The use of constant parameters in Monte Carlo engines

Issam Nasser
Marouane Idlimam
William Sauvé
Saad Belkhadir Mellas

Luigi BALLABIO

27 février 2017

Table des matières

Table des figures

1 Introduction

1.1 Background

In Monte Carlo engines, repeated calls to the process methods may cause a performance hit ; especially when the process is an instance of the GeneralizedBlackScholesProcess class, whose methods in turn make expensive method calls to the contained term structures.

The performance of the engine can be increased at the expense of some accuracy.We are creating a new class that models a Black-Scholes process with constant parameters (underlying value, risk-free rate, dividend yield, and volatility) ; then we modify the MCEuropeanEngine class so that it still takes a generic Black-Scholes process and an additional boolean parameter. If the boolean is false, the engine runs as usual ; if it is true, the engine extracts the constant parameters from the original process (based on the exercise date of the option ; for instance, the constant risk-free rate should be the zero-rate of the full risk-free curve at the exercise date) and runs the Monte Carlo simulation with an instance of the constant process.

1.2 Black-Scholes process with constant parameters

Here is an sample of constantBlackScholesModel code :

```
//! \name 1-D stochastic process interface
//@{
//! returns the initial value of the state variable

class constantBlackScholesModel : public StochasticProcess1D {
  /*! This class describes the stochastic process \f$ S \f$ governed by
  \f[
  d\ln S(t) = (r(t) - q(t) ) dt + \sigma dW_t.
  \f]
  */
private:
  Handle<Quote> underlyingValue_;
  Handle<YieldTermStructure> riskFreeRate_;
  Handle<YieldTermStructure>  dividendYield_;
  Handle<BlackVolTermStructure> blackVolatility_;
  Real driftC_;
  Real diffusionC_;
  Date exerciceDate_;
public:

  constantBlackScholesModel(
    const Handle <Quote>  underlyingValue,
    const Date exerciceDate,
    const Handle<YieldTermStructure>& riskFree,
    const Handle<BlackVolTermStructure>& blackVol,
    const Handle<YieldTermStructure>& dividendYield,
    boost::shared_ptr<discretization>& disc );

  Real drift(Time t, Real x) const;

  Real diffusion(Time t, Real x) const;


  Real variance(const StochasticProcess1D&,
    Time t0, Real x0, Time dt) const;

  Real x0() const;
};
```

A few interesting information are about drift and diffusion methods, that model the process :

$$\frac{dS}{S} = (r(t)) - q(t)) * dt + \sigma(t, x) * dW$$

the diffusion and drift are calculated inside the constructor, if it was the case we will end up doing a number of traits a lot of times and there would be no advantage in using a constant process.Then we store the result, so that we can simply return them from the methods.

```
constantBlackScholesModel::constantBlackScholesModel(
    const Handle<Quote> underlyingValue,
    const Date exerciceDate,
    const Handle<YieldTermStructure>& riskFree,
    const Handle<BlackVolTermStructure>& blackVol,
    const Handle<YieldTermStructure>& dividendYield,
    boost::shared_ptr<discretization>& disc)
    :StochasticProcess1D(disc), underlyingValue_(underlyingValue),
    riskFreeRate_(riskFree), dividendYield_(dividendYield), blackVolatility_(
    blackVol) {
    exerciceDate_ = exerciceDate;
    driftC_ = riskFreeRate_->zeroRate(exerciceDate_, riskFreeRate_->dayCounter
    (), Continuous,
      NoFrequency, true) - dividendYield_->zeroRate(exerciceDate_,
    riskFreeRate_->dayCounter(), Continuous,
        NoFrequency, true);
    diffusionC_ = blackVolatility_->blackVol(exerciceDate_, underlyingValue->
    value());
  }

  Real constantBlackScholesModel::drift(Time t, Real x) const {
    return driftC_*x;
  }

  Real constantBlackScholesModel::diffusion(Time t, Real x) const {
    return diffusionC_*x;
  }
```

Then we modify the MCEuropeanEngine class so it still takes a genericBlack-Scholesprocess and an additional boolean parameter like described bellow :

```
boost::shared_ptr<path_generator_type> pathGenerator() const {
      Size dimensions = process_->factors();

      TimeGrid grid = this->timeGrid();
      typename RNG::rsg_type generator =
        RNG::make_sequence_generator(dimensions*(grid.size() - 1), seed_);
      if (this->useConstantProcess_) {
      boost::shared_ptr<GeneralizedBlackScholesProcess> process =
        boost::dynamic_pointer_cast<GeneralizedBlackScholesProcess>(
          this->process_);
        return boost::shared_ptr<path_generator_type>(
          new path_generator_type(

            boost::shared_ptr<constantBlackScholesModel>(
              new constantBlackScholesModel(process->stateVariable(), this->
    arguments_.exercise->lastDate(),process->riskFreeRate(),

                process->blackVolatility(),process->dividendYield(), boost::
    shared_ptr<StochasticProcess1D::discretization>(new EulerDiscretization)))
    ,
            grid,
            generator,
            brownianBridge_)
          );

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

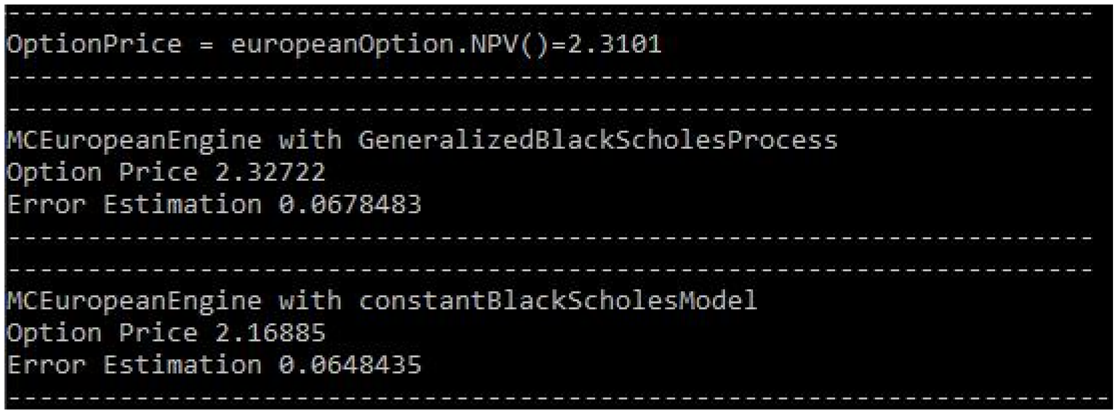
2 Result

Comparing the results(value,accuracy,time steps, number of samples) .

```
// constructor
  MCEuropeanEngine_2(
    const boost::shared_ptr<GeneralizedBlackScholesProcess>& process,
    Size timeSteps,
    Size timeStepsPerYear,
    bool brownianBridge,
    bool antitheticVariate,
    Size requiredSamples,
    Real requiredTolerance,
    Size maxSamples,
    BigNatural seed);
```

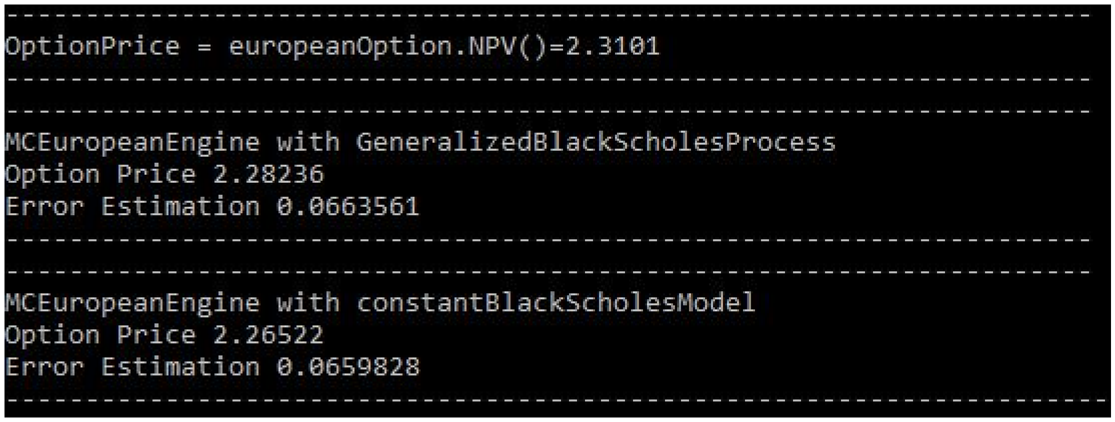
```
//Specification
    //Specify (1) STrike,(2) current stock price,(3)date count convention,(4)
    current date
    //(5) Maturity Dtae, (6) interest Rate, (7) dividend Yield, (8) Calendar
    for specific market
    //(9) Volatility, (10) Option Type
    Real strike = 100;
    Real underlying = 90;
    DayCounter dayCounter = Actual365Fixed();
    Date settlementDate(28, February, 2017);
    Date excerciceDate(28, February, 2018);
    Rate riskFreeRate = 0.05;
    Spread dividendYield = 0.00;
    Calendar calendar = TARGET();
    Volatility volatility = 0.20;
    Option::Type type(Option::Call);
```

2.1 Varying time steps



```
------------------------------------------------------------------
OptionPrice = europeanOption.NPV()=2.3101
------------------------------------------------------------------
------------------------------------------------------------------
MCEuropeanEngine with GeneralizedBlackScholesProcess
Option Price 2.32722
Error Estimation 0.0678483
------------------------------------------------------------------
------------------------------------------------------------------
MCEuropeanEngine with constantBlackScholesModel
Option Price 2.16885
Error Estimation 0.0648435
------------------------------------------------------------------
```

FIGURE 1 – Time steps =10 required Samples =10 000

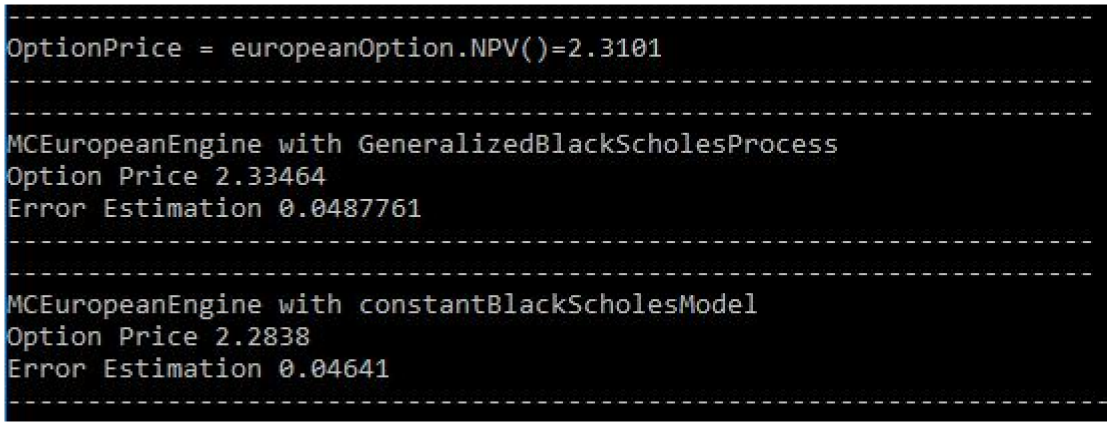


```
------------------------------------------------------------------
OptionPrice = europeanOption.NPV()=2.3101
------------------------------------------------------------------
------------------------------------------------------------------
MCEuropeanEngine with GeneralizedBlackScholesProcess
Option Price 2.28236
Error Estimation 0.0663561
------------------------------------------------------------------
------------------------------------------------------------------
MCEuropeanEngine with constantBlackScholesModel
Option Price 2.26522
Error Estimation 0.0659828
------------------------------------------------------------------
```

Figure 2 – Time steps =100 required Samples =10 000

The error of estimation is in both cases better with the black-Scholes Precess with Constant parameters. The execution Time Too.

2.2 Varying required samples size

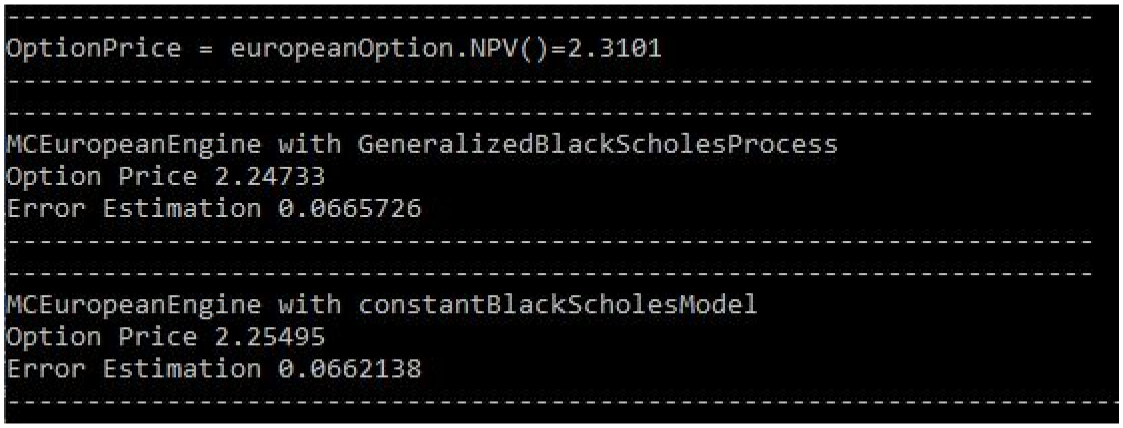


```
------------------------------------------------------------------
OptionPrice = europeanOption.NPV()=2.3101
------------------------------------------------------------------
------------------------------------------------------------------
MCEuropeanEngine with GeneralizedBlackScholesProcess
Option Price 2.33464
Error Estimation 0.0487761
------------------------------------------------------------------
------------------------------------------------------------------
MCEuropeanEngine with constantBlackScholesModel
Option Price 2.2838
Error Estimation 0.04641
------------------------------------------------------------------
```

Figure 3 – Time steps =10 required Samples =20 000

Still in this case, the new method perform better.

2.3 Adding a brownianBridge



```
---------------------------------------------------------------------
OptionPrice = europeanOption.NPV()=2.3101
---------------------------------------------------------------------
---------------------------------------------------------------------
MCEuropeanEngine with GeneralizedBlackScholesProcess
Option Price 2.24733
Error Estimation 0.0665726
---------------------------------------------------------------------
---------------------------------------------------------------------
MCEuropeanEngine with constantBlackScholesModel
Option Price 2.25495
Error Estimation 0.0662138
---------------------------------------------------------------------
```

FIGURE 4 – Time steps =10 required Samples =10 000 brownianBridge=true

Still in this case, the new method performs better overall.

3 Conclusion

In this solution, we have constant rates, we use less calls for calculating drift and diffusion, so it is natural that the execution time is better, but at the same time we observe no relevant differences between the two implementations besides little variation of the error of estimation.