Daniel Duffy and Joerg Kienitz

Monte Carlo Methods in Quantitative Finance Generic and Efficient MC Solver in C++

We describe how we have designed and implemented a software architecture in C++ to model one-factor and multi-factor option pricing problems. We pay attention to the fact that different kinds of applications have their own specific accuracy, performance and functional requirements. To this end, we apply the design patterns that we have discussed in previous editions of *Wilmott*. In this way we ensure that the software can be customized to suit new and changing requirements.

The software has been written in C++ and it makes extensive use of GOF design patterns, Standard Template Library (STL) and template classes. We apply the software to the pricing of three well known benchmark examples, namely a plain vanilla call, an arithmetic Asian option and an up-and-out call option.

Goals

The Monte Carlo method is a popular method that is widely used to price a range of derivative products (see Boyle 1977). There are a number of good references on the method, for example Glassermann 2004 and Jaeckel 2002. For a discussion of numerical solutions to stochastic differential equations see Kloeden 1997). In this article we describe how we have applied the Monte Carlo method to produce a software system that is able to price a range of one-factor and multi-factor options. In order to reduce the scope of this article, we examine only one-factor models.

It is worth mentioning at the outset what the goals of this article are. In general, we are interested in producing a software product that is robust, flexible and that performs well at run-time. In particular, we quantify these general requirements by listing a number of features that the software should have:

Suitability: the MC solver is able to model a wide range of one-factor and multi-factor derivative types. In fact, it would be nice if the solver knew as little as possible of the derivative products that it is modeling because it allows users to 'plug' their models into the solver without having to write new code.

Accuracy: it is well-known that the MC method gives us a convergent solution in general, albeit slowly. The solver that we write in C++ must reflect this accuracy.

Performance: the response time should be as good as possible. To this end, we could consider efficient random number generators, clever C++ data structures and use of Halton and Sobol sequences, for example. Furthermore, we would expect the performance to be a hundred times better than a corresponding application written in VBA.

Having defined our objectives, we now turn our attention to describing how we realized them using modern software design techniques.

High Level Architecture

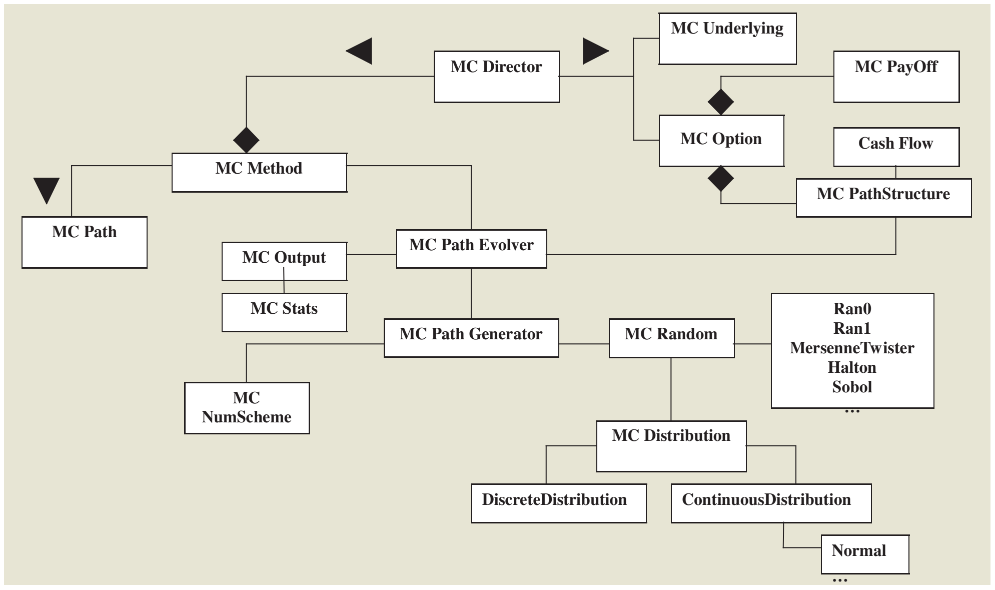
The first author has produced a generic software architecture and it has been applied to option pricing models using both the binomial and finite difference methods as discussed in previous editions of *Wilmott*. The architectures are based on the generic design principles in Duffy 2004. In this article we adopt the same approach by partitioning the problem into a network of loosely coupled subsystems and classes.

In general, we use a number of small-grained patterns and we connect them to form a larger configuration as shown in the figure below. In particular, we have implemented the following patterns:

Whole/Part and Aggregation Structures: in general, we partition large, complex objects ('whole') into smaller and specialized objects ('parts'). This approach promotes reusability and maintainability of the software. An example from the figure is the class representing an option. This has two main parts, namely a payoff and a cash flow structure.

Mediator: objects that act as intermediaries between a number of other objects. These communicating objects have no direct knowledge of each other. Instead, they must interface with the mediator which is this case plays the role of a façade. An example in the figure is the mediator class that relates the option class with the main Monte Carlo class MCMethod.

Delegation Mechanism: Classes operate on a client-server basis: the client class calls a member function in a server class. Server classes are specialized to carry out certain tasks such as random number generation, calculation of the mean and variance of statistical distribution functions. Examples can be seen in the figure where we see classes for random number generation and statistical distributions.



Implementing the Design in C++

Having designed the problem as shown in the figure we now must implement the classes in C++. To this end, we give some representative examples to give the reader a flavour of how we actually constructed the software.

The class that stores the values of the evolved asset is the Monte Carlo path class. This is a simple implementation of a C++ array class. The interface is given by:

```
class MCPath
{
private:
        double* ValuesPtr;
        double* EndPtr;

        unsigned long Size;
        unsigned long Capacity;

public:
        //constructor
        explicit MCPath(unsigned long size=0);
        MCPath(const MCPath& original);

        //destructor
        virtual ~MCPath();

        //operator overloading
        MCPath& operator=(const MCPath& original);
        MCPath& operator=(const double& val);
```

```
    inline double operator[](unsigned long i) const;
    inline double& operator[](unsigned long i);

    inline unsigned long size() const;
    void resize(unsigned long newSize);

    double last() const;
    double nth(unsigned long nthelement) const;
        ...
};
```

The classes for discrete and continuous probability distributions have been taken from Duffy 2004A. They have functionality that is needed in the current application. The classes for continuous distributions are placed in a class hierarchy whose base class is given by:

```
            template <class Domain, class Range> class
           ContinuousDistribution
                : public ProbabilityDistribution<Domain, Range>
                { // Abstract base class for continuous
                probability distributions

private:

public:
        // Constructors
        ContinuousDistribution();
        ContinuousDistribution(const
        ContinuousDistribution<Domain, Range>& d2);

        Virtual ~ContinuousDistribution();


        // Selector member functions
        virtual Range pdf(const Domain& x) const = 0;
        //density
        virtual Range cdf(const Domain& x) const = 0;
        //cumulative density
        virtual Range invcdf(const Domain& x) const = 0;
        //inverse cumulative density

        //Selectors
        virtual Range expected() const = 0;
        virtual Range variance() const = 0;
        virtual Range std() const {return :: sqrt(variance());}

        virtual ContinuousDistribution<Domain, Range>*
clone() const = 0;
};
```

Finally, we derive all random number class from the one base class as follows:

```
class Random
{

private:
        unsigned long Dimensionality;
        ContinuousDistribution<double, double>* cndist;
//Strategy

public:
        // Constructor and default Constructor
        Random(unsigned long Dimensionality,
        ContinuousDistribution<double, double>& cndist);
        Random(){};

        // Destructor
        virtual ~Random();

        //Copy constructor
        virtual Random* clone() const=0;

        // virtual functions
        virtual GenerateUniforms(MCPath& variates)=0;
        virtual Skip(unsigned long NumberOfPaths)=0;
        virtual void SetSeed(unsigned long Seed)=0;

        virtual void GenerateDueToDistribution(MCPath& variates,
        ContinuousDistribution<double,double>& cndist);
        virtual void Reset()=0;
        virtual void ResetDimensionality(unsigned long
        NewDimensionality); //for given dim
        virtual void
        SetDistribution(ContinuousDistribution<double,
        double>& newcndist);
         //for given distribution

        inline unsigned long GetDimensionality() const;
   };
```

The above design enables us to use pseudo random numbers as well as quasi random numbers because the base class implements all common functionality pertaining to random number generators and quasi random number sequence generators.

The heart of a Monte Carlo simulation is the path generator. The variable is evolved due to a numerical recipe taking into account the given stochastic dynamic and the current state X_t=x at time t starting at time t=0. The outcome of this procedure is the variable X_{t+1} at time t+1. To this end we actually have to use a variate sampled from the probability distribution $P(X_{t+1} | X_t)$. For more details, Kienitz 2005.

Summarising, we have implemented the classes from the figure. We have linked them into a library that can be used in actual computations.

Test Cases and Benchmarking

To illustrate the applicability of our design we study some examples, namely a simple plain vanilla Up and Out call and an arithmetic Asian call.

A European plain vanilla call has payoff $\max(S_T - K,0)$ at maturity T. We give the convergence table calculated via our setup with parameters Spot = €100, Strike = €100, Maturity = 1 year, Volatility = 25%, riskless rate r = 3%. The generator in this case uses the Sobol low discrepancy numbers. We did 32 batches with 8192 samples.

Number of Paths	Value	St. Error	Rel. Difference To BS
262144	11.3480	0.04	0.00%
253952	11.3483	0.04	0.00%
		...	
16384	11.3433	0.14	0.05%
8192	11.3400	0.20	0.07%

The first exotic option we examine ia a standard Up and Out option. This option has the same payoff as a plain vanilla call but only if the assets price stays below a barrier level at certain discretely chosen times up to maturity. We consider the parameters Spot = €150, Strike = €136, Maturity = 4.5 year, Volatility = 15%, riskless rate r = 2% and Barrier = €164 with monthly monitoring. To evolve the assets path we use the congruential generator Ran1 again with 32 batches of 8192 samples. The results are:

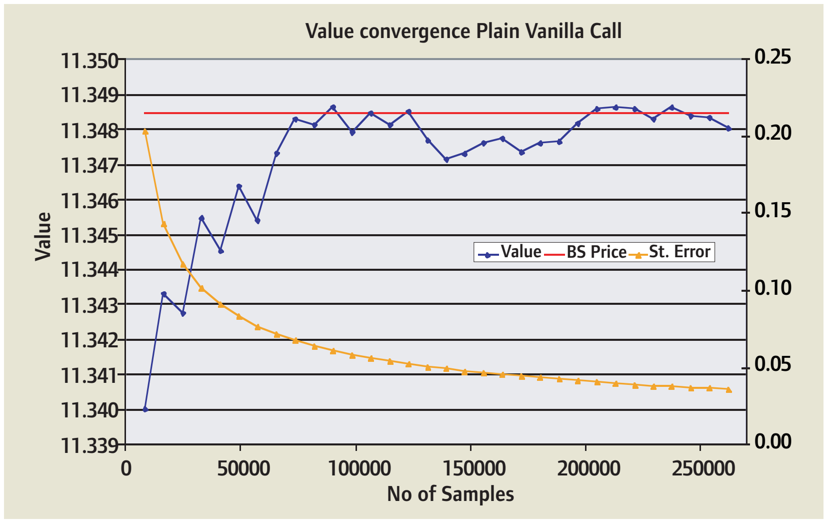
Number of Paths	Value	St. Error
262144	0.5042	0.0035
253952	0.5047	0.0036
16384	0.5002	0.0142
8192	0.4954	0.0202

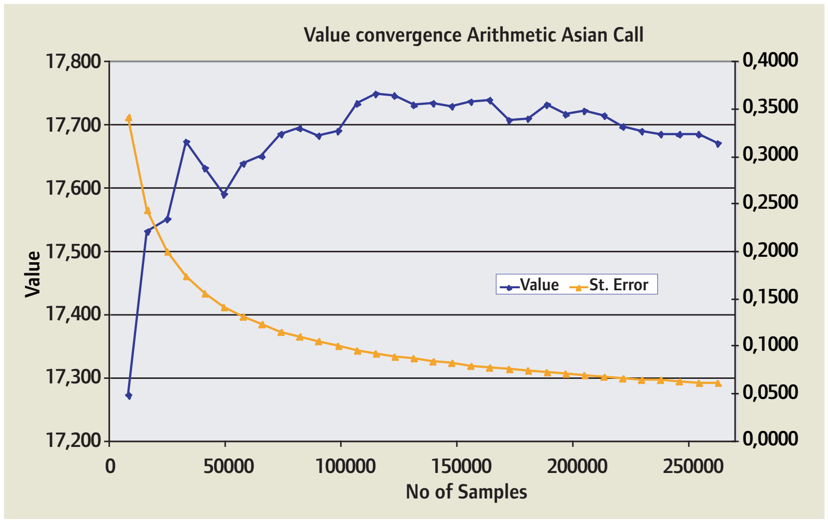
To get an idea of the accuracy we compare our computed price with the approximations developed in (Merton 1973) and further refined in (Reiner, Rubinstein 1991). For the above option we compute a price of 0.5221.

The last example is the case of an Arithmetic Asian call. In contrast to a geometric Asian option a closed form pricing formula is not available. The payoff is given by

$$\max\left(\frac{1}{N}\sum_{i=1}^{N} S_{t_i} - K, 0\right).$$

The parameters are chosen to be Spot = €100, Strike = €95, Maturity = 2 year, Volatility = 42.5%, riskless rate r = 3% and N=12. The random number generator used is the Marlene Twister, see (Matsumoto 1998) again with 32





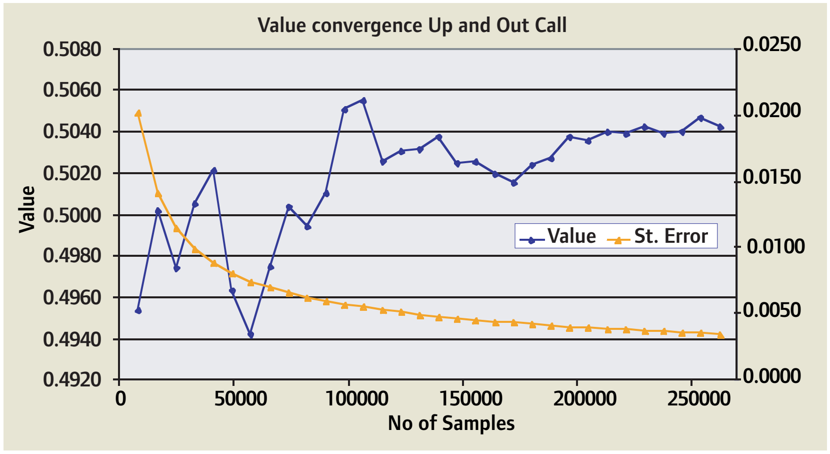
batches of 8192 samples.

Number of Paths	Value	St. Error
262144	17.6714	0.0611
253952	17.6847	0.0621
...		
16384	17.5318	0.2432
8192	17.2734	0.3415

Again, since analytical formulas are not available we compare our prices with well known approximations for arithmetic Asian options, namely with the methods developed in (Kemna 1990), (Turnbull 1991) and (Levy 1992). For the above option we computed an approximate value of 17.27 which underestimates the price of the option. The reason for this discrepancy is that the approximate formula assumes continuous observations while our MC method assumes that we observe the price every month. In general, the price of an arithmetic Asian option is a decreasing function of the number of observations.

Reflection : what have we achieved?

We have implemented a flexible and extendable Monte Carlo engine. There



is a significant speed-up when using the C++ implementation. The generation of certain quasi-random numbers such as Halton or Sobol numbers is a hundred times faster than using the same algorithm in a VBA setting.

The comparison with analytic formulas and with approximate formulae proves the accuracy of the method. Furthermore, we would like to stress that for the one dimensional case the results are stable with respect to the random number generator that we used.

We thank Mr. Andrea Germani (Banca Popolare Italiana) for important comments regarding the contents of this article.

REFERENCES

■Acklam, P.J. (2000), *An algorithm for computing the inverse normal cumulative distribution function*, University of Oslo, Statistics Division. http://www.math.uio.no/jacklam/notes/invnorm.
■Boyle, P.P. (1977) Options: a Monte Carlo approach, *Journal of Financial Economics* 4:323-338
■Duffy, D. J.(2004), *Domain Architectures: Models and Architectures for UML Application*, John Wiley and Sons
■Duffy, D. J. (2004A), *Financial Instrument Pricing using C++*, John Wiley and Sons
■Glassermann, P. (2004), *Monte Carlo Methods in Financial Engineering*, Springer
■Joshi, M. (2004), *C++ Design Patterns and Derivatives Pricing*, Cambridge University Press
■Jaeckel, P. (2002), *Monte Carlo Methods in Finance*, John Wiley and Sons
■Kemna, A. G. Z., Vorst, A. C. F. (1990) A Pricing Method for Options Based on Average Asset Values, *Journal of Banking and Finance*, 14, 113-129
■Kienitz, J. (2005), Stochastic Dynamics, Lecture Notes, University of Bonn
■Kloeden, P., Platen E. (1999), *Numerical Solution of Stochastic Differential Equations*, Springer 3rd edition
■Levy, E. (1992), Pricing European Average Rate Currency Options*Journal of International Money and Finance*, 14, 474-491
■Matsumoto, M, Nishimura, T. (1998), Mersenne Twister: a 623-dimensionally equidistributed uniform pseudorandom number generator, ACM Transactions on Modeling and Computer Simulation, 8(1): 3-30
and http://www.math.keio.ac.jp/~matsumoto/emt.html
■Merton, R. (1973), Theory of Rational Option Pricing, *Bell Journal of Economics & Management*
■Reiner, E., Rubinstein M. (1991), Breaking Down the Barriers, *Risk* 4, 8, pp. 28-35