CHAPTER 5
Monte-Carlo

In this chapter, we introduce the Monte-Carlo (MC) algorithm as a practical solution for pricing financial products in dynamic models and lay the groundwork for the serial implementation of Chapter 6 and the parallel implementation of Chapter 7. We establish the theoretical foundations of our implementation, but we don't cover all the details and facets of Monte-Carlo simulations. Monte-Carlo is more extensively covered in dedicated publications, the established references in finance being Glasserman's [62] and Jaeckel's [63]. Glasserman covers the many facets of financial simulations in deep detail while Jaeckel offers a practitioner's perspective, focused on implementation and including the most complete available presentation of Sobol's sequence.

5.1 THE MONTE-CARLO ALGORITHM

Introduction

The Monte-Carlo algorithm consists in the computer simulation of a number of independent outcomes of a random experiment with known probability distribution. It is applied in many scientific fields. In finance, the Monte-Carlo algorithm estimates the value of a financial product in contexts where faster methods (analytic, numerical integration, FDM) cannot be applied. With the notations of the previous

chapter, the value of a product is its expected (numeraire deflated) payoff under an appropriate probability measure :

where is the set of event dates, or timeline, and the scenario is the set of corresponding market samples. Each sample on the event date is a collection of market observations on the event date , including the numeraire.

The payoff function defines a financial product. We have seen that a model specifies a probability distribution for the scenario , although *dynamic* models specify it *indirectly*. Dynamic models specify:

1. The multidimensional dynamics of a state vector in dimension , called the Markov dimension of the model.[1] We restrict our presentation to diffusion models. In this case, the dynamics of the state vector, under the pricing measure, is a stochastic differential equation (SDE):

   where is a standard Brownian motion in dimension , called the number of factors in the model. In particular, the components of are independent, so the correlation structure of the model is contained in the matrix .
2. The mapping from state to market :

   where is the last fixing date for the product, called maturity. We have seen that the mapping can be *time-wise*:

   but this is not always the case. We have seen on page 175 that LGM does not define a time-wise mapping under the risk-neutral measure because the numeraire at time , , is not a function of the state on this date, but a functional of the past and present state . We have also seen that we can rewrite LGM under a different measure where the numeraire is a function of , so the mapping *is* time-wise. This is a general

property of many interest rate models: they typically don't admit a time-wise mapping under the risk-neutral measure, but they can be rewritten under a different measure, associated with a different numeraire, often a zero-coupon bond, where the mapping becomes time-wise, facilitating an efficient simulation.

The Monte-Carlo algorithm estimates the value by *brute force*:

1. Draw a large number of realizations of the scenario , also called paths , from its probability distribution under , specified by the model. Draw the realizations independently of one another.
2. Compute the payoff over each path, by evaluation of the product's payoff .
3. Approximate the value by the average payoff over the simulated paths.

Convergence and complexity

By construction, the s are IID with distribution the distribution of , therefore the MC estimate converges *almost surely* to by the Strong Law of Large Numbers. Furthermore, the Central Limit Theorem states that is asymptotically Gaussian with expectation and variance:

hence, a standard deviation inversely proportional to . This is a low convergence order. It means that in order to halve the error, we must quadruple the numbers of paths. To improve accuracy by a decimal takes a hundred times the number of paths, resulting in a hundred times slower simulation.

The complexity of MC is obviously linear in the number of paths. We will see that it is also linear in the number of time steps on the simulation timeline (number of time steps on the discrete path), the Markov dimension of the model (number of state variables updated on every time step), the dimension of the product (number of path-dependencies updated on event dates), and the dimension of the simulation (number of random variables drawn to simulate a path). Linear complexity in every direction, combined with an accuracy independent of dimension, are specific to Monte-Carlo and the reason why it is applicable in such a vast amount of situations. This is why MC is by far the most widely used valuation algorithm in modern finance, despite its slow convergence. The complexity of other algorithms, like FDM, grows more than linearly with dimension for a given accuracy, restricting their application to low-dimension contexts.

Path-wise simulations

The Monte-Carlo algorithm is natural, simple to understand, and practically implemented without major difficulty. It is often implemented path-wise, which means that paths are generated and evaluated one by one. An alternative implementation is step-wise, where all paths are simulated simultaneously, one time step at a time. A step-wise implementation has a number of benefits, including:

- Offers the opportunity to apply SIMD to vectorize calculations across paths; see page 9.
- Facilitates the computation of the empirical distribution of the observations in the scenario, and the modification of the paths to match theoretical distributions, a technique known as *control variate*. This improves the ability of a limited number of paths to accurately represent the distribution of the scenario, improving the accuracy of results.
  In addition, we have seen that the probability distribution of an observation on the event date is equivalent to the prices of European options of all strikes with maturity . It follows that a

special application of control variate in a step-wise Monte-Carlo simulation is to calibrate a model to European options prices, either in a self-contained manner or in combination with the valuation of an exotic product. An application to extensions of Dupire's model is known as the *particle method* of [9].
- Permits the application of the *Latin hypercube* variance reduction optimization: to simulate the successive time steps for all paths, reuse the same set of random numbers, but in a different order. We will review the Latin hypercube method later in this chapter in the context of Sobol's quasi-random number generator.

Step-wise simulations also cause difficulties with path-dependent products, and prevent path-wise differentiation in the context of AAD. For these reasons, we only consider path-wise simulations here. Note that this doesn't prevent pre-simulating paths in a step-wise manner, storing them in memory, and picking them there, path by path, instead of drawing new random paths, in a path-wise simulation.

Application to pricing

Discrete path-dependence It follows from linear complexity and dimension-independent accuracy that MC can be applied to price a wide variety of products, in a vast number of models, in reasonable time. It is also practically applicable in a large number of contexts. In particular, a path-wise simulation naturally handles path-dependent products: the entire path is simulated before the evaluation of the payoff, so all the information necessary to deal with path-dependencies is available when the payoff is computed.

There are two features in a financial product that the Monte-Carlo algorithm is not equipped to process naturally: *continuous* path-dependence and *callable* features. In both cases, a prior transformation step may turn these features into (discrete) path-dependence before the application of the Monte-Carlo algorithm. We don't implement or discuss

these steps in detail, but we introduce them quickly for the sake of completion and refer to Monte-Carlo literature for ample details.

Continuous path-dependence Monte-Carlo works with discrete scenarios, where the timeline is a finite set of event dates, so it cannot naturally handle continuous path-dependence. A classic example is the continuous barrier option, which dies if the barrier is breached *at any time* before expiry. The Monte-Carlo algorithm has no knowledge of what happens between event dates, so it cannot monitor the barrier continuously.

One solution is approximate the continuous barrier by one where the barrier is only active on a discrete set of event dates and remains unmonitored in between. This is called a *discretely monitored barrier*. It can be demonstrated (with the Brownian Bridge formula) that the value of the discretely monitored barrier converges to the continuous barrier when the time mesh between monitoring dates shrinks to 0, with error proportional to . This is therefore *not* a good approximation. The complexity of the simulation is inversely proportional to , so to reduce in the hope of approximating the continuous barrier may slow the computation down to an unacceptable extent.

A better solution is to find a discretely monitored barrier "equivalent" to a given continuous barrier. It should be intuitive that for a continuous up-and-out barrier and a monitoring period , there exists an equivalent discrete barrier on a *lower* barrier such that the two have the same price in a reasonably sizeable region below . A practical approximation was derived by [64], whereby is around 0.583 standard deviation on the barrier in between monitoring dates. We show an example in the next chapter. The approximation is very practical but unfortunately not precise enough to benchmark Monte-Carlo estimates.

Another solution is to deflate the payoffs of the paths where the barrier was not breached by the probability that the barrier was hit in between monitoring dates. This probability is given by the Brownian Bridge formula, under the approximation that the barrier index is Gaussian in between time steps,[2] and with an estimation of its volatility around the barrier.

These pre-calculations modify the product into an equivalent product[3] with discrete path-dependence, which is naturally handled with a subsequent Monte-Carlo valuation.

We don't implement the pre-processing of continuous path-dependent products in this publication. We only consider discrete path-dependence. Our implementation of Chapter 6 does not preclude continuous path-dependence, but it assumes that necessary pre-processing was performed in a prior step, so that our algorithms only deal with discrete path-dependence.

Callable products and xVA One exotic feature MC is not equipped to value is early exercise, as discussed in the previous chapter. With early exercise, cash-flows depend on the future values of the product, and when payoffs are computed, those future values are not available as part of the path. Regulatory calculations like CVA also subject cash-flows to future product values and are therefore vulnerable to the same problem.

One solution could be to estimate future values with *nested MC simulations*, as illustrated above, from each exercise/exposure date in every path of the outer MC. This approach is obviously extremely expensive and inappropriate in a time-critical environment. The better solution, and the established one in the industry, is the least squares method (LSM) algorithm of [14] and [13], illustrated below:

LSM approaches future values by regression over a set of basis functions of the state variables. The regression takes place *before* simulation. It uses simulations, too, which we call *pre-simulations* to distinguish them from the "main" Monte-Carlo valuation. Under these pre-simulations, we generate a *simulated learning set*: for every scenario, we store some basis functions of the state variables on the proxy date (exercise date) in a matrix , and the (numeraire deflated to the proxy date) sum of the cash flows *paid after the proxy date* in a vector . We use the simulated sample to *learn* the parameters of a functional dependence of the value of the transaction on a proxy date, the conditional expectation of its (deflated) future cash-flows, to a linear combination of basis functions of the state variables, hence, a function of the state vector, by linear regression of over .

In case we have multiple call dates (or proxy dates), the regression must be repeated for every proxy date, where the sum of cash flows for a proxy date depends on the exercise strategy after that date, hence, on the proxies for the future proxy dates. Therefore, regressions must be performed recursively in reverse chronological order.

The linear regressions may be performed with the well-known *normal equation*:

although this is vulnerable to co-linearity and over-fitting, see [65], Chapter 3. For this reason, best practice applies more sophisticated regressions based on singular value decomposition (SVD, see [20], Chapters 2 and 15), principal component analysis, or *regularization* (see [65], Chapter 3). Even more accurate proxies may be obtained out of the learning set with *deep learning*; see [65], chapter 5, where the basis functions are themselves learned.

The production of the regression proxies turns the product into a classic path-dependent, as discussed in the previous chapter. The transformed product is valued with Monte-Carlo simulations, like any other

path-dependent product, as long as the basis functions of the regression are included in the samples for their proxy date, so the proxies can be computed in the simulations.

LSM was initially designed to value callable exotics with Monte-Carlo, but it also became best practice for regulatory calculations after 2008–2011.

We do not cover LSM in this publication further than this brief introduction. We refer to the original papers of [14] and [13], chapter 8 of Glasserman [62] for details, and [31] for the modern implementation, application to xVA, and differentiation. Here, we only deal with (discrete) path-dependence, assuming that, if the product did have early exercise features, they were turned into path dependencies with a prior LSM step.

5.2 SIMULATION OF DYNAMIC MODELS

With the clarifications of the previous section, all that remains is to specify how exactly we "draw" paths from the distribution of the scenario specified in the model. The path is a collection of samples of observations on a discrete set of event dates . The definition of the timeline and of the samples on its event dates is part of the specification of the product. The responsibility to draw the paths belongs with the model, which specifies the probability distribution of . We note, once again, the separation of responsibilities between the model and the product. It is this separation that allows the design of a generic, versatile simulation libraries, where models and products are *loosely coupled* and only communicate through simulated paths. The responsibility of the model is to simulate the paths in accordance with the definition of the scenario in the product. The product consumes a simulated path to compute a payoff. This neat separation is somewhat muddied in the context of continuously path-dependent or callable products, although it may be reestablished by a pre-processing step, which allows to implement the simulation li-

brary itself with a clean separation. We will see in the next chapter how exactly this is implemented in code.

Drawing samples from a distribution

To draw a random number from a probability distribution , we can draw a *uniform* random number in and apply the transformation . It is immediate that follows the distribution , since:

We can only do this when the distribution is known. We can only do this *efficiently* when (or a good approximation of it) is known *and cheap to evaluate*.

To draw a standard Gaussian number, we apply the inverse cumulative Gaussian distribution to a uniform number. The function doesn't have a known analytic expression, but efficient approximations have been derived, like the one from Beasley, Springer, and Moro [66], widely implemented in financial systems, and which code is found in gaussians.h in our repository.[4] We can draw a Gaussian number with mean and variance with a trivial transformation of a standard Gaussian number, as it is immediately visible that:

follows the desired distribution.

To draw a number from a chi-square distribution with degrees of freedom is more difficult. There exists, to our knowledge, no efficient, accurate approximation of the inverse chi-square distribution . It is generally best to draw independent Gaussians and sum their squares. What is apparent in this example, where we need independent uniform numbers to draw *one* chi-square distributed number, is that the dimension of the uniform sample does not always correspond to the dimension of the drawn distribution. This consideration affects the design of generic simulation libraries.

In general, it takes independent uniform numbers (what we call a uniform *vector* of dimension ; we don't always ensure that the components are independent, but this is always assumed) to draw one random number from a given distribution , the most efficient transformation of the uniforms into one number with distribution depending on the nature of the distribution. It follows that, to draw independent random numbers from some distribution , we draw independent uniform vectors , each of dimension , and apply the transformation specific to .

To draw a random number from a *multidimensional* distribution , we follow the same steps. Depending on the nature of the distribution, an efficient transformation *may* exist that turns a uniform vector of dimension into a vector of dimension following the distribution .

To draw a multivariate Gaussian distribution in dimension , of mean and covariance matrix , we note that there exists a squared matrix of dimension such that . Denote a vector of independent standard Gaussian variables. Then:

is a Gaussian vector with mean and covariance matrix . A “square root” of can be found, for instance, by Cholesky's decomposition, of complexity cubic in , covered in chapter 2 of *Numerical Recipes* [20] with code. It follows that it is expensive to draw one sample from a multidimensional Gaussian distribution, but it cheap to draw a large number of independent, identically distributed samples, because the expensive Cholesky decomposition is performed only once. Also note that it takes independent uniforms to draw an -dimensional Gaussian; hence, in this particular case, the two dimensions coincide.

To draw a *path* in a model defined by SDE:

(where is a standard -dimensional Brownian motion with independent components and the state vector is in the Markov dimen-

sion) and the state to market mapping:

also follows the same steps. Draw a uniform vector in dimension and apply a model-specific transformation to turn it into a path . The dimension of the uniform vector necessary to draw one path is called the *dimension* of the simulation.[5]

The transformation of into is called a *simulation scheme*. Simulation schemes are model specific, and in particular, the means to efficiently turn into a realization of are dependent on the model, although general-purpose simulation schemes, applicable to many different models, exist in literature. We discuss the most common ones below.

Simulation schemes

A simulation scheme is a transformation that produces a path in a model out of a uniform vector. We discuss uniform number generators in the next section. In this section, we are given a uniform vector , and we investigate its transformation into a path.

What complicates the simulation of paths is that dynamic models don't explicitly specify the multidimensional distribution of the observations in the scenario, but the SDE of a state vector , and a mapping that transforms a path of into a collection of market samples on the event dates , and which may or may not be timewise, as previously explained. It follows that simulation schemes involve two steps:[6]

1. Generate a path of over a collection of time steps called *simulation timeline*, consuming the uniform vector . The simulation timeline doesn't necessarily correspond to the collection of event dates in the product. Some simulation schemes, like Euler's (pre-

sented shortly in detail), are inaccurate over big steps and require additional time steps to accurately simulate .

2. Apply the model specific state to market mapping to turn the path of over the simulation timeline into a collection of samples over the event dates. The transformation is conducted by iteration over the event dates , and consumes either the state in the case of a time-wise mapping or the path otherwise.

The number of uniform numbers required to generate a path of , the dimension of the simulation, depends on the simulation scheme.

Transition probability sampling

The simplest simulation scheme is the one that applies the transition probabilities of the SDE of , the distribution of conditional to :

with . is known, and is sampled from after , sequentially building a path over the simulation timeline . Despite its simplicity, the transitional scheme is extremely efficient, because it draws increments from the transition distributions over big steps without loss of accuracy. This scheme can accurately simulate over the event dates without the need for additional simulation time steps. It follows that the simulation timeline coincides with the product's timeline, and the simulation is conducted with the minimum possible number of time steps. Since its complexity is evidently linear in the number of time steps, the result is a particularly efficient simulation:

Of course, this scheme can only be applied with models where the transition probabilities are known and explicit. This not the case, for

example, in Dupire's model, where volatility changes with spot and time in accordance with an arbitrary specification of the local volatility structure, and the resulting transition probabilities are unknown.

In addition, the transitional scheme can only be applied *efficiently* when the transition probabilities are cheap to sample. Gaussian sampling is fast and efficient, but other distributions may be expensive to sample and defeat the benefits of the scheme. For instance, in the constant elasticity volatility (CEV) extension of Black and Scholes:

it was demonstrated in [67] that transition probabilities are displaced chi-square distributions with known parameters. To draw samples from a chi-square distribution is expensive, to the point where different simulation schemes, like Euler's, generally perform better, despite the additional time steps required on the simulation timeline.

In practice, the transitional scheme is mostly restricted to models with Gaussian transition probabilities, like Black-Scholes, LGM, or multidimensional extensions thereof. In addition, to simulate over big steps is only possible with a time-wise mapping . In this case, we can simulate with big steps over the event dates and apply the mapping on each event date. When the model doesn't provide a time-wise mapping, the sample on an event date is a functional of , and the path for cannot be simulated over big steps without loss of accuracy. This being said, as seen with LGM on page 175, it is generally possible to rewrite a model under an appropriate measure so it accepts a time-wise mapping. This is desirable for performance, the alternative being to simulate with small time steps over a dense simulation timeline, so that the mapping can be applied accurately, at the cost of expensive additional time steps.

A textbook illustration of the transitional scheme is the Black and Scholes model, where the risk-neutral transition probabilities of the spot price are log-normal, and all market variables are deterministic

functions of the spot price. We have seen on page 163 that the transition probability distributions in Black and Scholes, under the risk-neutral measure, are:

where

and is a standard Gaussian variable. In addition, increments are independent, and it follows that a path for the spot over the event dates , with the correct joint distribution, is produced with the simple recursion:

where is known, the s are independent uniform numbers and hence the successive are independent, standard Gaussians. Rates and dividend yields are deterministic, and forwards of all maturities are deterministic, explicit functions of the spot price :

It follows that market samples on an event date are all deterministic functions of , therefore, there is no need for additional time steps on the simulation timeline: the joint distribution of the spot is correctly sampled over big steps, and the mapping to market samples on an event date only requires the spot on the event date. It follows that we need independent uniform numbers to generate a path, the number of event dates in the product. The dimension of the simulation is therefore , the lowest possible dimension permitted by the context and resulting in a particularly fast simulation.

The transitional scheme is applicable to Gaussian models like Black and Scholes or LGM, including multidimensional extensions, but with

other models, like Dupire or non-Gaussian specifications of Cheyette, transition probabilities are either unknown or expensive to sample, and a different scheme must be invoked, one that does not require an explicit knowledge of transition probabilities. Euler's scheme offers a general-purpose fallback scheme, applicable to all kinds of diffusions, and always consuming Gaussian random numbers. The catch is that its accuracy deteriorates over big steps, so it requires additional time steps on the simulation timeline, with additional linear overhead.

Euler's scheme

Euler's scheme basically replaces the infinitesimal moves “d-something” in the SDE:

by small, but not infinitesimal, moves “ -something”:

More formally, and knowing that is a vector of [7] independent Gaussians with mean 0 and variance , Euler's scheme consists in the following recursion over the simulation timeline:

where the are a series of independent Gaussian vectors, each consisting in independent Gaussian numbers. It follows that we need random numbers for every step of Euler's scheme, a total simulation dimension of numbers for an entire path over a simulation timeline of time steps.

Euler's scheme corresponds to a Gaussian approximation with constant diffusion coefficients between time steps; hence, it becomes inaccurate for large time steps. On the other hand, all diffusions are locally Gaussian, so Euler's scheme does converge.[8]

Therefore, when the event dates are widely spaced, we must insert additional time steps into the simulation timeline for the purpose of an accurate simulation. For instance, if a product has annual fixings, and we force a maximum space of three months on the timeline to limit inaccuracy, we evenly insert three additional simulation steps in between each event date. Those additional time steps are for the purpose of the simulation only. In particular, they don't produce market samples. The product's timeline, the one consumed in the computation of the payoff, is unchanged. It is important to distinguish the product's timeline, which is the collection of *event dates* where market samples are observed for the computation of the payoff, and depend on the product, and the simulation timeline, which is the collection of *time steps*, inserted for the purpose of an accurate simulation of the state vector , depending on the model and its simulation scheme:

Alternative schemes

Simulation complexity is linear in the number of time steps, so it is desirable to minimize the number of the additional steps in the simulation timeline. To this end, among others, many alternative schemes were designed by researchers and practitioners, to provide a better approximation over big steps in certain situations. We cite the most interesting ones, referring to [62], [63], and the original publications for details.

Euler's scheme is said to be of order one because it converges in .
It was extended by Milstein to an order-two scheme, theoretically allowing longer time steps, but with smoothness constraints on the parameters of the SDE.

Andreasen and Huge's “random grids” [68] offer an original solution for the simulation of a wide class of local and stochastic volatility models, over big steps, with a scheme that combines the benefits of Monte-Carlo and FDM, but which implementation requires significant effort.

Special schemes for stochastic volatility models a la Heston were proposed by Andersen and Piterbarg in the Heston chapter of [6], and Andreasen in [8]. Heston models a stochastic variance, which cannot be correctly simulated over Euler's scheme: Euler's scheme is Gaussian in between time steps; therefore it allows negative samples over the discrete timeline even when the continuous time diffusion prohibits them. A variance obviously cannot be negative, so Euler's scheme is not suitable in this case. Heston, and stochastic volatility models in general, require specialized schemes.

As a curiosity, so-called *spectral* schemes produce *continuous* paths with a finite number of random numbers by sampling the harmonic basis functions of the Brownian motion, see [69], but turn out to be more interesting than useful in contexts of practical relevance:

where the s are standard Gaussian scalars.

In our code, we apply a transitional scheme for Black and Scholes and a log-Euler scheme (as the name indicates, Euler on) for Dupire. We don't implement other schemes, but the design of the library accommodates any scheme a client code may wish to implement in its concrete model classes.

5.3 RANDOM NUMBERS

Simulation schemes consume a number of uniform random numbers to generate a path. We complete our presentation of simulation algorithms with a discussion of random numbers. A program that produces (pseudo) random numbers is called a *random number generator* (RNG). Many programming languages like C++ offer RNGs in their standard libraries. The implementation is platform and compiler specific. Their statistical correctness is unknown: how well they span the uniform space and how independent are the successive draws.

Their implementation is hidden, which is a problem for a parallel application, as we will see in the next chapter.

For these reasons, a professional simulation library cannot rely on the standard library RNGs and must include a custom implementation of generators purposely designed for simulations with proven statistical correctness and convergence properties. The current industry standard is the combined recursive generator *mrg32k3a*, designed by L'Ecuyer in 1996 [70].

The design of random number generators is a discipline of its own. Interested readers will find a wealth of information on L'Ecuyer's page in the University of Montreal.[9] In particular, his synthetic note[10] offers a clear and useful overview in just 40 pages. L'Ecuyer also introduces efficient “skip-ahead” techniques introduced in the next chapter, and provides the C++ code for a number of generators.

We don't cover the design of random number generators, or investigate their statistical properties, which are discussed in deep detail in L'Ecuyer's documents. What we do is implement mrg32k3a. The algorithm is the following:

1. Initialize the first three values in two series of state variables and with and . The pair is called *seed*. The RNG produces the exact same sequence of numbers when seeded identically. The seed must satisfy where and .
2. Successive values of and are produced with the following recurrence:

   with:

3. The th random number is

5.4 BETTER RANDOM NUMBERS

Variance reduction methods and antithetic sampling

Monte-Carlo is easy to understand and implement, and adequate in virtually any valuation context, but it is expensive in CPU time and its convergence in the number of paths is slow. A vast amount of research, collectively known as *variance reduction methods*, conducted in the past decades, attempted to accelerate its convergence.

Perhaps the simplest variance reduction technique is *antithetic sampling*. For every path generated with the uniform vector , generate another one with its antithetic:

Not only does antithetic sampling balance the paths with a desirable symmetry, it also generates two paths for the cost of one random vector.

When the scheme consumes Gaussian numbers, antithetic sampling is even more efficient: for every path generated with the Gaussian vector , generate another one with , guaranteeing that the empirical mean of all components in the Gaussian vector is zero. This is a simple case of a family of variance reduction methods called *control variates*, where the simulation enforces that the empirical values of sample statistics such as means, standard deviations, or correlations match theoretical targets. In addition, we get two paths for the cost of one Gaussian vector, saving not only the generation of the uniforms, but also the not-so-cheap Gaussian transformations.

Antithetic sampling is a simple notion and translates into equally simple code. We implement antithetic sampling with mrg32k3a in the next chapter. We will mention other variance reduction methods, but we don't discuss them in detail, referring readers to chapter 4 of [62]. What we *do* cover is a special breed of "random" numbers particularly well suited to Monte-Carlo simulations.

Monte-Carlo as a numerical integration

We reiterate the sequence of steps involved in a Monte-Carlo valuation.[11] We generate and evaluate paths, each path number following the steps:

1. Pick a uniform random vector in the *hypercube* in dimension :
2. Turn into a Gaussian vector with the application of the inverse cumulative Gaussian distribution to its components:
3. Feed the Gaussian vector to the simulation scheme to produce a path for the model's state vector over the simulation timeline:
4. Use the model's mapping to turn the path into a collection of samples over the event dates:
5. Compute the product's payoff on this path:

Hence, is obtained from through a series of successive transformations. Therefore:

where :

The MC estimate of the value is:

The *theoretical* value (modulo discrete time) is:

Hence, Monte-Carlo is nothing more, nothing less, than the numerical approximation of the integral of a (complicated) function with a sequence of random samples drawn in the hypercube.

Koksma-Hhlawka inequality and low-discrepancy sequences

A random sequence of points is not necessarily the optimal choice for numerical integration. Standard numerical integration schemes, covered, for example, in [20], work in low dimension. Dimension is typically high in finance. With Euler's scheme, it is the number of time steps times the number of factors. For a simple weekly monitored 3y barrier in Black and Scholes or Dupire's model, the dimension is 156. Typical dimension for exotics is in the hundreds. For xVA and other regulations, it its generally in the thousands or tens of thousands, due to the large number of factors required to accurately simulate the large number of market variables affecting a netting set.

Numerical integration schemes would not work in such dimension, but there exists one helpful mathematical result known as the *Koksma-Hhlawka inequality*:

where the value and its estimate are defined above, the left-hand side is the absolute simulation error, is the variation of , something we have little control about, and is the *discrepancy* of the sequence of

points , a measure of how well it fills the hypercube.
Formally:

where is the volume of

the box . Hence, the discrepancy is the maximum error on the estimation of volumes, a measure of how *evenly* the sequence fills the space.

The Koksma-Hhlawka inequality nicely separates the characteristic of the integrated function and the discrepancy of the sequence of points for the estimation. It follows that random sampling is *not* optimal. Random sampling may cause clusters and holes in the hypercube, resulting in poor evenness and high discrepancy. It is best to use sequences of points purposely designed to minimize discrepancy. A number of such sequences were invented by Van Der Corput, Halton, or Faure, but the most successful one, by far, was designed by Sobol in 1967 USSR [71].

The following picture provides a visual intuition of the discrepancy of Sobol's sequence. It compares 512 points drawn randomly in
on the left to the 512 first Sobol points in dimension two on the right:

It is clear to the naked eye that Sobol's sequence effectively avoids the holes and clusters inevitable with a random sequence and fills the hypercube with a better “evenness.” The mathematical notion of discrepancy formalizes the notion of “evenness,” and the Koksma-Hhlawka inequality demonstrates the intuition that better evenness results in a more accurate integration.

Sobol's sequence

The construction speed of Sobol's sequence was massively improved in 1979 by Antonov and Saleev, whose algorithm generates successive Sobol points extremely fast, with just a few low-level bit-wise operations. It is this implementation that is typically presented in literature, including here. Over the past 20 years, Jaeckel [63] and Joe and Kuo [72], [73] performed considerable work on Sobol's *direction numbers*[12] so that the sequence could be practically applied in the very high dimension familiar to finance, achieving remarkable results. Sobol's sequence (with Antonov and Saleev's optimization and Jaeckel or Joe and Kuo's direction numbers) became a best practice in financial applications, which it remains to this day.

Sobol's sequence is not really a sequence of points in the hypercube , but a collection of sequences of numbers in . The sequences of scalar numbers on each axis of the sequence is self-contained, and the first coordinates of a sequence in dimension exactly correspond to the sequence in dimension .

Sobol generates sequences of *integers* between 0 and , so the th number on the axis is:

The integers on a given axis are produced by recursion: (the point 0 is not valid in Sobol; the first valid point is the point number 1) and:

where denotes the bit-wise exclusive or (xor), is the rightmost 0 bit in the binary expansion of , and are the 32 *direction numbers* for the sequence number .

The rightmost bit of every even number is 0; hence, once every two points, Sobol's recursion consists in xor-ing the first direction number to its state variable . The operation xor is associative and has the property that:

so flicks in and out of the state every two points. For the same reason, flicks in and out every four points, flicks in and out every eight points, and, more generally, (with) flicks in and out every points. Sobol's sequence is illustrated below:

More generally, for , the state is a combination of the first direction numbers:

where the weights are either zero or 1: flicks every 2 points, every 4 points, every 8 points, every points. It follows, importantly, that the first points in the sequence span *all the possible combinations* of the first direction numbers, each combination being represented exactly once.

It follows that the Sobol scalar in is:

where the are the normalized direction numbers.

Direction numbers on the first axis

On the first axis , the direction numbers are simply:

and it follows that the normalized numbers are:

so the sequence of the 32 normalized direction numbers is , , , , etc. Or, bit-wise, the th direction number has a bit of 1, spaces from the right, and 0 everywhere else:

It follows that the sequence of numbers on the first axis is:

or, graphically:

where we see that the normalized direction number does not kick in before all the combinations of the for were exhausted. It is visible on the chart that goes in the middle of the axis, and go in the middle of the spaces on the left and right of , and the following four numbers go straight in the middle of the spaces left in the previous sequence. The same pattern carries on indefinitely, progressively and incrementally filling the axis in the middle of the spaces left by the previous numbers.

It follows that the first Sobol scalars on the first axis (excluding the first and invalid scalar) evenly span the axis

:

and we begin to build an intuition for the performance of the sequence: the first “random” numbers are even quantiles of the uniform distribution. When transformed into a Gaussian or another distribution, these numbers sample even quantiles on the target distribution.

Latin hypercube

This property, where the first numbers are evenly spaced by on the axis , holds for all the axes. *Sobol samples the same numbers on all the axes, but in a different order*.

The direction numbers on all the axes have the following bitwise property:

The th bit is still one, so still flicks in and out every two numbers, every four numbers, every eight numbers and so forth. The bits on the right of are still 0, so will not kick in before all the combinations of , , and are exhausted. It follows that the first numbers are still the quantiles .

But the bits on the left of are no longer all zero. Some are zero, some are one, depending on the axis, and, crucially, they are different on different axes. When the direction number flicks in or out every points in the sequence, it doesn't only flick . Its bits on the left of also flick some for , shuffling their order of flickering.

It follows that the same numbers are sampled on all the axes, but in a different order. In addition, these numbers are even quantiles. The

chart below shows the first 15 Sobol points in dimension 2, where it is visible that the points sample the quantiles on both axes, but in a different order.

This property where points sample the hypercube in such a way that for every coordinate :

is not unique to Sobol and well known under the name *latin hypercube*. Latin hypercube samples the coordinates in a balanced manner. It is a form of control variate, since the points correctly sample the th quantiles of the marginal distributions by construction.

Latin hypercube may be implemented directly with a sampling strategy known as *balanced sampling*: to produce points in the hypercube, sample the first axis with the sequence for and all the other axes by random permutations of the same values. Balanced sampling improves the convergence of Monte-Carlo simulations over random sampling, sometimes noticeably, and its construction is extremely efficient. Instead of randomly shuffling uniform samples before applying the Gaussian transformation, we can randomly shuffle the Gaussian quantiles , so that the rather expensive Gaussian transformation is only applied on the first axis times instead of the usual . Balanced sampling may be further optimized, both in speed and quality, in combination with antithetic sampling.

One catch is that balanced sampling is not an *incremental* sampling strategy. We cannot sample points and then additional points. All the points are generated together, coordinate by coordinate and not point by point. To apply balance sampling in a path-wise simulation, the random vectors must be stored in memory and picked one by one, imposing a heavy load on RAM and cache. Sobol, on the contrary,

is an incremental variation of the latin hypercube, whereby the points are delivered in a sequence, one -dimensional point at a time.

Latin hypercube evenly samples the *marginal* distributions, but offers no guarantee for the *joint* distributions. An unfortunate draw may very well sample two coordinates in a similar order, as illustrated below with 15 points in dimension 2:

While the marginal distributions are still sampled evenly, it is clear that the empirical *joint* distribution is wrong: the two random numbers, supposedly independent, are 100% dependent in the sample. Seen from the discrepancy perspective, it is visibly poor, the points being clustered around the diagonal, leaving the rest of the space empty. Because the marginal distributions are correctly sampled, a simulation produced with the sample still correctly values all sets of European cash-flows, but the values of exotics would be heavily biased due to the poor sampling of the copula (see Section 4.3).

It therefore appears that, while the latin hypercube property of random numbers is a highly desirable one in the context of Monte-Carlo simulations, it is not in itself sufficient to guarantee a correct representation of joint distributions or a low discrepancy. Additional mechanisms should be in place so that the different axes are sampled in a dissimilar and independent *order*. Sobol's sequence achieves such “order independence” with the definition of its direction numbers.

Direction numbers on all axes

We did not, in our short presentation, specify the direction numbers applied in Sobol's sequence. We did define the 32 direction numbers of the first axis, and introduce a general property of all direction numbers, whereby the th bit of the th direction number is always one and the bits on the right of are always zero. This is what guarantees the latin hypercube property. But the sampling order depends on the

bits on the left of . These bits are zero on the first axis, and specified in a different manner on all other axes, the specification on a given axis determining the sampling order on that axis.

The quality of the sequence, its ability to sample independent uniform numbers on each axis, resulting in a low discrepancy over the hypercube, therefore depends on the specification of the (left bits of the) direction numbers. Sobol [71] delivered a recursive mechanism for the construction of the direction numbers, but it so happens that the starting values for the recursion, called *initializers*, massively affect the quality of the sequence in high dimension. Jaeckel [63] and Joe and Kuo [72], [73] found sets of direction numbers that result in a high-quality sequence in high dimension. Without such sets of high-quality direction numbers, Sobol's sequence is practically unusable in finance, the resulting Monte-Carlo estimates being heavily biased in high dimension. It is only after the construction of direction numbers was resolved that Sobol became best practice in finance.

The construction of the direction numbers is out of our scope. We refer the interested readers to chapter 8 of [63]. Joe and Kuo's dedicated page, http://web.maths.unsw.edu.au/~fkuo/sobol/, collects many resources, including papers, synthetic notes, various sets of direction numbers in dimension up to 21,201, and demonstration code in C++ for the construction of the direction numbers and the points in Sobol's sequence. Our implementation of the next chapter uses their direction numbers in dimension up to 1,111 derived in their 2003 paper [72].

NOTES

1 We only consider models of finite dimension here. Infinite or continuous dimension is not suitable for a practical implementation. With infinite dimensional models like HJM, it is generally a finite dimensional version like LMM that is simulated.

2 An approximation MC also makes for path generation with the *Euler* scheme, as we will see shortly.

3 At least approximately.

4 Moro's approximation is accurate and reasonably fast, but it contains control flow, and therefore, it cannot be vectorized; see our introduction to SIMD on page 9. Most known alternatives, like Acklam's interpolation cited by Jaeckel [63], also contain control flow. Intel's free IPP library offers a vectorized alternative but this is not standard, portable C++, and therefore out of our scope.

5 Not to be confused with the dimension of the scenario, the dimension of the samples, the Markov dimension of the model, or the number of its factors, all of those being different notions!

6 The two steps are often conducted simultaneously in practice, but it helps to separate them for the purpose of the presentation.

7 We recall that is the number of factors in the model, the dimension of the Brownian motion in its SDE.

8 Its weak convergence is in , which means that in the limit of an infinite number of paths, where the only error comes from the discrete simulation timeline, that error is asymptotically proportional to .

9 https://www.iro.umontreal.ca/~lecuyer/.

10 https://www.iro.umontreal.ca/~lecuyer/myftp/papers/handstat.pdf.

11 Assuming the simulation scheme consumes Gaussian numbers, as is almost always the case.

12 Explained shortly.