Quantlib project

Project 1 : Use constant parameters in Monte Carlo engines

Authors : Théophile UNG - Thibault SPRIET - Paul NANTAS - Paul POUPARD

1. 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 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.

2. Implementation

The Monte-Carlo process can be decomposed in few steps :

1. Random number of generations (2 types : MersemneTwister (totally random) and SobolRsg (random according to the dimensionality and the size of the vector chosen))
2. Stochastic processes (dX = mu(t,X)*dt + sigma(t,X)dW)
    - mu(t,X) = drift component
    - sigma(t,X) = diffusion component
    - dW = random component
3. Path generation (result of 1 and 2) (define the grid of time)
4. Path pricing
5. Simulations (result of 3 and 4)

Knowing the steps of the Monte-Carlo process, the idea is to reuse some components of the Black-Scholes process in order to get the Monte Carlo simulations of pricing better performance.

So, we created the ConstantBlanckScholesProcess class to reuse some components of the original process. This class takes constant parameters such as the underlying value, the risk-free rate, the volatility and the dividend. Moreover, this class defines 4 other methods necessary for the development of the stochastic process in our class. The draft of the class is shown below :

```cpp
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 underlyingValue;
        double riskFreeRate;
        double volatility;
        double dividend;

    };

}
```

ConstantBlackScholesProcess.hpp

Description of the ConstantBlackScholesProcess class :

First, the class takes 4 constant parameters considered as constant in the implementation of the class : the *underlyingVlaue, riskFreeRate, volatility* and *dividend*. Moreover the class redefines 4 methods inherited from StochasticProcess1D *x0(), drift(Time t, Real x), diffusion (Time t, Real x), apply(Real x0, Real dx)*. These methods are redefined because they are variable of the path generator through the call of “evolve” function in the StochasticProcess1D class. Indeed the “evolve” method plays a central role in the generation of paths.

```cpp
Real StochasticProcess1D::evolve(Time t0, Real x0,
                                 Time dt, Real dw) const {
    return apply(expectation(t0,x0,dt), stdDeviation(t0,x0,dt)*dw);
}
```

So the implementation of the “evolve” function calls our “apply” function but also the “expectation” function that calls our “drift” function, and the “stdDeviation” function that calls our “diffusion” function as you can see below.

```cpp
Real StochasticProcess1D::expectation(Time t0, Real x0, Time dt) const {
    return apply(x0, discretization_->drift(*this, t0, x0, dt));
}

Real StochasticProcess1D::stdDeviation(Time t0, Real x0, Time dt) const {
    return discretization_->diffusion(*this, t0, x0, dt);
}
```

So, these reasons lead to redefine these particular functions given above in the ConstantBlackScholesProcess class.

The next step was to add a boolean (named “isConstantBS” in our implementation) in the MCEuropeanEngine class :

- if the boolean is false, the engine runs as usual
- if the boolean is true, the engine runs with the implementation of our ConstantBlackScholesProcess class which takes constant parameters of the original process

The draft of the constructor with the boolean added is shown below :

```
// 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,
    bool isConstantBS);
```

Constructor of MCEuropeanEngine class with the added boolean parameter

Finally, we overrode the PathGenerator method inherited from the parent class MCVanillaEngine so that our implementation takes into account the boolean parameter added as we can see below :

```
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_);
    //bool isConstantBS=true;
    if (isConstantBS_)
    {
        std::cout << "Black Scholes Process with constant parameters" << std::endl;
        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 underlyingValue_ = BS_process->x0();
        // Instanciate a constantBSProcess with the extracted parameters
        ext::shared_ptr<ConstantBlackScholesProcess> Cst_BS_process(new ConstantBlackScholesProcess(underlyingValue_, riskFreeRate_, volatility_, dividend_));
        // 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_));
    }
    else
    { // return the classical path generator
        std::cout << "Black Scholes Process runs as usual (parameters extracted from GeneralizedBSProcess)" << std::endl;

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

Override of the PathGenerator method

Description of the overriden PathGenerator :

The implementation of the override of the pathGenerator() method is done in MCEuropeanEngine_2 class where the boolean parameter isConstantBS is added determine whether the engine should runs as usual or extracts constant parameters from the original process and runs the simulation with an instance of the constantBlackScholesProcess. So :

- if isConstantBS is false, the path generator runs as usual

```
else
{ // return the classical path generator
    std::cout << "Black Scholes Process runs as usual (parameters extracted from GeneralizedBSProcess)" << std::endl;

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

- if isConstantBS is true, we extract the constant parameters, defined in the ConstantBlackScholesProcess class which are *riskFreeRate, dividend, volatility, underlyingValue*, from the GeneralizedBlackScholesProcess (through the variable BS_process in the code). This allocation is calculated only one time as we can see below and will be used in the generation of the path.

```
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 underlyingValue_ = BS_process->x0();
```

  Then, we instanciate the constantBlackScholesProcess through the object Cst_BS_process that takes constant parameters calculated above, so that the path generator can use the process with constant parameters.

```
ext::shared_ptr<ConstantBlackScholesProcess> Cst_BS_process(new ConstantBlackScholesProcess(underlyingValue_, riskFreeRate_, volatility_, dividend_));
// 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_));
```

  The path is thereby generated with the object Cst_BS_process which tackes contant parameters..

Thus, we run our implementation modifying the main and compiling our new version of ConstantBlackScholesProcess class in order to get the results obtained in the next chapter.

3. Results

For our tests we compute the NPV value of the option, and we also check the elapsed time and estimated error. Those values are computed for the general and constant Black-Scholes process. In order to analyze our results, by passing an argument to our main, we store results in a csv file.

```
(base) sprietthibault:project1/ (master*) $ ./main results/simu.csv
```

calling main with path to store results

```
bool write_results = argc > 1;
if(write_results){
    std::ofstream file;
    file.open(argv[1],std::ios::app);
    file << "step,sample,c_err,c_npv,c_time,err,npv,time,\n";
    file.close();
}
```

create results file with columns

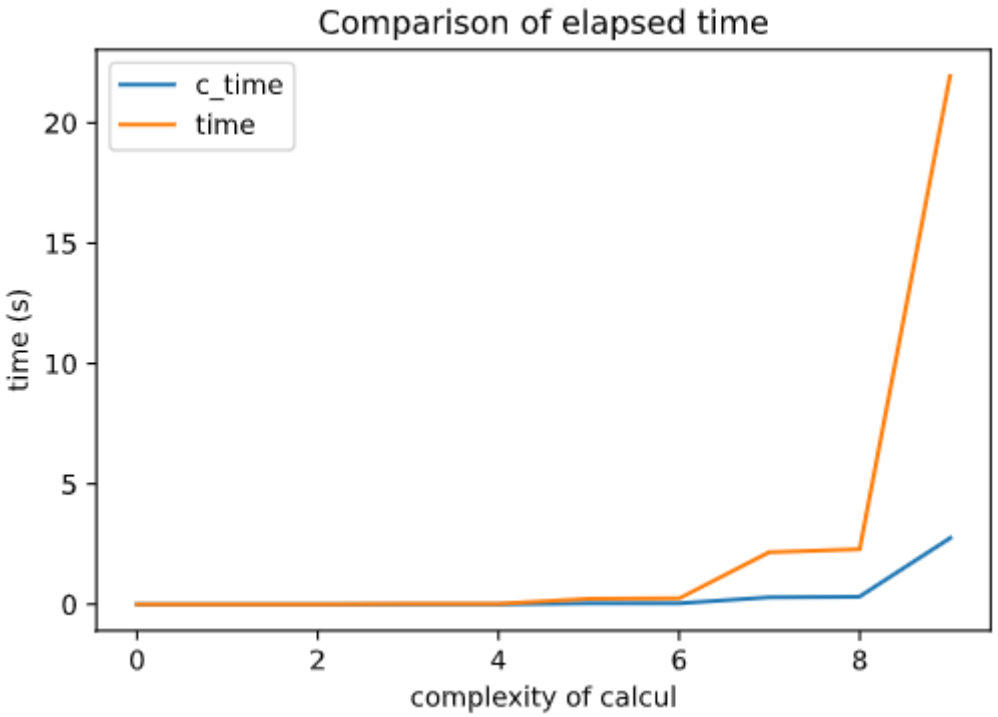
```
if(write_results){
    std::ofstream file;
    file.open(argv[1],std::ios::app);
    file << timeSteps << "," << n_samples << "," << errorEstimate1 << "," << NPV1 << ","
    << us1 << "," << errorEstimate2 << "," << NPV2 << "," << us2 << "," << std::endl;
    file.close();
}
```

write results in file

```
step,sample,c_err,c_npv,c_time,err,npv,time,
10,100,0.322852,4.13251,256,0.322852,4.13251,322,
10,1000,0.0994568,4.22306,349,0.0994568,4.22306,2541,
10,10000,0.0304028,4.17516,3527,0.0304028,4.17516,23431,
10,100000,0.00963383,4.16356,37449,0.00963383,4.16356,237482,
10,1000000,0.00304669,4.17073,315840,0.00304669,4.17073,2.28276e+06,
100,100,0.300936,4.32015,332,0.300936,4.32015,2298,
100,1000,0.0958765,4.18166,2843,0.0958765,4.18166,23083,
100,10000,0.030484,4.15086,35736,0.030484,4.15086,224465,
100,100000,0.00962119,4.17036,278741,0.00962119,4.17036,2.16902e+06,
100,1000000,0.00304455,4.17184,2.74929e+06,0.00304455,4.17184,2.19504e+07,
```

results

Firstly we compared the elapsed time. To do so, we sort time values in ascending order and we plot the two series on the same plot. We have done calculations for different timesteps and number of samples values.



In orange : time for general method ; in blue : time for constant method.

We notice that the more timesteps and samples the more gap between methods is important.

We also compute the sum of elapsed time and the results are :

- General method is 786 times longer in calculation time
- The relative difference is 686%

Then we compare value of NPV and estimated error and we notice that they were the same

	c_err	err
0	0.003045	0.003045
1	0.003047	0.003047
2	0.009621	0.009621
3	0.009634	0.009634
4	0.030403	0.030403
5	0.030484	0.030484
6	0.095877	0.095877
7	0.099457	0.099457
8	0.300936	0.300936
9	0.322852	0.322852

	c_npv	npv
0	4.13251	4.13251
1	4.15086	4.15086
2	4.16356	4.16356
3	4.17036	4.17036
4	4.17073	4.17073
5	4.17184	4.17184
6	4.17516	4.17516
7	4.18166	4.18166
8	4.22306	4.22306
9	4.32015	4.32015

left : estimated error / right : value of NPV
c_ for constant method

With these results for the NPV, we conclude that there should be an error in our implementation for the calculation of the NPV and so the estimation of the error that we did not find.