Monte Carlo Method

QF607 Numerical Methods

Zhenke Guan
zhenkeguan@smu.edu.sg

Outline

- MC Method Theory
- Random Number Generating Method
    - Middle Square Method
    - Congruential Generators
    - Mersenne Twister
- Uniform to Normal Distribution
    - Central Limit Theorem
    - Box Muller Method
    - Acceptance-rejection method
    - Inverse Transformation Method
- Discretization Scheme
    - Euler Scheme
    - Milstein Scheme
- Multi-dimensional MC
- Variance Reduction Techniques
- Quasi Monte Carlo

Martingality of Discounted Derivative Price

- Risk neutral pricing theory tells us that the discounted price of a derivative instrument is a martingale:

$$\frac{V_t}{B_t} = \mathbb{E}_{\mathbb{Q}}\left[\frac{V_T}{B_T}\right] \tag{1}$$

  where $\mathbb{Q}$ is the risk neutral measure and B_t is the price of a money market account starting at $B_0 = 1$.

- In our simplistic case of constant interest rate, $B_t = e^{rt}$.

- Note that (1) holds no matter B_t is stochastic or deterministic.

- Pricing derivative $\Longleftrightarrow$ calculating expectation

Expectation Through Monte Carlo

- Expectation to be calculated:

$$\frac{V_t}{B_t} = \mathbb{E}_{\mathbb{Q}}[h(\mathbf{X})] \tag{2}$$

  - $\mathbf{X}$ is the random **vector** involved in determining the payoff h.

- Monte Carlo offers a numerical method to approximate the expectation
- It offers a generic framework to price a wide range of derivative products
- Very useful when the dimension of the problem is high

Law of Large Numbers (LLN)

Let $\mathbf{X}_1$, $\mathbf{X}_2, \ldots$, $\mathbf{X}_n$ be independent trial process, with finite expected value $\mu = \mathbb{E}[h(\mathbf{X}_i)]$ and finite variance $v = V(h(\mathbf{X}))$. Let

$$\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} h(\mathbf{X}_i) \tag{3}$$

Then for any $\epsilon > 0$,

$$P(|\hat{\mu}_n - \mu| \leq \epsilon) \to 1 \quad \text{as} \quad n \to \infty. \tag{4}$$

- The average of the results obtained from a large number of experiments should be close to the expected value, and will tend to become closer as more experiments are performed
- $\hat{\mu}_n$ is **unbiased**: $\mathbb{E}[\hat{\mu}_n] = \frac{1}{n} E[\sum_{i=1}^{n} h(\mathbf{X}_i)] = \mu$
- On average the sample mean and variances are equal to their population counterparts. That is, over repeated samples, you will get the correct answer on average.

Monte Carlo Algorithm

Therefore, to calculate the expectation of $h(\mathbf{X})$ we just need to generate independent trail processes and take the average — **Monte Carlo simulation**. The overall algorithm for Monte Carlo is really simple:

Algorithm 1 $\hat{\mu}_n = \text{MC}\ (h)$

```
1: $s = 0$
2: for $i = 1$ to $n$ do
3:     Generate $\mathbf{X}_i$
4:     $h_i = h(\mathbf{X}_i)$
5:     $s \mathrel{+}= h_i$
6: end for
7: $\hat{\mu}_n = \text{s} \ / \ n$
8: return $\hat{\mu}_n$
```

Monte Carlo As Integrator

- An integral $\int_0^1 f(x)dx$ is an expectation $\mathbb{E}[f(x)]$ with uniformly distributed from 0 to 1 ($\mathcal{U}(0,1)$):

$$\int_0^1 f(x)dx = \int_0^1 f(x)p(x)dx = \mathbb{E}[f(x)] \tag{5}$$

because $p(x) = 1$ for $\mathcal{U}(0,1)$

- To integrate the interval $[a, b]$

$$\mathbb{E}[f(x)] = \int_a^b f(x)p(x)dx = \int_a^b f(x)\frac{1}{b-a}dx = \frac{1}{b-a}\int_a^b f(x)dx \tag{6}$$

So $\int_a^b f(x)dx = (b-a)\mathbb{E}[f(x)]$ for $x \sim \mathcal{U}(a, b)$

Estimating Probability Using Monte Carlo

- We can estimate probability using Monte Carlo by representing them as expectations.
- In particular, $P(\mathbf{X} \in A) = \mathbb{E}[I_A(\mathbf{X})]$ where

$$I_A(\mathbf{X}) = \begin{cases} 1 & \text{if } \ \mathbf{X} \in A \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

- Example: knowing that a uniformly drawn random point in a 2×2 square has the probability $\frac{\pi}{4}$ to fall inside a the unit circle inscribed within the square, we have

$$P = \frac{\pi}{4} = \mathbb{E}[I_A(x)] \tag{8}$$

where A represent x is inside the circle. We can use Monte Carlo to estimate the right hand side, thus obtain an estimate of π.

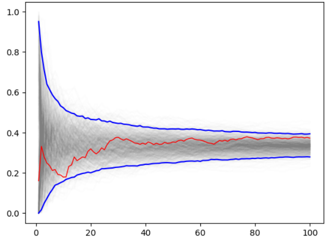
Example: Calculate Integral

Example: Solve

$$\int_0^1 x^2 dx$$

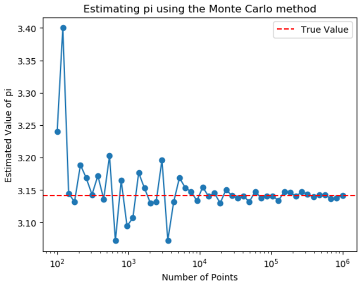
```
N = 1000000
accum = 0
for i in range(N):
    x = np.random.uniform(0, 1)
    accum += x**2
result = accum/float(N)
print("result: ", result)

result:  0.333651238336003
```

Sample size from 1 to 100 and calculate the value for 1000 replicates Plot 2.5th and 97.5th percentile of the 1000 values to see how the variation changes with sample size.



Example: Estimate Pi

```python
def estimate_pi(n):
    count = 0
    for i in range(n):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)
        if x**2 + y**2 <= 1:
            count += 1
    return 4 * count / n
```



How Large Is The Error?

Central Limit Theorem

Given a sequence of independent identically distributed variates ξ_i with expectation and variance

$$\mathbb{E}[\xi_i] = \mu, \quad V[\xi_i] = \sigma^2 \tag{9}$$

and the running sum $\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} \xi_i$. Then for increasing n, the composite variate

$$e_n := \frac{\hat{\mu}_n - \mu}{\sigma/\sqrt{n}} \tag{10}$$

converges in distribution to the standard normal distribution $\mathcal{N}(0,1)$.

Error Estimation for Monte Carlo Methods

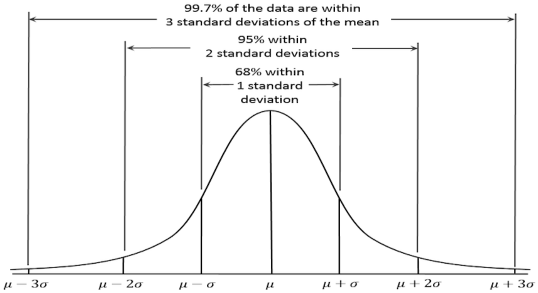
- From central limit theorem we know that our estimator $\hat{\mu}_n$ approaches a normal distribution: $\hat{\mu}_n \to \mathcal{N}(\mu, \frac{\sigma^2}{n})$
- A statistical measure for the uncertainty in any one simulation of result of $\hat{\mu}_n$ is then the standard deviation of $\hat{\mu}_n$: $\frac{\sigma}{\sqrt{n}}$
- In general we don't actually know σ — it's the standard deviation of ξ_i and our whole target is to estimate its expectation
- We can estimate σ using the samples:

$$\hat{\sigma}_n = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\xi_i^2 - \left(\frac{1}{n}\sum_{i=1}^{n}\xi_i\right)^2}. \tag{11}$$

And the Monte Carlo **standard error** is defined as: $\boxed{\epsilon_n = \frac{\hat{\sigma}_n}{\sqrt{n}}}$

Monte Carlo — Convergence

- From the standard error $\epsilon_n = \hat{\sigma}_n/\sqrt{n}$ we see that Monte Carlo method converges at the rate of $O(\sqrt{n})$ — to reduce the error by 10 times, you need to increase the number of samples by 100 times
- The standard error tells you the standard deviation of the estimator $\hat{\mu}_n$ — the probability that your estimation lies in $\mu \pm \epsilon_n$ is 68.27%
- The $68 - 95 - 99.7$ rule :


99.7% of the data are within 3 standard deviations of the mean
95% within 2 standard deviations
68% within 1 standard deviation
$\mu - 3\sigma$
$\mu - 2\sigma$
$\mu - \sigma$
μ
$\mu + \sigma$
$\mu + 2\sigma$
$\mu + 3\sigma$

Monte Carlo — Confidence Interval

Confidence Interval

As $n \to \infty$, a asymptotically valid $1 - \delta$ confidence interval for μ is an the interval

$$\left[\hat{\mu}_n - N^{-1}(1 - \delta/2)\epsilon_n,\ \hat{\mu}_n + N^{-1}(1 - \delta/2)\epsilon_n\right]$$

where $N(\cdot)$ is the standard cumulative normal function.

- A confidence interval displays the probability that a parameter will fall between a pair of values around the mean. The interval covers the true value μ with probability $1 - \delta$
- Rule of thumb: when $\delta = 0.05$, $N^{-1}(1 - \delta/2) \approx 1.96$, i.e., $\hat{\mu}_n \pm 1.96\epsilon_n$ gives a 95% confidence interval,

Generation of Random Process $\mathbf{X}_i$

- In the expectation of interest $\mathbb{E}[h(\mathbf{X}_i)]$,
  - h is the payoff function
  - the random vector $\mathbf{X}_i$ is the underlying asset
- The only non-trivial component in the Monte Carlo algorithm is the generation of $\mathbf{X}_i$
- The distribution of $\mathbf{X}_i$ might not be known analytically — they depend on the diffusion model
  - For Black-Scholes model the distribution is log-normal: $\frac{dS}{S} = (r-q)dt + \sigma dW_t$
  - For local volatility model the distribution has no closed form: $\frac{dS}{S} = (r-q)dt + \sigma(S,t)dW_t$
- However they are both adapted to the random processes with known distribution — Brownian motions
- Therefore we only need to generate the Brownian motions

Generation of Brownian Motions

- To simulate Brownian motions we need to be able to generate random numbers with normal distribution — any interval of a Brownian motion, $W_t - W_s$, is normally distributed with 0 mean and variance $t - s$, and is independent from other non-overlapping intervals
- Random numbers with normal distribution, or any other non-uniform distribution, can be generated from uniform random variates, using e.g.,
  - Approximation using central limit theorem
  - Box-Muller method
  - Inverse transformation method
- We discuss the generation of uniform random number $\mathcal{U}(0,1)$ first then the transformation

Random Number Generator

- Computer programs are designed to follow instructions in a deterministic way — in other words, they are **predictable**
- Computer will not be able to generate true random numbers unless the randomness comes as input. For example, random.org provides random number API whose randomness comes from atmospheric noise.
- Computer generated random numbers are referred to as **pseudo-random numbers** (PRN) because they are not truly random
- But is **pseudo** random number bad for us? Not really.

Desired Properties of Random Numbers in QF

- The random numbers we use should behave similarly to realization of independent, identically distributed random variables with a certain distribution. True randomness is better but pseudo randomness can achieve this with certain limitations.
- We use random numbers as a statistical tool for integration — true randomness does not add much value on this compared to pseudo randomness
- We need to be able to reproduce the random numbers — pricing an option twice using Monte-Carlo you do not want to see two different prices. In other words, we want it to be **deterministic** despite Monte Carlo errors. And pseudo random number wins.
- Be aware that there is no flawless pseudo random number generator — it's good practice to keep several alternatives of the pseudo random number generator in your library and cross test each other

PRNG — General Principle

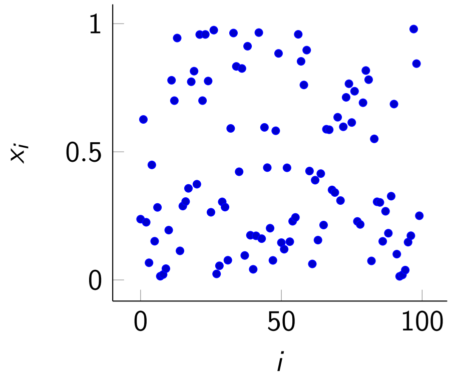
1. Given the current value of one or more state variables (usually stored internally in the generator)
2. Apply a mathematical iteration algorithm to obtain a new set of values for the state variables
3. Use a specific formula to obtain a new uniform (0, 1) variate from the current state variables

PRNG — Middle-Square Method

- The very first algorithm for the computer generation of pseudo random numbers due to John von Neumann et al.
- One state variable: x_i
- Iteration x_{i+1}: extract the middle four digits of x_i^2
- Random number: x_{i+1}
- Example: start from the seed $x_0 = 0.9876$

$$\begin{aligned} x_0 &= \quad 0.9876 \\ x_0^2 &= 0.97\underline{5353}76 \\ x_1 &= \quad 0.5353 \\ x_1^2 &= 0.28\underline{6546}09 \\ x_2 &= \quad 0.6546 \end{aligned}$$

Middle-Square Method — Example

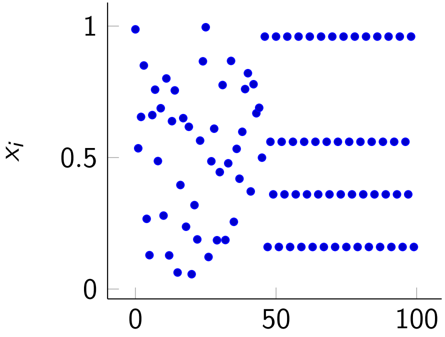


PRNs — mid-square method seed at 0.2372

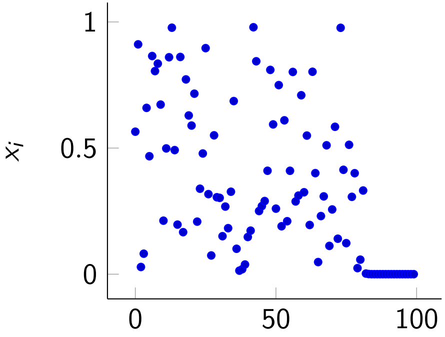
```python
import matplotlib.pyplot as plt

xs = [0] * 100
xs[0] = 0.2372 # seed
for i in range(1, 100):
    xs[i]=(int(xs[i-1]**2*1.0e6)
     %1e4)/1.0e4
plt.scatter(range(100), xs)
plt.show()
```

Middle-Square Method — Problem

- Problem of mid-pointe method: very likely to end up in a short periodic orbit or be absorbed at 0.



PRNs — mid-square method seed at 0.9876



PRNs — mid-square method seed at 0.5649

- Seed 0.9876: sequence starts to repeat from x_{46}, and is able to generate four numbers afterwards: `0.96, 0.16, 0.56, 0.36`
- Seed 0.5649: sequence absorbed at 0 at x_{84}

Congruential Generators

- Congruential generators update the state variable by the integer operation:

$$m_{n+1} = (a\ m_n + c) \mod M \tag{12}$$

where $0 < a < m$, $0 \leq c < m$ are constant integers, and **mod M** means **modulo M** which means it is divided by M and keep the remainder.$0 \leq m_0 < M$ is the seed.

- m_i's can be scaled to $[0, 1]$ by $x_i = \dfrac{m_i}{M-1}$
- If we choose a and M to be co-prime and $c \neq 0$, the sequence won't be absorbed at a fixed point (a pair of numbers are said to be co-prime when they have their highest common factor as 1.)
- When $c = 0$ this is called *linear* congruential generator. The system won't be absorbed at a fixed point if a and M are co-prime and the starting point is not 0.

Congruential Generators — Example

- For small M it's still easy to show that the sequence generated by congruential generators is periodic:

  ```
  M = 11, a = 3, c = 0, m0 = 3:
  3, 9, 5, 4, 1, 3, 9, ...
  ```

- The maximum length of the period is $M - 1$, but of course $M - 1$ is not guaranteed — example above

- With very large M we can have very long period such that the repetition becomes invisible to our application

- The values M and a have to be very carefully chosen [1]
  - In IBM's early days it used $a = 65539$, $M = 2^{31}$, and $m_0 = 1$ — this was reported to be highly inadequate
  - A choice of $a = 5^{17}$, $M = 2^{40}$, and $m_0 = 1$ was reported to work well with period 2^{38}

Congruential Generators — Extensions

- The minimal standard generator *Rand0* : a linear congruential generator with $a = 16807$ and $M = 2^{31} - 1$

- *Rand1* : enhancement of *Rand0* using a careful shuffling algorithm

- *Rand2* : coupling two linear congruential generators to construct one of a much longer period

Mersenne Twister

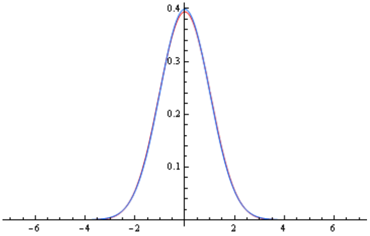
- Presented by Matsumoto and Nishimura in 1998. It was given this name because it has a period of 219937- 1 called the Mersenne prime.
- Utilizes many existing methods to rectify most of the flaws found in older PRNGs
- Full algorithm can be found at Wikipedia
- The period of the sequence is a Mersenne prime number: $2^n - 1$
- The popular one is **mt19937**: $n = 19937$ and period $2^{19937} - 1$ — equivalent to infinity periodicity for us
- mt19937 has equidistribution property in at least 623 dimensions (linear congruential generator has 5 dimensions in contrast)
- The PRNG of choice: provides fast generation of high-quality pseudo random numbers. Python `random` package generates numbers using **mt19937**.

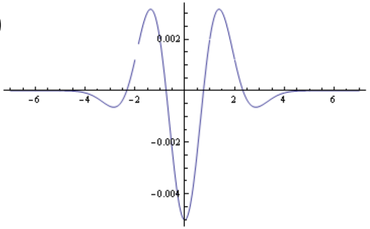
Uniform To Normal Distribution — Central Limit Theorem

- Now we have random numbers from the **uniform distribution** $\mathcal{U}(0,1)$
- To generate random numbers with **normal distribution**, one simple way is to utilize the central limit theorem:
  - Recall that $\frac{\hat{\mu}_n - \mu}{\sigma/\sqrt{n}} \sim \mathcal{N}(0,1)$
  - The variance of $\mathcal{U}(0,1)$ is $\int_0^1 (x-0.5)^2 dx = \frac{1}{12} = \sigma^2$
  - So we draw **12** uniform random numbers u_i, and let

$$m_k = \sum_{i=1}^{12} u_i - 6 \tag{13}$$

  - m approximates standard normal distribution due to central limit theorem
  - probability density function (blue) and error is shown at the right





Uniform To Normal Distribution — Box Muller Method

- Box-Muller method is another **easy to implement** algorithm to transform uniform distribution to normal distribution
- It is based on the property of the bivariate normal distribution: if $Z \sim \mathcal{N}(0, I_2)$, then
  - $R = Z_1^2 + Z_2^2$ is exponentially distributed with mean 2:
    $$P(R \leq x) = 1 - e^{-\frac{x}{2}}$$
  - Given R, the point (Z_1, Z_2) is uniformly distributed on the circle of radius $\sqrt{R}$ centered at origin
- The algorithm is
  1. Generate independent U_1, U_2 from $\mathcal{U}(0, 1)$
  2. $R \leftarrow -2\log(U_1)$ — uniform to exponential distribution (inverse CDF)
  3. $V \leftarrow 2\pi U_2$ — the angle to determine the point on the circle
  4. $Z_1 \leftarrow \sqrt{R}\cos(V)$, $Z_2 \leftarrow \sqrt{R}\sin(V)$
- Transform a pair (U_1, U_2) to (Z_1, Z_2)

Acceptance Rejection Method

- The previous two methods are specific to normal distribution
- Acceptance rejection method is generic to transform sample from one distribution (typically more convenient to generate) to another (not that convenient to generate)
- Let $g(x)$ be the pdf of a distribution we know how to sample, and $f(x)$ be the pdf of the target distribution, and $f(x) \leq cg(x)$ for all x.
- Idea is to generate x, then accept it as a sample for f with probability $\frac{f(x)}{cg(x)}$, to decide whether to accept the sample x we can just draw a random number u from $\mathcal{U}(0,1)$ and accept x if $u < \frac{f(x)}{cg(x)}$

Why Acceptance Rejection Method Works?

- The generated sample, denoted as y, has the distribution:

$$P(y \in A) = P\left(x \in A | u \leq \frac{f(x)}{cg(x)}\right) = \frac{P\left(x \in A, u \leq \frac{f(x)}{cg(x)}\right)}{P(u \leq \frac{f(x)}{cg(x)})}$$

- For a given x, the probability $u \leq \frac{f(x)}{cg(x)}$ is simply $\frac{f(x)}{cg(x)}$, so the denominator reads

$$P\left(u \leq \frac{f(x)}{cg(x)}\right) = \int_{\mathcal{X}} \frac{f(x)}{cg(x)} g(x) dx = \frac{1}{c} \tag{14}$$

- Thus y has the desired distribution:

$$P(y \in A) = c\ P\left(x \in A, u \leq \frac{f(x)}{cg(x)}\right) = c \int_A \frac{f(x)}{cg(x)} g(x) dx = \int_A f(x) dx \tag{15}$$

- (14) also tells the acceptance rate: larger c — more expensive

Acceptance Rejection — Algorithm

Algorithm 2 AcceptanceRejection

```
1: repeat
2:     Generate $x$ from distribution $g$
3:     Generate $u$ from $\mathcal{U}(0,1)$
4: until $U \leq f(x)/cg(x)$
5: return $x$
```

- The idea of acceptance rejection is used by many methods
- One example is the modified Box-Muller method for normal random number generation, namely polar rejection method (or Marsaglia-Bray algorithm),
    - Main modification to Box-Muller is to use acceptance rejection to generate uniformly distributed points in a unit disc
    - Avoid computing sin and cos as in Box-Muller method

Polar Rejection Method

Algorithm 3 PolarRejection

```
1: **repeat**
2:     Generate $u_1, u_2 \sim \mathcal{U}(0,1)$
3:     $u_1 \leftarrow 2u_1 - 1$
4:     $u_2 \leftarrow 2u_2 - 1$  $\{(u_1, u_2)$ uniformly distributed over $[-1,-1] \times [1,1]\}$
5:     $x \leftarrow u_1^2 + u_2^2$
6: **until** $x \leq 1$                    $\{x \sim \mathcal{U}(0,1)$ after acceptance rejection$\}$
7: $y \leftarrow \sqrt{-2\log x}$                $\{y$ is equivalent to $\sqrt{R}$ in Box-Muller$\}$
8: $Z_1 \leftarrow y \times u_1/\sqrt{x}$                  $\{u_1/\sqrt{x}$ — $\cos(V)$ in Box-Muller$\}$
9: $Z_2 \leftarrow y \times u_2/\sqrt{x}$                  $\{u_2/\sqrt{x}$ — $\sin(V)$ in Box-Muller$\}$
10: return $x$
```

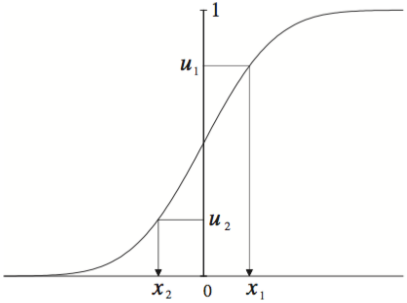
- An algorithm similar to Box-Muller method for generating normal random numbers
- Faster than Box-Muller despite the acceptance rate at first step is $\frac{\pi}{4}$

Inverse Transformation Method

- Another generic way to transform random samples from uniform distribution to any other distribution is through the inverse cumulative density function of the target distribution
- We want to generate random variable X with property that $P(X \leq x) = CDF(x)$ for all x. Inverse transformation method sets

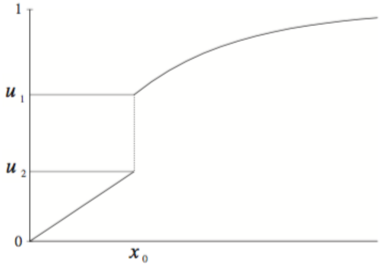
$$X = CDF^{-1}(u), \quad u \sim \mathcal{U}(0,1) \tag{16}$$

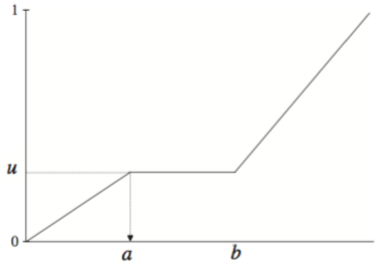
- Draw u from $\mathcal{U}(0,1)$ and from $CDF^{-1}(\cdot)$ we get the sample x
- We used the inverse transformation method to convert uniform random number to exponential distribution in Box-Muller



Inverse Transformation Method — Considerations

- The inverse CDF is not always one-to-one





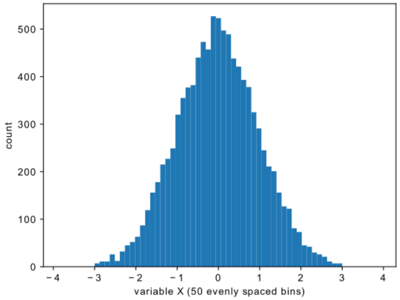
- There can be many to one (jump in CDF), one to many (flat CDF)
- If we use inverse transformation method we need to handle the one to many case with pre-defined conventions

Inverse Transformation Method For Normal Distribution

- CDF of normal distribution is not analytic — inverse of it is non-trivial
- Typical procedure:
  - Use an analytic formula that approximates N^{-1}, and use it as initial guess
  - Apply root search algorithm to find the solution
- Not necessarily fast
- Has the nice property of being monotonic ($N(\cdot)$ is strictly increasing) with respect to the uniform — helps in variance reduction techniques, e.g., antithetic variates
- Use exactly 1 uniform random number to generate 1 normal random number — preserve dimensionality and periodicity of the uniform PRNG

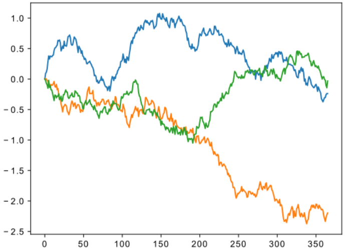
PRNG — Implementation

```
import numpy as np
import math
from matplotlib import pyplot as plt

np.random.seed(0) # we want to fix the seed so the numbers are reproducible
data = np.random.normal(0, 1, 10000)
bins = np.linspace(math.ceil(min(data)),math.floor(max(data)),50)  # fixed number of bins
plt.xlim([min(data)-0.5, max(data)+0.5])
plt.hist(data, bins=bins)
plt.xlabel('variable X (50 evenly spaced bins)')
plt.ylabel('count'), plt.show()
```



Brownian Motion Generation

```python
np.random.seed(0)
# generate 3 brownian motions for 1Y
nBrownians, nTimeSteps = 3, 366
brownians = np.zeros((nBrownians, nTimeSteps))
# each time step is 1 day,
# so standard deviation is sqrt(1/365.0)
stdev = math.sqrt(1/365.0)
for i in range(nBrownians):
    for j in range(1, nTimeSteps):
        dw = np.random.normal(0, stdev)
        brownians[i,j] = brownians[i,j-1] + dw

plt.plot(range(nTimeSteps), brownians[0])
plt.plot(range(nTimeSteps), brownians[1])
plt.plot(range(nTimeSteps), brownians[2])
plt.show()
```



Pricing European Option With Monte Carlo

- Pricing European option with Monte Carlo is easy, if the model is **Black-Scholes**

- We do not need to use the whole Brownian motion, because we have S_T in closed form:

$$S_T = S_0 e^{(r-q-\frac{1}{2}\sigma^2)T+\sigma W_T} \tag{17}$$

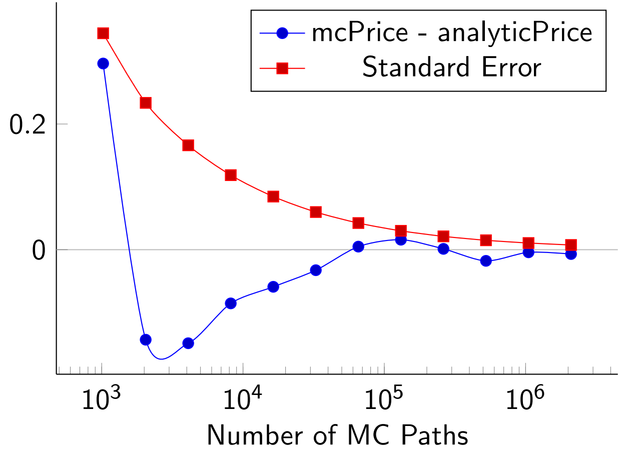
- Only the end value of the Brownian motion is needed

- So instead of simulating 365 time steps we only need to simulation 1 step

MC European Implementation

```python
import math
from numpy.random import random
import numpy as np

def mcEuropean(S0, T, r, q, vol, nPaths, trade):
    random.seed(0)
    sum,hsquare = 0,0
    stdev = np.math.sqrt(T)
    for i in range(nPaths):
        wT = np.random.normal(0, stdev)
        h = trade.payoff(S0 * math.exp((r - q - 0.5*vol*vol) * T + vol * wT))
        sum += h
        hsquare += h * h

    pv = math.exp(-r*T) * sum / nPaths
    stderr = math.sqrt((hsquare/nPaths - (sum/nPaths) * (sum/nPaths)) / nPaths)
    return pv, stderr
```

MC European Convergence



European call option, $K = 100, S = 100, r = 5\%, q = 2\%, T = 1, \sigma = 15\%$

Model Without The Closed Form Solution

- Black Scholes model is a special case, normally we do not have closed form solution to our SDE
- Look at the SDE of local volatility model:

$$dS = S(r-q)dt + S\sigma(S,t)dW_t \tag{18}$$

Closed form solution does not exist since $\sigma(S,t)$ is state dependent

- Consider a process X that satisfies the below SDE of a general form:

$$dX(t) = a(X(t))dt + b(X(t))dW(t). \tag{19}$$

- Integrating (19) we have

$$X(t_{i+1}) = X(t_i) + \int_{t_i}^{t_{i+1}} a(X(u))du + \int_{t_i}^{t_{i+1}} b(X(u))dW_u \tag{20}$$

- To solve the integrals we need to discretize them

Euler Discretization Scheme

- Using time step Δt, the Euler discretization scheme approximates

$$\int_t^{t+\Delta t} a(X(u))du = a(X(t))\Delta t \tag{21}$$

$$\int_t^{t+\Delta t} b(X(u))dW_u = b(X(t))\Delta W = b(X(t))\sqrt{\Delta t}Z \tag{22}$$

where $Z \sim N(0,1)$

- The stepwise induction is

$$X(t+\Delta t) = X(t) + a(X(t))\Delta t + b(X(t))\sqrt{\Delta t}Z \tag{23}$$

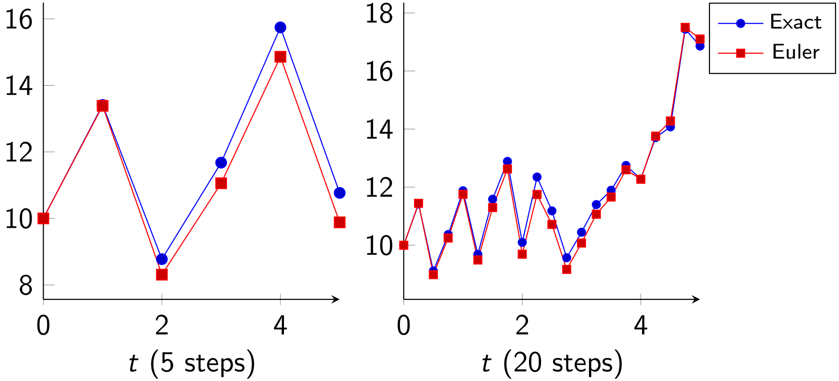
- The truncation error of the drift term is $O(\Delta t)$
- The truncation error of the diffusion term is $O(\Delta W)$, i.e., $O(\sqrt{\Delta t})$
- The overall convergence order of Euler scheme is therefore $O(\sqrt{\Delta t})$

Euler Scheme For Black Scholes Model

- For Black Scholes model Euler scheme can be written as

$$S_{t+\Delta t} = S_t(1 + \mu\Delta t + \sigma\Delta W_t) \tag{24}$$

- 5Y simulation of spot with $S_0 = 10$, $\sigma = 30\%$, $\mu = 10\%$



Change of State Variable

- **State variable** is the variable we record in each step of Monte Carlo simulation
- It is not necessary for us to use the spot price as state variable
- We only need to be able to **reconstruct** the spot price from the state variable
- If we choose $X = \log S$ at state variable, the diffusion can be written as

$$dX = \frac{\partial X}{\partial S} dS + \frac{1}{2}\frac{\partial^2 X}{\partial S^2} dS^2 = \left(\mu - \frac{1}{2}\sigma^2\right) dt + \sigma dW_t \tag{25}$$

- Euler scheme now gives us the exact solution:

$$X_{t+\Delta t} = X_t + (\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma \Delta W_t \quad \leftarrow \text{ Euler scheme} \tag{26}$$

$$S_{t+\Delta t} = S_t e^{(\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma \Delta W_t} \quad \leftarrow \text{ Reconstruction} \tag{27}$$

- This is due to the linearity of the diffusion process of X

Local Volatility Model Diffusion

- We can use the same state variable for the diffusion of local volatility model:

$$dX = \left(\mu - \frac{1}{2}\sigma(S,t)^2\right) dt + \sigma(S,t) dW_t \tag{28}$$

- Again the simplified constant drift does not affect the generality of our formulation, replacing it with a term structure $\mu(t)$ is trivial
- The diffusion can be discretized by Euler scheme as:

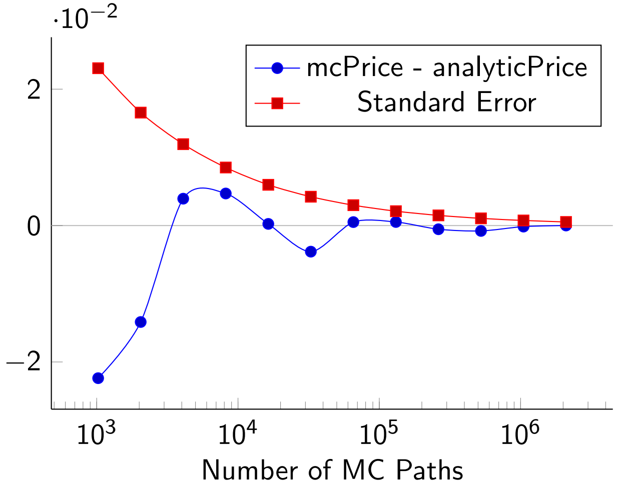
$$X_{t+\Delta t} = X_t + (\mu - \frac{1}{2}\sigma(X,t)^2)\Delta t + \sigma(X,t)\Delta W_t \tag{29}$$

- But now we are not exact — no closed form exact solution
- Our discretization error is $O(\sqrt{\Delta t})$

MC Local Volatility — Implementation

```python
def mcLocalVol(S0, T, r, q, lv, nT, nPaths, trade):
    random.seed(0)
    sum, hsquare = 0, 0
    dt = T / nT
    sqrtdt = math.sqrt(dt)
    for i in range(nPaths):
        X = math.log(S0)
        for j in range(1, nT+1):
            vol = lv.LV((j-1)*dt, math.exp(X))
            a = (r - q - 0.5*vol * vol) * dt # drift
            b = np.rand.normal(0, sqrtdt) * vol # diffusion
            X += a + b # update state variable
        h = trade.payoff(math.exp(X))
        sum += h
        hsquare += h * h
    pv = math.exp(-r * T) * sum / nPaths
    stderr = math.sqrt((hsquare/nPaths-(sum/nPaths)*(sum/nPaths))/nPaths)
    return pv, stderr
```

MC Local Volatility — Convergence



European call option, $K = 5.2, S = 6, r = 5\%, q = 2\%, T = 1, iv = 12.93\%$

Milstein Discretization Scheme

- We can achieve higher order convergence on discretization
- One popular scheme for Monte Carlo is the **Milstein scheme**.
- The target is to achieve $O(\Delta t)$, since the drift term is already $O(\Delta t)$, we need to refine only the diffusion term
- In Euler we just use $b(X(t))$ for the period $t \to t + \Delta t$, this is the place of truncation error so we need to refine this part

Milstein Discretization Scheme

- The diffusion process of $b(X(t))$ is

$$db(X(t)) = \frac{\partial b}{\partial X} dX + \frac{1}{2}\frac{\partial^2 b}{\partial X^2} dX^2 = b' \underbrace{(a \cdot dt + b \cdot dW_t)}_{dX} + \frac{1}{2} b'' b^2 dt$$

$$= \underbrace{(ab' + \frac{1}{2} b^2 b'')}_{c(X(t))} dt + b(X(t)b'(X(t)dW_t.$$

So

$$b(X(u)) = b(X(t)) + \int_t^u b(X(s))b'(X(s))dW_s + \int_t^u c(X(s))ds$$

- Since $\int_t^u c(X(t))dt$ is of order $O(u-t)$ and is higher than we need (because of the multiplication with ΔW_t is order $O(\sqrt{\Delta t})$, we can omit it. Therefore

$$b(X(u)) \approx b(X(t)) + \int_t^u b(X(s))b'(X(s))dW_s$$

Milstein Discretization Scheme

- Approximating the integral using Euler

$$\int_t^u b(X(u))b'(X(u))dW_u = b(X(t))b'(X(t))(W_u - W_t) + O(u-t) \tag{30}$$

- Therefore

$$b(X(u)) = b(X(t)) + b(X(t))b'(X(t))(W_u - W_t) + O(u-t). \tag{31}$$

- Now, come back to the target integral (20), the result is

$$\int_{t_i}^{t_{i+1}} b(X(u))dW_u = \underbrace{b(X(t))\sqrt{\Delta t}Z}_{\text{Euler term}} + \underbrace{\frac{1}{2}b(X(t))b'(X(t))\Delta t(Z^2-1)}_{\text{Milstein correction term}} \tag{32}$$

Milstein Discretization Scheme

And here is the last step of derivation of (32):

$$\int_t^{t+\Delta t} b(X(u))dW_u = \int_t^{t+\Delta t} [b(X(t)) + b(X(t))b'(X(t))(W_u - W_t)]dW_u$$

$$= b(X(t))(W_{t+\Delta t} - W_t) + b(X(t))b'(X(t)) \left(\underbrace{\int_t^{t+\Delta t} W_u dW_u}_{d\left(\frac{1}{2}W^2 - \frac{1}{2}t\right) = WdW} - W_t(W_{t+\Delta t} - W_t) \right)$$

$$= b(X(t))(W_{t+\Delta t} - W_t) + b(X(t))b'(X(t)) \left(\frac{1}{2}W_{t+\Delta t}^2 - \frac{1}{2}W_t^2 - \frac{1}{2}\Delta t - W_t(W_{t+\Delta t} - W_t) \right)$$

$$= b(X(t))(W_{t+\Delta t} - W_t) + \frac{1}{2}b(X(t))b'(X(t))[(W_{t+\Delta t} - W_t)^2 - \Delta t]$$

$$= \underbrace{b(X(t))\sqrt{\Delta t}Z}_{\text{Euler term}} + \underbrace{\frac{1}{2}b(X(t_i))b'(X(t_i))\Delta t(Z^2 - 1)}_{\text{Milstein correction term}}$$

Discretizing local volatility model diffusion we have:

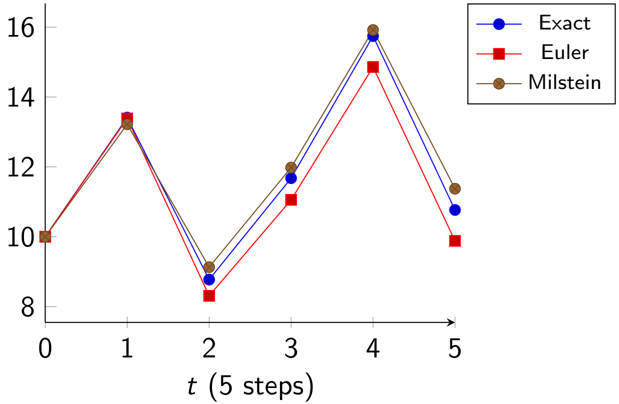
$$X_{t+\Delta t} = X_t + (\mu - \frac{\sigma(X,t)^2}{2})\Delta t + \sigma(X,t)\Delta W_t + \boxed{\frac{\sigma(X,t)}{2}\frac{\partial \sigma(X,t)}{\partial X}(\Delta W_t^2 - \Delta t)} \tag{33}$$

Error of Milstein Scheme

- We illustrate the error of Milstein scheme using Black Scholes model with state variable S. Milstein scheme is:

$$S_{t+\Delta t} = S_t(1 + \mu\Delta t + \sigma\Delta W_t + \frac{\sigma^2}{2}(\Delta W_t^2 - \Delta t)) \qquad (34)$$

- 5Y simulation of spot with $S_0 = 10$, $\sigma = 30\%$, $\mu = 10\%$



Monte Carlo Method — Pros and Cons

Pros:

- Easy to implement
- Easy to extend
- Dimension friendly

Cons:

- Monte Carlo noise
- Slow for low dimension problems compared to PDE or Tree
- Not efficient and reliable for American style options

Two Factor Monte Carlo — Correlating Brownian Motions

- When we have more than one model factors who are correlated with each other, we need to construct correlated Brownian motions
- The correlation of the increments of two Brownian motions is

$$\rho = \frac{Covar}{\sqrt{Var_1 \cdot Var_2}} = \frac{\mathbb{E}(\Delta W_1 \Delta W_2) - \mathbb{E}(\Delta W_1) E(\Delta W_2)}{\sqrt{Var(\Delta W_1) \cdot Var(\Delta W_2)}} \tag{35}$$

$$= \frac{\mathbb{E}(\Delta W_1 \Delta W_2)}{dt} \tag{36}$$

So

$$\mathbb{E}(\Delta W_1 \Delta W_2) = \rho dt \tag{37}$$

$$\Delta W_1 = \xi_1 \sqrt{dt} \tag{38}$$

$$\Delta W_2 = \xi_2 \sqrt{dt} \tag{39}$$

where ξ_1 and ξ_2 are two standard normal random variates, and

$$\mathbb{E}(\xi_1 \xi_2) = \rho \tag{40}$$

- If we represent the correlation matrix as

$$\mathbf{C} = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \tag{41}$$

And the random samples for ξ_1 and ξ_2 (to generate the correlated Brownian increments) are vectors:

$$\hat{\boldsymbol{\xi}}_1 = \left[\hat{\xi}_{1,0}, \quad \hat{\xi}_{1,1}, \quad \ldots, \quad \hat{\xi}_{n,1}\right], \tag{42}$$

$$\hat{\boldsymbol{\xi}}_2 = \left[\hat{\xi}_{2,1}, \quad \hat{\xi}_{2,2}, \quad \ldots, \quad \hat{\xi}_{2,n}\right] \tag{43}$$

and

$$\begin{bmatrix} \hat{\boldsymbol{\xi}}_1 \\ \hat{\boldsymbol{\xi}}_2 \end{bmatrix} \times \begin{bmatrix} \hat{\boldsymbol{\xi}}_1^\top, & \hat{\boldsymbol{\xi}}_2^\top \end{bmatrix} = \begin{pmatrix} \hat{\boldsymbol{\xi}}_1 \cdot \hat{\boldsymbol{\xi}}_1, & \hat{\boldsymbol{\xi}}_1 \cdot \hat{\boldsymbol{\xi}}_2 \\ \hat{\boldsymbol{\xi}}_2 \cdot \hat{\boldsymbol{\xi}}_1, & \hat{\boldsymbol{\xi}}_2 \cdot \hat{\boldsymbol{\xi}}_2 \end{pmatrix} = \begin{pmatrix} n, & n\rho \\ n\rho, & n \end{pmatrix} = n\,\mathbf{C} \tag{44}$$

- If we decompose $\mathbf{C}$ to the product of a matrix and its transpose:

$$\mathbf{C} = \mathbf{L}\mathbf{L}^\top \tag{45}$$

We have

$$\begin{bmatrix} \hat{\xi}_1 \\ \hat{\xi}_2 \end{bmatrix} \times \begin{bmatrix} \hat{\xi}_1^\top, & \hat{\xi}_2^\top \end{bmatrix} = n\, \mathbf{L}\mathbf{L}^\top \tag{46}$$

So

$$\mathbf{L}^{-1} \begin{bmatrix} \hat{\xi}_1 \\ \hat{\xi}_2 \end{bmatrix} \times \begin{bmatrix} \hat{\xi}_1^\top, & \hat{\xi}_2^\top \end{bmatrix} (\mathbf{L}^\top)^{-1} = n\, \mathbf{I} \tag{47}$$

where $\mathbf{I}$ is an identity matrix.

Correlating Standard Normal Vectors

- If we let

$$\begin{bmatrix}\hat{\zeta}_1\\ \hat{\zeta}_2\end{bmatrix} = (\mathbf{L})^{-1}\begin{bmatrix}\hat{\xi}_1\\ \hat{\xi}_2\end{bmatrix} \tag{48}$$

  The random vectors vectors ζ_1 and ζ_2 are uncorrelated because $\begin{bmatrix}\hat{\zeta}_1\\ \hat{\zeta}_2\end{bmatrix} \times \begin{bmatrix}\hat{\zeta}_1^\top, & \hat{\zeta}_2^\top\end{bmatrix} = n\,\mathbf{I}$.

- So if we generate independent standard normal random vectors ζ_1 and ζ_2, we can correlate them by

$$\boxed{\begin{bmatrix}\hat{\xi}_1\\ \hat{\xi}_2\end{bmatrix} = \mathbf{L}\begin{bmatrix}\hat{\zeta}_1\\ \hat{\zeta}_2\end{bmatrix}} \tag{49}$$

Cholesky Decomposition

- If a matrix is symmetric and positive definite, a special LU decomposition — Cholesky decomposition is faster than the other LU decomposition and satisfies the property that:

$$\mathbf{C} = \mathbf{L}\mathbf{L}^\top \tag{50}$$

  and $\mathbf{L}$ is a lower triangular matrix.

- The requirement that the Cholesky decomposition exists is the matrix $\mathbf{C}$ is symmetric positive definite. This is true if $\rho \neq \pm 1$:

$$\mathbf{x}^\top C\mathbf{x} = \frac{1}{n}\underbrace{\mathbf{x}^\top \begin{bmatrix} \hat{\xi}_1 \\ \hat{\xi}_2 \end{bmatrix}}_{(\mathbf{y}^\top)_{1\times n}} \times \underbrace{\begin{bmatrix} \hat{\xi}_1^\top, & \hat{\xi}_2^\top \end{bmatrix} \mathbf{x}}_{\mathbf{y}_{n\times 1}} = \frac{\| \mathbf{y} \|}{n} \geq 0 \tag{51}$$

  So (50) exists by definition.

- Now to generate two correlated Brownian motions we can
  1. Generate two independent standard normal samples ζ_1 and ζ_2
  2. Decompose the correlation matrix using Cholesky: $\mathbf{C} = \mathbf{L}\mathbf{L}^\top$
  3. Generate the correlated normal variates by: $\begin{bmatrix} \hat{\xi}_1 \\ \hat{\xi}_2 \end{bmatrix} = \mathbf{L} \begin{bmatrix} \hat{\zeta}_1 \\ \hat{\zeta}_2 \end{bmatrix}$

  And this routine works in general, for m random variates.
- The Cholesky decomposition algorithm is straight-forward:

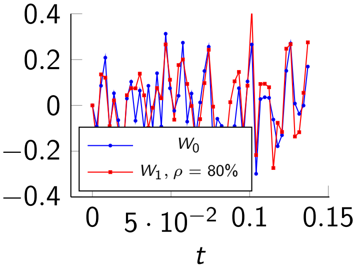
$$\mathbf{C} = \mathbf{L}\mathbf{L}^\top = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{23} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{23} \\ 0 & 0 & L_{33} \end{pmatrix} \tag{52}$$

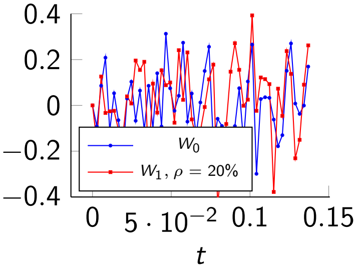
$$= \begin{pmatrix} L_{11}^2 & & \text{(symmetric)} \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & \sum_{i=1}^{3} L_{3i}^2 \end{pmatrix} \tag{53}$$

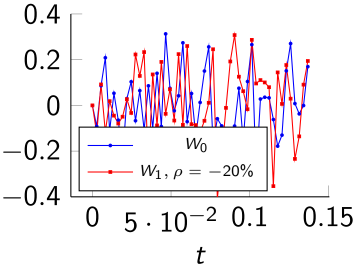
  Just need to start from the top-left corner and progressively solve for each L_{ij}
- In python implementation: `L = np.linalg.cholesky(C)`

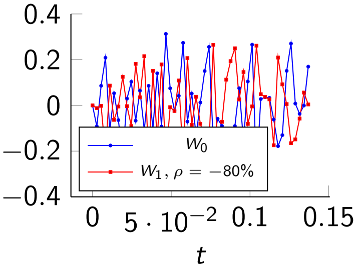
Correlated Brownian Motions — Examples

Now we can generate Brownian motions with correlations (example with different ρ)





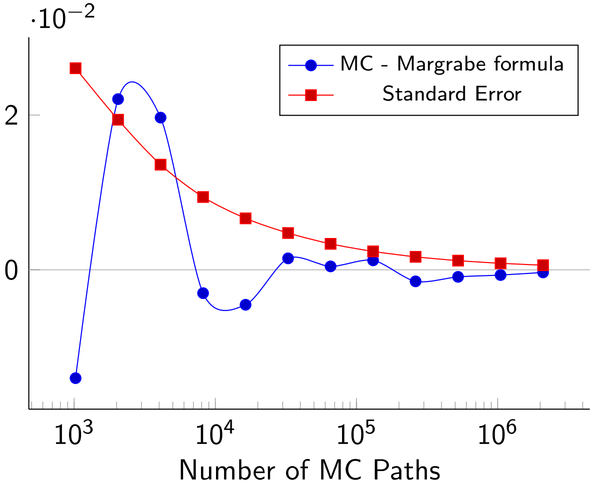




Pricing Spread Option — MC with Two Assets under BS

```python
def mcSpread(payoff, S1, S2, T, r, q1, q2, vol1, vol2, rho, nPaths, nT):
    np.random.seed(0)
    sum, hsquare, C = 0, 0, np.identity(2)
    C[0, 1] = C[1, 0] = rho
    L = np.linalg.cholesky(C)
    for i in range(nPaths):
        brownians = np.zeros((2, nT))
        dt = T / nT
        stdev = math.sqrt(dt)
        # generate brownian increments
        for j in range(2):
            brownians[j] = np.random.normal(0, stdev, nT)
        brownians = np.matmul(L, brownians)
        x1, x2 = math.log(S1), math.log(S2)
        for j in range(nT):
            # simulate asset 1
            a = (r-q1-0.5*vol1*vol1) * dt # drift for asset 1
            b = brownians[0, j] * vol1 # diffusion term for asset 1
            x1 += a + b  # update state variable
            # simulate asset 2
            a = (r-q2-0.5*vol2*vol1) * dt # drift for asset 1
            b = brownians[1, j] * vol2 # diffusion term for asset 1
            x2 += a + b  # update state variable
        h = payoff(math.exp(x1), math.exp(x2))
        sum += h
        hsquare += h*h
    pv = math.exp(-r * T) * sum / nPaths
    se = math.sqrt((hsquare/nPaths - (sum/nPaths)*(sum/nPaths))/nPaths)
    return pv, se
```

MC Spread Option Convergence



$S_1 = 10, S_2 = 10, r = 10\%, q_1 = 2\%, q_2 = 1\%, T = 1, \sigma_1 = 15\%, \sigma_2 = 10\%, \rho = 50\%$

Generic Monte Carlo Framework

- Similar to our Binomial tree framework, we can decompose the Monte Carlo pricer to obtain a generic MC framework
- The MC framework can be much more generic than the tree framework due to the capability of Monte Carlo — very natural to handle path dependant products
- Major components:
  - Market
  - Random number generator
  - Monte Carlo product
  - Monte Carlo diffusion model
  - MC pricer that plumbs the above all

MC Framework — Steps

- We want to handle trade with multiple underlying assets — the market should provide the correlation of all the relevant model factors
- The pricer uses random number generator to generate Brownian motions and then correlate them.
- The job of diffusion model is to take a Brownian motion, and simulate underlying asset's price process. For each asset there should be one diffusion model.
- Each simulation path gives one realization of the market, the trade should be able to take that to return the discounted cash flows of this realization — the $h(\mathbf{X}_i)$ in our expectation

Implementation

mcPricer.py

```python
def mcPricer(trade, models, corrmat, nPaths):
    assetNames = trade.assetNames() # get all the assets involved for the payoff
    numFactors = corrmat.size() # get total number of factors (brownians)
    L = np.linalg.cholesky(corrmta) # cholesky decomposition
    dts = [] # get simulation time steps
    for a in assetNames:
        dts.append(models[a].GetTimeSteps(trade.AllDates()))
    dts = np.unique(dts)
    sum, hsquare, nT = 0, 0, dts.size()
    for i in range(nPaths):
        # generate independent bronian increments,
        for j in range(numFactors):
            brownians[j] = np.random.normal(0, 1, nT)
        brownians = np.matmul(L, brownians) # correlate them using L
        bidx, fobs = 0, dict() # fobs is a dict from asset name to observable,
        # each observable if a function from t to the observation price
        for k in assetNames.size():
            # pass the brownians to the model to generate the observation functions
            model = models[assetNames[k]]
            nF = model.NumberOfFactors()
            bs = brownians.project(bidx, bidx + nF)
            fobs[assetNames[k]] = model.Diffuse(dts, bs)
            bidx += nF
        # call the payoff function to obtain the discounted cashflows
        h = trade.DiscountedMCPayoff(fobs)
        sum += h
        hsquare += h*h
    pv = sum / nPaths
    se = math.sqrt((hsquare/nPaths - pv*pv)/nPaths)
    return pv, se
```

`mcPricer` requires a diffusion model to provide below functions:

- Model diffusion that returns an observable - a function that takes a future date and returns the asset price of that date:

```
fobs[assetNames[k]] = model.Diffuse(dts, bs)
```

- Number of factors - number of brownian motion needed to diffuse the model

```
nF = model.NumberOfFactors()
```

- Given a list of event dates from the trade, the intermediate times steps the model requires to diffuse properly:

```
models[a].GetTimeSteps(trade.AllDates())
```

- A concrete diffusion model — Black-Scholes

```python
class BlackScholes:
    def __init__ (self, S0, vol, r, q):
        self.vol, self.S0, self.r, self.q = vol, S0, r, q
    def NumberOfFactors(self):
        return 1
    def GetTimeSteps(self, eventDates):
        # black scholes diffusion is exact, no need to add more dates
        return eventDates
    def Diffuse(self, dts, bs):
        xs = [math.log(self.S0)]
        for i in (1, dts.size()):
            a = (self.r - self.q - 0.5 * self.vol * self.vol) * dts[i]
            b = self.vol * bs[0, i] * math.sqrt(dts[i])
            xs.append(xs[i-1] + a + b)
        return (lambda t: np.interp(t, dts, xs))
```

- The trade is then able to take all the relevant observables and determine the payoff of this simulation path:

```python
h = trade.DiscountedMCPayoff(fobs)
```

- Besides the discounted payoff, the trade needs to provide

```python
trade.assetNames() # get all the assets involved for the payoff
trade.AllDates() # critical event dates for the trade
```

- Spread option implementation:

```python
class SpreadOption():
    def __init__(self, asset1, asset2, expiry):
        self.expiry = expiry
        self.asset1, self.asset2 = asset1, asset2
    def payoff(self, S1, S2):
        return max(S1-S2, 0)
    def valueAtNode(self, t, S1, S2, continuation):
        return continuation
    def AssetNames(self):
        return [self.asset1, self.asset2, "DF.USD"]
    def AllDates(self):
        return [self.expiry]
    def DiscountedMCPayoff(self, fobs):
        df = fobs["DF.USD"](self.expiry)
        s1 = fobs[self.asset1](self.expiry)
        s2 = fobs[self.asset2](self.expiry)
        return df * max(s1 - s2, 0)
```

The main Function For Pricing

```python
asset1,asset2 = "STOCK1","STOCK2"
trade = SpreadOption(asset1, asset2, 1.0)
models = dict( asset1 = BlackScholes(10, 0.15, 0.10, 0.02),
               asset2 = BlackScholes(10, 0.10, 0.10, 0.01), )
corrmat = 0, 0, np.identity(2)
corrmat[0, 1] = corrmat[1, 0] = 0.5
pv, se = mcPricer(trade, models, corrmat, nPaths = 1024*32);
```

More trade types — Asian Option

- An Asian call option pays the option holder at option maturity T:

$$\max\left(\frac{1}{n}\sum_{i=1}^{n} S(t_i) - K, 0\right) \tag{54}$$

- To create an Asian option is straight-forward now:

```python
class AsianOption:
    ...
    def AllDate(self):
        return self.fixings
    def DiscountedMCPayoff(self, fobs):
        df = fobs["DF.USD"](self.fixings[-1])
        avg = 0
        for t in self.fixings:
            avg += fobs[self.asset](t)
        return df * self.payoffFun(avg / self.nFix)
```

Target Redemption Forward

- A structured product which consists of a strip of forwards each of which has its payout as the difference between the underlying rate on a given fixing and a predefined strike level:

$$C_i = S(t_i) - K \tag{55}$$

  The overall structure is limited by the requirement that once the total payout exceeds a target level, the structure terminates (knocks out).
- A generalized version allows the coupon C_i itself being a structure (call, put, risk reversal, etc).

```python
class TRF: # all that matters is implementation of DiscountedMCPayoff()
    ...
    def DiscountedMCPayoff(self, fobs):
        df = fobs["DF.USD"](self.fixings[-1])
        accum, discountedPO = 0
        for t in self.fixings:
            df = fobs["DF.USD"](t)
            po = self.payoffFun(fobs[self.asset])(t)
            accum += po
            discountedPO += df * po;
            if (accum > self.targetGain):
                break # triggers knockout
        return discountedPO
```

Variance Reduction Techniques

The commonly discussed techniques to reduce Monte-Carlo noise are

- Antithetic sampling
- Variates recycling
- Control variates
- Stratified sampling
- Importance sampling

Antithetic Sampling

- When we simulation Brownian motion by constructing sample paths, we can make use of the fact that for any one drawn path its mirror image has equal probability
- If we draw a Brownian path W_i, and use it to evaluate the payoff $h(W_i)$, we can also use $-W_i$ and obtain $h(-W_i)$.
- We use $\hat{h}_i = \frac{1}{2}(h(W_i) + h(-W_i))$ as our random sample of the payoff. The estimator becomes $\frac{1}{n}\sum_{i=1}^{n} \hat{h}_i$.

Antithetic Sampling

- The variance of the new estimator is

$$Var\left[\frac{1}{2}(h(W_i)+h(-W_i))\right]=\frac{1}{2}Var[h(W_i)]+\frac{1}{2}Cov[h(W_i),h(-W_i)] \tag{56}$$

- When we compare the variance with the standard Monte Carlo, we should compare with the case with twice amount of simulation paths, whose variance is $\frac{1}{2}Var[h(W_i)]$
- So if $Cov[h(W_i),h(-W_i)]<0$ the Monte Carlo variance is reduced by antithetic sampling — always true if h is monotonic w.r.t W
- Antithetic sampling favours asymmetric payoff.
- For symmetric payoff such as straddle, it does not help but does not make it worse either — the payoff is symmetric on the spot which is exponential of W, and the payoff is not necessarily centerred at S_0.

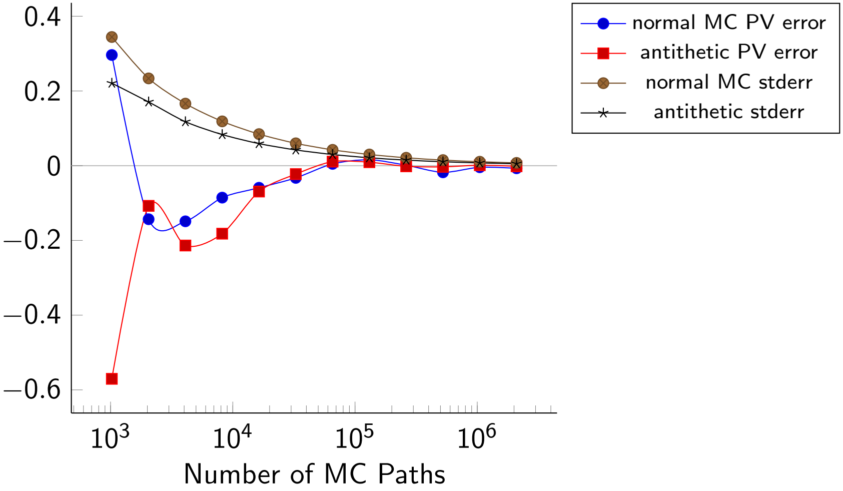
Antithetic Sampling — Implementation

- Implementation is trivial, and saves cost of random number generation — so nearly always turned on in practice.

- We take our `mcEuropean` as example:

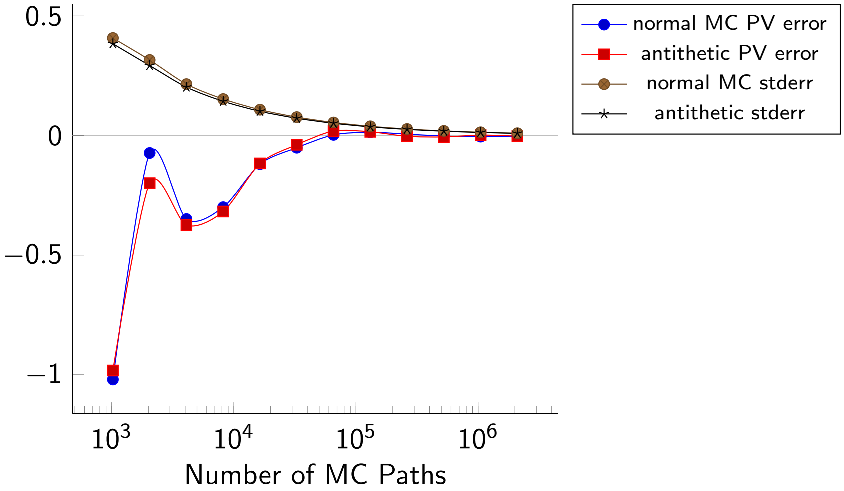
```python
sum,hsquare = 0,0
stdev = math.sqrt(T)
for i in range(nPaths):
    wT = np.random.normal(0, stdev)
    hA = trade.payoff(S0 * math.exp((r-q-0.5*vol*vol)*T + vol*wT))
    hB = trade.payoff(S0 * math.exp((r-q-0.5*vol*vol)*T - vol*wT))
    h = 0.5 * (hA + hB)
    sum += h
    hsquare += h * h
pv = math.exp(-r*T) * sum / nPaths
stderr = math.sqrt((hsquare/nPaths - (sum/nPaths)*(sum/nPaths))/nPaths)
```

Antithetic Sampling — Result (Call Option)



European call option, $K = 100, S = 100, r = 5\%, q = 2\%, T = 1, \sigma = 15\%$

Antithetic Sampling — Result (Straddle)



European straddle, $K = 100, S = 100, r = 5\%, q = 2\%, T = 1, \sigma = 15\%$

Variates Recycling

- Greeks (risk or sensitivities) are more important than the prices, when our pricer is numerical the usual way to calculate Greeks is finite difference:
- For example the delta of an instrument is approximated by

$$\delta = \frac{\partial PV}{\partial S} \approx \frac{PV(S + \Delta S) - PV(S - \Delta S)}{2\Delta S} \tag{57}$$

- The variance of delta is then

$$\frac{1}{4\Delta S^2}\left(Var(PV_{S+\Delta S}) + Var(PV_{S-\Delta S}) - 2Cov(PV_{S+\Delta S}, PV_{S-\Delta S})\right)$$

- If we reuse the same random samples to estimate the PV_+ and PV_-, we can maximize the covariance term and thus reduce the noise on our Greeks.
- This is also a reason why we would like our “random” numbers to be replicable

Control Variates

- We are trying to estimate the expectation of a function $h(\mathbf{X})$
- If we know the expectation of a function $g(\mathbf{X})$ analytically, we can use the estimator:

$$\mathbb{E}[h(\mathbf{X})] \approx \frac{1}{n}\sum_{i=1}^{n}\left(h(\mathbf{X}_i) + \beta(g^* - g(\mathbf{X}_i))\right) \tag{58}$$

  where g^* is the known expectation of $g(\mathbf{X})$ and β is a parameter.

- The variance of the samples is

$$Var[h] + \beta^2 Var[g] - 2\beta Cov[h, g] \tag{59}$$

- Taking the first derivative w.r.t β, the variance is minimized for $\beta = \dfrac{Cov[h, g]}{Var[g]}$ (can be estimated using simulation samples)

Control Variates

- The minimized variance is

$$Var[h] - \frac{Cov[h,g]^2}{Var[g]} = Var[h](1-\rho^2) \tag{60}$$

- The higher correlation g and h is, the more effective the technique is.
- Examples of control variates
  - Use forward as control variate for call option
  - Use geometric Asian option as control variate for arithmetic Asian option (possible only with Black-Scholes model).
- Cannot be applied in a general way, need to fine tune based on the problem

Stratified Sampling

- Idea is to divide the domain into k disjoint sets and calculate the expectation by

$$\mathbb{E}[h(\mathbf{X})] = \sum_{j=1}^{k} \mathbb{E}[h(\mathbf{X}|\mathbf{X} \in \mathbf{A}_j]p(\mathbf{X} \in \mathbf{A}_j) \tag{61}$$

- Depending on the function h, we can assign more paths to $\mathbf{A}_j$ whose h value varies more and less paths to the $\mathbf{A}$'s whose h value are more constant
- Inverse transformation method helps here because it allows us to draw uniform random numbers corresponding to each strata due to monotonicity
- However, this technique is difficult to generalize and the choice of strata is very problem-specific, difficult to define strata for multi-dimensional problem as well

Importance Sampling

- Idea is to apply a change of measure such that the probability mass is shifted to the more important region

$$\mathbb{E}_{\mathbb{Q}}[h(\mathbf{X})] = \int h(\mathbf{X})q(\mathbf{X})d\mathbf{X} = \int h(\mathbf{X})\frac{q(\mathbf{X})}{p(\mathbf{X})}p(\mathbf{X})d\mathbf{X} \tag{62}$$

$$= \mathbb{E}_{\mathbb{P}}\left[h(\mathbf{X})\frac{q(\mathbf{X})}{p(\mathbf{X})}\right] \tag{63}$$

- We can design a $p(\mathbf{X})$ such that it is centerred around the important region of $h(\mathbf{X})$, effectively more paths will be drawn on the important region and fewer paths on the non-varying region.
- Intrinsically the same as stratified sampling, so suffer the same problem:
  - very problem specific and difficult to generalize
  - difficult to apply to multi-dimensional problem

Summary of Variance Reduction Techniques

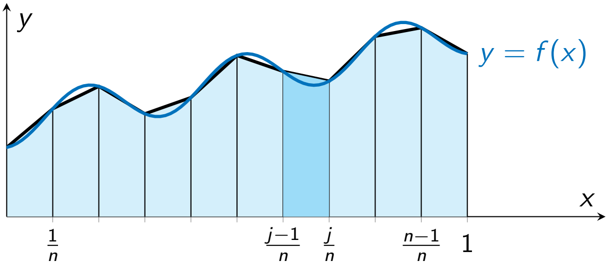
- We have discussed
  - Antithetic sampling
  - Variates recycling
  - Control variates
  - Stratified sampling
  - Importance sampling
- The simple ones are easy to implement and generalize (our preference)
- The complicated ones are difficult to implement and generalize but more effective when tackled in the right way
- But overall the improvement they can make on the convergence is the constant factor in front of $O(n^{-\frac{1}{2}})$. Is there any technique that can change the convergence in terms of magnitude?

Monte Carlo Versus Quadrature

- The Monte Carlo convergence rate is $O(n^{-1/2})$. In contrast, the simple *trapezoidal rule* (quadrature equivalent to Heun's method) estimates the integral of twice continuously differentiable function f's integral by

$$\alpha \approx \frac{f(0)+f(1)}{2n} + \frac{1}{n}\sum_{i=1}^{n-1} f(i/n).$$

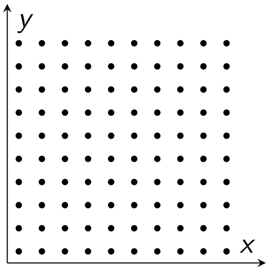
whose error is $O(n^{-2})$.



- It can be seen that Monte Carlo method is not a competitive method for one-dimensional integration.
- However, if we have a 2 dimensional problem and we discretize it to m points for each dimension, the total number $n = m^2$. And a quadrature rule basically says

$$\int_0^1 \int_0^1 f(x,y)dxdy \approx \frac{1}{m^2}\sum_i \sum_j f(x_i, y_j) = \frac{1}{n}\sum_i \sum_j f(x_i, y_j) \quad (64)$$

- The error of the 2D integration is $O(m^{-2}, m^{-2})$ — the lower order of the two dimensions, so $O(n^{-1})$.
- When the dimension d grows, the error of the quadrature remains the maximum of the dimensions. And since $m = n^{\frac{1}{d}}$ in general the trapezoid rule's error is $O(n^{-2/d})$ for d dimensional problem,



- Monte Carlo method remains $O(n^{-1/2})$ due to central limit theorem. Therefore Monte Carlo method is attractive in evaluating high dimensional integrals.
- But let's have a look again at the Monte Carlo estimator:

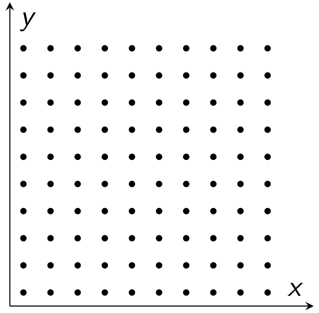
$$\int_{\mathbf{A}} f(\mathbf{X}) d\mathbf{X} \approx \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{X}_i) \tag{65}$$

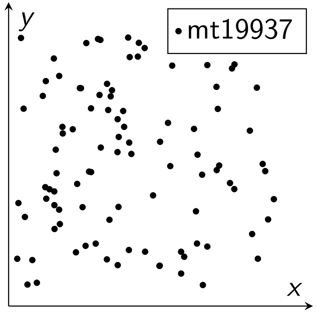
and compare it with the d dimensional quadrature with equal spacing:

$$\int_{\mathbf{A}} f(\mathbf{X}) d\mathbf{X} = \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{X}_i) \tag{66}$$

It's the same formula: taking average of sample evaluations of function $f(\mathbf{X})$. **So what's the difference?**

- The difference lies in the sample points chosen: quadrature chooses regular grid points while Monte Carlo chooses random points:





- Which one looks better? And why?
- The error is associated with the density of the points when you project them to each dimension. Random number is better in the sense that after projection there are still n points.

Quasi Monte Carlo

- Monte Carlo is good at higher dimension but if we can sample more regularly on each dimension, just like what quadrature does in one dimension, can the convergence rate be better?
- Quasi random numbers were proposed to be used with Monte Carlo for calculating integrations — Quasi Monte Carlo.
- The idea is to generate low discrepancy sequence — uniformly distributed without clustering
- The **discrepancy** is measured by

$$D_N^{(d)} = \sup_{\mathbf{y} \in [0,1]^d} \left| \frac{n_{S(\mathbf{y})}}{N} - \Pi_{k=1}^{d} y_k \right| \tag{67}$$

- Quasi Monte Carlo (QMC) has a convergence rate $O\left(\frac{(\log n)^d}{n}\right)$, and for $d << n$ it's approximately $O(n^{-1})$.

Halton Sequence

- Each dimension has a prime number base p_i
- From a generating number $\gamma(n)$ for the n-th draw, convert it to each dimension's base

$$\gamma(n) = \sum_{k=1}^{m_{n_i}} a_{k_i} p_i^{k-1} \tag{68}$$

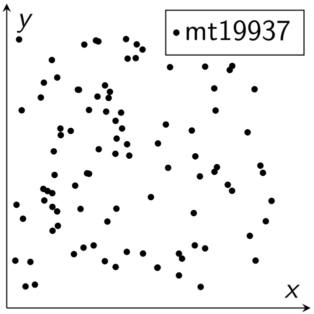
- The quasi random number of the n-th draw for dimension i is then

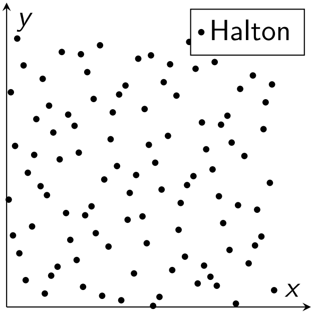
$$u(n) = \sum_{k=1}^{m_{n_i}} a_{k_i} p_i^{-k} \tag{69}$$

- For example, first dimension we let $\gamma(n) = n$ and $p_1 = 2$, the sequence is

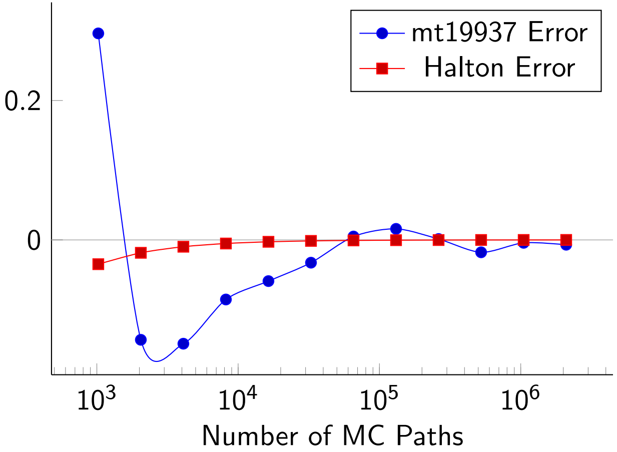
$$\begin{aligned}
1 &\longrightarrow 1_2 & &\longrightarrow 0.1_2 & &\longrightarrow 0.5 \\
2 &\longrightarrow 10_2 & &\longrightarrow 0.01_2 & &\longrightarrow 0.25 \\
3 &\longrightarrow 11_2 & &\longrightarrow 0.11_2 & &\longrightarrow 0.75 \\
4 &\longrightarrow 100_2 & &\longrightarrow 0.001_2 & &\longrightarrow 0.125
\end{aligned}$$

- Example implementation of Halton sequence:





QMC European Convergence

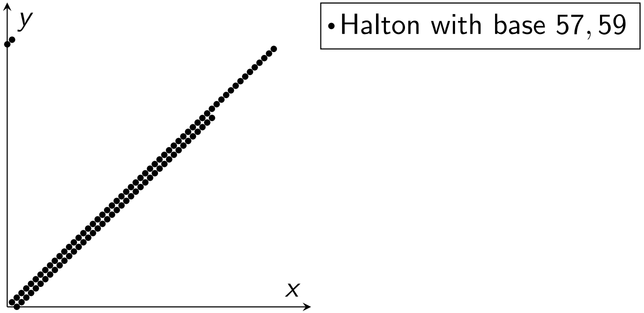


European call option, $K = 100, S = 100, r = 5\%, q = 2\%, T = 1, \sigma = 15\%$

- Note that to convert from uniform distribution to normal distribution, only inverse transformation method works for quasi random numbers

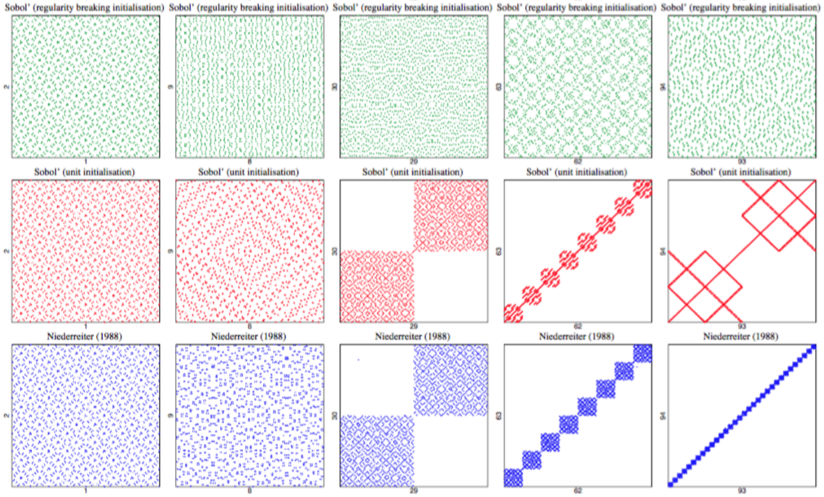
Quasi Random Numbers in Monte Carlo

- When using quasi random numbers in Monte Carlo simulation, we have to be very careful on drawing random numbers
- Each Brownian increment is one dimension, so if we have 100 time steps, we need to draw random vectors of length 100
- The limitation of Halton sequence is very obvious now: when the base number becomes larger, the discrepancy is not that low



Commonly Used Quasi Random Sequences

- Sobol sequences, Niederreiter, and Scrambled version of them



QMC — Dimensionality

- Quasi random numbers (low discrepancy sequences) have limited dimension and the quality of the dimension decreases with larger base.
- It is therefore important to be able utilize the good quality dimensions whose varieties are the most to the important simulation events, for example, option expiries. This can be achieved through
  - Brownian bridge construction: use the first dimension to construct the sample of the expiry date, the second dimension to construct the time in the middle, etc.
  - PCA construction: use the lower dimension numbers to construct the principal components of the Brownian motion, i.e., the skeleton of it.
  - See [2] for more details.

References and Future Readings

- P. Jackel. *Monte Carlo Methods in Finance*. 2002.
- P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2003.