Lab9_sol

October 27, 2017

1 Sample solutions to exercises of lab 9

1.1 Exercise 1

Write a function that for given values of $n, \rho > 1, r, D, S0, T, \sigma$ and for given functions p, ϕ_1 and ϕ_2 takes m to be equal to the smallest integer satisfying the stability constraint and returns the values $U_{i0},\ i = 0, \ldots, n$ of the approximate solution (option prices) obtained by solving the tarnsformed BS equation with the explicit finite difference method and the corresponding stock prices $S_i = e^{x_i}$ in the case $x_{min} = \ln \frac{S0}{\rho},\ x_{max} = \ln(\rho S0)$. Test the correctness of your code by comparing the results to the exact values obtained by Black-Scholes formula in the case $r = 0.03$, $\sigma = 0.5$, $D = 0.05$, $T = 0.5$, $E = 97$, $S0 = 100$, $p(s) = \max(s - E, 0)$, $\phi_1(t) = p(e^{x_{min}}),\ \phi_2(t) = p(e^{x_{max}})$.

Solution

```
In [1]: import numpy as np
        def explicit_solver(n,rho,r,D,S0,T,sigma,p,phi1,phi2):
            """sigma is assumed to be a constant
            phi1,phi2 are functions of xmin,t and xmax,t
            p is a function of stock price only
            """
            xmax=np.log(S0*rho)
            xmin=np.log(S0/rho)
            delta_x=(xmax-xmin)/n
            #find m from the stability condition
            m=T*(sigma**2/delta_x**2+r)
            m=np.int64(np.ceil(m)) #has to be an integer
            delta_t=T/m
            #define values of x_i
            x=np.linspace(xmin,xmax,n+1)
            #define matrix U with dimension (n+1)x(m+1)
            U=np.zeros(shape=(n+1,m+1))
            #fill in the final condition
            U[:,m]=p(np.exp(x))
            #define a,b,c
            alpha=sigma**2/2
            beta=r-D-alpha
            a=delta_t/delta_x**2*(alpha-beta*delta_x/2)
            b=1-2*delta_t/delta_x**2*alpha-r*delta_t
            c=delta_t/delta_x**2*(alpha+beta*delta_x/2)
```

```
    #compute all other values
    i=np.arange(1,n)
    t=np.linspace(0,T,m+1)
    for k in range(m,0,-1): #backward iteration, k=m,m-1,...
        #boundary conditions
        U[0,k-1]=phi1(xmin,t[k-1])
        U[n,k-1]=phi2(xmax,t[k-1])
        #all other values
        U[i,k-1]=a*U[i-1,k]+b*U[i,k]+c*U[i+1,k]
    return [U[:,0],np.exp(x)]
```

Let us test our solution. For this, we have to define the pay-off function and the functions ϕ_1, ϕ_2

```
In [2]: r = 0.03; sigma = 0.5; D = 0.05;
        T = 0.5; E = 97; S0 = 100;
        def p_call(s):
            return np.maximum(s-E,0)
        def phi1(xmin,t):
            return p_call(np.exp(xmin))
        def phi2(xmax,t):
            return p_call(np.exp(xmax))
        n=6;rho=4
        solution=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1,phi2)
        print("option prices:",solution[0])
        print("stock prices:",solution[1])
```

```
option prices: [   0.            0.            0.64273791   14.42641365   58.99043012
  149.76470435  303.        ]
stock prices: [  25.           39.6850263    62.99605249  100.          158.7401052
  251.98420998  400.        ]
```

Note that the relation to the matrix U computed inside the function `explicit_solver` and the option prices are as follows: if at the time moment $t = t_k$ the logarithm of the current stock price $\ln(S(t))$ is x_i, then the price is approximately U_{ik}. Since we return only to values corresponding to $t = t_0 = 0$, the first vector of the result gives the prices for different possible values of $S(0)$: if $\ln(S(0)) = x_i$, then the option price is the i-th value in the first vector. So x_{min}, x_{max} and n determine, for which stock prices we have approximate values in the matrix U (and thus also in the first component of the result of the function). Now, if we know in advance that we are mostly interested in the case $S(0) = S_0$, where S_0 is a given number, it is good to make sure that $\ln S_0$ is one of the grid points in x direction.

This is why we have defined x_{min} and x_{max} by the formulas given in the lab handout. Namely, since $\frac{x_{min}+x_{max}}{2} = \ln S_0$ (show it!), we have the price corresponding to $S(0) = S_0$ in the matrix U whenever $\frac{x_{min}+x_{max}}{2}$ is a grid point, and that is true for all even values of n. Moreover, the price is exactly in the middle of the $n+1$ values our function returns, which is clearly seen also by looking at the stock prices in the output: the value $S_0 = 100$ is exactly the middle element of the vector of stock prices.

Let us compare the approximate price of the option to the exact value computed by the BS formula:

```
In [3]: approx_sol=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1,phi2)
        approx_price=approx_sol[0][n//2] #the index of an array has to be an integer
        #compare with the exact option price
        import sys
        sys.path.append("c:/users/rkangro/compfin17")
        from BSformulas import Call
        exact_price=Call(S0,E,T,r,sigma,D)
        print(exact_price-approx_price)
```

```
0.130198614064
```

The difference is quite small even for $n = 6$. Check the result for a larger value of n:

```
In [4]: n=100
        approx_sol=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1,phi2)
        approx_price=approx_sol[0][n//2]
        print(exact_price-approx_price)
```

```
0.00415215498838
```

The difference is very small, so our function works correctly.

1.2 Exercise 2

Let $r = 0.02$, $\sigma = 0.6$, $D = 0.03$, $T = 0.5$, $E = 99$, $S0 = 100$, $p(s) = \max(s - E, 0)$. If we use the explicit method of previous exercise, then even if we let n go to infinity there is going to be a finite error between the exact option price at $t = 0, S(0) = S0$ and the corresponding approximate value. This error is caused by introducing artificial boundaries x_{min} and x_{max} and the boundary conditions specified at those boundaries. Use the boundary conditions $\phi_1(t) = p(e^{x_{min}})$, $\phi_2(t) = p(e^{x_{max}})$ and determine the value of the resulting error for $\rho = 1.5, 2, 2.5$. In order to see the resulting error you should do several computations with fixed ρ and increasing values of n (assuming m is determined from the stability condition, n should be increased by multiplying it by 2 each time). Use the knowledge that for large enough n the part of the error depending on the choice of n behaves approximately like $\frac{const.}{n^2}$ (so the difference of the last two computations divided by 3 is an estimate of this part of the error for the last computation) for determining how far your last computation is from the limiting value.

Solution

```
In [5]: r = 0.02; sigma = 0.6; D = 0.03;
        T = 0.5; E = 99; S0 = 100;
        exact_price=Call(S0,E,T,r,sigma,D)
```

First value of ρ:

In [6]:
```
rho=1.5
n=10
solution=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1,phi2)
price1=solution[0][n//2]
print(price1-exact_price)
#this error has two parts, truncation error + the part that depends on n (discretization
n=n*2
solution=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1,phi2)
price2=solution[0][n//2]
print(price2-exact_price)
#estimate how )much changing n can still change the answer
print("The approximate price may change by",np.abs((price2-price1)/3))
```

```
-0.661498128125
-0.696321913735
The approximate price may change by 0.0116079285367
```

So the limiting error (when $n \to \infty$) is approximately 0.69
The second value of ρ:

In [7]:
```
rho=2
n=10
solution=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1,phi2)
price1=solution[0][n//2]
print(price1-exact_price)
#this error has two parts, truncation error + the part that depends on n (discretization
n=n*2
solution=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1,phi2)
price2=solution[0][n//2]
print(price2-exact_price)
#estimate how )much changing n can still change the answer
print("The approximate price may change by",np.abs((price2-price1)/3))
```

```
-0.0152814642468
0.0317556020778
The approximate price may change by 0.0156790221082
```

Since the approximate price can change quite a lot compared to the computed difference (about 50%), let us keep increasing n

In [8]:
```
price1=price2
n=n*2
solution=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1,phi2)
price2=solution[0][n//2]
print(price2-exact_price)
#estimate how )much changing n can still change the answer
print("The approximate price may change by",np.abs((price2-price1)/3))
```

```
0.0343809095772
The approximate price may change by 0.000875102499802
```

So the limiting error is approximately 0.034
The last value of ρ:

```
In [10]: rho=2.5
         solution=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1,phi2)
         #let us continue computations until the relative change
         #of the approximate price is smaller than 1% of the
         #computed error
         price2=solution[0][n//2]
         change=1# to force the cycle to start
         error=1 # to force the cycle to start
         while(change/np.abs(error)>0.01):
             price1=price2
             n=n*2
             solution=explicit_solver(n,rho,r,D,S0,T,sigma,p_call,phi1,phi2)
             price2=solution[0][n//2]
             error=price2-exact_price
             change=np.abs((price2-price1)/3)
         print("n=",n," limiting error =",error)
```

```
n= 640  limiting error = 0.0117869590413
```

So we see that the error caused by the location of x_{min} and x_{max} is approaching 0 quickly when ρ increases. For most practical problems $\rho = 3$ is good enough but of cause we should do some computations to check if this is indeed the case.