Lab13_sol

November 29, 2017

1 Sample solutions of exercises of Lab13

1.1 Exercise 1

Modify solvers using the basic implicit methods for computing American options by solving original and transformed BS equation. Use the methods for computing approximate prices of the American put option in the case $r = 0.1$, $\sigma = 0.5$, $D = 0$, $T = 0.5$, $E = 100$, $S_0 = 100$, $p(s) = \max(E - s, 0)$ and for $n = 10, 20, 40, 80, 160$, $m = 5, 20, 80, 320, 1280$ (5 computations in total). Use x_{max} that corresponds to the maximal stock price $2 \cdot S_0$ (for transformed problem, it means that $x_{max} = \ln(2 \cdot S_0)$) and in the case of solving the transformed problem, $x_{min} = \ln \frac{S0}{2}$ ($x_{min} = 0$ in the case of solving untransformed problem). In each case, compute the logarithms of the absoculte values of differences between consecutive results and present them on a graph. Can we say that the values lie approximately on a straight line? If we can, what is the slope of the line?

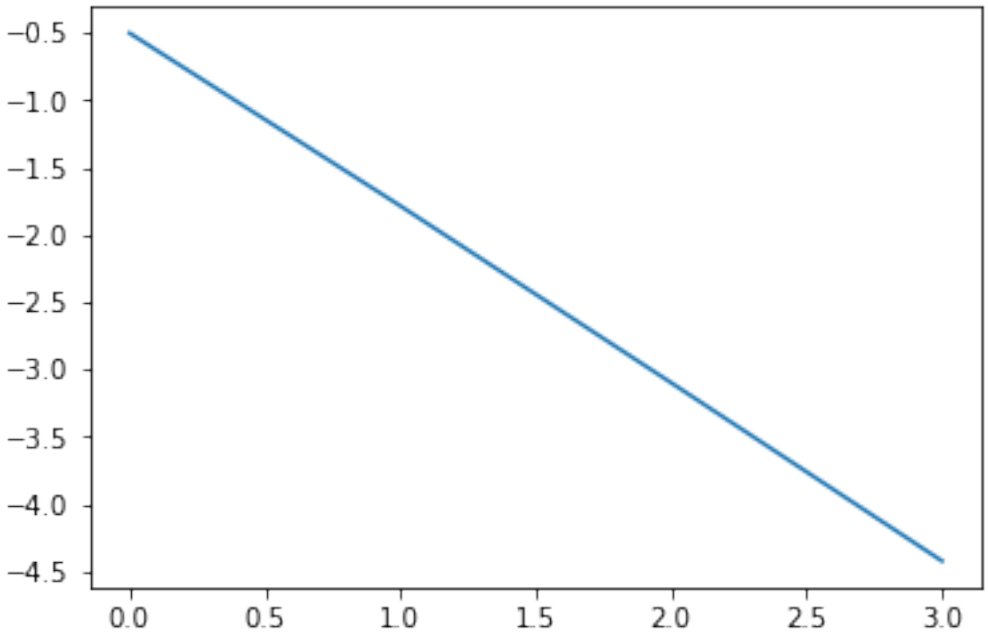
Solution Let us consider first the solver for transformed BS equation.

```
In [1]: import numpy as np
        from scipy import linalg

        def implicit_solver_TBS(m,n,xmin,xmax,r,D,T,sigma,p,phi1,phi2,American=False):
            """sigma is assumed to be a function of s and t
            phi1,phi2 are functions of xmin,t and xmax,t
            """
            delta_x=(xmax-xmin)/n
            x=np.linspace(xmin,xmax,n+1)
            #define the function alpha
            #for transformed BS equation
            def alpha(x,t):
                return sigma(np.exp(x),t)**2/2
            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))
            #compute all other values
            i=np.arange(1,n)
            t=np.linspace(0,T,m+1)
```

```
    #define matrix M
    M=np.zeros(shape=(n+1,n+1))
    M[0,0]=1
    M[n,n]=1
    #define vector F
    F=np.zeros(n+1)
    for k in range(m-1,-1,-1):
        #compute the coefficients
        beta=r-D-alpha(x[i],t[k])
        a=delta_t/delta_x**2*(-alpha(x[i],t[k])+beta*delta_x/2)
        b=1+2*delta_t/delta_x**2*alpha(x[i],t[k])+r*delta_t
        c=-delta_t/delta_x**2*(alpha(x[i],t[k])+beta*delta_x/2)
        #Fill M with right values
        M[i,i-1]=a
        M[i,i]=b
        M[i,i+1]=c
        #Fill F
        F[0]=phi1(xmin,t[k])
        F[n]=phi2(xmax,t[k])
        F[i]=U[i,k+1]
        #solve the system
        U[:,k]=linalg.solve(M,F)
        if American:
            U[:,k]=np.maximum(U[:,k],U[:,m])
    return U[:,0] #option price for t=0
#test the solver
r = 0.1; D = 0; T = 0.5; E = 100; S0 = 100

#for transformed equation, price corresponding to S0 is for this xmin, xmax in the mid
def sigma(s,t):
    return(0.5)
def p_put(s):
    return np.maximum(E-s,0)
def phi1_const_TBS(xmin,t):
    return p_put(np.exp(xmin))
def phi2_const_TBS(xmax,t):
    return p_put(np.exp(xmax))
rho=2
xmin=np.log(S0/rho)
xmax=np.log(S0*rho)
n=10
m=5
k=5 #how many results to compute
prices=np.zeros(k)
for i in range(k):
    prices[i]=implicit_solver_TBS(m,n,xmin,xmax,r,D,T,sigma,p_put,phi1_const_TBS,phi2_
    n*=2
    m*=4
```

```
        print(prices)
        dif=prices[1:]-prices[:-1]
        print(dif)
        import pylab as pl
        pl.plot(np.log(dif))
        pl.show()
```

```
[ 11.07012914  11.67524142  11.84278767  11.88782766  11.89981267]
[ 0.60511228  0.16754625  0.04503999  0.011985  ]
```



Looking at the differences, it is quite clear that at every step the differences are reduced approximately 4 times, which is consistent with the error estimate of the form $O(\Delta t + \Delta x^2)$. The logarithm of the differences displays linear decay, which is an alternative way to verify that at each new iteration the discretisation error is reduced by a fixed factor. The average slope of the line can be computed as follows:

```
In [2]: slope=np.log(dif[-1]/dif[0])/(len(dif)-1)
        slope
```

```
Out[2]: -1.3072526081530544
```

It is quite easy to figure out, that the factor of reduction at each iteration is e^{-slope}

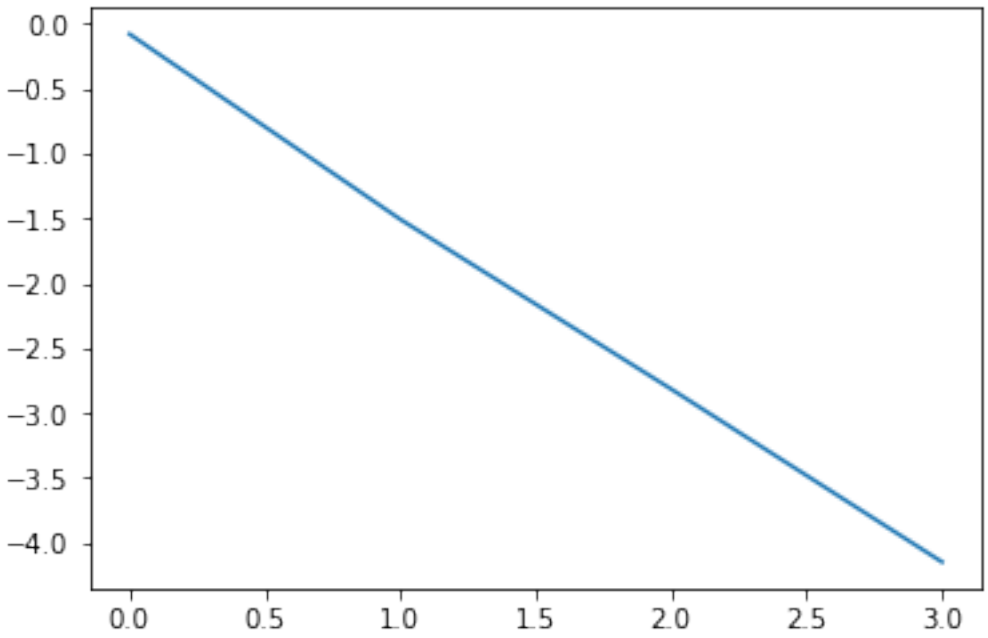
```
In [3]: np.exp(-slope)
```

```
Out[3]: 3.6960053753687965
```

Similar computations in the case of untransformed problem:

```
In [4]: def implicit_solver_BS(m,n,xmin,xmax,r,D,T,sigma,p,phi1,phi2,American=False):
            """sigma is assumed to be a function of s and t
            phi1,phi2 are functions of xmin,t and xmax,t
            """
            delta_x=(xmax-xmin)/n
            x=np.linspace(xmin,xmax,n+1)
            #define the function alpha
            #for transformed BS equation
            def alpha(x,t):
                return sigma(x,t)**2*x**2/2
            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(x)
            #compute all other values
            i=np.arange(1,n)
            t=np.linspace(0,T,m+1)
            #define matrix M
            M=np.zeros(shape=(n+1,n+1))
            M[0,0]=1
            M[n,n]=1
            #define vector F
            F=np.zeros(n+1)
            for k in range(m-1,-1,-1):
                #compute the coefficients
                beta=(r-D)*x[i]
                a=delta_t/delta_x**2*(-alpha(x[i],t[k])+beta*delta_x/2)
                b=1+2*delta_t/delta_x**2*alpha(x[i],t[k])+r*delta_t
                c=-delta_t/delta_x**2*(alpha(x[i],t[k])+beta*delta_x/2)
                #Fill M with right values
                M[i,i-1]=a
                M[i,i]=b
                M[i,i+1]=c
                #Fill F
                F[0]=phi1(xmin,t[k])
                F[n]=phi2(xmax,t[k])
                F[i]=U[i,k+1]
                #solve the system
                U[:,k]=linalg.solve(M,F)
                #if american option, the values should be above the payoff
                if American:
                    U[:,k]=np.maximum(U[:,k],U[:,m])
            return U[:,0] #option prices for t=0
```

```
def phi1_BS(xmin,t):
    return(p_put(xmin))
def phi2_BS(xmax,t):
    return p_put(xmax)
xmin=0
xmax=2*S0
m=5
n=10
k=5
prices=np.zeros(k)
for i in range(k):
    prices[i]=implicit_solver_BS(m,n,xmin,xmax,r,D,T,sigma,p_put,phi1_BS,phi2_BS,Ameri
    n*=2
    m*=4
dif_errors=prices[1:]-prices[:-1]
import pylab as pl
pl.plot(np.log(dif_errors))
pl.show()
```



1.2 Exercise 2

Find the price of the American put option considered in the previous exercise with maximal error 0.01 for current stock price $S0 = 100$.

Solution Let us compute the price by solving TBS with the basic implicit method. Let us improve the code of Lab 12 a little by defining a function for computing a result by a given discretisation error in the case when Runge's error estimate is used.

```
In [5]: def Runge(m0,n0,mfactor,nfactor,q,value,error):
            """value is a function of m,n, should return answer (a number)
            if we multipy m by mfactor,n by nfactor, the error is reduced by q
            error - how accurate answer we want
            """
            m=m0
            n=n0
            answer1=value(m,n)
            estimate=error+1
            while estimate>error:
                m=m*mfactor
                n=n*nfactor
                answer2=value(m,n)
                estimate=np.abs(answer2-answer1)/(q-1)
                answer1=answer2
                print("runge",m,n,rho,estimate)
            return answer2
```

Now, if we want to compute a price of option with a given accuracy, we should define a function that for given m and n returns an approximate price of the option. In the case of using the explicit method, the value of m given to the function is actually not used in the computation

```
In [6]: def price_implicit(m,n):
            return implicit_solver_TBS(m,n,xmin,xmax,r,D,T,sigma,p_put,phi1_const_TBS,phi2_cons
```

Now we can start the procedure of computing the price with a given accuracy. Of cause this part of the code can also be improved by defining a suitable function but this improvement is left as an exercise for the reader.

```
In [7]: total_error=0.01
        rho=2
        #after changing rho, correct xmin and xmax should be computed
        xmin=np.log(S0/rho)
        xmax=np.log(S0*rho)
        answer_rho1=Runge(5,10,4,2,4,price_implicit,total_error/2)
        estimate_rho=total_error
        z=1
        while estimate_rho>total_error/2:
           z=z+1
           rho=rho*2
           xmin=np.log(S0/rho)
           xmax=np.log(S0*rho)
           answer_rho2=Runge(10,10*z,4,2,4,price_implicit,total_error/2)
           estimate_rho=np.abs(answer_rho1-answer_rho2)
           answer_rho1=answer_rho2
        print("the price of the option is",answer_rho2)
```

```
runge 20 20 2 0.201704094174
runge 80 40 2 0.0558487490321
```

```
runge 320 80 2 0.0150133300565
runge 1280 160 2 0.00399500135595
runge 40 40 4 0.143406147661
runge 160 80 4 0.0374175907393
runge 640 160 4 0.00991818598061
runge 2560 320 4 0.00258860008506
the price of the option is 11.9016463335
```

1.3 Exercise 3

If we use the transformed equation for finding approximate prices, we can approximate the derivative of the option price with respect to s by

$$\frac{\partial v}{\partial s}(S(0),0) = \frac{1}{S(0)}\frac{\partial u}{\partial x}(\ln S(0),0) \approx \frac{1}{S(0)}\frac{U_{i+1,0}-U_{i-1,0}}{2\Delta x},$$

where i is such that $x_i = \ln S(0)$. If we use the untransformed equation for finding approximate prices, we can approximate the derivative of the option price with respect to s by

$$\frac{\partial v}{\partial s}(S(0),0) \approx \frac{U_{i+1,0}-U_{i-1,0}}{2\Delta s},$$

where i is such that $s_i = S(0)$. If we use our standard methods of choosing m and n values, then the discretization error of those approximations is $O(\Delta x)$ for the transformed equation and $O(\Delta s)$ for the untransformed equation and hence is reduced approximately by two when we multiply n by 2. Find the value $\frac{\partial v}{\partial s}(100,0)$ of the price v of American put option with maximal error 0.001 for the option considered in the previous exercise by using the basic implicit method for solving untransformed BS equation.

Solution Here a solution by using the solver for TBS using the basic implicit method is given.

Let us define the funciton for computing the derivative in the case of the untransformed problem:

```
In [8]: def derivative(m,n,American,i_price):
            prices=implicit_solver_BS(m,n,xmin,xmax,r,D,T,sigma,p_put,phi1_BS,phi2_BS,American
            delta_s=(xmax-xmin)/n
            return (prices[i_price+1]-prices[i_price-1])/(2*delta_s)
        m=5
        n=10
        xmin=0
        xmax=2*S0
        derivative(m,n,True,n//2)

Out[8]: -0.41052489238768908
```

Now we can use same procedure as before, but have to take into account that the error of the answer is reduced by 2, when n is multiplied by 2 and m is multiplied by 4.

```
In [9]: total_error=0.001
        rho=2
```

```
        xmin=0
        American=True
        def value(m,n):
            return derivative(m,n,American,n//rho)
        answer_rho1=Runge(10,10,4,2,2,value,total_error/2)
        estimate_rho=total_error
        z=1
        while estimate_rho>total_error/2:
           z=z*2 #untransformed problem, xmin is 0
           rho=rho*2
           xmax=rho*S0
           answer_rho2=Runge(10,10*z,4,2,2,value,total_error/2)
           estimate_rho=np.abs(answer_rho1-answer_rho2)
           answer_rho1=answer_rho2
        print("the delta of the option is",answer_rho2)
```

```
runge 40 20 2 0.00906788367614
runge 160 40 2 0.00170816565212
runge 640 80 2 0.000448905656197
runge 40 40 4 0.0089144041926
runge 160 80 4 0.00167544166334
runge 640 160 4 0.000441312190629
the delta of the option is -0.400293261017
```