Monte Carlo Methods

By Matthew Fried

Abstract

This project will discuss Monte Carlo methods with applications to derivatives and asset pricing. We will discuss random number generation using linear congruential generators and sampling methods. We will justify our approach and then analyze several variance reduction techniques. We will conclude this project with applications, such as option modeling, showing the impressive reduction in trials with the control variate method, among others. Each model will be tested for accuracy, with results reported below and in the accompanying slide set..

This thesis is in partial fulfillment of the requirements for a Masters in Mathematics from Yeshiva University

I want to thank Dr. Roldan for generously giving of his time and expertise. I am indebted to him for everything he has done to assist and guide me in this field.

0 Introduction

In this paper we will introduce the concept of Monte Carlo methods. We start from the necessary background in probability and follow through to several applications, with variance reduction techniques used to limit the computational power required thereof. The foundation is to first understand that the Strong Law of Large Numbers allows us to create many samples that will retain our desired distribution, which will eventually result in a Gaussian process via a stochastic random sampling for each time t in our model. After this, we develop concepts in Wiener processes and then Brownian motion. There are several approaches to building a Brownian model, of which we discuss one of the more application oriented techniques, principal component construction. From there we briefly discuss geometric Brownian motion, an established model in areas such as physics and finance.

The Euler-Maruyama and Milstein methods are then explained, as their discretization techniques are typically used in many applications. We show the Black-Scholes formula with respect to financial contingency claims (often referred to as options); we then show code that models calls and puts based on the above methods. Our output of the discretized Monte Carlo results is compared to the famed Black-Scholes formula and we then spend time justifying our model (vis-a-vis the transformation from a stochastic differential equation to a discretized sampling of unbiased estimators).

Due to the necessary requirement of using pseudo random numbers, we discuss the conditions and metrics used in choosing these values for our Monte Carlo models. And, lastly, we show several variance reduction techniques (such as control variates, antithetic reduction, and sampling) that allow faster convergence of Monte Carlo models.

Monte Carlo methods are based on sampling large random data sets with distributions that follow test vector parameters. This is used to advance a probabilistic analysis within the universe of possible outcomes as applied to our state space (as will be described below). As such, one can test *many* possible outcomes of some function and thereby measure integrals on some predefined metric. For instance, in $[0,1] \times [0,1]$ (which can be extended to any number of dimensions) we can define a function f, and solve for the integral by sampling “enough” points in one of two possible way: (1) take the average of a large number of $f(x_i)$, or (2) get the ratio of the (true) boolean values associated with our given function [8]. Each of these approaches will be discussed further, see Section 1 below.

For another introductory example to gain a useful intuition of the topic we turn to financial engineering applications. Monte Carlo methods allow us to build a probabilistic and statistical sampling with a volume that formalizes the notion of measure. It is an intuitive, probabilistic design associated with an event, relative to the random outcomes. The law of large numbers ensures that our probabilistic estimates converge, and as we increase the volume of our sampling we reduce error estimates. The central limit theorem provides bounds on the

magnitude of the error based on our increasing draws, allowing us to effectively model continuous processes within given bounds.

In other words, we can choose some starting point and build a time series using the Euler-Maruyama or Milstein methods to create a cross-section of data series with the same average and standard deviation as our initial, underlying asset (or data). This “cloud” can be used for many purposes, such as when the Federal Reserve lays out the expected probabilities of rate changes (among other important metrics).

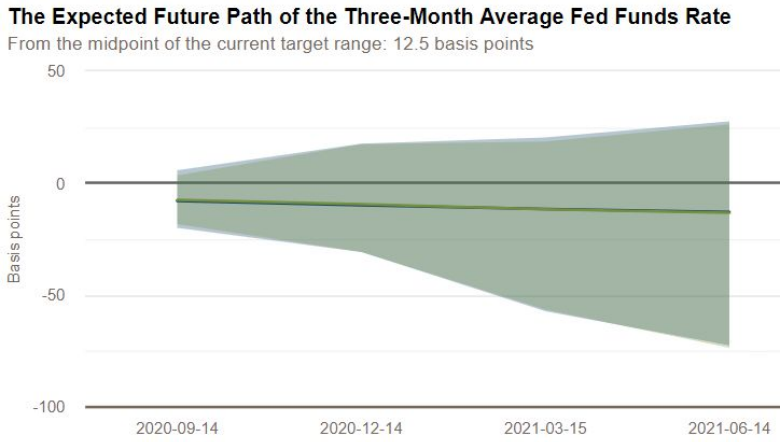


chart from the Federal Reserve Bank in Atlanta

We now state the Strong Law of Large Numbers (SLLN) which provides us with the above-mentioned guarantee that our random samples are bounded and converge to the expected average via the sample mean. Conceptually, the SLLN says that the more sample means we create the closer we will get to the true mean.

Theorem: (The Strong Law of Large Numbers) For X_n an independent, identically distributed (i.i.d.) sequence of random variables (r.v.) with $E[|X_n|] < \infty$, for $E[X_n] = \mu$,

$$\lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} X_k = \mu \ \ a.s. \ (almost\ surely).$$

As such, we can justify laws of asset pricing theory and represent derivative valuations as an expected value. As an aside, an example of a law of asset pricing theory is the law of one price that guarantees no arbitrage. The relation here is that Monte Carlo method allows us to compute enough samples to accurately derive a price and expect the law to hold within some epsilon bound. As stated, this process is often the product of computing multiple expectations. Also, we note, our error will be an outgrowth of the central limit theorem (CLT).

Theorem: (Central Limit Theorem) *Let X_n be an i.i.d. sequence of r.v. such that Var[X_n] =* σ^2 *<∞. Let* μ *= E[X_n], then as* $n \to \infty$ *the r.v. For any sample sequence* $\overline{X_n}$, $\sqrt{n}(\overline{X_n} - \mu)/\sigma$ *converges in distribution to a standard normal random variable* $N(0,1)$.

In terms of the sample mean $\widehat{\mu_n}$, the CLT states that $(\widehat{\mu_n} - \mu)\sqrt{n}/\sigma$ converges in distribution to a standard normal distribution N(0,1). As such, for $z \in R$, with a sample variance s^2 (which converges as well due to Slutzky's theorem [9]), we have the usual confidence intervals $\widehat{\mu_n} \pm z_{\frac{\delta}{2}} \frac{s_n}{\sqrt{n}}$, with z_δ being the $1-\delta$ quantile of the standard normal distribution.

Definition: The root mean squared error (RMSE)

$$[E([\tfrac{1}{N}\sum_i^N X_i - EX]^2)]^{\frac{1}{2}} = [Var(\tfrac{1}{N}\sum_i^N X_i)]^{\frac{1}{2}} = [\tfrac{1}{N^2} Var(\sum_i^N X_i)]^{\frac{1}{2}} = [\tfrac{1}{N^2} N\sigma^2]^{\frac{1}{2}} = \sigma/\sqrt{n}$$

The RMSE tells us how far the approximation will be from the true value, on average. On a more practical front, we need to be able to draw the samples with $O(n^{\frac{1}{2}})$ [such that $\frac{\sigma}{\sqrt{n}} < \varepsilon$] . This means that, unlike other methods which are often more computationally-efficient but have use-cases that can vary greatly for simpler dimensions, the relatively low rate of convergence may seem unattractive compared to numerical methods such as the Trapezoidal rule, with convergence rates of $O(n^2)$, however for d-dimensional stochastic processes, effective numerical methods often do not exist or are quite ineffective, as such, despite the above, Monte Carlo methods are an excellent and sometimes necessary choice; and, thus for many problems where the sample space is large or possibly infinite the square root convergence is quite competitive even against existing alternatives.

We will also discuss Quasi-Monte Carlo methods in which we use a low discrepancy sequence rather than random numbers, such as a Halton or Sobol sequence, and thereby get a convergence rate of $O(n)$.

Superior sampling methods can reduce the implicit constant in the rate of convergence significantly, and in this paper, beyond the general application to finance (and an explication of generating pseudo-random numbers and sample paths), we discuss specific variance reduction techniques.

As an aside, please note that all code contained here and in the colab notebook (posted in the github repository mentioned in the bibliography) is our own, except for the two snippets of the Control Variates section [2], which will be mentioned again in its place.

1 Motivation

As a further motivation, we use Monte Carlo methods to solve integrals with what Sauer [8] calls Monte Carlo Type 1.

We use a linear congruential generator (LCG), as discussed later on in this paper, to create random numbers and find the mean value of a function on [a,b], as defined by

$\frac{1}{(b-a)}\int_a^b f(x)dx$,by discretizing the function and using $\frac{1}{n}\sum_{i=1}^{n} f(u_i)$ where $x_i = ax_{i-1} + b \ \ (mod\ m)\ and\ u_i = x_i/m$ for multiplier a, offset b, and modulus m. The function evaluates a random number, with the final output being the average of n trials. This allows us to quickly solve integrals (some of which may be intractable otherwise). As an exploratory test, we use $\int_0^1 x^2\,dx = \frac{1}{3} \approx 0.33333$ and the Euler-Mascheroni equation $\int_0^1 \left(\frac{1}{\ln x} + \frac{1}{1-x}\right) dx$ = 0.57721566490.

The chart below details three different approaches, each with between one thousand and one billion samples. We list the output value and absolute error for each function.

The first case draws random values from a regular LCG with ANSI C parameters, the second case uses stratified sampling (converting the random samples into strata) as will be explained later in the paper, and the third case uses antithetic sampling (which takes a *u* and *1-u* for each sample) as will be explained later as well.

We can see that both our approaches did significantly better than that without variance reduction.

Regular	Num Trials	Func1	Func1_Error	Func2	Func2_Error
1	1,000	0.359841	0.026507	0.571216	0.006000
2	10,000	0.341231	0.007898	0.575420	0.001796
3	100,000	0.335710	0.002377	0.576900	0.000316
4	1,000,000	0.333119	0.000214	0.577243	0.000027
5	10,000,000	0.333260	0.000074	0.577169	0.000046
6	100,000,000	0.333386	0.000053	0.577195	0.000021
7	1,000,000,000	0.333338	0.000005	0.577218	0.000002

Stratified	Num Trials	Func1	Func1_Error	Func2	Func2_Error
1	1,000	0.335123	0.001790	0.576847	0.000368
2	10,000	0.333915	0.000582	0.577076	0.000140
3	100,000	0.333544	0.000211	0.577180	0.000036
4	1,000,000	0.333300	0.000033	0.577230	0.000014
5	10,000,000	0.333272	0.000061	0.577233	0.000018
6	100,000,000	0.333333	0.000000	0.577217	0.000001
7	1,000,000,000	0.333337	0.000004	0.577214	0.000001

Antithetic	Num Trials	Func1	Func1_Error	Func2	Func2_Error
1	1,000	0.337096	0.003763	0.576675	0.000541
2	10,000	0.334077	0.000744	0.576946	0.000270
3	100,000	0.333602	0.000269	0.577290	0.000075
4	1,000,000	0.333451	0.000117	0.577218	0.000002
5	10,000,000	0.333248	0.000085	0.577194	0.000022
6	100,000,000	0.333302	0.000031	0.577209	0.000006
7	1,000,000,000	0.333335	0.000002	0.577217	0.000001

Fig 1. The above chart identifies the function value and error for the indicated function and error.

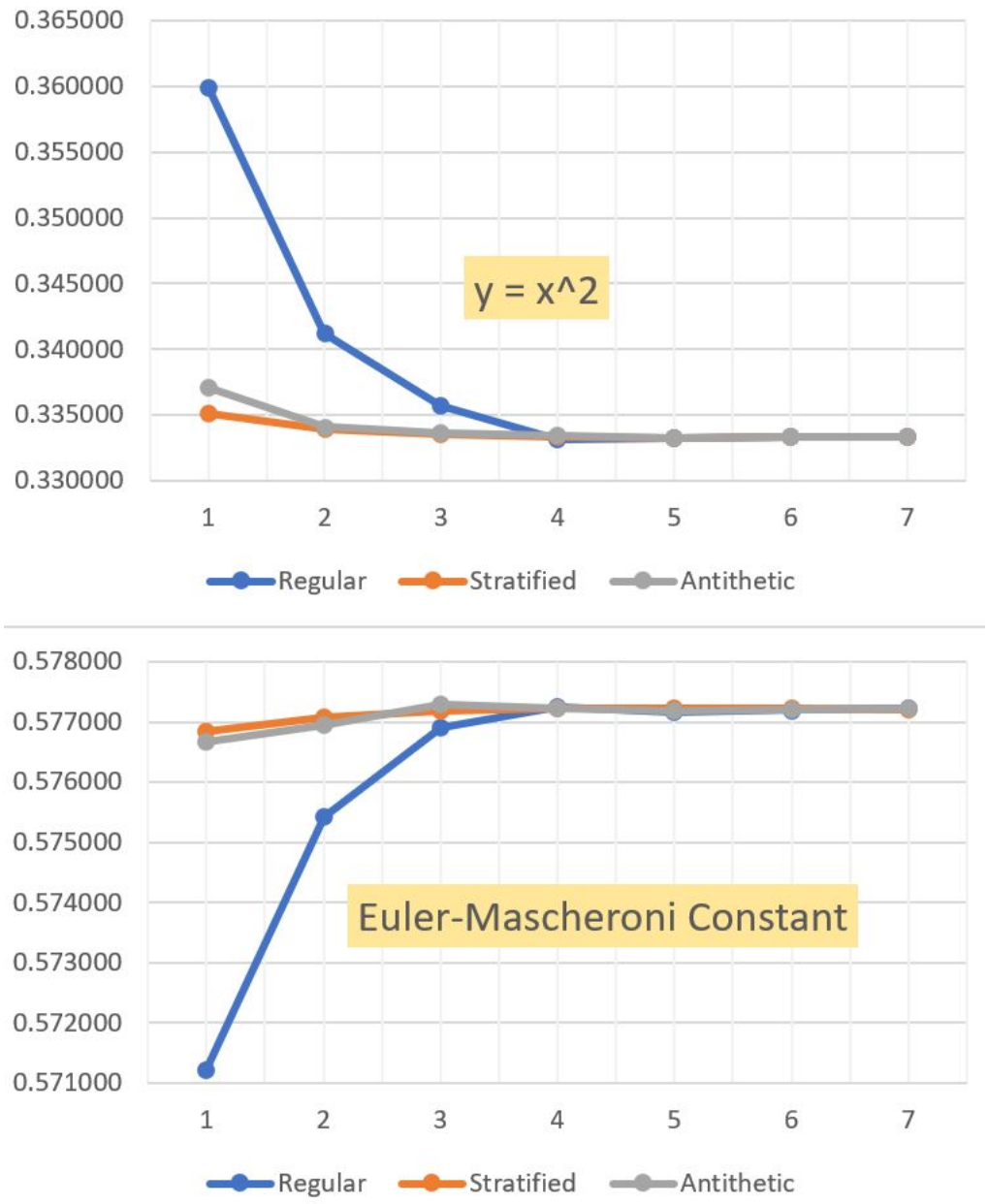


Fig 2. The above shows the convergence for each function.

The standard variance reduction techniques to be described below (e.g. stratified sampling and antithetic reduction), do indeed improve the error and hasten the speed of convergence. The code for the colab notebook used to determine this is noted with explanation in the bibliography.

1.1 Wiener Process

A stochastic process X is a collection of random variables random variables $(X_t,\ t \varepsilon T) = X_t(\omega),\ t \varepsilon T,\ \omega \varepsilon \Omega$ defined on some space Ω, with the sample space Ω as the set of all possible outcomes, T an event space, with an event being a set of outcomes in the sample space, and P as the probability function which assigns each event in the event space a probability between 0 and 1 [pg 23, 19].

Definition: A *Wiener process* is a stochastic process $W = \{W_t,\ t \geq 0\}$ with the properties:

(1) $\forall\ t_1 < t_2 \leq t_3 < t_4,\ W_{t_4} - W_{t_3} and\ W_{t_2} - W_{t_1}\ in\ W$ are independent random variables

(2) $\forall\ t \geq s \geq 0,\ W_t - W_s \sim N(0, t-s)$

(3) $W_0 = 0\ and\ \{W_t\}$ has continuous paths

As such, a Wiener process is a stochastic process with independent increments that have Gaussian stationarity and continuity of paths. A stochastic process is called Gaussian if all of its finite-dimensional distributions are multivariate Gaussian [pg 27, 19]. A Gaussian process has the properties of said distribution as well, with some of the following properties as well:

(1) For W, a Gaussian process, $E(W_t) = 0$

(2) $Cov(W_s, W_t) = min\{s, t\}$

(3) For W on [0,T],

$$\lim_{n\to\infty} \sum_{i=1}^{n} |W_{t_i} - W_{t_{i-1}}| = \infty \quad almost\ surely\ where\ 0 = t_0 < ... < t_n = T\ and\ \lim_{n\to\infty} max_i\{t_{i+1} - t_i\} = 0$$

This means that each sample path has infinite length and is not differentiable; in other words, they do not have bounded variation.

The first point is a matter of definition. The second point can be shown below.

Proof of (2):

$Cov(W_s, W_t) = E[W_sW_t] - E[W_s]E[W_t] = E[W_s(W_s + W_t - W_s)] = E[W^2_s] + E[W_s]E[W_t - W_s] = Var(W_s) + 0 = s$ for $s<t$.

(Step 4 is based on the independence of $W_t - W_s$.)

There are several other properties of a Wiener process that are more well-known and will not be explicated in depth here. These includes the following:

(1) quadratic variation. The definition of which is that

$$[X]_t = \lim_{\|P\|\to 0} \sum_{k=1}^{n} (X_{t_k} - X_{t_{k-1}})^2\ for\ X_t\ a\ real\ valued\ stochastic\ process\ with\ time\ t$$

, *and* P *over the partitions of the interval* $[0, t]$. So while the sample paths of W on [0,t] do

not have bounded variation, the Wiener process is the unique continuous martingale (see below) with $W_0 = 0$ and where quadratic variation is bounded [5, Pg 178].

(2) being martingales with respect to their own filtration. (Note: A filtration on a probability space (Ω, F, P) is a family (F(t) : t ≥ 0) of σ-algebras such that F(s) ⊂ F(t) ⊂ F for all s < t.)

(3) having the time-homogenous strong Markov property $(W_{\tau+t} | W_u,\ u \le \tau) \sim (W_{\tau+t} \mid W_\tau)$ [i.e. the Markov stochastic process is where only the current value of a variable is relevant for predicting the future movement], and

(4) invariance under scaling.

From the definition, the exact algorithm to generate a Wiener process follows[5, Pg 180]:

- For distinct t_i and $Z_i \sim N(0,1)$, $W_{t_k} = \sum_{i=1}^{k} \sqrt{t_k - t_{k-1}} * Z_i \qquad k = 1, ..., n$

While this does not give a continuous path, and introduces error, an infinite process of linear interpolation would correct this.

1.2 Brownian Motion

A process $\{B_t,\ t \ge 0\}\ satisfying\ B_t = \mu t + \sigma W_t,\quad t \ge 0$ is called a Brownian motion with drift μ and diffusion coefficient σ^2. Here $\{W_t,\ t \ge 0\}$ is a Wiener process. The Brownian motion process is the solution to the stochastic differential equation $dB_t = \mu\, dt + \sigma\, dW_t,\ t \ge 0,\ B_0 = 0.$

In a more generalized form, a stochastic differential equation $dX_t = a(X_t, t)dt + b(X_t, t)dW_t$ has a(x,t) and b(x,t) as deterministic functions with $a = drift$ and $b = diffusion\ coefficient.$ SDEs are based on the same principles as ODEs, however here the unknown function is driven by randomness. The equation can be interpreted as the sum of infinitesimal displacement $a(X_t, t)dt$ and an infinitesimal noise term $b(X_t, t)dW_t$.

The regular construction of a standard one-dimensional Brownian motion on [0,T] is a stochastic process $\{W_t,\ 0 \le t \le T\}$ with the properties outlined above for a Wiener process (the name Brownian motion is used due to historical significance).

A process X_t is a Brownian motion $(X_t \sim BM(\mu, \sigma^2)$ if $(X_t - \mu t)/\sigma$ is a standard Brownian motion defined by $B_t = \mu t + \sigma W_t$. This implies $(X_t \sim N(\mu t, \sigma^2 t)$ with the same SDE as above $X_t = X_0 + \int_0^t \mu\, ds + \int_0^t \sigma\, dW_s$. As mentioned above, each increment is normally distributed with mean $E[X_t - X_s] = \int_s^t \mu\, du$ and $Var[X_t - X_s] = Var[\int_s^t \sigma\, dW_u = \int_s^t \sigma^2 du$.

The construction of a simulation would then be $X_{t_{i+1}} = X_{t_i} + \mu_{t_{i+1}-t_i} + \sigma\sqrt{t_{i+1} - t_i} * Z_{i+1} \quad for\ i = 0, .., n-1$. As above, these methods are ***exact*** in the sense that the joint distribution of the simulated values coincides with the joint distribution of the corresponding Brownian motion.

1.3 Principal Components and the Connection to Brownian Motion

While there are several different approaches to constructing Brownian motion instead of a random walk construction, such as creating a Brownian bridge (mentioned above), I will focus on a method similar to PCA, which has some surprising results.

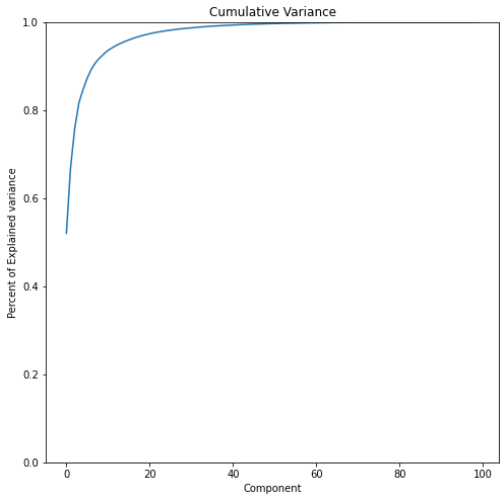
As a prerequisite, we note that $Cov(W_s, W_t) = min\{s, t\}$ as above, and therefore, for the covariance matrix, $C_{ij} = min(t_i, t_j)$ take a vector W_{t_i} with distribution N(0,C) and let $AA^T = C$ and $AZ \sim N(0, I)\ for\ Z = (Z_1, ..., Z_n)^T$. This is a typical result used in finding correlated random numbers [11 ,Ch. 28]. We apply the Cholesky method and find that AZ is a matrix-vector representation of the previous Brownian recursive process. In other words, the random walk construction above is an efficient implementation of AZ.

Let us now look at PCA as a discrete Brownian path with the construction: $\left(W_{t_i}\right) = (a_{i1})\, Z_i + (a_{i2})\, Z_2 + ... + (a_{in})\, Z_n$ where each parentheses represents a vector. Using standard diagnolazations techniques from linear algebra with $AA^T = \Sigma = V \Lambda V^T$ where V is an orthonormal matrix, if $X \sim N(0, \Sigma)$ and $Z \sim N(0, I)$ then generating X as AZ for any A means: $X = a_1 Z_1 + a_2 Z_2 \ ... + a_n Z_n$ with Z_j being independent factors driving the components of X, and A_{ij} being the factor-loading of Z_j on X_i. Finding the linear combinations that capture the variability of the components of X leads us to maximize the variance of $w^T X$, which is given by $w^T \Sigma w$, after a normalization through a constraint of the form $w^T w = 1$. The problem of finding the best factor orthogonal to the previous factor is exactly what principal component analysis is. The approximation error for this is: $E[\|W - \sum_{i=1}^{k} a_i Z_i\|^2]$.

We solve for the first k eigenvalues in $Cv_i = \lambda_i v_i$ normalized for $\left|v_i\right| = 1.$ For example with a 100-step Brownian path with equal time increments $t_{i+1} - t_i = .01$ we have $C_{ij} = min(i,j)/100$. Surprisingly, the magnitude of these eigenvalues decreases rapidly and the explained variability for Z_i requires only a low k.

See 'Code Figures' section at the end of this document wherein we simulate Brownian motion and use PCA (in Python). This toy example shows how few principal components are needed to

describe the data. As one can see in the chart that is produced by said code, around 15 features are enough to contribute ~95% of the data. While the actual build of Brownian motion via PCA is not practical, the reverse process, in which we extract important features, is an interesting note on noise in Brownian motion. In fact, while some may think that the noise entirely masks the information in a random process, the contribution of mu and sigma in defining the process enables us to work backwards and extract important features that contribute to the evaluation of the model, as per our PCA example.



1.4 Geometric Brownian Motion

Geometric Brownian motion for a stochastic process S(t) is an exponentiated Brownian motion. This is a desirable model insofar as it cannot take on negative values and the percentage change $(S(t_n)-S(t_{n-1}))/S(t_{n-1})$ are independent for t_n. Taking W, a standard Brownian motion (which is a one dimensional Wiener process) such that $X \sim BM(\mu, \sigma^2)$ satisfies $dX(t) = \mu dt + \sigma dW(t)$, we set $S(t) = S(0)exp(X(t)) = f(X(t))$ and apply Ito's formula to get

$dS(t) = f'(X(t))dX(t) + \frac{1}{2}\sigma^2 f''(X(t))dt$

$= S(0)exp(X(t))[\mu dt + \sigma dW(t)] + \frac{1}{2}\sigma^2 S(0)exp(X(t))dt = S(t)(\mu + \frac{1}{2}\sigma^2)dt + S(t)\sigma dW(t)$.

The above is the standard form used in mathematical, financial and physical applications.

The solution to the above SDE is $S(t) = S(u)exp([\mu - \frac{1}{2}\sigma^2](t-u) + \sigma(W(t) - W(u)) \quad for\ u < t$.

This gives us the recursive procedure used for simulating values since our increments of W are independent and normally distributed: $S(t_{i+1}) = S(t_i)exp([\mu - \frac{1}{2}\sigma^2](t_{i+1} - t_i) + \sigma\sqrt{t_{i+1} - t_i} * Z_{i+1}$

1.5 Euler-Maruyama and Milstein Methods

The Euler-Maruyama method is a generalization of the Euler method for solving ODEs. It replaces the SDE with a stochastic difference equation that is exact up until step size h. The Euler-Maruyama method follows directly from the discussion above.

For the stochastic process X satisfying the SDE $dX(t) = a(X(t))dt + b(X(t))dW(t),$ a time-discretized approximation to X would be:
$X(t_{i+1}) = X(t_i) + a(X(t_i))[t_{i+1} - t_i] + b(X(t_i))\sqrt{t_{i+1} - t_i} * Z_{i+1}$.

If we use the above and set $h = t_{i+1} - t_i$ we see the typical presentation of the Euler method for ODEs.

For the one dimensional SDE case we have, by definition:
$X(t) = X(0) + \int_0^t a(X(u))du + \int_0^t b(X(u))dW(u)$ which, when using the Euler-Maruyama formula gives $\int_t^{t+h} a(X(u))du \approx a(X(t))h$ and $\int_t^{t+h} b(X(u))dW(u) \approx b(X(t))[W(t+h) - W(t)].$ However, using Ito's formula for $db(X(t)) = b'(X(t))dX(t) + \frac{1}{2}b''(X(t))b^2(X(t))dt$
$= [b'(X(t))a(X(t)) + \frac{1}{2}b''(X(t))b^2(X(t))]dt + b'(X(t))b(X(t))dW(t)$ we apply the Euler approximation to
$b(X(u)) \approx b(X(t)) + (b'(X(t))a(X(t)) + \frac{1}{2}b''(X(t))b^2(X(t)))[u - t] + b'(X(t))b(X(t))[W(u) - W(t)] \quad for\ t \le u \le t + h.$

When we drop the higher order term and now solve $\int_t^{t+h} b(X(u))dW(u)$ we get
$\approx b(X(t))[W(t+h) - W(t)] + \frac{1}{2}b'(X(t))b(X(t))([W(t+h) - W(t)]^2 - h)$ which gives us the Milstein approximation (of higher order than Euler):
$X(i+1) = X(i) + a(X(i))h + b(X(i))\sqrt{h}Z_{i+1} + \frac{1}{2}b'(X(i))b(X(i))h(Z_{i+1}^2 - 1)$. The new term has mean zero and is uncorrelated with the Euler terms and is thus a better approximation scheme.

See the end notes 'Code Figure' section for C++ code that implements both the Euler-Maruyama and Milstein methods in approximating option prices using geometric brownian motion. The details of said code are described in the next section.

1.6 Financial Application

For a stock at time t denoted S(t), consider a call option granting the holder the right to buy the stock at fixed price K at time T in the future. The option payoff is

$(S(T)-K)^+ = max\{0, S(T)-K\}$. The present value of this payoff with a discount factor e^{-rT}, with *r* being the continuously compounded interest rate, is denoted $E[e^{-rT}(S(T)-K)^+]$.

As is known, the Black-Scholes (BS) model describes the evolution of the stock price as a stochastic differential equation $\frac{dS(T)}{S(T)} = r\,dt + \sigma\,dW(t)$ with *W* a standard Brownian motion. The BS model can be interpreted as showing the progression as a percentage change in the stock price as the increments of Brownian motion change. We implicitly describe the interest rate *r* as a proxy for the expected return in a risk-neutral dynamic.

Assume a r.v. *W(T)* is normally distributed with mean 0 and variance T. This is also the distribution of $\sqrt{T}Z$ for *Z ~ N(0,1).* We may therefore present the solution to the stochastic differential equation in terms of the terminal stock price as $S(T) = S(0)exp([r-\frac{1}{2}\sigma^2]T + \sigma\sqrt{T}Z$. Accordingly, the log of the stock price is normally distributed, and the stock price has a lognormal distribution.

The expectation, $E[e^{-rT}(S(T)-K)^+]$ as above, is an integral with respect to the lognormal density. This integral can be evaluated with the standard normal cumulative distribution function Φ as the celebrated BS formula:

$$\mathrm{BS}(S,\sigma,T,r,K) = S\Phi\left(\frac{\log(S/K)+(r+\frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}\right) - e^{-rT}K\Phi\left(\frac{\log(S/K)+(r-\frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}\right)$$

The above shows us that to draw samples of S(T) it is sufficient to draw samples from the standard normal distribution. As such, we can produce a sequence of independent random variables $Z_1, Z_2, Z_3, ..., Z_n$ and estimate $E[e^{-rT}(S(T)-K)^+]$ by the following simple algorithm:

1. Loop for i = 1 to n
2. $S(T) = S(0)exp([r-\frac{1}{2}\sigma^2]T + \sigma\sqrt{T}Z_i$
3. $C_i = e^{-rT}(S(T)-K)^+$
4. $\widehat{C_n} = (C_1 + C_2 + C_3 + ... + C_n)/n$

In the above C is the call price, with $\widehat{C_n}$ being the average call price. Here $\widehat{C_n} \to C \text{ as } n \to \infty \text{ with probability } 1$.

For finite n, at least moderately large, we supplement the point estimate $\widehat{C_n}$ with the sample standard deviation $s_C = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\widehat{C_i}-C_n)^2}$ and attain the confidence interval $\widehat{C_n} \pm z_{\frac{\delta}{2}}\frac{s_n}{\sqrt{n}}$, as mentioned above.

Schematic illustrations are simple enough that basic spreadsheet programming can be easily done. Below is Matlab code using a similar implementation.

```
function Price = BlsMC1(S0,K,r,T,sigma,NRepl)
nuT = (r - 0.5*sigma^2)*T;
siT = sigma * sqrt(T);
DiscPayoff = exp(-r*T)*max(0, S0*exp(nuT+siT*randn(NRepl,1))-K);
Price = mean(DiscPayoff);
```

```
>> S0=50;
>> K=60;
>> r=0.05;
>> T=1;
>> sigma=0.2;
```

```
>> BlsMC1(S0,K,r,T,sigma,1000)
```

One can think of this method as layered transformations: $Z_i \rightarrow S(T_i) \rightarrow C_i$. Although, at the lowest level, we usually start from U(0,1) and transform that to Z_i.

1.7 Path Dependent Simulations

In a scenario where the payoff of a derivative depends explicitly on the values of the underlying at multiple dates we need to sample transitions of the underlying at each time interval from [0,T]. That is, we break it into smaller subintervals to obtain a more accurate approximation to sampling from the distribution at time T. The intuition here is similar to how the binomial tree model converges to the BS equation.

Let the volatility σ depend on the current level of S by generalizing the BS model:
$dS(T) = rS(T)\,dt + \sigma(S(T))S(T)\,dW(t)$.

This equation does not have an explicit solution of the type above. We do not have an exact mechanism for sampling from the distribution of S(T), so instead, we partition [0, T] into *m* subintervals of length $\Delta t = T/m$ and over each subinterval $[t, t+\Delta t]$ we simulate a transition using discretization methods such as the Euler-Maruyama or Milstein method, with examples above.

For each step in the above, we use an i.i.d. r.v. X_i, and repeat the process for the *m* subintervals to find the terminal value S(T) whose distribution approximates the unknown distribution of S(T)

implied by $dS(T) = rS(T)\,dt + \sigma(S(T))S(T)\,dW(t)$, and as *m* becomes larger (and *t* becomes smaller) the approximation converges to the true solution.

Sometimes though, even assuming the dynamics of the BS model, we may need knowledge of intermediate times, such as with Asian options where the payoff depends on the average level of the underlying asset at T. In Asian options, the payoff $(\overline{S} - K)+$ for $\overline{S} = \frac{1}{m}\sum_{j=1}^{m} S(t_j)$, $0 = t_0 < t_1 < \ldots < t_m = T$. Since we need to generate samples of the average $\overline{S}$, we simulate the path and then compute the average along that path. The only substitution we need to make is setting T to $t_{j+1} - t_j$. Examples of pricing Asian options in Matlab will be given later.

2 Efficiency of the Estimators

Bias, computing time, and variance are three considerations in any Monte Carlo simulation. Bias (the expected estimator value difference against the true value being estimated) has already been dealt with to some extent, as the examples given above uses the central limit theorem, with $\lim_{n\to\infty}$, to assert that the standard estimator $(\widehat{C_n} - C)/(\sigma_C/\sqrt{n})$ converges in distribution to the standard normal. In general, we are only interested in unbiased estimators, $E(\widehat{\theta}) = \theta$.

In regard to variance, we can rephrase the CLT to tell us that the error in our simulation estimate is $\widehat{C_n} - C \approx N(0, \sigma_C/\sqrt{n})$, meaning that the error on the left has approximately the distribution on the right. As such, in comparing two estimators we should prefer the one with lower variance, as is somewhat intuitive. As such, we address variance reduction techniques in an effort to speed-up the convergence.

One more important point in efficiency of estimators is that of model *discretization error*. This occurs when an exact model is not known and the result of time-discretization of a continuous-time dynamic model is inconsistent (or slightly biased) with respect to the underlying stochastic process. When exact sampling of the continuous-time dynamic is infeasible this error will proliferate. While this error may vanish as shorter Δt are taken, the computational encumbrance may not be a satisfactory trade-off.

An example of this is pricing an Asian option with the continuous average (rather than the discrete average discussed above). Since we cannot calculate $\overline{S}$ exactly we must approximate the continuous average. As such, variance reduction techniques are needed to limit the computational burden.

A general measure of estimator performance that balances bias and variance is the mean square error. Explicitly, if $\hat{a}$ is an estimator of a then $MSE(\hat{a}) = E[(\hat{a} - a)^2]$, as explained above in Section 0.

3 Pseudo-Random Number Generators and Transforms

Before developing the variance reduction techniques, we must mention some approaches to generating pseudo-random numbers. There are three classes of generators: linear congruential generators (LCG), linear feedback shift registers (LFSR), and cellular automata (CA). And while an LFSR may be thought of as nothing more than a recurrence relation, we will only briefly describe the first of these possible approaches, as the others are too far off topic.

A good PRNG (pseudo-random number generator) will have independent uniform numbers. The generator should have a large period, be easily reproducible, and be efficient.

A pseudorandom bit generator is said to pass all polynomial-time statistical tests if no polynomial-time algorithm can correctly distinguish between an output sequence of the generator and a truly random sequence of the same length with probability significantly greater than .5. [6]

While there are several statistical tests that must be passed to verify that a sequence is not random, Mauerer's universal test encompasses all the others. The idea is that it should not be possible to compress the output sequence without loss of information. We take a bit sequence *s* and partition it into non-overlapping L-bit blocks and then a log likelihood function is run on the data [12][20]. The output is a measure of information, thus, similar to regular entropy measures, we are looking for an optimal measure of .5. Effectively, we are testing the information in the data set.

For an effective generator we require that each random number U_i is uniformly distributed between 0 and 1, and that each u_k is mutually independent. We need to create a pseudo random number generator that produces a finite sequence that obeys the above and is also extremely fast. While there are regular statistical tests to test for independence, computational considerations with respect to bit size is also extremely important.

A linear congruential generator uses modular arithmetic to create a seemingly random sequence of numbers with the relation $x_{i+1} = ax_i \bmod m$ and $u_{i+1} = x_{i+1}/m$. For multiplier *a* and modulus *m* we take an initial seed x_0 and generate our sequence. If a sequence includes all values less than *m* it is said to have a full period. The factors that distinguish the specific versions of the algorithm include: period length, reproducibility, speed, portability, and randomness.

To reach a full period, let *m* be prime, and choose values such that $a^{m-1} - 1$ is a multiple of *m,* and $a^j - 1$ is not a multiple of *m* for *j* = 1,...,*m*-2 [6]. While there are many good choices for

parameters, implementation still needs to be suitable for a computer of *k* bits, without overflow or errors.

The major drawback to LCGs is, firstly, the restrictive size of moduli unless specific algorithms are used, and, secondly, it is known that as the dimensionality grows a distinguishable lattice structure forms in d-dimensions.

On a separate note, we mention that some techniques to transform uniformly random numbers to any other distribution require computational consideration as well.

To convert to a Poisson distribution let $X = F^{-1}(U)$, and $P\{X \leq x\} = P\{F^{-1}(U) \leq x\} = P\{U \leq F(x)\} = (x)$ and for $(x) = 1 - e^{-x/\theta}$ we have $X = -\theta log(1-U) = -\theta log(1-U)$ where *U* and 1 - *U* have the same distribution. This transform, as well as many others, are straightforward, however, other techniques, such as the Box-Muller method, which converts uniformly distributed numbers to a Gaussian distribution, may be simple but slightly slower. Further exploration is warranted, but not directly our discussion here.

4 Several Important and Common Variance Reduction Techniques

A. Control Variates

The method of control variates is one of the most powerful methods in reducing error. It takes the error of a known quantity and uses it to reduce the error of an unknown quantity.

For Y_i , the output of a simulation, such as the discounted payoff of a derivative on a simulated path, we wish to estimate the true mean E[Y_i], looking for $E[\overline{Y_i}]$. We have an estimator of the sample mean $\overline{Y} = (Y_1 + ... + Y_n)/n$ that is unbiased and converges with probability 1 as $n \to \infty$. However, we can speed up this process by choosing an X_i with the same initial bias as Y_i and use the known expectation E[*X*] to calculate the pair (X_i, Y_i).

For each iteration we calculate the following for any fixed b:

$$Y_i(b) = Y_i - b(X_i - E[X])$$

We then compute the sample mean: $\overline{Y}(b) = \overline{Y} - b(\overline{X} - E[X]) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - b(X_i - E[X])$

This process is a control variate estimator with the known value X as the control. This is unbiased because $E[\overline{Y}(b)] = E[\overline{Y} - b(\overline{X} - E[X])] = E[\overline{Y}] = E[Y]$.

The reason why this process has a lower variance is that given

$$Var[Y_i(b)] = Var[Y_i - b(X_i - E[X])] = \sigma_Y^2 - 2b\sigma_X\sigma_Y\rho_{XY} + b^2\sigma_X^2 = \sigma^2(b)$$

where the third term is less than the second we have reduced the variance. To achieve this we can use the optimal *b* = Cov[X,Y]/Var[X]. As such, the ratio of the variance of the controlled to uncontrolled estimator is $1 - \rho_{XY}^2$.

Two important observations here include that the effectiveness of a control variate is determined by the correlation between X and Y, and, secondly, that the variance reduction factor $1/(1 - \rho_{XY}^2)$ increases sharply as $|\rho_{XY}|$ approaches 1.

A typical example of using a control variate is pricing an Asian option via the arithmetic average using the control variate of the same option priced via the geometric average. The former does not have a closed form which is known, while the latter is straightforward. As should be clear, the two cases are highly correlated. The below code is from [2], and was the basis for a large experiment incorporating multiple tests, as explained below, each of which was our own code. Furthermore, all subsequent code here and in the accompanying colab notebooks is our own.

In (1) below, we price an Asian option using the closed form for a geometric Asian option. In (2), we use Monte Carlo Control Variates, as explained above. Initially, we create n samples of asset paths, then we create a payoff vector of strikes against the geometric Asian option. We run the above calculations with n replications, using Monte Carlo, and output the normfit of our process.

1.
```
function P = GeometricAsian(S0,K,r,T,sigma,delta,NSamples)
dT = T/NSamples;
nu = r - sigma^2/2-delta;
a = log(S0)+nu*dT+0.5*nu*(T-dT);
b = sigma^2*dT + sigma^2*(T-dT)*(2*NSamples-1)/6/NSamples;
x = (a-log(K)+b)/sqrt(b);
P = exp(-r*T)*(exp(a+b/2)*normcdf(x) - K*normcdf(x-sqrt(b)));
```

2.

```
function [P,CI] = AsianMCGeoCV(S0,K,r,T,sigma,NSamples,NRepl,NPilot)
% precompute quantities
DF = exp(-r*T);
GeoExact = GeometricAsian(S0,K,r,T,sigma,0,NSamples);
% pilot replications to set control parameter
GeoPrices = zeros(NPilot,1);
AriPrices = zeros(NPilot,1);
for i=1:NPilot
    Path=AssetPaths(S0,r,sigma,T,NSamples,1);
    GeoPrices(i)=DF*max(0,(prod(Path(2:(NSamples+1))))^(1/NSamples) - K);
    AriPrices(i)=DF*max(0,mean(Path(2:(NSamples+1))) - K);
end
MatCov = cov(GeoPrices, AriPrices);
c = - MatCov(1,2) / var(GeoPrices);
% MC run
ControlVars = zeros(NRepl,1);
for i=1:NRepl
   Path = AssetPaths(S0,r,sigma,T,NSamples,1);
   GeoPrice = DF*max(0, (prod(Path(2:(NSamples+1))))^(1/NSamples) - K);
   AriPrice = DF*max(0, mean(Path(2:(NSamples+1))) - K);
   ControlVars(i) = AriPrice + c * (GeoPrice - GeoExact);
end
[P,aux,CI] = normfit(ControlVars);
```

We found that testing without the control variate for 10^3 to 10^7 number of runs gave:

```
>> AsianOpt

call =

    5.2227
    5.3225
    5.3468
    5.3520
    5.3552


error =

    0.0158
    0.0049
    0.0015
    0.0005
    0.0002
```

```
CI =

    5.0609    5.3845
    5.2713    5.3736
    5.3306    5.3631
    5.3469    5.3572
    5.3536    5.3568
```

where CI is the confidence interval as explained above and in the code. The output for the code with the control variate is:

```
>> AsianOptCV

ellCall =

    5.3556


CI =

    5.3533    5.3580
```

where we can see that the CI for a calculated 1000 draws is equivalent to 10^7 in the crude Monte Carlo approach. We have thus reduced the order of magnitude by 4!

(In an effort to save space, the actual script run to test this is an accompanying slide set. Although the code for the Asian option is the same as above.)

B. Antithetic Reduction

Antithetic reduction reduces the variance by introducing negative dependence among the replications. The typical approach uses the observation that if U is uniformly distributed then so is 1-U. As such, we can trace two simultaneous paths without much extra work and have them balance out the variance.

In general, $\overline{X}(n) = \frac{1}{2m}\sum_{j=1}^{2m} X_j = \frac{1}{m}\sum_{j=1}^{m} Y_j = \overline{Y}(m)$ $for\ n = 2m.$ Here the two estimators have the same expected value yet for $Y = \frac{X_i+X_j}{2}$ we have $Var(Y) = \frac{1}{4}(\sigma^2 + \sigma^2 + 2Cov(X_i, X_j)$ where in the case of iid rv Cov =0 and then $Var(Y) = \frac{\sigma^2}{2}$. And for $Cov(X_i, X_j)$ <0 we have $Var(Y) < \frac{\sigma^2}{2}$. Thus our approach is to introduce negative correlation. So, as above, we chose vectors U_i and 1-U_i which have the same distribution, but still have an induced negative correlation. The below code uses randn and -randn as vectors, averaging the iteration, reducing the Big-O (although U = randn and V = 1 - U, may yield slightly better results).

```
function [Price, CI, dif] = BlsMCAV(S0,K,r,T,sigma,NPairs)

Payoff1 = max( 0 , S0*exp((r - 0.5*sigma^2)*T+sigma * sqrt(T)*randn(NPairs,1)) - K);
Payoff2 = max( 0 , S0*exp((r - 0.5*sigma^2)*T+sigma * sqrt(T)*(-randn(NPairs,1))) - K);
DiscPayoff = exp(-r*T) * 0.5 * (Payoff1+Payoff2);
[Price, VarPrice, CI] = normfit(DiscPayoff);
dif = (CI(2)-CI(1))/Price;

end
```

```
S0=50;
K=55;
r=0.05;
T=5/12;
sigma=0.2;
Call = blsprice(S0,K,r,T,sigma);
[CallMC1, CI1, dif1] = BlsMC2(S0,K,r,T,sigma,1000000);
[CallMC2, CI2, dif2] = BlsMCAV(S0,K,r,T,sigma,1000000);

Call
CallMC1
CallMC2
```

```
Call =

    1.1718


CallMC1 =

    1.1745


CallMC2 =

    1.1675
```

In the above we have an error of .0027 and .0043, respectively. This technique is not that strong and not always effective, as can be seen above [2]. The crude Monte Carlo technique (MC1) is closer to the true BS price than the Antithetic variate (MC2).

C. Stratified Reduction

The concept of stratified reduction is to take random numbers from various *strata* in order to ensure that the sample is effectively testing each range and not clustering in any way.

We now prove that stratified sampling is unbiased:

Partition D into mutually exclusive regions D_j, $for\, j = 1, ..., J$, each of which will be our *strata*.

Assume $f(x)$ is some function with probability $p(x)$, to estimate $\mu = \int_D f(x)p(x)dx$ we let $\omega_j = P(X \in D_j)$, with $p_j(x)$ as the conditional density of X given $X \in D_j$. Assuming we know the size of each strata ω_j and how to sample $X \sim p_j$ for j = 1,..., J, we can create the stratified sampling estimate of $\mu_{strata} = \sum_{j=1}^{J} \frac{\omega_j}{n_j} \sum_{i=1}^{n_j} f(X_{ij})$

With $E(\mu_{strata}) = \sum_{j=1}^{J} \omega_j E(\frac{1}{n_j} \sum_{i=1}^{n_j} f(X_{ij})) = \sum_{j=1}^{J} \omega_j \int_{D_j} f(x)p_j(x)dx = \sum_{j=1}^{J} \int_{D_j} f(x)p(x)dx = \int_D f(x)p(x)dx = \mu$

The idea of the above is that we are showing that the stratified sampling estimate covers all of the mutually exclusive regions and is therefore an unbiased estimator; and while one can show that there is a reduction in variance as well [13], the clear idea of uniform sampling is enough to argue strongly for this method. It is known [2], that stratified sampling is an effective method in lesser dimensions, however, as the dimensionality increases the technique becomes increasingly weaker. Below we show an example comparing several techniques in low dimensional cases of integrals.

D. Quasi-Monte Carlo

Quasi-Monte Carlo is a low discrepancy method which makes no attempt to use random numbers. Instead, it increases accuracy by choosing points that are evenly distributed, often using a sequence such as the Halton sequence which is maximally balanced. This method often requires formulating the problem as an integral [1, Ch.5], and, as such, is not as effective in certain applications.

As mentioned earlier, this technique can potentially be far more powerful than other approaches to Monte Carlo, in that the convergence rate can reach $O(\frac{1}{n})$.

The idea is that instead of searching through a grid to check d dimensional space, whose points would grow very quickly as we try to fill in a hypercube, instead we choose a low discrepancy sequence that guarantees uniformity over bounded-length extensions of an initial segment of the sequence. Glasserman [1, pg 283] defines discrepancy as "the supremum over errors in integrating the indicator function of A (a collection of Lebesgue measurable subsets of $[0,1)^d$) using the points $x_1, ...x_n$."

A typical foundation in Quasi-Monte Carlo is the Van der Corput Sequence. It takes a base b and sets $k = \sum_{j=0}^{\infty} a_j(k) b^j$ and then uses the transformation $\psi_b(k) = \sum_{j=0}^{\infty} \frac{a_j(k)}{b^{j+1}}$. This procedure takes a number k to a base b and then flips the coefficients of k about the base-b "decimal" point to get a base-b fraction. Building on this, the Halton sequence takes two coprime bases and builds a set of points to test in a Monte Carlo simulation. The main drawback with this method,similar to other Monte Carlo methods, is that ineffective implementation in high dimensional space can lead to long monotone sequences.

As a test, the accompanying chart displays the results comparing Quasi Monte Carlo methods using a Van der Corput sequence against stratified and antithetic variance reduction. The functions tested are the same as in the Introduction section (y =x^2 and an integral which gives the Euler-Mascheroni gamma constant). As can be seen, Quasi Monte Carlo methods are often significantly better, although through many tests there are occasionally outliers.

STRATIFIED	Num Trials	Func1	Func1 Error	Func2	Func2 Error
0	1,000	0.3351230	0.0017897	0.5768470	0.0003680
1	10,000	0.3339150	0.0005820	0.5770760	0.0001400
2	100,000	0.3335440	0.0002110	0.5771800	0.0000360
3	1,000,000	0.3333000	0.0000332	0.5772300	0.0000140

ANTITHETIC	Num Trials	Func1	Func1 Error	Func2	Func2 Error
0	1,000	0.33710	0.00376	0.57668	0.00054
1	10,000	0.33408	0.00074	0.57695	0.00027
2	100,000	0.33360	0.00027	0.57729	0.00008
3	1,000,000	0.33345	0.00012	0.57722	0.00000

Quasi-M.C.	Num Trials	Func1	Func1 Error	Func2	Func2 Error
0	1,000	0.32222	0.01112	0.57912	0.00190
1	10,000	0.33211	0.00122	0.57742	0.00021
2	100,000	0.33316	0.00017	0.57727	0.00005
3	1,000,000	0.33331	0.00002	0.57722	0.00001

Table 1. This table summarizes the results of testing (1) $y=x^2$ and (2) the Euler-Mascheroni function using 3 different variance reduction techniques. The code can be found in the accompanying colab notebooks. The error function here is the distance from the correct solution to the Monte Carlo output for n trials. The table shows that an average, the Quasi-Monte Carlo method is the most effective.

6 Application

In order to test our variance reduction methods in a “real-world” setting outside of integrals and finance, we created a genetic algorithm that will be explained below briefly. We use this to measure computational run-time of Monte Carlo applications against a known algorithm that can be extremely time consuming.

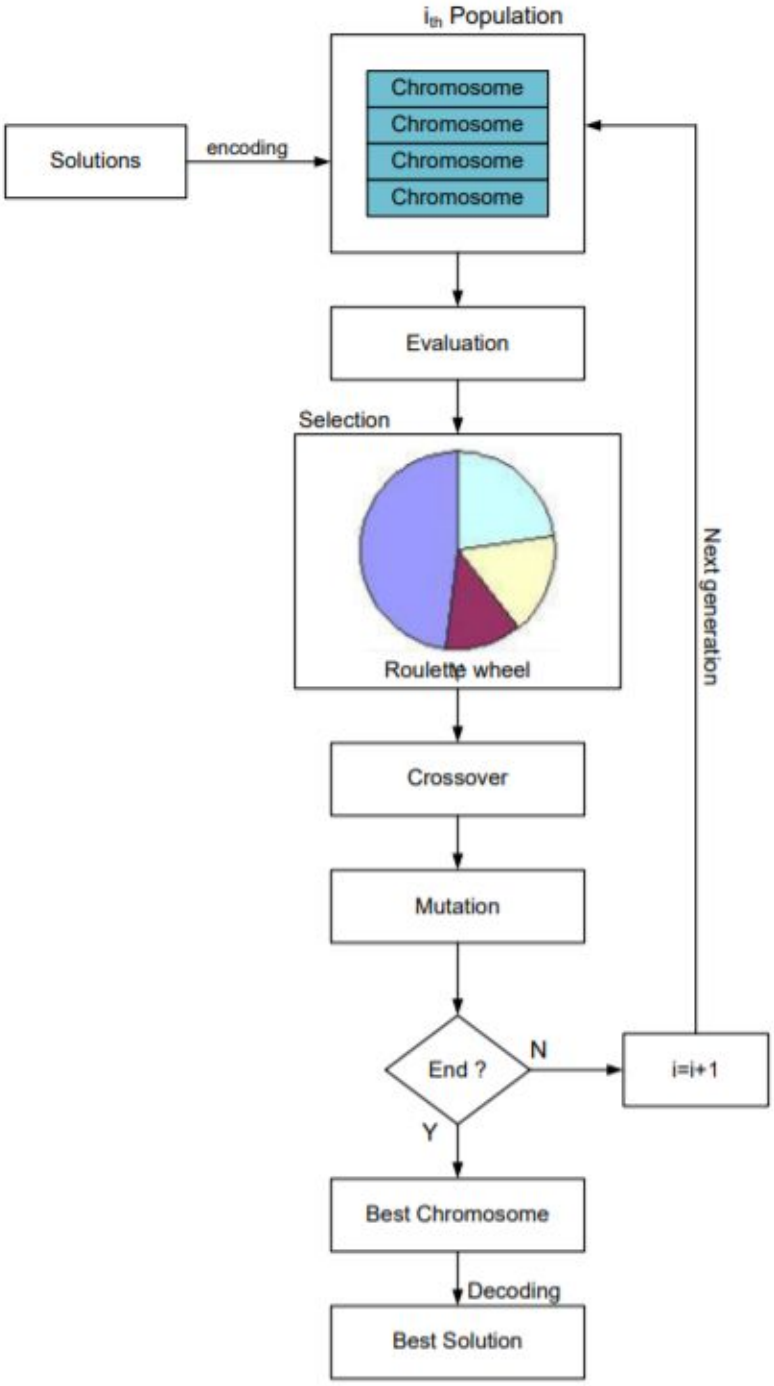
In general, a genetic algorithm follows the path given in the accompanying chart below. Initially we populate a possible solution with random test values called chromosomes. We evaluate the possible solutions with a fitness function, then we use a sieve to extract parts of the population based on the feature function, one of the most effective methods being a roulette wheel. We take the outcome of the roulette wheel, mutate and recombine the remaining chromosomes based on probabilistic evaluations, and then, assuming we have not found a solution to our problem, we run the program again, utilizing the updated chromosomes, slowly zeroing in on the solution.

This process mimics an “evolutionary growth” process, as a theoretical learning model. We regularly employ genetic algorithms to solve NP (non-deterministic polynomial) problems. We created an algorithm to solve *subset-sum/knapsack*, an NPC (non-deterministic polynomial complete) problem, listed as one of Karp’s 21[1], reducible to SAT-3 [14]. The mathematical

[1]Karp’s original paper lists 21 NP problems. He shows that each of the “categories” of problems are equivalent to showing the “basis” problem, SAT-3. SAT-3 stands for satisfiability of 3 variables. It is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. Karp shows that solving one NP problem is

justification for the optimization of *subset-sum/knapsack* has been thoroughly evaluated and proven here [16].

There are several mathematical approaches to categorizing and constructing such models, irrespective of their NPC nature [15], but we will focus exclusively on Monte Carlo methods.



equivalent to solving all of them, thus, the question of P = NP need only be answered once, and existence (or non-existence) would be true for all NP problems.

Subset Sum is a combinatorial optimization problem wherein we are given values and asked to find whether some subset sums to s exactly. The approach to coding the problem is discussed in a video presentation referenced in the last section and the code itself is in a github repository, referenced in the bibliography as well, both of which are periphery to our goal of assessing Monte Carlo methods.

We tested a list of k = 26 random numbers (thus we are mimicking 26 "weights" in our knapsack), with 2^{26} possible combinations of base-2 representations (more than this seemed to crash Google Colab). We used a test base of 25 runs and found that the basic algorithm ran with a mean time as displayed in the chart below. The feature selection tool used is a roulette wheel, the mutation rate starts at 4%, the recombination probability is 8%, there were 20 chromosomes targeted, and 100,000 trials was the max-bound per iteration of the algorithm, all in line with accepted standards [17].

Our Monte Carlo addition utilizes the idea of Control Variates mentioned above. Since we know the target value, we use the same parameters as our regular algorithm, but we instead test a function called *MonteCarlo* instead of *recombine*.

The *MonteCarlo* function takes two random values and checks to see where they are in relation to our target t. If they are on opposite sides of t, it reflects one of the values through t, thereby reducing the variance of those values with respect to the intended solution.

We modelled our approach on Control Variates (only conceptually, as can be seen from the code, we did not employ a specific case of it), as well as a conceptual application of Conway's approach to unique factorization of the Hurwitz integers (of quaternions), in which unit-migration, recombination, and metacommutation are here interpreted in a lower dimensional space [18].

	Recombine Function	Monte Carlo Function
Average of cases where solution found	42.86781383	41.04172173
Average of cases where no solution was found	69.6723589	69.93710618

Table 2. This table shows the number of seconds Google colab took to search for a solution under two different scenarios. The first column uses the *recombine* function that does a typical genetic algorithm recombination, while the second column uses our own Monte Carlo function.

Our results indicate that recombination was essentially equivalent to Monte Carlo with variance reduction. While it may seem that the Monte Carlo method did not speed up the convergence to a solution, it is quite surprising that it was equivalent to a known technique that is a staple of all genetic algorithms. That is, the point of recombination is to introduce randomness which fuels the advance toward a solution without plateauing; it seems that "enough" Monte Carlo runs with a variance reduction has done the same thing in equivalent time!

6 Final Notes and Conclusion

There are several other variance reduction techniques, some of which we tested and ran; however, given the nature and requirements of this paper and the amount of space each piece of code takes up, we will leave them in the slides (and the colab notebook) and only mention that the positive gains in reducing variance were often less significant than control variates and quasi-Monte Carlo, but still quite high.

Other avenues to explore include using Monte Carlo methods to test the randomness of cryptographic systems. That is, an encrypted message should have randomness throughout, if there were a hidden message in a jpeg file, we could use Monte Carlo methods to assess whether the data has true information or includes random encrypted messages (although there are already some techniques developed outside this framework which are relatively fast).

Another place to apply Monte Carlo methods is the field of computational group theory, in which search methods may benefit from the variance reduction methods.

A third direction to explore further is that of optimizing Latin Hypercube methods. This approach was not mentioned in this paper, as we were unsuccessful at creating an effective method (however it is still in the colab notebook). Our approach was to apply the norm of the Heegner number and test for optimal coverage of the hyper-plane. While this did not yield effective results, testing other planar representations, such as expressions of the various crystallography groups could possibly bring better results.

In summary, Monte Carlo techniques are extremely powerful and relatively fast to run for d-dimensional problems. The issues that come up in Monte Carlo, such as a large variance and discretization error, can be fixed with many of the techniques mentioned herewithin. While closed form expressions may be difficult to find, unbiased estimators can be created with the error reduction techniques; solutions within predetermined confidence intervals are both measurable and reliable.

Code Figures

PCA Test

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
import math
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

def brownianProcess(mu, sigma, time):
  S = []
  S.append(100)
  for i in range(time):
    val = S[-1]*np.exp((mu-(sigma**2)/2)*(1/time)+sigma*math.sqrt(1/time)*np.random.normal(0,1))
    S.append(val)
  return S

def brownianVectors(mu, sigma, time, numVectors):
 df = pd.DataFrame()
 for  i in range(numVectors):
   a = brownianProcess(mu,sigma,time)
   df[len(df.columns)] = a
 return df

Y = brownianVectors(.3,.3,100,100)

pca = PCA(random_state=42)
X_pca = pca.fit_transform(StandardScaler().fit_transform(Y))
#Amount of variance for each component normalized (sums to 1)
pca.explained_variance_ratio_

#cumulative variance plot
fig, ax = plt.subplots(figsize=(8, 8))
ax.plot(np.cumsum(pca.explained_variance_ratio_))
ax.set( xlabel="Component", ylabel="Percent of Explained variance", title="Cumulative Variance",
ylim=(0, 1))
```

Euler-Maruyama and Milstein BS code

```
default_random_engine generator;
normal_distribution<double> distribution(0, 1);

int main()
{
        double S = 12;
        double K = 15;
        double T = .5;
        double r = .05;
        double sigma = .25;
        double h = .01;
        int numSamples = 10000;
```

```
        double* output1 = filledValues_CalleulerM(r, sigma, T, S, h, numSamples, K);
        double* output2 = filledValues_PuteulerM(r, sigma, T, S, h, numSamples, K);
        cout << "EM Call: " << callPrice_eulerM(output1, numSamples) << endl;
        cout << "EM Put: " << putPrice_eulerM(output2, numSamples) << endl;
        cout << "BS Call: " << BS_CallPrice(r, S, K, T, sigma) << endl;
        cout << "BS Put: " << BS_PutPrice(r, S, K, T, sigma) << endl;
}

double* filledValues_CalleulerM(double r, double sigma, double T, double S, double h, int numSamples,
double K) {
        double* val = new double[numSamples];
        for (int i = 0; i < numSamples; i++)
        {
                *(val + i) = regCallVal_eulerM(r, sigma, T, S, h, K);
        }
        return val;
}

double* filledValues_PuteulerM(double r, double sigma, double T, double S, double h, int numSamples,
double K) {
        double* val = new double[numSamples];
        for (int i = 0; i < numSamples; i++)
        {
                *(val + i) = regPutVal_eulerM(r, sigma, T, S, h, K);
        }
        return val;
}

double eulerMaruyama(double r, double sigma, double T, double S, double h) {
        double count{ 0 };
        while (count <= T) {
                count += h;
                S = S + (r)* S*h + (sigma)* S*sqrt(h)*distribution(generator);
        }
        return S;
}

double regCallVal_eulerM(double r, double sigma, double T, double S, double h, double K) {
        double X = eulerMaruyama(r, sigma, T, S, h);
        if (X - K > 0) {
                return exp(-r * T)*(X - K);
        }
        else return 0;
}

double regPutVal_eulerM(double r, double sigma, double T, double S, double h, double K) {
        double X = eulerMaruyama(r, sigma, T, S, h);
        if (K-X > 0) {
                return exp(-r * T)*(K-X);
        }
        else return 0;
}

double callPrice_eulerM(double *val, int numSamples) {
        double total = 0;
        for (int i = 0; i < numSamples; i++)
```

```
        {
                total += *(val + i);
        }
        return total / numSamples;
}

double putPrice_eulerM(double *val, int numSamples) {
        double total = 0;
        for (int i = 0; i < numSamples; i++)
        {
                total += *(val + i);
        }
        return total / numSamples;
}

double BS_CallPrice(double r, double S, double K, double T, double sigma) {
        double d1 = (log(S / K) + (r + .5*sigma*sigma)*T) / (sigma*sqrt(T));
        double d2 = (log(S / K) + (r - .5*sigma*sigma)*T) / (sigma*sqrt(T));
        double Nd1 = (1 + erf(d1 / sqrt(2))) / 2.0;
        double Nd2 = (1 + erf(d2 / sqrt(2))) / 2.0;

        double val = S * Nd1 - K * exp(-r * T)*Nd2;
        return val;
}

double BS_PutPrice(double r, double S, double K, double T, double sigma) {
        double d1 = -((log(S / K) + (r + .5*sigma*sigma)*T) / (sigma*sqrt(T)));
        double d2 = -((log(S / K) + (r - .5*sigma*sigma)*T) / (sigma*sqrt(T)));
        double Nd1 = (1 + erf(d1 / sqrt(2))) / 2.0;
        double Nd2 = (1 + erf(d2 / sqrt(2))) / 2.0;

        double val = K * exp(-r * T)*Nd2- S * Nd1;
        return val;
}

double* filledValues_CallMilstein(double r, double sigma, double T, double S, double h,int numSamples,
double K) {
        double* val = new double[numSamples];
        for (int i = 0; i < numSamples; i++)
        {
                *(val + i) = regCallVal_Milstein(r, sigma, T, S, h, K);
        }
        return val;
}

double* filledValues_PutMilstein(double r, double sigma, double T, double S, double h, int numSamples,
double K) {
        double* val = new double[numSamples];
        for (int i = 0; i < numSamples; i++)
        {
                *(val + i) = regPutVal_Milstein(r, sigma, T, S, h, K);
        }
        return val;
}

double Milstein(double r, double sigma, double T, double S, double h) {
```

```
        double count{ 0 };
        while (count <= T) {
                count += h;
                S = S + (r * S*h) + (sigma)* S*distribution(generator)*sqrt(h) +
                        .5*sigma*sigma*S*(pow((distribution(generator))*sqrt(h), 2) - h);

        }
        return S;
}

double regCallVal_Milstein(double r, double sigma, double T, double S, double h, double K) {
        double X = Milstein(r, sigma, T, S, h);
        if (X - K > 0) {
                return exp(-r * T)*(X-K);
        }
        else return 0;
}

double regPutVal_Milstein(double r, double sigma, double T, double S, double h, double K) {
        double X = Milstein(r, sigma, T, S, h);
        if (K - X > 0) {
                return exp(-r * T)*(K-X);
        }
        else return 0;
}

double callPrice_Milstein(double *val, int numSamples) {
        double total = 0;
        for (int i = 0; i < numSamples; i++)
        {
                total += *(val + i);
        }
        return total / numSamples;
}

double putPrice_Milstein(double *val, int numSamples) {
        double total = 0;
        for (int i = 0; i < numSamples; i++)
        {
                total += *(val + i);
        }
        return total / numSamples;
}
```

```
double S = 12;
double K = 15;
double T = .5;
double r = .05;
double sigma = .25;
double h = .01;
int numSamples = 10000;

Euler-Maruyama Call: 0.156445
Euler-Maruyama Put: 2.75039
BS Call: 0.153806
```

```
BS Put: 2.78345

Call_Milstein: 0.143618
Put_Milstein: 2.78344
Call_BS: 0.153806
Put_BS: 2.78345
```

References and Bibliography

Github repository: https://github.com/MatthewFried/Masters_Thesis
Video explaining colab code: https://youtu.be/DFJ8lUrajWA

1. Monte Carlo Methods in Financial Engineering, Paul Glasserman, Springer, 2000
2. Numerical Methods in Finance and Economics, Paolo Brandimarte, Wiley, 2006
3. Financial Modelling: Theory, Implementation, and Practice with Matlab Source, Kienitz and Wetterau, Wiley, 2012
4. Principles of Financial Engineering, Kosowski, Academic Press, 2014
5. Handbook of Monte Carlo Methods, Kroese, Wiley, 2011
6. Handbook of Applied Cryptography, Menezes, 1997, CRC Press
7. Introduction to reducing variance in Monte Carlo simulations, Karl Sigman, 2007 http://www.columbia.edu/~ks20/4703-Sigman/4703-07-Notes-ATV.pdf
8. Numerical Analysis 2nd edition, Timothy Sauer, Pearson, 2012
9. Über stochastische Asymptoten und Grenzwerte, E. Slutzky, Metron, 1925
10. http://ludkovski.faculty.pstat.ucsb.edu/bmNotes.pdf, Thrm 2
11. Financial Modeling, Benninga, MIT Press, 1997
12. https://link.springer.com/article/10.1007/BF00193563
13. https://statweb.stanford.edu/~owen/mc/Ch-var-basic.pdf
14. https://people.eecs.berkeley.edu/~luca/cs172/karp.pdf
15. An Introduction to Sequential Dynamical Systems, Mortveit, Henning, Reidys, Christian; 2008, Springer
16. http://www.muehlenbein.org/mathanal02.PDF
17. There is no hard and fast rule for many of these parameters (outside of experience). The bounds suggested are in line with suggestions given by Dr. Robert Goldberg of Queens College, an expert in the field of genetic algorithms.
18. On Quaternions and Octonions, John Horton Conway, Derek Smith, A.K. Peters, 2003
19. Elementary Stochastic Calculus with Finance in View, Mikosch, World Scientific, 1998
20. https://crypto.ethz.ch/publications/files/Maurer92a.pdf