Lecture 10: Monte-Carlo Simulation

Cheng Lu

Overview

- Black-Scholes Model (Review)
- Monte-Carlo Simulation
- Improve Efficiency for MC Simulation

Black-Scholes Model

Black-Scholes Model is a mathematical model for pricing financial instruments (or financial derivatives). The main assumption is that the stock price follows geometric Brownian Motion

$$dS(t) = \mu S(t)dt + \sigma S(t)dW(t), S(0) = S_0$$

where μ is the constant drift, σ is the constant volatility, and $W(t)$ is a Brownian Motion. Under risk neutral measure, the stock dynamic is given by

$$dS(t) = rS(t)dt + \sigma S(t)dW^Q(t), S(0) = S_0$$

where r is the risk free rate (or interest rate).

Black-Scholes Model

The payoff of the a call option[1] is

$$(S(T) - K)_+ = \max\{S(T) - K, 0\}$$

Then the risk neutral pricing formula implies the call option price c at time 0 is given by

$$c(S_0, K, T, \sigma, r) = e^{-rT} E^Q[(S(T) - K)_+] = S_0 N(d_1) - e^{-rT} K N(d_2)$$

$$d_1 = \frac{\ln \frac{S_0}{K} + (r + \frac{1}{2}\sigma^2) T}{\sigma\sqrt{T}}; d_2 = \frac{\ln \frac{S_0}{K} + (r - \frac{1}{2}\sigma^2) T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}$$

where $N(\cdot)$ is the cumulative distribution function of standard normal distribution (i.e. mean $= 0$, sd $= 1$). And dividend is not considered here.

[1] Recall a call option is a contract that the investor has right to buy the underlying stock at specific time T with specific price K. Here T is called maturity (or expiration date, expiry date) and K is called strike price.

Black-Sccholes Model

Then we can write it into a function

Example

```
> bs.call <- function(S0, K, T1, sigma, r){
+   d1 <- (log(S0/K) + (r+0.5*sigma^2)*T1)/(sigma*sqrt(T1))
+   d2 <- d1 - sigma*sqrt(T1)
+   S0*pnorm(d1) - exp(-r*T1)*K*pnorm(d2)
+   # return(S0*pnorm(d1) - exp(-r*T1)*K*pnorm(d2))
+ }
> bs.call(100, 100, 1, r = 0.05, sigma = 0.2)
[1] 10.45058
> bs.call(100, 100, 1, 0.2, 0.05)
[1] 10.45058
```

Monte-Carlo Simulation

Monte-Carlo simulation is a technique to obtain numerical results by random sampling.

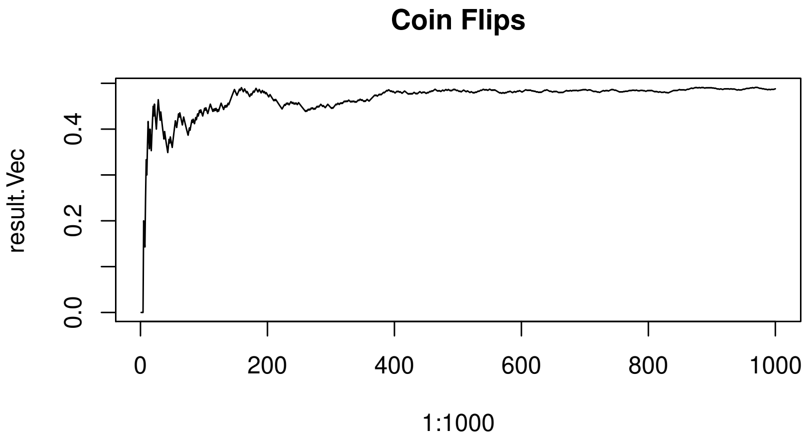
- Recall coin flip example in L3. X_j be the outcome from flipping a coin, 1 for heads and 0 for tails. Then $\frac{1}{m}\sum_{j=1}^{m} X_j \to E[X_j] = 0.5, j = 1, \ldots, m$
- Recall mean and sample mean in L6. Sample mean is an unbiased estimator for population mean

Strong Law of Large Number

Suppose $X_1, X_2, \ldots, X_m$ are i.i.d. (independent identically distributed), let $E[X_1] = \mu < \infty$. Then sample mean converge to population mean almost surely. i.e.

$$P\left(\lim_{m\to\infty} \frac{1}{m}\sum_{j=1}^{m} X_j = \mu\right) = 1$$

Monte-Carlo Simulation



Monte-Carlo Simulation

For Black-Scholes model, we can do the same way, the call option price is

$$c(S_0, K, T, \sigma, r) = e^{-rT} E^Q[(S(T) - K)_+]$$

We can approximate the expectation by it sample mean:

$$e^{-rT} E^Q[(S(T) - K)_+] \approx e^{-rT} \frac{1}{m} \sum_{j=1}^{m} (S^{(j)}(T) - K)_+$$

where $S^{(j)}(T)$ is the jth copy of $S(T)$.

Since the risk neutral dynamic of S is

$$dS(t) = rS(t)dt + \sigma S(t)dW^Q(t), S(0) = S_0$$

Similar to ordinary differential equations, we can approximate each copy of $S(T)$ by replacing differential to difference.
Define a partition of $[0, T]$ by $0 = t_0 < t_1 < \ldots < t_n = T$.

- Replace $dS(t)$ by $S(t_{i+1}) - S(t_i)$
- Replace dt by $t_{i+1} - t_i$
- Replace $dW^Q(t)$ by $W^Q(t_{i+1}) - W^Q(t_i)$

then

$$S(t_{i+1}) - S(t_i) \approx rS(t_i)(t_{i+1} - t_i) + \sigma S(t_i)(W^Q(t_{i+1}) - W^Q(t_i))$$

Monte-Carlo Simulation

- Let $S_i = S(t_i)$, $W_i^Q = W^Q(t_i)$
- Let $t_i = \frac{iT}{n}$ and $h = \frac{T}{n} = t_{i+1} - t_i$ for all $i = 1, \dots, n$

Then[2]

$$\begin{aligned} S_{i+1} &= S_i + rS_ih + \sigma S_i(W_{i+1}^Q - W_i^Q) \\ &= S_i + rS_ih + \sigma S_i\sqrt{h}Z_i, i = 0, \dots, n-1 \end{aligned}$$

where $\sqrt{h}Z_i = W_{i+1}^Q - W_i^Q \sim N(0, h)$, and $Z_i \sim N(0, 1)$.

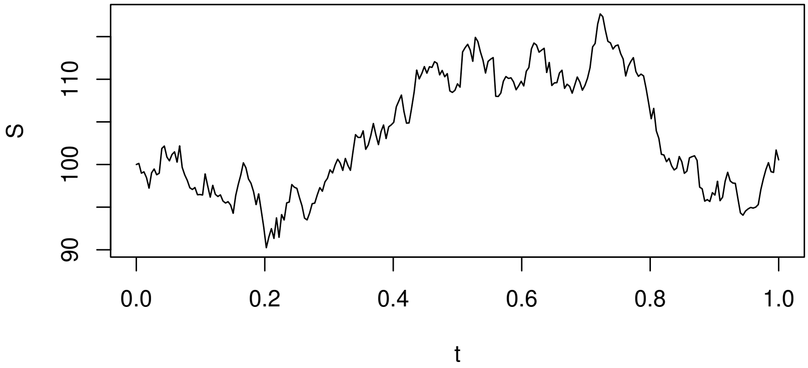
[2]This is called Euler discretization, and $S_n \approx S(t_n) = S(T)$.

Monte-Carlo Simulation

Example

```
> S0 <- 100
> K <- 100
> T1 <- 1
> sigma <- 0.2
> r <- 0.05
> n <- 252 # number of steps
> h <- T1/n # one day time difference
>
> S <- c() # same as "S <- NULL"
> Z <- c()
> S[1] <- S0 # start at S[1], since S[0] not available in R
> for (i in 1:n) {
+   Z[i] <- rnorm(1)
+   S[i+1] <- S[i] + r*S[i]*h + sigma*S[i]*Z[i]*sqrt(h)
+ }
> t <- seq(from = 0, to = T1, by = h)
> plot(t, S, type = "l")
```

Monte-Carlo Simulation



Monte-Carlo Simulation

Since geometric Brownian Motion has explicit solution

$$S(t_i) = S_0 \exp\left((r - \frac{1}{2}\sigma^2)t_i + \sigma W(t_i)\right)$$

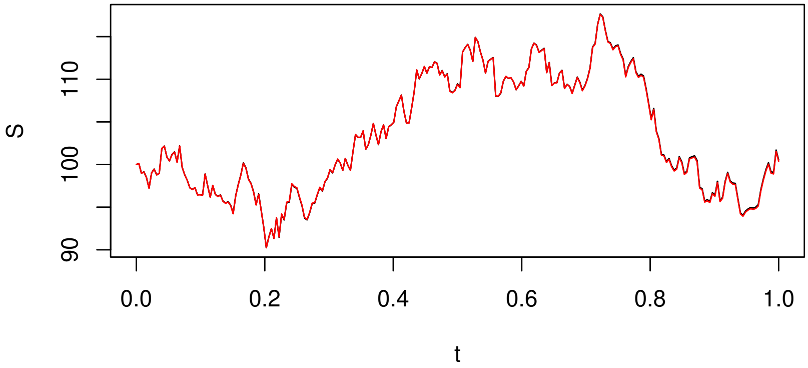
and we have

$$W(t_i) = W_i = \sum_{j=0}^{i-1}(W_{j+1} - W_j) = \sum_{j=0}^{i-1}\sqrt{h}Z_j$$

Example

```
> W <- c(0, cumsum(sqrt(h)*Z))
> S.explicit <- S0*exp((r - 0.5*sigma^2)*t + sigma*W)
> lines(t, S.explicit, col = "red")
```

Monte-Carlo Simulation



Monte-Carlo Simulation

Example

```
> S[n+1]
[1] 101.7012
> S.explicit[n+1]
[1] 101.5435
```

- S is an approximation of S.explicit
- The difference of S and S.explicit is the discretization error for Euler scheme
- The error goes to 0 as n goes to infinity

For pricing a call option, we need to simulate m paths, then

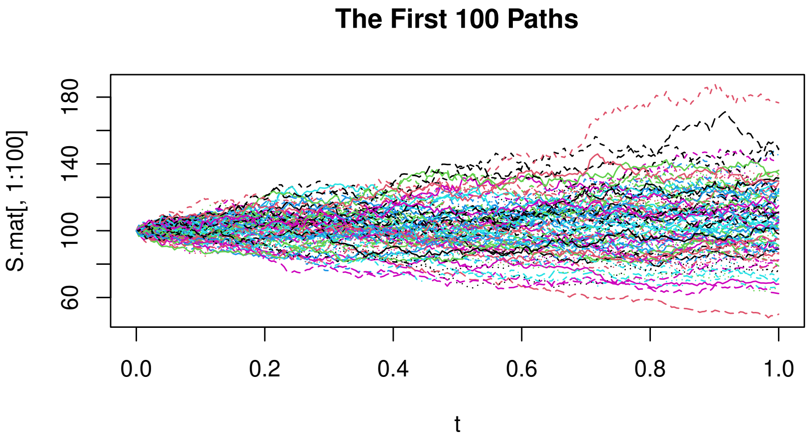
$$c(S_0, K, T, \sigma, r) = e^{-rT} E^Q[(S(T) - K)_+] \approx e^{-rT} \frac{1}{m} \sum_{j=1}^{m} (S^{(j)}(T) - K)_+$$

Monte-Carlo Simulation

Example

```
> m <- 10000
> S.mat <- NULL
> for(j in 1:m){
+     for (i in 1:n) {
+         Z[i] <- rnorm(1)
+         S[i+1] <- S[i] + r*S[i]*h + sigma*S[i]*Z[i]*sqrt(h)
+       }
+     S.mat <- cbind(S.mat,S)
+ }
> plot(t,S.mat[,1], type = "l", main = "The First Path")
> plot(t,S.mat[,2], type = "l", main = "The Second Path")
> matplot(t, S.mat[,1:100], type = "l", main = "The First 100 Paths")
> ST <- S.mat[n+1,] # S(T) for all paths as a vector
> exp(-r*T1)*mean(pmax(ST - 100, 0)) # call option price from MC simulation
[1] 10.20994
> bs.call(S0, 100, T1, sigma, r)# call option price from BS formula
[1] 10.45058
```

Monte-Carlo Simulation



Improve Efficiency for MC Simulation

Since the simulation take long time, we can use **system.time()** to estimate the CPU time

Example

```
> MC <- function(){
+   m <- 10000
+   S.mat <- NULL
+   for (j in 1:m) {
+     S[1] <- S0
+     for(i in 1:n){
+       Z[i] <- rnorm(1)
+       S[i+1] <- S[i] + r*S[i]*h + sigma*S[i]*Z[i]*sqrt(h)
+     }
+     S.mat <- cbind(S.mat, S)
+   }
+   print(exp(-r*T1)*mean(pmax(S.mat[n+1,] - 100, 0)))
+ }
> system.time(MC())
[1] 10.22624
   user  system elapsed
  42.21   23.78   66.11
```

Improve Efficiency for MC Simulation

More efficient if we change the "for" loop and **cbind()** to apply functions e.g. **replicate()**

Example

```
> MC1 <- function(){# use replicate
+   m <- 10000
+   one.path <- function(){
+     S[1] <- S0
+     for(i in 1:n){
+       Z[i] <- rnorm(1)
+       S[i+1] <- S[i] + r*S[i]*h + sigma*S[i]*Z[i]*sqrt(h)
+     }
+     return(S)
+   }
+   S.mat <- replicate(m, one.path())
+   print(exp(-r*T1)*mean(pmax(S.mat[n+1,] - 100, 0)))
+ }
> system.time(MC1())
[1] 10.69769
   user  system elapsed
   7.73    0.03    7.80
```

Improve Efficiency for MC Simulation

More efficient when we generate random number together:

Example

```
> MC2 <- function(){# initialize Z as vectors and use replicate
+   m <- 10000
+   one.path <- function(){
+     S <- rep(0, n+1)
+     S[1] <- S0
+     Z <- rnorm(n)
+     for(i in 1:n){
+       S[i+1] <- S[i] + r*S[i]*h + sigma*S[i]*Z[i]*sqrt(h)
+     }
+     return(S)
+   }
+   S.mat <- replicate(m, one.path())
+   print(exp(-r*T1)*mean(pmax(S.mat[n+1,] - 100, 0)))
+ }
> system.time(MC2())
[1] 10.26749
   user  system elapsed
   1.64    0.03    1.69
```

Improve Efficiency for MC Simulation

More efficient if we update the target matrix directly

Example

```
> MC3 <- function(){# initialize S and Z as matrices and update matrix
+   m <- 10000
+   S.mat <- matrix(0, nrow = n+1, ncol = m)
+   Z <- matrix(rnorm(n*m), nrow = n)
+   S.mat[1,] <- S0
+   for (i in 1:n) {
+     S.mat[i+1,] <- S.mat[i,] + r*S.mat[i,]*h + sigma*S.mat[i,]*Z[i,]*sqrt(h)
+   }
+   print(exp(-r*T1)*mean(pmax(S.mat[n+1,] - 100, 0)))
+ }
> system.time(MC3())
[1] 10.55469
   user  system elapsed
   0.38    0.02    0.40
```

Improve Efficiency for MC Simulation

Increase the number of paths when we think it is efficient enough.

Example

```
> MC3 <- function(m){
+   #m <- 10000
+   S.mat <- matrix(0, nrow = n+1, ncol = m)
+   Z <- matrix(rnorm(n*m), nrow = n)
+   S.mat[1,] <- S0
+   for (i in 1:n) {
+     S.mat[i+1,] <- S.mat[i,] + r*S.mat[i,]*h + sigma*S.mat[i,]*Z[i,]*sqrt(h)
+   }
+   print(exp(-r*T1)*mean(pmax(S.mat[n+1,] - 100, 0)))
+ }
> system.time(MC3(1000000))
[1] 10.42647
   user  system elapsed
  77.53   30.16  130.55
```

Improve Efficiency for MC Simulation

- The option payoff $(S(T) - K)_+$ only depends on the final price $S(T)$. i.e. The European option is not path-dependent
- We can only keep and update the vector of prices at the current step, since we don't need to store the history of the stock prices.

Example

```
> MC4 <- function(m){
+   #m <- 10000
+   S.vec <- rep(S0, m)
+   Z <- matrix(rnorm(n*m), nrow = n)
+   for (i in 1:n) {
+     S.vec <- S.vec + r*S.vec*h + sigma*S.vec*Z[i,]*sqrt(h)
+   }
+   print(exp(-r*T1)*mean(pmax(S.vec - 100, 0)))
+ }
> system.time(MC4(1000000))
[1] 10.43147
   user  system elapsed
  29.25    2.56   31.90
```

Improve Efficiency of MC Simulation

To improve efficiency, we can

- use "apply" functions to avoid changing the size of vectors or matrices
- generate the random numbers together instead one at each time
- update the target vectors/matrices directly