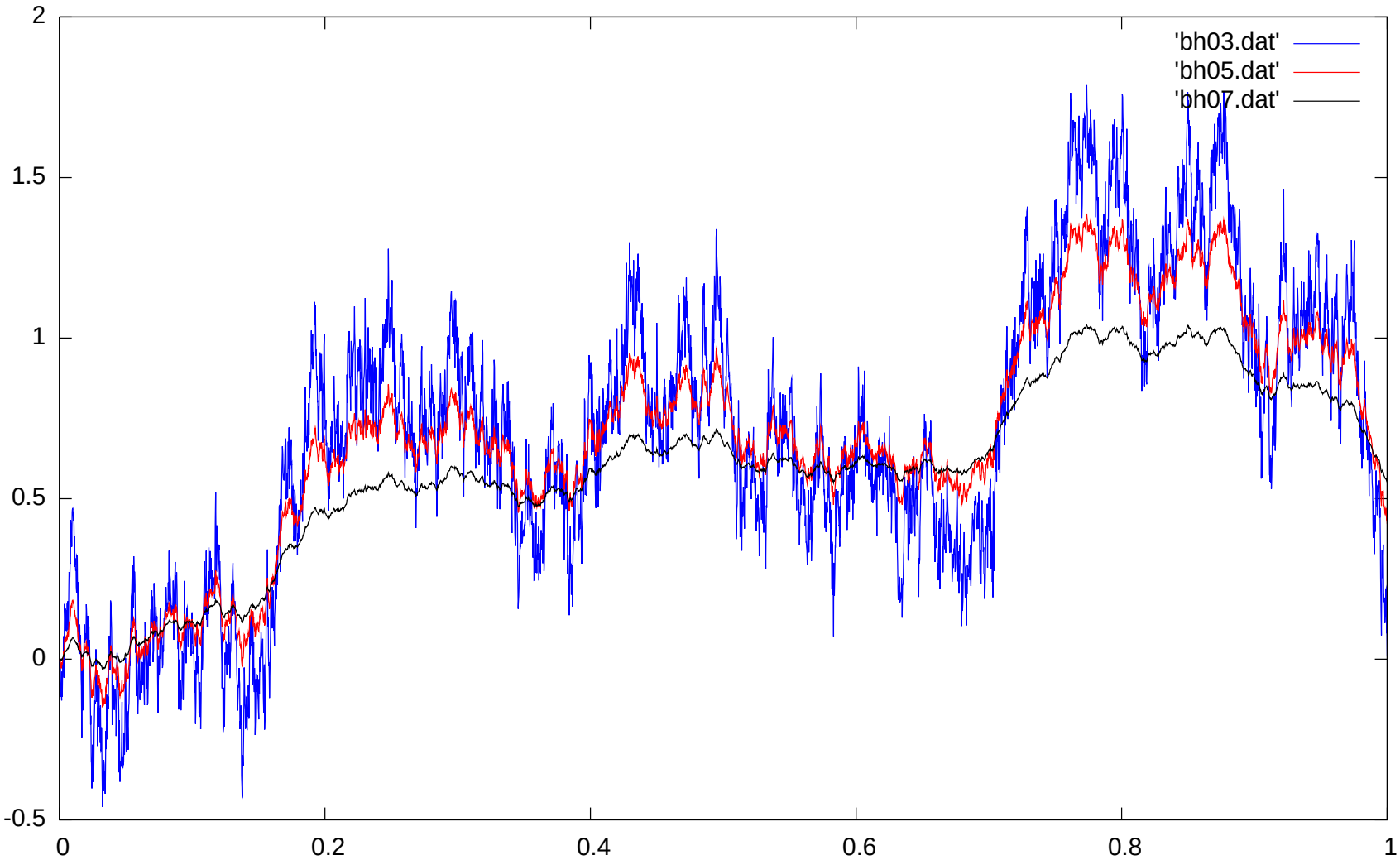
Un mot sur `std::complex` et la FFT

Voici une version très naïve de la FFT.
Vérifier que la complexité est bien $n \log_2(n)$.

```
typedef std::vector< std::complex<double> > vec_comp;
vec_comp fft(vec_comp x) {
    if (x.size() == 1) { return x; }
    std::pair< vec_comp, vec_comp > xy = decoupe(x);
    vec_comp f_k_e = fft(xy.first, signe);
    vec_comp f_k_o = fft(xy.second, signe);
    vec_comp result(x.size());
    int m = x.size() / 2;
    std::complex<double> g =
        exp(-2*M_PI*std::complex<double>(0,1)/(double) x.size());
    for (int k = 0; k < x.size(); k++)
        result[k] = f_k_e[k % m] + pow(g, k) * f_k_o[k % m];
    return result;
};
```

La fonction decoupe renvoie une paire de vecteurs dont le premier est constitué des composantes paires et le second des composantes impaires.

Brownien fractionnaire ($H = 0.3, 0.5, 0.7$)



Exact simulation (Beskos and Roberts)

We consider the one dimensional SDE on $[0, T]$

$$\mathrm{d}X_t = b(X_t)\,\mathrm{d}t + \mathrm{d}W_t, \quad X_0 = 0.$$

Let $\mathbb{W}$ the Wiener measure on $\mathscr{C}$ so that W is a BM and $\mathbb{X}$ the probability measure induced on $\mathscr{C}$ by $(X_t)_{t\in[0,T]}$.

We assume that b is $\mathcal{C}^1$ bounded with bounded derivative.

By Girsanov

$$\frac{\mathrm{d}\mathbb{X}}{\mathrm{d}\mathbb{W}} = e^{\int_0^T b(W_t)\,\mathrm{d}W_t - \frac{1}{2}\int_0^T b^2(W_t)\,\mathrm{d}t}$$

Let $B(t) = \int_0^t b(u)\,\mathrm{d}u$. Itô's formula then gives

$$\int_0^T b(W_t)\,\mathrm{d}W_t = B(W_T) - B(0) - \frac{1}{2}\int_0^T b'(W_t)\,\mathrm{d}t$$

Exact simulation (Beskos and Roberts)

Proposition

Let $\mathbb{Z}$ the probability measure induced by the μ biased Brownian motion where μ is with density proportional to h

$$h(x) = e^{B(x) - \frac{x^2}{2T}}.$$

Then $\mathbb{Z}$ is equivalent to $\mathbb{W}$ and

$$\frac{\mathrm{d}\mathbb{Z}}{\mathrm{d}\mathbb{W}} = \frac{h(W_T)}{(1/\sqrt{2\pi T})e^{-W_T^2/(2T)}}.$$

It is easy to check that

$$\frac{\mathrm{d}\mathbb{X}}{\mathrm{d}\mathbb{Z}} = \frac{\mathrm{d}\mathbb{X}}{\mathrm{d}\mathbb{W}} \frac{\mathrm{d}\mathbb{W}}{\mathrm{d}\mathbb{Z}} \propto e^{-\int_0^T (\frac{1}{2} b^2(W_t) + \frac{1}{2} b'(W_t))\, \mathrm{d}t}$$

Exact simulation (Beskos and Roberts)

- Let $\phi(x) = \frac{1}{2}(b^2(x) + b'(x)) - k$ where k is such that $\phi \geqslant 0$.
- We choose the length T of the time interval such that

$$\forall u \in \mathbf{R}, \quad 0 \leqslant \phi(u) \leqslant T^{-1}.$$

- Let $H = \int_0^T \phi(W_t)\,\mathrm{d}t$ so that

$$\frac{\mathrm{d}\mathbb{X}}{\mathrm{d}\mathbb{Z}}(\mathrm{d}\omega) = e^{-H(\omega)} \quad \text{and} \quad 0 \leqslant H \leqslant 1.$$

▷ Idea of the algorithm :

1. Simulate a realisation of the biased Brownian motion $X(\omega)$
2. Compute $H(\omega)$
3. Produce a binary indicator I such that $\mathbf{P}\left[I = 1 \,|\, \omega\right] = e^{-H(\omega)}$
4. If $I = 0$ go to 1.