MÄLARDALENS HÖGSKOLA

Mälardalen University

Department of Mathematics and Physics

Monte Carlo Simulation:

European and Asian Options Pricing

Supervisor:
Jan R. M. Röman

Authors:
Dmytro S. Yefymov, MSc
Natalia Arango Mesa
Yunior Gutierrez Berrones

Västerås - 2007

SUMMARY

The task of the paper is to describe simulation program for European and Asian options. Numerical example represents calculation for European Call option and accompanied with histogram of simulated values. The option pricing is performed using Monte Carlo simulation algorithm. The flowchart and the source code of the algorithm realization as well as graphical user interface description are given. Appendix contains computer realization of Marsaglia-Bray algorithm for Box-Müller method. The code is written in C# using .Net Framework 2.0.

Key words: *Monte Carlo Simulation, European and Asian Options*

Contents

1. The Task and Realization

The Task

The task of the paper is to describe simulation program for European and Asian options. The option pricing is performed using Monte Carlo simulation algorithm.

European Options

Fist we will use Monte Carlo for getting price for a European call option. Although for this purpose we can use Black-Scholes formula, computer simulation is also a suitable tool. The algorithm is the following.

We know that the stock price of a share at moment *T* could be defined as

$$S(T) = S(0)exp\left(\left[r - \frac{1}{2}\sigma^2\right]T + \sigma\sqrt{T}Z\right)$$

where *S(0)* and *S(T)* are the prices of the share at moment 0 and T respectively, r is a risk-free interest rate, σ is volatility, $Z \sim N(0,1)$.

Basing on this formula we define price for a European call option as

$$C = e^{-rT}max(S(T) - K, 0)$$

where *K* is a strike price. The unbiased estimator of *C* is

$$E[\hat{C}_n] = C \overset{\text{def}}{=} E[e^{-rT}(S(T) - K)]^+$$

The estimator is strongly consistent, meaning

$$P\left(\hat{C}_n \to C\right) = 1 \text{ as } n \to \infty$$

For a finite but at least moderately large n, we can supplement the point estimate $\hat{C}_n$ with a confidence interval

$$\hat{C}_n \pm z_{\delta/2}\frac{1}{\sqrt{n}}\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(C_i - \hat{C}_n\right)^2}$$

Asian Options

An Asian option is a generic name for the class of options whose terminal payoff is based on average asset values during some period within the options' lifetimes. Due to their averaging feature, Asian options are particularly suitable for thinly traded assets or commodities. In contrast to standard options, Asian options are more robust with respect to manipulations near their expiry dates. Typically, they are less expensive than standard options.

Let T be the exercise dote, and let $0 \leq T_0 \leq T$ stand for the beginning date of the averaging period. Then the payoff at expiry of an Asian call option equals

$$C_T^A \overset{\text{def}}{=} (A_S(T_0, T) - K)^+,$$

where

$$A_S(T_0, T) = \frac{1}{T - T_0} \int_{T_0}^{T} S_u du$$

Is the arithmetic average of the asset price over the time interval$[T - T_0]$, K is the fixed strike price, and S is the price of the stock that is assumed to follow a geometric Brownian motion.

The main difficulty in pricing and hedging Asian options is due to the fact that the random variable $A\ (T_0,\ T)$ does not have a lognormal distribution. This feature makes the task of finding an explicit formula for the price of an Asian option difficult. So the Monte Carlo is a reasonable solution.

Numerical Example

To solve this part we will use Monte Carlo simulation program built according to principles stated above. We take the European Call option with the following parameters:

- Initial price (S_0) = 200;
- Strike price (K) = 190;
- Volatility (σ) = 0.59;
- Interest Rate (r) = 0.14;
- Length (T) = 64 days.

After 10 000 iterations we get the simulated value of the option equal to 10.56 SEK (see Figure 2). The next figure shows the histogram of the simulated values.

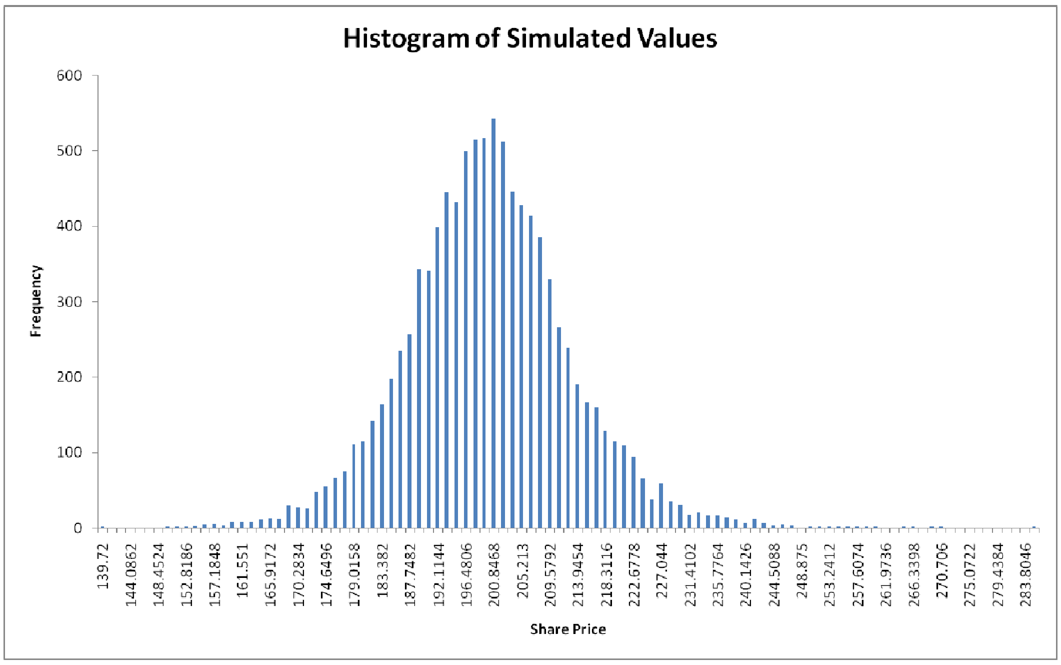
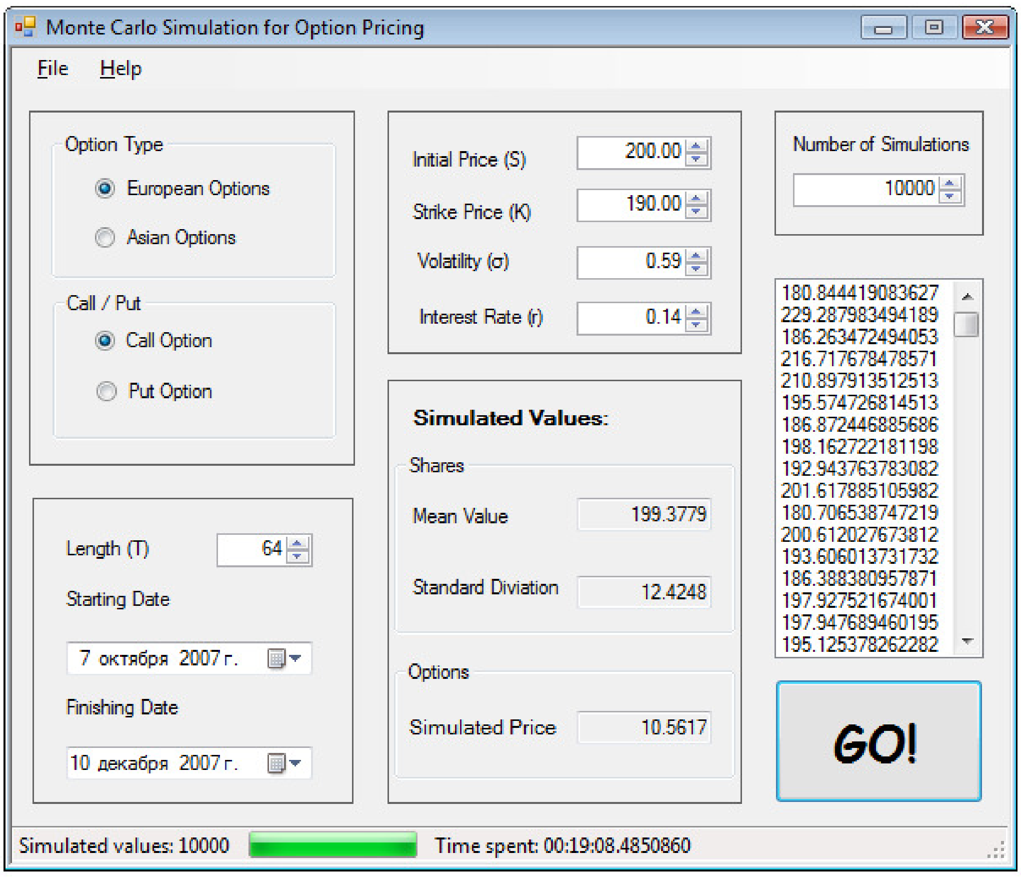


Figure 1. Histogram of Simulated Stock Prices

2. Graphical User Interface


Monte Carlo Simulation for Option Pricing

File Help

Option Type
[✓ European Options]
Asian Options

Call / Put
[✓ Call Option]
Put Option

Initial Price (S) 200.00
Strike Price (K) 190.00
Volatility (σ) 0.59
Interest Rate (r) 0.14

Number of Simulations 10000

```
180.844419083627
229.287983494189
186.263472494053
216.717678478571
210.897913512513
195.574726814513
186.872446885686
198.162722181198
192.943763783082
201.617885105982
180.706538747219
200.612027673812
193.606013731732
186.388380957871
197.927521674001
197.947689460195
195.125378262282
```

Length (T) 64
Starting Date
7 октября 2007 г.
Finishing Date
10 декабря 2007 г.

Simulated Values:

Shares
Mean Value 199.3779
Standard Diviation 12.4248

Options
Simulated Price 10.5617

GO!

Simulated values: 10000 Time spent: 00:19:08.4850860

Figure 2. Graphical User Interface

To get simulated value of an option select option type (European/Asian) and Call/Put, length (in days) or Starting and Finishing dates, initial and strike prices, volatility and risk-free interest rate. After defining the number of iterations and pushing [GO!] button the simulated shares' values (for European options) or average price for the period (for Asian). These simulated values are added to element listShare.

The Status Bar shows the number of simulated shares and the time spent for it. Note that with the large number of iteration, the simulation may take a long time. This is primary due to the low speed of random number generation. However, even if the program will stop to respond to command for some period of time, it still works and performs simulation.

To export simulated values to .txt file go [File] -> [Export Simulated] and choose the name of the new file. These values then could be processed with other software (*R*, SPSS, Excel, etc).

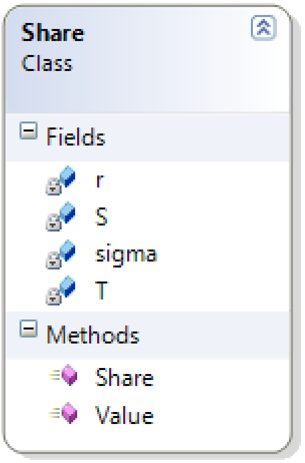
The description of source code of GUI realization is beyond the scope of the paper. The next part will explain the internal structure of the program which is written in C#.

3. Inside the Program

The computer realization of the Monte Carlo Simulator consists of Main() function and two classes: ‘Shares’ and ‘Options’. First classes are given, then are the flowchart and code of Main().

Class ‘Shares’

Class ‘Shares’ simulates a value of a share basing on the variables:



```
double S; // initial prise
double r; // risk-free interest rate
double sigma; // volatility
double T; // time to maturity
```

The creator of the class is the following:

```
/// </summary>
/// Creates Share with given parametors
/// </summary>
/// <param name="cS">initial prise</param>
/// <param name="cr">risk-free interest rate</param>
/// <param name="csigma">volatility</param>
/// <param name="cT">time to maturity (days)</param>
public Share(decimal cS, decimal cr,decimal csigma, decimal cT)
{
    S = (double)cS;
    r = (double)cr;
    sigma = (double)csigma;
    T = (double)cT/365;
}
```

The main function of interest in the class is method ‘Value()’:

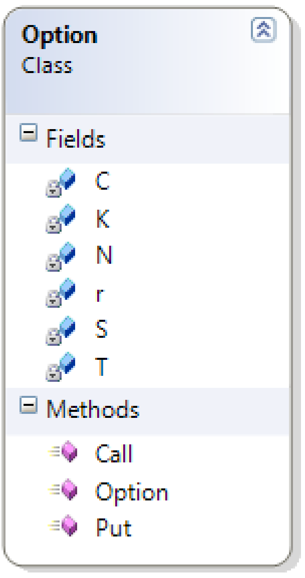
```
/// <summary>
/// Calculates the value of the share
/// </summary>
public decimal Value()
{
    double Z = norm.rand();
    double St = S * Math.Exp((r-5*sigma*sigma)*T+sigma*Math.Sqrt(T)*Z);
    return (decimal)St;
}
```

The random variable *Z~N(0,1)* is generated using Marsaglia-Bray algorithm for Box-Müller method that has the following pseudo-code:

```
While (X > 1)
        generate $U_1,U_2$ ~ Unif[0,1]
        $U_1 \leftarrow 2*U_1-1$,  $U_2 \leftarrow 2*U_2-1$
        $X \leftarrow U_1^2+U_2^2$
End
$Y \leftarrow \sqrt{-2\ln X/X}$
$Z_1 \leftarrow U_1Y$ ,   $Z_2 \leftarrow U_2Y$
Return  $Z_1,Z_2$.
```

Its computer realization is presented in Appendix A.

Class ‘Options’



Class ‘Options’ is needed to simulate the value of call and put options basing on the following variables:

```
decimal r; // risk-free interest rate
decimal T; // time to maturity
decimal K; // strike price
int N; // number of iteration
decimal[] S; // array of share prices
```

It returns the expected price of an option:

```
decimal C=0; // price of the option
```

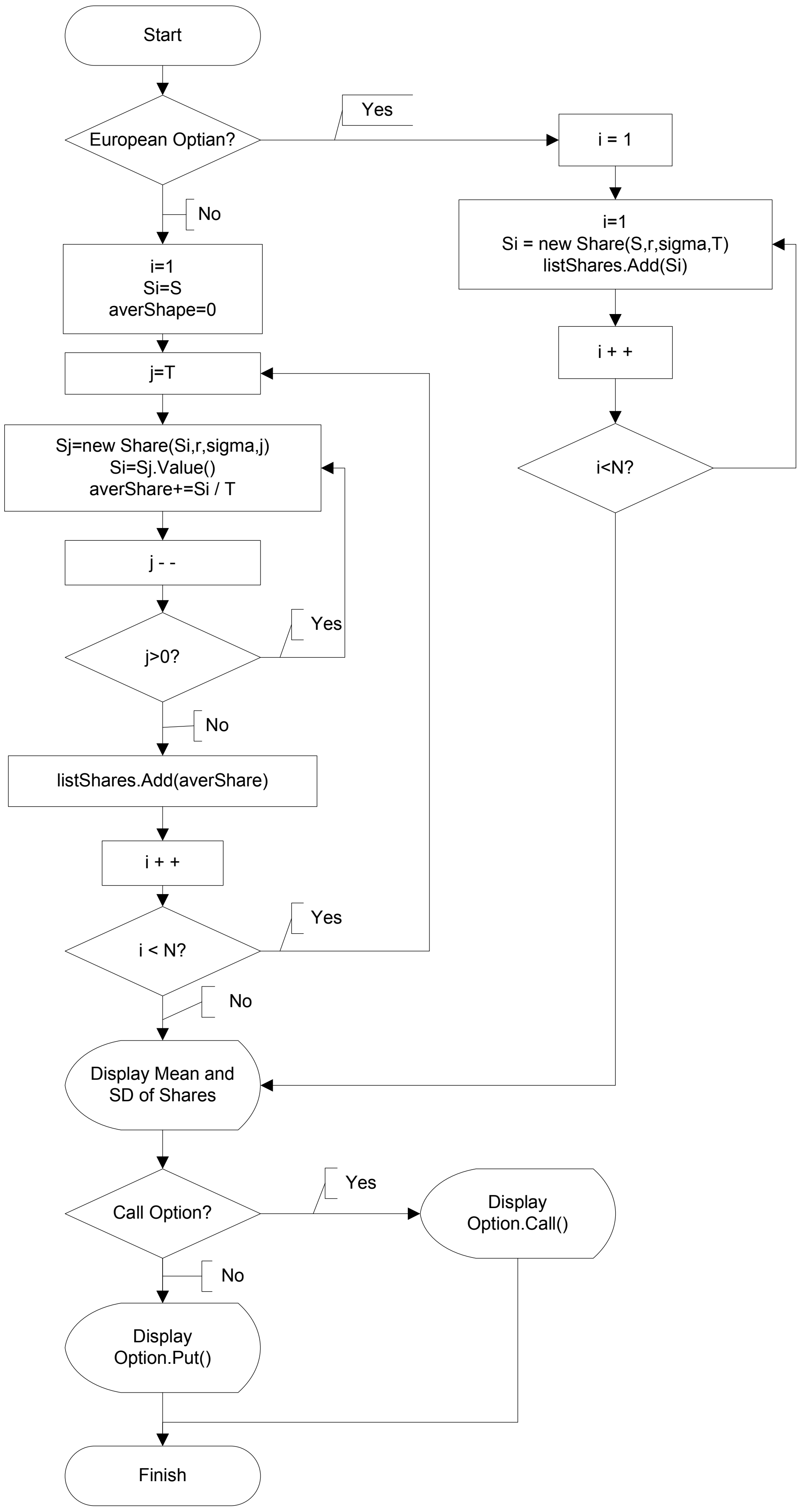
The creator of the class is the following:

```
/// <summary>
/// Creates European Option
/// </summary>
/// <param name="cr">risk-free interest rate</param>
/// <param name="cT">time to maturity (days)</param>
/// <param name="cK">strike price</param>
/// <param name="cN">number of iteration</param>
/// <param name="cS">simulated prices</param>
public Option(decimal cr,decimal cT,decimal cK,int cN,decimal[]cS)
{
    r = cr;
    T = cT/365;
    K = cK;
    N = cN;
    S = cS;
}
```

Functions ‘Call ()’ and ‘Put ()’ resemble each other and have the following code:

```
// Returns value of a Call option
public decimal Call()
{
    for (int i = 0; i < N; i++)
    {
       C += (decimal)(Math.Exp((double)(-r*T))*Math.Max((double)(S[i]-K),0));
    }
    C /= N;
    return C;
}
// Returns value of a Call option
public decimal Put()
{
    for (int i = 0; i < N; i++)
    {
       C += (decimal)(Math.Exp((double)(-r*T))*Math.Max((double)(K-S[i]), 0));
    }
    C /= N;
    return C;
}
```

Flowchart of the Monte Carlo Simulator



Source Code of Monte Carlo Simulator

The Main() is realized using three subroutins: SharesMeanSd(), SimulateEurope() and SimulateAsia(). The simplified code is the following:

```
private void Main()
{
    // …
    // int N = number of iterations
    // European or Asian option?
    if (this.rbEuropean.Checked == true)
    {
        SimulateEurope();
    }
    else
    {
        SimulateAsia();
    }
    decimal[]Shares=SharesMeanSd();
    Option Opt = new Option(r.Value, T.Value, K.Value, N,Shares);
    // Call or Put option?
    if (rbCall.Checked == true)
    {
        this.txtOptionPrice.Text = Math.Round(Opt.Call(), 4).ToString();
    }
    else
    {
        this.txtOptionPrice.Text = Math.Round(Opt.Put(),4).ToString();
    }
}
```

Next function calculates and shows Mean value and Standard Diviation of stock prices. The prices are taken from listShare control element and returned as decimal[] array.

```
private decimal[]SharesMeanSd()
{
    // int N = number of iterations
    decimal Mean = 0;//Mean of Shares' price
    decimal SD = 0; //SD of Shares' price
    decimal[] Shares = new decimal[N];//Array of prices
    for (int i = 0; i < N; i++)
    {
        Shares[i] = //take the next Share price from listShare
        Mean += Shares[i];
    }
    Mean /= N;
    for (int i = 0; i < N; i++)
    {
        SD += (Shares[i] - Mean) * (Shares[i] - Mean);
    }
    SD = (decimal)Math.Sqrt((double)SD / N);
    // display Mean and SD
    // …
    return Shares;
}
```

The following function is used in case of simulating prices for European rptions. It simulates N objects of 'Share' class and adds the shares value to listShare control element.

```
private void SimulateEurope()
{
    // int N = number of iterations
    for (int i = 1; i <= N; i++)
    {
        Share Si = new Share(S, r, sigma, T);
        this.listShare.Items.Add(S.Value());
    }
}
```

The next function is similar to the previous one. It is used for Asian options and adds to listShare element the average price of the shares for the whole requested period.

```
private void SimulateAsia()
{
    decimal averShare;
    decimal Si;
    for (int i = 1; i <= N; i++)
    {
        Si=S;
        averShare = 0;
        for (int j = T; j >0; j--)
        {
            Share Sj = new Share(Si, r, sigma, j);
            Si = Sj.Value();
            averShare += Si / T;
        }
        this.listShare.Items.Add(averShare);
    }
```

Appendix A

```
using System;
using System.Collections.Generic;
using System.Threading;

//this class is needed to simulate random variable~N(0,1)
  class norm
  {
      public static double rand()
      {
          Random autoRand = new Random();
          double z = GetOneGaussianByBoxMuller();
          if ((autoRand.Next(1000)) % 2 == 0)
          {
              z *= -1; // Define the sign of z
          }
          return z;
      }

      static double GetOneGaussianByBoxMuller()
      {
          double z;
          double x;
          double y;
          double sizeSquared;
          Random autoRand = new Random();
          do
          {
              Thread.Sleep(1);
              x = 2.0 * autoRand.NextDouble();
              y = 2.0 * autoRand.NextDouble();
              sizeSquared = x * x + y * y;
          } while (sizeSquared >= 1.0);

          z = x * Math.Sqrt(-2 * Math.Log(sizeSquared) / sizeSquared);
          return z; // z ~ N(0,1)
      }
  }
```