QuantLib Project : Monte Carlo

MOHMOH Abdellah
OLIVIER Julien
TRAORE Idrissa
NALIATO Anthony

Contents

1. Introduction

1.1 Overview of Instructions

In Monte Carlo engines, repeatedly calling process methods can lead to performance issues, especially when the process is an instance of the `GeneralizedBlackScholesProcess` class. This is because its methods make expensive calls to the underlying term structures.
To improve performance (at the cost of some accuracy), the task is to:

1. Create a new class that models a **Black-Scholes process with constant parameters** (underlying value, risk-free rate, dividend yield, and volatility).
2. Modify the Monte Carlo engines in the repository to accept:
    - A generic Black-Scholes process.
    - An additional boolean parameter.
        - If `false`, the engine runs as usual.
        - If `true`, the engine extracts constant parameters from the original process (e.g., the risk-free rate is the zero-rate of the full risk-free curve at the option's exercise date) and runs the simulation using the constant process.

1.2 Description of Project Architecture

The QuantLib project consists of three main files:

- `mc_discr_arith_av_strike.hpp`
- `mcbarrierengine.hpp`
- `mceuropeanengine.hpp`

Each file uses the Monte Carlo process for different types of options: European, Asian, and Barrier.
The Monte Carlo process involves the following steps:

1. **Random number generation**:
    - Two types: `MersenneTwister` (fully random) and `SobolRsg` (random based on dimensionality and vector size).
2. **Stochastic processes**:
    - Modeled as $dX = \mu(t,X)\,dt + \sigma(t,X)\,dW$
        i. $\mu(t,X)$: *Drift component.*
        ii. $\sigma(t,X)$: *Diffusion component.*
        iii. dW : *Random component*
3. **Path generation**:
    - Combines random numbers and stochastic processes to define the time grid.
4. **Path pricing**:
    - Calculates the option price based on the generated paths.
5. **Simulations**:
    - Aggregates results from path generation and pricing.

To improve pricing performance, the method reuses components of the Black-Scholes process. The next section introduces two additional files:

- `constantblackscholesprocess.cpp`
- `constantblackscholesprocess.hpp`

These files implement a **Black-Scholes process with constant parameters**.

2. Implementation

2.1 Explanation of the Black-Scholes Process

The `ConstantBlackScholesProcess` class models a **Black-Scholes process with constant parameters**, such as:

- Underlying value (`underlyingValue`)
- Risk-free rate (`riskFreeRate`)
- Dividend yield (`dividend`)
- Volatility (`volatility`)

This class inherits from the `StochasticProcess1D` class and implements four key methods:

1. **`x0()`**: Returns the initial value of the underlying asset.
2. **`drift(Time t, Real x)`**: Computes the drift component of the process.
3. **`diffusion(Time t, Real x)`**: Computes the diffusion (volatility) component.
4. **`apply(Real x0, Real dx)`**: Applies a change `dx` to the initial value `x0`.

These methods are used by the **`evolve`** function in the `StochasticProcess1D` class, which plays a central role in generating paths for the Monte Carlo simulation. The `evolve` function relies on three key methods:

- **`apply`**: Applies the change to the underlying value.
- **`expectation`**: Computes the expected value of the process.
- **`stdDeviation`**: Computes the standard deviation of the process.

Here's a snippet of the `ConstantBlackScholesProcess` class in C++:

```
class ConstantBlackScholesProcess : public StochasticProcess1D {

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 underlyingValue;
    double riskFreeRate;
    double volatility;
    double dividend;

};
```

- **`x0()`**: Returns the initial underlying value.
- **`drift()`** : Computes the drift as $r - q - 0.5\,\sigma$ ** 2 where r is the risk-free rate, q is the dividend yield, and σ is the volatility.
- **`diffusion()`**: Returns the constant volatility.
- **`apply()`**: Applies the change using $x0 \cdot e$ ** dx

2.2 Role of the Boolean Parameter

The goal here is to introduce a **boolean parameter** (`constantParameters`) in each Monte Carlo engine. This parameter determines whether the engine uses:

- The **original process** (if `false`).
- A **constant Black-Scholes process** (if `true`).

The boolean parameter is added to the constructors of three engine classes:

1. `MCEuropeanEngine_2`
2. `MCDiscreteArithmeticASEngine_2`
3. `MCBarrierEngine_2`

Each engine overrides the **`pathGenerator`** method to handle the boolean parameter:

- If `constantParameters` is `true`, the engine extracts constant parameters (underlying value, risk-free rate, dividend yield, and volatility) from the original process and creates an instance of `ConstantBlackScholesProcess`.
- If `constantParameters` is `false`, the engine uses the original process.

Here's how the `pathGenerator` method is implemented in the `MCEuropeanEngine_2` class:

```
//bool constantParameters=true;
if (this->constantParameters)
{
    ext::shared_ptr<GeneralizedBlackScholesProcess> BS_process =
        ext::dynamic_pointer_cast<GeneralizedBlackScholesProcess>(this->process_);

    // Get the parameters from the ConstantBlackScholesProcess class
    Time time_of_extraction = grid.back();
    double strike = ext::dynamic_pointer_cast<StrikedTypePayoff>(MCVanillaEngine<SingleVariate, RNG, S>::arguments_.payoff
    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 underlyingValue_ = BS_process->x0();

    // We instanciate a constantBSProcess named cst_BS_process with the parameters
    ext::shared_ptr<ConstantBlackScholesProcess> cst_BS_process(new ConstantBlackScholesProcess(underlyingValue_, riskFree

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

Key Points

- **Performance Improvement**: Using the constant process reduces computational costs by avoiding repeated calls to expensive methods in the original process.
- **Flexibility**: The boolean parameter allows users to switch between the original and constant processes easily.
- **Inheritance**: Each engine class inherits its `pathGenerator` method from a parent class (`MCVanillaEngine`, `MCDiscreteAveragingAsianEngineBase`, or `McSimulation`), ensuring consistency across different option types.

In the last section, the results of simulations are presented. They determine whether the constantBlackScholesProcess class makes computational costs lower and the performance is increased.

3. Results

3.1 Execution of the Main Program

The results showcase the performance and Net Present Values (NPV) for three types of options (European, Asian, and Barrier) using three different configurations:

- **Old Engine**: The original Monte Carlo engine without optimization.
- **Non Constant**: The new Monte Carlo engine without using constant parameters.
- **Constant**: The new Monte Carlo engine with constant parameters enabled.

1. European Options

NPV: The NPV remains exactly the same across all configurations. This indicates that using constant parameters does not affect the accuracy of the calculation for European options.
Time: The execution time is significantly reduced with the constant parameters (1.41447 s) compared to the old engine (9.88391 s) and the non-constant engine (15.3154 s). This demonstrates that the optimization works well for European options.

Configuration	NPV	Time (s)
Old Engine	4.17073	9.88391
Non Constant	4.17073	15.3154
Constant	4.17073	1.41447

2. Asian Options

NPV: The NPV is slightly different for the constant configuration (0.731168) compared to the others (0.729431). This minor difference may be due to the approximation introduced by using constant parameters.
Time: The execution time is drastically reduced with the constant parameters (0.884244 s) compared to the old engine (11.0943 s) and the non-constant engine (8.32357 s). This shows that the optimization is highly effective for Asian options.

Configuration	NPV	Time (s)
Old Engine	0.729431	11.0943
Non Constant	0.729431	8.32357
Constant	0.731168	0.884244

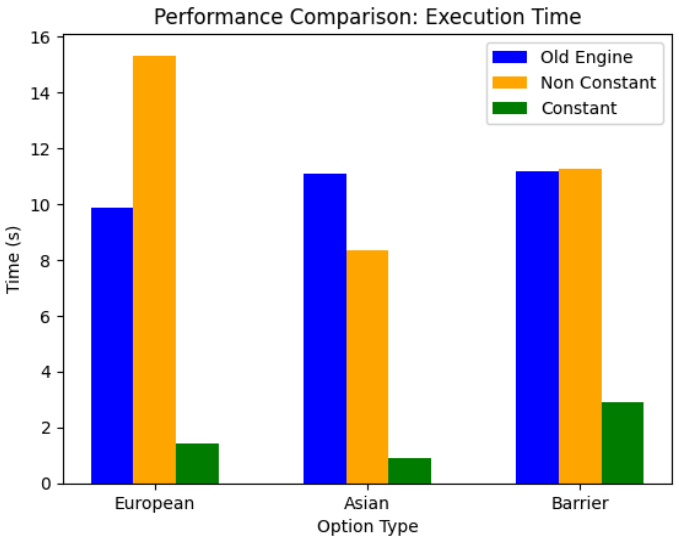
3. Barrier Options

NPV: The NPV is slightly different for the constant configuration (0.274946) compared to the others (0.273727). This small difference may be due to the approximation introduced by using constant parameters.
Time: The execution time is reduced with the constant parameters (2.91737 s) compared to the old engine (11.1662 s) and the non-constant engine (11.267 s). This demonstrates that the optimization works well for Barrier options.

Configuration	NPV	Time (s)
Old Engine	0.273727	11.1662
Non Constant	0.273727	11.267
Constant	0.274946	2.91737

3.2 Global analysis

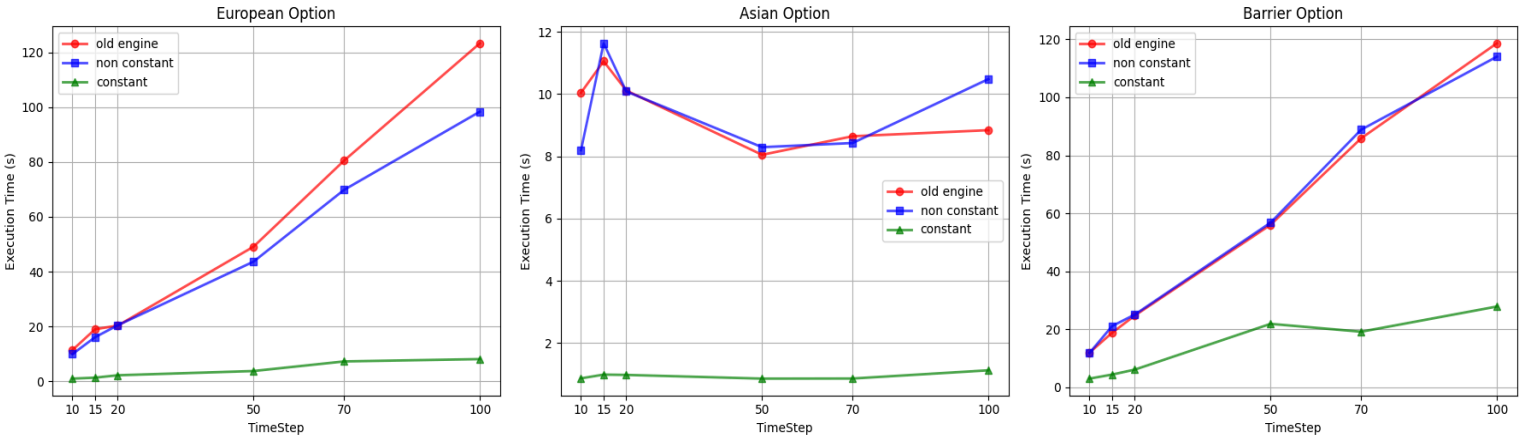
1. Accuracy (NPV):
For European options, the NPV remains exactly the same, showing that using constant parameters does not affect precision.
For Asian and Barrier options, the NPV is slightly different with constant parameters. This is likely due to the approximation introduced by using constant parameters, but the difference is minimal.



2. Performance (Execution Time):
Using constant parameters significantly reduces execution time for all three types of options.
The performance gain is particularly notable for European options (85% reduction) and Asian options (92% reduction).

In this section, we present all the simulations we have done : one parameter has been varied and the others were fixed at the original values (timeSteps, samples, strike, maturity, volatility 3 months).

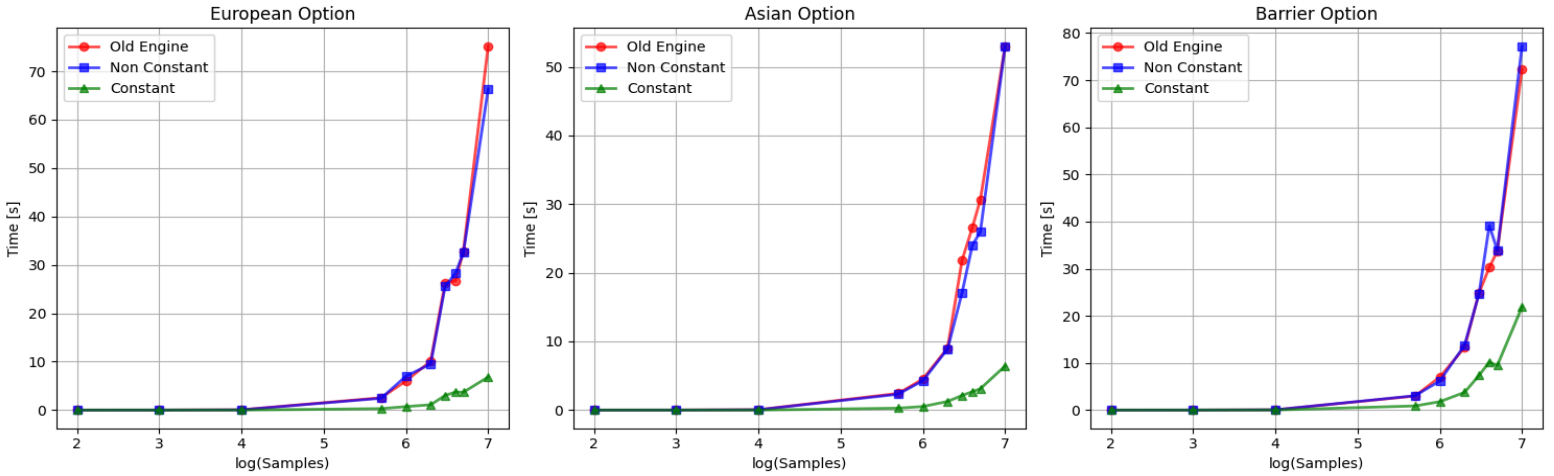
3.3 TimeSteps Variation



In our initial simulation, we progressively varied the timesteps from 10 to 100 while keeping all other parameters constant, then executed the main computation. The results demonstrate that:

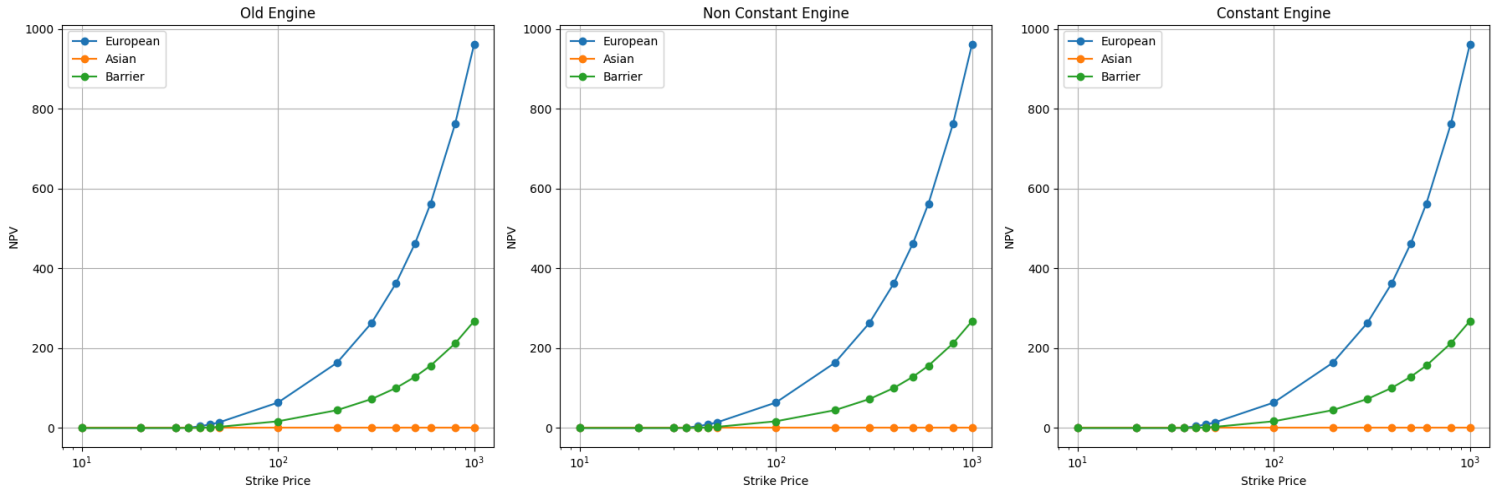
1. The **constant engine** consistently maintains the lowest execution times across all three option types (European, Asian, and Barrier).
2. Its performance remains remarkably stable despite **TimeStep** variations, showing only a marginal increase for Barrier options.
3. In contrast, both the old and non-constant engines exhibit:
    - Significantly longer execution times
    - A sharp, near-linear growth in computational duration as timesteps increase
    - Similar performance patterns between them

3.4 Samples Variation



By varying sample sizes from 100 to 10^n , n = 7, we observe exponential growth in execution time for both the old and non-constant engines, as evidenced by the linear trend in log-scale plotting. The constant engine alone maintains near-constant performance (O(1)), demonstrating superior algorithmic efficiency. Barrier options exhibit the steepest exponential curves, highlighting their heightened sensitivity to sample size increases. This confirms the constant engine's unique ability to bypass the exponential scaling constraints affecting other implementations.

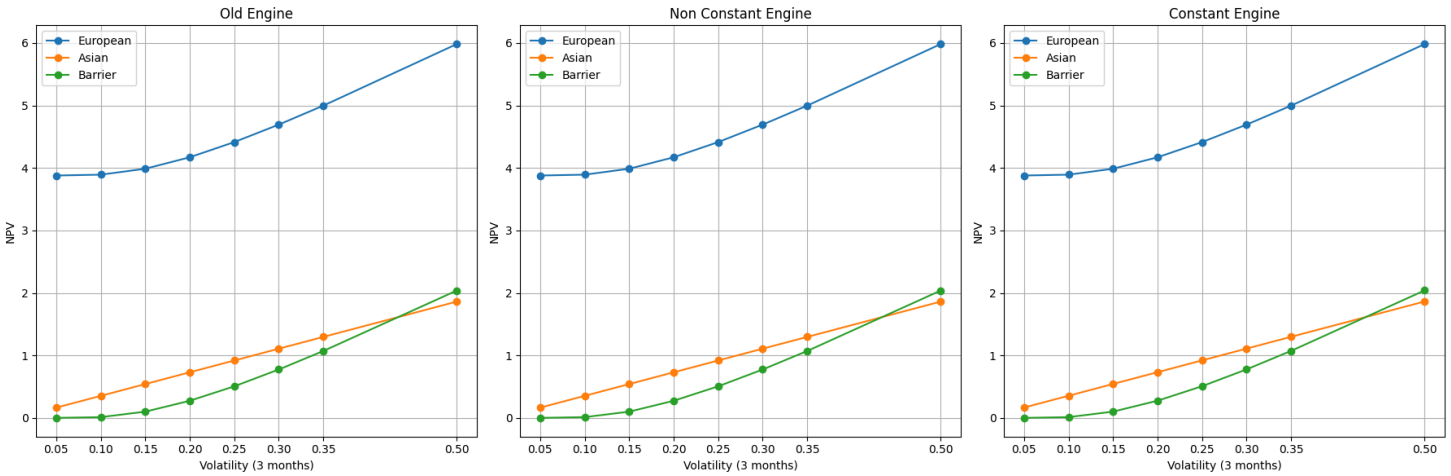
3.5 Strike Prices Variation



In this part, we check how changing the strike price changes the NPV for all three engines. What's interesting is that even though the engines work differently, they all give exactly the same NPV when you use the same strike price! The graph shows this clearly - all three lines (old, non-constant, and constant engines) follow the same path as the strike price goes up or down. This means the strike price affects NPV the same way no matter which engine you use.

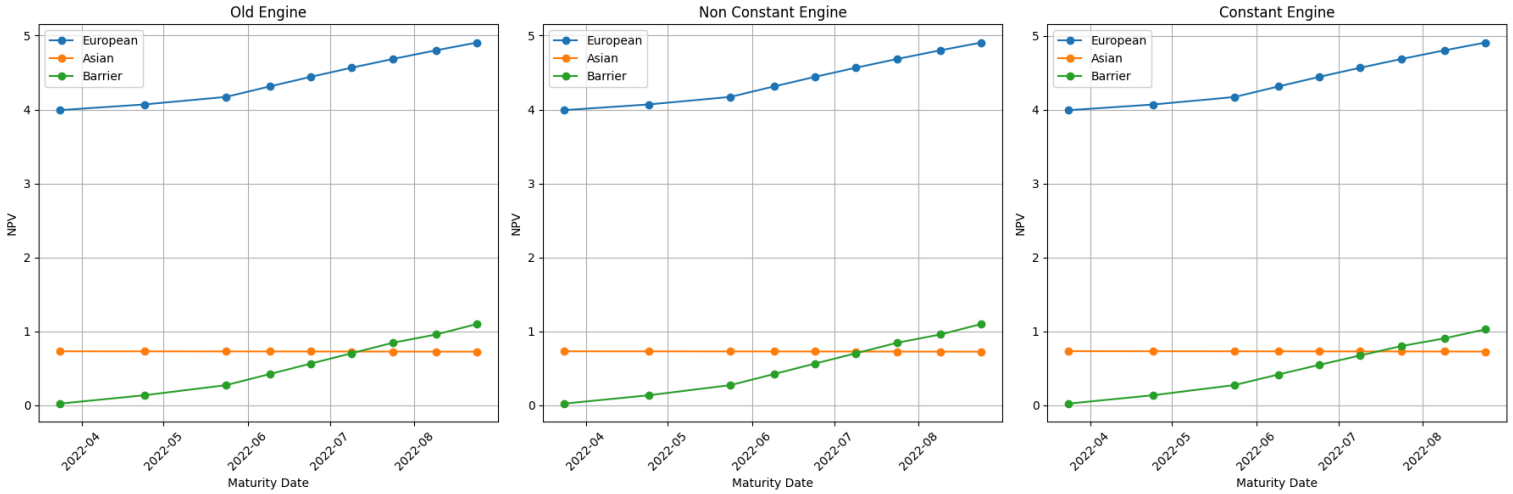
Both the European option and the Barrier one increase when the strike price raises. However, the asian option seems to be indifferent to the strike price. This result might be refutable even if it could be pertinent.

3.6 Volatility Variation



The Net Present Value (NPV) calculates the current worth of an investment's future cash flows. Our simulations show NPV is highly sensitive to volatility. When volatility increases, potential cash flow outcomes vary more widely, making NPV estimates less certain. Conversely, lower volatility narrows the range of possible outcomes, resulting in more reliable NPV estimates.
Additionally, we tested a volatility of 0.5 to verify consistency and check for potential computational errors.

3.7 Maturity Variation



Changes in cash flow maturity can impact the NPV estimate by influencing the discount rate used to calculate present value. Shorter maturities often justify a lower discount rate due to lower risk, while longer maturities may require a higher rate to account for increased uncertainty. In Monte Carlo simulations, sensitivity analysis helps assess how NPV responds to changes in maturity by adjusting this parameter while keeping others constant. This analysis provides investors with a clearer understanding of the investment's risk and helps in making informed decisions about its profitability and feasibility.