# 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 la valeur de plusieurs paramètres relatifs à une option (call ou put). Les paramètres considérés sont: sous-jacent, taux sans risque, taux de rendement et volatilité.
QuantLib se sert d’une classe GeneralizedBlackScholesProcess pour prévoir l’évolution de ces paramètres.

L’objectif de ce projet est de créer une nouvelle classe ConstantBlackScholesProcess qui considère les paramètres définis ci-dessus comme constants et de comparer les résultats obtenus avec ceux de la classe GeneralizedBlackScholesProcess en terme de valeur, précision et temps de calcul.

# II. CODE

## 1) ConstantBlackSholesProcess

Nous avons créé une nouvelle classe ConstantBlackScholesProcess, inspirée de la classe GeneralizedBlackScholesProcess mais à paramètres constants.

# 2) MCEuropeanEngine

Nous avons modifié la classe MCEuropeanEngine en ajoutant un boolean permettant de choisir quelle classe utiliser entre ConstantBlackScholesProcess et GeneralizedBlackScholesProcess. Lorsque le boolean est dans l'état "false", la simulation de Monte Carlo est lancée avec la classe initiale. Si le boolean est dans l'état "true", l'algorithme prend en compte la classe ConstantBlackScholesProcess.

```
80      bool constant_; //! définition de l'attribut boolean
81      boost::shared_ptr<GeneralizedBlackScholesProcess> process_;
82      bool antithetic_;
83      Size steps_, stepsPerYear_, samples_, maxSamples_;
84      Real tolerance_;
85      bool brownianBridge_;
86      BigNatural seed_;
87
88    public: //! méthode pathGenerator() reprise de la classe MCVanillaEngine
89      boost::shared_ptr<path_generator_type> pathGenerator() const {
90          Size dimensions = process_->factors();
91          TimeGrid grid = this->timeGrid();
92          typename RNG::rsg_type generator =
93                  RNG::make_sequence_generator(dimensions*(grid.size()-1),seed_);
94          if (this->constant_ ){
95              return boost::shared_ptr<path_generator_type>(
96                  new path_generator_type(process_, grid,
97                  generator, brownianBridge_));
98          }
99
100
101
102         else{
103             boost::shared_ptr<GeneralizedBlackScholesProcess> process =boost::dynamic_pointer_cast<GeneralizedBlackScholesProcess>(this->process_);
104             boost::shared_ptr<PlainVanillaPayoff> payoff = boost::dynamic_pointer_cast<PlainVanillaPayoff>(this->arguments_.payoff);
105                     QL_REQUIRE(payoff, "non-plain payoff given");
106
107
108             return boost::shared_ptr<path_generator_type>(
109                                     new path_generator_type(boost::shared_ptr<constantBlackScholesProcess> (new constantBlackScholesProcess(process->stateVariable(),
110                                     this->arguments_.exercise->lastDate(),
111                                     payoff->strike(),
112                                     process->riskFreeRate(),
113
114                                     process->blackVolatility(),
115                                     process->dividendYield()
116                                                                     )),grid,
117                                         generator, brownianBridge_)
118
119                                       );
120                     }
121
122
123         }
```

Le boolean que l'on a appelé "constant_" est défini ligne 80.
La boucle "if" de la ligne 94 permet de tester l'état du boolean et de choisir la classe qu'il exécutera.

```cpp
#include <ql/time/calendars/nullcalendar.hpp>
#include <ql/time/daycounters/actual365fixed.hpp>
#include "constantBlackScholesProcess.hpp"

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;
    }


}
```

Dans la classe constantBlackScholesProcess, les valeurs “strike” et “exercie_Date” sont constantes contrairement à leur définition dans la classe GeneralizedBlackScholesProcess. De plus, les fonctions “drift” et “diffusion” sont très simples comme on peut le voir aux lignes 62 et 66. Elles ne modifient pas les valeurs “risk_drift” et “diffusion_” qui sont calculées dans le constructeur.

Cela permet d’économiser le temps de calcul.

# 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 la fréquence d'échantillonnage.

## 1) Variation du nombre d'échantillons

## 2) Variation de la fréquence d'échantillonnage

# IV. CONCLUSION

On observe, suite aux tests, que l'utilisation de la classe ConstantBlackScholesProcess permet d'obtenir des résultats similaires en précision à ceux obtenus avec GeneralizedBlackScholesProcess. Cependant, ConstantBlackScholesProcess permet d'obtenir les résultats avec un temps de calcul nettement inférieur.