CQF Exam 2

Boyan Davidov

April 14, 2020

1 Binary Option Pricing

1.1 Payoff Scheme

Binary[1] (cash-or-nothing) option pays some fixed amount of cash if the option expires in-the-money or nothing at all if out-of-the-money(OTM). For simplicity we can assume that it is European option, that pays 1$ if ITM (and 0$ otherwise). We assume also zero dividends. Using the expected value of the discounted payoff under the risk-neutral density Q, we have:

$$V(S,t) = e^{-r(T-t)} E^Q[Payoff(S_T)]$$

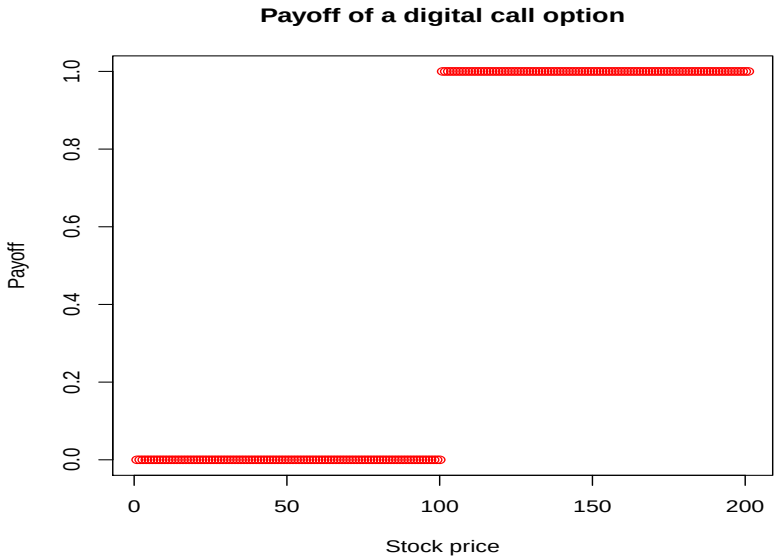
With the heavyside unit step function, we can write the payoff of a binary call option as:

$$Payoff_{call}(S_T) = \mathcal{H}(S_T - E) = \begin{cases} 1 & S_T > E \\ 0 & S_T \leq E \end{cases}$$

where S_T is the price of the underlying at time T, E is the execution price(strike). Respectively, a binary put option will have payoff of:

$$Payoff_{put}(S_T) = \mathcal{H}(E - S_T) = \begin{cases} 1 & S_T < E \\ 0 & S_T \geq E \end{cases}$$

The diagram below shows the value (in red points) of binary call option with strike of 100.



Another type of binary option is the asset-or-nothing binary option, which pays the asset at time S_T or nothing. Analogically to the cash-or-nothing type:

$$Payoff_{call}(S_T) = \mathcal{H}(S_T - E) = \begin{cases} S & S_T > E \\ 0 & S_T \leq E \end{cases}$$

[1] also called digital

1.2 Theoretical Price based on BS Model

Using the risk-neutral approach one can write the value of the option as:

$$e^{-r(T-t)} \int_{-\infty}^{+\infty} V(s,t) f\{s(t)\} ds_t$$

Since the option is priceless below the strike X, the integral can be written as:

$$e^{-r(T-t)} \int_{x}^{+\infty} f\{s(t)\} ds_t = e^{-r(T-t)} P(S_T > x)$$

The probability of ending ITM is simply $N(d_2)$. Thus the closed-form solution for a binary cash-or-nothing call in the Black-Scholes world is:

$$V(S,t) = e^{-r(T-t)} N(d_2)$$

where:

$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{\omega^2}{2}} dw$

and

$d_2 = \frac{log(S/E)+(r-D-0.5\sigma^2)(T-t)}{\sigma\sqrt{T-t}}$

Respectively, the value of a binary cash-or-nothing put option can be found as:

$$V(S,t) = e^{-r(T-t)} N(-d_2)$$

From the text we are given the following set of sample data:

- Today's stock price $S_0 = 100$,
- Strike Price E=100,
- Time to expiration T-t=1year,
- Annual Volatility $\sigma = 20\%$,
- risk-free rate r=5%.

```
> #Function to calculate the theoretical (BS) price
> # Type = 1 for call, Type=-1 for put
>
> #T - time to maturity in years
> #S0 - initial price
> #K - execution price or strike
> #sigma - annual volatility
>
> BS_digital<-function(S0, K, sigma, r, T, Type, show_price=TRUE)
+ {
+   d2 = (log(S0/K) + (r - 0.5 * sigma^2) * T)/ (sigma*sqrt(T))
+   Put_or_Call<-c()
+   ifelse(Type==1, Put_or_Call<-"Call", Put_or_Call<-"Put")
+   price<-exp(-r * T) * pnorm(Type*d2)
+   if(show_price==TRUE) {print(paste("The BS price of the digital ", paste(Put_or_Call), " is ",
+                                     round(price,4),sep=""))} else {
+                                     print(paste("The BS probability of ending ITM of the digital",
+                                                 round(pnorm(Type*d2),4),sep=""))
+                                     }
+  return(price)
+ }
```

The theoretical prices accroding to the Black-Scholes model for the initial data will be:

```
> BS_binary_call_theoretical_price<-BS_digital(100,100,0.2,0.05,1,1)

[1] "The BS price of the digital Call is 0.5323"

> BS_binary_put_theoretical_price<-BS_digital(100,100,0.2,0.05,1,-1)

[1] "The BS price of the digital Put is 0.4189"
```

Following the same logic, the Black-Scholes theoretical price of the asset-or-nothing is

$$V(S,t) = Se^{-r(T-t)}N(\phi_2)$$

with $\phi = 1$ for calls $\phi = -1$ for puts resp.

2 Monte Carlo Stock Price Simulation

2.1 Euler-Maruyama Method

We assume that the underlying price (non dividend paying stock) for the binary option follows the Geometric Brownian Motion (GBM):

$$dS_t = r_t S_t dt + \sigma_t S_t dW_t$$

where r_t and σ_t are the short-term rate (or risk-free rate) and the volatility at time t (in our case we assume constant through the whole path), and $W_t$is Wiener process.

The Forward Euler-Maruyama Method is a way to simulate the time series of the option's underlying S in discrete time. To simulate the price of an asset for GBM the method uses the following formula:

$$\delta S_t = S_{t+\delta t} - S_t \sim rS_t\delta t + \sigma S_t \phi\sqrt{\delta t}$$

where δt is the discrete time step and ϕ is a standard normal variable. Equivalently:

$$S_{t+\delta t} \sim S_t(1 + r\delta t + \sigma\phi\sqrt{\delta t})$$

The following pseudo code describes the simulation procedure:

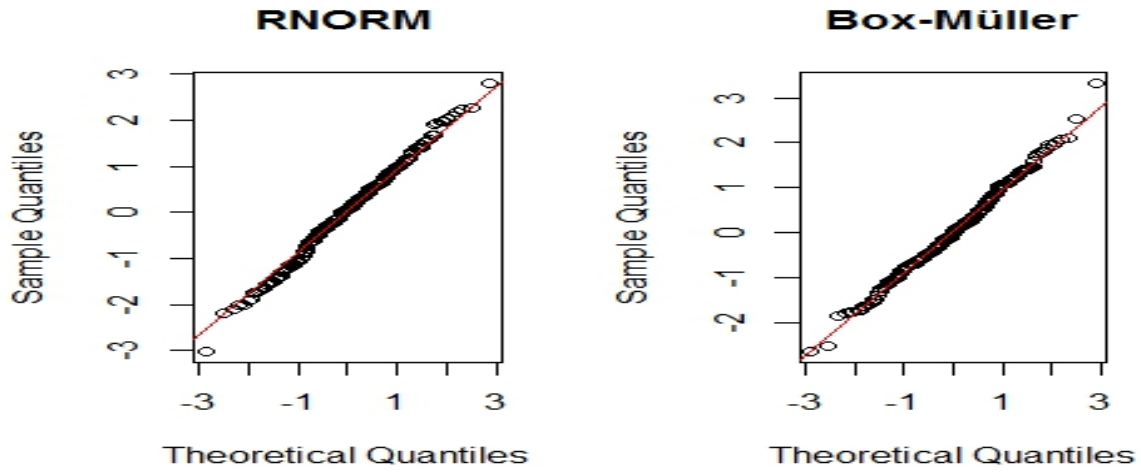
Algorithm 1 Euler-Maruyama Scheme

Description:

Procedure S_T Projection $(S_0, T-t, \sigma, r)$
$S(0,:) = S_0$
$\Delta t = 1/N$ //define dt
for i = 1 to M **do** //define number of paths
  for j = 2 to N+1 **do** //define number of time steps
    Generate ϕ
    $S(j,i) = S(j-1,i) * (1 + r * \Delta t + \sigma * \phi * \sqrt{\Delta t})$
return S

To generate the random numbers we utilize the R built-in "rnorm" function, based on the Mersenne-Twister method. Alternatively one can use the Box-Müller method and generate number on the principle of $\phi = \sqrt{-2lnU_1}cos(2\pi U_2)$, where U_1, U_2 are random uniform numbers. We can see from the plot below that both method produce good results, however the built-in function is from computing perspective better. [2]

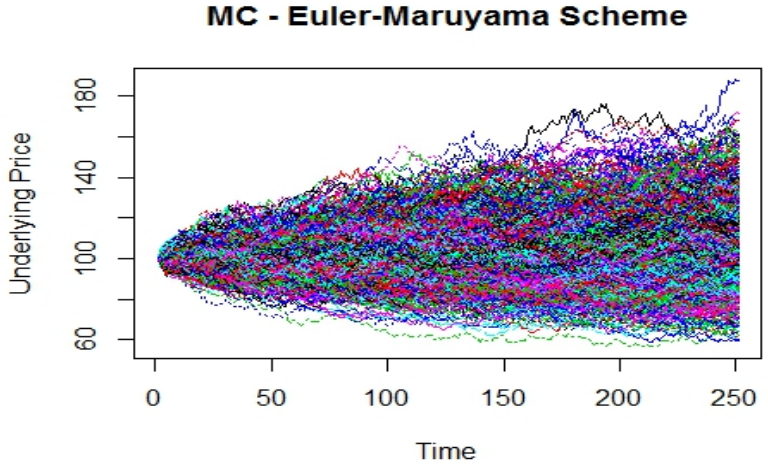
[2]Another approach is to introduce a second process S_2 and generate pairs of random numbers as ϕ and $-\phi$



The code chunk below is implementation of the Euler-Maruyama Scheme in R.

```
> # run simulation using Euler - Maruyama Scheme
>
> monte_carlo_sim_euler<-function(simulations, days, init_price,annual_vol, risk_free) {
+   price_matrix<-matrix(0,nrow=(days+1),ncol=simulations)
+   dt<-1/days
+   for (i in 1:simulations) {
+   price_matrix[1,i]<-init_price
+     for (j in 2:(days+1)) {
+       price_matrix[j,i]=price_matrix[j-1,i]*(1+risk_free*dt+annual_vol*sqrt(dt)*rnorm(1))
+     }
+   }
+   matplot(price_matrix, main="MC - Euler-Maruyama Scheme", ylab="Path", xlab="Time", type="l")
+   price_matrix
+ return(price_matrix)
+ }
```

The graph below shows 1000 simulated paths for a time step of $\delta t = 0.004$, which corresponds to 250 trading days in one year.



2.2 Averaging Method for Binary Options

After simulating n paths we can create a set with final prices of the underlying by extracting the last row of the price matrix created by the MC simulation. By storing the values higher than the initial price and the values lower than the initial price, we get two subsets which determine the call or put payoff. The value of the options will be simply the discounted mean of the respective set:

$$C(T) = e^{-r(T-t)} Average(Set_{call\_payoffs})$$

$$P(T) = e^{-r(T-t)} Average(Set_{put\_payoffs})$$

Further we will consider only the cash-or-nothing type of binary options since conceptually the procedure is analogical for the asset-or-nothing type[3], the only difference in the implementation of the asset-or-nothing case being that one has to store and sum the final underlying prices higher than strike, whereas in the case of cash-or-nothing option a simple counter is enough to indicate whether the option ends ITM or OTM.

To evaluate the payoff of the option the only price we need is the final one, thus it is redundant to store the whole path evolution of the underlying. In the loop the final price is compared with the strike price and a counter is increased by one if the option is ITM. The following code implements this logic:

```
> #The code has been refined to accomodate more simulated paths with greater speed
> #only the final price is relevant and only payoffs are stored, rather than the whole path
>
> binary_option_value<-function(simulations, days, init_price, strike, annual_vol, risk_free) {
+   call_payoffs<-0                    #counter of ITM-call options
+   put_payoffs<-0                     #counter of ITM-put options
+   call_value<-0
+   put_value<-0
+   dt<-1/days
+   price_vector<-c()
+   price_vector[1]<-init_price
+   for (i in 1:simulations) {           #define number of sim.paths
+     price_vector<-c()                  #the price vector is reset
+     price_vector[1]<-init_price
+     for (j in 2:(days+1)) {            #define number of time steps
+       price_vector[j]=price_vector[j-1]*(1+risk_free*dt+annual_vol*sqrt(dt)*rnorm(1))
+     }
+     if (price_vector[length(price_vector)]>strike) {
+       call_payoffs<-call_payoffs+1 } else { put_payoffs<-put_payoffs+1}
+   }
+   call_value<-exp(-risk_free)*(call_payoffs/simulations)
+   put_value<-exp(-risk_free)*(put_payoffs/simulations)
+ return(list("Put" = put_value, "Call" =call_value))
+ }
```

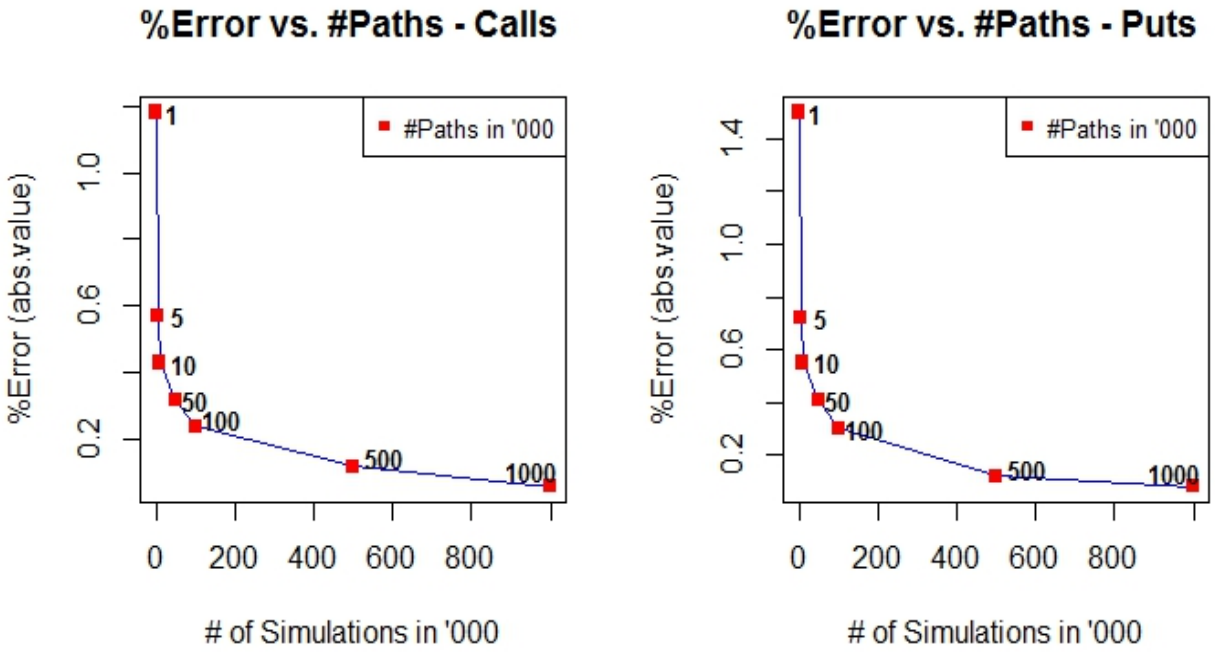
To reinforce the simulation, in case one is running millions of simulations, one could execute the procedure for smaller set of simulations, then repeat several times and average the results of the option values. For instance we run the simulation for 1000 paths, we obtain a value for the call or put by averaging the payoffs from all paths and then we repeat this procedure 10 times again taking average of the obtained 10 values. This would be memory more efficient than a single simulation of 10000 paths. In our case of binary option this is allowed. However, in other cases this method won't work for all functions. For instance if the value of the option was a convex function of the payoff (eg. e^{payoff}) then according to Yensen's inequality $f\left(\sum_1^n p_i x_i\right) \leq \sum_1^n p_i f(x_i)$, the method with repetitions would tend to give higher values.

[3] code for the asset-or-nothing case can be found in the appendix

2.3 Error Analysis and Observations on Binary Options

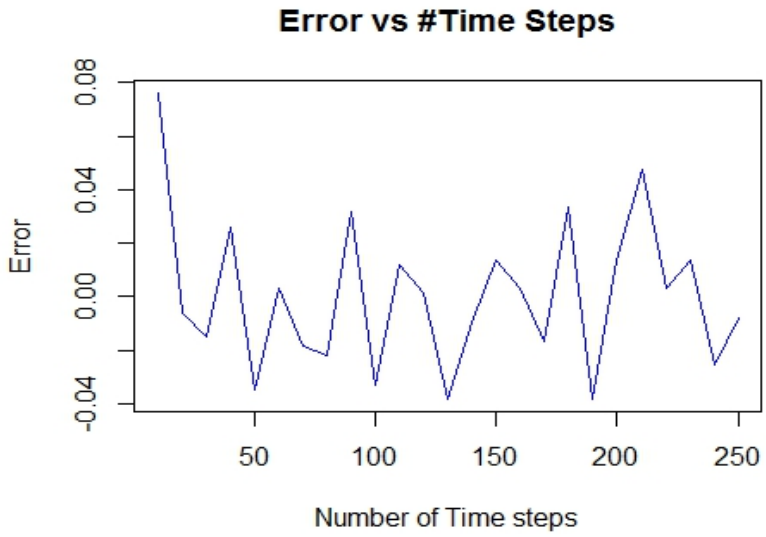
The comparison between the theoretical option price and simulation option price is presented in the table below.

Euler-Maruyama Method Numerical Results				
Type	Paths	Monte Carlo Price	Theo.Price	% Diff
Call	1000	0.5260	0.5323	-1.18%
	5000	0.5293	0.5323	-0.57%
	10000	0.5300	0.5323	-0.43%
	50000	0.5306	0.5323	-0.32%
	100000	0.5310	0.5323	-0.24%
	500000	0.5328	0.5323	-0.12%
	1000000	0.5326	0.5323	0.06%
Put	1000	0.4251	0.4189	1.50%
	5000	0.4219	0.4189	0.72%
	10000	0.4212	0.4189	0.55%
	50000	0.4206	0.4189	0.41%
	100000	0.4201	0.4189	0.30%
	500000	0.4184	0.4189	-0.12%
	1000000	0.4186	0.4189	-0.08%



In general, for both call option and put options, simulated price gets closer to the theoretical value when the number of simulations is increased. The declines exponentially up to 50-100 thousands of simulated paths and then linearly with further increasing of simulated paths. It could be noted also that the method was generally more accurate for the call option price simulations, in comparison to the put simulations. Another observation is that call values tend to be underpriced, whereas put values to be overpriced but the tendency inverts at around 500000 simulations. This could be of course due to the built-in random numbers generator (eg. higher probability of extreme negative numbers). Finally, it is worth mentioning that the observed run time of the Monte Carlo chunk of code (or CPU system time) increased only linearly with the number of simulated paths.

We have analyzed also if the number of steps for the generation of the path of the underlying has some influence. The diagram below shows the relative error for binary call value when increasing the time steps and simulating 10000 paths.



There seems to be initially an correction in the error rate when increasing number of time steps, which then transitions to inconsistent error decay as the number of price steps continues to increase. One can conclude that that the model will probably not become more accurate if we use more than the 250 time steps which we were using in the simulations.

3 Extensions

3.1 Milstein Method

The Milstein method adds to the Euler-Maruyama an additional second order term by expanding the Taylor series:

$$e^{(r-\frac{1}{2}\sigma^2)\delta t+\sigma\phi\sqrt{\delta t}} \sim 1 + (r - \frac{1}{2}\sigma^2)\delta t + \sigma\phi\sqrt{\delta t} + \frac{1}{2}\sigma^2\phi^2\delta t$$

or equivalently:

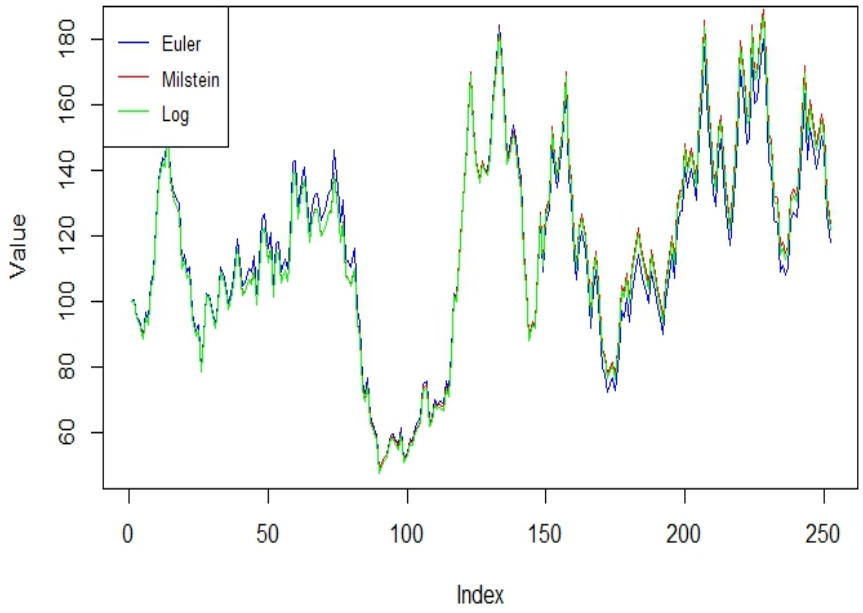
$$S_{t+\delta t} \sim S_t(1 + r\delta t + \sigma\phi\sqrt{\delta t} + \frac{1}{2}\sigma^2(\phi^2 - 1)\delta t + ...)$$

The formula differs by the term $\frac{1}{2}\sigma^2(\phi^2-1)\delta t$, the so-called Milstein correction. If one juxtaposes the Euler, Milstein and log models[4], as we have done on Figure 1, it would be hard to distinguish difference especially for small time-steps.

However, while the Euler-Maruyama method has an error to extent of $O(\delta t)$, the Milstein method reduces the error to $O(\delta t^2)$.

[4]code in appendix

Figure 1: Simulated path with Euler,Milstein and GBM.



One can seen from the table below, that the error indeed reduces faster than the Euler-Maruyama method:

Milstein Method Numerical Results				
Type	Paths	Monte Carlo Price	Theo.Price	% Diff
Call	5000	0.5259	0.5323	-1.21%
	10000	0.5302	0.5323	-0.39%
	50000	0.5310	0.5323	0.02%
Put	5000	0.4254	0.4189	1.55%
	10000	0.4210	0.4189	0.50%
	50000	0.4187	0.4189	-0.03%

3.2 Stochastic Volatility

So far we have assumed volatility is constant through the whole path of the underlying. A more realistic approach would be to set the volatility to vary with the price. The Heston model is one way to implement this. The underlying path is defined as:

$$dS_t = \mu S_t dt + \sqrt{v_t} S_t dW_t^1$$

whereas the volatility is computed as:

$$dv_t = \kappa(\theta - v_t)dt + \xi\sqrt{v_t} dW_t^2$$

As it can be seen the underlying path and the volatility's dynamics depend on different Brownian motions W_t^1 and W_t^2 resp., with $dW_t^1 dW_t^2 = \rho * dt$. This can be ensured via Cholesky Decomposition $W^2 = \rho W^1 + \sqrt{1-\rho^2} W^2$.

The following parameters define the model:

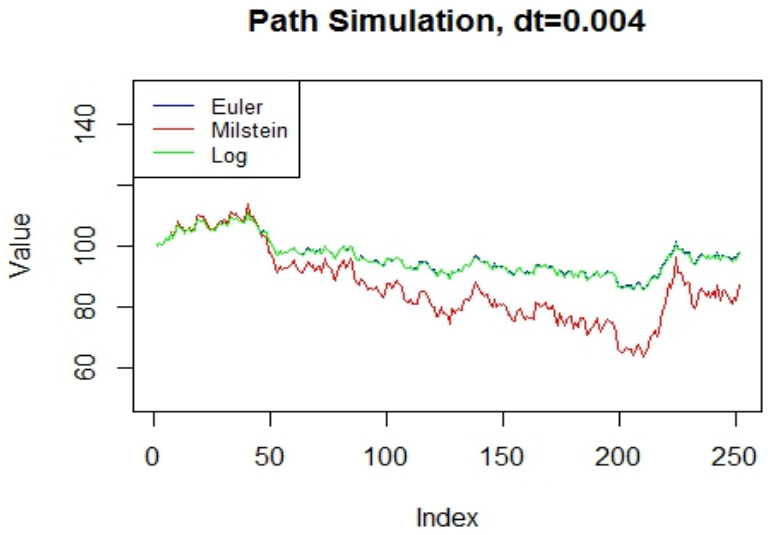
- ρ - correlation between the W_t^1 and W_t^2
- κ - mean reversion rate
- θ - long-term volatility
- ξ - volatility of volatility

The variance of the underlying's variance is then updated as:

$$v_i = v_{i-1} + \kappa(\theta - f(v_{i-1})dt + \xi\sqrt{f(v_{i-1})dt}W_2 + \frac{1}{4}\xi^2(W_2 * W_2 - 1)dt$$

where $f(x) = max(0, x)$. The chart below shows paths simulated with the Euler-Maruyama method, simple GBM (log) and the Milstein method but with Heston stochastic volatility:(assuming $\rho = -0.5, \kappa = 1.05, \theta = 0.044, \xi = 0.20$)

Figure 2: Refined Milstein with Stochastic Vola.



As it can be seen the volatility can magnify the underlying movements before reverting to predefined long-term level. This looks definitely more realistic from finance perspective. The pseudocode in Algorithm 2 describes the code implementation[5]

Finally, below we present the results on the error analysis:

Milstein Method with Stochastic Volatility Numerical Results				
Type	Paths	Monte Carlo Price	Theo.Price	% Diff
Call	1000	0.5372	0.5323	0.92%
	5000	0.5336	0.5323	0.247%
	10000	0.5334	0.5323	0.21%
Put	1000	0.4177	0.4189	-1.17%
	5000	0.4210	0.4189	-0.31%
	10000	0.4187	0.4189	-0.26%

[5]code in appendix

Algorithm 2 Milstein Method with Stochastic Volatility

Description:

Procedure S_T Projection $(S_0, T-t, \sigma, r)$
$S(0,:) = S_0$
$\Delta t = 1/N$ //define dt
for i = 1 to M **do** //define number of paths
  for j = 2 to N+1 **do** //define number of time steps
    Generate W_1 and W_2
    $v(j,i) = v(j-1,i) + \kappa(\theta - f(v(j-1,i))dt + \xi\sqrt{f(v(j-1,i))dt}W_2 + \frac{1}{4}\xi^2(W_2 * W_2 - 1)dt$
    $S(j,i) = S(j-1,i) * (1 + r * \Delta t + v(j,i) * \phi * \sqrt{\Delta t})$
return S

The results up to 10000 simulated paths resemble those of the Milstein Method and Euler-Maruyama, however, increasing the number of simulated paths didn't make the error converge. In fact, on some occasions with tests of ¿10000 simulated paths the error increased rather than decreased. This doesn't mean per se the model is not good. Though, a more sophisticated model won't give more accurate results if it is not well calibrated. The set of parameters was arbitrary, based on trial and error with the only constraint of the so-called Feller condition $2k\theta > \xi^2$ (it is known that the model doesn't work well if this condition is not satisfied). In fact, just by choosing higher (or lower) long-term volatility (θ) than the initial of 20%, divergence from the Black-Scholes value will be expected.

4 Conclusion

In this assignment, the Euler-Maruyama method for Monte Carlo simulations has been examined with respect to pricing of European style binary options. In addition, extensions of the model considering the Milstein correction and using stochastic volatility (Heston) have been also explored.

Based on the simulations, the following observations could be made:

1. As the increase of the simulation times, both the numerical simulated price of put and call converges to the BS-Model theoretical price.
2. The simulated price of call options is more accurate than the simulated price of put options. It may be caused by the computer pseudo number generating regimes.
3. The Milstein correction adds more precision to the Euler-Maruyama method and the model error converges faster.
4. Stochastic volatility could provide more market associated and consistent stock price dynamics, however the model needs to be well calibrated.

The Euler-Maruyama error becomes relatively insensitive to increasing simulations around 50000-100000 simulations, whereas the Milstein method achieves best accuracy between 10000-50000 simulations. Introducing a process for the underlying variance has not improved the accuracy but can make the model more flexible and realistic.

5 References

CQF Module 3, Lectures 1,4,5
Peter Jäckel - Monte Carlo Mathods in Finance
https://en.wikipedia.org/wiki/Heston_model
https://www.quantstart.com/articles/Heston-Stochastic-Volatility-Model

6 Appendix

```
> ## This is to compare the simulated underlying path
> ## USING MILSTEIN, EULER AND THE REAL GBM
> ## Random numbers generated using the Box-Muller Method
>
> mu<-0.05
> sigma<-0.2
> dt<-0.01
> U1<-runif(251)
> U2<-runif(251)
> N_0_1<-sqrt(-2*log(U1))*cos(2*pi*U2)
> W<-c()
> W<-N_0_1*sqrt(dt)
> Euler<-Milstein<-Log<-c()
> Euler[1]<-Milstein[1]<-Log[1]<-100
> for ( i in 2:length(W))
+ {
+   Euler[i]<-Euler[i-1]+mu*Euler[i-1]*dt+sigma*Euler[i-1]*W[i-1]
+   Milstein[i]<-Milstein[i-1]+mu*Milstein[i-1]*dt+sigma*Milstein[i-1]*W[i-1]+
+     0.5*(W[i-1]^2-dt)*Milstein[i-1]*sigma^2
+   Log[i]<-Log[i-1]*exp((mu-0.5*sigma^2)*dt+sigma*W[i-1])
+ }
> R<-cbind(Euler,Milstein,Log)
> colnames(R)<-c('Euler','Milstein','Log')
> plot(Euler,col="blue",type="l",ylab="Value", main = "Path Simulation, dt=0.01")
> lines(Milstein,col="red")
> lines(Log,col="green")
> legend("topleft",c('Euler','Milstein','Log'),
+        col=c("blue","red","green"),lty=c(1,1,1),cex=0.8);
>
```

Code for calculating value of option with the Milstein method

```
> ####################################
> # Using Milstein method
> ####################################
>
> binary_option_value_Milstein<-function(simulations, days, init_price, strike, sigma, mu) {
+   call_payoffs<-0
+   put_payoffs<-0
+   call_value<-0
+   put_value<-0
+   dt<-1/days
+   Milstein<-c()
+   Milstein[1]<-init_price
+   for (i in 1:simulations) {
+     Milstein<-c()
+     Milstein[1]<-strike
+     for (j in 2:(days+1)) {
+       Milstein[j]<-Milstein[j-1]*(1+mu*dt+sigma*rnorm(1)*sqrt(dt)+
```

```
+         0.5*(rnorm(1)^2-1)*dt*sigma^2)}
+     if (Milstein[length(Milstein)]>strike) {
+       call_payoffs<-call_payoffs+1 } else { put_payoffs<-put_payoffs+1}
+   }
+   call_value<-exp(-mu)*(call_payoffs/simulations)
+   put_value<-exp(-mu)*(put_payoffs/simulations)
+   return(list("Put" = put_value, "Call" =call_value))
+ }
```

This is the Milstein method using stochastic volatility:

```
> binary_option_value_Milstein_Heston<-function(simulations, days, init_price, strike,
+                                     sigma, mu, kappa=1.05,theta=0.0225,xi=0.15,rho=-0.5) {
+   call_payoffs<-0
+   put_payoffs<-0
+   call_value<-0
+   put_value<-0
+   dt<-1/days
+   annual_var<-c()
+ annual_var[1]<-sigma^2
+   for (i in 1:simulations) {
+     Milstein<-c()                          #free memory
+     Milstein[1]<-init_price
+     annual_var<-c()
+     annual_var[1]<-sigma^2
+       for (j in 2:(days+1)) {
+       W1 <- rnorm(1);                     #generate random numbers
+       W2 <- rnorm(1);
+       W2 <- rho*W1 + sqrt(1 - rho^2)*W2;   #generate correlated brownian motions
+       annual_var[j] = annual_var[j-1] + kappa*(theta - max(annual_var[j-1],0))*dt + xi*
+       W2*sqrt(max(annual_var[j-1],0)*dt)+0.25*xi*xi*(W2*W2 - 1)*dt
+       Milstein[j]<-Milstein[j-1]*(1+mu*dt+(sqrt(annual_var[j]))*W1*sqrt(dt)+
+                                     0.5*(W1^2-1)*dt*annual_var[j])}
+     if (Milstein[length(Milstein)]>strike) {
+       call_payoffs<-call_payoffs+1 } else { put_payoffs<-put_payoffs+1}
+   }
+   call_value<-exp(-mu)*(call_payoffs/simulations)
+   put_value<-exp(-mu)*(put_payoffs/simulations)
+   return(list("Put" = put_value, "Call" =call_value))
+ }
```

```
> ##########################################################
> # Binary Asset-or-Nothing option value
> ##########################################################
>
> binary_option_value_Asset<-function(simulations, days, init_price, strike, annual_vol, risk_free)
+   call_payoffs<-0
+   put_payoffs<-0
+   call_value<-0
+   put_value<-0
+   value<-0
+   dt<-1/days
+   price_vector<-c()
+   price_vector[1]<-init_price
+   for (i in 1:simulations) {
+     price_vector<-c()
+     price_vector[1]<-init_price
```

```
+     for (j in 2:(days+1)) {
+       price_vector[j]=price_vector[j-1]*(1+risk_free*dt+annual_vol*sqrt(dt)*rnorm(1))
+     }
+     if (price_vector[length(price_vector)]>strike) {
+       call_payoffs<-c(call_payoffs,price_vector[j])} else
+       { put_payoffs<-c(put_payoffs,price_vector[j])}
+   }
+   call_value<-exp(-risk_free)*(sum(call_payoffs)/simulations)
+   put_value<-exp(-risk_free)*(sum(put_payoffs)/simulations)
+   return(list("Put" = put_value, "Call" =call_value))
+ }
```