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.

```
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.

```cpp
        boost::shared_ptr<path_generator_type> pathGenerator() const {
            Size dimensions = this->process_->factors();
            TimeGrid grid = this->timeGrid();
            typename RNG::rsg_type generator =
                    RNG::make_sequence_generator(dimensions*(grid.size()-1),this->seed_);

            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<constantBlackScholesProcess> (
                                    new constantBlackScholesProcess(process->stateVariable(),
                                    this->arguments_.exercise->lastDate(),
                                    payoff->strike(),
                                    process->riskFreeRate(),

                                    process->blackVolatility(),
                                    process->dividendYield()
                                                            )),grid,
                                        generator, this->brownianBridge_));
                                }

            else{
                return boost::shared_ptr<path_generator_type>(
                    new path_generator_type(this->process_, grid
                                        ,generator, this->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 :*

```
//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 = 100;
//Prix d'exercice
Real strike = 120;
//Taux de dividende
Spread q = 0.03;
//Taux d'interet
Rate r = 0.05;
//volatilite
Volatility vol = 0.15;
```

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)));
```

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

```cpp
option_1.setPricingEngine(boost::shared_ptr<PricingEngine>(new MCEuropeanEngine_2<PseudoRandom>(process_BS,60,Null<Size>(),
                                                                            true,false,
                                                                            10000,Null<Real>(),Null<Size>(),
                                                                            SeedGenerator::instance().get(),
                                                                            false)));

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

option_2.setPricingEngine(boost::shared_ptr<PricingEngine>(new MCEuropeanEngine_2<PseudoRandom>(process_BS,60,Null<Size>(),
                                                                        true,false,10000,Null<Real>(),
                                                                        Null<Size>(), SeedGenerator::instance().get(),
                                                                        true)));

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;
std::cout << "     " << std::endl;
```

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

III. TESTS

1) Comparaison des deux process

Pas temporel : 10 s
Nombre d'échantillons : 10 000

<u>Call:</u>
Paramètres:

```
//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 = 100;
//Prix d'exercice
Real strike = 120;
//Taux de dividende
Spread q = 0.03;
//Taux d'interet
Rate r = 0.05;
//volatilite
Volatility vol = 0.15;
```

```
prix = option_1.NPV()=1.12554

MCEuropeanEngine avec GeneralizedBlackScholesProcess

Prix de l'option 1.13475
Temps d'execution: 0.09s
Erreur d'estimation 0.0409357


MCEuropeanEngine avec constantBlackScholesProcess

Prix de l'option  1.04638
Temps d'execution: 0.02s
Erreur d'estimation 0.0389865
```

Put:

Paramètres:

```
DayCounter dayCounter = Actual365Fixed();
//date du jour
Date t0(1, March, 2019);
//maturite
Date T(1, March, 2020);
//type d'option
Option::Type type(Option::Put);
//Sous jacent
Real stock_price = 100;
//Prix d'exercice
Real strike = 80;
//Taux de dividende
Spread q = 0.03;
//Taux d'interet
Rate r = 0.05;
//volatilite
Volatility vol = 0.15;
```

```
prix = option_1.NPV()=0.286122

MCEuropeanEngine avec GeneralizedBlackScholesProcess

Prix de l'option 0.273311
Temps d'execution: 0.10s
Erreur d'estimation 0.0143534


MCEuropeanEngine avec constantBlackScholesProcess

Prix de l'option  0.32883
Temps d'execution: 0.02s
Erreur d'estimation 0.0160807
```

Observations:

On constate que le temps d'exécution du process constantBlackScholesProcess est respectivement 80% (77%) inférieur pour le put (call) à celui de GeneralizedBlackScholesProcess.

On remarque que l'erreur d'estimation est approximativement la même à 0.1% près.

Pour nous permettre de comparer l’efficacité des deux classes, nous allons effectuer des tests en modifiant 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. Nous avons réalisé des tests sur plusieurs valeurs. Dans un souci de lisibilité, nous en présentons uniquement deux par critères.

2) Variation du nombre d’échantillons

On lance les process pour différentes valeurs du nombre d’échantillons:

Pas temporel : 10 s :

Call :

Nombre d’échantillons : 10 000:

```
prix = option_1.NPV()=1.12554

MCEuropeanEngine avec GeneralizedBlackScholesProcess

Prix de l'option 1.13475
Temps d'execution: 0.09s
Erreur d'estimation 0.0409357


MCEuropeanEngine avec constantBlackScholesProcess

Prix de l'option  1.04638
Temps d'execution: 0.02s
Erreur d'estimation 0.0389865
```

Nombre d’échantillons : 100 000

```
prix = option_1.NPV()=1.12554

MCEuropeanEngine avec GeneralizedBlackScholesProcess

Prix de l'option 1.13905
Temps d'execution: 0.91s
Erreur d'estimation 0.0133628


MCEuropeanEngine avec constantBlackScholesProcess

Prix de l'option  1.0865
Temps d'execution: 0.17s
Erreur d'estimation 0.0127943
```

Put :

nombre d’échantillons : 10 000:

```
prix = option_1.NPV()=0.286122

MCEuropeanEngine avec GeneralizedBlackScholesProcess

Prix de l'option 0.273311
Temps d'execution: 0.10s
Erreur d'estimation 0.0143534


MCEuropeanEngine avec constantBlackScholesProcess

Prix de l'option  0.32883
Temps d'execution: 0.02s
Erreur d'estimation 0.0160807
```

Nombre d’échantillons : 100 000:

```
prix = option_1.NPV()=0.286122

MCEuropeanEngine avec GeneralizedBlackScholesProcess

Prix de l'option 0.288461
Temps d'execution: 0.91s
Erreur d'estimation 0.00473418


MCEuropeanEngine avec constantBlackScholesProcess

Prix de l'option  0.305577
Temps d'execution: 0.17s
Erreur d'estimation 0.00492029
```

Observations:

On remarque que quelques soit le nombre d’échantillons, le temps d’exécution diminue d’environ 80% entre les deux process.
L’erreur d’estimation diminue respectivement de 75% (69%) pour le put (Call) quand le nombre d’échantillons est multiplié par 10.
Le temps d’exécution est multiplié par 9 quand on augmente le nombre d’échantillons.

3) Variation du pas temporel

On lance la simulation pour différentes valeurs du pas temporel. Par exemple 10s et 60s.

On note à chaque simulation la variation du temps de calcul et de la précision de l'estimation.
Nombre d'échantillons : 10 000:

Call :
Pas temporel : 10 s:

```
prix = option_1.NPV()=1.12554

MCEuropeanEngine avec GeneralizedBlackScholesProcess

Prix de l'option 1.13475
Temps d'execution: 0.09s
Erreur d'estimation 0.0409357


MCEuropeanEngine avec constantBlackScholesProcess

Prix de l'option  1.04638
Temps d'execution: 0.02s
Erreur d'estimation 0.0389865
```

Pas temporel : 60s :

```
prix = option_1.NPV()=1.12554

MCEuropeanEngine avec GeneralizedBlackScholesProcess

Prix de l'option 1.1722
Temps d'execution: 0.50s
Erreur d'estimation 0.0424082


MCEuropeanEngine avec constantBlackScholesProcess

Prix de l'option  1.10739
Temps d'execution: 0.08s
Erreur d'estimation 0.0413278
```

Put :

Pas temporel : 10s

```
prix = option_1.NPV()=0.286122

MCEuropeanEngine avec GeneralizedBlackScholesProcess

Prix de l'option 0.273311
Temps d'execution: 0.10s
Erreur d'estimation 0.0143534


MCEuropeanEngine avec constantBlackScholesProcess

Prix de l'option  0.32883
Temps d'execution: 0.02s
Erreur d'estimation 0.0160807
```

Pas temporel : 60 s:

```
prix = option_1.NPV()=0.286122

MCEuropeanEngine avec GeneralizedBlackScholesProcess

Prix de l'option 0.255502
Temps d'execution: 0.50s
Erreur d'estimation 0.0137993


MCEuropeanEngine avec constantBlackScholesProcess

Prix de l'option  0.280136
Temps d'execution: 0.08s
Erreur d'estimation 0.0145879
```

Observations:

Le temps d'exécution est multiplié approximativement par 5 pour les types d'options quand on augmente le pas temporel.

L'erreur stagne aux alentours respectivement de 0.015 (0.04) pour le put (call).

IV. CONCLUSION

En comparant les différentes simulations, nous constatons que le temps d'exécution pour le constantBlackScholesProcess est nettement plus faible que pour celui de GeneralizedBlackScholesProcess (environ 80% inférieur).

Les erreurs d'estimation sont quasiment identiques pour les deux process.

On en déduit que l'utilisation de constantBlackScholesProcess est plus efficiente dans le cadre de la simulation du Monte Carlo engine, et cela reste valable même en faisant varier les paramètres de simulation (nombre d'échantillons et pas temporel).