Monte Carlo for the newbies

Simon Léger

April 06, 2006

Abstract

The purpose of this paper is to be an introduction to Monte Carlo techniques used in finance, mainly to price complicated products, called *exotics*. This paper has been designed for a non-expert public and focuses on giving a broad overview of Monte Carlo techniques from a practical point of view and presents also some improvements that can be done to increase the efficiency.

After explaining where this technique comes from, we will see how we can price financial products with it and how to implement such a system. We will also get an overview of more complicated techniques that allow us to improve its efficiency.

Contents

Introduction

Why for "the newbies" ?
Because this article can be read by anyone who is interested by this technique and how it is applied to finance. It looks like a complete introduction, as long as this is something possible... meaning, we are going to treat a broad domain without getting too much in the details as many articles on specific subjects can be easily found on the web.

This article has essentially been written for students or curious people who want to discover this wonderful tool.. I am myself one of its biggest fan and I have to say that it is very difficult to find easy enough articles to understand them and complete enough to learn new things. I hope that this paper will fill in this lack.

For those who are curious to know more about me, you can visit my website at http://homepages.nyu.edu/~sl1544/index.html. You can also contact me by email at simon.leger@nyu.edu.

1 Origin

1.1 Geometrical application

The Monte Carlo principle comes from the geometry and more precisely from area calculation. This technique has been enabled thanks to the development of powerful computers.

Let us take an easy exemple, for exemple let us try to compute an approximated value of π, or the area of a unit circle which is equal to : $\mathcal{A} = \pi * r^2$. For this we will consider a unit square with a quarter of a unit circle included inside like on figure 1 below:

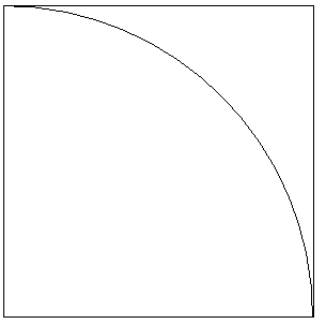

Figure 1: Introduction exemple

We are now going to simulate random points in the square [0,1], i.e. we need to simulate couple of points (x,y) with x and y being uniform on [0,1].

This means that for all $a < b, c < d$ in [0,1],

$$\mathbb{P}\left(X \in [a,b], Y \in [c,d]\right) = (b-a) * (d-c)$$

We now obtain a figure similar to this one, with only a few random points :

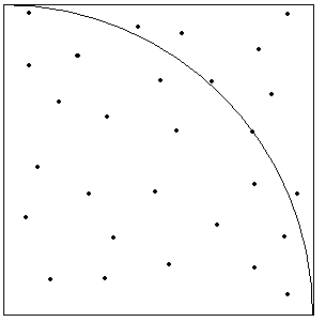

Figure 2: Measure of occupied space

Let us suppose that we have simulated 100 points, how does this enable us to compute the area of the quarter of the circle and hence an approximated value of π ? If we think about it, the space of this unit square can be filled by a lot of points and then the area of this quarter of circle would just be the ratio of points inside the circle to the total number of simulated points inside of the square of area 1... If the area of the square were to be 2 then we would need to multiply this ratio by 2.

Some are going to wonder at this point how to count the number of points inside the circle from a practical point of view. Well, then we just need to remember that a point with coordinates (x,y) is in a circle with radius r if and only if $x^2 + y^2 < r$.

We now see how the algorithm is designed :

```
double N=maximum number of simulations
double count=0
for i=1 to N
   x=uniform [0,1]
   y=uniform [0,1]
   if (x^2+y^2<1) do
      count=count+1
end i
double result=count/N
```

We get a quite good approximation of the area of our circle for N big enough (we will precise this later...). By multiplying this by 4 we get a good approximation of π. We will note here that the convergence speed of our algorithm is directly related to the performance of our random numbers generator, i.e. the more the points are equally distributed in our square, the faster the algorithm will converge to the true value. Actually, we can even say that these points do not really need to have a randomness, as long as they fill in other conditions...

2 Financial modelling

2.1 Hypothesis

Now that we have understood how this technique can be applied to area calculation, let us try to apply it to finance, and particularly to the pricing of derivatives products. One has to understand here that this principle can not be applied to the prediction of the evolution of an asset, but to the simulation of many possibilities once we have assumed a certain diffusion process. We get then many paths on which we can price our product and by admitting that these paths have the same probability, we compute the average of these prices to get an approximation of its real price under the assumptions.

This all seems very theoretical now, but let us use a simple exemple, by using the Black-Scholes diffusion equation.

This is a particular case of the general equation :

$$dS_t = \mu(t, S_t)dt + \sigma(t, S_t)dW_t$$

where :

$$\begin{cases} \mu(t, S_t) & = \mu S_t \\ \sigma(t, S_t) & = \sigma S_t \end{cases}$$

The stock then evolves according to a given equation, it evolves at the rate r and chocs occur. The well-known solution of this equation is :

$$S_T = S_t * e^{(r-\frac{1}{2}\sigma^2)(T-t)+\sigma(W_T-W_t)}$$

where $W_T - W_t \rightsquigarrow \mathcal{N}(0, T-t)$. We can write the solution under the following form :

$$S_T = S_t * e^{(r-\frac{1}{2}\sigma^2)(T-t)+X}$$

where $X \rightsquigarrow \mathcal{N}(0, \sigma^2(T-t))$. If we can generate a $\mathcal{N}(0,1)$, we just need to multiply it by $\sigma\sqrt{T-t}$ to get our X. We can now simulate our diffusion process of the asset S_t.

2.2 Classic *Call* case

Let us now use the previous exemple to price a classic *european* call whose payoff is $[S_T - K]^+$ where K is the *Strike* of the call chosen at the beginning of the contract and T is the *maturity*. *European* means that the payoff can only be exercised at the maturity T and not before in which case the call would be called *American*.

On the figure below we can see an exemple of a simulation and we can see how to determine the price of our option. Note that the simulated paths never have such a smooth aspect, but it is very hard (it is impossible actually) to draw a Brownian motion (due to the fact that the length of its path is infinite, whatever the size of the interval you look at it).

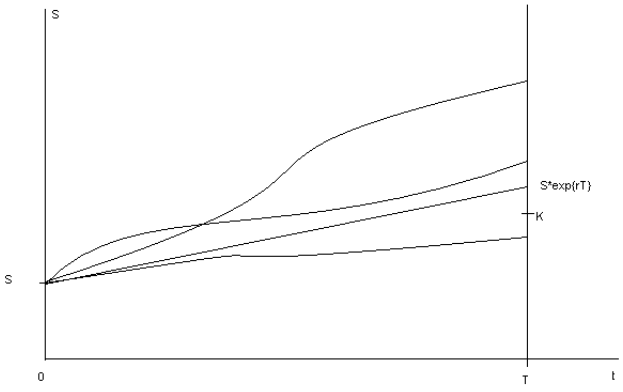

Figure 3: Small simulation

The simulated paths have a lognormal distribution and the expectation of the price of the stock S_T satisfies :$\mathbb{E}[S_T] = S_t e^{r(T-t)}$. This comes from the fact that for a normal random variable $X \rightsquigarrow \mathcal{N}(m, \sigma^2)$, we have :

$$\mathbb{E}_t(e^X) = e^{m+\frac{1}{2}\sigma^2}$$

which gives here :

$$\begin{cases} \mathbb{E}_t[S_T] & = S_t e^{(r-\frac{1}{2}\sigma^2)(T-t)} \mathbb{E}_t e^{\mathcal{N}(0,\sigma^2(T-t))} \\ & = S_t e^{(r-\frac{1}{2}\sigma^2)(T-t)+\frac{1}{2}\sigma^2(T-t)} \\ & = S_t e^{r(T-t)} \end{cases}$$

We get an array of possible prices at T satisfying the chosen distribution of the asset, let us call them $\left(S_T^1, S_T^2, ..., S_T^N\right)$ and let us apply to them the

payoff function, the array of payoffs is then : $\left(\left[S_T^1 - K\right]^+, \left[S_T^2 - K\right]^+, , ..., \left[S_T^N - K\right]^+\right)$ and the price P of the option is given by :

$$P = \frac{\sum_{i=1}^{N} \left[S_T^i - K\right]^+}{N}$$

2.3 Results obtained

Now that we know how to price a simple option, we want to know if this really works and how close the result is to reality. This exemple has not been chosen randomly and is particularly useful in this case since there exists a closed formula for its price, i.e. a theoretical price under the assumptions. The price of a call is given by :

$$P = S_t \mathcal{N}(d_1) - K e^{-r(T-t)} \mathcal{N}(d_2)$$

where :

$$\begin{cases} d_1 &= \frac{ln(\frac{S_t}{K}) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}} \\ d_2 &= d_1 - \sigma\sqrt{T-t} \end{cases}$$

Let us take an exemple in which case the theoretical value is 4.94387 and program an algorithm in C++ (to be prefered to VBA for instance because of its speed, important factor for this kind of application). We will use here the generator of random variables `rand` of C++. Here are the prices and times (in seconds) depending on the number of simulations. This test has been ran on a 3Ghz computer :

	300k		1M		10M		50M	
Generator	Price	Time	Price	Time	Price	Time	Price	Time
rand C++	4.96	2.156	4.935	7.172	4.9432	70.98	4.9461	355.62

We see that the price converges quite fast on this simple product, but one should ask why we find the same price...

2.4 Why does this work ?

Well, until now we have seen a few exemples of applications of this quite intuitive principle, but I did not tell you *why* this works. The principle relies on the large numbers law : if $X_1, X_2, ..., X_N$ is a set of iid random variables of law X(i.e. if they are N independent values of law X, then :

$$\lim_{n \to \infty} \frac{X_1 + X_2 + ... + X_n}{n} = \mathbb{E}[X]$$

Now that we believe that this principle works (and especially now that we know why, because intellectual satisfaction has no price...), i.e. that the average of prices found will converge to the true price for enough simulations.

2.5 *Asian* options

Here is another exemple, but for other reasons, since here the Monte Carlo principle becomes really necessary, even if there exists other ways to price this kind of products. Why ? Because this time, there is no closed formula giving the price of an asian call. But wait, what is by the way an asian option ? The easiest is to give the payoff formula :

$$P = \left[\frac{1}{T-t}\int_t^T S_u du - K\right]^+$$

This product allows to avoid the huge fluctuations of the price of the asset at maturity by averaging its price on the time interval considered. In practice, we just pick up some dates, one value per month for exemple, this is what we are going to consider in this section.

By analogy with the european call we see how to solve our problem with a Monte Carlo simulation, we are going to simulate some paths followed by the stock at the considered dates and apply to them our new payoff.

We feel that it will be harder to get a precise price due to the high number of possible paths... We can now switch to the general case.

2.6 General case for 1 asset

The general case when we only focus on one asset could be summarized as follows : can you give me the price at t of an option whose payoff function is the following : $f(S_t, t <= T)$ where f will be applied at T only and when you assume a certain diffusion process for the asset ? And the answer is then yes, and the process is very similar and easy : just simulate the stock at the dates you need for the computation of f a big number of times, apply the function f to each path and take the average of the prices which gives us the desired result. There *only* remains a programmation issue...

2.7 General case for n assets

The problem here is very similar, except that the only difference is that you need to simulate all stocks at different dates and that usually they are not independent (once you assume a certain form of correlation between the assets). As a consequence, instead of simulating n independent brownians, we want to simulate n brownians satisfying :

$$\mathbb{E}[dW_i dW_j] = \rho_{ij} dt$$

The *only* problem is then a numerical simulation issue. How from n independent brownians can we simulate correlated brownians ? The answer is easy, one just needs to use a numerical method to compute the Cholesky decomposition, which is a triangular matrix. This matrix is applied to the

matrix of simulated brownians to get the correlated values.We can test the precision of this result by calculating the correlation matrix and verifying that they are quite similar. We now have an easy way to simulate correlated brownians which allows us to go from one asset to n assets.

3 How to build a Monte Carlo pricer

3.1 Approach used

The aim of this article is not to explain step by step how to code a Monte Carlo pricer, we wont get into the details of programmation, everyone is supposed to know how to use inheritance methods and so on...

We will focus more on the theoretical aspects of the implementation of such a pricer, like the generation of random numbers, the calculation of greeks and how to improve the efficiency of these calculations.

3.2 Generation of random numbers

3.2.1 Introduction to the problem

We now get into one of the most important aspects of the implementation of a Monte Carlo pricer : the generation of random numbers. The efficiency in terms of precision in results and in terms of speed will depend on it.

To give more sense to its importance, here are some results with different generators, we will come back later to them. The exact price of the call in this exemple is 4.94387 :

	300k		1M		10M		50M	
Generator	Price	Time	Price	Time	Price	Time	Price	Time
RandC	4.96	2.156	4.935	7.172	4.9432	70.98	4.9461	355.62
ParkMiller	4.987	1.968	4.962	6.532	4.9448	67.7	4.9446	324.09
MersenneTwister	4.936	1.984	4.949	6.547	4.9461	65.08	4.9435	325.73
Sobol	4.94354	1.95	4.94372	6.42	4.94385	64.26	4.94387	319.95

The generator of random numbers *RandC* is the default random number generator in C++ for uniform numbers. As we can see, this one does not manage with 50 millions of simulations to get the efficiency of *Sobol* with only 300 000 simulations !

3.2.2 The different types of generators

First of all there exists two main categories of random number generators : the *pseudo* generators and the *quasi* generators, which lead to the techniques of Monte Carlo and of *quasi* Monte Carlo. The first one are the easiest to use, since these generators are supposed to produce truly random numbers,

according to a common point of view, i.e.e one can not predict the next number. However, it seems necessary to precise here that it is impossible anyway to produce real random numbers with a computer code. These generators are actually based on the linear congruence :

$$x_{n+1} = (ax_n + b)mod(m)$$

This type of generator produces a series (pseudo random) of numbers between 0 and $m-1$ whose period is then m. By dividing these numbers by m, we get a series of uniform numbers over [0,1].

Some generators have improvements based on this principle to increase the period, such that the Mersenne Twister generator has a period of 10^{6000} ! It uses 624 generator words and is equi-distributed in 623 dimensions, which can be useful in the case of many assets.

The other kind of generators to which *Sobol* belongs corresponds to the quasi random numbers generators. Here the goal is not to generate real random numbers, but rather to build numbers that fill in perfectly the interval [0,1] from a uniform point of view (*low discrepancy sequences*), and also in multiple dimensions. One can understand that this is enough for what we are doing if he thinks to the exemple at the beginning of this article. Indeed, the area calculation problem supposes that we consider all spaces of our square in the same way, whatever the real randomness of these numbers. This is the same for financial applications, we want to explore all possibilities according to our model, without preferring one particular direction, and this type of generators is very well fitted for this, as we can see on the preceding chart, which shows the efficiency of this generator.

The problem of these generators (of course there is one...) is the difficulty of their implementation and especially in multiple dimensions to avoid clustering effects. Foe exemple Sobol can be used until dimensions 160-600 depending on the source, and some people pretend to be able to use it until dimension 6000, but it becomes really hard to find informations on this... We note that there also exists other quasi random generators that can be more or less efficient depending on the case.

We also note that the convergence speed of Monte Carlo process in general case is $\mathcal{O}(n^{-1/2})$ and $\mathcal{O}(log^d n/n)$ for quasi Monte Carlo in d dimensions.

3.2.3 From uniform to normal

Now that we know how to generate uniform numbers in [0,1], we just need to go from this to a normal law $\mathcal{N}(0,1)$. Indeed, if we want to go from a $\mathcal{N}(0,1)$ to a $\mathcal{N}(m,\sigma^2)$ we just need to use the following property :

$$X \rightsquigarrow \mathcal{N}(0,1) \Rightarrow (m + \sigma X) \rightsquigarrow \mathcal{N}(m,\sigma^2)$$

The classic way to realize this inversion is to use the Box-Muller algorithm which transforms two uniform numbers in two normal numbers at the same

time. However, this algorithm is not very accurate, especially for quasi random number generators since it destroys some properties of these series (order and uniformity) and it is slower than Moro's inversion algorithm for exemple. But let us explain briefly how it works since it is very easy : if we have two uniform variables : x_1, x_2 then by applying :

$$\begin{cases} y_1 &= \sqrt{-2log(x_1)}cos(2\pi x_2) \\ y_2 &= \sqrt{-2log(x_1)}sin(2\pi x_2) \end{cases}$$

we get two independent normal variables y_1, y_2.

This algorithm is based on Beasley&Springer (1977) algorithm which is very efficient, except for the tails of the distribution, for which Moro's one brings its flavor. The tails are in this case modelled using truncated Chebyshev's series. Usually, we use Beasley&Springer for $|x| \leq 0.42$ and Moro's improvement for $|x| > 0.42$. We should note here one important practical fact, the function NORMSINV() of Excel which is supposed to do this job returns very bad results for tails values, although Moro's algorithm is very precise on the whole distribution. Moreover, it is really not difficult to find a C++ source code for this on internet.

3.3 How to build correlated arrays

Now that we know how to generate a normal law, we just need to learn how to correlate multiple random arrays in order to simulate joint distributions, necessary for the pricing of rainbow options.

In the case of just two correlated assets, there is an easy way to build two correlated normal variables. If we have X et Y and we wish to correlate them with a correlation ρ we just need to use X et $Z = \rho X + \sqrt{1-\rho^2}Y$. One can easily check that Z et X are correlated with the good correlation ρ and that they are still standard normals.

In the case of n assets, we need to use a more complicated method but which relies on the same principle. For this, we need as input the correlation matrix of these n assets that we call Σ. We then use the *Cholesky* decomposition matrix by writing :

$$\Sigma = U^T U$$

where U is a triangular matrix. Then if $N_1, ..., N_n$ are n independent normals,

$$\begin{pmatrix} X_1 \\ \\ \vdots \\ \\ X_n \end{pmatrix} = U \begin{pmatrix} N_1 \\ \\ \vdots \\ \\ N_n \end{pmatrix}$$

the X_i are now n correlated normals with the given correlation matrix Σ. I would advise here to check this new correlation matrix to be sure that there is no error in the decomposition process and that the number of simulated numbers is important enough to get close enough values.

We are now able to jointly simulate as many variables as we wish and then to price approximately any option or exotic product. Once this work is achieved, there remains a very important step, which is to know our risks and be able to hedge them. For this, we need to compute the greeks relative to the considered product.

4 Greek computation

4.1 What are these *greeks* ?

These greeks are numerical values that indicate the sensibility of the price of the considered product at the courant date with respect to one of the inputs of the price.

For exemple, in the case of a classic call option, the *delta* is the derivative of the price of the call now with respect to the price of the underlying stock. Here are the main greeks :

- the delta (δ) : it is the derivative of the price of the option with respect to the price of the underlying.
- the gamma (γ) : it is the second derivative of the price of the option with respect to the price of the underlying.
- the vega : it is the derivative of the price of the option with respect to the volatility.
- the theta (θ) : it is the derivative of the price of the option with respect to the time.
- the rho (ρ) : it is the derivative of the price of the option with respect to the interest rate.

One can of course imagine all greeks he can. For exemple in the case of rainbow options, we can compute the greek letter relative to the correlation between two stocks and all partial greeks we want...

When we have a closed formula of the option price, we can compute the greek analytically by deriving this price. We are more interested here in products for which we do not have a closed formula. In this case, how to estimate the greeks ?

4.2 Simple approach

In this case we use again our Monte Carlo framework, and there is not a lot to add to get these greeks. We use a method called *finite difference*. For exemple in the case of the delta, we use the following formula :

$$\delta_P = \frac{P(S_0 + \epsilon) - P(S_0 - \epsilon)}{2\epsilon}$$

and for the gamma for exemple :

$$\gamma_P = \frac{P(S_0 + \epsilon) + P(S_0 - \epsilon) - 2P(S_0)}{\epsilon^2}$$

Or even better in order to eliminate terms of order 2 and 3 :

$$\gamma_P = \frac{-P(S_0 + 2\epsilon) + 16P(S_0 + \epsilon) - 30P(S_0) + 16P(S_0 - \epsilon) - P(S_0 - 2\epsilon)}{12\epsilon^2}$$

Now, how to do this in practice ? First, in order to do this in an efficient way, we need to use the same path to price the option and the *shifted* option. Indeed, the difference between these two prices is going to be very small and if we use different paths, the Monte Carlo approximation error will be higher, and this difference will not be seen. We want the only difference to come from the different in prices due to the difference in inputs, and this is then the sensibility we want to compute, and the convergence speed of the greeks will be in this case improved a lot.

In maths terms, what we are doing is the following. Let us say ou price is a function of a given parameter θ and we want to compute the sensibility with respect to this parameter :

$$\begin{cases} \frac{\partial}{\partial\theta}\mathbb{E}^{\mathbb{Q}}\left(f(Y(\theta))|\mathcal{F}_{T_0}\right) & \simeq \frac{\partial}{\partial\theta}\hat{\mathbb{E}}^{\mathbb{Q}}\left(f(Y(\theta))|\mathcal{F}_{T_0}\right) \\ & \simeq \frac{1}{2\epsilon}\left(\hat{\mathbb{E}}^{\mathbb{Q}}\left(f(Y(\theta+\epsilon))|\mathcal{F}_{T_0}\right) - \hat{\mathbb{E}}^{\mathbb{Q}}\left(f(Y(\theta-\epsilon))|\mathcal{F}_{T_0}\right)\right) \\ & = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2\epsilon}\left(f(Y(\omega_i,\theta+\epsilon) - f(Y(\omega_i,\theta-\epsilon)\right) \end{cases}$$

If ϵ is not small enough, we are not sure to converge to the exact value, even if the number of simulations is high. If ϵ is small enough, the convergence can sometimes be very slow due to some irregularities of the payoff function. We are then going to see more advanced methods to compute these greeks.

4.3 More advanced methods

4.3.1 Path differenciation

We start again from our previous equation and we can end it in the following way :

$$\begin{cases} \frac{\partial}{\partial\theta}\mathbb{E}^{\mathbb{Q}}\left(f(Y(\theta))|\mathcal{F}_{T_0}\right) & \simeq \frac{\partial}{\partial\theta}\hat{\mathbb{E}}^{\mathbb{Q}}\left(f(Y(\theta))|\mathcal{F}_{T_0}\right) \\ & = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial\theta}\left(f(Y(\omega_i,\theta)\right) = \frac{1}{n}\sum_{i=1}^{n}\left(f'(Y(\omega_i,\theta)\right)\frac{\partial Y(\omega_i,\theta)}{\partial\theta} \end{cases}$$

However this requires more information on the payoff : f must be differentiable, and we also need more information on θ. But this computation can be ended like this :

$$\begin{cases} \frac{\partial}{\partial\theta}\mathbb{E}^{\mathbb{Q}}\left(f(Y(\theta))|\mathcal{F}_{T_0}\right) & = \frac{\partial}{\partial\theta}\int_\Omega f(Y(\omega,\theta))d\mathbb{Q}(\omega) = \int_\Omega \frac{\partial}{\partial\theta}f(Y(\omega,\theta))d\mathbb{Q}(\omega) \\ & = \int_\Omega f'(Y(\omega,\theta))\frac{\partial Y(\omega,\theta)}{\partial\theta} = \mathbb{E}^{\mathbb{Q}}\left(f'(Y(\theta))\frac{\partial Y(\omega,\theta)}{\partial\theta}|\mathcal{F}_{T_0}\right) \\ & \simeq \hat{\mathbb{E}}^{\mathbb{Q}}\left(f'(Y(\theta))\frac{\partial Y(\omega,\theta)}{\partial\theta}|\mathcal{F}_{T_0}\right) = \frac{1}{n}\sum_{i=1}^{n}\left(f'(Y(\omega_i,\theta)\right)\frac{\partial Y(\omega_i,\theta)}{\partial\theta} \end{cases}$$

The advantage of this formulation is that discontinuous payoffs can be handled this way, by interpreting f as a distribution.

4.3.2 Other methods

More complex methods exist and are more efficient in certain cases, like variance reduction method, or Malliavin calculus, which are above the scope of this article, and for which a huge documentation can be found on internet, so we will not spend more time on this.

5 Conclusion

We have spoken about many aspects of Monte Carlo methods applied to finance. We first tried to understand the principle of this tool, then we have seen how to apply it to the pricing of options in finance and how to realize the pricer step by step. Then, we have introduced the greeks and have seen an easy way to compute way to compute them. WE have indicated other ways to improve them but without getting too much in the details, the curious reader will be able find a lot of articles for this.