Constant parameters in Monte Carlo engines

Authors :

Yasmine LAARECH
Marie Camille MEMET
Amaury BERNARDIN
Hamid AMRANI

21/03/2025

Contents :

I. Introduction :

Monte Carlo simulations are often very time-consuming. In most applications, a trade-off must be made between two conflicting objectives: maximizing both accuracy and the richness of empirical information obtained, which increases with the number of simulations performed, and minimizing computation time. An efficient procedure ensures sufficient accuracy while keeping computation time limited.

In Monte Carlo engines, repeated calls to the process methods can significantly impact performance, particularly when using the GeneralizedBlackScholesProcess. This class relies on term structures for risk-free rates, dividend yields, and volatility, leading to costly computations during each simulation step. To improve efficiency at the cost of some accuracy, we propose implementing a Black-Scholes process with constant parameters, where the underlying value, risk-free rate, dividend yield, and volatility are fixed based on the option's exercise date.

We will modify the MCEuropeanEngine to support both the standard GeneralizedBlackScholesProcess and the new constant-parameter Black-Scholes process. A new boolean parameter will control this behavior :

- If false, the engine runs as usual.
- If true, the engine extracts constant values from the original process at the exercise date and runs simulations with the simplified model.

To achieve this, we will override the pathGenerator method in MCEuropeanEngine, ensuring that it can switch between the two processes dynamically. We will then compare the results in terms of option value, accuracy, and computational time to evaluate the trade-offs between efficiency and precision.

Additionally, we will extend this approach to a path-dependent option engine, such as MCDiscreteArithmeticAPEngine, which prices Asian options. We will analyze how the use of constant parameters affects pricing accuracy and performance in this more complex scenario.

II. Path generation :

Dynamic Simulation of the trajectories of Y(T) and the value of V(t,Y(t)) of a title (we suppose only one risk factor) in the interval (0,T)

We begin by simulating Y(t). This simulation is based on the stochastic equation that is assumed to govern the evolution of over time. Depending on the context, this evolution is initially expressed either in discrete time or continuous time. In the case of continuous time, it is typically modeled as a diffusion process:

$$dY = \mu(t, Y(t))dt + \sigma(t, Y(t))dW$$

where dW represents the increment of a standard Brownian motion, σ and μ are two known functions representing, respectively, the drift and diffusion coefficient of the process.

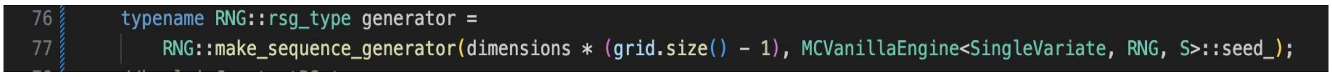
The interval is divided into N periods of equal length T/N . The number of periods is chosen to be sufficiently large so that the common duration is short (typically, a business day or a week). We simulate Y(t) and V(t,Y(t)) at the times t = $t_1,\ldots, t_N$. To do so, we discretize equation (1), leading to: $Y(t_j) - Y(t_{j-1}) = \mu(t_{j-1}, Y(t_{j-1}))\Delta t + \sigma(t_{j-1}, Y(t_{j-1}))\sqrt{\Delta t}U(j)$

where U(j) (for j=1,.....,N) are independent standard normal variables.

In some cases, the dynamics of are directly given by a discrete-time process of the same type as equation (2), which allows us to compute from the pair (Y(t_{j-1}),U(j)).

a) Random number generation

A particular trajectory $Y_{j=1\ldots N}$ is obtained by drawing independent values from a standard normal distribution $U_{j=1\ldots N}$ and iteratively applying equation (2). These values can be generated using MersemneTwister (totally random) and SobolRsg (random according to the dimensionality and the size of the vector chosen)).



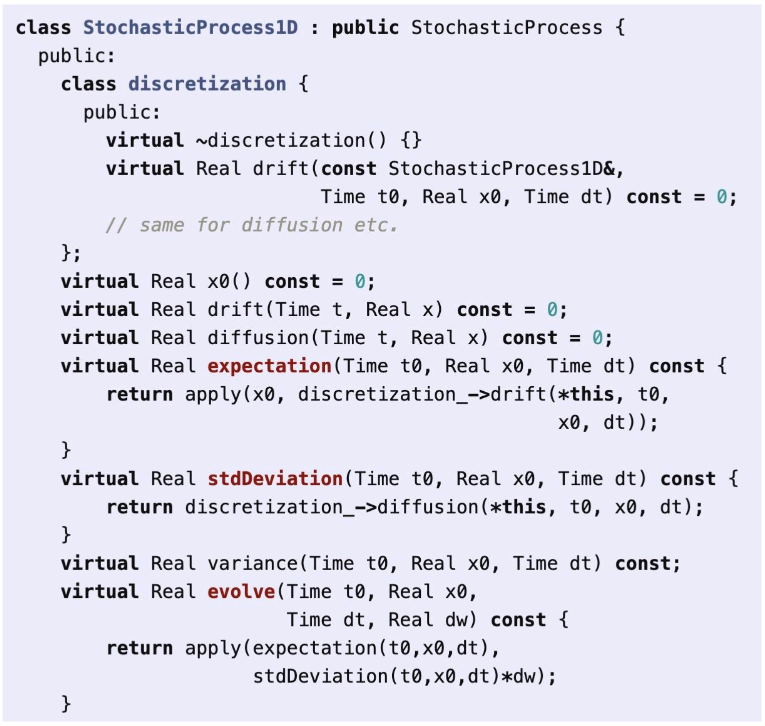
```
    typename RNG::rsg_type generator =
        RNG::make_sequence_generator(dimensions * (grid.size() - 1), MCVanillaEngine<SingleVariate, RNG, S>::seed_);
```

1.Random number generation

b) Constant Black Scholes Process

The target here is to create a new Class ConstantBlackScholesProcess, inherited from StochasticProcess1D class that models a Black-Scholes process with constant parameters (underlying value,risk-free rate, dividend yield, and volatility);

Firstly, it's important to know a minimum about the parent class StochasticProcess1D that we are going to use in order to implement this new class :



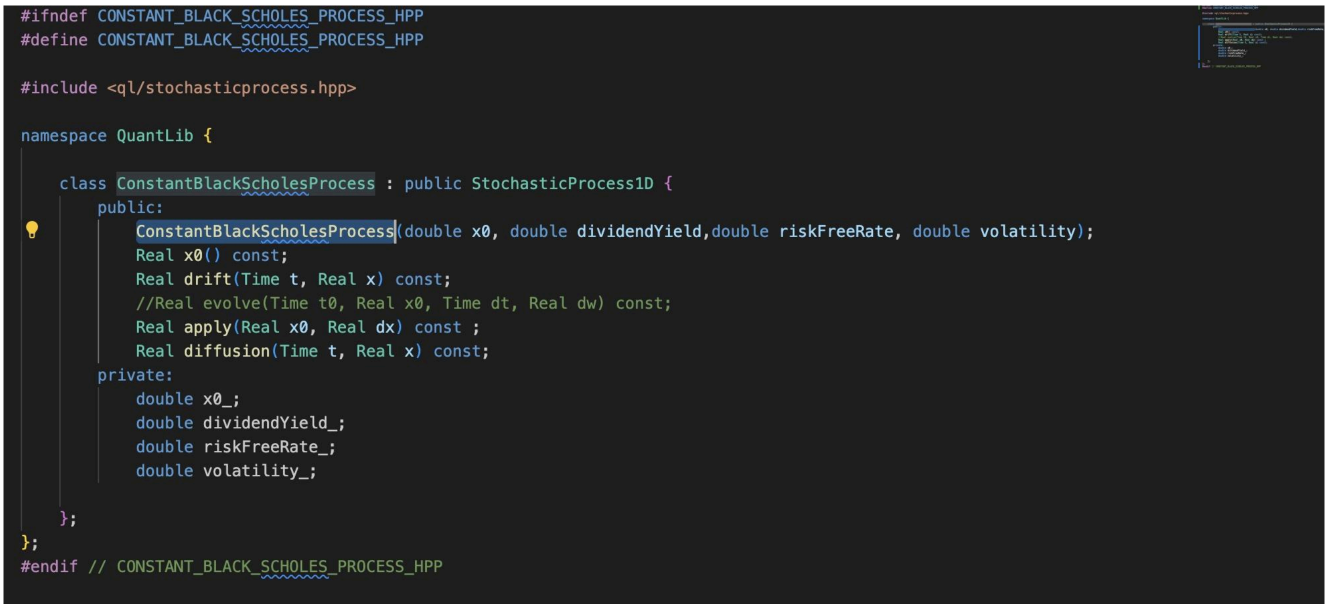
```
class StochasticProcess1D : public StochasticProcess {
  public:
    class discretization {
      public:
        virtual ~discretization() {}
        virtual Real drift(const StochasticProcess1D&,
                           Time t0, Real x0, Time dt) const = 0;
        // same for diffusion etc.
    };
    virtual Real x0() const = 0;
    virtual Real drift(Time t, Real x) const = 0;
    virtual Real diffusion(Time t, Real x) const = 0;
    virtual Real expectation(Time t0, Real x0, Time dt) const {
        return apply(x0, discretization_->drift(*this, t0,
                                                x0, dt));
    }
    virtual Real stdDeviation(Time t0, Real x0, Time dt) const {
        return discretization_->diffusion(*this, t0, x0, dt);
    }
    virtual Real variance(Time t0, Real x0, Time dt) const;
    virtual Real evolve(Time t0, Real x0,
                        Time dt, Real dw) const {
        return apply(expectation(t0,x0,dt),
                     stdDeviation(t0,x0,dt)*dw);
    }
```

2.1 Partial implementation of StochasticProcess1D class

This class implements the following stochastic equation (corresponding to **evolve** method in the inherited StochasticProcess1D class) $dln(Y) = \mu(t, Y(t))dt + \sigma(t, Y(t))dW$ expressed in log-space.

In the original implementation, the **drift** and **diffusion** methods frequently access term structures to retrieve values like volatility, risk-free rate, or dividend yield, which can lead to performance issues. This happens because these values are fetched through virtual method calls, and depending on the implementation, this may involve unnecessary calculations or repeated queries to the same data, which is inefficient.

By creating a class with constant parameters, we eliminate the need to query term structures. Instead of calling virtual methods to retrieve data at each time step, we directly use the constant values for volatility, risk-free rate, and dividend yield. Redefining the **drift** and **diffusion** methods in such a class allows them to return values based on these fixed parameters rather than fetching them dynamically. This simplification removes the overhead associated with virtual method calls, reducing the computational cost. As a result, the new class improves performance, especially in scenarios like Monte Carlo simulations, where numerous iterations are required.



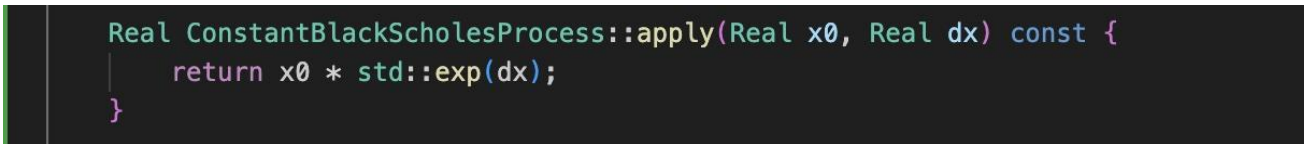
```cpp
#ifndef CONSTANT_BLACK_SCHOLES_PROCESS_HPP
#define CONSTANT_BLACK_SCHOLES_PROCESS_HPP

#include <ql/stochasticprocess.hpp>

namespace QuantLib {

    class ConstantBlackScholesProcess : public StochasticProcess1D {
        public:
            ConstantBlackScholesProcess(double x0, double dividendYield,double riskFreeRate, double volatility);
            Real x0() const;
            Real drift(Time t, Real x) const;
            //Real evolve(Time t0, Real x0, Time dt, Real dw) const;
            Real apply(Real x0, Real dx) const ;
            Real diffusion(Time t, Real x) const;
        private:
            double x0_;
            double dividendYield_;
            double riskFreeRate_;
            double volatility_;

    };
};
#endif // CONSTANT_BLACK_SCHOLES_PROCESS_HPP
```

2.2 The ConstantBlackScholesProcess Class

Apply method:

apply() permits to map back to the actual asset price.

Here ,the **apply()** method is redefined to directly apply the evolution of the underlying asset's value using a simple exponential formula, without the need to recalculate external values such as interest rates or volatility.

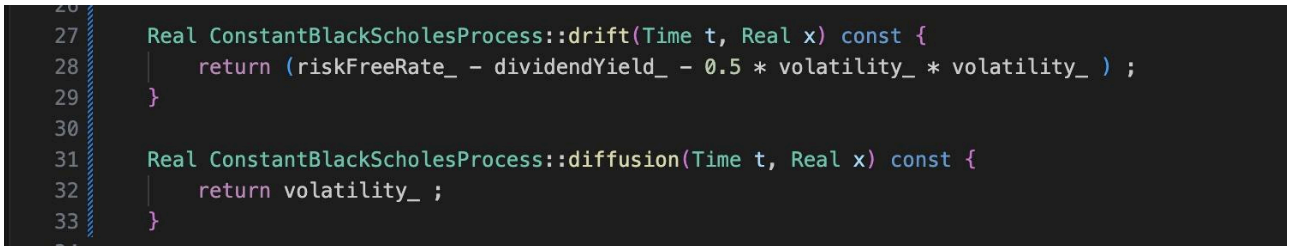


```
Real ConstantBlackScholesProcess::apply(Real x0, Real dx) const {
    return x0 * std::exp(dx);
}
```

Drift and diffusion coefficient:

As we said, the drift and diffusion coefficient of the process with constant parameters have to be redefined to optimize calculations because we don’t need to use those from the StochasticProcess1D Class. It limits calls to virtual methods of rate and usage structures (and useless recalculations because rates and volatilities are constant).

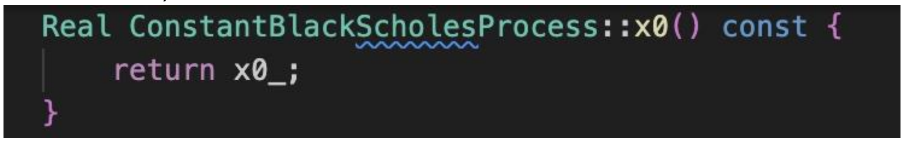
.



```
Real ConstantBlackScholesProcess::drift(Time t, Real x) const {
    return (riskFreeRate_ - dividendYield_ - 0.5 * volatility_ * volatility_ ) ;
}

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

x0() method :

The x0() method returns the initial value of the underlying asset (for example, the initial stock price). In the case of constant parameters, the initial value is given directly by the user when initializing the class (in other words, the x0 value does not change over time or according to market conditions).

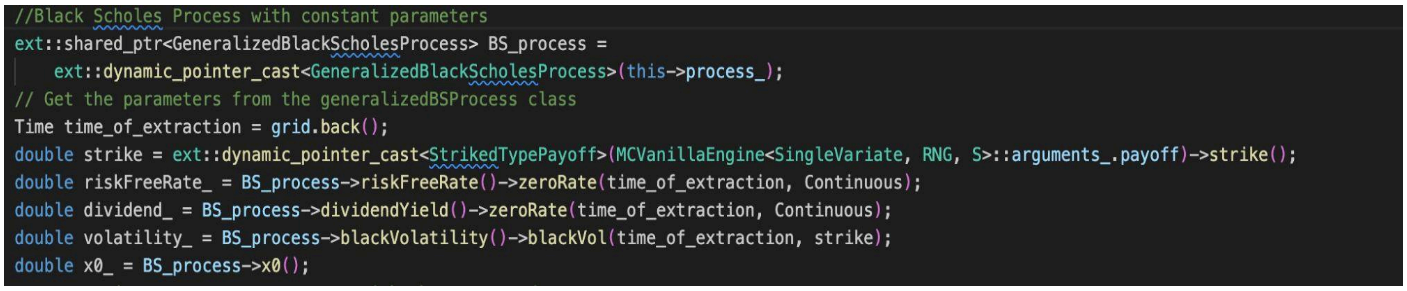


```
Real ConstantBlackScholesProcess::x0() const {
    return x0_;
}
```

III. Path Pricing : Putting it all together

The boolean parameter :

In this section, the objective is to introduce a boolean parameter into each Monte Carlo engine. When this parameter is set to false, the engine operates as usual, using StochasticProcess1D . However, when set to true, the engine extracts constant parameters from the original process and runs the Monte Carlo simulation using an instance of the constant process instead.



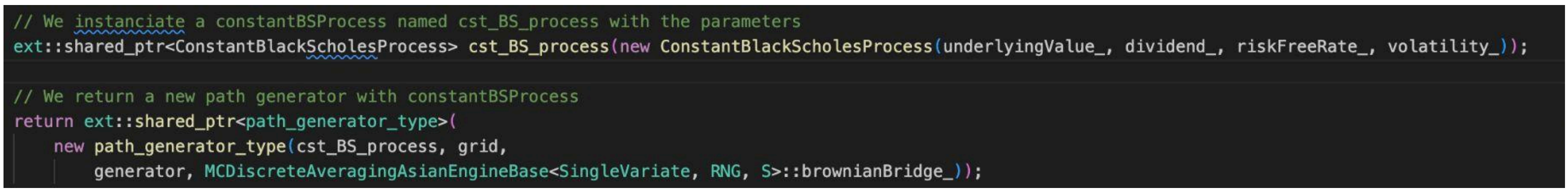
```cpp
//Black Scholes Process with constant parameters
ext::shared_ptr<GeneralizedBlackScholesProcess> BS_process =
    ext::dynamic_pointer_cast<GeneralizedBlackScholesProcess>(this->process_);
// Get the parameters from the generalizedBSProcess class
Time time_of_extraction = grid.back();
double strike = ext::dynamic_pointer_cast<StrikedTypePayoff>(MCVanillaEngine<SingleVariate, RNG, S>::arguments_.payoff)->strike();
double riskFreeRate_ = BS_process->riskFreeRate()->zeroRate(time_of_extraction, Continuous);
double dividend_ = BS_process->dividendYield()->zeroRate(time_of_extraction, Continuous);
double volatility_ = BS_process->blackVolatility()->blackVol(time_of_extraction, strike);
double x0_ = BS_process->x0();
```

The three constructors ,MCEuropeanEngine2, MCDiscreteArithmeticASEngine2, and MCBarrierEngine2, are modified to include this boolean parameter, named constantParameters. Once this parameter is defined, we need to carefully examine the engines and the inherited methods they use:

- EuropeanEngine: The PathGenerator method is inherited from the parent class MCVanillaEngine.
- DiscreteArithmeticASEngine: The PathGenerator method is inherited from the parent class MCDiscreteAveragingAsianEngineBase.
- BarrierEngine: The PathGenerator method is inherited from the parent class McSimulation.

We then override the path generator in each of these engine classes (MCEuropeanEngine2, MCDiscreteArithmeticASEngine2, and MCBarrierEngine2). The implementation takes into account the constantParameters boolean. A conditional check is made at this point: when the boolean is set to true, an instance of the constantBlackScholesProcess is created through the cst BS process object. This allows the path generator to use the process with constant parameters. These parameters, such as the underlying value, risk-free rate, dividend yield, and volatility, are extracted from the constantBlackScholesProcess (as we see in the section before), and the allocation is computed once.

This approach is applied across all three engines, in accordance with the parent class they inherit from (MCVanillaEngine, MCDiscreteAveragingAsianEngineBase, and McSimulation).



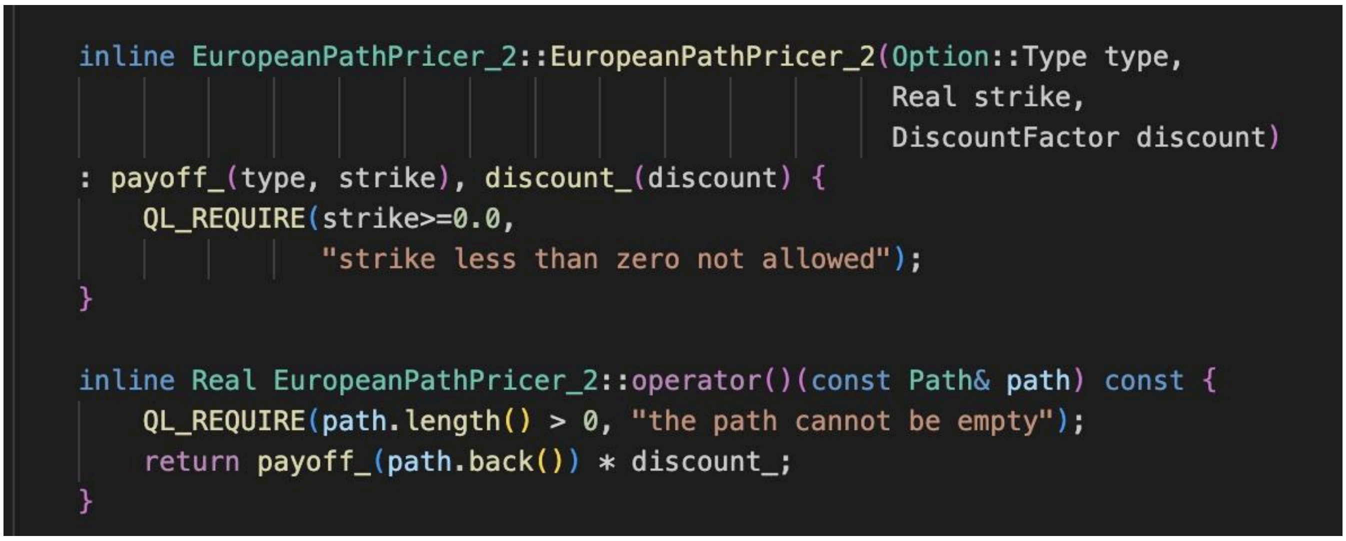
```cpp
// We instanciate a constantBSProcess named cst_BS_process with the parameters
ext::shared_ptr<ConstantBlackScholesProcess> cst_BS_process(new ConstantBlackScholesProcess(underlyingValue_, dividend_, riskFreeRate_, volatility_));

// We return a new path generator with constantBSProcess
return ext::shared_ptr<path_generator_type>(
    new path_generator_type(cst_BS_process, grid,
        generator, MCDiscreteAveragingAsianEngineBase<SingleVariate, RNG, S>::brownianBridge_));
```

Path pricing implementation

1. European Option :

The operator() method in the EuropeanPathPricer_2 class calculates the option price by first evaluating the payoff at the final time point of the path, which depends on the option type (call or put) and the strike price.

Once the payoff is calculated, it is then discounted to the present by multiplying the payoff by the discount factor. This gives the present value of the option based on the path's final price and the discounting factor passed during the construction of the EuropeanPathPricer_2 object.



```cpp
inline EuropeanPathPricer_2::EuropeanPathPricer_2(Option::Type type,
                                                  Real strike,
                                                  DiscountFactor discount)
: payoff_(type, strike), discount_(discount) {
    QL_REQUIRE(strike>=0.0,
               "strike less than zero not allowed");
}

inline Real EuropeanPathPricer_2::operator()(const Path& path) const {
    QL_REQUIRE(path.length() > 0, "the path cannot be empty");
    return payoff_(path.back()) * discount_;
}
```

2. Asian Option :

The key difference between an Asian option and a European option is that an Asian option’s payoff depends on the average of the underlying asset prices over a period, rather than just the final price at maturity.

In the MCDiscreteArithmeticASEngine_2 class, the pathPricer() method is responsible for defining how each simulated path is priced. It first retrieves the payoff type, ensuring it is a plain vanilla payoff, and confirms that the option follows a European exercise style, meaning it can only be exercised at the final time step. It also ensures that the underlying asset follows a Generalized Black-Scholes Process.

The actual pricing along the simulated paths is done using the ArithmeticASOPathPricer. This class computes the arithmetic average of the underlying asset’s prices over the monitoring dates, rather than just taking the final price. The discounted value of the payoff is then calculated using the risk-free rate.

By using Monte Carlo simulation, the engine generates multiple paths of the underlying asset price evolution. For each path, the average price over the monitoring dates is computed, the payoff is determined based on this average, and then the expected value is estimated by averaging over all simulated payoffs, which is finally discounted to the present.

3. Barrier Option ;

Then we implement a Monte Carlo simulation engine for pricing barrier options in QuantLib. A barrier option is an exotic option whose value depends on whether the underlying asset's price crosses a predefined threshold before expiration.

The calculate() method ensures that input parameters are valid, checking that the initial underlying asset price is strictly positive and that the barrier has not already been breached. It then runs the Monte Carlo simulation and stores the estimated option price in results_.value. If the random number generator allows error estimation, the computed error is stored in results_.errorEstimate:

```cpp
void calculate() const override {
    Real spot = process_->x0();
    QL_REQUIRE(spot > 0.0, "negative or null underlying given");
    QL_REQUIRE(!triggered(spot), "barrier touched");
    McSimulation<SingleVariate, RNG, S>::calculate(requiredTolerance_,
        requiredSamples_,
        maxSamples_);
    results_.value = this->mcModel_->sampleAccumulator().mean();
    if (RNG::allowsErrorEstimate)
        results_.errorEstimate =
        this->mcModel_->sampleAccumulator().errorEstimate();
}
```

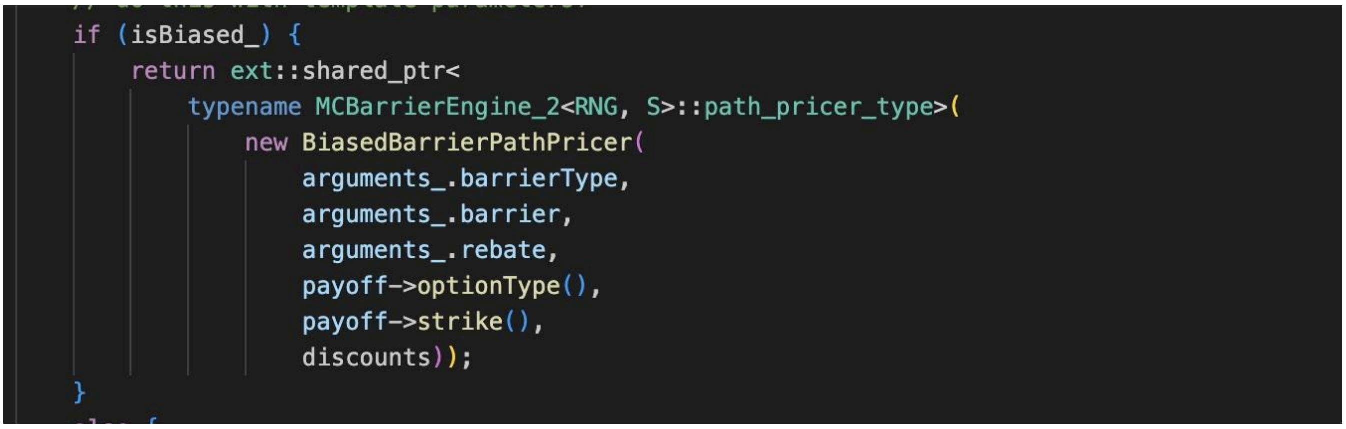
The pathPricer() method in MCBarrierEngine_2 determines the value of each simulated barrier option path. It begins by extracting the option's payoff using a dynamic cast, as seen in the following code:

```cpp
ext::shared_ptr<typename MCBarrierEngine_2<RNG, S>::path_pricer_type>
MCBarrierEngine_2<RNG, S>::pathPricer() const {
ext::shared_ptr<PlainVanillaPayoff> payoff =
    ext::dynamic_pointer_cast<PlainVanillaPayoff>(arguments_.payoff);
QL_REQUIRE(payoff, "non-plain payoff given");
```

This ensures that the payoff is of the expected type, throwing an error if it is not. Next, the method creates a time grid and calculates discount factors for each time point, which are essential for discounting the future cash flows to their present value:

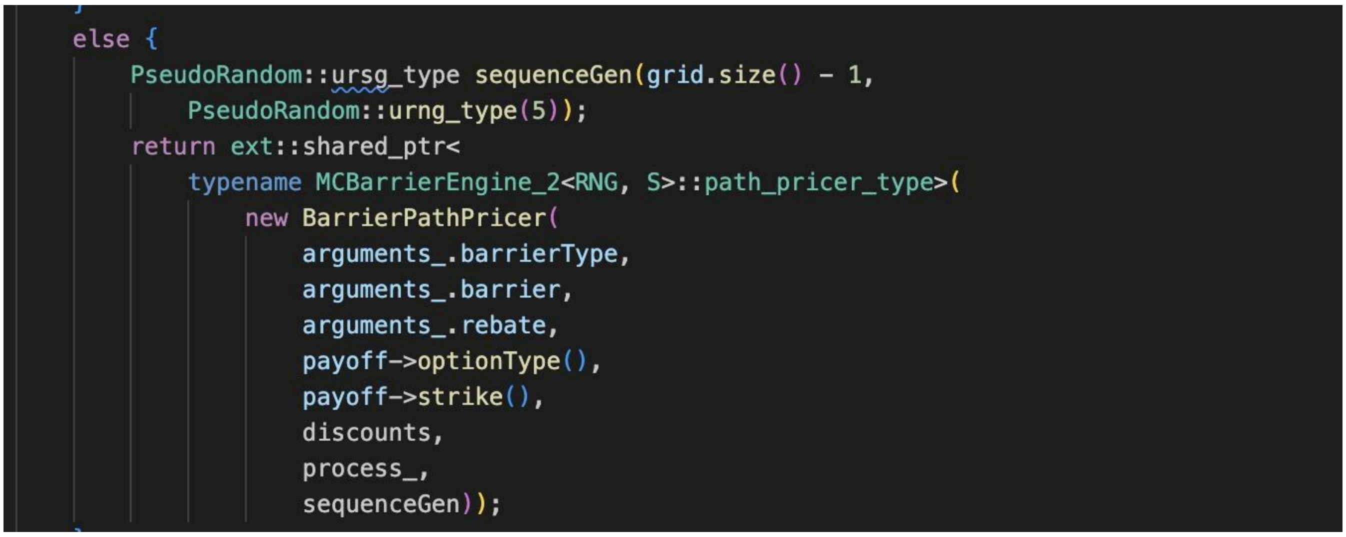
```cpp
TimeGrid grid = timeGrid();
std::vector<DiscountFactor> discounts(grid.size());
for (Size i = 0; i < grid.size(); i++)
    discounts[i] = process_->riskFreeRate()->discount(grid[i]);
```

The method then chooses between two different path pricers based on the isBiased_ flag. If the flag is true, a BiasedBarrierPathPricer is instantiated:



```
if (isBiased_) {
    return ext::shared_ptr<
        typename MCBarrierEngine_2<RNG, S>::path_pricer_type>(
            new BiasedBarrierPathPricer(
                arguments_.barrierType,
                arguments_.barrier,
                arguments_.rebate,
                payoff->optionType(),
                payoff->strike(),
                discounts));
}
```

This pricer applies a bias correction for barrier option pricing. Otherwise, the standard BarrierPathPricer is used, which involves generating an auxiliary random sequence for its calculations:

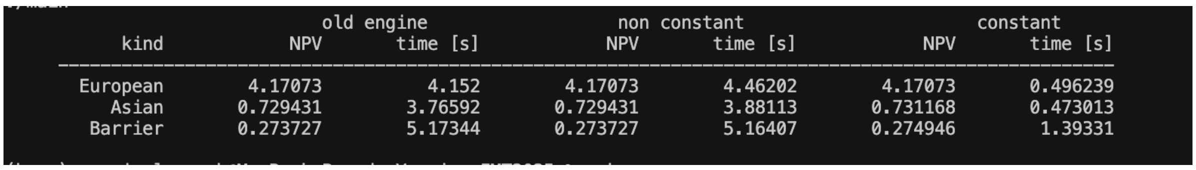


```
else {
    PseudoRandom::ursg_type sequenceGen(grid.size() - 1,
        PseudoRandom::urng_type(5));
    return ext::shared_ptr<
        typename MCBarrierEngine_2<RNG, S>::path_pricer_type>(
            new BarrierPathPricer(
                arguments_.barrierType,
                arguments_.barrier,
                arguments_.rebate,
                payoff->optionType(),
                payoff->strike(),
                discounts,
                process_,
                sequenceGen));
```

This pricer evaluates whether the barrier is breached along the simulated path and calculates the payoff accordingly, taking into account features such as knock-in or knock-out conditions and any applicable rebate.

Finally, in the results section, we assess the impact of the constantBlackScholesProcess class on computational efficiency. The expectation is that the use of constant parameters will reduce computational costs and improve performance.

IV.Simulations and results :



```
                              old engine                         non constant                                constant
        kind               NPV        time [s]             NPV        time [s]               NPV        time [s]
      -------------------------------------------------------------------------------------------------------------
      European        4.17073           4.152         4.17073         4.46202           4.17073        0.496239
         Asian       0.729431         3.76592        0.729431         3.88113          0.731168        0.473013
       Barrier       0.273727         5.17344        0.273727         5.16407          0.274946         1.39331
```

The results presented above show the Net Present Value (NPV) and execution time (time [s]) for three different types of simulation engines: Old Engine, Non-constant, and Constant. The results are analyzed for three types of options: European, Asian, and Barrier

It is observed that by using constant parameters, the execution time decreases significantly for all three types of options. Additionally, there is a noticeable difference in the NPV values, with a greater variation seen for the Asian and Barrier options compared to the European option.

The reason why the NPV changes for the Asian and Barrier options but not for the European option in the constant case likely lies in the specific characteristics of the options and how they are affected by the constant parameters.

For the European option, the NPV remains the same because it is a simpler option in terms of its pricing mechanics. The European option only requires the final payoff at maturity to be considered, and with constant parameters, there is less room for variation in its pricing. Essentially, the NPV for the European option might be relatively stable across different simulation conditions, especially when parameters are kept constant.

On the other hand, the Asian and Barrier options are more complex. The Asian option depends on the average price over the life of the option, and the Barrier option's payoff depends on whether the underlying asset reaches a certain threshold. Both of these options are more sensitive to changes in the input parameters, especially volatility and time to maturity, which could explain why the NPV for these options varies when using constant parameters.