Lab2_sol

September 14, 2017

1 Sample solutions of lab 2

1.1 Exercise 1

Write a module containing Black-Scholes Call an Put pricing functions

$$Call(S,E,T,r,\sigma,D) = Se^{-DT}\Phi(d_1) - Ee^{-rT}\Phi(d_2)$$

$$Put(S,E,T,r,\sigma,D) = -Se^{-DT}\Phi(-d_1) + Ee^{-rT}\Phi(-d_2)$$

where

$$d_1 = \frac{\ln(\frac{S}{E}) + (r - D + \frac{\sigma^2}{2}) \cdot T}{\sigma\sqrt{T}},\ d_2 = d_1 - \sigma\sqrt{T}$$

and Φ is the cumulative distribution function of the standard normal distribution

Solution The content of the file `BSformulas.py` shoulb be as follows

```
In [13]: import numpy as np
         from scipy import stats
         Phi=stats.norm.cdf
         def Call(S,E,T,r,sigma,D):
             d1=(np.log(S/E)+(r-D+sigma**2/2)*T)/(sigma*np.sqrt(T))
             d2=d1-sigma*np.sqrt(T)
             return(S*np.exp(-D*T)*Phi(d1)-E*np.exp(-r*T)*Phi(d2))
         def Put(S,E,T,r,sigma,D):
             d1=(np.log(S/E)+(r-D+sigma**2/2)*T)/(sigma*np.sqrt(T))
             d2=d1-sigma*np.sqrt(T)
             return(-S*np.exp(-D*T)*Phi(-d1)+E*np.exp(-r*T)*Phi(-d2))
```

After running the previous definitions (or importing Call and Put from the file), execution the following lines in a console window one should give the same results as shown below

```
In [14]: Call(S=100,E=100,T=0.5,sigma=0.5,r=0.05,D=0.01)

Out[14]: 14.830417641356284

In [15]: Put(S=100,E=100,T=0.5,sigma=0.5,r=0.05,D=0.01)

Out[15]: 12.86016092492131
```

1.2 Exercise 2

The most common situation is that our programs compute the vector of values U_i, $i = 0, 1, \ldots, n$ that correspond to the values $u(x_i)$, $i = 0, 1, \ldots, n$, where x_i are equally spaced in some interval $[a, b]$, so that $x_i = a + i \cdot h$, $i = 0, 1, \ldots, n$, where $h = \frac{b-a}{n}$. Define a Python function `u1(x,U,a,b)` that returns the value of the function u for given argument x in the case when vector U contains values of the function u for equally spaces points in the interval $[a,\ b]$, assuming that x is one of the points for which the there is the corresponding value in the vector U. Use the following algorithm:

1. Determine the number of intervals n from the fact that U contains $n + 1$ values.
2. Compute h (the length of intervals (x_i, x_{i+1}), $i = 0, 1, \ldots, n - 1$).
3. Compute the index i corresponding to the given value of x so that $x_i = x$.
4. Return corresponding value U_i of the vector U as the answer.

Test your code in the case when values U are computed according to the function $u(x) = x^2$.

Solution

```
In [16]: def u1(x,U,a,b):
             n=np.size(U)-1
             h=(b-a)/n
             i=np.int64((x-a)/h)
             # the result should be an integer and it is good
             #if this works also when x is a vector of values
             return U[i]
```

Testing the code:

```
In [17]: n=5
         x=np.linspace(0,1,n+1)
         U=x**2
         print(u1(0.4,U,0,1))
```

```
0.16
```

1.3 Exercise 3

The function `u1` should work also when $x \in [a, b]$ is not one of the points x_i, but it does not work very well: if the value of x is close to but smaller than x_5, for example, then the function returns the value of u corresponding to x_4, which may be quite different from the actual value of $u(x)$. A better idea is to make the values returned by `u1` to change continuously when x moves in the interval $[a, b]$ so that at the points x_i we get the values in the vector U. The simplest idea to achieve that is to use linear interpolation: if we know the value u_c for $x = c$ and u_d for $x = d$, then for all $x \in [c, d]$ we compute values according to the linear function

$$u_c \frac{d-x}{d-c} + u_d \frac{x-c}{d-c}.$$

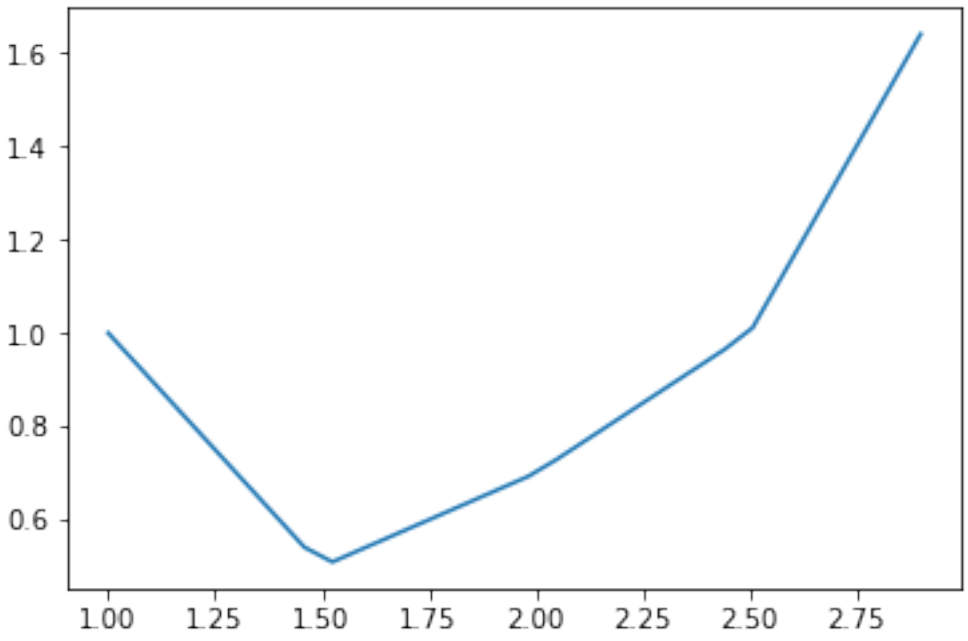
Define a function `u2(x,U,a,b)` that uses linear interpolation for computing approximate values of the function u for any value of $x \in [a, b]$. Plot the graph of the function `u2` in the case $U = (1, 0.5, 0.7, 1, 1.8)$, $a = 1$, $b = 3$ for $x \in [1, 2.9]$ by using the values of the function at 30 points.

Solution

```
In [18]: def u2(x,U,a,b):
             n=np.size(U)-1
             h=(b-a)/n
             xi=np.linspace(a,b,n+1)
             i=np.int64((x-a)/h)
             i=np.minimum(i,n-1)# the following code does not work if i>=n
             i=np.maximum(i,0)# this way it works also if x
             #(or some value of x, if x is a vector) is smaller than a
             c=xi[i]
             d=xi[i+1]
             uc=U[i]
             ud=U[i+1]
             h=d-c #the length of the interval containing x
             return uc*(d-x)/h+ud*(x-c)/h
```

Test the correctness:

```
In [19]: U=np.array([1,0.5,0.7,1,1.8])
         n=30
         x=np.linspace(1,2.9,n)
         import pylab as pl
         pl.plot(x,u2(x,U,1,3))
         pl.show()
```



1.4 Exercise 4

Sometimes we compute values of the function u for unevenly spaced argument values. Define a function `u3(x,U,X)` that computes the approximate values of $u(x)$ by linear interpolation. Here the vector X contains x-values in the increasing order for which we have values of $u(x)$ in the vector U. Verify the correctness of your function by comparing it's value to the value computed by the function `interp` from scipy packege in the case $x = 0.9$, $U = (1.5, 0.8, 1, 4)$, $X = (-1, 0.5, 1.7, 3)$).

Solution Now it is not very easy to use vector operations to define such function so that it works also when the argument `x` is a vector. A possible solution that works only if `x` is a real number is as follows:

```
In [20]: def u3(x,U,X):
             n=np.size(U)-1
             #find i such that X[i]<=x<=X[i+1]
             i=0
             while i<n-1 and X[i+1]<x:
                 i=i+1
             c=X[i]
             d=X[i+1]
             uc=U[i]
             ud=U[i+1]
             h=d-c
             return uc*(d-x)/h+ud*(x-c)/h
```

Let us check our solution: (note that you can see the information about `interp` by typing help(np.interp) in the console)

```
In [21]: U=np.array([1.5,0.8,1,4])
         X=np.array([-1,0.5,1.7,3])
         print(u3(0.9,U,X))
         print(np.interp(0.9,xp=X,fp=U))

0.866666666667
0.8666666666666667
```