Derivatives Pricing Course

Lecture 5 – Monte Carlo methods

Artem Isaev

2016 CMF

Agenda

- Fundamentals.
- Generation of random samples.
- Digital options.
- C++ implementation.
- QuantLib results.

Monte Carlo methods

Previously discussed

- First way to price a derivative security is to find a closed-form solution of a parabolic PDE or to explicitly evaluate an expectation of a random variable.

- Second way is to find solutions numerically, and we have discovered how to find solutions to PDEs using finite-difference methods.

Monte Carlo methods
Following steps

- While the finite difference method is flexible and powerful, it has a number of limitations.
- First, its usage is restricted to problems where the state variable dynamics are Markovian.
- Second, for strongly path-dependent problems, the method often does not apply.
- And third, it is unsuited for problems where the dimension of the underlying is high (exponential growth in p, where p – the number of dimensions).

Monte Carlo methods

Previously discussed

- We will study the Monte Carlo method, a numerical technique where the computational effort grows only linearly in the problem dimension p.
- While the convergence of the Monte Carlo method is relatively slow, it is nearly always the method of choice for high-dimensional pricing problems.
- On the other hand, as Monte Carlo inherently run forward in time, pricing of American/Bermudan options becomes tedious.

Monte Carlo methods

Fundamentals

- Consider a European-style derivative V with time T payout $V(T) = g(T)$. Where finite-difference methods start with a PDE representation of the price of a contingent claim at times $t < T$, the starting point for the Monte Carlo method is the basic martingale relation:

$$V(t) = D(t){E_t}^Q(g(T)/D(T))$$

- To evaluate this expression numerically, we need a numerical technique to compute expectations of a random variable.

Monte Carlo methods

Fundamentals

- Theorem 5.1(Strong Law of Large Numbers) Let $Y_1, Y_2, \ldots$ be a sequence of i.i.d. random variables with expectation $\mu < \infty$. Define the sample mean:

$$\overline{Y}_n = \frac{1}{n}\sum_{i=1}^{n} Y_i \text{ then } \lim_{n\to\infty} \overline{Y}_n = \mu, \quad a.s.$$

- This result forms the basis for Monte Carlo method, which computes the expectation by simply
  1. generation independent realization of $g(T)/D(T)$
  2. forming their average

- Consider the expected convergence rate of the Monte Carlo method as n is increased.

Monte Carlo methods
Fundamentals

- Theorem 5.2(Central Limit Theorem) Let $Y_1, Y_2, \ldots$ be a sequence of i.i.d. random variables with expectation μ and standard deviation $\sigma < \infty$. Then, for $n \to \infty$:

$$\frac{\overline{Y}_n - \mu}{\sigma/\sqrt{n}} \to \mathcal{N}(0,1).$$

- Let's define a sample standard deviation:

$$s_n \triangleq \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{g_i D(t)}{D_i} - \overline{V}(t)\right)^2}$$

- The quantity $s_n/\sqrt{n}$ is known as the *standard error*.
- For a given level of percentile we can build a confidence interval for $V(t)$.

Monte Carlo methods

Fundamentals

- The rate at which the confidence interval contracts is $O(n^{-\frac{1}{2}})$.

- This is relatively slow: to reduce the width of the interval by a factor of 2, n must increase by a factor of 4.

- On the other hand, we notice that the convergence rate only depends on n, not on the specifics of g_i.

Monte Carlo methods

Generation of random samples

- At the most basic level, the Monte Carlo method requires the ability to draw independent realizations of a scalar random variable Z with a specified cumulative distribution function $F(z) = P(Z \leq z)$, where P is a probability measure.
- On a computer, the starting point for this exercise is a *pseudo-random number generator*, a software program that will generate a sequence of numbers uniformly distributed on $[0,1]$.
- The externally specified starting point is the *seed* of a generator.
- Also one should pay attention to the *period* length of the generator.
- Now we need to convert uniformly distributed numbers into draws from the distribution F of Z.

Monte Carlo methods

Inverse Transform Method

- Let U be a random variable uniformly distributed on $[0,1]$, and consider setting:

$$Z = F^{-1}(U)$$

as desired it has the needed distribution.

- Many distributions allow for closed-form inversion, however method depends on being able to compute F^{-1} fast.

- For the Gaussian distribution, no closed-form expression for the inverse distribution exists.

Monte Carlo methods

Acceptance-Rejection

- In cases where F^{-1} is cumbersome to compute, this method may be preferable.
- Suppose that we want to sample from a density
$$f(z) = dF(z)/dz$$
- And also suppose that we have a good method to sample from a density $e(z)$, where
$$e(z)c \geq f(z), z \in \mathbb{R}$$
for some positive constant c.
- Steps:
  1. Draw a sample Z from $\mathrm{e(z)}$.
  2. Draw an independent uniform variable U.
  3. Accept the sample Z if $U \leq f(Z)/\big(ce(Z)\big)$; otherwise discard it.

Monte Carlo methods

Correlated Gaussian samples

- In applications one may face the task of generating vectors of random variables, drawn from a joint multi-variate distribution.
- Recall that a p-dimensional Gaussian distribution $\mathcal{N}(\mu, \Sigma)$ is characterized by a density: $\varphi_p(z; \mu, \Sigma) = \frac{1}{(2\pi)^{p/2}(\det \Sigma)^{1/2}} \exp\left(-\frac{1}{2}(z-\mu)^T \Sigma^{-1}(z-\mu)\right)$
- Lemma 3.3. Let $Z \sim \mathcal{N}(\mu, \Sigma)$ be a p-dimensional. Given a $d \times p$ matrix A and a d-dimensional vector B, then
$$AZ + B \sim \mathcal{N}(A\mu + B, A\Sigma A^T)$$
- We can use this lemma as follows. Suppose that we generate p independent standard Gaussian samples - X vector. Define a $(p \times p) -$ dimensional matrix C satisfying $CC^T = \Sigma$.
- Then $Z = \mu + CX$ is distributed $\mathcal{N}(\mu, \Sigma)$.
- We need to determine a matrix C.

Monte Carlo methods

Matrix decomposition

- Cholesky decomposition – we impose the constraint that the matrix C is lower triangular.

- For instance, if $\Sigma = \begin{pmatrix} {\sigma_1}^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & {\sigma_2}^2 \end{pmatrix}$

- Then $\mathrm{C} = \begin{pmatrix} \sigma_1 & 0 \\ \sigma_2\rho & \sigma_2\sqrt{1-\rho^2} \end{pmatrix}$

- If the matrix Σ is only positive semi-definite (but not positive definite), the Cholesky decomposition will fail.

Monte Carlo methods

Matrix decomposition

- As an alternative to Cholesky decomposition, we can also consider diagonalizing Σ through an eigenvalue decomposition.

$$\Sigma = E\Lambda E^T$$

  where Λ is a diagonal matrix of eigenvalues.

- Implies that one choice of C is

$$C = E\sqrt{\Lambda}$$

Monte Carlo methods

Double No-Touch option

- Double No-Touch option pays the nominal value N if the spot price of the underlying asset is between two numbers, the lower K_1 and upper K_2 strikes of the option.

- The payoff:

$$g(T) = N \cdot \mathbb{I}_{\{K_1 \le S(T) \le K_2\}}$$

Monte Carlo methods

Double One-Touch option

- Double One-Touch option pays the nominal value N if the spot price of the underlying asset reaches any of two numbers, the lower K_1 and upper K_2 strikes of the option.

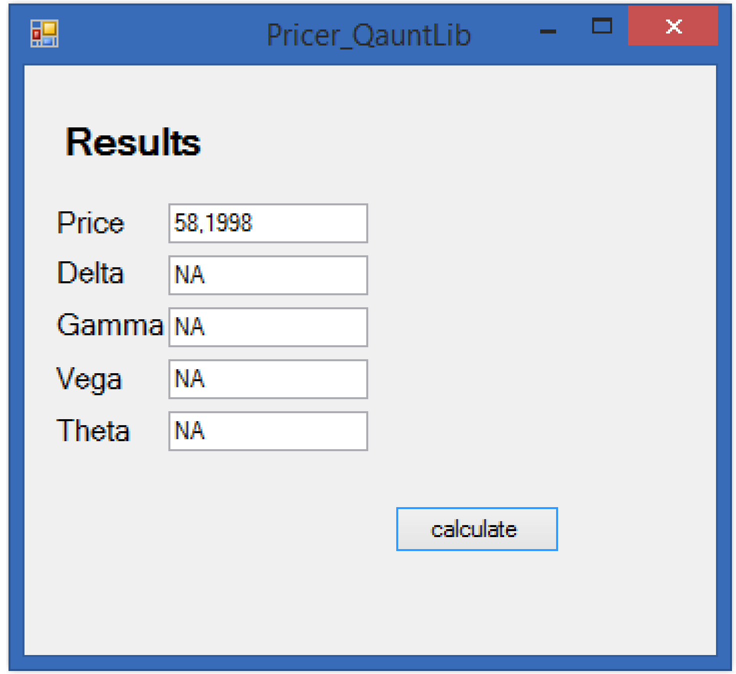
- The payoff:

$$g(T) = N \cdot \mathbb{I}_{\{K_1 \leq S(T)\} \cup \{S(T) \geq K_2\}}$$

Monte Carlo methods

What is the price of one-touch with the same parameters?

- No-Touch option
  - $S = 100$
  - $r = 0$
  - $\sigma = 20\%$
  - $K_1 = 70$
  - $K_2 = 120$
  - $N = 100$



- Closed-form formula as an infinite series exists.

C++ code

Payoff(.h)

```
#ifndef PAYOFF_H
#define PAYOFF_H

//Abstract class to define the interface
class Payoff
{
public:
            Payoff(){};
            virtual ~Payoff(){}
            virtual double operator()(double Spot) const = 0;
            virtual Payoff* clone() const = 0;
};

class PayoffCall : public Payoff
{
public:
            PayoffCall(double Strike_);
            double operator()(double Spot) const;
            ~PayoffCall(){}
            Payoff* clone() const;
private:
            double Strike;
};

#endif
```

C++ code

Payoff(.cpp)

```
#include "Payoff.h"

PayoffCall::PayoffCall(double Strike_) : Strike(Strike_){}

double PayoffCall::operator()(double Spot) const
{
            double result = (Spot > Strike) ? (Spot - Strike) : 0;
            return result;
}

Payoff* PayoffCall::clone() const{
            //Copy constructor is called, ok since no dynamic objects
            return new PayoffCall(*this);
}
```

C++ code

OptionClass(.h)

```cpp
#ifndef OPTIONCLASS_H
#define OPTIONCLASS_H

#include "Bridge.h"

//Abstract class for Option - defined interface, overload payoff function in inherited classes
//to return payoff at expiry
class SimpleOptionMC
{
public:
		SimpleOptionMC(const Bridge& thePayoff_, double Time_);
		double optionPayoff(double Spot) const;
		double getTime() const;
private:
		double Time;
		//Bridge to separate Option class from Payoff class realization
		Bridge thePayoff;
};

#endif
```

C++ code

OptionClass(.cpp)

```
#include "OptionClass.h"


SimpleOptionMC::SimpleOptionMC(const Bridge& thePayoff_, double Time_) :

            thePayoff(thePayoff_), Time(Time_)
{
}

double SimpleOptionMC::optionPayoff(double Spot) const
{
            return thePayoff(Spot);
}

double SimpleOptionMC::getTime() const
{
            return Time;
}
```

C++ code

Bridge(.h)

```cpp
#ifndef BRIDGE_H
#define BRIDGE_H

#include "Payoff.h"

//New class created for memory handling and copying
class Bridge
{
public:
            //For type conversion Payoff <-> Bridge
            Bridge(const Payoff& initialPayoff);
            //Copy constructor for Bridge
            Bridge(const Bridge& initial);
            double operator()(double Spot) const;
            //Class stores a pointer, memory was allocated with new
            ~Bridge();
private:
            Payoff* thePayoff;
};

#endif
```

C++ code
DoubleNoTouch(.h)

```
#ifndef DOUBLENOTOUCH_H
#define DOUBLENOTOUCH_H

#include "Payoff.h"

class PayOffDoubleNoTouch : public Payoff
{
public:
		PayOffDoubleNoTouch(double LowerBarrier_, double UpperBarrier_);
		virtual double operator()(double Spot) const;
		virtual ~PayOffDoubleNoTouch(){}
		Payoff* clone() const;
private:
		double LowerBarrier;
		double UpperBarrier;
};

#endif
```

C++ code

DoubleNoTouch(.cpp)

```cpp
#include "DoubleNoTouch.h"

PayOffDoubleNoTouch::PayOffDoubleNoTouch(double LowerBarrier_, double UpperBarrier_)
            : LowerBarrier(LowerBarrier_), UpperBarrier(UpperBarrier_)
{
}
double PayOffDoubleNoTouch::operator()(double Spot) const
{
            if (Spot <= LowerBarrier)
                        return 0.0;
            if (Spot >= UpperBarrier)
                        return 0.0;
            return 100.0;
}

Payoff* PayOffDoubleNoTouch::clone() const{
            //Copy constructor is called, ok since no dynamic objects
            return new PayOffDoubleNoTouch(*this);
}
```

C++ code

MonteCarloRoutine(.h)

```
#ifndef MONTECARLOROUTINE_H
#define MONTECARLOROUTINE_H

#include "OptionClass.h"

double* MC(const SimpleOptionMC& theOption, double Spot,
                                            double Vol,
                                            double Rate,
                                            int numberOfPaths);


#endif
```

Homework assignment 4

- Modify program to price one-touch option.
- Suggest alternative Normal random variables generating algorithm.
- Convergence plot
    - Fix any spot level
    - X-axis – number of simulations, Y-axis – value of an option

- **Deadline** – 6th May EOD