Project 1: Use constant parameters in Monte Carlo engines

Audepin Nicolas
Khabou Salma
Elasri Amine
Krichen Fatma Azzahra
Le Forestier Elise

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

UV F4B304
C++ pour la finance : QuantLib
Luigi Ballabio

I. INTRODUCTION

L’algorithme de Monte Carlo permet de faire des prévisions sur le prix relatif à une option.

Dans la librairie QuantLib, lors d’une prévision, plusieurs paramètres sont mis à jours à chaque itération ce qui occupe une partie non négligeable du temps de calcul de la prévision. Les paramètres considérés sont: le sous-jacent, taux sans risque, taux de dividende et volatilité.

QuantLib se sert d’une classe GeneralizedBlackScholesProcess pour prévoir l’évolution de ces prix.

L’objectif de ce projet est d’ajouter la possibilité de lancer l’algorithme de Monte-Carlo sur des paramètres fixes pour économiser le temps de calcul et d’étudier le gain en temps et la perte de précision de cette méthode.

II. CODE

1) ConstantBlackSholesProcess

Nous avons crée une nouvelle classe ConstantBlackScholesProcess, inspirée de la classe GeneralizedBlackScholesProcess mais qui ne prend pas en compte l'évolution des paramètres dans le temps.
Dans la classe constantBlackScholesProcess, les valeurs "strike" et "exercie_Date" sont constantes.
Les fonctions "drift" et "diffusion" sont calculés dans le constructeur car elles sont désormais constantes.

```
namespace QuantLib {
    constantBlackScholesProcess::constantBlackScholesProcess(
        const Handle<Quote> x0,
        const Date exercice_Date,
        const Real strike,
        const Handle<YieldTermStructure>& risk_free_BS,
        const Handle<BlackVolTermStructure>& volatility_BS,
        const Handle<YieldTermStructure>& dividend_yield_BS,
        const boost::shared_ptr<discretization>& disc)
        :StochasticProcess1D(disc), x0_(x0), strike_(strike),

        riskFreeRate_(risk_free_BS), dividendYield_(dividend_yield_BS), blackVolatility_(volatility_BS)
        {
        exercice_date = exercice_Date;
        risk_drift = riskFreeRate_->zeroRate(exercice_date,
                                    riskFreeRate_->dayCounter(),
                                    Continuous,
                                    NoFrequency,
                                    true)
                    - dividendYield_->zeroRate(exercice_date,
                                               riskFreeRate_->dayCounter(),
                                               Continuous,
                                               NoFrequency,
                                               true);
        diffusion_ = blackVolatility_->blackVol(exercice_date,strike_);
    }
    Real constantBlackScholesProcess::x0()const {
        return x0_->value();
    }
    Real constantBlackScholesProcess::drift(Time t, Real s) const {
        return risk_drift*s;
    }
    Real constantBlackScholesProcess::diffusion(Time t, Real s) const {
        return diffusion_*s;
    }
}
```

2) MCEuropeanEngine

Nous avons modifié la classe MCEuropeanEngine en ajoutant un boolean permettant de choisir quelle classe utiliser entre ConstantBlackScholesProcess et GeneralizedBlackScholesProcess.

Nous avons surchargé la méthode pathGenerator de la manière suivante : Lorsque le boolean est dans l'état "false", la simulation de Monte Carlo est lancée avec la classe standard. Si le boolean est dans l'état "true", l'algorithme prend en compte la classe ConstantBlackScholesProcess.

On utilise les paramètres extraits du processus actuel pour la construction d'une instance de la classe constanBlackScholesProcess.

```
if (this->constant_ ){
    boost::shared_ptr<GeneralizedBlackScholesProcess> process
    =boost::dynamic_pointer_cast<GeneralizedBlackScholesProcess>(this->process_);
    boost::shared_ptr<PlainVanillaPayoff> payoff =
    boost::dynamic_pointer_cast<PlainVanillaPayoff>(this->arguments_.payoff);
    QL_REQUIRE(payoff, "non-plain payoff given");

    return boost::shared_ptr<path_generator_type>(
                            new
                            path_generator_type(boost::shared_ptr<constantBlackScholesProces
                            s> (new constantBlackScholesProcess(process->stateVariable(),
                            this->arguments_.exercise->lastDate(),
                            payoff->strike(),
                            process->riskFreeRate(),

                            process->blackVolatility(),
                            process->dividendYield()
                                                    )),grid,
                                generator, brownianBridge_)

                              );

}
else{

    return boost::shared_ptr<path_generator_type>(
        new path_generator_type(process_, grid,
                                generator, brownianBridge_));
     }
}
```

3) Main :

Afin de tester les codes précédents, nous avons écrit un main qui permet de créer des options et de retourner leurs prix après un laps de temps. Ces prix seront calculer en utilisant les deux process.

a) Initialisation des paramètres :

```
using namespace QuantLib;

int main() {

    try {
        int rand_num = 0;
        srand(time(NULL));
        rand_num = rand();
        std::cout << rand_num << std::endl;
        Calendar calendar = TARGET();
        DayCounter dayCounter = Actual365Fixed();
        //date du jour
        Date t0(1, March, 2019);
        //maturite
        Date T(1, March, 2020);
        //type d'option
        Option::Type type(Option::Call);
        //Sous jacent
        Real stock_price = 90;
        //Prix d'exercice
        Real strike = 100;
        //Taux de dividende
        Spread q = 0.02;
        //Taux d'interet
        Rate r = 0.05;
        //volatilite
        Volatility vol = 0.2;
```

b) Construction des paramètres du processus :
sous-jacent, taux d'intérêt, taux de dividende, volatilité

```
// Construction du taux d'interet
Handle<YieldTermStructure> rate(boost::shared_ptr<YieldTermStructure>(new FlatForward(t0, r,
dayCounter)));
// Construction du sous jacent
Handle<Quote> underlying(boost::shared_ptr<Quote>(new SimpleQuote(stock_price)));
// Construction de la vol
Handle<BlackVolTermStructure> volatility(boost::shared_ptr<BlackVolTermStructure>(new
BlackConstantVol(t0, calendar, vol, dayCounter)));
// Construction du dividende
Handle<YieldTermStructure> dividend(boost::shared_ptr<YieldTermStructure>(new FlatForward(t0, q,
dayCounter)));
// Construction du Process Black and scholes
boost::shared_ptr<GeneralizedBlackScholesProcess> process_BS(new
GeneralizedBlackScholesProcess(underlying, dividend, rate, volatility));
// Construction de deux options europeennes
boost::shared_ptr<Exercise> europeanExercise(new EuropeanExercise(T));
boost::shared_ptr<StrikedTypePayoff> payoff(new PlainVanillaPayoff(type, strike));
VanillaOption option_1(payoff, europeanExercise);
VanillaOption option_2(payoff, europeanExercise);

//Modele de princing
option_1.setPricingEngine(boost::shared_ptr<PricingEngine>(new
AnalyticEuropeanEngine(process_BS)));
option_2.setPricingEngine(boost::shared_ptr<PricingEngine>(new
AnalyticEuropeanEngine(process_BS)));


Real price = option_1.NPV();
std::cout << "MCEuropeanEngine avec GeneralizedBlackScholesProcess" << std::endl;
std::cout << "price = option_1.NPV()=" << price << std::endl;
std::cout << "test 1 " << std::endl;
```

c) Simulation du prix de l'option en utilisant les deux classes :

option_1 : Le prix sera calculé par GeneralizedBlackScholesProcess
option_2 : Le prix sera calculé par ConstantBlackScholesProcess

```
    boost::shared_ptr<PricingEngine> eng(new MCEuropeanEngine_2<PseudoRandom>
    (process_BS,10,Null<Size>(),true,false,10000,Null<Real>(),Null<Size>(),
    SeedGenerator::instance().get(),false));

    option_1.setPricingEngine(eng);


    std::cout << "test 2" << std::endl;

    clock_t t_debut = clock();
    Real price1 = option_1.NPV();
    std::cout << "Prix de l'option " << price1 << std::endl;
    printf("Temps d'execution: %.2fs\n", (double)(clock() - t_debut) / CLOCKS_PER_SEC);
    std::cout << "Erreur d'estimation " << option_1.errorEstimate() << std::endl;


    std::cout << "MCEuropeanEngine avec constantBlackScholesProcess" << std::endl;

    boost::shared_ptr<PricingEngine>  eng1 (new MCEuropeanEngine_2<PseudoRandom>
    (process_BS,10,Null<Size>(),true,false,10000,Null<Real>(),Null<Size>(),
    SeedGenerator::instance().get(),true));
    option_2.setPricingEngine(eng1);

    clock_t t_debut_2 = clock();

    std::cout << "Prix de l'option " << option_2.NPV() << std::endl;
    printf("Temps d'execution: %.2fs\n", (double)(clock() - t_debut_2) / CLOCKS_PER_SEC);
    std::cout << "Erreur d'estimation " << option_2.errorEstimate() << std::endl;
    system("pause");

    return 0;

}
```

Nous retournons pour chaque process le temps d'exécution et l'erreur de l'estimation du prix.

III. TESTS

Pour nous permettre de comparer l'efficacité des deux classes, nous allons effectuer des tests en modifier deux critères l'un après l'autre.
Tout d'abord, nous allons faire varier le nombre d'échantillons. Ensuite, nous ferons varier le pas temporel.

1) Variation du nombre d'échantillons

On lance les process pour différentes valeurs du nombre d'échantillons: Par exemple 10 000 et 50 000.

2) Variation du pas temporel

On lance la simulation pour différentes valeurs du pas temporel. Par exemple 5s et 10s.

On note à chaque simulation la variation du temps de calcul et de la précision de l'estimation.

Problème :
Nous n'arrivons pas à afficher les résultats à cause d'une erreur de code dans le 'main'. (Le reste du code compile)

IV. CONCLUSION

Nous nous attendons à ce que le temps d'exécution soit plus court pour le ConstantBlackScholesProcess que pour le GeneralizedBlackScholesProcess puisqu'on ne prend pas en compte les variations des differents paramètres au cours du temps. On espère avoir une perte de précision minime par rapport au gain de temps.