Monte Carlo Method - Part 2

QF607 Numerical Methods

Zhenke Guan
zhenkeguan@smu.edu.sg

Outline

Last 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

This session, we will cover:

- Discretization Scheme
  - Euler Scheme
  - Milstein Scheme
- Multi-dimensional MC
- Variance Reduction Techniques
- Quasi Monte Carlo

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:

$$dS = S(r-q)dt + S\sigma(S,t)dW_t \tag{1}$$

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:

$$dX(t) = a(X(t))dt + b(X(t))dW(t). \tag{2}$$

- Integrating (2) we have

$$X(t_{i+1}) = X(t_i) + \int_{t_i}^{t_{i+1}} a(X(u))du + \int_{t_i}^{t_{i+1}} b(X(u))dW_u \tag{3}$$

- To solve the integrals we need to discretize them

Euler Discretization Scheme

- Using time step Δt, the Euler discretization scheme approximates

$$\int_t^{t+\Delta t} a(X(u))du = a(X(t))\Delta t \tag{4}$$

$$\int_t^{t+\Delta t} b(X(u))dW_u = b(X(t))\Delta W = b(X(t))\sqrt{\Delta t}Z \tag{5}$$

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

- The stepwise induction is

$$X(t+\Delta t) = X(t) + a(X(t))\Delta t + b(X(t))\sqrt{\Delta t}Z \tag{6}$$

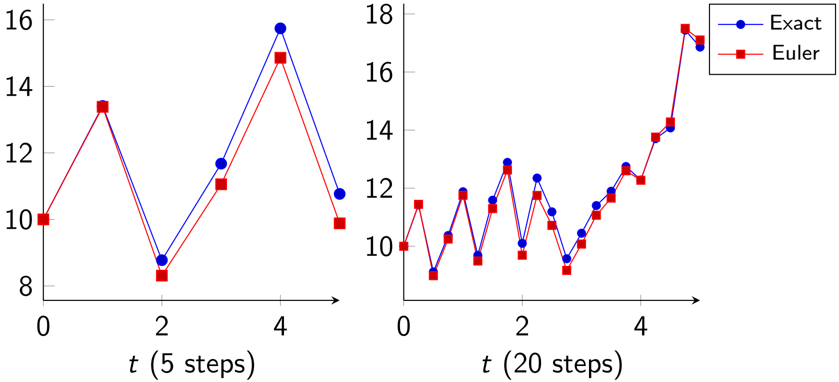
- The truncation error of the drift term is $O(\Delta t)$
- The truncation error of the diffusion term is $O(\Delta W)$, i.e., $O(\sqrt{\Delta t})$
- The overall convergence order of Euler scheme is therefore $O(\sqrt{\Delta t})$

Euler Scheme For Black Scholes Model

- For Black Scholes model Euler scheme can be written as

$$S_{t+\Delta t} = S_t(1 + \mu \Delta t + \sigma \Delta W_t) \tag{7}$$

- 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

$$dX = \frac{\partial X}{\partial S} dS + \frac{1}{2} \frac{\partial^2 X}{\partial S^2} dS^2 = \left(\mu - \frac{1}{2}\sigma^2 \right) dt + \sigma dW_t \tag{8}$$

- Euler scheme now gives us the exact solution:

$$X_{t+\Delta t} = X_t + (\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma \Delta W_t \quad \leftarrow \text{ Euler scheme} \tag{9}$$

$$S_{t+\Delta t} = S_t e^{(\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma \Delta W_t} \quad \leftarrow \text{ Reconstruction} \tag{10}$$

- 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:

$$dX = \left(\mu - \frac{1}{2}\sigma(S,t)^2\right) dt + \sigma(S,t) dW_t \tag{11}$$

- 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:

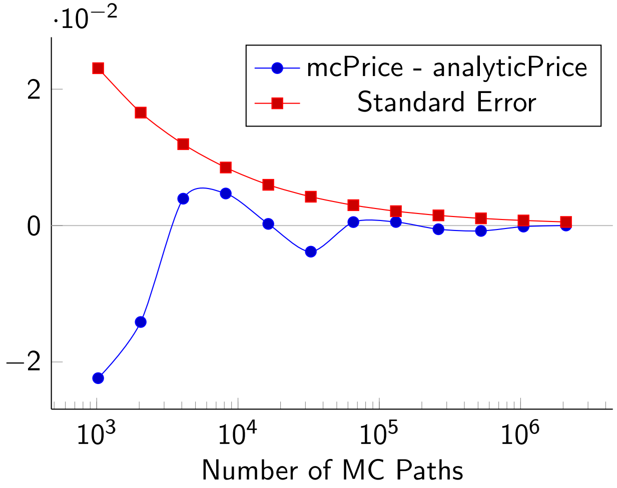
$$X_{t+\Delta t} = X_t + (\mu - \frac{1}{2}\sigma(X,t)^2)\Delta t + \sigma(X,t)\Delta W_t \tag{12}$$

- 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

$$\int_t^u b(X(u))b'(X(u))dW_u = b(X(t))b'(X(t))(W_u - W_t) + O(u-t) \tag{13}$$

- Therefore

$$b(X(u)) = b(X(t)) + b(X(t))b'(X(t))(W_u - W_t) + O(u-t). \tag{14}$$

- Now, come back to the target integral (3), the result is

$$\int_{t_i}^{t_{i+1}} b(X(u))dW_u = \underbrace{b(X(t))\sqrt{\Delta t}Z}_{\text{Euler term}} + \underbrace{\frac{1}{2}b(X(t))b'(X(t))\Delta t(Z^2-1)}_{\text{Milstein correction term}} \tag{15}$$

Milstein Discretization Scheme

And here is the last step of derivation of (15):

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

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

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

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

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

Discretizing local volatility model diffusion we have:

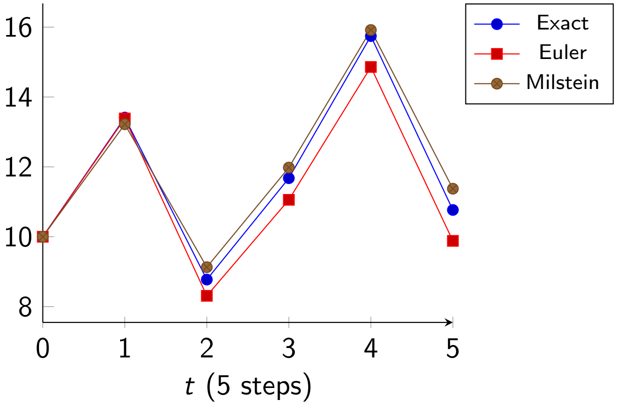
$$X_{t+\Delta t} = X_t + (\mu - \frac{\sigma(X,t)^2}{2})\Delta t + \sigma(X,t)\Delta W_t + \boxed{\frac{\sigma(X,t)}{2}\frac{\partial \sigma(X,t)}{\partial X}(\Delta W_t^2 - \Delta t)} \tag{16}$$

Error of Milstein Scheme

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

$$S_{t+\Delta t} = S_t(1 + \mu\Delta t + \sigma\Delta W_t + \frac{\sigma^2}{2}(\Delta W_t^2 - \Delta t)) \qquad (17)$$

- 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

$$\rho = \frac{Covar}{\sqrt{Var_1 \cdot Var_2}} = \frac{\mathbb{E}(\Delta W_1 \Delta W_2) - \mathbb{E}(\Delta W_1)E(\Delta W_2)}{\sqrt{Var(\Delta W_1) \cdot Var(\Delta W_2)}} \tag{18}$$

$$= \frac{\mathbb{E}(\Delta W_1 \Delta W_2)}{dt} \tag{19}$$

So

$$\mathbb{E}(\Delta W_1 \Delta W_2) = \rho dt \tag{20}$$

$$\Delta W_1 = \xi_1 \sqrt{dt} \tag{21}$$

$$\Delta W_2 = \xi_2 \sqrt{dt} \tag{22}$$

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

$$\mathbb{E}(\xi_1 \xi_2) = \rho \tag{23}$$

- If we represent the correlation matrix as

$$\mathbf{C} = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \tag{24}$$

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

$$\hat{\boldsymbol{\xi}}_1 = \left[\hat{\xi}_{1,0}, \quad \hat{\xi}_{1,1}, \quad \dots, \quad \hat{\xi}_{n,1}\right], \tag{25}$$

$$\hat{\boldsymbol{\xi}}_2 = \left[\hat{\xi}_{2,1}, \quad \hat{\xi}_{2,2}, \quad \dots, \quad \hat{\xi}_{2,n}\right] \tag{26}$$

and

$$\begin{bmatrix} \hat{\boldsymbol{\xi}}_1 \\ \hat{\boldsymbol{\xi}}_2 \end{bmatrix} \times \begin{bmatrix} \hat{\boldsymbol{\xi}}_1^\top, & \hat{\boldsymbol{\xi}}_2^\top \end{bmatrix} = \begin{pmatrix} \hat{\boldsymbol{\xi}}_1 \cdot \hat{\boldsymbol{\xi}}_1, & \hat{\boldsymbol{\xi}}_1 \cdot \hat{\boldsymbol{\xi}}_2 \\ \hat{\boldsymbol{\xi}}_2 \cdot \hat{\boldsymbol{\xi}}_1, & \hat{\boldsymbol{\xi}}_2 \cdot \hat{\boldsymbol{\xi}}_2 \end{pmatrix} = \begin{pmatrix} n, & n\rho \\ n\rho, & n \end{pmatrix} = n\,\mathbf{C} \tag{27}$$

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

$$\mathbf{C} = \mathbf{L}\mathbf{L}^\top \tag{28}$$

We have

$$\begin{bmatrix} \hat{\xi}_1 \\ \hat{\xi}_2 \end{bmatrix} \times \begin{bmatrix} \hat{\xi}_1^\top, & \hat{\xi}_2^\top \end{bmatrix} = n\, \mathbf{L}\mathbf{L}^\top \tag{29}$$

So

$$\mathbf{L}^{-1} \begin{bmatrix} \hat{\xi}_1 \\ \hat{\xi}_2 \end{bmatrix} \times \begin{bmatrix} \hat{\xi}_1^\top, & \hat{\xi}_2^\top \end{bmatrix} (\mathbf{L}^\top)^{-1} = n\, \mathbf{I} \tag{30}$$

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

Correlating Standard Normal Vectors

- If we let

$$\begin{bmatrix} \hat{\zeta}_1 \\ \hat{\zeta}_2 \end{bmatrix} = (\mathbf{L})^{-1} \begin{bmatrix} \hat{\xi}_1 \\ \hat{\xi}_2 \end{bmatrix} \tag{31}$$

  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

$$\boxed{\begin{bmatrix} \hat{\xi}_1 \\ \hat{\xi}_2 \end{bmatrix} = \mathbf{L} \begin{bmatrix} \hat{\zeta}_1 \\ \hat{\zeta}_2 \end{bmatrix}} \tag{32}$$

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:

$$\mathbf{C} = \mathbf{L}\mathbf{L}^\top \tag{33}$$

  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 $\rho \neq \pm 1$:

$$\mathbf{x}^\top C\mathbf{x} = \frac{1}{n} \underbrace{\mathbf{x}^\top \begin{bmatrix} \hat{\xi}_1 \\ \hat{\xi}_2 \end{bmatrix}}_{(\mathbf{y}^\top)_{1\times n}} \times \underbrace{\begin{bmatrix} \hat{\xi}_1^\top, & \hat{\xi}_2^\top \end{bmatrix} \mathbf{x}}_{\mathbf{y}_{n\times 1}} = \frac{\parallel \mathbf{y} \parallel}{n} \geq 0 \tag{34}$$

  So (33) 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:

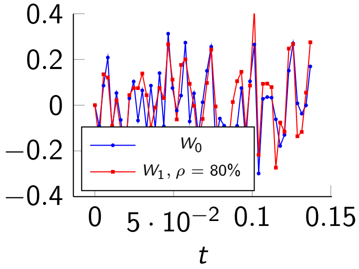
$$\mathbf{C} = \mathbf{L}\mathbf{L}^\top = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{23} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{23} \\ 0 & 0 & L_{33} \end{pmatrix} \tag{35}$$

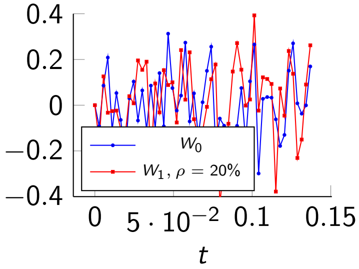
$$= \begin{pmatrix} L_{11}^2 & & \text{(symmetric)} \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & \sum_{i=1}^3 L_{3i}^2 \end{pmatrix} \tag{36}$$

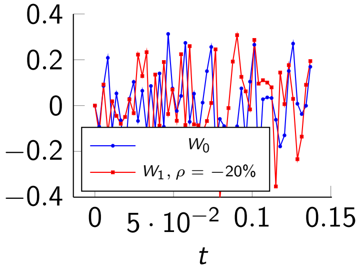
  Just need to start from the top-left corner and progressively solve for each L_{ij}
- In python implementation: `L = np.linalg.cholesky(C)`

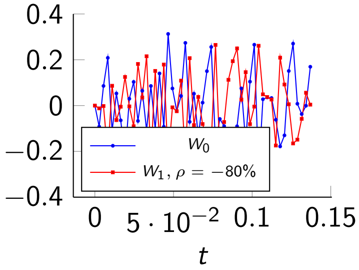
Correlated Brownian Motions — Examples

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





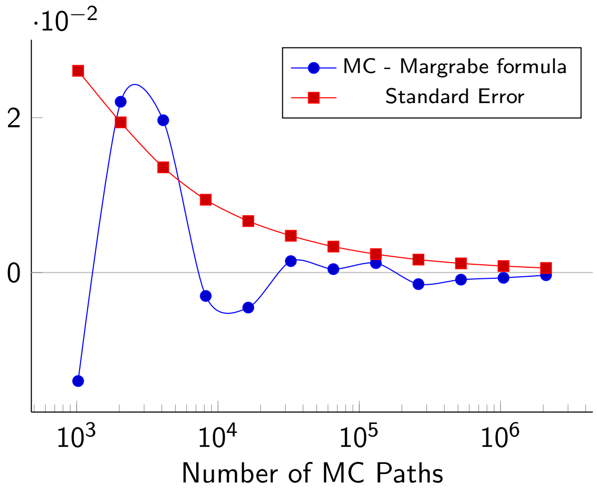




Pricing Spread Option — MC with Two Assets under BS

```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:

$$\max\left(\frac{1}{n}\sum_{i=1}^{n} S(t_i) - K, 0\right) \tag{37}$$

- 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:

$$C_i = S(t_i) - K \tag{38}$$

  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

$$Var\left[\frac{1}{2}(h(W_i) + h(-W_i))\right] = \frac{1}{2}Var[h(W_i)] + \frac{1}{2}Cov[h(W_i), h(-W_i)] \tag{39}$$

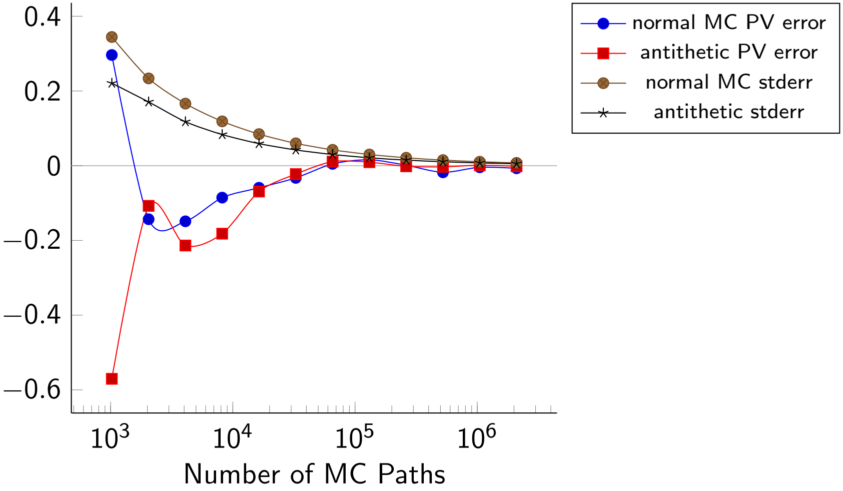
- When we compare the variance with the standard Monte Carlo, we should compare with the case with twice amount of simulation paths, whose variance is $\frac{1}{2}Var[h(W_i)]$
- So if $Cov[h(W_i), h(-W_i)] < 0$ the Monte Carlo variance is reduced by antithetic sampling — always true if h is monotonic 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:

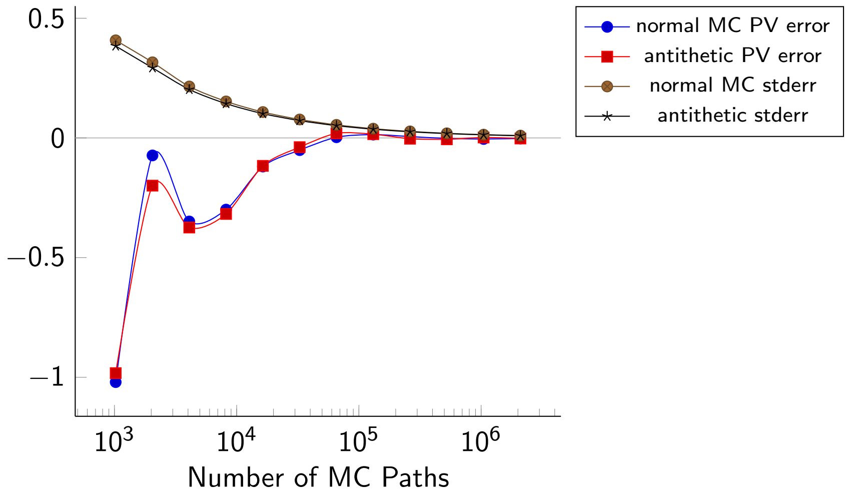
```
sum,hsquare = 0,0
stdev = math.sqrt(T)
for i in range(nPaths):
    wT = np.random.normal(0, stdev)
    hA = trade.payoff(S0 * math.exp((r-q-0.5*vol*vol)*T + vol*wT))
    hB = trade.payoff(S0 * math.exp((r-q-0.5*vol*vol)*T - vol*wT))
    h = 0.5 * (hA + hB)
    sum += h
    hsquare += h * h
pv = math.exp(-r*T)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

$$\delta = \frac{\partial PV}{\partial S} \approx \frac{PV(S + \Delta S) - PV(S - \Delta S)}{2\Delta S} \tag{40}$$

- 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:

$$\mathbb{E}[h(\mathbf{X})] \approx \frac{1}{n}\sum_{i=1}^{n}(h(\mathbf{X}_i) + \beta(g^* - g(\mathbf{X}_i))) \tag{41}$$

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

- The variance of the samples is

$$Var[h] + \beta^2 Var[g] - 2\beta Cov[h, g] \tag{42}$$

- 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

$$Var[h] - \frac{Cov[h,g]^2}{Var[g]} = Var[h](1-\rho^2) \tag{43}$$

- 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

$$\mathbb{E}[h(\mathbf{X})] = \sum_{j=1}^{k} \mathbb{E}[h(\mathbf{X}|\mathbf{X} \in \mathbf{A}_j]p(\mathbf{X} \in \mathbf{A}_j) \qquad (44)$$

- 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

$$\mathbb{E}_{\mathbb{Q}}[h(\mathbf{X})] = \int h(\mathbf{X})q(\mathbf{X})d\mathbf{X} = \int h(\mathbf{X})\frac{q(\mathbf{X})}{p(\mathbf{X})}p(\mathbf{X})d\mathbf{X} \tag{45}$$

$$= \mathbb{E}_{\mathbb{P}}\left[h(\mathbf{X})\frac{q(\mathbf{X})}{p(\mathbf{X})}\right] \tag{46}$$

- 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

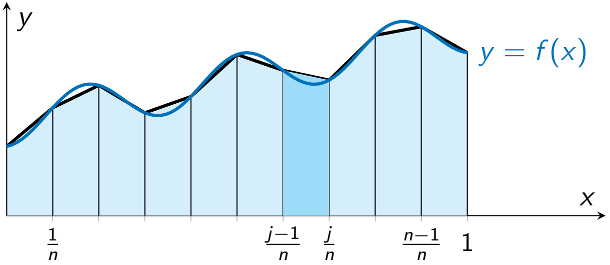
- We have discussed
  - Antithetic sampling
  - Variates recycling
  - Control variates
  - Stratified sampling
  - Importance sampling
- The simple ones are easy to implement and generalize (our preference)
- The complicated ones are difficult to implement and generalize but more effective when tackled 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

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

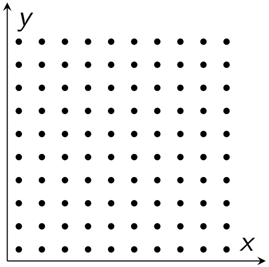
whose error is $O(n^{-2})$.



- 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

$$\int_0^1 \int_0^1 f(x,y)dxdy \approx \frac{1}{m^2}\sum_i \sum_j f(x_i, y_j) = \frac{1}{n}\sum_i \sum_j f(x_i, y_j) \quad (47)$$

- 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:

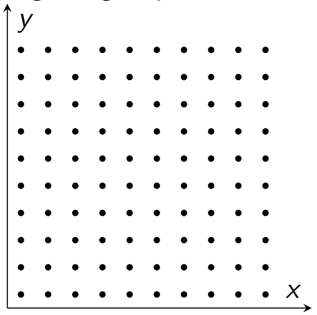
$$\int_{\mathbf{A}} f(\mathbf{X}) d\mathbf{X} \approx \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{X}_i) \tag{48}$$

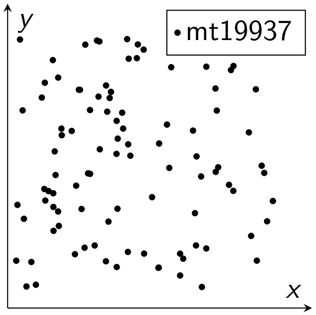
and compare it with the d dimensional quadrature with equal spacing:

$$\int_{\mathbf{A}} f(\mathbf{X}) d\mathbf{X} = \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{X}_i) \tag{49}$$

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

$$D_N^{(d)} = \sup_{\mathbf{y}\in[0,1]^d} \left|\frac{n_{S(\mathbf{y})}}{N} - \Pi_{k=1}^d y_k\right| \tag{50}$$

- 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

$$\gamma(n) = \sum_{k=1}^{m_{n_i}} a_{k_i} p_i^{k-1} \tag{51}$$

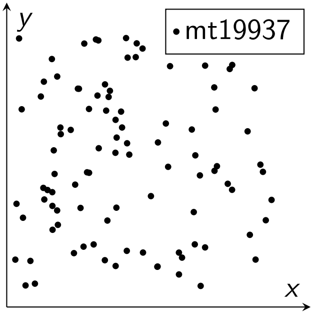
- The quasi random number of the n-th draw for dimension i is then

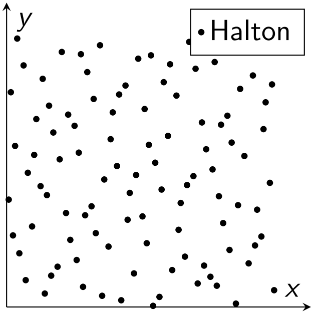
$$u(n) = \sum_{k=1}^{m_{n_i}} a_{k_i} p_i^{-k} \tag{52}$$

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

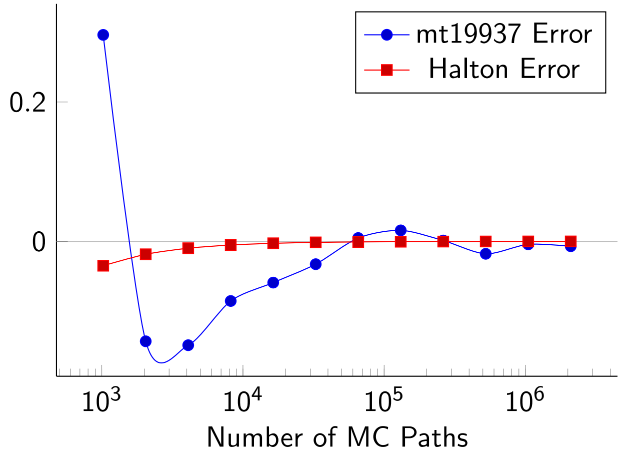
$$\begin{aligned}
1 &\longrightarrow 1_2 & &\longrightarrow 0.1_2 & &\longrightarrow 0.5\\
2 &\longrightarrow 10_2 & &\longrightarrow 0.01_2 & &\longrightarrow 0.25\\
3 &\longrightarrow 11_2 & &\longrightarrow 0.11_2 & &\longrightarrow 0.75\\
4 &\longrightarrow 100_2 & &\longrightarrow 0.001_2 & &\longrightarrow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

- Example implementation of Halton sequence:





QMC European Convergence

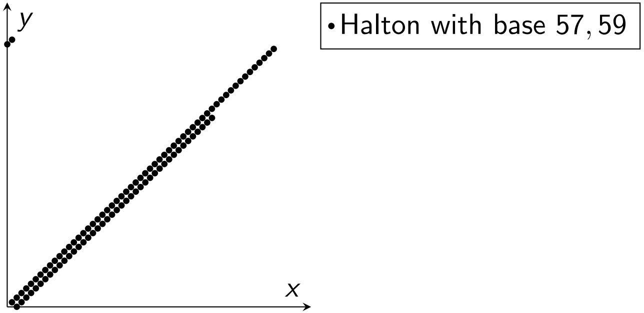


European call option, $K = 100, S = 100, r = 5\%, q = 2\%, T = 1, \sigma = 15\%$

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

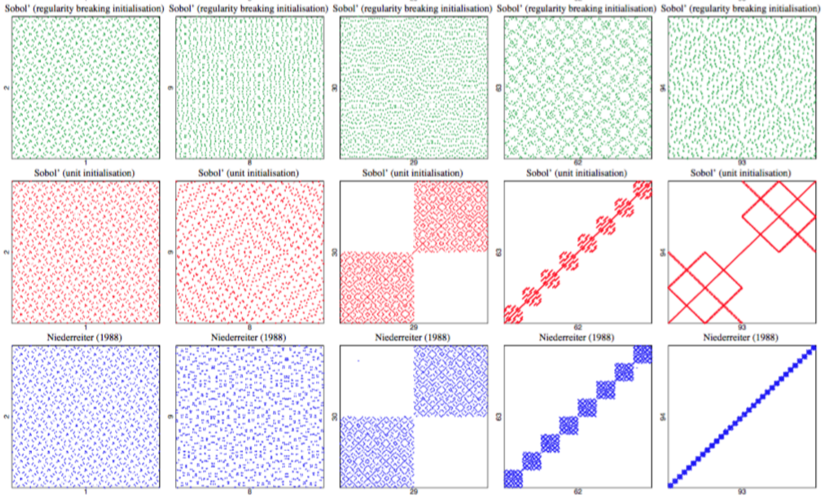
Quasi Random Numbers in Monte Carlo

- 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.

Other important topics not covered

Some examples:

- Simulation of other processes (CIR, Heston)

- MC for callable products using regression based method (LS)

References and Future Readings

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

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