Instructor: Jingyi Zhu
Student: Yuliya Kovalenko

REU Fall 2005

Summary

"When it's a casino they call it gambling, when it's a stock market they call it investing", said my Probability professor S. Ethier.

The goal of my REU project was to study the basic stochastic model for financial asset pricing with experiments in the Monte Carlo simulations (to turn the art of gambling into the science of investment). That was a very fruitful project. I learned and put together concepts from different areas of studies, such as Calculus, Probability, Statistics, Economics and Programming. I examined and showed in my programs written in C how powerful the mathematical methods are for Economics.

I analyzed the stock market and predicted the call and put option prices using the Black-Scholes formula. There are mathematical tools and economical ideas behind the Black-Scholes analysis.

Mathematical tool is the basic stochastic model based on Brownian motion. The paths in the model of the stock price movements are based on the Brownian motion idea. Brownian motion is used in the mathematical models that describe the random movements. It is one of the simplest stochastic processes on a continuous domain, it serves as basis for many other stochastic processes. A stochastic model is one that involves uncertainties. Probability distributions of potential outcomes can be estimated if we allow for random variation in one or more inputs over time. First we need to select a period (standard time-series techniques) and then observe the fluctuations in historical data. We simulate the needed processes (in my case that was the change in stock prices during the 3 months) and derive the distributions of potential outcomes from those simulations (stochastic projections) which reflect the random variation in the input.

Economics idea that justifies the Black-Scholes formula is no-arbitrage principle. To understand the Black-Scholes methodology it is important to understand the idea of hedging. Hedging is a way of reducing some of the risk involved in investment. For instance, there is a chance that the stock that we own may fall, so we may consider to hedge our position by buying a put option for the stock that we own.

We have to consider the elimination of arbitrage opportunities while analyzing the stock market. Arbitrage is the process of buying assets in one market and selling them in another to profit from unjustifiable price differences. It is a profit without risk. In this case we know that the same product is not sold for the same price in different markets. The elimination of arbitrage opportunities leads to the fair price. This is how the price is justified (and that is what we have behind the Black-Scholes formula). On the other hand, a different price (higher or lower) will lead to arbitrage opportunities, therefore cannot sustain. The only price that can sustain is the one that would eliminate arbitrage opportunities. Arbitrage-free prices are seeing as a benchmark that can structure asset prices.

We must also consider the volatility when applying the Black-Scholes formula. Volatility measures the level of uncertainly. The option price as given by the Black-Scholes formula increases with volatility (the option can be a call or a put option). Therefore an option gives us a way to speculate on the volatility.

Black-Scholes formula for European call option is

$c=SN(d_1)-Xe-rTN(d_2)$,

where

$d_1=[\ln(S/X)+(r+\sigma^2/2)T]/[\ \sigma\sqrt{T}]$
$d_2=[\ln(S/X)+(r-\sigma^2/2)T]/[\ \sigma\sqrt{T}]$

N(x) is the cumulative probability function for a standardized normal variable. S is a stock price, X is a strike price, r is a risk-free interest rate, T is the time to expiration, and σ is the volatility of a stock price.

I implied the Black-Scholes formula for the actual Microsoft stocks. I got the data from finance.yahoo.com (I analyzed the stock prices for a year period), estimated the volatility and matched it with the volatility that I got using the Black-Scholes formula.

Volatility was estimated this way:

Let S_i be the stock price, r – risk-free interest rate, Δt=1 business day=1/253 year, T is a period (1 year=253 business days), σ - volatility.

$\Delta S_i/S_i=r\ \Delta t+\sigma\Delta w$
$\Sigma(\Delta Si/Si=r\ \Delta t+\sigma\Delta w$
$\Sigma\Delta w\approx 0$
$r=\ \Sigma(\Delta Si/Si=r\ \Delta t+\sigma\Delta w$
$\Sigma(\Delta Si/Si=r\ \Delta t+\sigma\Delta w$

In my C program I compared the results that I got for European call and put options using the Black-Scholes formula with the results of the Monte-Carlo simulation (MCs) given the same parameters (strike price, volatility, time periods etc) in both cases. I estimated the price of the call option (I tried different initial data to understand its input to the call option price). Monte-Carlo simulation is based on a pseudo random number generation (Box-Muller algorithm). The Box-Muller algorithm generates normal (Gaussian) distributions for uniform distributions. Uniform distribution is a distribution with equal probability over a finite range. So, we have two random numbers that are taken from the uniformly distributed range and then transform them to the numbers from the normal distribution (those numbers still remain random).

Transformations:

$$y_1 = \sqrt{-2\ln(x_1)}\cos(2\pi x_2)$$
$$y_2 = \sqrt{-2\ln(x_1)}\sin(2\pi x_2)$$

where the distribution of each **y** is described by:

$$p(y)dy = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{y^2}{2}\right) dy \qquad (4.5)$$

and x_1 and x_2 are uniform deviates.

In MCs I simulated the option payoffs to find the value of the stock and the option at every given time. After simulating the payoffs (MCs) I averaged them and compared the result with the one that I got using the Black-Scholes formula. The more iteration steps I used in my simulation the more precise formula I got.

I enclose the code of one of my programs that simulates the option payoffs as an example:

```
/* #include<conio.h> */
        #include<math.h>
        #include<stdlib.h>
        #define K 50
        #define M 0.05
        #define N 1000000
        #define R 0.02
        #define S 50
        #define SIGMA 0.15
        #define T 0.25
        #define X 50
        #define max(a,b) ((a) >= (b) ? (a) : (b))

        double nrand(int j)
        {
          double x1 = 0, x2 = 0, z1, z2, epsilon;

          while (!x1) x1 = (double)rand() / RAND_MAX;
          while (!x2) x2 = (double)rand() / RAND_MAX;
          z1 = sqrt(-2 * log(x1)) * cos(2 * M_PI * x2);
          z2 = sqrt(-2 * log(x1)) * sin(2 * M_PI * x2);
          epsilon = (j % 2) ? z1 : z2;

          return z1;
        }

        double n_of_arg(double arg)
        {
          double    k,
                    n_x, n_stroke;
          if (arg < 0) n_x = 1.0 - n_of_arg(-arg);
          else
```

```
        {
                k = 1 / (1 + 0.33267 * arg);
                n_stroke = exp (-arg * arg / 2) / sqrt(2 * M_PI);
                n_x = 1 - (0.4361836 * k - 0.1201676 * k * k + 0.937298 * k * k * k)
* n_stroke;
        }

        return n_x;
      }

      double mcs()
      {
        int i, j;
        double  x1, x2, z1, z2,
                s = 50, c = 0,
                t_delta,
                epsilon;

        for (i = 0; i < N; i++)
        {
                t_delta = 0.0025;

                s = 50;
                for (j = 0; j < 100; j++)
                {
                 /*     t_delta += 0.0025; */
                        epsilon = nrand(j);
                        s += s * (R * t_delta + SIGMA * epsilon * sqrt(t_delta));
                        if (!i && !(j % 10)) printf("j is %d  %lf\n", j, s);
                }
                c += max(s - K, 0) * exp(-R * T);
        }

        c /= N;

        return c;
      }

      double bsf()
      {
        double  d1, d2, c;

        d1 = (log(S / X) + (R + SIGMA * SIGMA / 2) * T) / (SIGMA * sqrt(T));
        d2 = (log(S / X) + (R - SIGMA * SIGMA / 2) * T) / (SIGMA * sqrt(T));

        c = S * n_of_arg(d1) - X * exp(-R * T) * n_of_arg(d2);
```

```
  return c;
}

int main ()
{
  double c_mcs, c_bsf;

  /*          clrscr();
    randomize();  */

  printf("Examples \n");
  c_mcs = mcs();
  c_bsf = bsf();
  printf("\nc_mcs = %f\n\nc_bsf = %f\n", c_mcs, c_bsf);


  /*          getch(); */
}
```