OPTION PRICING SIMULATION USING C++

Argamaya

Faculty of Economics, Atmajaya University

Jl. Jend. Sudirman Jakarta Pusat

Email: argamaya@telkom.net

ABSTRACT

In the field of financial mathematics, many problems, for instance the problem of finding the arbitrage-free value of a particular derivative security, boil down to the computation of a particular integral. In many cases these integrals can be valued analytically, and in still more cases they can be valued using numerical integration. However when the number of dimensions (or degrees of freedom) in the problem is large, numerical integration methods become intractable. In these cases it is common to resort to the more widely applicable ***Monte Carlo method*** *to solve the problem. For large dimension integrals as can very often happen in financial problems, Monte Carlo methods converge to the solution more quickly than numerical integration methods. The advantage of Monte Carlo methods increases as the dimension of the problem gets larger. This article introduces Monte Carlo techniques for option pricing. It also touches on the use of so-called "variance reduction technique". This method can produce enormous speed-ups compared with standard Monte Carlo.*

Keywords: Monte Carlo, financial problems, variance reduction technique.

INTRODUCTION

The pricing of options and related instruments has been a major breakthrough for the use of financial theory in practical application. Since the original papers of Black and Scholes (1973) and Merton (1973), there has been a wealth of practical and theoretical applications. In this paper we will discuss ways of calculating the price of an option in the setting discussed in these original papers. The discussion is not completed, it needs to be supplemented by one of the standard textbooks, like Hull (2005).

We consider using Monte Carlo methods to estimate the price of an European option, and let us first consider the case of the ''usual" European Call, which is priced by the Black Scholes equation. Since there is already a closed form solution for this case, it is not really necessary to use simulations, but we use the case of the standard call for illustrative purposes.

At maturity, a call option is worth

$$c_T = \max(0, S_T - X)$$

At an earlier date t, the option value will be the expected present value of this.

$$c_t = E\left[PV\left(\max(0, S_T - X)\right)\right]$$

Now, an important simplifying feature of option pricing is the ''risk neutral result," which implies that we can treat the (suitably transformed) problem as the decision of a risk neutral decision maker, if we also modify the expected return of the underlying asset such that this earns the risk free rate.

$$c_t = e^{-r(T-t)} E^*\left[\max(0, S_{T-X})\right]$$

where $E^*[\bullet]$ is a transformation of the original expectation. One way to estimate the value of the call is to simulate a large number of sample values of S_T according to the assumed price process, and find the estimated call price as the average of the simulated values. By appealing to a law of large numbers, this average will converge to the actual call value, where the rate of convergence will depend on how many simulations we perform.

Simulating Lognormally Distributed Random Variables

Lognormal variables are simulated as follows. Let $\tilde{x}$ be normally distributed with mean zero and variance one. If S_t follows a lognormal distribution, then the one-period-later price S_{t+1} is simulated as

$$S_{t+1} = S_t e^{\left(r-\frac{1}{2}\sigma^2\right)+\sigma\tilde{x}},$$

or more generally, if the current time is t and terminal date is T, with a time between t and T of $(T-t)$,

$$S_T = S_t e^{\left(r-\frac{1}{2}\sigma^2\right)(T-t)+\sigma\sqrt{T-t}\,\tilde{x}},$$

Simulation of lognormal random variables is illustrated by listing 1.

```
#include <cmath>
using namespace std;
#include "normdist.h"

double simulate_lognormal_random_variable(const double& S, // current value of variable
                                          const double& r, // interest rate
                                          const double& sigma, // volatility
                                          const double& time) { // time to final date
   double R = (r - 0.5 * pow(sigma,2) )*time;
   double SD = sigma * sqrt(time);
   return S * exp(R + SD * random_normal());
};
```

Listing 1: Simulating a lognormally distributed random variable

Pricing Of European Call Options

For the purposes of doing the Monte Carlo estimation of the price if an European call

$$c_t = e^{-r(T-t)}E[\max(0,S_T - X)],$$

note that here one merely need to simulate the terminal price of the underlying, S_T, the price of the underlying at any time between t and T is not relevant for pricing. We proceed by simulating lognormally distributed random variables, which gives us a set of observations of the terminal price S_T. If we let $S_{T,1}, S_{T,2}, S_{T,3}, \ldots, S_{T,n}$ denote the n simulated values, we will estimate $E^*[\max(0,S_T - X)]$ as the average of option payoffs at

maturity, discounted at the risk free rate.

$$\hat{c}_t = e^{-r(T-t)}\left(\sum_{i=1}^{n}\max\left(0, S_{T,i} - X\right)\right)$$

Listing 2 shows the implementation of a Monte Carlo estimation of an European call option.

```
#include <cmath> // standard mathematical functions
#include <algorithm> // define the max() function
using namespace std;
#include "normdist.h" // definition of random number generator

double
option_price_call_european_simulated( const double& S, // price of underlying
                                      const double& X, // exercise price
                                      const double& r, // risk free interest rate
                                      const double& sigma, // volatility of underlying
                                      const double& time, // time to maturity (in years)
                                      const int& no_sims){ // number of simulations
    double R = (r - 0.5 * pow(sigma,2))*time;
    double SD = sigma * sqrt(time);
    double sum_payoffs = 0.0;
    for (int n=1; n<=no_sims; n++) {
        double S_T = S* exp(R + SD * random_normal());
        sum_payoffs += max(0.0, S_T-X);
    };
    return exp(-r*time) * (sum_payoffs/double(no_sims));
};
```

Listing 2: European Call option priced by simulation

Hedge Parameters

It is of course, just as in the standard case, desirable to estimate hedge parameters as well as option prices. We will show how one can find an estimate of the option *delta*, the first derivative of the call price with respect to the underlying security: $\Delta = \frac{\partial c_t}{\partial S}$. To understand how one goes about estimating this, let us recall that the first derivative of a function f is defined as the limit

$$f'(x) = \lim_{h\to 0}\frac{f(x+h) - f(x)}{h}$$

Thinking of $f(S)$ as the option price formula $c_t = f(S; X, \tau, \sigma, (T-t))$, we see that we can evaluate the option price at two different values of the underlying, S and $S+q$, where q is a small quantity, and estimate the option delta as

$$\hat{\Delta} = \frac{f(S+q) - f(S)}{q}$$

In the case of Monte Carlo estimation, it is very important that this is done by using the same sequence of random variables to estimate the two option prices with prices of the underlying S and $S+q$. Listing 3 implements this estimation of the option delta.

One can estimate other hedge parameters in a simular way.

```
#include <cmath> // standard mathematical functions
#include <algorithm> // define the max() function
using namespace std;
#include "normdist.h" // definition of random number generator

double option_price_delta_call_european_simulated(const double& S,
                                                  const double& X,
                                                  const double& r,
                                                  const double& sigma,
                                                  const double& time,
                                                  const int& no_sims){
    double R = (r - 0.5 * pow(sigma,2))*time;
    double SD = sigma * sqrt(time);
    double sum_payoffs = 0.0;
    double sum_payoffs_q = 0.0;
    double q = S*0.01;
    for (int n=1; n<=no_sims; n++) {
        double Z = random_normal();
        double S_T = S* exp(R + SD * Z);
        sum_payoffs += max(0.0, S_T-X);
        double S_T_q = (S+q)* exp(R + SD * Z);
        sum_payoffs_q += max(0.0, S_T_q-X);
    };
    double c = exp(-r*time) * ( sum_payoffs/no_sims);
    double c_q = exp(-r*time) * ( sum_payoffs_q/no_sims);
    return (c_q-c)/q;
};
```

Listing 3: Estimate Delta of European Call option priced by Monte Carlo

More General Payoffs. Function Prototypes

The above shows the case for a call option. If we want to price other types of options, with different payoffs we could write similar routines for every possible case. But this would be wasteful, instead a bit of thought allows us to write option valuations for any kind of option whose payoff depend on the value of the underlying at maturity, only. Let us now move toward a generic routine for pricing derivatives with Monte Carlo. This relies on the ability of C ** to write subroutines which one call with *function prototypes*, i.e. that in the call to the subroutine/function one provides a function instead of a variable. Consider pricing of standard European put and call options. At maturity each option only depend on the value of the underlying S_T and the exercise price X through the relations

$$C_T = \max(S_T - X, 0)$$

$$P_T = \max(X - S_T, 0)$$

Listing 4 shows two C** functions which calculates this.

```
#include <algorithm>
using namespace std;

double payoff_call(const double& price, const double& X){
    return max(0.0,price-X);
};

double payoff_put (const double& price, const double& X) {
    return max(0.0,X-price);
};
```

Listing 4: Payoff call and put options

The interesting part comes when one realizes one can write a generic simulation routine to which one provide one of these functions, or some other function describing a payoff which only depends on the price of the underlying and some constant. Listing 5 shows how this is done.

```
#include <cmath>
using namespace std;
#include "fin_recipes.h"

double
derivative_price_simulate_european_option_genetic(const double& S, // price of underlying
                                                   const double& X, // used by user provided payoff function
                                                   const double& r, // risk free interest rate
                                                   const double& sigma, // volatility
                                                   const double& time, // time to maturity
                                                   double payoff(const double& price, const double& X),
                                                   // user provided function
                                                   const int& no_sims) { // number of simulations to run
    double sum_payoffs=0;
    for (int n=0; n<no_sims; n++) {
      double S_T = simulate_lognormal_random_variable(S,r,sigma,time);
      sum_payoffs += payoff(S_T,X);
    };
    return exp(-r*time) * (sum_payoffs/no_sims);
};
```

Listing 5: Generic simulation pricing

Note the presence of the line double payoff(const double& price, const double& X), in the subroutine call. When this function is called, the calling program will need to provide a function to put there, such as the Black Scholes example above. Listing 6 shows a complete example of how this is done.

```
#include "fin_recipes.h"
#include <algorithm>
#include <iostream>
using namespace std;

double payoff_european_call(const double& price, const double& X){ return max(0.0,price-X); };
double payoff_european_put (const double& price, const double& X) { return max(0.0,X-price); };

int main(){
    double S     = 100.0;
    double X     = 100.0;
    double r     = 0.1;
    double sigma = 0.25;
    double time  = 1.0;
    int no_sims  = 50000;
    cout << "Black Scholes call option price = "
         << option_price_call_black_scholes(S,X,r,sigma,time)
         << endl;
    cout << "Simulated call option price = "
         << derivative_price_simulate_european_option_generic(S,X,r,sigma,time,payoff_european_call,no_sims)
         << endl;
    cout << "Black Scholes put option price = "
         << option_price_put_black_scholes(S,X,r,sigma,time)
         << endl;
    cout << "Simulated put option price = "
         << derivative_price_simulate_european_option_genetic(S,X,r,sigma,time,payoff_european_put,no_sims)
         << endl;
};
```

Listing 6: Simulating Black Scholes values using the generic routine

Running the program in listing 6 results in the output:

Simulated call option price = 14.995

Black Scholes call option price = 14.9758

Simulated put option price = 5.5599

Black Scholes put option price = 5.45954

As we see, even with as many as 50,000 simulations, the option prices estimated using Monte Carlo still differs substantially from the ``true'' values.

Improving The Efficiency In Simulation

There are a number of ways of ``improving'' the implementation of Monte Carlo estimation such that the estimate is closer to the true value. Two Variance Reduction techniques, the method of Antithetic Variates and the method of Control Variates will be discussed. More information about the general use of variance reduction techniques can be found in the textbook by Law and Kelton (2000).

Control variates

One is the method of control variates. The idea is simple. When one generates the set of terminal values of the underlying security, one can value several derivatives using the same set of terminal values. What if one of the derivatives we value using the terminal values is one which we have an analytical solution to? For example, suppose we calculate the value of an at the money European call option using both the (analytical) Black

Scholes formula and Monte Carlo simulation. If it turns out that the Monte Carlo estimate overvalues the option price, we think that this will also be the case for other derivatives valued using the same set of simulated terminal values. We therefore move the estimate of the price of the derivative of interest downwards.

Thus, suppose we want to value an European put and we use the price of an at the money Euro-pean call as the control variate. Using the same set of simulated terminal values $S_{T,i}$, we estimate the two options using Monte Carlo as:

$$\hat{p}_t = e^{-r(T-t)}\left(\sum_{i=1}^{n}\max\left(0, X - S_{T,i}\right)\right)$$

$$\hat{c}_t = e^{-r(T-t)}\left(\sum_{i=1}^{n}\max\left(0, S_{T,i} - X\right)\right)$$

We calculate the Black Scholes value of the call $\hat{c}_t^{bs}$, and calculate p_t^{cv}, the estimate of the put price with a control variate adjustment, as follows

$$\hat{p}_t^{cv} = \hat{p}_t + \left(c_t^{bs} - \hat{c}_t\right)$$

One can use other derivatives than the at-the-money call as the control variate, the only limitation being that it has a tractable analytical solution.

Listing 7 shows the implementation of a Monte Carlo estimation using an at-the-money European call as the control variate.

```
#include <cmath>
using namespace std;
#include "fin_recipes.h"
#include "payoff_black_scholes_case.h"

double
derivative_price_simulate_european_option_genetic_with_control_variate(const double& S,
                                                                        const double& X,
                                                                        const double& r,
                                                                        const double& sigma,
                                                                        const double& time,
                                                                        double payoff(const double& S,
                                                                                      const double& X),
                                                                        const int& no_sims) {
    double c_bs = option_price_call_black_scholes(S,S,r,sigma,time);// price an at the money Black Scholes call
    double sum_payoffs=0;
    double sum_payoffs_bs=0;
    for (int n=0; n<no_sims; n++) {
        double S_T= simulate_lognormal_random_variable(S,r,sigma,time);
        sum_payoffs += payoff(S_T,X);
        sum_payoffs_bs += payoff_call(S_T,S); // simulate at the money Black Scholes price
    };
    double c_sim = exp(-r*time) * (sum_payoffs/no_sims);
    double c_bs_sim = exp(-r*time) * (sum_payoffs_bs/no_sims);
    c_sim += (c_bs-c_bs_sim);
    return c_sim;
};
```

Listing 7: Generic with control variate

Antithetic variates.

An alternative to using control variates is to consider the method of *antithetic* variates. The idea behind this is that Monte Carlo works best if the simulated variables are "spread" out as closely as possible to the true distribution. Here we are simulating unit normal random variables. One property of the normal is that it is symmetric around zero, and the median value is zero. Why don't we enforce this in the simulated terminal values? An easy way to do this is to first simulate a unit random normal variable Z, and then use both Z and $-Z$ to generate the lognormal random variables. Listing 8 shows the implementation of this idea.

```
#include "fin_recipes.h"
#include "normdist.h"
#include <cmath>
using namespace std;

double
derivative_price_simulate_european_option_generic_with_antithetic_variate(const double& S,
                                                                           const double& X,
                                                                           const double& r,
                                                                           const double& sigma,
                                                                           const double& time,
                                                                           double payoff(const double& S,
                                                                                         const double& X),
                                                                           const int& no_sims) {
    double R = (r - 0.5 * pow(sigma,2) )*time;
    double SD = sigma * sqrt(time);
    double sum_payoffs=0;
    for (int n=0; n<no_sims; n++) {
        double x=random_normal();
        double S1 = S * exp(R + SD * x);
        sum_payoffs += payoff(S1,X);
        double S2 = S * exp(R + SD * (-x));
        sum_payoffs += payoff(S2,X);
    };
    return exp(-r*time) * (sum_payoffs/(2*no_sims));
};
```

Listing 8: Generic with antithetic variates

Boyle (1977) shows that the efficiency gain with antithetic variates is not particularly large. There are other ways of ensuring that the simulated values really span the whole sample space, sometimes called ``pseudo Monte Carlo."

Simulation Example

Let us see how these improvements change actual values. We use the same numbers as in listing 6, but add estimation using control and antithetic variates. Listing 9 shows a complete example of how this is done.

```
#include "fin_recipes.h"
#include <algorithm>
#include <iostream>
using namespace std;

double payoff_european_call(const double& price, const double& X){ return max(0.0,price-X); };
double payoff_european_put (const double& price, const double& X) { return max(0.0,X-price); };

int main(){
    double S     = 100.0;
    double X     = 100.0;
    double r     = 0.1;
    double sigma = 0.25;
    double time  = 1.0;
    int no_sims  = 50000;
    cout << "Black Scholes call option price = "
         << option_price_call_black_scholes(S,X,r,sigma,time)
         << endl;
    cout << "Simulated call option price = "
         << derivative_price_simulate_european_option_generic(S,X,r,sigma,time,
                                                    payoff_european_call,no_sims)
         << endl;
    cout << "Simulated call option price, CV = "
         << derivative_price_simulate_european_option_generic_with_control_variate(S,X,r,sigma,time,
                                                        payoff_european_call,no_sims)
         << endl;
    cout << "Simulated call option price, AV = "
         << derivative_price_simulate_european_option_generic_with_antithetic_variate(S,X,r,sigma,time,
                                                        payoff_european_call,no_sims)


       << endl;
   cout << "Black Scholes put option price = "
        << option_price_put_black_scholes(S,X,r,sigma,time)
        << endl;
   cout << "Simulated put option price = "
        << derivative_price_simulate_european_option_generic(S,X,r,sigma,time,payoff_european_put,no_sims)
        << endl;
   cout << "Simulated put option price, CV = "
        << derivative_price_simulate_european_option_generic_with_control_variate(S,X,r,sigma,time,
                                                        payoff_european_put,no_sims)
        << endl;
   cout << "Simulated put option price, AV = "
        << derivative_price_simulate_european_option_generic_with_antithetic_variate(S,X,r,sigma,time,
                                                        payoff_european_put,no_sims)
        << endl;
};
```

Listing 9: Simulationg Black Scholes values using the generic Monte Carlo routine, with efficiency improvements

Running this program results in the output:

Black Scholes call option price = 14.9758
Simulated call option price = 14.995
Simulated call option price, CV = 14.9758
Simulated call option price, AV = 14.9919
Black Scholes put option price = 5.45954
Simulated put option price = 5.41861
Simulated put option price, CV = 5.42541
Simulated put option price, AV = 5.46043

More Exotic Options

These generic routines can also be used to price other options. Any European option that only depends on the terminal value of the price of the underlying security can be valued. Consider the *binary* options discussed by e.g. Hull (2005). An *cash or nothing call* pays a fixed amount Q if the price of the asset is above the exercise price at maturity, otherwise nothing. An *asset or nothing call* pays the price of the asset if the price is above the exercise price at maturity, otherwise nothing. Both of these options are easy to implement using the generic routines above, all that is necessary is to provide the payoff functions as shown in listing 10..

```
double payoff_cash_or_nothing_call(const double& price, const double& X){
    double Q=1;
    if (price>=X) return Q;
    return 0;
};

double payoff_asset_or_nothing_call(const double& price, const double& X){
    if (price>=X) return price;
    return 0;
};
```

Listing 10: Payoff binary options

Now, many exotic options are not simply functions of the terminal price of the underlying security, but depend on the evolution of the price from ``now'' till the terminal date of the option. For example options that depend on the average of the price of the underlying (Asian options). For such cases one will have to simulate the whole path. We will return to these cases in the chapter on pricing of exotic options.

CONCLUSION

Variance reduction techniques offer potentially large increases in the precision of estimated derivative values. The method of Antithetic Variates (AV) is generally less effective than Control Variates (CV), but AV can be easily applied to more types of derivatives than CV because CV requires that a control variate is available. When applicable, the combination of AV and CV can increase precision even further.

Interest in use of Monte Carlo methods for derivatives pricing is increasing because of the flexibility of the method in handling complex financial instruments. Monte Carlo simulation will continue to gain appeal as financial instruments become more complex, workstations become faster, and simulation software is adopted by more users. The use of variance reduction techniques along with the greater power of today's workstations can help to reduce the execution time required for achieving acceptable precision to the point that simulation can be used by financial traders to value derivatives in real time.

REFERENCES

Black, F. and Scholes, M.S.(1973). "The pricing of options and corporate liabilities." **Journal of Political Economy**, 7: 637-54

Boyle, P. (1977). "Options: A Monte Carlo Approach". **Journal of Financial Economics**, May 1977, pp.323-338

Glasserman, P. (2004); **Monte Carlo Methods in Financial Engineering,** Springer-Verlag New York, Inc.

Hull, J. (2005). **Options, Futures and other Derivatives.** Sixth Edition, Prentice-Hall.

Jackel, P. (2002). **Monte Carlo methods in finance**. John Wiley and Sons.

Law, A. M. and Kelton, W. D. (2000). **Simulation Modeling and Analysis,** 3rd edition. New York: McGraw-Hill.

Merton, R. C.(1973). "The theory of rational option pricing" **Bell Journal**, 4: 141-183