Monte Carlo Method

QF607 Numerical Methods

Zhenke Guan
zhenkeguan@smu.edu.sg

Outline

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 random variables with the same underlying distribution (i.e., i.i.d), with finite expected value $\mu = \mathbb{E}[h(\mathbf{X}_i)]$ and finite variance $v = Var(h(\mathbf{X}_i))$. Let

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

Then for any $\epsilon > 0$,

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

- 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 $\mathcal{U}(0,1)$

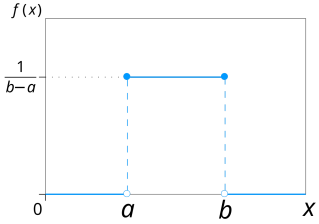
- To integrate the interval $[a, b]$

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

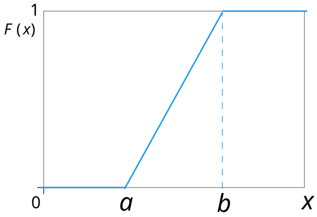
So $\int_a^b f(x)dx = (b-a)\mathbb{E}[f(x)]$ for $x \sim \mathcal{U}(a,b)$

Uniform Distribution

PDF



CDF



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

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

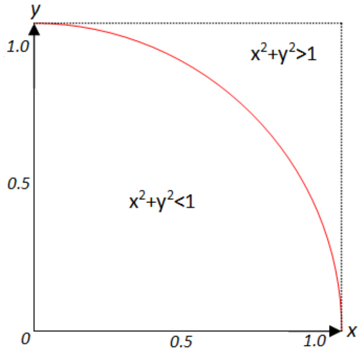
- Example: knowing that a uniformly drawn random point in a 2×2 square has the probability $\frac{\pi}{4}$ to fall inside a the unit circle inscribed within the square, we have

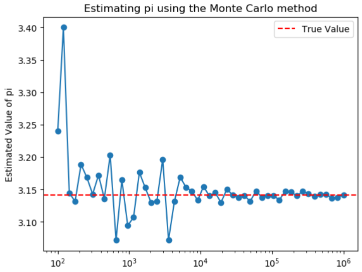
$$P = \frac{\pi}{4} = \mathbb{E}[I_A(x)] \tag{8}$$

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: 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
```





Example: Calculate Integral

Example: Solve

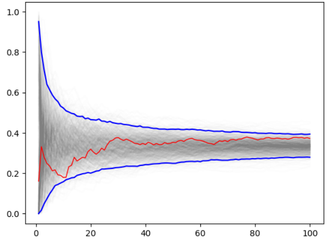
$$\int_0^1 x^2 dx$$

```
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
```

Q: how to update the codes to solve

$$\int_0^3 x^2 dx$$

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.



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

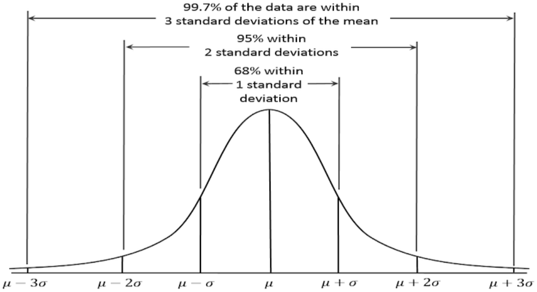
- From central limit theorem we know that our estimator $\hat{\mu}_n$ approaches a normal distribution: $\hat{\mu}_n \to \mathcal{N}(\mu, \frac{\sigma^2}{n})$
- A statistical measure for the uncertainty in any one simulation of result of $\hat{\mu}_n$ is then the standard deviation of $\hat{\mu}_n$: $\frac{\sigma}{\sqrt{n}}$
- 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
$\mu + \sigma$
$\mu + 2\sigma$
$\mu + 3\sigma$

Convergence example

```
def fx(x):
    return x*x;

def integralX2():
 N = 1000000
 accum , hsquare = 0, 0
 for i in range(N):
        x = np.random.uniform(0, 1)
        xsqur = fx(x)
        accum += xsqur
        hsquare += xsqur * xsqur
result = accum/float(N)
stderr = math.sqrt((hsquare/float(N) -result*result)/float(N))

n=10,000, result/stderr:  0.3318172956616281 0.002972303607124641
n=1,000,000, result/stderr:  0.33335766275838513 0.0002981101485913653
```

Variance reduction technique - 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:

$$\mathbb{E}[h(\mathbf{X})] \approx \frac{1}{n}\sum_{i=1}^{n}(h(\mathbf{X}_i) + \beta(g^* - g(\mathbf{X}_i))) \tag{12}$$

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

- The variance of the samples is

$$Var[h] + \beta^2 Var[g] - 2\beta Cov[h, g] \tag{13}$$

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

Control Variates

- The minimized variance is

$$Var[h] - \frac{Cov[h,g]^2}{Var[g]} = Var[h](1-\rho^2) \tag{14}$$

- The higher correlation g and h is, the more effective the technique is.
- With variance reduction technique, we need fewer simulation and thus a shorter run time.
- β can be estimated using an initial simulation with fewer iterates than the main one.
- Cannot be applied in a general way, need to fine tune based on the problem

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).

Control Variates example

Example: Solve

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

with

$$g(x) = x$$

as control variate
Solution:

- $f(x) = x^2$
- $g(x) = x$
- $g^* = 0.5$
- $beta = 1$ if solve analytically or numericaly using simulation samples (few paths of course)

Control Variates example

```
def integralX2WithCV():
    N = 1000000
    accum , hsquare = 0, 0
    beta = estimateBeta()
    print("beta:", beta)
    mean_gxValue = mean_gx()
    for i in range(N):
        x = np.random.uniform(0, 1)
        fxValue = fx(x) + beta *( mean_gxValue -  gx(x))
        accum += fxValue
        hsquare += fxValue * fxValue
    result = accum/float(N)
    stderr = math.sqrt((hsquare/float(N) - result*result)/float(N))
    print("result/stderr with CV: ", result, stderr)

beta: 0.9961436953790417
result/stderr with CV:  0.33333896559524545 7.451386799977835e-05
result/stderr:  0.3330024789036267 0.0002982051624623227

```

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

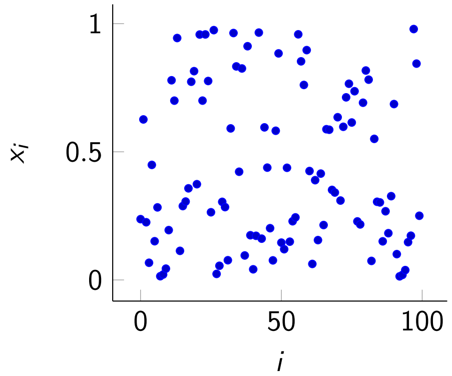
3. Use a specific formula to obtain a new uniform (0, 1) variate from the current state variables

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

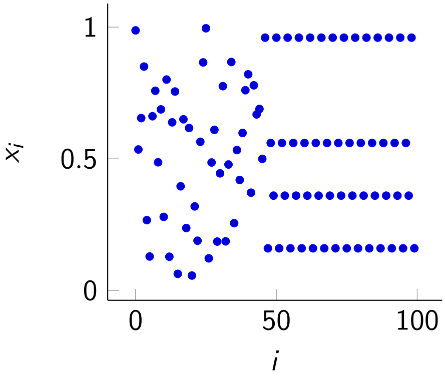


PRNs — mid-square method seed at 0.2372

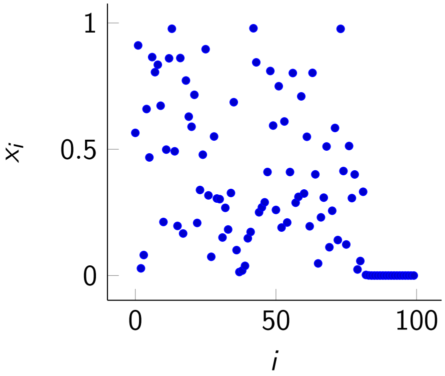
```python
import matplotlib.pyplot as plt

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:

$$m_{n+1} = (a\ m_n + c) \mod M \tag{15}$$

  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

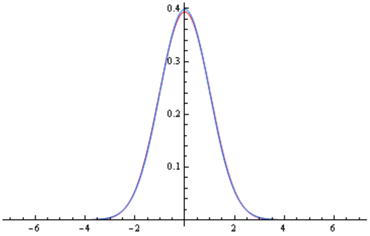
- Presented by Matsumoto and Nishimura in 1998. It was given this name because it has a period of $2^{19937} - 1$ called the Mersenne prime.
- Utilizes many existing methods to rectify most of the flaws found in older PRNGs
- Full algorithm can be found at Wikipedia
- The period of the sequence is a 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**.

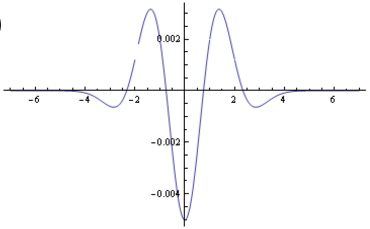
Uniform To Normal Distribution — Central Limit Theorem

- 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

$$m_k = \sum_{i=1}^{12} u_i - 6 \tag{16}$$

  - 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

$$P\left(u \leq \frac{f(x)}{cg(x)}\right) = \int_{\mathcal{X}} \frac{f(x)}{cg(x)} g(x)dx = \frac{1}{c} \tag{17}$$

- Thus y has the desired distribution:

$$P(y \in A) = c\ P\left(x \in A, u \leq \frac{f(x)}{cg(x)}\right) = c\int_A \frac{f(x)}{cg(x)} g(x)dx = \int_A f(x)dx \tag{18}$$

- (17) 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 $x$
```

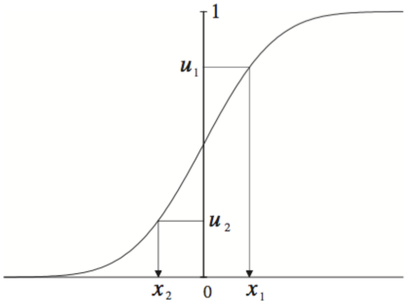
- An algorithm similar to Box-Muller method for generating normal random numbers
- Faster than Box-Muller despite the acceptance rate at first step is $\frac{\pi}{4}$

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

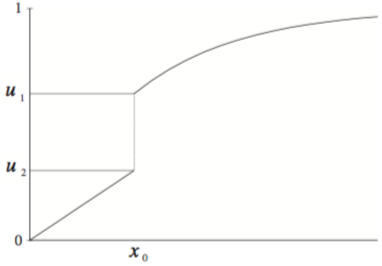
$$X = CDF^{-1}(u), \quad u \sim \mathcal{U}(0,1) \tag{19}$$

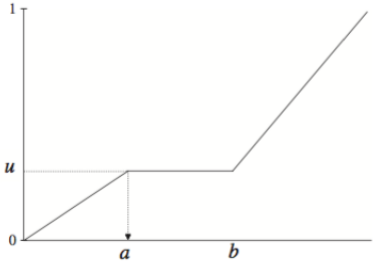
- Draw u from $\mathcal{U}(0,1)$ and from $CDF^{-1}(\cdot)$ we get the sample x
- We used the inverse transformation method to convert uniform random number to exponential distribution in Box-Muller



Inverse Transformation Method — Considerations

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





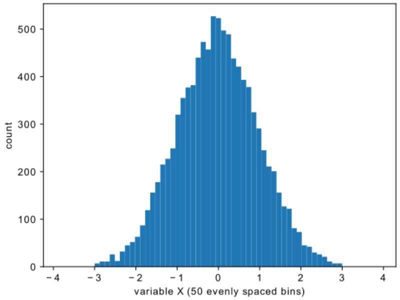
- There can be many to one (jump in CDF), one to many (flat CDF)
- If we use inverse transformation method we need to handle the one to many case with pre-defined conventions

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

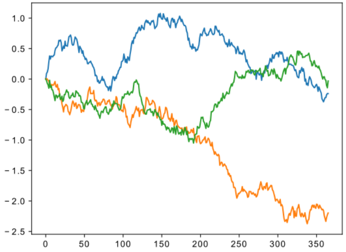
PRNG — Implementation

```
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:

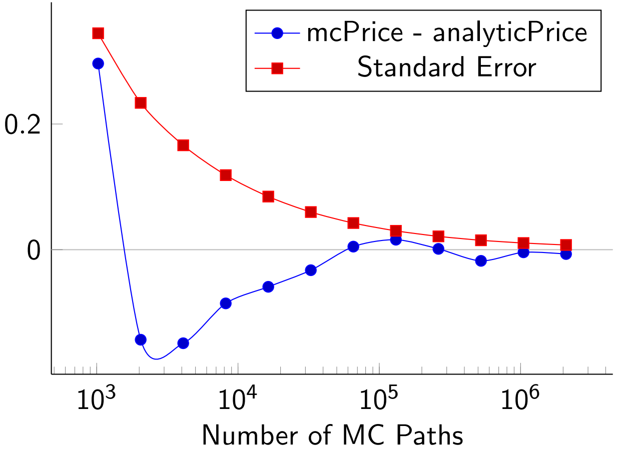
$$S_T = S_0 e^{(r-q-\frac{1}{2}\sigma^2)T+\sigma W_T} \tag{20}$$

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

Summary

This session, we covered:

- MC Theory and Implementation
- Random Number Generation Methods : Uniform and from Uniform to Normal
- Price Option using 1-Step MC under BS model

Next session, we will cover:

- MC for Model Without Closed Form
- Discretization Schemes
- Multi Factor MC
- Generic MC Framework
- Variance Reduction Techniques
- Quasi MC

References and Future Readings

P. Jackel. *Monte Carlo Methods in Finance*. 2002.

P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2003.