Computational Finance

Christian Bayer, Antonis Papapantoleon, Raul Tempone

Version of June 25, 2018

Contents

1 Introduction **1**

2 Monte Carlo simulation **4**

2.1 Random number generation 4

2.2 Monte Carlo simulation 14

Chapter 1

Introduction

One of the goals in mathematical finance is the pricing of derivatives such as options. While there are certainly also many other mathematically and computationally challenging areas of mathematical finance (such as portfolio optimization or risk measures), we will concentrate on the problems arising from option pricing. The techniques presented in this course are also often used in computational finance in general, as well as in many other areas of applied mathematics, science and engineering.

The most fundamental model of a financial market consists of a probability space $(\Omega, \mathcal{F}, P)$, on which a random variable S is defined. In the simplest case, S is $\mathbb{R}$ (or $[0, \infty[$) valued and simply means the value of a stock at some time T. However, S might also represent the collection of all stock prices S_t for $t \in [0, T]$. Then S is a random variable taking values in the (infinite-dimensional) path space, i.e., either the space of continuous functions $C([0,T]; \mathbb{R}^d)$ or the space of càdlàg functions $D([0,T]; \mathbb{R}^d)$ taking values in $\mathbb{R}^d$. Then the payoff function of almost any *European option* can be represented as $f(S)$ for some functional f.

Example 1.1. The European call option (on the asset S^1) is given by

$$f(S) = \left(S^1_T - K\right)^+.$$

Example 1.2. An example of a look-back option, consider the contract with payoff function

$$f(S) = \left(S^1_T - \min_{t \in [0,T]} S^1_t\right)^+.$$

Example 1.3. A simple barrier option (down-and-out) could look like this (for the barrier $B > 0$):

$$f(S) = \left(S^1_T - K\right)^+ \mathbf{1}_{\min_{t \in [0,T]} S^1_t > B}.$$

In all these cases, the problem of pricing the option can therefore be reduced to the problem of computing

$$E\left[f(S)\right]. \tag{1.1}$$

Indeed, here we have assumed that we already started with the (or a) risk neutral measure P. Moreover, if the interest rate is deterministic, then discounting is trivial. For stochastic interest rates, we may assume that the stochastic interest rate is a part of S (depending on the interest rate model, this might imply that the state space of the stochastic process S_t is infinite-dimensional, if we use the Heath-Jarrow-Morton model, see [5]). Therefore, the option pricing problem can still be written in the form (1.1) in the case of stochastic interest rates by incorporating the discount factor in the "payoff function" f.

Of course, we have to assume that $X := f(S) \in L^1(\Omega, \mathcal{F}, P)$. Then the most general form of the option pricing problem is to compute $E[X]$ for an integrable random variable X. Corresponding to

this extremely general modeling situation is an extremely general numerical method called *Monte-Carlo simulation*. Assume that we can generate a sequence $(X_i)_{i\in\mathbb{N}}$ of independent copies of X.[1] Then, the strong law of large numbers implies that

$$\frac{1}{M}\sum_{i=1}^{M} X_i \xrightarrow[M\to\infty]{} E[X] \tag{1.2}$$

almost surely. Since the assumptions of the Monte-Carlo simulations are extremely weak, we should not be surprised that the rate of convergence is rather slow: Indeed, we shall see in Section 2.2 that the error of the Monte-Carlo simulation decreases only like $\frac{1}{\sqrt{M}}$ for $M \to \infty$ in a certain sense – note that the error will be random. Nevertheless, Monte-Carlo simulation as a very powerful numerical method, and we are going to discuss it together with several modifications in Chapter 2.

While the assumption that we can generate samples from the distribution of S might seem innocent, it poses problems in many typical modeling situations, namely when S is defined as the solution of a *stochastic differential equation* (SDE). Let $\left(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\in[0,T]}, P\right)$ be a filtered probability space satisfying the usual conditions. In many models, the stock price S_t is given as solution of an SDE of the form

$$dS_t = V(S_t)dt + \sum_{i=1}^{d} V_i(S_t)dB_t^i, \tag{1.3}$$

where $V, V_1, \ldots, V_d : \mathbb{R}^n \to \mathbb{R}^n$ are vector fields and B denotes a d-dimensional Brownian motion. (If we replace the Brownian motion by a Lévy process, we can also obtain jump-processes in this way.) In general, it is not possible to solve the equation (1.3) explicitly, thus we do not know the distribution of the random variable $X = f(S)$ and cannot sample from it. In Chapter **??** we are going to discuss how to solve SDEs in a numerical way, in analogy to numerical solvers for ODEs (ordinary differential equations). Then, the option price (1.1) can be computed by a combination of the numerical SDE-solver (producing samples from an approximation of $f(S)$) and the Monte-Carlo method (1.2) (applied to those approximate samples).

If the option under consideration is "Markovian" in the sense that the payoff function only depends on the value of the underlying at time T, i.e., the payoff is given by $f(S_T)$, then the option price satisfies a partial differential equation (PDE).[2] Indeed, let

$$u(s,t) = E\left[f(S_T)|S_t = s\right],$$

and define the partial differential operator L by

$$Lg(s) = V_0 g(s) + \frac{1}{2}\sum_{i=1}^{d} V_i^2 g(s),$$

$s \in \mathbb{R}^n$, where the vector field V is applied to a function $g : \mathbb{R}^n \to \mathbb{R}$ giving another function $Vg(s) := \nabla g(s) \cdot V(s)$ from $\mathbb{R}^n$ to $\mathbb{R}$ and $V_i^2 g(s)$ is defined by applying the vector field V_i to the function $V_i g$. Moreover, we have

$$V_0(x) := V(x) - \frac{1}{2}\sum_{i=1}^{d} DV_i(x) \cdot V_i(x),$$

with DV denoting the Jacobian matrix of the vector field V. Then we have (under some rather mild regularity conditions)

$$\begin{cases} \dfrac{\partial}{\partial t} u(t,s) + Lu(t,s) = 0, \\ u(T,s) = f(s). \end{cases} \tag{1.4}$$

[1] By this statement we mean that we have a random number generator producing (potentially infinitely many) random numbers according to the distribution of X, which are independent of each other.

[2] In fact, we can find such PDEs in much more general situations!

Therefore, another approach to solve our option pricing problem in a numerical way is to use the well-known techniques from numerics of PDEs, such as the finite difference or finite element methods. We will present the finite difference method in Section **??**. We note that a similar partial differential equation also holds when the SDE is driven by a Lévy process. Then the partial differential operator L is non-local, i.e., there is an integral term. Note that there are also finite difference and finite element schemes for the resulting partial integro-differential equations, see [3] and [10], respectively.

There is a very fast, specialized method for pricing European call options (and certain similar options) on stocks S_T, such that the characteristic function of $\log(S_T)$ is known (we take S_T to be one-dimensional). This condition is actually satisfied in quite a large class of important financial models. Let ϕ_T denote the characteristic function of $\log(S_T)$ and let $C_T = C_T(K)$ denote the price of the European call option with strike price K. Moreover, we denote its Fourier transform by $\hat{C}_T$. Then

$$\hat{C}_T(\mu) = \frac{\phi_T(\mu - i)}{i\mu - \mu^2},$$

i.e., we have an explicit formula for the Fourier transform of the option price.[3] Now we only need to compute the inverse Fourier transform, which is numerically feasible because of the FFT-algorithm.

Unfortunately, most options encountered in practise are American options, and the above treated methods do not directly apply for American options. Indeed, the pricing problem for an American option is to find

$$\sup_{\tau \leqslant T} E\left[f(S_\tau)\right], \tag{1.5}$$

where τ ranges through all stopping times in the filtered probability space. So, it is not obvious how to apply any of the methods presented above. We will discuss one numerical method for American options in detail and hint at some modifications of the standard methods suitable for computing prices of American options, see Section **??**

The book of Glasserman [5] is a wonderful text book on Monte Carlos based methods in computational finance, i.e., it covers Chapter 2 and Chapter **??** in great detail. On the other hand, Seydel [16] does also treat Monte Carlo methods, but concentrates more on finite difference and element methods. Wilmott [19] is a very popular, easily accessible book on quantitative finance. It covers many of the topics of the course, but the level of mathematics is rather low. For the prerequisites in stochastic analysis, the reader is referred to Øksendal [13] for an introduction of SDEs driven by Brownian motion. Cont and Tankov [2] is the text book of choice for Lévy processes, and Protter [14] treats stochastic integration and SDEs in full generality.

[3] For integrability reasons, the above formula is not true. Indeed, we have to dampen the option price, introducing a damping parameter. For the precise formulation, see Section **??**.

Chapter 2

Monte Carlo simulation

2.1 Random number generation

The key ingredient of the Monte Carlo simulation is sampling of independent realizations of a given distribution. This poses the question of how we can obtain such samples on a computer. We will break the problem into two parts: First we try to find a method to get independent samples from a *uniform distribution* (on the interval $]0,1[$), then we discuss how to get samples from general distributions provided we know how to sample the uniform distribution.

Uniform pseudorandom numbers

Computers do not know about randomness, so it is rather obvious that we cannot get *truly* random numbers if we trust a computer to provide them for us. Therefore, the numbers produced by a random number generator (RNG) on a computer are often referred to as *pseudorandom numbers*. If the "random" numbers, say, $u_1, u_2, \ldots$ produced by a random number generator, are not random but deterministic, they cannot really be realizations of a sequence $U_1, U_2, \ldots$ of independent, uniformly distributed random variables. So what do we actually mean by a random number generator? More precisely, what do we mean by a *good* random number generator?

Remark 2.1. Even though the questions raised here are somehow vague, they are really important for the success of the simulation. Bad random number generators can lead to huge errors in your simulation, and therefore must be avoided. Unfortunately, there are still many bad random number generators around. So you should rely on "standard" random number generators which have been extensively tested. In particular, you should not use a random number generator of your own. Therefore, the goal of this section is not to enable you to construct and implement a random number generator, but rather to make you aware of a few issues around random number generation.

Before coming back to these questions, let us first note that a computer usually works with finite arithmetic. Therefore, there is only a finite number of floating point numbers which can be taken by the stream random numbers $u_1, u_2, \ldots$. Therefore, we can equivalently consider a random string of integers $i_1, i_2, \ldots$ taking values in a set $\{0, \ldots, m\}$ with $u_l = i_l/m$.[1] Then the uniform random number generator producing $u_1, u_2, \ldots$ is good, if and only if the random number generator producing $i_1, i_2, \ldots$ is a good random number generator for the uniform distribution on $\{0, 1, \ldots, m-1\}$. Of course, this trick has not solved our problems. For the remainder of the section, we study the problem of generating random numbers $i_1, i_2, \ldots$ on a finite set $\{0, 1, \ldots, m-1\}$.

Formally, a random number generator can be defined as follows, see L'Ecuyer [7]:

Definition 2.2. A random number generator is a structure $(X, x_0, T, G, \{0, 1, \ldots, m-1\})$ where X is a finite set (the *state space*), $x_0 \in X$ is the initial state (the *seed*), $T : X \to X$ is a *transition*

[1] Integer is here used in its mathematical meaning not in the sense of a data type.

function, and $G : X \to \{0, \dots, m-1\}$ is the *output function*. Given a random number generator, the pseudorandom numbers are computed via the recursion

$$x_l = T(x_{l-1}) \quad \text{and} \quad i_l := G(x_l) \quad \text{for} \quad l = 1, 2, \dots.$$

Remark 2.3. There is an immediate unfortunate consequence of this definition: since X is finite, the sequence of random numbers (i_l) must be periodic. Indeed, there must exist an index ℓ such that $x_\ell = x_l$ for some $l < \ell$. This implies that $x_{\ell+1} = x_{l+1}$ and so forth. Note that this index ℓ can occur much later than the first occurrence of $i_k = i_{k'}$ for some $k' < k$! Nonetheless, Definition 2.2 arguably contains all possible candidates for good random number generators.

The following criteria for goodness have evolved in the literature on random number generators, see L'Ecuyer [7], L'Ecuyer et al. [8], and Glasserman [5]:

Statistical uniformity: The sequence of random numbers $i_1, i_2, \dots$ produced by the generator for a given seed should be hard to distinguish from truly random samples from the uniform distribution on $\{0, \dots, m-1\}$. This basically means that no *computationally feasible* statistical test for uniformity should be able to distinguish $(i_l)_{l \in \mathbb{N}}$ from a truly random sample. The restraint to computationally feasible tests is important: since we know that the sequence is actually deterministic (even periodic), it is easy to construct tests which can make the distinction. (The trivial test would be to wait for the period; then we see that the pseudorandom sequence repeats itself.) The requirement of statistical uniformity basically means that we cannot guess the next number i_{l+1} given only the previously realized numbers $i_1, \dots, i_l$, at least not better than by choosing at random among $\{0, \dots, m-1\}$, if we assume that *we do not know the algorithm*.[2] Note that by statistical uniformity we require more than just uniformity of the one-dimensional marginals. Indeed, for any dimension d we require that sequences of d-dimensional outputs are difficult to distinguish from truly random sequences according to the uniform distribution on $\{0, \dots, m-1\}^d$. Of course, this property would be a consequence of independence of the numbers $i_1, i_2, \dots$.

Theoretical support: Many properties of random number generators, like the period length and the lattice structure (or hyperplane property), can be studied at a theoretical level; see e.g the remarks below about linear congruential generators). RNGs with strong theoretical support should be used and the others should be avoided. In principle, the optimal approach in choosing random number generators is to first screen their theoretical properties and then submit to empirical tests those with convincing theoretical support.

Speed: In modern applications, a lot of random numbers are needed. In molecular dynamics simulations for example, up to 10^{18} random numbers might be used (during several months of computer time). In finance, most applications do not require more than, say, 10^6 random numbers. However, the generation of random numbers is often the bottleneck during a simulation. Therefore, it is very important that the RNG is fast.

Period length: If we need 10^{18} random numbers, then the period length of the RNG must be at least as high. In fact, usually the quality of randomness deteriorates well below the actual period length. As a rule of thumb it has been suggested that the period length should be one order of magnitude larger than the square of the number of values used; cf. Ripley [15].

Reproducibility: In order to debug code, for instance, it is very convenient to have a way of exactly reproducing a sequence of random numbers generated before. (By setting the seed this is, of course, possible for any RNG satisfying Definition 2.2.)

Portability: The RNG should be portable to different computers. Reliable implementations should be available for different operating systems and various programming languages.

[2] There is a stronger notion of *cryptographic security* which requires that we cannot guess i_{l+1} even if we are intelligent in the sense that we do know and use the RNG. In essence, cryptographic security thus means that we cannot compute the state x_l from $i_1, \dots, i_l$. While this property is essential in cryptography, it is not important for Monte Carlo simulations.

Jumping ahead: By "jumping ahead" we mean the possibility to quickly get to the state x_{l+n} given the state x_l for large n (i.e., without having to generate all the states inbetween). This is important for parallelization.

How do RNGs implemented on the computer actually look like? The prototypical class of RNGs are *linear congruential generators* (LCG). In the class of LCGs, the state space is $X = \{0, \dots, m-1\}$, the output function is the identity function $x_l = i_l$ and the transition map is provided by

$$x_{l+1} = (ax_l + c) \mod m. \tag{2.1}$$

Remark 2.4. Linear congruential generators are very well analyzed from a theoretical point of view, see Knuth [6]. For instance, we know that the RNG (2.1) has full period (i.e., the period length is m) if $c \neq 0$ and the following conditions are satisfied:

- c and a are relatively prime,
- every prime number dividing m also divides $a - 1$,
- if m is divisible by 4 then so is $a - 1$.

Nonetheless, it should be stressed that a high period is only one of the many requirements identified above. In particular, the requirement of statistical uniformity is very hard to analyse by theoretical tools alone. The choice of parameters a, c, m of an LCG is a largely empirical task, where suites of statistical tests are run on large sequences of pseudo-random numbers.

Source	m	a	c
Numerical Recipes	2^{32}	1664525	1013904223
glibc (GCC)	2^{32}	1103515245	12345
Microsoft C/C++	2^{32}	214013	2531011
Apple Carbonlib	$2^{31} - 1$	16807	0
Java	2^{48}	25214903917	11

Table 2.1: List of linear congruential RNGs as reported in [18].

Table 2.1 presents a list of linear congruential RNGs used in prominent libraries. Note that $m = 2^{32}$ is popular, since computing the remainder of a power of 2 in base-2 only means truncating the representation.

We conclude this discussion by pointing out a common weakness of all linear congruential RNGs. Fix $d \geqslant 1$ and consider the sequence of vectors $(i_l, i_{l+1}, \dots, i_{l+d-1})$ indexed by $l \in \mathbb{N}$. Note that for every l the truly random vector $(I_l, \dots, I_{l+d-1})$ is uniformly distributed on the set $\{0, \dots, m-1\}^d$. On the other hand, the pseudorandom vectors generated by linear congruential RNGs fail in that regard: they tend to lie on a (possibly) small number of hyperplanes in the hypercube $\{0, \dots, m-1\}^d$; see Figure 2.1 for an example in $d = 2$. It has been proved that they can lie at most on $(d!m)^{1/d}$ hyperplanes, but often the actual figure is much smaller.

One of the most popular modern random number generators as of today is the Mersenne Twister algorithm[3]. This RNG produces 32-bit integers, the state space is $\mathbb{F}_2^{19968}$ (in its most popular version), where $\mathbb{F}_2$ denotes the finite field of size two, and the period is $2^{19937} - 1$. It is not a linear congruential generator, but the basis of the transformation map T is a linear map in X – with additional transformations, though. Note that in this case, the size of the state space (2^{19968}) is much larger than the $m = 2^{32}$.

Let us finally comment on the parallel generation of random numbers. As we shall see later in Chapter 2, it is often desirable or even necessary to have the possibility to generate random numbers on many cores in parallel. Indeed, as a general trend in computing one can observe that computers are generally no longer accelerated by making processors ever faster, but instead by adding multiple

[3]Available at http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.

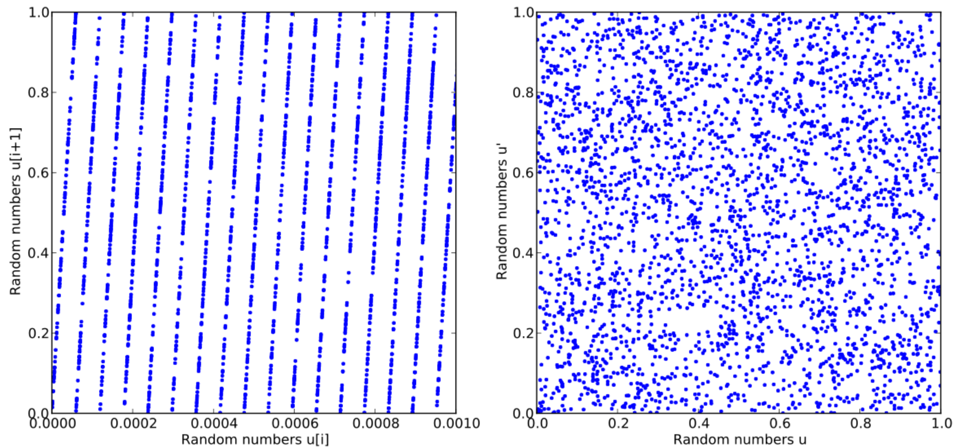

Figure 2.1: Hyperplane property for the linear congruential generator with $a = 16807$, $c = 0$, $m = 2^{31} - 1$. On the left, we have plotted 2 000 000 points (u_i, u_{i+1}), on the right 3000 pairs (i.e., 6000 random numbers plotted as pairs).

cores. This is especially true in graphics processors, where typical GPUs (graphical processing units) installed on average computers have dozens or even hundreds of cores, which are increasingly used also for general numerical purposes. In fact, vendors of GPUs are actively promoting these new applications; see e.g. NVIDIA [12]. [4] To cite L'Ecuyer et al. [8]:

> In highly parallel systems, one may need thousands or even millions of virtual RNGs which [...] run in parallel without exchanging data between one another, and behave from the user's viewpoint just like independent RNGs.

Before continuing, let us have a very cursory look at parallelization in general. Let us consider a simple program, which runs as a single process on the computer. Such a process can now start different *threads* which behave like processes of their own in as much as they can be executed on different cores in parallel, but have the big distinction that all the threads within a process share the same memory. This allows them to work with the same data and even use the output of other threads. As a simple example, think of a for-loop adding all the numbers stored in a very large array a (of size n): a natural parallelization would be to start l threads each summing up n/l (distinct) numbers (say, thread 1 computes $a[0] + \cdots + a[n/l - 1]$, thread 2 computes $a[n/l] + \cdots + a[2n/l - 1]$, ...), which are finally added to form the total sum. Hence, shared memory is necessary for successful parallelization, but it comes with a danger as different threads my end up using the same chunk of memory in incompatible ways. In general, problems come in the form of a *race-condition*, when the output of a process depends on the timing of threads within it, which produces a bug when this timing is different from the one anticipated by the programmer.

Example 2.5. As an example we consider the following highly simplified (and artificial) example in the context of RNGs. Let us assume we have one thread (thread 1), which runs an RNG and puts a random number into a double variable x. Whenever another thread accesses x, thread 1 will produce a new random number, which is again stored in x. We further have two threads (thread 2 and 3) which use random numbers produced by thread 1 for simulation. Now the intended sequence of events is that, say, thread 2 picks up the random number stored in x, then thread 1 updates x, and then thread 3 picks up the updated number in x. But in the absence of safety mechanisms, it could be that thread 3 is too fast, i.e., it accesses x already when thread 1 has not yet updated x, resulting in threads 2 and 3 using the same random number instead of independent ones.

[4] One limitation of GPUs as compared to classical CPUs is the rather small amount of rapidly accessible memory, which puts real restraints on the size of the seed or the dimensionality of the state space in an RNG context.

These problems mainly occur because developers for many decades were not concerned with parallel execution of code, which only became mainstream in the '90s. *Thread-safety* is the absence of any kind of race conditions, guaranteeing the safe execution of parallelized code. It is always important to check whether libraries or other pieces of code used in a parallel program are thread-safe!

Now, how can we generate parallel streams of random numbers? Let us describe several possible ways, along with their advantages and drawbacks:

- Use a central source of randomness for all threads, i.e., one thread produces all the random numbers for all other threads. As random number generation is often a bottleneck for applications (especially in a Monte Carlo framework) *and* data exchange between different threads is often the bottleneck in parallelization, this simple method is typically not acceptable.
- Use different RNGs for different threads, i.e., either truly different RNGs or the same class of RNGs but with different parameters. This requires one to have many good parameters / good RNGs available, and, besides, even if parameters / RNGs are individually good, their combination may fail the independence requirement. Hence, any such combination needs to be tested statistically, which makes it cumbersome to use this method for an arbitrary (high) number of streams.
- Use a single RNG split into equally-spaced blocks. Say we know that we have n threads which all may require (at most) ν random numbers. We use our favorite RNG with seed x_0 for thread 1. We jump ahead to step ν and use the RNG with seed x_ν for thread 2. In the same way, each thread uses the same RNG with seeds obtained by jumping ahead ν steps from the seed used by the previous thread. From a theoretical point of view, this method is most satisfactory, since good statistical properties of the RNG used imply good statistical properties of the sequence of streams constructed in that way. However, good RNGs can only be used for this method when they allow for rapid jumping-ahead. As in most varieties of RNGs the transition function T has the form of a matrix multiplication (say with a matrix A), this means that there must be a rapid way of computing A^ν, which is often not possible, especially if the state space X is extremely high-dimensional, such as in the case of the Mersenne Twister. Hence, it may be simpler to use an RNG constructed by the combination of two simpler RNGs defined on relatively low-dimensional state spaces. We refer to L'Ecuyer et al. [8] for references on good RNGs and suitable implementations for this purpose.
- Use one RNG with random seeds. If we have a good RNG with very high period, but bad jumping-ahead capability like the Mersenne Twister, then we may want to use n copies of the RNG with n seeds drawn from the state space X with the help of another RNG. While overlaps between the different streams are possible, they are extremely unlikely. Indeed, if the period of the RNG is ρ, then the probability of an overlap is approximately $(1 - n\nu/\rho)^{n-1}$. For instance, L'Ecuyer et al. [8] report that this probability is close to 2^{-964} when $l = \nu = 2^{20}$ and $\rho = 2^{1024}$. An added benefit of this method is that it is applicable when the number of random streams is not known beforehand, for instance because new random streams need to be generated depending on random events.

Finally, let us note that reproducibility may become an issue with parallelization, as the organization of threads and the assignment of tasks to a thread may be determined at execution time and may differ between two different executions. Hence, it may be advisable to assign streams at an abstract level, i.e., to distinct computational tasks instead of individual threads, the number and speed of execution of which may be hard to predict for the programmer.

Non-uniform random numbers

In many applications, we do not need uniform random numbers, but random numbers from another distribution. In the Black-Scholes model for instance, the stock price has the following dynamics:

$$S_T = S_0 \exp\left(\sigma B_T + \left(\mu - \frac{1}{2}\sigma^2\right) T\right).$$

Therefore, the stock price S_T has a log-normal distribution, while B_T has a normal distribution. Thus, there are two ways to sample the stock price: we can either sample from the log-normal or from the normal distribution.

For the rest of this section, and indeed, the whole text, we assume that we are given a perfect (i.e., truly random) RNG producing a sequence $U_1, U_2, \dots$ of independent $\mathcal{U}(]0,1[)$-distributed random numbers. We will present some general techniques to produce samples from other distributions, and then some specialized methods for generating normal (Gaussian) random numbers. An exhaustive treatment of random number generation can be found in the classical book of Devroye [4].

We start with a well-known result from probability theory, which readily implies the first general method for random number generation.

Proposition 2.6. *Let F be a cumulative distribution function and define*

$$F^{-1}(u) := \inf \{ x \mid F(x) \geqslant u \} .$$

Given a uniform random variable U, the random variable $X := F^{-1}(U)$ has the distribution function F.

Proof. By definition of F^{-1} we have $F^{-1}(u) \leqslant x \iff F(x) \geqslant u$, therefore

$$P(X \leqslant x) = P(F^{-1}(U) \leqslant x) = P(U \leqslant F(x)) = F(x). \qquad \square$$

Proposition 2.6 is the basis of Algorithm 2.7.

Algorithm 2.7 (Inversion method)**.** *Given F^{-1} and $U \sim \mathcal{U}(]0,1[)$, return $X = F^{-1}(U)$.*

Example 2.8. The exponential distribution with parameter $\lambda > 0$ has the distribution function $F(x) = 1 - \mathrm{e}^{-\lambda x}$, which is explicitly invertible with $F^{-1}(u) = -\frac{1}{\lambda}\log(1-u)$. Thus, using the fact that $1-U$ is uniformly distributed if U is, we can generate samples from the exponential distribution by

$$X = -\frac{1}{\lambda}\log(U).$$

Remark 2.9. If an explicit formula for the distribution function F is available, but not for its inverse F^{-1}, we can try to use numerical inversion. Of course, this results in random numbers, which are samples from an approximation of the distribution F only. Nevertheless, if the error is small and/or the inversion can be done efficiently, this method might be competitive even if more direct, "exact" methods are available.[5] For instance, approximations of the inverse of the distribution function Φ of the standard normal distribution have been suggested for the simulation of normal random variables, see Glasserman [5].

Remark 2.10. The transparent relation between the uniform random numbers $U_1, \dots, U_l$ and the transformed random numbers $X_1, \dots, X_l$ (with distribution F) underlying the inversion method allows to translate many structural properties on the level of the uniform random numbers to corresponding properties for the transformed random numbers. For instance, if we want the random numbers $X_1, \dots, X_l$ to be correlated, we can choose the uniforms to be correlated. Another example is the generation of the maximum $X^* := \max(X_1, \dots, X_l)$. Apart from the obvious solution (generating $X_1, \dots, X_l$ and finding their maximum), there are also two other possible methods for generating X^* based on the inversion method:

- Since X^* has the distribution function F^l, we can compute a sample from X^* by $(F^l)^{-1}(U_1)$. Efficiency of this method depends on the tractability of F^l.
- Let $U^* = \max(U_1, \dots, U_l)$. Then, using the monotonicity of F^{-1}, $X^* = F^{-1}(U^*)$. Since we only have to do one inversion instead of l, this method is usually much more efficient than the obvious method.

[5] We should note that many elementary functions like exp and log cannot be evaluated exactly on a computer. Therefore, one might argue that even the simple inversion situation of Example 2.8 suffers from this defect.

- Combining both approaches, we see that the c.d.f. of U^* is given by x^l, $0 \leqslant x \leqslant 1$, with inverse function $x^{1/l}$. So we obtain one sample from the distribution of U^* simply by $U_1^{1/l}$, and X^* has the same distribution as $F^{-1}(U_1^{1/l})$.

Next we present another general purpose method, which is based on the densities of the distributions involved instead of their distribution functions. More precisely, let $g : \mathbb{R}^d \to [0, \infty[$ be the density of a d-dimensional distribution, from which we can sample efficiently (by whatever method). We want to sample from another d-dimensional distribution with density f. The *acceptance-rejection method* works if we can find a bound $c \geqslant 1$ such that

$$f(x) \leqslant cg(x), \quad x \in \mathbb{R}^d. \tag{2.2}$$

Algorithm 2.11 (Acceptence-rejection method)**.** *Given an RNG producing independent samples X from the distribution with density g and an RNG producing independent samples U of the uniform distribution, independent of the samples X.*

1. *Generate one instance of X and one instance of U.*
2. *If $U \leqslant f(X)/(cg(X))$ return X;*[6] *else go back to 1.*

Proposition 2.12. *Let Y be the outcome of Algorithm 2.11. Then Y has the distribution given by the density f. Moreover, the loop in the algorithm has to be traversed c times on average.*

Proof. By construction, Y has the distribution of X conditioned on $U \leqslant \frac{f(X)}{cg(X)}$. Thus, for any measurable set $A \subset \mathbb{R}^d$, we have

$$\begin{aligned} P(Y \in A) &= P\left(X \in A \middle| U \leqslant \frac{f(X)}{cg(X)}\right) \\ &= \frac{P\left(X \in A,\ U \leqslant \frac{f(X)}{cg(X)}\right)}{P\left(U \leqslant \frac{f(X)}{cg(X)}\right)}. \end{aligned}$$

We compute the numerator by conditioning on X, i.e.,

$$\begin{aligned} P\left(X \in A,\ U \leqslant \frac{f(X)}{cg(X)}\right) &= \int_{\mathbb{R}^d} P\left(X \in A,\ U \leqslant \frac{f(X)}{cg(X)} \middle| X = x\right) g(x)dx \\ &= \int_A P\left(U \leqslant \frac{f(x)}{cg(x)}\right) g(x)dx = \int_A \frac{f(x)}{cg(x)} g(x)dx \\ &= \frac{1}{c}\int_A f(x)dx. \end{aligned}$$

On the other hand, a similar computation shows that $P\left(U \leqslant \frac{f(X)}{cg(X)}\right) = \frac{1}{c}$, and together we get

$$P(Y \in A) = \int_A f(x)dx.$$

Moreover, we have seen that the probability that the sample X is accepted is given by $1/c$. Since the different runs of the loop in the algorithm are independent, this implies that the expected "waiting time" is c, the expectation of a geometric distribution with parameter $1/c$. □

Naturally, we want c to be as small as possible. That is, in fact, the tricky part of the endeavour. Exercise 2.2 asks for a method to sample normal random variables starting from the exponential distribution, which we can sample by Example 2.8.

[6] Note that $P(g(X) = 0) = 0$.

Example 2.13. The *double exponential distribution* (with parameter $\lambda = 1$) has the density $g(x) = \frac{1}{2}\exp(-|x|)$ for $x \in \mathbb{R}$. Let $f = \varphi$ denote the density of the standard normal distribution. Then

$$\frac{\varphi(x)}{g(x)} = \sqrt{\frac{2}{\pi}} e^{-\frac{x^2}{2}+|x|} \leqslant \sqrt{\frac{2e}{\pi}} \approx 1.315 =: c.$$

Although the acceptance-rejection algorithm is a very general and exact transformation algorithm, i.e., if fed with truly random numbers it will produce random numbers which are exactly distributed according to the desired density, it can be quite inefficient if the parameter c is large. Marsaglia and Tsang [9] have constructed a fast and efficient variant of the importance sampling algorithm, which is still applicable in the majority of cases. For reasons to become clear later, they call their algorithm *Ziggurat* algorithm.

Like in the acceptance-rejection algorithm, the fundamental idea of the Ziggurat algorithm is based on the principle that sampling from the distribution given by a (say, univariate) density f is equivalent to sampling a point from the (say, bi-variate) uniform distribution in the area between 0 and the graph of f. The situation would be especially simple if this area was "pyramid" or "Ziggurat" shaped, i.e., had the form of rectangles (parallel to the abscissa) put on top of each other. In this case, we could first choose the rectangle at random (according to their respective volumes) and then we would only have to sample a uniform random number on the lower side of the rectangle – note that the second coordinate of the chosen random number in $\mathbb{R}^2$ does not really matter for the acceptance-rejection method, as long as it is guaranteed that the two-dimensional random variate is below the graph of the density. Now the idea of the Ziggurat algorithm is simply to approximate the area under the graph by such a Ziggurat-shaped polygon using tabulated values for the respective density, and accompany this by a classical acceptance-rejection method for the "remainder" of the area.

More precisely, assume we are given a density $f : [0,\infty[\to [0,\infty[$ which is monotonically decreasing like the density of the exponential distribution. Moreover, fix some $n \in \mathbb{N}$ and assume we aregiven a sequence $0 = x_0 < x_1 < \cdots < x_n$ such that the following condition holds with $y_i := f(x_i)$, $i = 0,\dots,n$:

$$x_i(y_{i-1} - y_i) = x_n y_n + \int_{x_n}^{\infty} f(x)dx =: v, \quad i = 1,\dots,n-1. \tag{2.3}$$

Obviously, the values $x_0,\dots,x_n$ depend on the distribution under consideration and on the numerical parameter n. Hence, these values are best treated as pre-computed, tabulated parameters; we will comment further below.

Equation (2.3) means that the areas of the $n-1$ rectangles with corners $(0, y_i)$, (x_i, y_i), (x_i, y_{i-1}) and $(0, y_{i-1})$ (denoted by R_i), $i = 1,\dots,n-1$, are all equal to v, just as the area of the last rectangle with corners $(0,0)$, $(x_n, 0)$, (x_n, y_n) and $(0, y_n)$ together with the area below the graph f on $[x_n,\infty[$. This surface will be denoted by R_n. Moreover, the area below the graph of f is contained in the surface $\bigcup_{i=1}^n R_i$ composed of all the surfaces described above. Furthermore, we assume that we have a specialized algorithm for sampling from the tail distribution $X \sim f$ conditioned on $X > x_n$. See also Figure 2.2.

Algorithm 2.14 (Ziggurat algorithm)**.** *Goal: Sample a random variable $X \sim f$.*

1. *Generate i uniform in $\{1,\dots,n\}$.*
2. *If $i = n$, go to (6).*
3. *Generate $U_1 \sim \mathcal{U}(0,1)$ and set $x := U_1 x_i$.*
4. *If $x < x_{i-1}$ return x.*
5. *Otherwise, generate $U_2 \sim \mathcal{U}(0,1)$ and set $y := y_i + U_2(y_{i-1} - y_i)$. If $y \leqslant f(x)$ return x; else go back to (1).*
6. *Generate $U_1 \sim \mathcal{U}(0,1)$ and set $x := vU_1/y_n$.*

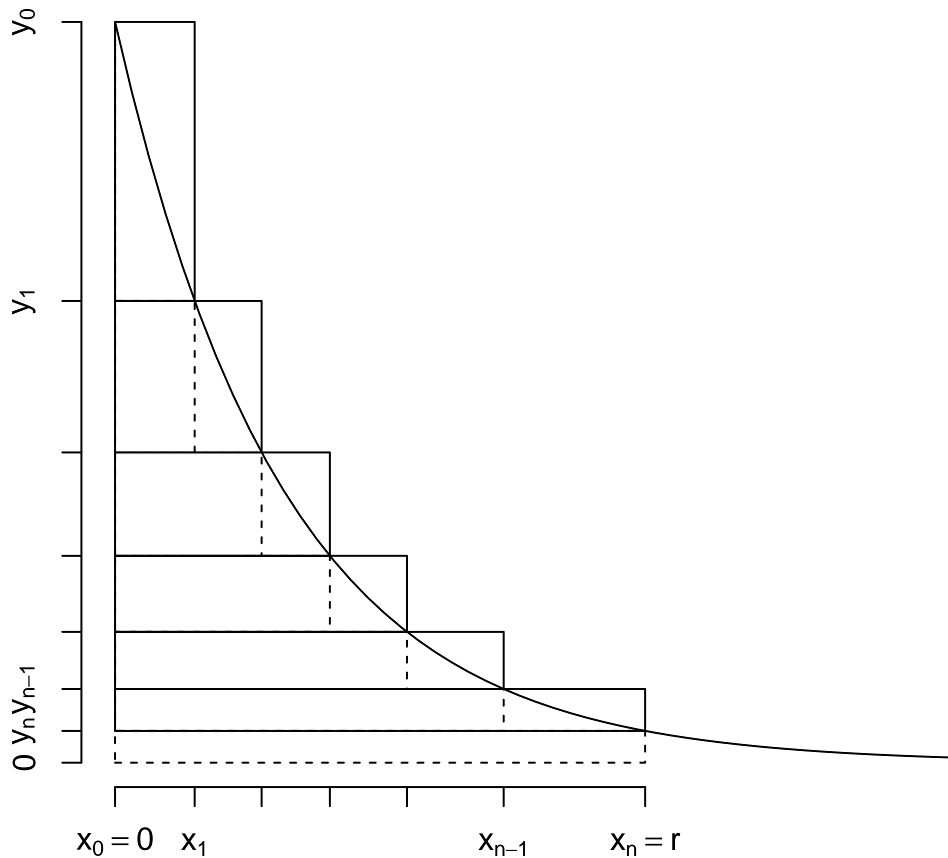

Figure 2.2: Ziggurat algorithm

7. If $x < x_n$, return x. Otherwise, return a sample from the tail distribution $X|X > x_n$.

Remark 2.15. Most of the time the algorithm stops in step (4), in which case we save one uniform random number generation and do not even need to evaluate f once. Moreover, it is obvious how to extend the algorithm to a symmetric or uni-modal distribution.

We round up this discussion with two examples, namely the Ziggurat algorithm for the exponential and the standard normal distributions. In both cases, the Ziggurat algorithm is highly competitive in speed.

Example 2.16. For the exponential distribution $Exp(1)$, we can use as tail sample $x_n - \log U$ for $U \sim \mathcal{U}(0, 1)$. For $n = 255$ (a typical value), a possible choice of x_n is $7.697\ldots$ implying $v = 0.0039\ldots$, which results in an efficiency of 98.9%, i.e., the probability of needing only one iteration of the algorithm to produce one sample of the target distribution is 98.9%.

Example 2.17. For the standard normal distribution $\mathcal{N}(0, 1)$, for $n = 255$, a possible choice of x_n

is $3.65\ldots$ implying $v = 0.0049\ldots$, which results in an efficiency of 99.33%. [9] suggest the following algorithm for sampling from the tail distribution:

1. Generate $U_1, U_2 \sim \mathcal{U}(0,1)$.
2. Set $x := -\log(U_1)/x_n$, $y := -\log U_2$.
3. If $2y > x^2$, return $x + x_n$, else go back to (1).

We conclude this section by presenting two methods designed specifically for generating standard normal random numbers. The *Box–Muller method* and the *polar method* are probably two of the simplest such methods, although not the most efficient ones. A comprehensive list of random number generators specifically available for Gaussian random numbers is available in the survey article by Thomas et al. [17].

Algorithm 2.18 (Box–Muller method)**.** *1. Generate two independent uniform randoms numbers U_1, U_2;*

2. Set $\theta = 2\pi U_2$, $\rho = \sqrt{-2\log(U_1)}$;

3. Return two independent standard normals $X_1 = \rho\cos(\theta)$, $X_2 = \rho\sin(\theta)$.

Algorithm 2.19 (Polar method)**.** *1. Generate two independent uniform random numbers U_1, U_2 from the interval $]-1, 1[$;*

2. Set $S = U_1^2 + U_2^2$;

3. If $S < 1$, return the independent standard normals

$$Y_1 = U_1\sqrt{\frac{-2\ln(S)}{S}} \quad \text{and} \quad Y_2 = U_2\sqrt{\frac{-2\ln(S)}{S}};$$

else, return to 1.

The polar method is more efficient than the Box–Muller algorithm, because it avoids the evaluation of the computationally expensive trigonometric functions.

Remark 2.20. In order to generate samples from the general, d-dimensional normal distribution $\mathcal{N}(\mu, \Sigma)$, we first generate a d-dimensional vector of independent standard normal variates $X = (X_1, \ldots, X_d)$ using, for instance, the Box-Muller method. Then we obtain the sample from the general normal distribution by

$$\mu + AX,$$

where A satisfies $\Sigma = AA^T$. Note that A can be obtained from Σ by Cholesky factorization.

Exercise 2.1. Explain why c in (2.2) can only be greater than or equal to 1. What does $c = 1$ imply?

Exercise 2.2. Provide a method for generating double exponential random variables using only *one* uniform random number per output. Moreover, justify the bound c in Example 2.13.

Exercise 2.3. Show that (X_1, X_2) generated by the Box–Muller method have the two-dimensional standard normal distribution.
Hint: Show that the density of the two-dimensional uniform variate (U_1, U_2) is transformed to the density of the two-dimensional standard normal distribution.

Exercise 2.4. Show that (Y_1, Y_2) generated by the polar method have the two-dimensional standard normal distribution.

Exercise 2.5. Implement the different methods for generating Gaussian random numbers and compare their efficiency.

2.2 Monte Carlo simulation

The Monte Carlo simulation method is one of the most important numerical methods available. It was developed by giants of mathematics and physics like J. von Neumann, E. Teller, S. Ulam and N. Metropolis during the development of the H-bomb. A short account of the origins of Monte Carlo simulation can be found in Metropolis [11]. Today, it is widely used in fields like statistical mechanics, particle physics, computational chemistry, molecular dynamics, computational biology and, of course, computational finance! An overview of the mathematics behind the Monte Carlo method is available, for instance, in the survey paper of Caflisch [1] or, as usual, in Glasserman [5].

The Monte Carlo method

As we have already discussed in the introduction, we want to compute the quantity

$$I[f;X] := \mathbb{E}\left[f(X)\right], \tag{2.4}$$

assuming only that $f(X)$ is integrable, *i.e.*, $I[|f|;X] < \infty$, and that we can actually sample from the distribution of X. Taking a sequence $X_1, X_2, \ldots$ of independent realizations of X, the law of large numbers implies that

$$I[f;X] = \lim_{M\to\infty} \frac{1}{M}\sum_{i=1}^{M} f(X_i), \quad \mathbb{P}-\text{a.s.} \tag{2.5}$$

However, in numerics we are usually not quite satisfied with a mere convergence statement like in (2.5). Indeed, we would like to be able to control the error, *i.e.* we would like to have an error estimate or bound, and we would also like to know how fast the error goes to zero if we increase M. Before continuing the discussion, let us formally introduce the Monte Carlo integration error ϵ_M by

$$\epsilon_M = \epsilon_M(f;X) := I[f;X] - I_M[f;X], \quad \text{where} \quad I_M[f;X] := \frac{1}{M}\sum_{i=1}^{M} f(X_i) \tag{2.6}$$

is the estimate based on the first M samples. Note that $I_M[f;X]$ is an *unbiased* estimate for $I[f;X]$ in the statistical sense, *i.e.* $\mathbb{E}\left[I_M[f;X]\right] = I[f;X]$, implying $\mathbb{E}\left[\epsilon_M\right] = 0$. Let us also introduce the *mean square error* and its square root, the error in L^2, via

$$\mathrm{MSE}[I_M] = \mathbb{E}\left[\epsilon_M(f;X)^2\right] \quad \text{and} \quad \mathrm{RMSE}[I_M] = \mathbb{E}\left[\epsilon_M(f;X)^2\right]^{1/2}. \tag{2.7}$$

The *central limit theorem* immediately implies both an error bound and a convergence rate provided that $f(X)$ is square integrable.

Proposition 2.21. *Let $\sigma = \sigma(f;X) < \infty$ denote the standard deviation of the random variable $f(X)$. Then the root mean square error satisfies*

$$\mathbb{E}\left[\epsilon_M(f;X)^2\right]^{1/2} = \frac{\sigma}{\sqrt{M}}.$$

Moreover, $\sqrt{M}\epsilon_M(f;X)$ is asymptotically normally distributed with standard deviation $\sigma(f;X)$. That is, for any constants $a < b \in \mathbb{R}$ we have

$$\lim_{M\to\infty} \mathbb{P}\left(\frac{\sigma a}{\sqrt{M}} < \epsilon_M < \frac{\sigma b}{\sqrt{M}}\right) = \Phi(b) - \Phi(a),$$

where Φ denotes the cumulative distribution function of the standard normal random variable.

Proof. Using the independence of the X_i's and the fact that $I_M[f;X]$ is an unbiased estimator of $I[f;X]$, we get

$$\mathbb{E}\left[\epsilon_M^2\right] = \mathrm{var}\left(\frac{1}{M}\sum_{i=1}^{M} f(X_i)\right) = \frac{1}{M^2}\sum_{i=1}^{M}\mathrm{var}(f(X_i)) = \frac{M\,\mathrm{var}(f(X_1))}{M^2} = \frac{\sigma^2}{M}.$$

In addition, from the central limit theorem we know that

$$\frac{\sum_{i=1}^{M} f(X_i) - M \cdot I[f;X]}{\sigma\sqrt{M}} \xrightarrow[M\to\infty]{} \mathcal{N}(0,1)$$

which yields the asymptotic normality of the error. □

Remark 2.22. Proposition 2.21 has two important implications:

1. The error is *probabilistic*: there is no deterministic error bound. In other words, for a particular simulation and a given sample size M, the error of the simulation can be arbitrarily large. However, large errors only occur with probabilities decreasing in M.
2. The "typical" error, *e.g.* the root mean square error $\sqrt{E\left[\epsilon_M^2\right]}$, decreases to zero like $1/\sqrt{M}$. In other words, if we want to increase the accuracy of the result tenfold, *i.e.* if we want to obtain one more significant digit, then we have to increase the sample size M by a factor $10^2 = 100$. We thus say that the Monte Carlo method *converges with rate* $1/2$.

Let us now discuss the merits of Monte Carlo simulation. We assume, for simplicity, that X is a d-dimensional uniform random variable, *i.e.*,

$$I[f] \coloneqq I[f;U] = \int_{[0,1]^d} f(x)\mathrm{d}x.$$

Observe that the dimension of the space did not enter into our discussion of the convergence rate and of the error bounds at all. This is remarkable if we compare the Monte Carlo method to traditional methods for numerical integration. Those methods are usually based on a grid $0 \leqslant x_1 < x_2 < \cdots < x_N \leqslant 1$ of arbitrary length N. The corresponding d-dimensional grid is simply given by $\{x_1,\ldots,x_N\}^d$, a set of size $n \coloneqq N^d$. The function f is evaluated on the grid points and an approximation of the integral is computed based on interpolation of the function between grid points by suitable functions (*e.g.* piecewise polynomials), whose integral can be explicitly computed. Given a numerical integration method of order k, the error is then proportional to $(1/N)^k$. However, we have to evaluate the function on n points, implying that the total computational work is proportional to n rather then N. Therefore, the accuracy in terms of the complexity n, *i.e.* the ratio of the error relative to the computational work, behaves like $n^{-k/d}$. Thus, the rate of convergence in terms of the computational cost is only k/d, which rapidly decreases in the dimension d. This phenomenon is known as the *curse of dimensionality*: methods which are very well suited in low dimensions, deteriorate very fast in higher dimensions.

The curse of dimensionality is the main reason for the popularity of the Monte Carlo method. As we will see later, in financial applications the dimension of the state space can easily be in the order of 100 (or much higher), which already makes traditional numerical integration methods completely unfeasible. In other applications, like molecular dynamics, the dimension of the state space might be in the magnitude of 10^{12}!

Error control and confidence intervals

Next, we discuss how to control the error of the Monte Carlo method taking its random nature into account. The question here is, how do we have to choose M, the only parameter available, such that the probability of an error larger than a given tolerance level $\varepsilon > 0$ is smaller than a given $\delta > 0$, symbolically

$$\mathbb{P}\big(\,|\epsilon_M(f;X)| > \varepsilon\big) < \delta.$$

Fortunately, this question is already almost answered in Proposition 2.21. Indeed, it implies that

$$\mathbb{P}\left(|\epsilon_M| > \varepsilon\right) = 1 - \mathbb{P}\left(-\frac{\sigma\tilde{\varepsilon}}{\sqrt{M}} < \epsilon_M < \frac{\sigma\tilde{\varepsilon}}{\sqrt{M}}\right) \sim 1 - \Phi(\tilde{\varepsilon}) + \Phi(-\tilde{\varepsilon}) = 2 - 2\Phi(\tilde{\varepsilon}),$$

where $\tilde{\varepsilon} = \sqrt{M}\varepsilon/\sigma$. Of course, the normalized Monte Carlo error is only asymptotically normal, which means the equality between the left and the right hand side of the above equation only holds for $M \to \infty$, which is signified by the "$\sim$"-symbol. Equating the right hand side with δ and solving for M yields

$$M = \left(\Phi^{-1}\left(\frac{2-\delta}{2}\right)\right)^2 \frac{\sigma^2}{\varepsilon^2}. \tag{2.8}$$

Thus, as we have already observed before, the number of samples depends on the tolerance like $1/\varepsilon^2$.

Remark 2.23. This analysis tacitly assumed that we know $\sigma = \sigma(f;X)$. Since we started the whole endeavor in order to compute the mean of $f(X)$, $I[f;X]$, it is, however, very unlikely that we already know the variance of $f(X)$. Therefore, in practice we will have to replace $\sigma(f;X)$ by a sample estimate. See Exercise 2.6 for a sample estimator of σ. (This is not unproblematic: what about the Monte Carlo error for the approximation of $\sigma(f;X)$?)

In addition, since the Monte Carlo estimator is a random variable, when computing expectations via this method it is not very helpful to report just the value $I_M[f;X]$. This estimator is a function of the sample size M and we do not know how accurate the estimation is unless we also have information about the sample size. Therefore, it is more meaningful to report the estimator and some *confidence interval*.

Definition 2.24. Let Z be a random variable and consider some level $\alpha \in (0,1)$. The $1-\alpha$-level *confidence interval* is defined by

$$[-z_{1-\frac{\alpha}{2}}, z_{1-\frac{\alpha}{2}}]$$

such that the *critical number* $z_{1-\frac{\alpha}{2}}$ satisfies:

$$\mathbb{P}\Big(|Z| \leqslant z_{1-\frac{\alpha}{2}}\Big) = 1-\alpha. \tag{2.9}$$

The critical number $z_{1-\frac{\alpha}{2}}$ for a given level $1-\alpha$ can be computed from the inverse cumulative distribution function. Consider, for example, the normal distribution; then we get that

$$z_{1-\frac{\alpha}{2}} = \Phi^{-1}\Big(1-\frac{\alpha}{2}\Big).$$

In particular, using the inverse cdf of the normal distribution, we get that for $\alpha = 5\%$ the critical number equals 1.96, while for $\alpha = 1\%$ it equals 2.58.

Now, we can use the asymptotic normality of the Monte Carlo error ϵ_M to derive confidence intervals for $I_M[f;X]$. Indeed, using Proposition 2.21 and denoting $\epsilon_M = I - I_M$, we have

$$\begin{aligned} 1-\alpha &\approx \mathbb{P}\left(-\frac{\sigma z_{1-\frac{\alpha}{2}}}{\sqrt{M}} \leqslant \epsilon_M \leqslant \frac{\sigma z_{1-\frac{\alpha}{2}}}{\sqrt{M}}\right) \\ &= \mathbb{P}\left(I_M - \frac{\sigma z_{1-\frac{\alpha}{2}}}{\sqrt{M}} \leqslant I \leqslant I_M + \frac{\sigma z_{1-\frac{\alpha}{2}}}{\sqrt{M}}\right). \end{aligned}$$

Thus, the $1-\alpha$-level confidence interval for $I = I[f;X]$ is

$$\mathrm{CI}_\alpha[I_M] := \left[I_M - \frac{\sigma z_{1-\frac{\alpha}{2}}}{\sqrt{M}}, I_M + \frac{\sigma z_{1-\frac{\alpha}{2}}}{\sqrt{M}}\right]. \tag{2.10}$$

Example 2.25. We consider the Black–Scholes–Samuelson model, where the dynamics of the underlying asset have the form

$$\mathrm{d}S_t = rS_t\mathrm{d}t + \sigma S_t\mathrm{d}W_t, \quad S_0 = s \in \mathbb{R}_+, \tag{2.11}$$

where W is a standard Brownian motion, while we assume we are already under the martingale measure. We want to compute the price of a European call option with payoff function

$$f(S_T) = (S_T - K)^+, \tag{2.12}$$

together with the 95% and 99% confidence intervals. Algorithm 1 contains pseudo-code for the Black–Scholes formula for a European call, while Algorithm 2 contains pseudo-code for the computation of the Monte Carlo price and the RMSE. An outcome of this example is shown in Figure 2.3, where the Monte Carlo price for different sample sizes together with the corresponding confidence intervals are plotted together with the Black–Scholes price. One should notice how the Monte Carlo price converges to the Black–Scholes price and how the confidence intervals shrink as the sample size M increases.

Algorithm 1 Pseudo-code for the Black–Scholes formula

1: input: S_0, K, T, r, σ
2: $d_1 \leftarrow (\log(S_0/K) + (r + \sigma^2/2) \cdot T)/(\sigma \cdot \sqrt{T})$
3: $d_2 \leftarrow (\log(S_0/K) + (r - \sigma^2/2) \cdot T)/(\sigma \cdot \sqrt{T})$
4: price $\leftarrow S_0 \cdot \Phi(d_1) - K \cdot \mathrm{e}^{-r \cdot T} \cdot \Phi(d_2)$
5: output: price

Algorithm 2 Pseudo-code for MC simulation in the Black–Scholes model

1: input: S_0, K, T, r, σ, M
2: $W \leftarrow M$ independent samples from the standard normal distribution
3: $S \leftarrow S0 \cdot \exp(\sigma \cdot \sqrt{T} \cdot W + (r - \sigma^2/2) \cdot T)$
4: $C \leftarrow \exp(-r \cdot T) \cdot \max\{S - K, 0\}$
5: price $\leftarrow \text{sum}\{C\}/M$
6: varest $\leftarrow \text{sum}\{(\text{price} - C)^2\}/(M - 1)$
7: rmse $\leftarrow \sqrt{\text{varest}/M}$
8: output: price, rmse

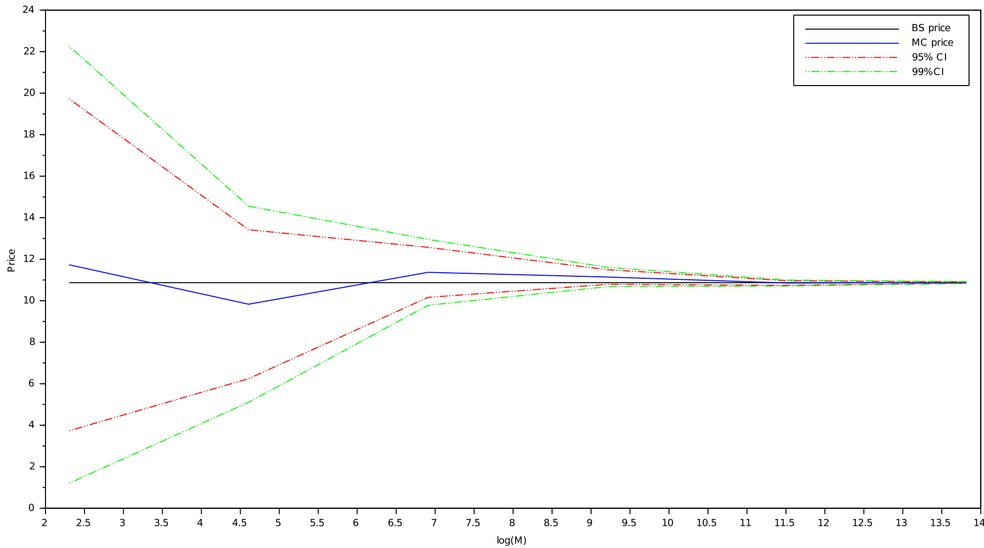

Figure 2.3: Convergence of the Monte Carlo price of a European call option to the Black–Scholes price as a function of the sample size M, together with the 95% and 99% confidence intervals.

Variance reduction

Although there are no obvious handles on how to increase the convergence rate in Proposition 2.21, we might be able to improve the constant factor in the RMSE by reducing the variance $\sigma(f;X)^2 = \mathrm{var}(f(X))$. The idea is to obtain, in a systematic way, random variables Y and functions g such that $\mathbb{E}[g(Y)] = \mathbb{E}[f(X)]$, but with smaller variance $\mathrm{var}(g(Y)) < \mathrm{var}(f(X))$. Inserting $\sigma(g;Y) = \sqrt{\mathrm{var}(g(Y))}$ into (2.8) shows that such an approach will decrease the computational work—proportional to the number of trajectories—provided that the generation of samples from $g(Y)$ is not prohibitively more expensive than the generation of samples from $f(X)$. This leads then to a faster numerical scheme, since the same error can be achieved with fewer samples.

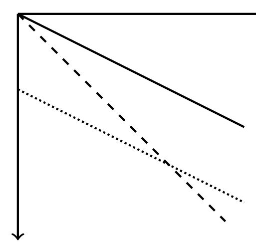

Figure 2.4: Improved convergence rate vs. improved constant.

A pictorial representation of the potential improvement is available in Figure 2.4, where the log-error (y-axis) is plotted against the log-number of samples (x-axis). The convergence rate of the Monte Carlo method is depicted with the solid line with slope $\frac{1}{2}$. An improved convergence rate would lead to a line with different slope, *e.g.* the dashed line with slope 1 in the figure above. On the other hand, an improved constant leads to a parallel shift of the line with slope $\frac{1}{2}$, see the dotted line in the figure above.

Antithetic variates

Let us start with the following simple observation: If U has the uniform distribution, then the same is true for $1-U$. Similarly, if B has the d-dimensional normal distribution, then so does $-B$. Therefore, these transformations do not change the expected value $\mathbb{E}[f(X)]$, if $X = U$ or $X = B$.[7] In general, assume there exists a (simple) transformation $\widetilde{X}$ having the same law as X, such that a realization of $\widetilde{X}$ can be computed from a realization of X by a deterministic transformation. Define the *antithetic variates* Monte Carlo estimate by

$$I_M^A[f;X] = \frac{1}{M}\sum_{i=1}^{M} \frac{f(X_i)+f(\widetilde{X}_i)}{2}. \tag{2.13}$$

Since $\mathbb{E}[(f(X_i)+f(\widetilde{X}_i))/2] = \mathbb{E}[f(X)]$, (2.13) is another unbiased estimator for $I[f;X]$. If we assume that the actual simulation of $(f(X_i)+f(\widetilde{X}_i))/2$ takes at most two times the computing time as the simulation of $f(X_i)$, then the computing time necessary for the computation of the estimate $I_M^A[f;X]$ does not exceed the computing time for the computation of $I_{2M}[f;X]$.[8] Then, the application of antithetic variates makes sense if the mean square error of $I_M^A[f;X]$ is smaller than the MSE of $I_{2M}[f;X]$, *i.e.* if

$$\frac{\mathrm{var}\left(\frac{f(X_i)+f(\widetilde{X}_i)}{2}\right)}{M} < \frac{\mathrm{var}(f(X_i))}{2M}.$$

This is equivalent to $\mathrm{var}(f(X_i)+f(\widetilde{X}_i)) < 2\,\mathrm{var}(f(X_i))$. Since $\mathrm{var}(f(X_i)+f(\widetilde{X}_i)) = 2\,\mathrm{var}(f(X_i)) + 2\,\mathrm{cov}(f(X_i), f(\widetilde{X}_i))$, antithetic variates can speed up a Monte Carlo simulation if and only if

$$\mathrm{cov}\left(f(X), f(\widetilde{X})\right) < 0. \tag{2.14}$$

In other words, the antithetic variates Monte Carlo method should be used when the negative dependence between the input variables X and $\widetilde{X}$ (think of U and $1-U$ or B and $-B$) produces

[7] Since many random number generators for non-uniform distributions are based on uniform ones, we can often view our integration problem as being of this type.

[8] Since we only need to sample one random number X_i and obtain $\widetilde{X}_i$ by a simple deterministic transformation, in many situations it is much faster to compute $(f(X_i)+f(\widetilde{X}_i))/2$ than to compute two realizations of $f(X_i)$.

also negative dependence between the output variables $f(X)$ and $f(\widetilde{X})$. A simple, sufficient condition for the latter is the monotonicity of the function f that maps inputs to outputs.

The calculations above yield also the following decomposition for the MSE of the antithetic variates Monte Carlo method:

$$\text{MSE}\big[I_M^A\big] = \text{MSE}\big[I_{2M}\big] + \frac{\text{cov}\big(f(X), f(\widetilde{X})\big)}{2M}. \tag{2.15}$$

In other words, the improvement over the standard Monte Carlo method, if any, comes in the form of an *additive* factor (which obviously tends to zero as $M \to \infty$). The larger the negative dependence between $f(X)$ and $f(\widetilde{X})$, the larger this factor as well, for fixed M.

Remark 2.26. Exercise 2.9 asks the reader to justify the application of the antithetic variates Monte Carlo method for pricing a European call option in the Black–Scholes model, theoretically and by computing the sample covariance. Once the sample estimator for the covariance is coded, one could notice that the speed-up factor depends on the strike K (all other parameters equal), and is larger for deep-in-the-money options, *i.e.* for $K \to 0$.

Control variates

Assume there exists a random variable Y and a functional g such that we know the exact value of $I[g;Y] = \mathbb{E}[g(Y)]$. (Note that we allow for $Y = X$.) Then obviously

$$I[f;X] = \mathbb{E}\left[f(X) - \lambda(g(Y) - I[g;Y])\right],$$

for any deterministic parameter λ. Thus, a Monte Carlo estimate for $I[f;X]$ is given by

$$I_M^{C,\lambda}[f;X] := \frac{1}{M}\sum_{i=1}^{M}\Big(f(X_i) - \lambda g(Y_i)\Big) + \lambda I[g;Y], \tag{2.16}$$

where (X_i, Y_i) are independent realizations of (X, Y). Similar to the situation with antithetic variates, we may assume that the simulation of $I_M^{C,\lambda}[f]$ takes at most c times the computing time of the simulation of $I_M[f]$, where $c > 1$ often is quite small, especially if $X = Y$. We are going to choose the parameter λ such that $\text{var}(f(X) - \lambda g(Y))$ is minimized. A simple calculation yields that

$$\text{var}(f(X) - \lambda g(Y)) = \text{var}(f(X)) - 2\lambda\,\text{cov}(f(X), g(Y)) + \lambda^2\,\text{var}(g(Y)),$$

which is minimized by choosing λ to be equal to

$$\lambda^\star = \frac{\text{cov}(f(X), g(Y))}{\text{var}(g(Y))}. \tag{2.17}$$

Applying Proposition 2.21, we get that the mean square error for the standard and the control variates Monte Carlo simulations compare as follows:

$$\text{MSE}\big[I_M^{C,\lambda^\star}\big] = \frac{\text{var}(f(X))}{M}(1-\rho^2) \leqslant \frac{\text{var}(f(X))}{M} = \text{MSE}\big[I_M\big], \tag{2.18}$$

where ρ denotes the correlation coefficient between $f(X)$ and $g(Y)$. In other words, the improvement of the control variates Monte Carlo simulation over the standard Monte Carlo method comes in the form of a *multiplicative* factor. Assuming that the computational work per realization is c times higher using control variates, (2.8) implies that the control variates technique is $1/(c(1-\rho^2))$-times faster than standard Monte Carlo. In particular, the improvement in speed from the use of control variates is larger as the correlation between $f(X)$ and $g(Y)$ becomes higher. If, for example, $\rho = 0.8$ and $c = 2$ the speed-up factor equals 1.38, while if $\rho = 0.95$ the speed-up factor equals 5.

Remark 2.27. We can determine the optimal factor $\lambda^\star$ only if we know $\text{cov}(f(X), g(Y))$ and $\text{var}(g(Y))$. If we are not in this highly unusual situation, we can use sample estimates instead—see Exercise 2.6—obtained by (standard) Monte Carlo simulations with a smaller sample size.

A natural question now is how to find, or construct, good control variates. There does not exist a general answer since these are typically specified by the problem at hand. However, in option pricing the underlying asset provides a virtually universal source of control variates, because

$$e^{-rT}\mathbb{E}[S_t] = S_0 \tag{2.19}$$

for every $t \geqslant 0$, assuming that $\mathbb{E}$ denotes the expectation with respect to a martingale measure. Moreover, simple options that admit a closed-form solution can be used as control variates for the pricing of more complex derivatives, see *e.g.* Exercise ... with geometric and arithmetic Asian options. In additions, simple models can be used as control variates for option pricing in more advanced models; for example, the Black–Scholes model can serve as control variate for stochastic volatility models.

Example 2.28. Assume we want to compute the price of an option with payoff $f(S_T)$ and we are given a sample $S_T^1, \dots, S_T^M$ from the law of S_T. The control variates Monte Carlo estimator takes the form

$$I_M^{C,\lambda^\star}[f; S_T] = \frac{1}{M}\sum_{i=1}^M \left\{ f(S_T^i) - \lambda^\star S_T^i \right\} + \lambda^\star S_0, \tag{2.20}$$

where $\lambda^\star$ can be also replaced by the sample estimator $\lambda_M^\star$. The interest rate is set to zero, for simplicity. If $f(S_T) = (S_T - K)^+$, *i.e.* we are pricing a call option, then

$$\lambda^\star = \frac{\operatorname{cov}\big((S_T - K)^+, S_T\big)}{\operatorname{var}(S_T)}, \tag{2.21}$$

and the efficiency of the control variate depends, essentially, on the strike K (all other parameters equal). In particular, for $K = 0$ we obviously have perfect correlation and the method is very effective. On the other hand, for deep out-of-the-money options (*i.e.* for large K) the correlation becomes quite low and the effectiveness of the method deteriorates.

Stratified sampling

The main principle of stratified sampling is to partition the sample space into disjoint subsets, called *strata*, and to constrain the number of samples selected from each stratum. Let $A_1, \dots, A_L$ be disjoint subsets of $\mathbb{R}^d$ such that $\mathbb{P}(X \in \cup_l A_l) = 1$. Then, using the law of total probability, we can estimate $f(X)$ as follows

$$\mathbb{E}[f(X)] = \sum_{l=1}^L \mathbb{E}[f(X)|X \in A_l]\mathbb{P}(X \in A_l) = \sum_{l=1}^L p_l \mathbb{E}[f(X)|X \in A_l], \tag{2.22}$$

where $p_l = \mathbb{P}(X \in A_l)$. In the standard Monte Carlo method, we generate $X_1, \dots, X_M$ which are independent and distributed identically to X, and the fraction of samples that belong to each stratum A_l is in general not equal to p_l, although it converges to p_l as $M \to \infty$. In contrast, in stratified sampling we preselect what fraction of samples should belong to each stratum, and every sample drawn from A_l has the distribution of X *conditional on* $X \in A_l$.

Let M denote the total size of the sample. For every $l = 1, \dots, L$, let $q_l = \frac{M_l}{M}$ denote the fraction of observations from the stratum A_l, and X_{lk}, $k = 1, \dots, M_l$, be i.i.d. realizations from the distribution of X conditional on $X \in A_l$. An unbiased estimator for the expectation in the RHS of (2.22) is provided by the sample average, *i.e.* by $\frac{1}{M_l}\sum_{k=1}^{M_l} f(X_{lk})$. Therefore, the *stratified sampling estimator* takes the form

$$I_M^{ST}[f; X] = \sum_{l=1}^L p_l \frac{1}{M_l}\sum_{k=1}^{M_l} f(X_{lk}) = \frac{1}{M}\sum_{l=1}^L \frac{p_l}{q_l}\sum_{k=1}^{M_l} f(X_{lk}). \tag{2.23}$$

Remark 2.29. The strata can also depend on another variable Z, called the *stratifying variable*, which is possibly dependent on X. In that case, the estimator has the same form, *i.e.*

$$I_M^{ST}[f;X] = \frac{1}{M}\sum_{l=1}^{L}\frac{p_l}{q_l}\sum_{k=1}^{M_l} f(X_{lk}), \tag{2.24}$$

where now $p_l = \mathbb{P}(Z \in A_l)$ and $(X_{lk})_k$ are i.i.d. realizations from the distribution of X conditional on $Z \in A_l$. We will use this more general formulation from now on.

Therefore, in order to effectively implement a stratified sampling estimator we should select and optimize the following variables: the stratification variable Z, the strata $A_1, \dots, A_L$ and the allocations $M_1, \dots, M_L$. Moreover, we should also know how to efficiently sample from the law of (X, Z) conditional $Z \in A_l$.

Let us now compare the variance of the stratified sampling estimator with the variance of the standard Monte Carlo estimator. We will use the following notation:

$$\mu_l = \mathbb{E}[f(X_{lk})] = \mathbb{E}[f(X)|Z \in A_l] \quad \text{and} \quad \sigma_l^2 = \text{var}[f(X_{lk})] = \text{var}[f(X)|Z \in A_l], \tag{2.25}$$

and then the variance of the stratified sampling estimator, using the *proportional allocation* $q_l = p_l$, is provided by

$$\text{var}\left(I_M^{ST}\right) = \frac{1}{M}\sum_{l=1}^{L} p_l \sigma_l^2. \tag{2.26}$$

On the other hand, the variance of the standard Monte Carlo estimator equals $\text{var}(I_M) = \text{var}(f(X))/M$, where

$$\begin{aligned}
\text{var}(f(X)) &= \mathbb{E}[f(X)^2] - \mathbb{E}[f(X)]^2 \\
&= \sum_{l=1}^{L} p_l \mathbb{E}[f(X)^2|Z \in A_l] - \left(\sum_{l=1}^{L} p_l \mathbb{E}[f(X)|Z \in A_l]\right)^2 \\
&= \sum_{l=1}^{L} p_l(\sigma_l^2 + \mu_l^2) - \left(\sum_{l=1}^{L} p_l \mu_l\right)^2. \qquad (2.27)
\end{aligned}$$

Therefore, the MSE of the stratified sampling Monte Carlo estimator admits the following decomposition:

$$\text{MSE}\big[I_M^{ST}\big] = \text{MSE}\big[I_M\big] + \frac{1}{M}\sum_{l=1}^{L} p_l \mu_l^2 - \frac{1}{M}\left(\sum_{l=1}^{L} p_l \mu_l\right)^2, \tag{2.28}$$

therefore any potential improvement over the standard Monte Carlo method comes in the form of an additive factor again. Now, Jensen's inequality yields that

$$\sum_{l=1}^{L} p_l \mu_l^2 \geqslant \left(\sum_{l=1}^{L} p_l \mu_l\right)^2,$$

therefore stratified sampling Monte Carlo with proportional allocation leads to a reduction of the variance of the estimator.

One can achieve a further reduction of the variance by optimizing the allocations, *i.e.* by selecting the fractions q_l such that the variance of the estimator is minimized. The variance of the stratified sampling estimator in general has the form

$$\text{var}\left(I_M^{ST}\right) = \frac{1}{M}\sum_{l=1}^{L}\frac{p_l^2}{q_l}\sigma_l^2,$$

and minimizing this quantity subject to the constraints $q_l \in (0,1)$ and $\sum_l q_l = 1$ leads to the optimal allocation provided by

$$q_l^\star = \frac{p_l \sigma_l}{\sum_k p_k \sigma_k}.$$

The variance of the estimator with the optimal allocation equals then

$$\operatorname{var}\left(I_M^{ST,\star}\right) = \frac{1}{M}\left(\sum_{l=1}^{L} p_l \sigma_l\right)^2.$$

Using Jensen's inequality once again and comparing with (2.26) we observe that optimizing the allocations leads to a further reduction of the variance.

Remark 2.30. Similar to other methods, the variances σ_l are typically not known explicitly. One could then use sample estimators with a smaller sample size to compute $q_l^\star$ and then use the estimated optimal allocations in a second simulation run.

Importance sampling

Importance sampling is related to the acceptance-rejection method and also to Girsanov's theorem (or changes of measures). The idea is to sample more often in regions where the variance is higher, thus increasing the sampling efficiency. Assume that the underlying random variable X has a density p (on $\mathbb{R}^d$). Moreover, let q be another probability density. Then we can obviously write

$$I[f;X] = \int_{\mathbb{R}^d} f(x)p(x)dx = \int_{\mathbb{R}^d} f(x)\frac{p(x)}{q(x)}q(x)dx = E\left[f(Y)\frac{p(Y)}{q(Y)}\right] = I\left[f\frac{p}{q};Y\right],$$

where Y is a d-dimensional random variable with density q. The quantity p/q is called the likelihood ratio or the Radon–Nikodym derivative. Thus, a Monte Carlo estimate for $I[f]$ is given by

$$I_M^{IS}[f;X] = \frac{1}{M}\sum_{i=1}^{M} f(Y_i)\frac{p(Y_i)}{q(Y_i)} = I_M\left[f\frac{p}{q};Y\right]. \tag{2.29}$$

As usual, a possible speed up is governed by the variance of $f(Y)\frac{p(Y)}{q(Y)}$, which is determined by

$$\operatorname{var}\left(f(Y)\frac{p(Y)}{q(Y)}\right) + I[f;X]^2 = E\left[\left(f(Y)\frac{p(Y)}{q(Y)}\right)^2\right] = E\left[f(X)^2\frac{p(X)}{q(X)}\right]. \tag{2.30}$$

So how do we have to choose q? Assume for a moment that $f \geqslant 0$ itself. Take q proportional to $f \cdot p$. Then, the new estimator is based on the random variable

$$f(Y)\frac{p(Y)}{q(Y)} \equiv 1,$$

thus, the variance is zero! Of course, there is a catch: q needs to be normalized to one, therefore in order to actually construct q, we need to know the integral of $f \cdot p$, *i.e.*, we would need to know our quantity of interest $I[f]$. However, we can gain some intuition on how to construct a good importance sample estimate: we should choose q in such a way that $f \cdot p/q$ is almost flat.

There are several applications of importance sampling in financial and insurance mathematics, and the choice of the density q depends on the problem at hand. We will describe next an application of importance sampling to option pricing that highlights also the relation to Girsanov's theorem.

Example 2.31. Consider the setting of Example 2.25, that is, we are interested in pricing a European call option with payoff $f(x) = (x-K)^+$, where the underlying asset price follows the Black–Scholes model under a risk-neutral measure P, *i.e.*

$$\mathrm{d}S_t = rS_t\mathrm{d}t + \sigma S_t \mathrm{d}W_t,$$

where $S_0 = s \in \mathbb{R}_+$. Let $\theta \in \mathbb{R}$, consider the exponential martingale M with

$$M_t = \exp\left(\theta W_t - \frac{1}{2}\theta^2 t\right)$$

and define a measure P_θ, equivalent to P, via the Radon–Nikodym density

$$\frac{\mathrm{d}P_\theta}{\mathrm{d}P} = M_T. \tag{2.31}$$

Using Girsanov's theorem we know that $W^\theta = W - \theta\cdot$ is a P_θ-Brownian motion and the dynamics of S under P_θ are provided by

$$\mathrm{d}S_t = (r - \sigma\theta)S_t\mathrm{d}t + \sigma S_t \mathrm{d}W_t^\theta.$$

Now, the price of the option equals

$$I[f; S_T] = E_\theta[f(S_T)L_T] \tag{2.32}$$

where L is the P_θ-martingale

$$L_t = \exp\left(-\theta W_t + \frac{1}{2}\theta^2 t\right).$$

Therefore, the importance sampling estimator is provided by

$$I_M^{IS}[f; S_T] = \frac{1}{M}\sum_{i=1}^{M} f(S_T^i)L_T^i, \tag{2.33}$$

where (S_T^i, L_T^i) are independent realizations of (S_T, L_T) under the measure P_θ. The variance of the importance sampling estimator, using (2.30), equals

$$\operatorname{var}\left(I^{IS}\right) = E_\theta[f(S_T)^2 L_T^2] + I[f; S_T]^2. \tag{2.34}$$

Thus, if we want to minimize the variance of the importance sampling estimator we should select the density, *i.e.* the parameter θ, such that

$$E_\theta[f(S_T)^2 L_T^2] \tag{2.35}$$

is minimized over θ. Now, define the functions

$$g(x) = (S_0 \mathrm{e}^{\sigma x + (r - \sigma\theta - \frac{1}{2}\sigma^2)T} - K)^+ \quad \text{and} \quad G(x) = \log g(x),$$

and using that $W^\theta = W - \theta\cdot$, we get that

$$\begin{aligned} E_\theta[f(S_T)^2 L_T^2] &= E_\theta\left[\exp\left(2G(W_T^\theta) - 2\theta W_T + \theta^2 T\right)\right] \\ &= E_\theta\left[\exp\left(2G(W_T + \theta T) - 2\theta W_T + \theta^2 T\right)\right]. \end{aligned} \tag{2.36}$$

Let us turn to some heuristic arguments in order to find a candidate optimizer for the above expression. Ignore the dependence on time and consider a Taylor expansion of $G(W + \theta)$ around θ, which yields for the exponent

$$G(W + \theta) - \theta W + \frac{1}{2}\theta^2 \approx G(\theta) + G'(\theta)(W - \theta) - \theta W + \frac{1}{2}\theta^2 + \dots. \tag{2.37}$$

If we choose θ such that $G'(\theta) = \theta$ then the exponent will not depend on W (at least up to a first order approximation) and the variance of the importance sampling estimator is minimized.

Conclusions

Comparing the three methods of variance reduction presented here, we see that antithetic variates are the easiest to implement, but can only give a limited speed-up. On the other hand, both control variates and importance sampling can allow us to use very specific properties of the problem at hand. Therefore, the potential gain can be large (in theory, the variance can be reduced almost to zero). However, this also means that there is no general way to implement control variates or importance sampling.

Exercise 2.6. Show that an unbiased estimator of $\sigma^2(f;X)$ is

$$\sigma_M^2(f;X) = \frac{1}{M-1}\sum_{i=1}^{M}\Big(f(X_i) - I_M[f;X]\Big)^2 \tag{2.38}$$

and an unbiased estimator of $\operatorname{cov}(f(X), g(Y))$ is

$$\operatorname{cov}_M(f(X), g(Y)) = \frac{1}{M-1}\sum_{i=1}^{M}\Big(f(X_i) - I_M[f;X]\Big)\Big(g(Y_i) - I_M[g;Y]\Big). \tag{2.39}$$

Exercise 2.7. Compute the price of a European call option in the Black–Scholes model using Monte Carlo simulation, as well as the 95% and 99% confidence intervals. Study the convergence and the asymptotic normality of the error. Then, use (2.8) for a more systematic approach.

Exercise 2.8. Compute the expected value of $1/\sqrt{U}$ for a uniform random variable U using Monte Carlo simulation. Study the speed of convergence and whether the errors are still asymptotically normal.

Hint: This exercise shows that if we want to compute the expected value of an integrable random variable, which is not square integrable, the above analysis does not apply.

Exercise 2.9. Compute the price of a European call option in the Black–Scholes model using the antithetic variates Monte Carlo method. Justify why the method works

(i) numerically, by computing the sample covariance;

(ii) theoretically, by showing that the map from inputs to outputs is monotone.

Exercise 2.10. Compute the price of a European call option in the Black–Scholes model using the control variates Monte Carlo method where the underlying price is the control. Study how the efficiency of the method depends on the strike price and compare the convergence rates with Exercises 2.7 and 2.9.

Bibliography

[1] R. E. Caflisch. Monte Carlo and quasi-Monte Carlo methods. In *Acta numerica, 1998*, volume 7 of *Acta Numer.*, pages 1–49. Cambridge Univ. Press, Cambridge, 1998.

[2] R. Cont and P. Tankov. *Financial Modelling with Jump Processes*. Chapman & Hall/CRC, 2004.

[3] R. Cont and E. Voltchkova. A finite difference scheme for option pricing in jump diffusion and exponential Lévy models. *SIAM J. Numer. Anal.*, 43:1596–1626, 2005.

[4] L. Devroye. *Nonuniform Random Variate Generation*. Springer, 1986. Available online from http://cg.scs.carleton.ca/~luc/rnbookindex.html.

[5] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[6] D. E. Knuth. *The Art of Computer Programming. Vol. 2*. Addison-Wesley, second edition, 1981.

[7] P. L'Ecuyer. Uniform random number generation. *Ann. Oper. Res.*, 53:77–120, 1994.

[8] P. L'Ecuyer, B. Oreshkin, and R. Simard. Random numbers for prallel computers: Requirements and methods. Preprint, 2014.

[9] G. Marsaglia and W. W. Tsang. The Ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8), 2000.

[10] A.-M. Matache, P.-A. Nitsche, and C. Schwab. Wavelet Galerkin pricing of American options on Lévy driven assets. *Quant. Finance*, 5:403–424, 2005.

[11] N. Metropolis. The beginning of the Monte Carlo method. *Los Alamos Sci.*, 15, Special Issue: 125–130, 1987.

[12] NVIDIA. CUDA. http://www.nvidia.com/object/cuda_home_new.html. Accessed on August 24, 2014.

[13] B. Øksendal. *Stochastic Differential Equations*. Springer, 6th edition, 2003.

[14] P. E. Protter. *Stochastic Integration and Differential Equations*. Springer, 2nd edition, 2005.

[15] B. D. Ripley. *Stochastic Simulation*. John Wiley & Sons, 1987.

[16] R. Seydel. *Tools for Computational Finance*. Springer, 4th edition, 2009.

[17] D. B. Thomas, W. Luk, P. H. W. Leong, and J. D. Villasenor. Gaussian random number generators. *ACM Comput. Surv.*, 39(4), 2007.

[18] Wikipedia. Linear congruential generator — wikipedia, the free encyclopedia, 2010. [Online; accessed 22-March-2010].

[19] P. Wilmott. *Paul Wilmott on Quantitative Finance. 3 Vols*. John Wiley & Sons, 2nd edition, 2006.