Quantlib Project - IMT Atlantique 2022

Nicolas Gouraud
Cyprien Le Rouge de Guerdavid

Link to our Github : https://github.com/CyprienLRG/IMT2022/tree/main/project1

Project 1 : Use constant parameters in Monte Carlo engines

Introduction :

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. Create a new class that models a Black-Scholes process with constant parameters (underlying value, risk-free rate, dividend yield, and volatility); then 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.
Compare the results (value, accuracy, elapsed time) obtained with and without constant parameters and discuss.

Implementation :

First of all, in order to modify the monte-carlo method, we need to understand how it works.

Briefly, this method generates a large number of random and possible paths for the underlying via simulation, and then calculates the payoff of each path. Finally, an average of all prices is done and the price is discounted to today. (source : wikipedia).

To generate those paths, a stochastic process is needed. It is there that we can modify the original implementation by no longer using evolutive parameters but constant in this Black Scholes Process.

That's why we create that class below :

```hpp
#ifndef CONSTANTBLACKSCHOLESPROCESS_H
#define CONSTANTBLACKSCHOLESPROCESS_H
#include <ql/stochasticprocess.hpp>

namespace QuantLib {

    class ConstantBlackScholesProcess : public StochasticProcess1D {

        //your implementation goes here

        public :
            ConstantBlackScholesProcess(double underlyingValue, double riskFreeRate, double volatility,
                double dividend);

            Real x0() const;
            Real drift(Time t, Real x) const;
            Real diffusion(Time t, Real x) const;
            Real apply(Real x0, Real dx) const;

        private :
            double underlying_value;
            double risk_free_rate;
            double volatility;
            double dividend;
    };

}

#endif
```

Figure 1 : ConstantBlackScholesProcess.hpp

In this class we consider 4 parameters which are the underlying value, the risk free rate, the volatility and the dividend. Moreover, we need to redefine some methods (x0(), drift(Time t, Real x), diffusion (Time t, Real x) and apply(Real x0, Real dx)) inherited from StochasticProcess1D class because they are used in the path generator through the method evolve.

This method evolve uses a method apply, that we have redefined, and two others that use drift, diffusion and x0 functions. They are presented below.

```
Disposable<Array> StochasticProcess::expectation(Time t0,
                                                 const Array& x0,
                                                 Time dt) const {
    return apply(x0, discretization_->drift(*this, t0, x0, dt));
}

Disposable<Matrix> StochasticProcess::stdDeviation(Time t0,
                                                   const Array& x0,
                                                   Time dt) const {
    return discretization_->diffusion(*this, t0, x0, dt);
}

Disposable<Array> StochasticProcess::evolve(
                                          Time t0, const Array& x0,
                                          Time dt, const Array& dw) const {
    return apply(expectation(t0,x0,dt), stdDeviation(t0,x0,dt)*dw);
}
```

Figure 2 : Important Methods in StochasticProcess1D class

All this conducts us to redefine methods that you'll find below in the ConstantBlackScholesProcess.cpp figure.

```
#include "constantblackscholesprocess.hpp"
#include <iostream>

#include <ql/processes/eulerdiscretization.hpp>

namespace QuantLib {

    ConstantBlackScholesProcess::ConstantBlackScholesProcess(
                                        double underlyingValue,
                                        double riskFreeRate,
                                        double volatility_,
                                        double dividend_)
                                            :StochasticProcess1D
                                            (ext::make_shared<EulerDiscretization>
                                            ())
    {underlying_value = underlyingValue;
    risk_free_rate = riskFreeRate;
    volatility = volatility_;
    dividend = dividend_;}

Real ConstantBlackScholesProcess::x0() const {
    return underlying_value;
}

Real ConstantBlackScholesProcess::drift(Time t, Real x) const {
    return risk_free_rate - dividend - 0.5*volatility*volatility;
}

Real ConstantBlackScholesProcess::diffusion(Time t, Real x) const {
    return volatility;
}

Real ConstantBlackScholesProcess::apply(Real x0, Real dx) const {
    return x0 * std::exp(dx);
}

}
```

Figure 3 : ConstantBlackScholesProcess.cpp

Finally we need to modify the mceuropeanengine class to add a boolean parameter which permits to precise, in the constructor of MCEuropeanEngine_2, if we want to execute the Monte-Carlo method with or without constant Black-Scholes parameter.

```
MCEuropeanEngine_2(
        const boost::shared_ptr<GeneralizedBlackScholesProcess>& process,
        Size timeSteps,
        Size timeStepsPerYear,
        bool brownianBridge,
        bool antitheticVariate,
        Size requiredSamples,
        Real requiredTolerance,
        Size maxSamples,
        BigNatural seed,
        bool constantParameters); // add of a boolean which tells if parameters in BS
            are constant or not
```

Figure 4 : Constructor of MCEuropeanEngine2

Moreover, we have to add a verification of this boolean parameter and, in this way, associate the correct values of the Black-Scholes process. If it is true, we run the engine with our new class ConstantBlackScholesParameters. If it is false, we use the engine as usual. To do that we need to override the path generator method inherited from the MCVanillaEngine Class.

```
ext::shared_ptr<path_generator_type> pathGenerator() const override {

    Size dimensions = MCVanillaEngine<SingleVariate, RNG, S>::process_->factors();
    TimeGrid grid = this->timeGrid();
    typename RNG::rsg_type generator =
        RNG::make_sequence_generator(dimensions*(grid.size()-1),
        MCVanillaEngine<SingleVariate,RNG,S>::seed_);

    // If the parameters of the BS process are constant we call our new class
        ConstantBlackScholesParameters to evaluate parameters
    if (constantParameters_) {
        ext::shared_ptr<GeneralizedBlackScholesProcess> BS =
            ext::dynamic_pointer_cast<GeneralizedBlackScholesProcess>(this->process_);
        Time extractionTime = grid.back();

        double underlyingValue = BS -> x0();

        double strike =
            ext::dynamic_pointer_cast<StrikedTypePayoff>(MCVanillaEngine<SingleVariate,
            RNG,S>::arguments_.payoff)->strike();

        double volatility = BS->blackVolatility()->blackVol(extractionTime,strike);

        double riskFreeRate = BS->riskFreeRate()->zeroRate(extractionTime, Continuous);

        double dividend = BS->dividendYield()->zeroRate(extractionTime,Continuous);

        ext::shared_ptr<ConstantBlackScholesProcess> constBS(new
            ConstantBlackScholesProcess(underlyingValue,riskFreeRate,volatility,
            dividend));

        return ext::shared_ptr<path_generator_type>(new path_generator_type(constBS, grid
            generator, MCVanillaEngine<SingleVariate, RNG,S>::brownianBridge_));
    }

    else {
        return ext::shared_ptr<path_generator_type>(new
            path_generator_type(MCVanillaEngine<SingleVariate, RNG, S>::process_, grid,
            generator, MCVanillaEngine<SingleVariate, RNG,S>::brownianBridge_));
    }
}
```

Figure 5 : Override of the path generator in MCEurpeanEngine2

Finally, we can run the main class with our modifications. We can then compare both methods : with or without constant parameters.

Results :

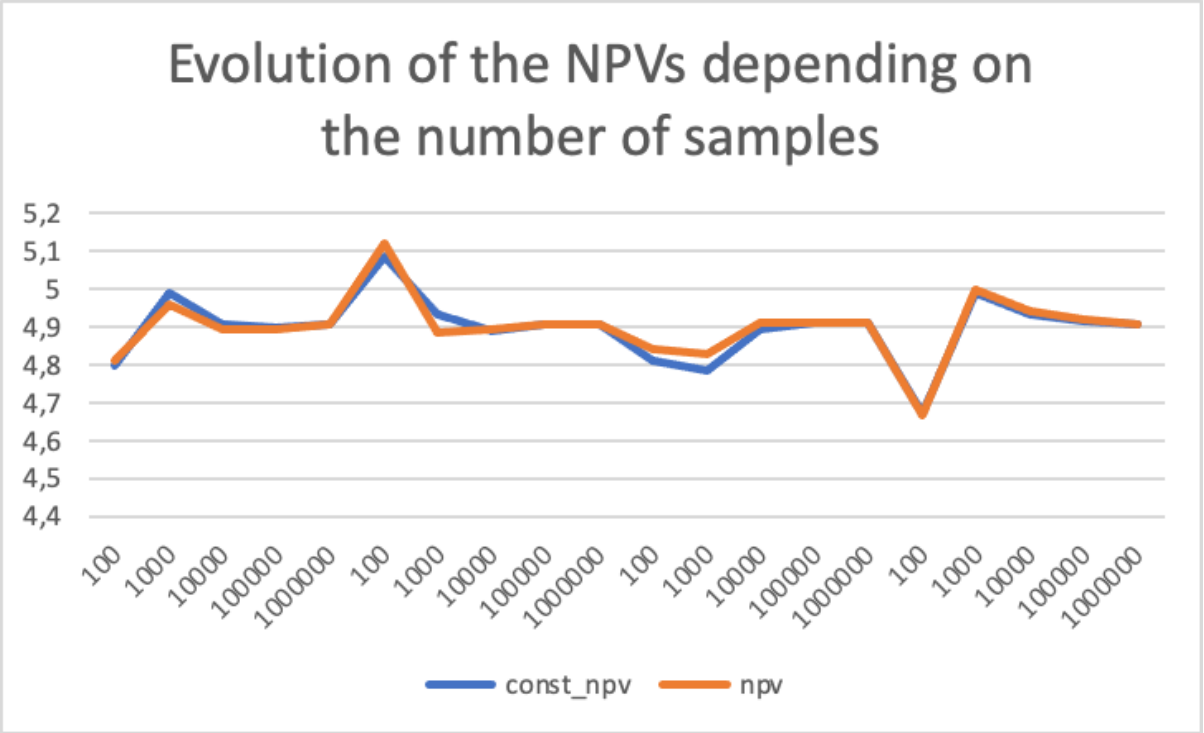
Parameters :

Figure 6 : parameters of our option

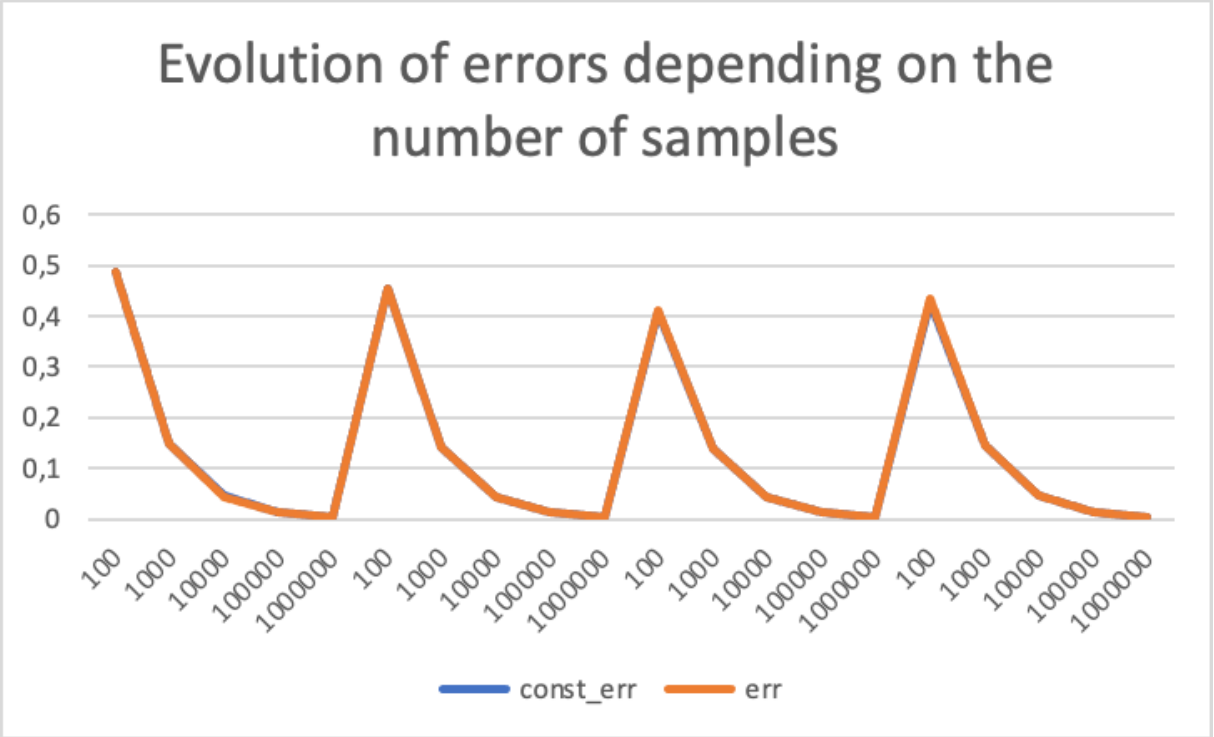
Results of the test :

step	sample	const_npv	const_err	const_time	npv	err	time
10	100	4,79904	0,488338	332	4,80919	0,48771	1104
10	1000	4,98782	0,148709	1317	4,96131	0,149273	6496
10	10000	4,90705	0,0453394	12596	4,89277	0,0452357	40055
10	100000	4,89782	0,0143312	60510	4,89459	0,0143305	306655
10	1000000	4,90756	0,0045334	565241	4,90605	0,00453322	2,97E+06
100	100	5,08521	0,452807	443	5,11977	0,454478	2598
100	1000	4,93454	0,142048	4124	4,88636	0,141668	25049
100	10000	4,88916	0,0452306	42340	4,89365	0,0452289	315140
100	100000	4,90686	0,0143121	587259	4,90583	0,0143069	2,69E+06
100	1000000	4,90886	0,00453009	3,97E+06	4,90887	0,00452998	2.62786e+07
1000	100	4,81236	0,408234	3943	4,84189	0,412258	25535
1000	1000	4,7848	0,139671	40951	4,82833	0,139471	272848
1000	10000	4,89401	0,0450761	414957	4,91218	0,0452139	2,71E+06
1000	100000	4,90976	0,014322	4,22E+06	4,91121	0,0143145	2,79E+07
1000	1000000	4,90934	0,00453015	4,65E+07	4,90931	0,0045289	2,90E+08
10000	100	4,67066	0,427946	46500	4,66826	0,4356	277186
10000	1000	4,98812	0,14477	418501	4,99718	0,145044	2,72E+06
10000	10000	4,93225	0,0455466	4,34E+06	4,94029	0,0455877	2,83E+07
10000	100000	4,91763	0,014322	5,35E+07	4,91937	0,0143295	3,32E+08
10000	1000000	4,90599	0,00452795	5,04E+08	4,90651	0,00452728	3,03E+09

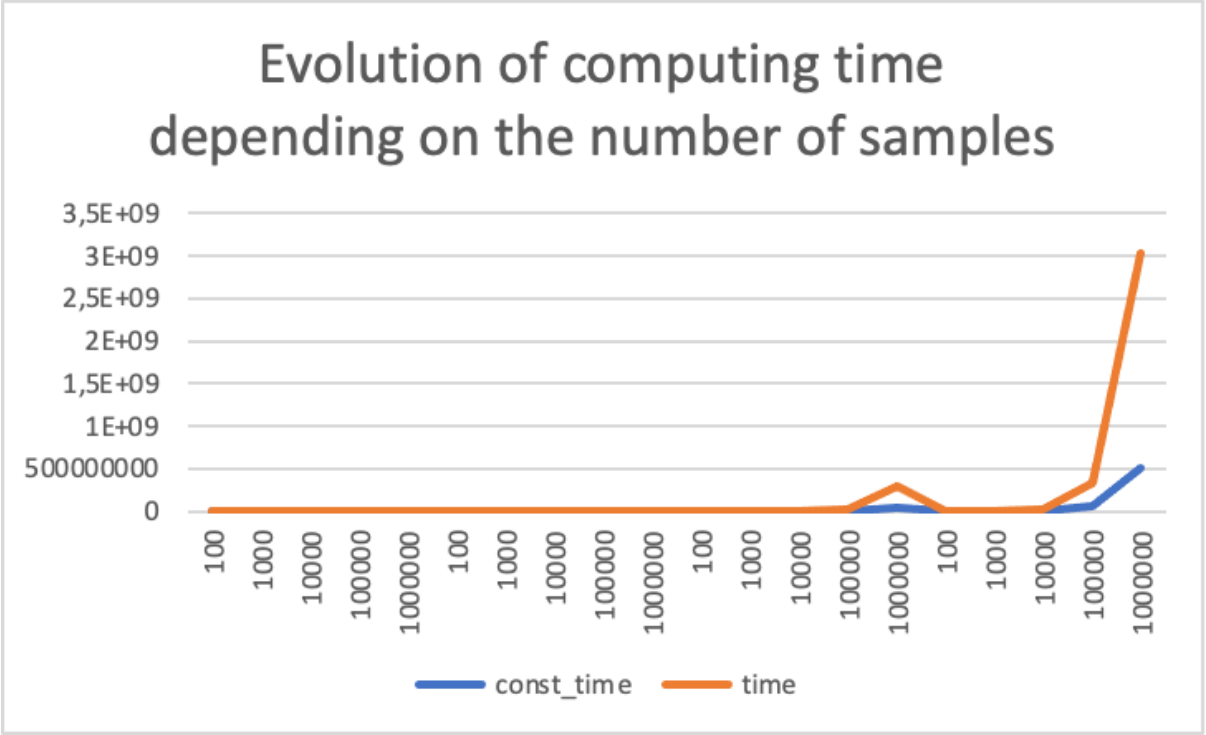
Let’s analyse our results :



We observe that both methods give results that are close. However, even if the differences are minimal, if we buy only one option this could be considered negligible. Nevertheless, we usually buy more than one option and by cumulating those gaps it could finally be paramount. Moreover, it seems that the test needs a minimum of 10.000 samples to converge to a correct estimation of the option price.



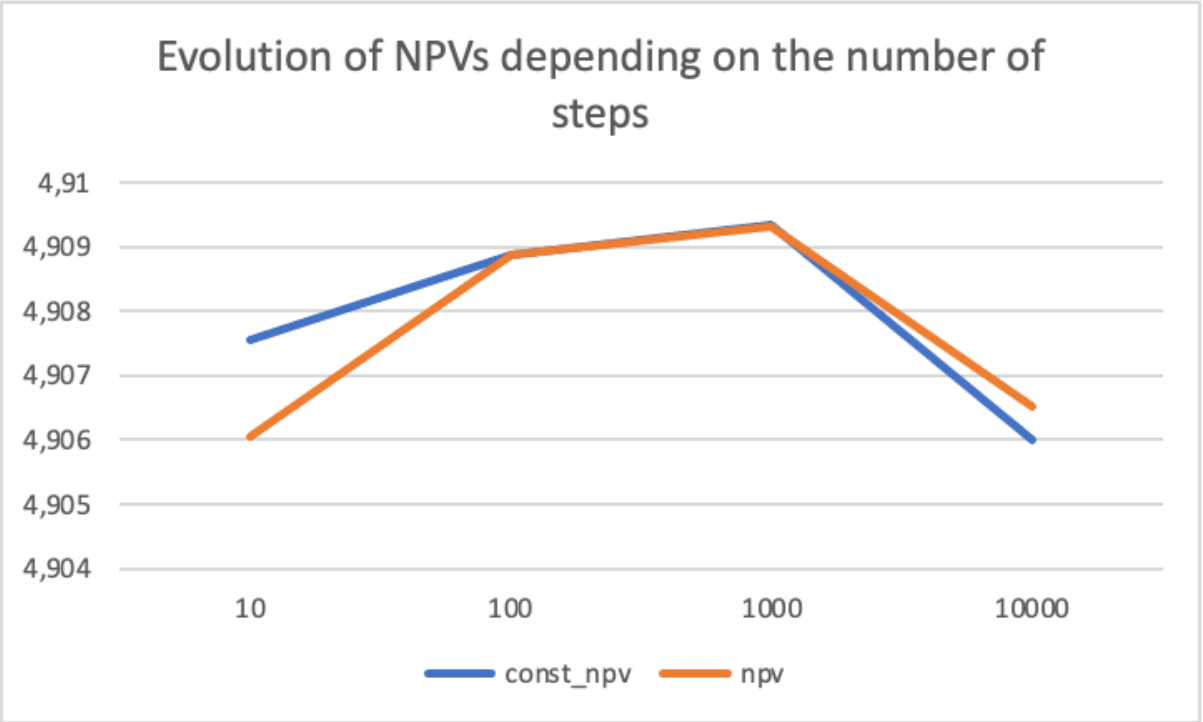
Concerning the accuracy of our methods, there are similar, which is satisfying. Moreover, it is interessant to notice that the more samples we have, the less the error is high. Also, those curves of errors confirm our intuition of the minimum of 10.000 to have a price closed to the real.



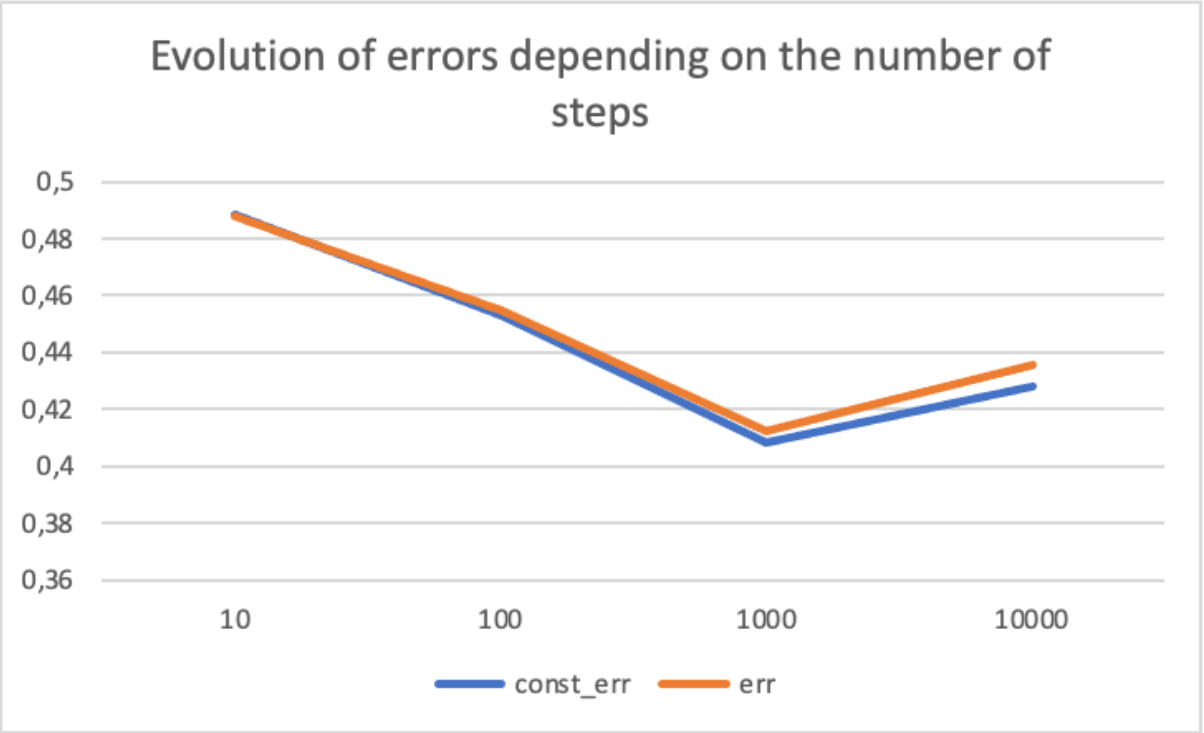
Regarding the computing time, we pinpoint that this constant parameter method is highly efficient. Indeed, the constant method is six times faster than the usual one for the last calculation. This is very interesting because it saves a lot of time to calculate with it.

Furthermore, this result needs to be compared with the error evolution. Indeed, even if it is better to have more samples, it increases sharply the computing time due to these, particularly for the usual method.

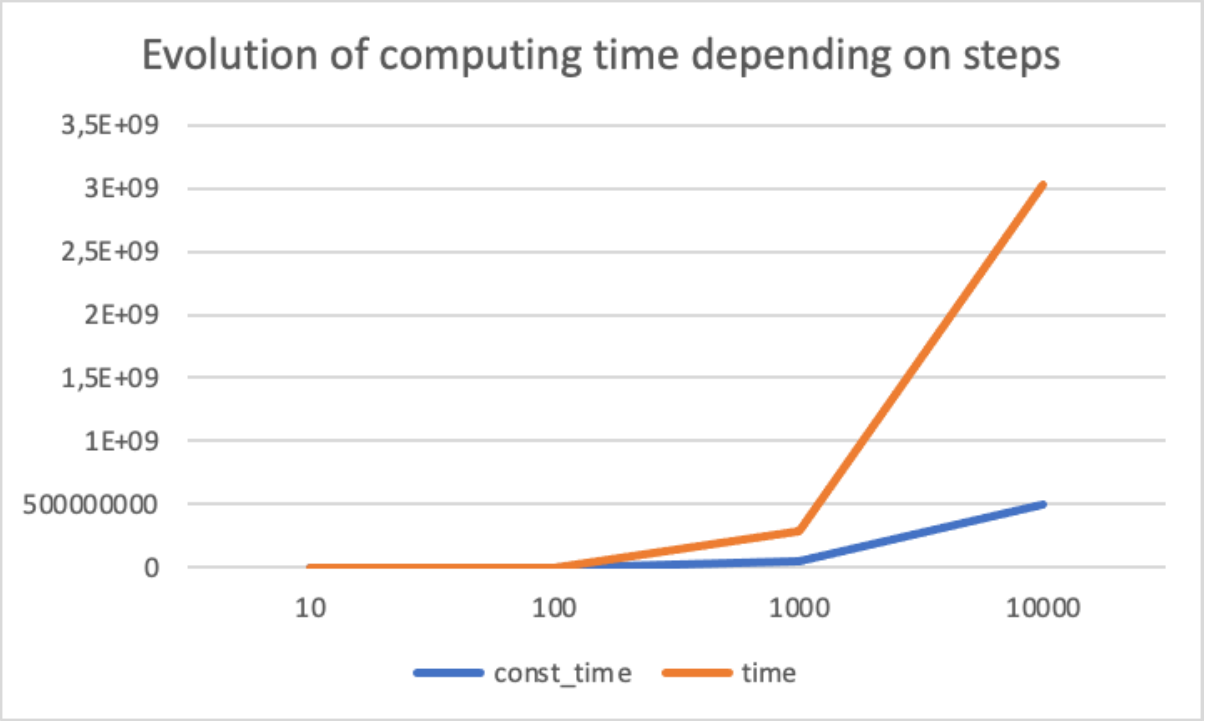
Finally, 100.000 samples seems to be an optimal value of our new method because it combines both accuracy and saving time-consuming.



This graph pinpoints how the number of steps can impact on the difference between the NPV values. We notice that between 100 and 1.000 steps, option prices seem to be the same. Let's see if the errors evolution shows a similar result.



Concerning the errors, it seems that they decrease to 1.000 steps. It can be considered, by simplifying, as a maximum we do not have to reach. Finally, let's see if this range of steps (between 100 and 1.000) is correct in time terms.



This graph highlights that after 1.000 steps time computing seems to be too important, even for the constant method. However, thanks to all of those results, we

find a range of step values, which is 100 to 1000 steps, to use for the constant parameters method. It combines both accuracy, and saving time-consuming to give a price close to the usual method.

Conclusion :

We have seen that this new class, ConstantBlackScholesParameters, permits to save time consuming without increasing errors by giving a very close and similar price to the usual method. Moreover, to use this new class optimally, we should use between 10.000 to 100.000 samples and between 100 to 1.000 steps.