Lab10_sol

November 3, 2017

1 Sample solutions to exercises of lab 10

In the exercises 1 and 2 we consider an European option with $T = 0.5$ and payoff function

$$p(s) = \begin{cases} \frac{195-2\cdot s}{8}, & s < 95, \\ \frac{(s-100)^2}{40}, & 95 \leq s \leq 120, \\ s - 110, & s > 120. \end{cases}$$

1.1 Exercise 1

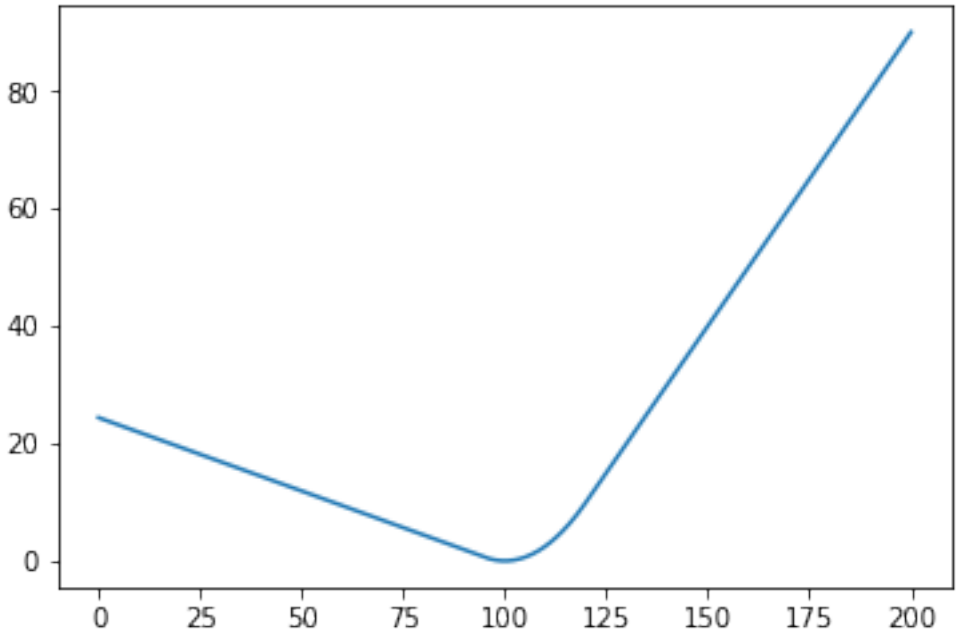
Derive suitable boundary conditions (based on special solutions of BS equation) for the transformed Black-Scholes equation for the payoff function defined above.

Solution

Let us first define the payoff function and look at it's graph

```
In [1]: import numpy as np
        def p(s):
            return (195-2*s)/8*(s<95)+(s-100)**2/40*(s>=95)*(s<=120) + (s-110)*(s>120)

        import pylab as pl
        s=np.linspace(0,200,400)
        pl.plot(s,p(s))
        pl.show()
```



Using the formula for the special solution and matching it to the form of the payoff function for small s values and for large s values, we get the following formulas for the functions ϕ_1 and ϕ_2:

```
In [4]: def phi1(xmin,t):
            c1=-2/8
            c2=195/8
            return c1*np.exp(-D*(T-t))*np.exp(xmin)+c2*np.exp(-r*(T-t))

        def phi2(xmax,t):
            c1=1
            c2=-110
            return c1*np.exp(-D*(T-t))*np.exp(xmax)+c2*np.exp(-r*(T-t))
```

1.2 Exercise 2

Implement the explicit finite difference method described in the previous lab so that it works with non-constant α and β. Test your solver by using the boundary conditions derived in the previous exercise in the case $n = 20$, $\rho = 1.5$ and

$$\sigma(s,t) = 0.5 + 0.2 \cdot e^{-t} \arctan(0.1\,s - 10).$$

The answer depends on how a suitable value of m is determined from the stability condition, but should be close to 10.67.

Solution Let us modify the solver from Lab 9 so that it works in the case when volatility is a function of stock price and time

```
In [8]: def explicit_solver2(n,rho,r,D,S0,T,sigma,p,phi1,phi2):
            """sigma is assumed to be a function of s and t
            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
            x=np.linspace(xmin,xmax,n+1)
            #find m from the stability condition
            #for this, we first have to estimate the maximum of the coefficient alpha
            #for transformed BS equation, alpha is defined as follows
            def alpha(x,t):
                return sigma(np.exp(x),t)**2/2
            #estimate max value of alpha
            #choose m for estimating max of alpha
            m=100
            t=np.linspace(0,T,m+1)
            alpha_max=0 #the max value seen so far
            for i in range(1,n):
                current_max=np.max(alpha(x[i],t))
                alpha_max=np.max((alpha_max,current_max))
            #now max value is computed
            #increase the estimate a little to be on the safe side
            alpha_max=alpha_max+10*T/(2*m)
            #now we are ready to estimate m from the stability condition
            m=T*(2*alpha_max/delta_x**2+r)
            m=np.int64(np.ceil(m)) #has to be an integer
            delta_t=T/m
            #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))
            #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,...
                #compute a,b,c
                alpha_vec=alpha(x[i],t[k])
                beta=r-D-alpha_vec
                a=delta_t/delta_x**2*(alpha_vec-beta*delta_x/2)
                b=1-2*delta_t/delta_x**2*alpha_vec-r*delta_t
                c=delta_t/delta_x**2*(alpha_vec+beta*delta_x/2)
                #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[n//2,0] #option price at t=0 when S(0)=S0
```

Check the solver:

```
In [9]: r = 0.01;  D = 0.05;
        T = 0.5;  S0 = 97;
        def sigma(s,t):
            return 0.5+0.2*np.exp(-t)*np.arctan(0.1*s-10)
        rho=1.5
        n=20
        approx_sol=explicit_solver2(n,rho,r,D,S0,T,sigma,p,phi1,phi2)
        print(approx_sol)
```

```
10.6714133004
```

1.3 Exercise 3

Consider the European option with the pay-off function

$$p(s) = \begin{cases} \frac{s}{4}, & 0 \le s < 40, \\ \frac{1}{100}(s-40)^3 - \frac{1}{6}(s-40)^2 + \frac{s}{4}, & 40 \le s \le 55, \\ 2(s-50) & s > 55. \end{cases}$$

Assume $r = 0.01$, $D = 0.05$, $S(0) = 50$, $T = 1$ and that the volatility function is given by

$$\sigma(s,t) = 0.5 + \frac{0.2 \cdot e^{-t}}{1 + 0.01 \cdot (s-40)^2}.$$

For $\rho = 2.5$ and both in the case of the boundary conditions given by appropriate special functions and in the case of using the simple (constant) boundary conditions, determine the limiting value (as $n \to \infty$) of the approximate option price $v(50,0)$ with the accuracy 0.005 by computing the approximate values by the explicit finite difference method for $n = 10, 20, 40, \ldots$ and by estimating the accuracy of the last result by

$$\frac{1}{3}|\,\text{last\_result} - \text{previous\_result}\,|.$$

Define the parameters and the pay-off function needed for solving the problem.

```
In [10]: r = 0.01; D = 0.05;
         T = 1; S0 = 50;
         def p(s):
             return s/4*(s<40)+((s-40)**3/100-(s-40)**2/6+s/4)*(s>=40)*(s<=55)+2*(s-50)*(s>55)
         def phi1_const(xmin,t):
             return p(np.exp(xmin))
         def phi2_const(xmax,t):
             return p(np.exp(xmax))
         def phi1_spec(xmin,t):
             c1=1/4
```

```
        c2=0
        return c1*np.exp(-D*(T-t))*np.exp(xmin)+c2*np.exp(-r*(T-t))

    def phi2_spec(xmax,t):
        c1=2
        c2=-100
        return c1*np.exp(-D*(T-t))*np.exp(xmax)+c2*np.exp(-r*(T-t))
    def sigma(s,t):
        return 0.5+0.2*np.exp(-t)/(1+0.01*(s-40)**2)
    rho=2.5
    limitingError=0.005
```

First, consider the case of constant boundary conditions

```
In [11]: error=limitingError+1 #to force the for cycle to start
         n=10
         previous_result=explicit_solver2(n,rho,r,D,S0,T,sigma,p,phi1_const,phi2_const)
         while(error>limitingError):
             n=n*2
             last_result=explicit_solver2(n,rho,r,D,S0,T,sigma,p,phi1_const,phi2_const)
             error=np.abs(previous_result-last_result)/3
             print("n=",n," last_result =", last_result, "error=",error)
             previous_result=last_result

n= 20  last_result = 24.1453994342 error= 0.0033588135703
```

So the limiting value in the case of constant boundary conditions is 24.145
Now the same with the special boundary conditions:

```
In [21]: error=limitingError+1 #to force the for cycle to start
         n=10
         previous_result=explicit_solver2(n,rho,r,D,S0,T,sigma,p,phi1_spec,phi2_spec)
         while(error>limitingError):
             n=n*2
             last_result=explicit_solver2(n,rho,r,D,S0,T,sigma,p,phi1_spec,phi2_spec)
             error=abs(previous_result-last_result)/3
             print("n=",n," last_result =", last_result, "error=",error)
             previous_result=last_result

n= 20  last_result = 23.9824635281 error= 0.00127806170795
```

The limiting value is in this case 23.982. The difference is more than 0.01, so limiting values are clearly different. By using a relatively large value of rho (like 8, for example), we could estimate what is the actual option price and then it would be clear, which boundary conditions gave a better result.

```
In [12]: rho=8
         error=limitingError+1 #to force the for cycle to start
         n=10
         previous_result=explicit_solver2(n,rho,r,D,S0,T,sigma,p,phi1_spec,phi2_spec)
         while(error>limitingError):
             n=n*2
             last_result=explicit_solver2(n,rho,r,D,S0,T,sigma,p,phi1_spec,phi2_spec)
             error=np.abs(previous_result-last_result)/3
             print("n=",n," last_result =", last_result, "error=",error)
             previous_result=last_result

n= 20  last_result = 23.9439114802 error= 0.284173982222
n= 40  last_result = 23.9907287507 error= 0.0156057568208
n= 80  last_result = 23.9963802258 error= 0.00188382504691
```

As we see, the result was more accurate in the case of special boundary condition. Of cause, considering the result for rho=8 to be the exact value requires some justification and in later labs we'll learn about a procedure for estimating the effect of rho on the accuracy of the final result.