Project 1: Use constant parameters in Monte Carlo engines

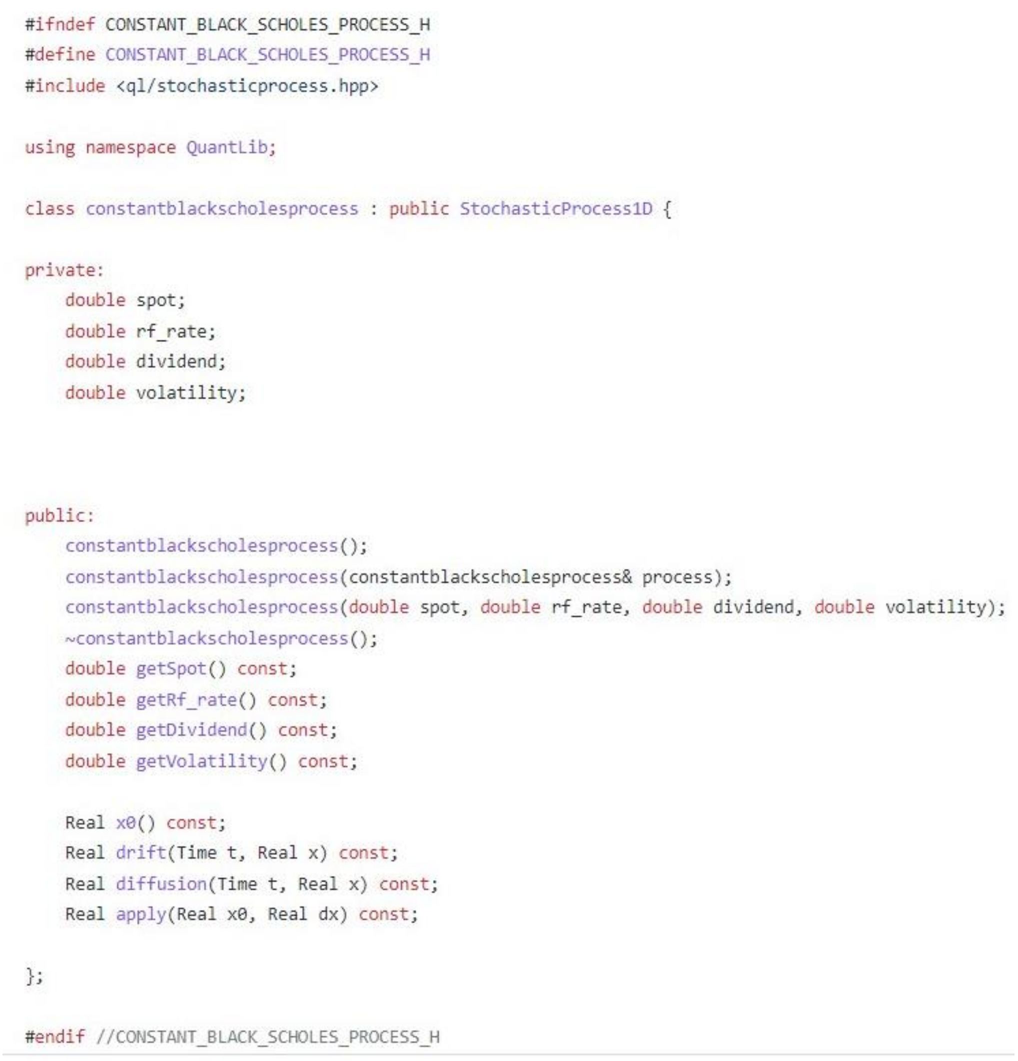
Yassine Khalsi
Benjamin Renault
Severine Guezello
Robin Clair

Presentation of the problem

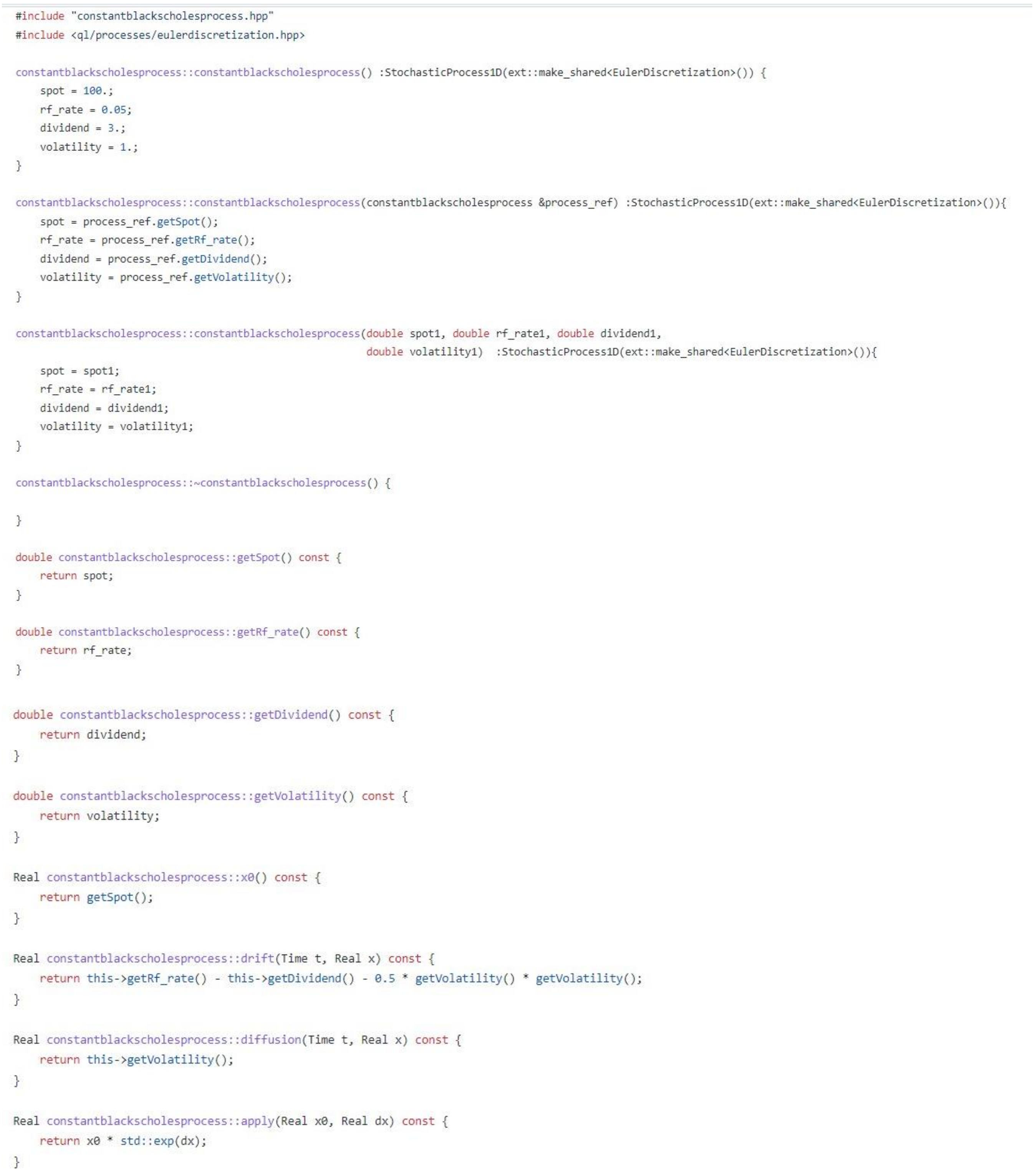
In Monte Carlo engines, repeated calls to the process methods may cause a performance hit, especially when the process is an instance of the GeneralizedBlackScholesProcess class,whose methods in turn make expensive method calls to the contained term structures.
The performance of the engine can be increased 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 Mote Carlo simulation with an instance of the constant process.
Compare the results (value, accuracy, elapsed time) obtained with and without constant parameters and discuss.

Our work on Quantlib:

First, we started by defining a new class: `constantblackscholesprocess` wich is supposed to model a Black Scholes process with constant parameters. We also overridden the methods we believe will be reused and useful for PathGenerator class:

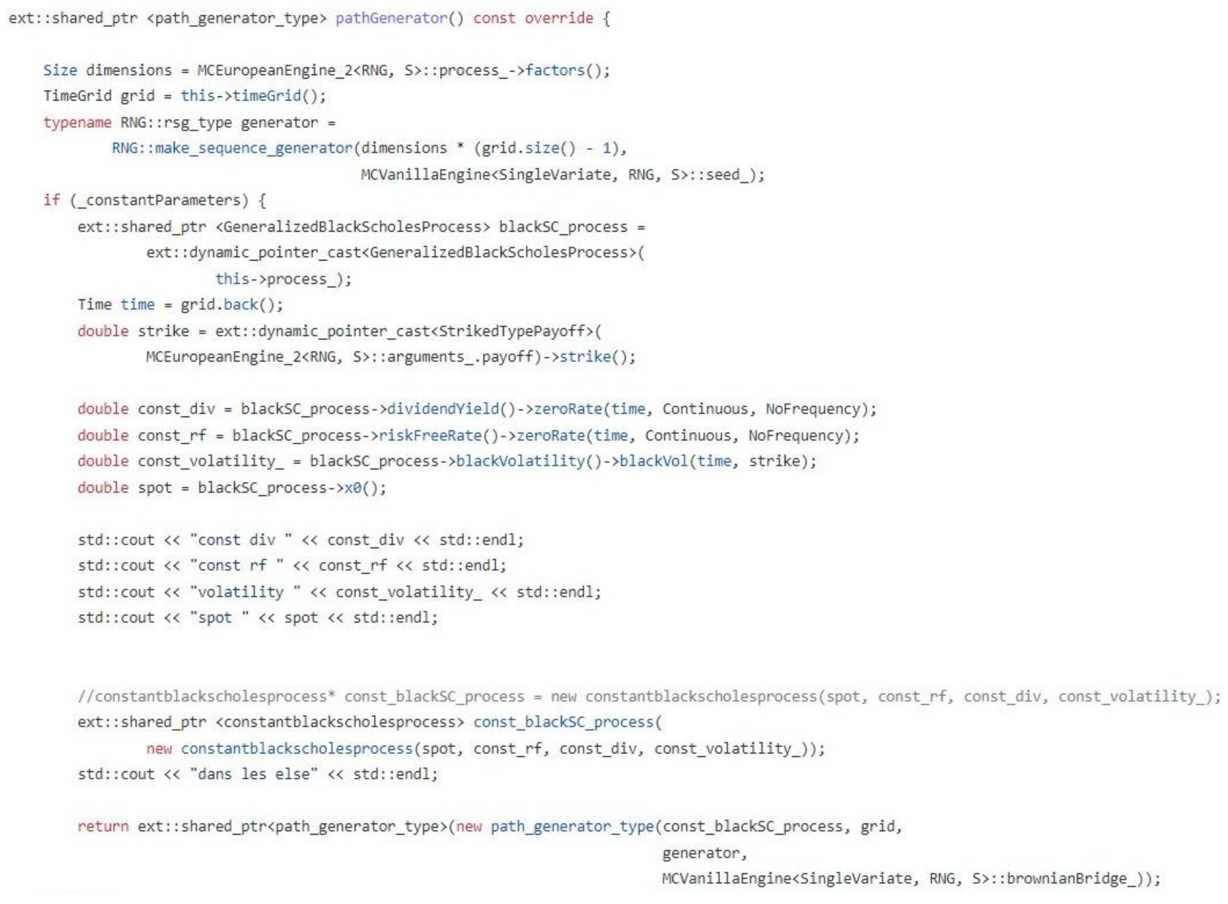


```cpp
#ifndef CONSTANT_BLACK_SCHOLES_PROCESS_H
#define CONSTANT_BLACK_SCHOLES_PROCESS_H
#include <ql/stochasticprocess.hpp>

using namespace QuantLib;

class constantblackscholesprocess : public StochasticProcess1D {

private:
    double spot;
    double rf_rate;
    double dividend;
    double volatility;



public:
    constantblackscholesprocess();
    constantblackscholesprocess(constantblackscholesprocess& process);
    constantblackscholesprocess(double spot, double rf_rate, double dividend, double volatility);
    ~constantblackscholesprocess();
    double getSpot() const;
    double getRf_rate() const;
    double getDividend() const;
    double getVolatility() const;

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

};

#endif //CONSTANT_BLACK_SCHOLES_PROCESS_H
```

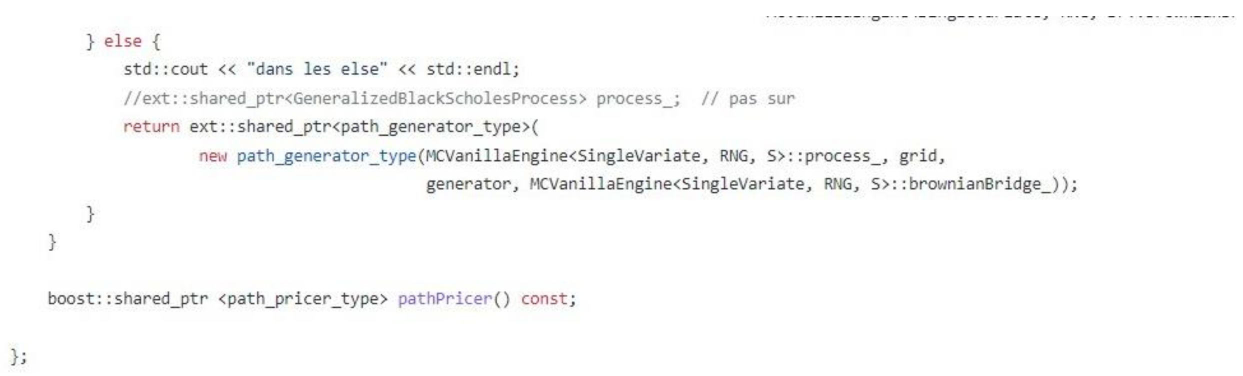


```cpp
#include "constantblackscholesprocess.hpp"
#include <ql/processes/eulerdiscretization.hpp>

constantblackscholesprocess::constantblackscholesprocess() :StochasticProcess1D(ext::make_shared<EulerDiscretization>()) {
    spot = 100.;
    rf_rate = 0.05;
    dividend = 3.;
    volatility = 1.;
}

constantblackscholesprocess::constantblackscholesprocess(constantblackscholesprocess &process_ref) :StochasticProcess1D(ext::make_shared<EulerDiscretization>()){
    spot = process_ref.getSpot();
    rf_rate = process_ref.getRf_rate();
    dividend = process_ref.getDividend();
    volatility = process_ref.getVolatility();
}

constantblackscholesprocess::constantblackscholesprocess(double spot1, double rf_rate1, double dividend1,
                                                         double volatility1)  :StochasticProcess1D(ext::make_shared<EulerDiscretization>()){
    spot = spot1;
    rf_rate = rf_rate1;
    dividend = dividend1;
    volatility = volatility1;
}

constantblackscholesprocess::~constantblackscholesprocess() {

}

double constantblackscholesprocess::getSpot() const {
    return spot;
}

double constantblackscholesprocess::getRf_rate() const {
    return rf_rate;
}

double constantblackscholesprocess::getDividend() const {
    return dividend;
}

double constantblackscholesprocess::getVolatility() const {
    return volatility;
}

Real constantblackscholesprocess::x0() const {
    return getSpot();
}

Real constantblackscholesprocess::drift(Time t, Real x) const {
    return this->getRf_rate() - this->getDividend() - 0.5 * getVolatility() * getVolatility();
}

Real constantblackscholesprocess::diffusion(Time t, Real x) const {
    return this->getVolatility();
}

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

Then, in the mceuropeanengine.cpp, we have overridden the path generator method based on whether we use the constant parameters or not:

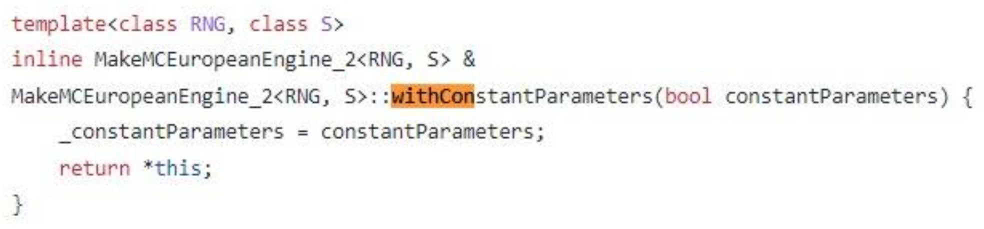


```
ext::shared_ptr <path_generator_type> pathGenerator() const override {

    Size dimensions = MCEuropeanEngine_2<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 (_constantParameters) {
        ext::shared_ptr <GeneralizedBlackScholesProcess> blackSC_process =
                ext::dynamic_pointer_cast<GeneralizedBlackScholesProcess>(
                        this->process_);
        Time time = grid.back();
        double strike = ext::dynamic_pointer_cast<StrikedTypePayoff>(
                MCEuropeanEngine_2<RNG, S>::arguments_.payoff)->strike();

        double const_div = blackSC_process->dividendYield()->zeroRate(time, Continuous, NoFrequency);
        double const_rf = blackSC_process->riskFreeRate()->zeroRate(time, Continuous, NoFrequency);
        double const_volatility_ = blackSC_process->blackVolatility()->blackVol(time, strike);
        double spot = blackSC_process->x0();

        std::cout << "const div " << const_div << std::endl;
        std::cout << "const rf " << const_rf << std::endl;
        std::cout << "volatility " << const_volatility_ << std::endl;
        std::cout << "spot " << spot << std::endl;


        //constantblackscholesprocess* const_blackSC_process = new constantblackscholesprocess(spot, const_rf, const_div, const_volatility_);
        ext::shared_ptr <constantblackscholesprocess> const_blackSC_process(
                new constantblackscholesprocess(spot, const_rf, const_div, const_volatility_));
        std::cout << "dans les else" << std::endl;

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



```
    } else {
        std::cout << "dans les else" << std::endl;
        //ext::shared_ptr<GeneralizedBlackScholesProcess> process_;  // pas sur
        return ext::shared_ptr<path_generator_type>(
                new path_generator_type(MCVanillaEngine<SingleVariate, RNG, S>::process_, grid,
                                        generator, MCVanillaEngine<SingleVariate, RNG, S>::brownianBridge_));
    }
}

boost::shared_ptr <path_pricer_type> pathPricer() const;

};
```

And finally to take into account the constant parameters or not, we added the following code lines:



```
template<class RNG, class S>
inline MakeMCEuropeanEngine_2<RNG, S> &
MakeMCEuropeanEngine_2<RNG, S>::withConstantParameters(bool constantParameters) {
    _constantParameters = constantParameters;
    return *this;
}
```

Results:

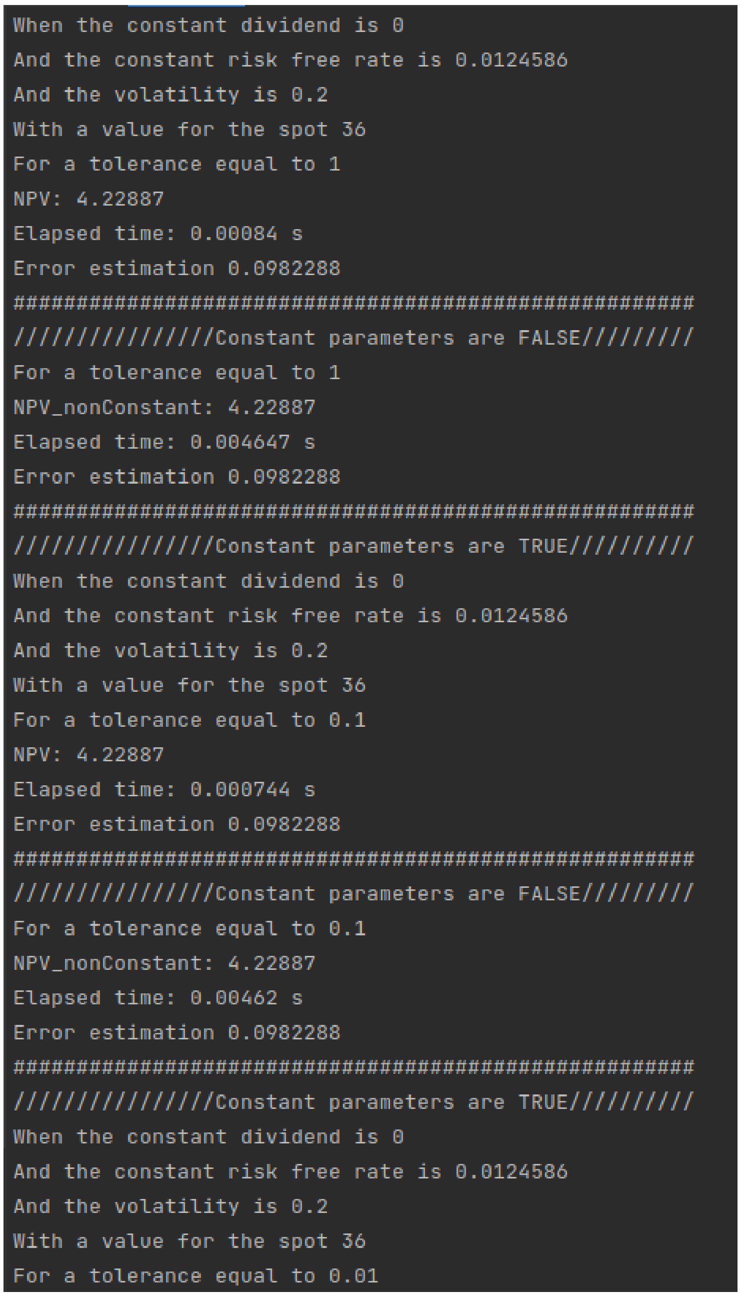
To test the ability of our engine to produce accurate results with less time than the engine with non-constant parameters, we calculated the put price NPV for different values of tolerance and different values of number of samples. We know that the tolerance actually define by itself the number of samples since the tolerance add samples until the required absolute tolerance is reached.

However, since shown below (They are also shown if we compile and execute the project):

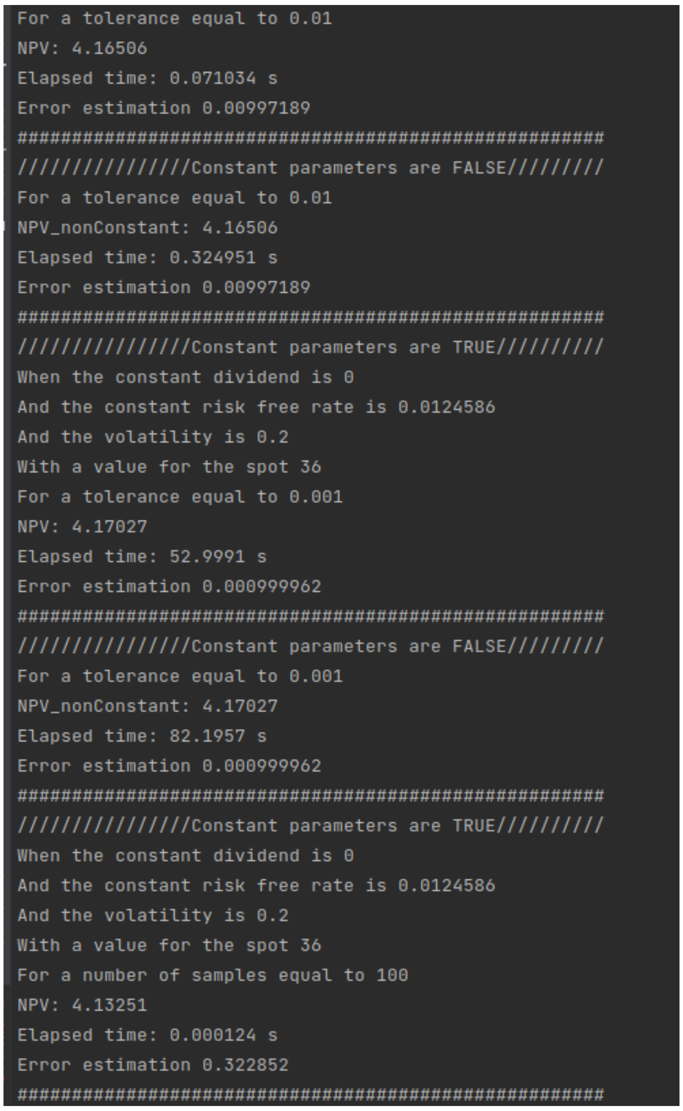
This is the price option - in the 24th of February 2022

- With an underlying value of 36
- A strike of 40
- And a maturity date 24th of May 2022

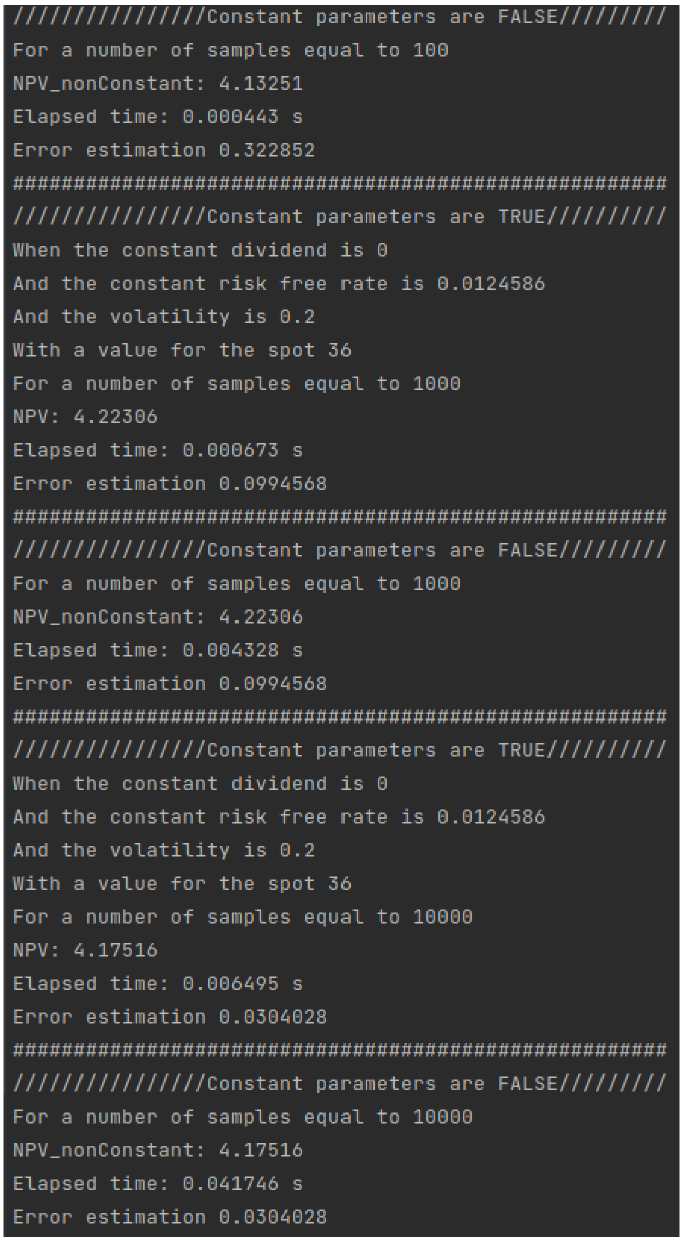
The number of samples seems more intuitive, we decided to test with different values of both.
The results are



```
When the constant dividend is 0
And the constant risk free rate is 0.0124586
And the volatility is 0.2
With a value for the spot 36
For a tolerance equal to 1
NPV: 4.22887
Elapsed time: 0.00084 s
Error estimation 0.0982288
######################################################
///////////////Constant parameters are FALSE/////////
For a tolerance equal to 1
NPV_nonConstant: 4.22887
Elapsed time: 0.004647 s
Error estimation 0.0982288
######################################################
///////////////Constant parameters are TRUE//////////
When the constant dividend is 0
And the constant risk free rate is 0.0124586
And the volatility is 0.2
With a value for the spot 36
For a tolerance equal to 0.1
NPV: 4.22887
Elapsed time: 0.000744 s
Error estimation 0.0982288
######################################################
///////////////Constant parameters are FALSE/////////
For a tolerance equal to 0.1
NPV_nonConstant: 4.22887
Elapsed time: 0.00462 s
Error estimation 0.0982288
######################################################
///////////////Constant parameters are TRUE//////////
When the constant dividend is 0
And the constant risk free rate is 0.0124586
And the volatility is 0.2
With a value for the spot 36
For a tolerance equal to 0.01
```



```
For a tolerance equal to 0.01
NPV: 4.16506
Elapsed time: 0.071034 s
Error estimation 0.00997189
######################################################
///////////////Constant parameters are FALSE/////////
For a tolerance equal to 0.01
NPV_nonConstant: 4.16506
Elapsed time: 0.324951 s
Error estimation 0.00997189
######################################################
///////////////Constant parameters are TRUE//////////
When the constant dividend is 0
And the constant risk free rate is 0.0124586
And the volatility is 0.2
With a value for the spot 36
For a tolerance equal to 0.001
NPV: 4.17027
Elapsed time: 52.9991 s
Error estimation 0.000999962
######################################################
///////////////Constant parameters are FALSE/////////
For a tolerance equal to 0.001
NPV_nonConstant: 4.17027
Elapsed time: 82.1957 s
Error estimation 0.000999962
######################################################
///////////////Constant parameters are TRUE//////////
When the constant dividend is 0
And the constant risk free rate is 0.0124586
And the volatility is 0.2
With a value for the spot 36
For a number of samples equal to 100
NPV: 4.13251
Elapsed time: 0.000124 s
Error estimation 0.322852
######################################################
```



```
///////////////Constant parameters are FALSE////////
For a number of samples equal to 100
NPV_nonConstant: 4.13251
Elapsed time: 0.000443 s
Error estimation 0.322852
######################################################
///////////////Constant parameters are TRUE/////////
When the constant dividend is 0
And the constant risk free rate is 0.0124586
And the volatility is 0.2
With a value for the spot 36
For a number of samples equal to 1000
NPV: 4.22306
Elapsed time: 0.000673 s
Error estimation 0.0994568
######################################################
///////////////Constant parameters are FALSE////////
For a number of samples equal to 1000
NPV_nonConstant: 4.22306
Elapsed time: 0.004328 s
Error estimation 0.0994568
######################################################
///////////////Constant parameters are TRUE/////////
When the constant dividend is 0
And the constant risk free rate is 0.0124586
And the volatility is 0.2
With a value for the spot 36
For a number of samples equal to 10000
NPV: 4.17516
Elapsed time: 0.006495 s
Error estimation 0.0304028
######################################################
///////////////Constant parameters are FALSE////////
For a number of samples equal to 10000
NPV_nonConstant: 4.17516
Elapsed time: 0.041746 s
Error estimation 0.0304028
```

The parameters are more readable this way:

Constant Parameters	TRUE

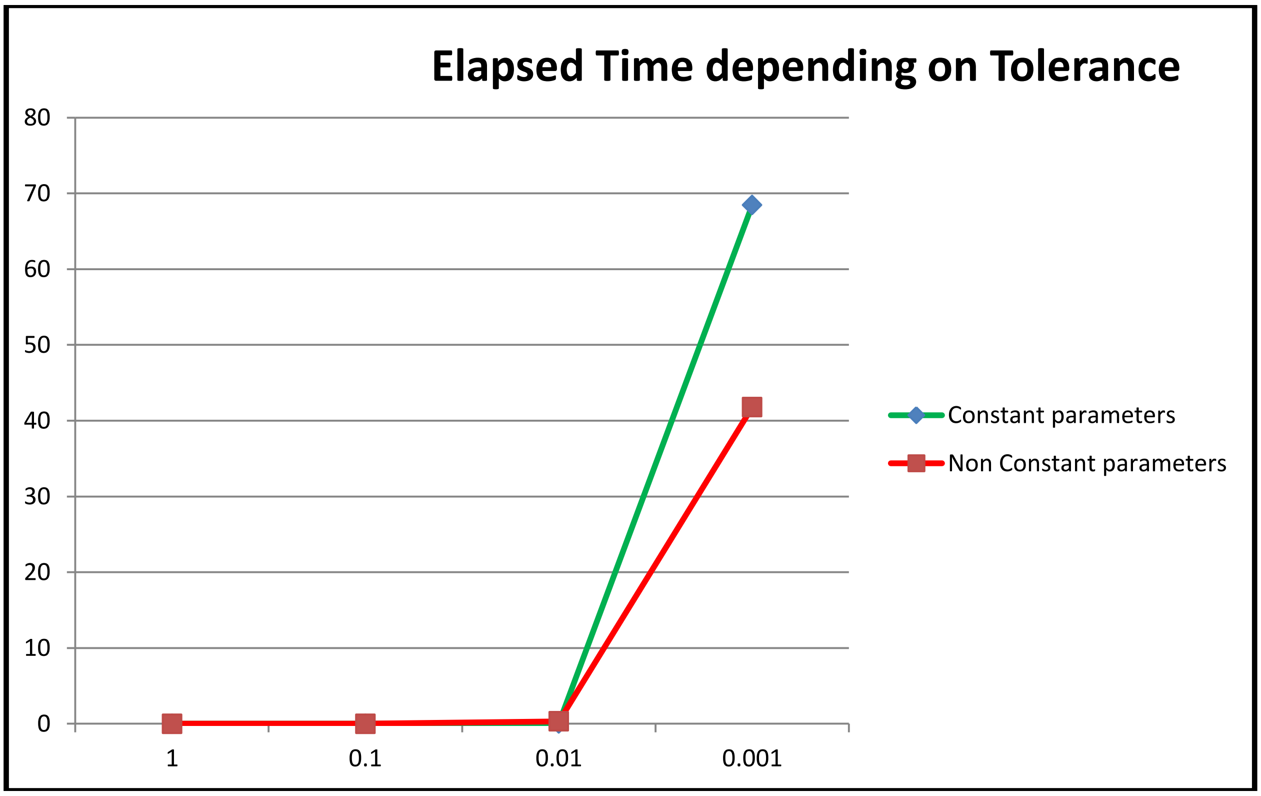
Tolerance	1	0.1	0.01	0.001	Sample	100	1000	10000
NPV	4.22887	4.22887	4.16506	4.17027		4.13251	4.22306	4.17516
Elapsed Time(s)	0.00097	0.000506	0.052265	68.465		0.000085	0.000453	0.005019
Accuracy	0.0982288	0.0982288	0.009971	0.00099		0.322852	0.994568	0.03040

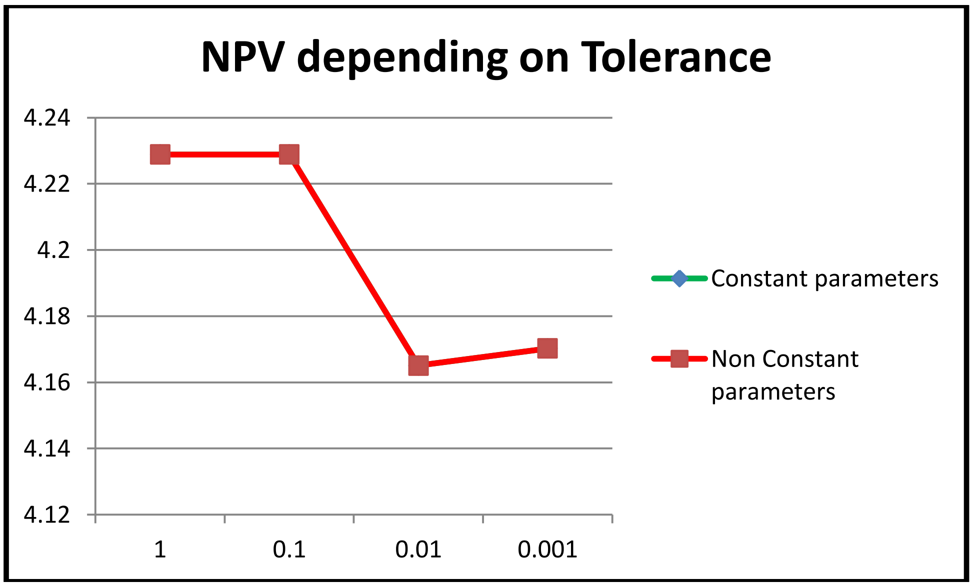
Constant Parameters	FALSE

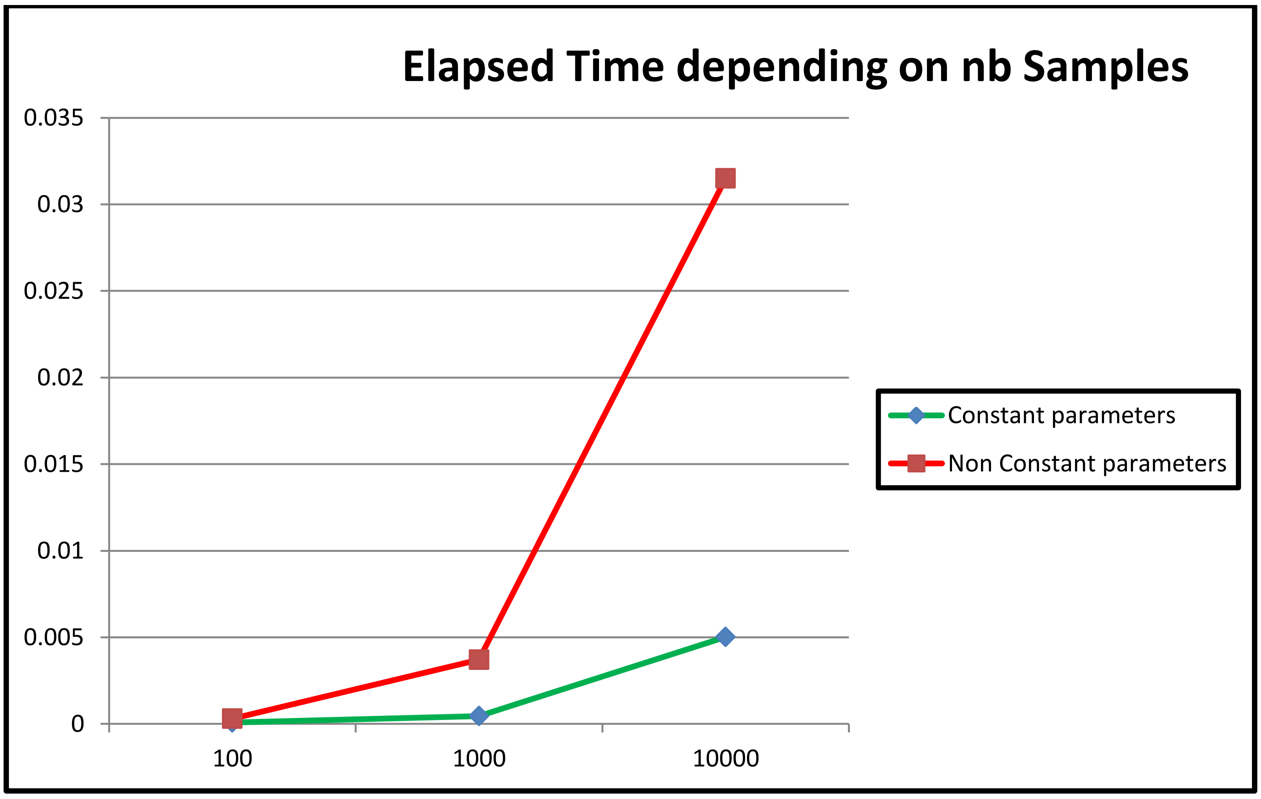
Tolerance	1	0.1	0.01	0.001	Sample	100	1000	10000
NPV	4.22887	4.22887	4.16506	4.17027		4.13251	4.22306	4.17516
Elapsed Time(s)	0.003681	0.005154	0.320378	41.7796		0.000308	0.003711	0.031502
Accuracy	0.0982288	0.0982288	0.009971	0.00099		0.322852	0.994568	0.03040

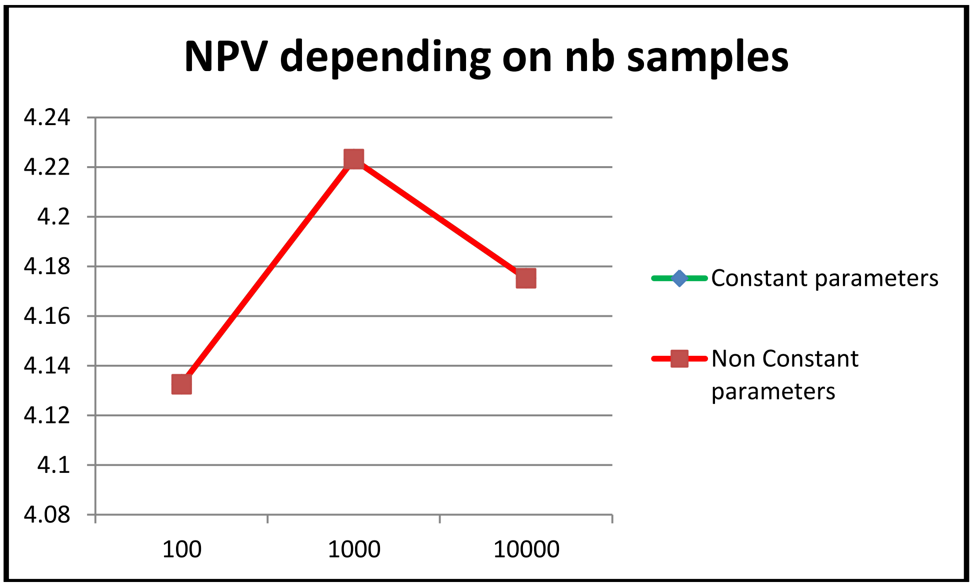
We can see the NPV are equals. However the elapsed time is way bigger with the non-Constant parameters. The difference is indeed very important for a tolerance value between 1 and 0.01 (can reach 10 times more). The ratio is certainly smaller for the tolerance superior to 0.01, but it still remains important (about 1.5 times bigger with non-constant parameters).

Graphically,






Elapsed Time depending on nb Samples
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
100
1000
10000
Constant parameters
Non Constant parameters


NPV depending on nb samples
4.24
4.22
4.2
4.18
4.16
4.14
4.12
4.1
4.08
100
1000
10000
Constant parameters
Non Constant parameters

Therefore, we can see that the mcengine with constant parameters is indeed faster than the one with non-constant parameters. It is also practically as accurate as the one with non-constant parameters. The error estimates chosen indicate the same error estimate and practically the same value for NPV. However, for a very small tolerance (and therefore a very big number of samples), the estimate error will not be the same for both engines and therefore, the NPV will also be different, but these differences can be neglected.