Monte Carlo Methods

Ross Bennett

March 27, 2014

Abstract

The purpose of this vignette is to demonstrate monte carlo methods as outlined in Chapter 9 of Foundations of Quantitative Analysis.

Contents

1 Monte Carlo

Monte Carlo methods are simulation techniques uses in valuing derivatives and in measuring risk. The Monte Carlo method we will focus on is the simple case with one random variable. The stochastic model we will use to model the price of an asset is the Geometric Brownian Motion (GBM) model. The model assumes that small changes in price are described by

$$dS = \mu S dt + \sigma S dz \tag{1}$$

where

μ represents the instantaneous drift rate

σ represents the instantaneous volatility rate

S represents the asset price

dz is a normally distributed random variable with mean 0 and variance dt

In order to simulate the price path followed by S, we can discretize the process by dividing the overall time of the asset into N intervals of length δt to get

$$dS = S_{t-1}(\mu dt + \sigma \epsilon \sqrt{dt}) \tag{2}$$

where ϵ is a standard normal random variable, $\epsilon \sim N(0, 1)$.

The simulated price for S_1 is computed as

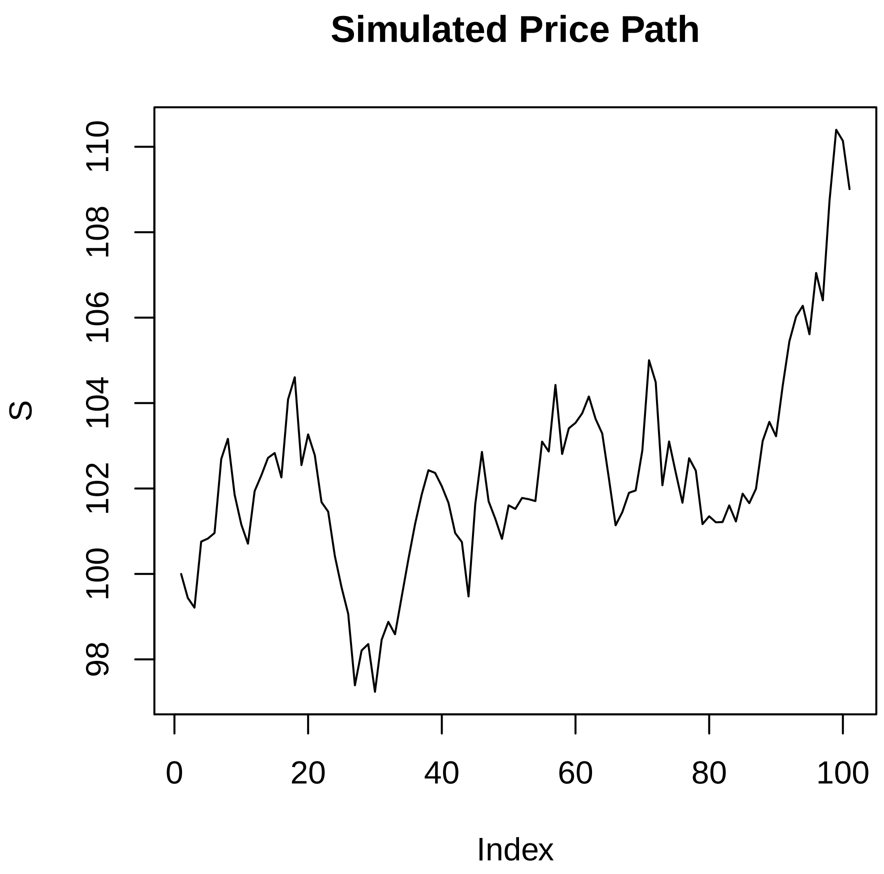
$$S_1 = S_0 + S_0(\mu \Delta t + \sigma \epsilon \sqrt{dt}) \tag{3}$$

The general equation to simulate the price path for S is

$$S_{t+1} = S_{t-1} + S_{t-1}(\mu \Delta t + \sigma \epsilon \sqrt{\Delta t}) \tag{4}$$

We can easy simulate this in `R`.

```r
suppressPackageStartupMessages(library(GARPFRM))
# drift rate
mu <- 0

# volatility rate
sigma <- 0.1

# starting price
S0 <- 100

# number of steps
N <- 100

dt <- 1 / N

# Generate N standard normal random variables
set.seed(123)
eps <- rnorm(N)

# Allocate a vector to hold the prices
S <- vector("numeric", N+1)
S[1] <- S0

# Precompute some of the terms
mu_dt <- mu * dt
sig_dt <- sigma * sqrt(dt)

for(i in 2:length(S)){
  S[i] <- S[i-1] + S[i-1] * (mu_dt + sig_dt * eps[i-1])
}
head(S)

## [1] 100.00  99.44  99.21 100.76 100.83 100.96

plot(S, main="Simulated Price Path", type="l")
```



In practice, simulating $\ln S$ instead of S gives more accuracy. By applying Ito's lemma, the process followed by $\ln S$ is

$$d \ln S = \left(\mu - \frac{\sigma^2}{2} \right) dt + \sigma dz \tag{5}$$

Equivantly we can write this as the discretized version used for simulation purposes.

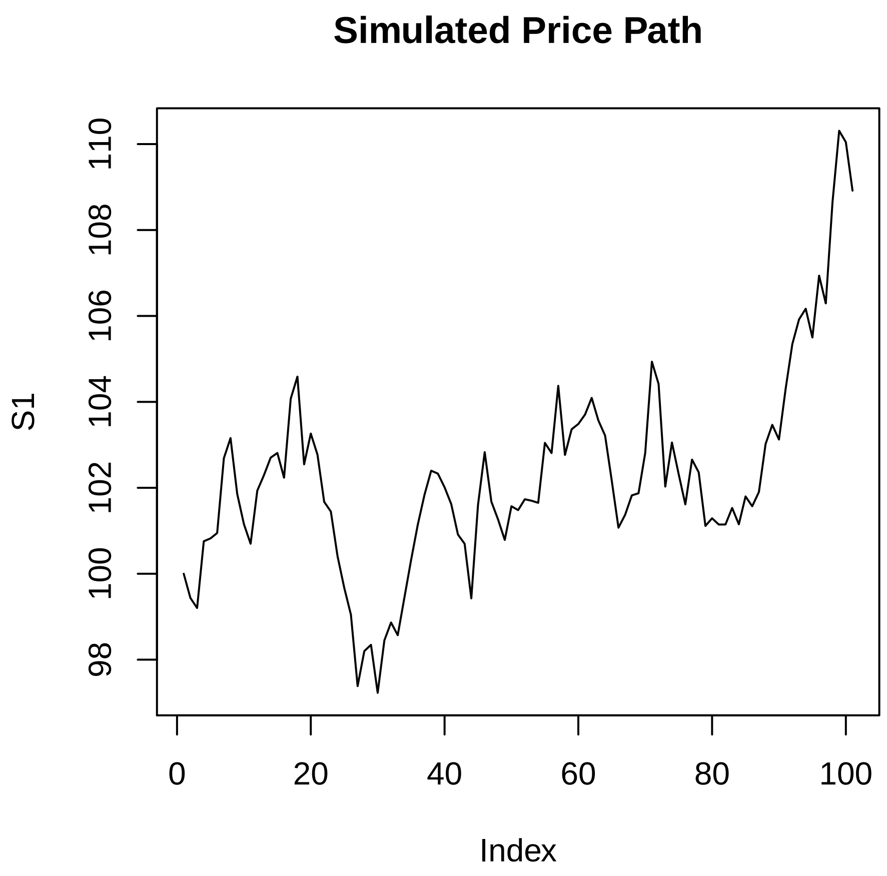
$$S_{t+1} = S_t exp \left[\left(\mu - \frac{\sigma^2}{2} \right) \Delta t + \sigma \epsilon \sqrt{\Delta t} \right] \tag{6}$$

Two key assumptions with this model are that S_T is lognormally distributed and the percentage changes of S are normally distributed.

We can easily simulate this in `R` using the variables we previously defined.

```r
# Allocate a vector to hold the prices
S1 <- vector("numeric", N+1)
S1[1] <- S0

# Precompute terms
```

```r
mu_sig_dt <- (mu - 0.5 * sigma^2) * dt
sig_dt <- sigma * sqrt(dt)

for(i in 2:length(S1)){
  S1[i] <- S1[i-1] * exp(mu_sig_dt + sig_dt * eps[i-1])
}
head(S1)

## [1] 100.00  99.44  99.20 100.76 100.82 100.95

plot(S1, main="Simulated Price Path", type="l")
```



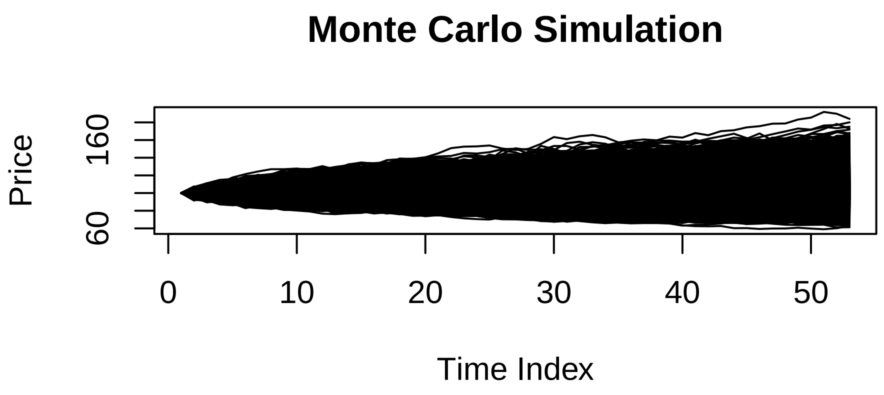
The above R examples simulated only one price path. To carry out the Monte Carlo simulation to value derivatives or manager risk, the process above is carried out K times to simulate K price paths. Here we simulate 10,000 price paths of an asset with a time horizon of 1 year and 52 time steps.

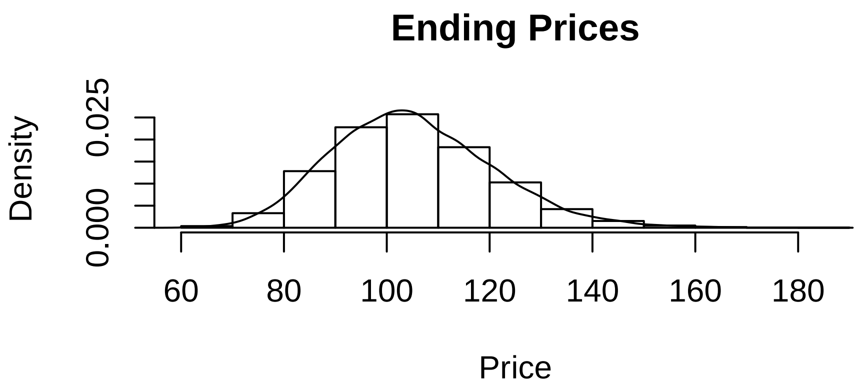
```
mu <- 0.05
sigma <- 0.15
N <- 10000
time <- 1
steps <- 52
startingValue <- 100

# Run Monte Carlo simulation and store simulated price paths
mcSim <- monteCarlo(mu, sigma, N, time, steps, startingValue)
summary(endingPrices(mcSim))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    61.3    94.0   104.0   105.0   115.0   184.0
```

Plot the simulated price paths and distribution of ending prices.

```
par(mfrow=c(2,1))
plot(mcSim)
plotEndingPrices(mcSim)
```





```r
par(mfrow=c(1,1))
```

The Monte Carlo simulation method is useful for valuing options that are path dependent such as lookback or asian options, do not have an analytical solution, or have a complex payoff. Valuing options and variance reduction techniques using Monte Carlo simulation are beyond the scope of this vignette.

2 Bootstrap

An alternative simulation method to generating random numbers from a model with assumptions of the distribution is to sample directly from historical data. Bootstrapping is a statistical method for estimating the sampling distribution of an estimator by sampling with replacement from the original sample. One key assumption is that returns are independent and identically distributed. Note that by random resampling, we break any pattern of time variation in returns. Another drawback is that resampling requires large sample sizes (a small sample size may lead to a poor approximation of the actual distribution) and is relatively computationally intensive.

A major advantage of this approach is that we do not need to make any assumption about the distribution of the data. The bootstrap will capture any departure from the normal distribution or if the data is skewed, has fat tails, or jumps.

We can use the bootstrap method to project prices, returns, or calculate statistics such as standard deviation or Value-at-Risk.

As an example, suppose the ending price of MSFT is $25 and we want to project the prices 5 periods into the future using the bootstrap method.

```
data(crsp_weekly)
R.MSFT <- largecap_weekly[, "MSFT"]

# Project number of periods ahead
nAhead <- 5

# Previous price
S.p <- 25

# Using a for loop
bootS <- vector("numeric", h)

## Error:  object 'h' not found

for(i in 1:nAhead){
  bootS[i] <- S.p * (1 + sample(R.MSFT, 1, TRUE))
  S.p <- bootS[i]
}

## Error:  object 'bootS' not found

bootS

## Error:  object 'bootS' not found


# Vectorized solution
S.p <- 25
bootS1 <- S.p * cumprod(1 + sample(coredata(R.MSFT), nAhead, TRUE))
bootS1

## [1] 24.67 22.83 23.31 20.29 20.27
```

We can also use the bootstrap method to compute various statistics such as Value-at-Risk. Here is an example of how to calculate historical Value-at-Risk with bootstrapped returns.

```
# Number of boostrap replications
rep <- 10000

# Allocate vector to hold VaR statistic
out <- vector("numeric", rep)
for(i in 1:rep){
  out[i] <- VaR(R.MSFT[sample.int(nrow(R.MSFT), replace=TRUE)],
                 method="historical")
}

# Bootstrapped VaR
mean(out)

## [1] -0.06727


# Standard error of Bootstrapped VaR
sd(out)

## [1] 0.006031
```

The GARPFRM package (Bennett et al., 2013) provides several functions for bootstrapped statistics as well as a bootFUN function that will calculate a bootstrapped statistic of any valid R function.

```
R <- largecap_weekly[,1:4]

# function to calculate the annualized StdDev using the most recent n periods
foo <- function(R, n){
  StdDev.annualized(tail(R, n), geometric=TRUE)
}

bootFUN(R[,1], FUN="foo", n=104, replications=1000)

##         [,1]
## foo       NA
## std.err   NA


# Bootstrap mean estimate.
bootMean(R)

##             ORCL     MSFT      HON      EMC
## mean    0.004933 0.002764 0.002280 0.004885
## std.err 0.002364 0.001644 0.001861 0.002653


# Bootstrap standard deviation estimate.
bootSD(R)

##             ORCL     MSFT      HON     EMC
## sd      0.066910 0.045874 0.050944 0.07053
## std.err 0.003095 0.001801 0.003941 0.00295


# Bootstrap standard deviation estimate using the StdDev function from
# PerformanceAnalytics.
bootStdDev(R)
```

```
##              ORCL     MSFT      HON      EMC
## StdDev  0.067108 0.045980 0.050830 0.070621
## std.err 0.003192 0.001849 0.004048 0.002816


# Bootstrap simpleVolatility estimate.
bootSimpleVolatility(R)

##                       ORCL     MSFT      HON      EMC
## SimpleVolatility 0.067091 0.045976 0.050856 0.070760
## std.err          0.002969 0.001807 0.004012 0.002878


# Bootstrap correlation estimate.
bootCor(R)

##         ORCL.MSFT ORCL.HON ORCL.EMC MSFT.HON MSFT.EMC HON.EMC
## cor       0.44930  0.31469  0.56859  0.37178  0.49528 0.34408
## std.err   0.04712  0.03719  0.03034  0.05574  0.03998 0.04672


# Bootstrap covariance estimate.
bootCov(R)

##         ORCL.MSFT  ORCL.HON  ORCL.EMC  MSFT.HON  MSFT.EMC   HON.EMC
## cov     0.0013847 0.0010656 0.0026838 0.0008654 0.0016071 0.0012206
## std.err 0.0001824 0.0001541 0.0002787 0.0001918 0.0002015 0.0001668


# Bootstrap Value-at-Risk (VaR) estimate using the VaR function from
# PerformanceAnalytics.
bootVaR(R, p=0.9, method="historical", invert=FALSE)

##              ORCL     MSFT      HON      EMC
## VaR     0.062671 0.046334 0.053183 0.073095
## std.err 0.003897 0.002572 0.004913 0.006284

bootVaR(R, p=0.9, method="gaussian", invert=FALSE)

##              ORCL     MSFT     HON      EMC
## VaR     0.080948 0.056365 0.06266 0.085324
## std.err 0.004139 0.002802 0.00505 0.004773


# Bootstrap Expected Shortfall (ES) estimate using the ES function from
# PerformanceAnalytics. Also known as Conditional Value-at-Risk (CVaR) and
# Expected Tail Loss (ETL).
bootES(R, p=0.9, method="historical")

##               ORCL     MSFT      HON       EMC
## ES      -0.111713 -0.07732 -0.08840 -0.125738
## std.err  0.007269  0.00518  0.00616  0.008387
```

```
bootES(R, p=0.9, method="gaussian")

##             ORCL      MSFT       HON       EMC
## ES      -0.1124 -0.077632 -0.086333 -0.119102
## std.err  0.0057  0.003587  0.006439  0.005655
```

To improve speed and performance, we can run the bootstrap in parallel. We leverage the `foreach` package (Analytics and Weston, 2013) to perform the computations in parallel.

```
# Register multicore parallel backend with 3 cores
# Note that this example does not work on Windows
# Windows users should use doSNOW
library(doMC)

## Loading required package:  foreach
## Loading required package:  iterators
## Loading required package:  parallel

registerDoMC(3)

# Estimate VaR via bootstrap
bootVaR(R[,1], p=0.9, method="historical", replications=10000, parallel=TRUE)

##               ORCL
## VaR     -0.062739
## std.err  0.004195



# Benchmark the performance of running the bootstrap in parallel
# Bootstrap VaR with parallel=TRUE
bootPar <- function(){
  bootVaR(R[,1], p=0.9, method="historical", replications=10000, parallel=TRUE)
}

# Bootstrap VaR with parallel=FALSE
bootSeq <- function(){
  bootVaR(R[,1], p=0.9, method="historical", replications=10000, parallel=FALSE)
}

# Benchmark these functions
library(rbenchmark)
benchmark(bootPar(), bootSeq(), replications=1)[,1:4]

##        test replications elapsed relative
## 1 bootPar()            1   90.28    4.124
## 2 bootSeq()            1   21.89    1.000
```

References

R. Analytics and S. Weston. *foreach: Foreach looping construct for R*, 2013. URL `http://CRAN.R-project.org/package=foreach`. R package version 1.4.1.

R. Bennett, T. Fillebeen, and G. Yollin. *GARPFRM: Global Association of Risk Professionals: Financial Risk Manager*, 2013. R package version 0.1.0.