



# **Monte Carlo European Options Pricing Implementation Using Various Industry Library Solutions**

By Sergey A. Maidanov

---

Monte Carlo simulation is one of the recognized numerical tools for pricing derivative securities, particularly flexible and useful for complex models of real markets.

The goal of this article is to compare performance advantages and simplicity of using random number generators available in some industrial numerical libraries. For that purpose a simple and well-known Black-Scholes option pricing model, is used as a framework for illustrating the option pricing use.

The paper is intended for software developers interested in efficient implementations of Monte Carlo simulations.

---

## INTRODUCTION

The pioneering works of Black and Scholes [1] and Merton [2] resulted in the tremendous development of the theory of option pricing and has been extended to a wide range of other financial instruments. The fact of that Myron Scholes and Robert Merton were awarded the Nobel prize in economics in 1997 shows a great importance of the Theory of Options they developed. The original paper of Black and Scholes proposes a model and derives a closed form solution for European options on a single common stock. In spite of strong limitations, Black-Scholes model is a *de-facto* standard in financial world.

The limitations involved in the Black-Scholes model are based on the following assumptions:

- The market is assumed to be liquid, to be fair and provide all participants with equal access to the available information. This implies that there are no transaction costs.
- The market price is changed continuously according to one model, specifically the geometric Brownian motion process.
- Underlying security is perfectly divisible, and short selling with full use of proceeds is possible.
- The trading process is assumed as a continuous process.
- Constant risk-free rates are assumed.
- The principle of no arbitrage is assumed to be satisfied.

In practice, however, none of these principles can be perfectly satisfied. Nevertheless, as we mentioned above, Black-Scholes model is an important tool for real market analysis, and Black-Scholes prices provide good approximations to the prices of options.

We advisably consider option pricing and the Monte Carlo method in their simplest forms to focus on use of industry libraries vector type random number generators. Additional attention is paid to efficient implementation for the best performance, specifically on Intel® processor based platforms. The article also might be considered as a guiding line for integrating numerical libraries with random number generator capabilities into a broad range of financial instrument simulations.

Subsequent sections provide some information about the model of interest and a numerical solution for that. Since this numerical method is based on a random number

generation from a non-uniform distribution, the section “Random Number Generators in MKL, IMSL\* and NAG\* Libraries” provides a brief overview of random number generator capabilities of three industrial numerical libraries. The subsequent section demonstrates how to use these libraries in a C-language program implementing the algorithm discussed previously. Finally, the “Numerical and Performance Results” section provides numerical simulation performance results obtained on the Intel® Xeon™ processor system.

## BLACK-SCHOLES MODEL

A derivative, such as an option, is a financial instrument whose value depends on a value of underlying security. Regulated options trading began in 1973 at the Chicago Board Options Exchange. Nowadays derivatives become valuable and increasingly complex financial instruments. Derivative assets of interest in this article are European call and put options. European call options give the owner rights to buy underlying securities on a certain date of option expiration for a specified price. The expiration date is called the day until *maturity*. The specified price is called a *strike price*. We denote here a strike price by  $E$ . European put option gives the owner rights to sell underlying security on a certain date of option expiration for a specified price.

The efficient market hypothesis assumptions allow describing security price changes mathematically by the Markov processes. The Brownian motion is one of the forms of Markov processes, used as a model for stock price movements as far back as in 1900 by L. Bachelier. The Brownian motion model states that the value  $S$  of a security follows the stochastic process

$$dS = \mu S dt + \sigma S dW ,$$

where  $\mu$  is the *drift rate* and  $\sigma$  is the *volatility*. In our case we consider both parameters to be constants, although in long terms these values could not be considered constants. The  $W(t)$  variable is a *standard Brownian motion*, while  $dt$  is a *time increment*. At  $t = 0$  the value  $S(0)$  is  $S_0$ .

The Black-Scholes model states that the value  $V(t, S)$  of the derivative changes in time  $0 \leq t \leq T$  ( $T$  is a maturity) according to the following partial differential equation:

$$\frac{\partial V}{\partial t} + \frac{1}{2} \sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS \frac{\partial V}{\partial S} - rV = 0 ,$$

where  $r$  is the *risk-free interest rate* (the rate of changing of short-term risk-free securities — bank accounts or government bonds, such as a Treasury bill). For European call options, the boundary condition is at  $t = T$  and equals to  $V(T, S) = \max(S(T) - E, 0)$ . Value  $V(T, S)$  is called the *payoff*.

For European put options, the payoff is  $V(T, S) = \max(E - S(T), 0)$ .

The Feynmann-Kac formula gives us the method of option price valuation at specified time  $t$ . The Feynmann-Kac formula states that the option price should be the discounted expectation of the payoff under the assumption that  $S$  follows the Brownian motion process with the drift rate equal to the risk-free rate  $r$ , i.e.

$$dS = rSdt + \sigma SdW$$

For European call option

$$V(t, s) = e^{-r(T-t)} E(\max(S(T) - E, 0) | S(t) = s)$$

For European put option

$$V(t, s) = e^{-r(T-t)} E(\max(E - S(T), 0) | S(t) = s)$$

Since  $S(t)$  follows the Brownian motion model the values of call and put options can be rewritten as

$$V(t, s) = e^{-r(T-t)} E\left(\max\left(S_0 \exp\left((r - \sigma^2/2) + \sigma Z \sqrt{T-t}\right) - E, 0\right) | S(t) = s\right)$$

and

$$V(t, s) = e^{-r(T-t)} E\left(\max\left(E - S_0 \exp\left((r - \sigma^2/2) + \sigma Z \sqrt{T-t}\right), 0\right) | S(t) = s\right)$$

respectively, where  $Z$  is a random variate of the standard normal distribution. In other words,  $S(T)$  has a lognormal distribution.

The solution of the Black-Scholes equation can be found in a closed form. Unfortunately, only a few other cases can be solved analytically, where the solution for other derivative pricing models can be found by numerical simulation only. In the case of multiple securities Monte Carlo simulation is often a good choice of a numerical method.

In the following section we provide Monte Carlo algorithm to estimate the value  $V$  of the option for the Black-Scholes model. As mentioned, Black-Scholes equation can be solved analytically in this case, so we can compare how accurate is the Monte Carlo estimation.

## MONTE CARLO EVALUATION OF THE OPTION VALUES

The latter two equations from the previous section reduce a valuation of the Black-Scholes equation for call and put options to finding the expectation. For any random variate  $X$  with a distribution function  $F(x)$  the expected value of  $h(X)$  can be approximated by the sample mean:

$$E(h(X)) \cong \frac{1}{n} \sum_{i=1}^n h(x_i),$$

where  $x_1, x_2, \dots, x_n$  are random numbers from the  $F(x)$  distribution. The larger sample size  $n$  is the more accurate approximation is. As a numerical measure of accuracy we use the standard error.

For our case  $X$  corresponds to the  $S(T)$  variate, which has a lognormal distribution with a probability density

$$f(x) = \begin{cases} \frac{1}{(x-b)\tilde{\sigma}\sqrt{2\pi}} \exp\left(-\frac{[\ln((x-b)/\beta) - a]^2}{2\tilde{\sigma}^2}\right), & x > b \\ 0, & x \leq b \end{cases}$$

In our case the distribution parameters are

$$\begin{aligned}
 a &= r - \sigma^2/2 \\
 \tilde{\sigma} &= \sigma\sqrt{T-t} \\
 b &= -E \\
 \beta &= S_0
 \end{aligned}$$

The  $h(x)$  function is  $\max(x,0)$  for European call options and  $\max(-x,0)$  for European put options.

### Monte Carlo Algorithm for European Call Options Valuation

Taking an example, we evaluate European call options with a starting price  $S_0 = 100$ , a strike price  $E = 100$ , risk-free rate  $r = 0.1$ , volatility  $\sigma = 0.3$  and a maturity  $T = 1$ . The corresponding variable names we use in the algorithm are S, E, R, VOLATILITY and T. To approximate the option value we need to generate a huge number, `nsamples`, of random numbers. Libraries may provide two sorts of random number generation routine interfaces. The first is a scalar type random number generator interface, when a routine generates only one random number in a call. The second is a vector type random number generator interface, when a routine generates a vector of random numbers in a call. Due to architectural features of the modern computers vector type library routines often performs much more efficiently than scalar type routines. In other words the overhead expenses are often comparable with the total time required for generating a random number. Of course, users might call vector random number generator routine to generate just one random number, however, this is rather inefficient. That is why the random number generator subroutines in many industrial libraries have vector type interface even in spite of the fact that sometimes vector interface requires from user more careful programming. Nevertheless, a reward for more careful programming is a substantial speedup in the application performance. Below we consider only vector type random number generators, since our goal is the efficient implementation of the method using numerical libraries with random number generator capabilities.

The typical scheme of Monte Carlo simulation can be implemented in three steps:

1. **Initialization.** Initializing random number generators. Some numerical libraries provide several random number generators, so initialization step may also include selection of appropriate random number generator. Other initializations necessary to start simulation step.
2. **Simulation.** This step basically is the loop where pseudorandom (or quasi-random) numbers from appropriate distribution are used to estimate the quantity of interest. In addition, one may calculate variance estimator and/or standard error.
3. **Finalization.** Finalization step may include random number generator finalization (if required).

European option pricing scheme fits to the scheme discussed above. Steps 1–2 correspond to the initialization step. Steps 3–8 are the simulation where values of interest are calculated. Step 9 corresponds to the finalization step, where, if required, random number generator finalization is performed.

1. Initialize random number generator.
2. The number of blocks of BLOCK\_SIZE, the number of random numbers in the tail.  
`iNBlocks:=FLOOR(nsamples/BLOCK_SIZE);`  
`iNTail:=nsamples-iNBlocks*BLOCK_SIZE.`
3. Calculate the drift rate and the volatility.  
`dbDrift:=R-0.5*VOLATILITY*VOLATILITY;`  
`dbVolatilitySqrtOfT:=VOLATILITY*SquareRoot(T);`
4. Calculate the parameters of the lognormal distribution.  
`dbA:=dbDrift;`  
`dbSigma:=dbVolatilitySqrtOfT;`  
`dbB:=-E;`  
`dbBeta:=S0;`
5. Summation variables initialization. `dbSum:=0.0; dbSqrSum:=0.0;`
6. Main loop.
  - 6.1. Generate a vector of random numbers of the lognormal distribution with parameters dbA, dbSigma, dbB, dbBeta into dbBuf[j] array (j=1...BLOCK\_SIZE).
  - 6.2. For j=1...BLOCK\_SIZE do  
`dbPayoff:=MAX(dbBuf[j],0);`  
`dbSum:=dbSum+dbPayoff;`  
`dbSqrSum:=dbSqrSum+dbPayoff*dbPayoff.`  
 End do
7. Tail loop.
  - 7.1. Generate a vector of random numbers of the lognormal distribution with parameters dbA, dbSigma, dbB, dbBeta into dbBuf[j] array (j=1...iNTail).
  - 7.2. For j=1... iNTail do  
`dbPayoff:=MAX(dbBuf[j],0);`  
`dbSum:=dbSum+dbPayoff;`  
`dbSqrSum:=dbSqrSum+dbPayoff*dbPayoff.`  
 End do
8. Expected payoff, discounted payoff (option value at  $t=0$ ) and standard error.  
`dbExpectedPayoff:=dbSum/nsamples;`  
`dbDiscountedPayoff:=exp(-R*T)*dbExpectedPayoff;`  
`dbStdErr:=SquareRoot( (dbSqrSum-dbSum*dbSum/nsamples) /`  
`(nsamples-1)/nsamples ).`
9. Finalize random number generator.

Above algorithm requires further discussion. Instead of generating random numbers via a single call of vector random number generator we generate them by portions of the BLOCK\_SIZE size. Each generated random number represents  $S(T)$ . To calculate the expected payoff we need to calculate the sum of generated payoffs into dbSum variable. In addition, we calculate the sum of squares of generated payoffs into dbSqrSum

variable to calculate the standard error. After the expected payoff `dbExpectedPayoff` is calculated we need to discount its value to get the value of the option `dbDiscountedPayoff` at  $t = 0$ .

### Monte Carlo Algorithm for European Put Options Valuation

The put option is very similar to the call option, with the difference being that with the put option `dbPayoff` is evaluated as  
`dbPayoff := MAX( -dbBuf[j], 0.0 )`.

## RANDOM NUMBER GENERATORS IN MKL, IMSL\* AND NAG\* LIBRARIES

The use of random number generators from three different market available library products is outlined below.

### Intel® Math Kernel Library

Random number generation capabilities recently appeared in the Intel® MKL 6.0 as the Vector Statistical Library, or VSL. While version 6.0 contains only random number generators today, VSL assumes functionality extensions, which helps a user to achieve the best results (in terms of the library flexibility, reliability, accuracy and performance on the Intel® architectures) in various areas statistical methods are used, including financial functionality. All VSL random number generators are highly optimized for the latest features and capabilities of the Intel® Pentium® 4 processor, Intel Xeon processor and the Intel® Itanium® 2 processor.

There are 17 pseudorandom number generators for widely used probability distributions, such as uniform, Gaussian, exponential, Poisson, etc. Non-uniform distribution random numbers are generated via different transformation techniques applied to the source of random numbers of the uniform distribution called the *basic random number generator*, or BRNG. There are 5 basic generators implemented in MKL 6.0. Four of them are multiplicative generator MCG31m1, generalized feedback shift register generator R250, multiple recursive generator MRG32k3a and 59-bit multiplicative generator MCG59. The fifth is a Wichmann-Hill basic generator, which in turn is a set of 273 basic generators designed to produce statistically independent sequences. In addition, VSL provides an option of including user-designed basic generators and can use them in the same way as original VSL basic generators. This provides software developers with flexibility to meet their specific needs. For further discussion please refer to the VSL Notes document [6]. All VSL generators are of vector type. Discrete distribution generators return integer-valued array of generated random numbers. Continuous distribution generators return array of floating-point numbers. Both single precision and double precision implementations are for continuous distribution generators.

Instead of directly pointing to a particular basic generator to be used as the source of the uniformly distributed random numbers in transformation subroutines, the *random stream* descriptor is used. The random stream is the basic notion in VSL. It points to a particular random sequence generated by particular basic generator. The random streams mechanism allows generating several random sequences produced by one or more basic generators simultaneously as well as slitting the original sequence into several

subsequences by the *leapfrog* and/or *skip-ahead* methods. Since all these mechanisms are thread-safe, VSL random number generators can be efficiently used in parallel programming on the systems with shared and distributed memory.

For more details please refer to the Intel MKL documentation [6].

### **IMSL Libraries**

Visual Numerics provides a collection of mathematical and statistical analysis subroutines for Fortran and C/C++ users known as IMSL\* F90 Library and IMSL\* C Numerical Library respectively. There is also a Java library JMSL\*, which combines numerical analysis functions with visualization capabilities.

IMSL Libraries are targeted on a wide range of machine platforms including Intel® architecture based platforms.

IMSL F90 MP Library 4.0 has a random number generation capabilities for 23 univariate distributions, 4 multivariate distributions as well as for random orthogonal and correlation matrices, two-way tables, order statistics from a normal and uniform distributions, random samples and permutations, and stochastic processes simulation. The IMSL Fortran Library 5.0 introduces Faure low-discrepancy sequences generation capabilities.

All these subroutines are based on internal basic uniform generators. The user can select only one of 7 basic generators: either GFSR generator, or three multiplicative congruential generators, or the same multiplicative generators with shuffling.

One subroutine is used to initialize basic generators. However, to get a state of a basic generator, the user should call appropriate subroutines, which are different for GFSR, multiplicative and shuffled basic generators. In addition, the tables representing basic generator state for the shuffled generators are different for single and double precision. So, if precisions are mixed in a program, it is necessary to manage each precision separately for the shuffled generators.

The user has an ability to replace internal basic generators with a customized uniform basic generator. However, many service subroutines cannot be used with a customized basic generator.

IMSL Libraries provide some options to control more than one random sequence simultaneously. For multiplicative generators without shuffling there is subroutine to skip 100000 random numbers in a sequence.

Some generation routines have a scalar form and a vector form implementations, while other are of a vector type only or of a scalar type only. Both single and double precision floating-point implementations are for continuous distributions.

For further details, please refer to the Visual Numerics web site [7].

## **NAG Fortran Libraries**

The NAG numerical libraries, available from The Numerical Algorithms Group [5] are targeted for different platforms including Intel® architecture based platforms.

The Chapter G05 of the NAG\* Fortran 77 Library (Mark 20) Manual is concerned with pseudo- and quasi-random number generators from various distributions as well as with pseudo-random time series from some time-series models, random matrices and random samples and permutations.

NAG Fortran 77 Library provides generators for 28 distributions as well as subroutines for time series, permutations, random matrices generation and sampling. All random number generation subroutines call basic generator: either multiplicative congruential (NAG calls “original”) or Wichmann-Hill. There are two distinct sets of generation routines representing two distinct methods for communicating the data representing the current state of a given generator. In the first set the data is stored and passed internally, while in the second set the data is held in parameters that are passed through the routine interfaces. NAG recommends the second set be used.

Routines with an internal communication use some thread-unsafe constructs and cannot be safely used in multithreaded applications. Besides routines to select and initialize basic generators, there are subroutines to save and restore the state of the selected basic generator. This is the only way to produce two or more sequences at the time application runs. NAG does not recommend saving and restoring states between machines.

Routines communicating through the interface are recommended for random number generation as well as for time series generation. These routines pass information relating to the generator and its current state through their interfaces, and don't use thread-unsafe constructs. The state of the generator can be saved and restored manually by copying from a variable representing generator state to another one.

Some generation subroutines are of scalar type, other produce vector of  $n$  successive random numbers.

NAG\* Fortran 90 Library provides 12 distribution generators based on the multiplicative congruential basic generator (MKL notation is MCG59). NAG Fortran 90 library uses mechanism to pass generator state through a routine interface only.

There are two special versions of NAG libraries known as NAG\* SMP Library and NAG\* Parallel Library designed for SMP and distributed memory systems respectively. These libraries incorporate a mechanism for generating independent sequences of random numbers in parallel. The SMP Library functionality is similar to the Fortran 77 Library. The Parallel Library functionality, however, is substantially restricted to the uniform, discrete uniform, normal and exponential distribution generators based on Wichmann-Hill basic generator. Wichmann-Hill generator allows producing up to 273 independent

random sequences simultaneously. All NAG numerical libraries do not support sequence splitting techniques.

Depending on the library name and platform, NAG numerical libraries have either double precision or single precision floating-point implementations. For example, NAG SMP Library and NAG Parallel Library support double precision floating-point arithmetic only.

For further details please refer to the NAG web site [8].

**Does User Need Extended Random Number Generation Functionality in Derivative Pricing Models?**

Many financial derivative models and Monte Carlo solutions are of interest in financial engineering during past decade. They are dealing with path-dependent instruments (for example, Asian options), interest rate dependent instruments (bond options, mortgage-backed securities), and models considering the stochastic nature of interest rates as well as the stochastic volatility. This list is not exhaustive, for more details see [4], for example, and references therein. Many models operate with multiple securities rather than with a single security we considered in our example.

Such diversity of derivatives (and models dealing with them) generates diverse Monte Carlo solutions. As a result, random numbers from various continuous and discrete distributions, univariate and multivariate are at the kernel of those Monte Carlo solutions.

Variance reduction is an important constituent of the in Monte Carlo method. To reduce variance Monte Carlo methods for derivatives pricing often deal with quasi-random numbers rather than with pseudorandom. Other variance reduction techniques are of great importance as well (see [3] and [5] as an example). Parallel Monte Carlo solutions are provided in the literature on derivatives pricing. All this requires from the libraries with random number generator capabilities the availability of flexible mechanisms for simultaneous operating with random number sequences, such as the leapfrog and skip-ahead methods.

The table below presents summary information on random number generation capabilities existing in Intel MKL 6.0, IMSL F90 MP 4.0 and NAG Fortran 77 (Mark 20) libraries.

Feature	Intel MKL	IMSL	NAG
Sequence selection	By initializing a random stream: <ul style="list-style-type: none"> <li>• Creating new stream by passing a basic generator and an initial value or an array of initial values;</li> <li>• Creating the copy of an original stream;</li> <li>• Copying the stream state from an original stream to an existing one</li> </ul>	<ul style="list-style-type: none"> <li>• By selecting a basic generator and setting an initial value;</li> <li>• By restoring previously saved seed and tables</li> </ul>	<ul style="list-style-type: none"> <li>• By selecting a basic generator and setting an initial value;</li> <li>• By restoring previously saved seed(s)</li> </ul>

User-designed BRNGs	The same functionality as in predefined basic generators. Several user-designed BRNGs registration is possible	Predefined library BRNGs become unavailable. Only one user-designed BRNG registration.	Not supported
Splitting into sub-sequences	Both leapfrog and skip-ahead methods. Leapfrog method allow splitting a sequence into arbitrary number of non-overlapping subsequences. Skip-ahead method allows skipping arbitrary number of elements in a sequence.	Skipping 100000 elements in a sequence	Not supported
Multiple transformation methods for non-uniform random number generators	Available. Selection is by passing method ID as parameter	Available. Different subroutines are for different transformation methods	No
Vector/Scalar implementations	All random number generation subroutines are of vector type	Either vector or scalar type implementations available. Some generators have both vector and scalar type implementations	Either vector or scalar type implementations available. Some generators have both vector and scalar type implementations
Multivariate distribution generators	No. Possible extensions in future MKL releases.	Yes	Yes
Low-discrepancy sequences	No. Possible extensions in future MKL releases.	No in IMSL F90 MP 4.0. Faure QRNG support starting from 5.0 version	Yes
C/Fortran Interface	Both C and Fortran style subroutine interface	Fortran style subroutine interface only	Fortran style subroutine interface only

## Using Intel MKL Random Number Generators

In this section we provide a C example on the European call option valuation discussed previously using Intel® MKL 6.0 random number generators. The code for European put option is similar.

The `MCEuropeanCallOption` function estimates and returns the value of a European call option with initial conditions discussed earlier. The standard error value is returned via `sterr` variable.

```
double MCEuropeanCallOption( int nsamples, double* sterr )
{
    VSLStreamStatePtr  stream;
    static double dbBuf[BLOCK_SIZE];
    double dbDrift, dbVolatilitySqrtOfT;
    double dbA, dbSigma, dbB, dbBeta;
    double dbPayoff;
    double dbSum, dbSqrSum;
    double dbExpectedPayoff;
    double dbDiscountedPayoff;
    double dbStdErr;
    int iNBlocks, iNTail;
    int i,j;

    iNBlocks = nsamples / BLOCK_SIZE;
    iNTail = nsamples - iNBlocks*BLOCK_SIZE;
```

To generate random numbers from a given distribution, you should initialize a random stream. Below code creates new stream `stream` generated by the basic generator `VSL_BRNG_MCG31` with an initial value `SEED`.

```
/* *****
   Step 1. Random number generator initialization
   ***** */
vslNewStream( &stream, VSL_BRNG_MCG31, SEED );
To start a simulation you should initialize necessary variables:
/* *****
   Step 2. Simulation
   ***** */
dbDrift = R - 0.5*VOLATILITY*VOLATILITY;
dbVolatilitySqrtOfT = VOLATILITY*sqrt(T);

dbA = dbDrift;
dbSigma = dbVolatilitySqrtOfT;
dbB = -E;
dbBeta = S0;

dbSum = 0.0;
dbSqrSum = 0.0;
```

The main loop generates vectors of payoffs and sums up them and their squares. Double precision lognormal distribution random number generator subroutine `vdRngLognormal` is used to fill the buffer `dbBuf`. The first parameter is the transformation method (Inverse Cumulative Distribution Function method in this case).

VSL allows more than one transformation method for a given distribution generator. See VSL Notes document for further information [6]. The second parameter in `vdRngLognormal` is the random stream. Other parameters are the number of random numbers to generate in a call, buffer for generated random numbers and four parameters of the lognormal distribution.

```

/* Main loop */
for ( i = 0; i < iNBlocks; i++ )
{
    vdRngLognormal( VSL_METHOD_DLOGNORMAL_ICDF, stream,
        BLOCK_SIZE, dbBuf, dbA, dbSigma, dbB, dbBeta );
    for ( j = 0; j < BLOCK_SIZE; j++ )
    {
        dbPayoff = MAX( dbBuf[j], 0.0 );
        dbSum += dbPayoff;
        dbSqrSum += dbPayoff*dbPayoff;
    }
}

```

The same operations are done in the tail loop.

```

/* Tail loop */
{
    vdRngLognormal( VSL_METHOD_DLOGNORMAL_ICDF, stream,
        iTail, dbBuf, dbA, dbSigma, dbB, dbBeta );
    for ( j = 0; j < iTail; j++ )
    {
        dbPayoff = MAX( dbBuf[j], 0.0 );
        dbSum += dbPayoff;
        dbSqrSum += dbPayoff*dbPayoff;
    }
}

```

The following code calculates the payoffs and the standard error.

```

dbExpectedPayoff = dbSum / (double)nsamples;
dbDiscountedPayoff = exp(-R*T)*dbExpectedPayoff;
dbStdErr = sqrt( (dbSqrSum-dbSum*dbSum/(double)nsamples)/
    (double)(nsamples-1)/(double)(nsamples) );

```

Finally we should delete created stream by the `vslDeleteStream` function and return estimated values.

```

/*****
    Step 3. Deleting the stream
    *****/
vslDeleteStream( &stream );

*sterr = dbStdErr;
return dbDiscountedPayoff;
}

```

## Using IMSL Random Number Generators

Using the IMSL F90 Library, parameters are passed into subroutines in Fortran style (by reference), so we need to define additional local variables for such parameters:

```
/* IMSL specific variables */
int iOpt; // basic RNG used
int iSeed; // initial value for basic RNG
int iNr; // number of random numbers to generate
```

At the initialization step we call RNOPT subroutine to select the basic generator and RNSET subroutine to pass initial value.

```
/*
*****
Step 1. Random number generator initialization
*****
iOpt = 1; // Minimal Standard basic generator is used
RNOPT( &iOpt );
iSeed = SEED; // initial value for basic generator
RNSET( &iSeed );
*/
```

To generate random numbers from the lognormal distribution the DRNLNL vector generator is used. The iNr variable holds the number of random numbers to generate. Only two parameters of the lognormal distribution should be passed for the subroutine in question:

```
DRNLNL( &iNr, &dbA, &dbSigma, dbBuf );
```

So, the payoff calculation is slightly different:

```
dbPayoff = MAX( dbBeta*dbBuf[j]+dbB, 0.0 );
```

There is no need to finalize IMSL random number generators as we did in MKL by deleting the random stream.

## Using NAG Random Number Generators

Additional local variables are needed for the example using the NAG Fortran Library.

```
/* NAG variables */
int iGen; // basic RNG used
int iSeed[4]; // initial values for basic RNG
int iFail; // error status variable
int iNr; // number of random numbers to generate in a call
double dbVar; // instead of dbSigma parameter dbVar=dbSigma*dbSigma should
// be passed into the lognormal distribution RNG
```

At the initialization step we use G05KBF subroutine to select and initialize a basic generator:

```
/*
*****
Step 1. Random number generator initialization
*****
iGen = 0; // MCG59 basic generator is used
iSeed[0] = SEED;
iSeed[1] = 0;
```

```

iSeed[2] = 0;
iSeed[3] = 0;
iFail = 0;
G05LKF( &iGen, iSeed );

```

The G05LKF subroutine is used to generate a vector of random numbers from the lognormal distribution:

```

G05LKF( &dbA, &dbVar, &iNr, dbBuf, &iGen, iSeed, &iFail );

```

Instead of dbSigma parameter (the standard deviation of the underlying normal distribution), the dbVar should be passed (the variance of the underlying normal distribution). Besides the number iNr of random numbers to generate, and the buffer dbBuf, you should pass basic generator ID iGen and the array of initial values iSeed initialized at the initialization step. The error status iFail is returned on exit.

## NUMERICAL AND PERFORMANCE RESULTS

In this section we present numerical and performance results obtained with the program we discussed in the previous section. Similar experiments were performed using the Intel MKL as well as with IMSL F90 MP 4.0 and NAG Fortran 77 Library (Mark 20) as well.

Library	Basic Generator	Option Value (Exact Value)		Absolute Error (Standard Error)		Time	Speedup <sup>1</sup>
		Call	Put	Call	Put		
MKL	MCG31	16.7291 (16.7341)	7.2184 (7.2179)	0.0050 (0.0029)	0.0005 (0.0014)	7.246 sec	8.46 times
	MCG59	16.7366 (16.7341)	7.2164 (7.2179)	0.0025 (0.0029)	0.0015 (0.0014)	7.625 sec	8.04 times
IMSL	Minimal Standard	16.7330 (16.7341)	7.2178 (7.2179)	0.0011 (0.0029)	0.0001 (0.0014)	29.630 sec	2.07 times
NAG	Original	16.7394 (16.7341)	7.2162 (7.2179)	0.0053 (0.0029)	0.0017 (0.0014)	61.306 sec	1.00 times

Results were obtained using an Intel Xeon 2.2GHz processor based platform, (512 MB RAM, 512K L2 cache size) and the Intel® C/C++ Compiler 7.1 (/O2 and /QxW options). Please note that the MKL 6.0 random number generators were aggressively tuned to achieve the best performance on Intel® processors.

To generate random numbers from the lognormal distribution we used following library routines: vdRngLognormal (Box-Muller-2 method) in MKL 6.0, DRNLNL in the IMSL and G05LKF subroutine in the NAG Fortran 77 library.

We used the “minimal standard” basic generator as the fastest one in IMSL library. For the same reason we used the “original” basic generator in the NAG library. On the MKL 6.0 side we used two basic generators,

<sup>1</sup> Speedup is measured against the slowest application.

namely MCG31 and MCG59. The latter is identical with the “original” basic generator in NAG Fortran Libraries, while properties of MCG31 are similar to the “minimal standard” basic generator from IMSL. We should notice again that IMSL F90 MP 4.0 has 7 basic generators; NAG Fortran 77 library has 2 basic generators (remembering that Wichmann-Hill is the set of 273 basic generators) and MKL 6.0 has 5 basic generators (again, Wichmann-Hill basic generator, implemented in MKL\VSL is the set of 273 basic generators).

## CONCLUSIONS

The  $1/\sqrt{N}$  convergence of the crude Monte Carlo method we considered for European option pricing model requires extensive use of the computing resources. There are a lot of general variance reduction techniques designed to improve the speed of the convergence, i.e. antithetic variates, low discrepancy sequences, as well as importance and stratified sampling. Our results demonstrate on the simple model of the option pricing that among these variance reduction techniques it is also important to use efficient random number generation algorithms.

Strengths of each library are outlined, specifically IMSL F90 MP 4.0 and NAG Fortran 77 libraries have wider random number generator functionality than Intel MKL 6.0. On the other hand, from the discussion earlier, we might conclude that Intel Math Kernel Library provides the advanced thread-safe and flexible mechanism to operate with random number streams. This mechanism allows making use of random number generators more natural and easier in various programming environments including environments for parallel programming.

Intel MKL 6.0 random number generators are highly optimized for the Intel Pentium 4 processor, Intel Xeon processor and the Intel Itanium 2 processor. Results from the Section 6 shows that using Intel Math Kernel Library random number generators our option pricing simulation runs at least 4 times faster on the Intel Xeon processor than using other widely used libraries with random number generator capabilities.

## REFERENCES

- [1] Fischer Black and Myron S. Scholes. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, 1973, vol. 81, issue 3, pages 637–654
- [2] Robert Merton. Theory of Rational Option Pricing. *Bell Journal of Economics and Management Science*, 4, Spring 1973, pages 141–183
- [3] Marc Potters, Jean-Philippe Bouchaud, Dragan Sestovic. Hedged Monte-Carlo: low variance derivative pricing with objective probabilities. *Physica A* 289, 2001, pages 517–525
- [4] Jerome Barraquand. Numerical Valuation of High Dimensional Multivariate European Securities. Digital PRL, Research Report No. 26, March 1993, <ftp://ftp.digital.com/pub/DEC/PRL/research-reports/PRL-RR-26.pdf>\*
- [5] Jean-Pierre Fouque, Tracey Andrew Tullie. Variance Reduction for Monte Carlo Simulation in a Stochastic Volatility Environment. *Quantitative Finance* 2, February 2002, pages 24–30

[6] <http://www.intel.com/software/products/mkl/docs/manuals.htm>

[7] <http://www.vni.com/>\*

[8] <http://www.nag.co.uk/>\*

## ABOUT THE AUTHOR

**Sergey A. Maidanov** is a Senior Software Engineer for Intel® Corporation. He is a lead designer of the Vector Statistical Library portion of the Intel® Math Kernel Library Product. In 1997 Sergey received a B.S. and in 1999 M.S. from the computer science department at Nizhny Novgorod State University (NNSU) in Russia. In 2000 he joined a research staff at International Centre of Studies in Financial Institutions at NNSU. His research interests are in random number generation, numerical analysis and operations research.

---

® Intel, Intel Xeon, Itanium, Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.