Lab6_sol

October 9, 2017

1 Sample solutions of exercises of lab 6

1.1 Exercise 1

If we assume that the Black-Scholes market model with constant volatility holds, then we have to generate $S(T)$ according to the stochastic differential equation

$$dS(t) = S(t)((r - D)\,dt + \sigma\,dB(t).$$

From the lecture we know that the solution of the equation is

$$S(t) = S(0)e^{(r-D-\frac{\sigma^2}{2})t+\sigma B(t)},$$

so $S(T) = S(0)e^{(r-D-\frac{\sigma^2}{2})T+\sigma X}$, where $X \sim N(0, \sqrt{T})$. Write a function MC1, that for given values of $S(0), r, D, \sigma, T, \alpha$ and n and for given payoff function p computes an approximate option value and its error estimate holding with the probability $(1 - \alpha)$ by Monte-Carlo method, using n generated stock prices. Verify the correctness of the function by Black-Scholes formulas for put and call options in the case $S(0) = 100$, $E = 100$, $\sigma = 0.6$, $T = 0.5$, $r = 0.02$, $D = 0.03$, $\alpha = 0.05$ and $n = 10000$. How often the actual error is larger than the error estimate if you use MC1 1000 times?

Solution

```
In [1]: import numpy as np
        from scipy import stats
        phi_inv=stats.norm.ppf
        def MC1(S0,r,D,sigma,T,alpha,n,p):
            ST=S0*np.exp((r-D-sigma**2/2)*T+sigma*np.sqrt(T)*np.random.randn(n))
            Y=np.exp(-r*T)*p(ST)
            price=np.mean(Y)
            error=-phi_inv(alpha/2)*np.std(Y)/np.sqrt(n)
            return [price,error]
```

To verify the correctness of the code, we need the function that computes the price of call options according to Black-Scholes formula. If this function is in a file, it can be imported from it.

```
In [2]: import sys
        #indicate the location of the file
        sys.path.append("h:/compfin_labs")
        import BSformulas as bs
```

Verify the correctness

```
In [3]: def p_call(s,E=100):
            return np.maximum(s-E,0)
        exact_price=bs.Call(S=100,E=100,sigma=0.6,T=0.5,r=0.02,D=0.03)
        MC_price=MC1(S0=100,sigma=0.6,T=0.5,r=0.02,D=0.03,alpha=0.05,n=10000,p=p_call)
        print("Exact price:", exact_price)
        print("MC price with error estimate:",MC_price)
```

```
Exact price: 16.3452012899
MC price with error estimate: [16.099992086938226, 0.6031542557768631]
```

So the function `MC1` works correctly - the difference of the approximate price (16.099...) and the exact price (16.34...) is smaller than error estimate (as it should be with probability 0.95; the actual error may be larger than the estimate with probability 0.05, so such situation is possible, but not very likely). To verify further the correctness of the function, let us do 1000 computations and check, how often the error estimate is wrong. If the code is correct, the result should be wrong approximately 50 times. More precisely, the number of times the estimate is wrong should behave like random variable with binomial distribution Bin(1000,0.05), having mean 50 and standard deviation $\sqrt{1000 \cdot 0.05 \cdot 0.95} \approx 6.892$

```
In [4]: N=1000
        prices=np.zeros(N)
        errors=np.zeros(N)
        for i in range(0,N):
            result=MC1(S0=100,sigma=0.6,T=0.5,r=0.02,D=0.03,alpha=0.05,n=10000,p=p_call)
            prices[i]=result[0]
            errors[i]=result[1]
        print(np.sum(np.abs(prices-exact_price)>errors))
```

```
55
```

So it is very likely that the code is correct and that the error estimate computed for a given α has indeed the probabilistic meaning given by theory.

1.2 Exercise 2

Write a function MC2 that computes approximate option prices so that the stock prices are generated according to Euler's method. Determine how large is the difference between V_m (the expected value of $e^{-rT}p(S_m)$, where S_m is computed by Euler's method) and the correct option price in the case of European call option, using the same parameters as in the previous exercise for $m = 2, 4, 8, 16$. In order to see the difference, large enough value for n should be used (if possible, the corresponding MC error should be at least 5 times smaller than the computed difference).

Solution

```
In [5]: def MC2(S0,r,D,sigma,T,alpha,n,p,m):
            #use Euler's method to compute ST
```

```
        delta_t=T/m
        S=np.zeros(shape=(m+1,n))
        S[0,:]=S0
        for i in range(1,m+1):
            S[i,:]=S[i-1,:]*(1+(r-D)*delta_t+sigma*np.sqrt(delta_t)*np.random.randn(n))
        ST=S[m,:]
        Y=np.exp(-r*T)*p(ST)
        price=np.mean(Y)
        error=-phi_inv(alpha/2)*np.std(Y)/np.sqrt(n)
        return [price,error]
```

Remark: Since we need only the value of S_m, we do not need to fill the full matrix. It is enough if we have only the values
for the previous time moment available to compute the next one. So a more efficient version of the function `MC2` is the following:

```
In [6]: def MC2_better(S0,r,D,sigma,T,alpha,n,p,m):
            #use Euler's method to compute ST
            delta_t=T/m
            sqrt_delta_t=np.sqrt(delta_t)
            S=S0
            for i in range(1,m+1):
                S=S*(1+(r-D)*delta_t+sigma*np.sqrt_delta_t*np.randn.randn(n))
            #now S contains values for time T
            Y=np.exp(-r*T)*p(S)
            price=np.mean(Y)
            error=-phi_inv(alpha/2)*np.std(Y)/np.sqrt(n)
            return [price,error]
```

Let us verify the correctness of the function `MC2`.

```
In [7]: n=100000
        m=2
        result=MC2(S0=100,sigma=0.6,T=0.5,r=0.02,D=0.03,alpha=0.05,n=n,p=p_call,m=m)
        price_2=result[0]
        print(np.abs(price_2-exact_price))
        print(result[1])
```

```
0.408230460237
0.171244918001
```

The first component of the result of MC2 is $V_m + MC\_error$, so if we compute the difference of it and the exact price, we see the $V_m - V + MC\_error$. When the error of the MC method is relatively large compared to the difference we computed, we do not see clearly what we wanted to see (the actual difference $V_m - V$). So we have to use much larger n.

```
In [8]: n=10000000
        m=2
```

```
        result=MC2(S0=100,sigma=0.6,T=0.5,r=0.02,D=0.03,alpha=0.05,n=n,p=p_call,m=m)
        price_4=result[0]
        error_2=np.abs(price_4-exact_price)
        print(error_2)
        print(result[1])
```

```
0.41493153805
0.0171348488896
```

Now we see that the the actual difference of V_m and V is approximately 0.415 ± 0.017

```
In [9]: m=4
        result=MC2(S0=100,sigma=0.6,T=0.5,r=0.02,D=0.03,alpha=0.05,n=n,p=p_call,m=m)
        price_8=result[0]
        error_4=np.abs(price_8-exact_price)
        print("|V_m-V| for m={0} is {1}".format(m,error_4))
        print(result[1])
```

```
|V_m-V| for m=4 is 0.31479888321197436
0.0183285165554
```

```
In [10]: m=8
         result=MC2(S0=100,sigma=0.6,T=0.5,r=0.02,D=0.03,alpha=0.05,n=n,p=p_call,m=m)
         price_8=result[0]
         error_8=np.abs(price_8-exact_price)
         print("|V_m-V| for m={0} is {1}".format(m,error_8))
         print(result[1])
```

```
|V_m-V| for m=8 is 0.16048848147381278
0.0188936439358
```

```
In [11]: m=16
         result=MC2(S0=100,sigma=0.6,T=0.5,r=0.02,D=0.03,alpha=0.05,n=n,p=p_call,m=m)
         price_16=result[0]
         error_16=np.abs(price_16-exact_price)
         print("|V_m-V| for m={} is {}".format(m,error_16))
         print(result[1])
```

```
|V_m-V| for m=16 is 0.09053751013566469
0.0192276514134
```

```
In [12]: print([error_4/error_2,error_8/error_4,error_16/error_8])
```

```
[0.75867668360748497, 0.50981274087222717, 0.56413712251640857]
```

Theoretically, for large enough m we should have $\frac{error_{2m}}{error_m} \approx 0.5$. Here we see that $m = 2$ is clearly not large enough to see that behaviour, but starting from $m = 4$ this property holds.

1.3 Exercise 3

Find an approximate value of C (see lab handout) by fitting a linear regression line to the approximate option prices V_2, V_4, V_8, V_{16} computed in the previous exercise. Using the value of C, find a value of m such that $\frac{C}{m} \leq 0.03$.

Solution We have to take into account that in the command `stats.linregress` the fitted line has a form $y = a + b \cdot x$, where x values should be in the first parameter of the command and y values should be in the second parameter of the command. So the first parameter should be $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$. The value of constant C we want to estimate is the coefficient of x in the equation and this is also called slope of the regression line.

```
In [13]: prices=np.array([price_2,price_4,price_8,price_16])
         x=1/2**np.arange(1,5)
         model=stats.linregress(x,prices)
         print(model)
```

```
LinregressResult(slope=0.72841980942941631, intercept=16.443024894559244, rvalue=0.84073048461
```

We see, that the value of the parameter C is estimated to be approximately 0.7284. So we want to find the smallest value of m for which $\frac{0.7284}{m} \leq 0.03$

```
In [14]: m=np.int64(np.ceil(model[0]/0.03))
         print(m)
```

```
25
```

1.4 Exercise 4

With the value of m found in the previous exercise, compute V_m so that MC error is less than 0.02 with probability 0.9. Find also the actual difference of V_m you found and the exact option price. Is the actual difference smaller than 0.06?

Solution Let us do a computation with a reasonably large value of n:

```
In [15]: n=1000000
         answer_1=MC2(S0=100,sigma=0.6,T=0.5,r=0.02,D=0.03,alpha=0.05,n=n,p=p_call,m=m)
         print(answer_1)
```

```
[16.422725180088342, 0.061245754237993597]
```

We see, that the MC error is not small enough. If we look at how the MC error estimate depends on n, it is clear that the error estimate is reduced k times, if we multiply n by k^2

```
In [16]: k=(answer_1[1]/0.03)
         n=np.int64(np.ceil(n*k**2))
         print("New value of n:",n)
```

```
New value of n: 4167825
```

The value of n is quite large, so in order to compute the final answer, we should not store the full matrix of stock prices when generating the values of $S(T)$.

```
In [17]: final_answer=MC2(S0=100,sigma=0.6,T=0.5,r=0.02,D=0.03,alpha=0.05,n=n,p=p_call,m=m)
         print("The final answer:",final_answer)
         print("Actual error:", abs(final_answer[0]-exact_price))
```

```
The final answer: [16.405883252218885, 0.029935431693475404]
Actual error: 0.0606819622988
```

So the final answer was computed with error that is approximately equal to 0.06.