Schémas numériques: options exotiques

Exemples en C++

Vincent Lemaire
vincent.lemaire@upmc.fr

Classe Process
- Motivations
- Classe Process
- Classe Kernel
- Struct algo

Méthode du pont Brownien
- Options Asiatiques
- Options sur max

Extrapolation de Richardson

Classe Process
- Motivations
- Classe Process
- Classe Kernel
- Struct algo

Méthode du pont Brownien
- Options Asiatiques
- Options sur max

Extrapolation de Richardson

Motivations

Le but est d'illustrer une programmation totalement générique sans utiliser le polymorphisme dynamique i.e. sans utiliser

- classes abstraites
- méthodes virtuelles
- la « compatibilité » des adresses entre classe fille et classe mère

Rappel : On a utilisé ce mécanisme dans les classes `var_alea`, `processus` et `scheme_diff_unidim`.

De plus en plus de librairies remplacent ce mécanisme par du polymorphisme statique, avantages :

- plus de travail à la compilation moins à l'execution : code plus rapide
- si ça compile, ça doit fonctionner...
- moins d'erreurs liés aux « casts »

Classe Process

```
template <typename KERNEL>
class Process {
  public:
    typedef typename KERNEL::state_type result_type;
    typedef typename KERNEL::omega_type omega_type;
    typedef KERNEL kernel_type;
    Process(double t0, result_type val0, KERNEL K, double T = 1)
        : K(K), t0(t0), T(T), state0(val0) { reinit(); }
    void reinit() { t = t0; state = state0; }
    double time() const { return t; }
    result_type value() const { return state; }
    omega_type last_alea() const { return K.last_alea(); }
    double last_time() const { return T; }
    bool not_end() const { return (t < T); }
    Process & operator++() { K(t, state); return *this; }
    Process & operator+=(omega_type o) { K(t, state, o);return *this;}
    result_type operator()();
  private:
    KERNEL K;
    double const t0, T; double t;
    result_type const state0; result_type state;
};
```

Définition de 2 fonctions associées

```
template <typename KERNEL>
typename Process<KERNEL>::result_type Process<KERNEL>::operator()(){
  reinit();
  K.mem_clean();
  while (not_end())
    this->operator++();
  return state;
}
```

```
template <typename KERNEL>
std::ostream & operator<<(std::ostream & out, Process<KERNEL> & X) {
  X.reinit();
  out << X.time() << "\t" << X.value() << std::endl;
  while (X.not_end()) {
    ++X;
    out << X.time() << "\t" << X.value() << std::endl;
  }
  return out;
};
```

Exercice : lister toutes les contraintes sur le type KERNEL.

Etant donné un KERNEL K on peut définir un Process X associé en déclarant :

```
Process< KERNEL > X(0, x0, K, 1);
```

Etant donné des variables aléatoires $(\varepsilon_n)_{n\geqslant 0}$ définies sur un espace de probabilité $(\Omega, \mathscr{F}, \mathbb{P})$ à valeurs dans W (muni d'une tribu $\mathscr{W}$) et un espace d'état $(E, \mathscr{E})$ on considère une équation d'évolution

$$X_{t_{n+1}} = K(t_n, X_{t_n}, \varepsilon_n)$$

où $X_{t_0} = x_0 \in E$, $K : (\mathbf{R}_+, E, W) \to E$ mesurable.

La suite $(X_{t_n})_{n\geqslant 0}$ est une chaîne de Markov inhomogène de probabilités de transition P_n

$$P_n(x, A) = \mathbf{P}\left[K(t_n, x, \varepsilon_n) \in A\right].$$

Exercice : lister toutes les contraintes sur le type KERNEL.

Etant donné un KERNEL K on peut définir un Process X associé en déclarant :

```
Process< KERNEL > X(0, x0, K, 1);
```

Etant donné des variables aléatoires $(\varepsilon_n)_{n\geqslant 0}$ définies sur un espace de probabilité $(\Omega, \mathscr{F}, \mathbb{P})$ à valeurs dans W (muni d'une tribu $\mathscr{W}$) et un espace d'état $(E, \mathscr{E})$ on considère une équation d'évolution

$$X_{t_{n+1}} = K(t_n, X_{t_n}, \varepsilon_n)$$

où $X_{t_0} = x_0 \in E$, $K : (\mathbf{R}_+, E, W) \to E$ mesurable.

La suite $(X_{t_n})_{n\geqslant 0}$ est une chaîne de Markov inhomogène de probabilités de transition P_n

$$P_n(x, A) = \mathbf{P}\left[K(t_n, x, \varepsilon_n) \in A\right].$$

(en fait on va autoriser à être un peu plus que Markovien pour faire l'extrapolation de Richardson efficacement...)

Classe Kernel

```
template <typename STATE, typename ALG, typename VA>
struct Kernel {
  typedef STATE state_type;
  typedef ALG algo_type;
  typedef VA alea_type;
  typedef typename VA::result_type omega_type;
  Kernel(ALG algo, VA alea) : algo(algo), alea(alea) {}
  omega_type last_alea() const { return alea.current(); }
  typename Kernel::omega_type omega() { return alea(); }
  typename Kernel::omega_type last_omega() const { return alea.current(); }
  void mem_clean() { algo.mem_clean(); }
  typename Kernel::omega_type mem() { return algo.mem(); }
  void operator()(double & t, state_type & x, omega_type omega) {
    algo(t,x,omega); }
  void operator()(double & t, state_type & x) { algo(t,x,alea()); }
  protected:
    VA alea;
    ALG algo;
};
```

Chaînon « abstrait », la définition de la fonction K se trouve dans `algo`...

Exemple du schéma d'Euler !

Pour un objet SDE donné on définit l'algorithme du schéma d'Euler de la façon suivante :

```
template <typename SDE, typename ST, typename OT>
struct algo_euler {
  algo_euler(double h, SDE eds) : h(h), eds(eds) {}
  void operator()(double & t, ST & x, OT omega) {
    t += h;
    x += eds.drift(x) * h + eds.sigma(x) * omega;
  }
  private:
    double h;
    SDE eds;
};
```

Il manque la méthode `mem_clean()` ! On verra ça plus tard...

Et maintenant que faut-il faire ?

Définir un Kernel pour algo_euler !

```
template <typename SDE>
struct Euler : public
Kernel<double, algo_euler<SDE, double, double>, gaussian>
{
  Euler(double h, SDE eds) :
  Kernel<double, algo_euler<SDE,double,double>, gaussian>
  (algo_euler<SDE, double, double>(h, eds), gaussian(0,h)) {}
};
```

C'est un peu lourd à définir mais ce sera la même chose pour tous les algorithmes, il faut juste faire attention aux types.

- Erreurs détectables à la compilation
- Une fois ce Kernel défini on peut se concentrer sur l'algorithme...

Exemple

```
double x0 = 100;
double K = x0-10;
BS X(0, 0.3);
int M = 1e6;
double h = pow(2., -6);

Euler<BS> K(h, X);
Process< Euler<BS> > EL(0, 100, K, 1);
vector<double> result = monte_carlo(M, VanillaCall(K), EL);
```

Est-ce que tout ceci est souple et vraiment générique ?
Exemples sur les options asiatiques, les options sur max et l'extrapolation de Richardson.

Classe Process
Motivations
Classe Process
Classe Kernel
Struct algo

Méthode du pont Brownien
Options Asiatiques
Options sur max

Extrapolation de Richardson

Schéma d'Euler continu

On rappelle que pour un pas $h > 0$ le schéma d'Euler discret aux instants $t_k = kh$ s'écrit

$$\bar{X}_{t_{k+1}} = \bar{X}_{t_k} + b(\bar{X}_{t_k})h + \sigma(\bar{X}_{t_k})(W_{t_{k+1}} - W_{t_k}),$$

et on définit le schéma d'Euler continu

$$\forall t \in [t_k, t_{k+1}], \quad \bar{X}_t = \bar{X}_{t_k} + b(\bar{X}_{t_k})(t - t_k) + \sigma(\bar{X}_{t_k})(W_t - W_{t_k}).$$

Le processus $(\bar{X}_t)_{t\in[0,1]}$ est un processus d'Itô mais pas une diffusion brownienne ! Il est solution d'une EDS avec retard... et est non markovien.

Dans les options *path-dependant* il est utile de considérer ce schéma continu au schéma discret naïf.

Options Asiatiques ($T = 1$)

$$\varphi\left(\int_0^1 X_s \mathrm{d}s, X_1\right)$$

On doit approcher (sur la même trajectoire brownienne) X_1 et $\int_0^1 X_s \mathrm{d}s = \sum_{k=0}^{N-1} \int_{t_k}^{t_{k+1}} X_s \mathrm{d}s.$

Tois solutions :

- version rectangle : $\bar{X}_1$ et $\sum_{k=0}^{N-1} \bar{X}_{t_k} h$
- version trapèze : $\bar{X}_1$ et $\sum_{k=0}^{N-1} \frac{\bar{X}_{t_k} + \bar{X}_{t_{k+1}}}{2} h$
- version continue : $\bar{X}_1$ et $\sum_{k=0}^{N-1} \int_{t_k}^{t_{k+1}} \bar{X}_s \mathrm{d}s$

Exercice : comment simuler la version continue ?

Solution, codes et expérimentations numériques

En live !

Options sur max : barrier, lookback, etc.

$$\varphi\left(\max_{s\in[0,1]} X_s, X_1\right)$$

On doit approcher (sur la même trajectoire brownienne) X_1 et $\max_{s\in[0,1]} X_s = \max_{k=0,\dots,N-1} \max_{s\in[t_k,t_{k+1}]} X_s$.

Deux solutions :

- version naïve : $\bar{X}_1$ et $\max_{k=0,\dots,N-1} \bar{X}_{t_k}$,
- version continue : $\bar{X}_1$ et $\max_{k=0,\dots,N-1} \max_{s\in[t_k,t_{k+1}]} \bar{X}_s$.

Exercice : comment simuler la version continue ?

Solution, codes et expérimentations numériques

En live !

Classe Process
- Motivations
- Classe Process
- Classe Kernel
- Struct algo

Méthode du pont Brownien
- Options Asiatiques
- Options sur max

Extrapolation de Richardson

Extrapolation de Richardson

C'est pour ça qu'on avait besoin de mémoire : `mem_state`, `mem()` et `mem_clean()` dans les algorithmes précédents.

```
template<typename KERN1, typename KERN2, typename ST, typename OT>
struct algo_richardson {
  algo_richardson(KERN1 K1, KERN2 K2, int n) : K1(K1), K2(K2), n(n) {
    mem_clean(); }
  void operator()(double & t, ST & x, OT omega) {
    double _t = t;
    K1.mem_clean();
    K2.mem_clean();
    for (int k = 0; k < n; ++k)
      K1(_t, x.first);
    K2(t, x.second, K1.mem());
  }
  void mem_clean() { K1.mem_clean(); K2.mem_clean(); }
  private:
    KERN1 K1;
    KERN2 K2;
    int n;
};
```

Solution, codes et expérimentations numériques

En live !