Computational Finance
Computer Lab 2

The aim of the Lab is to define some useful functions given by Black-Scholes option pricing formulas and to learn to do simple computations based on a table of values of a function.
Two important functions in Mathematical Finance are Black-Scholes formulas for Call and Put option prices (giving the right to buy or sell one share of the stock S after time T years for the price E) under the assumption, that the Black-Scholes market model

$$dS(t) = S(t)(\mu\, dt + \sigma dB(t)),$$

where B is the standard Brownian motion, holds with constant volatility σ. The formulas are as follows:

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

where

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

S is the current stock price, T is the time to expiry of the option and Φ is the cumulative distribution function of the standard normal distribution and D is the continuously payable dividend rate of the stock ($D = 0$ for usual stocks, but is different from 0 if the underlying is a foreign currency). We are going to use the functions often for making sure that our code is working correctly.

Exercise 1. Let us write a module containing Black-Scholes Call an Put pricing functions so that later we can import them similarly to importing functions from other python packages. For this it is good to know that in Python the exponent function e^x is written as `np.exp(x)`, $\ln(x)$ is `np.log(x)`, $\sqrt{x}$ is `np.sqrt(x)` and that the multiplication sign that is often not present in mathematical texts, has to be explicitly written as $*$ in Python. We also need to define the function Φ as indicated in the help file "'Python commands needed for this course"'.

In order to get you going, the code for the Call and Put options is partially given as follows:

```
import numpy as np
from scipy import stats #needed for Phi
Phi=stats.norm.cdf #the cumulative distribution function for the standard normal
#now we can define the functions
def Call(S,E,T,r,sigma,D):
     d1=(np.log(S/E)+
     d2=d1-
     answer=S*np.exp(-D*T)*Phi(
     return(answer)
def Put(
```

Please complete the code. Important: a common mistake is to write something like `a/b*c` when you want to compute $\frac{a}{b\,c}$. This is not correct, the former code divides a with b and multiplies the result with c; the correct way is to use appropriate parentheses like `a/(b*c)` to indicate clearly what goes into the denominator.

Save the code in a file named BSformulas.py in your usual directory (so that you can look at the code later when needed). Then execute the file (so that it defines the functions) and check the correctness by typing the lines

```
Call(S=100,E=100,T=0.5,sigma=0.5,r=0.05,D=0.01)
Put(S=100,E=100,T=0.5,sigma=0.5,r=0.05,D=0.01)
```

into the interpreter window. The answers should be 14.830417641356284 for Call and 12.86016092492131 for Put. Later it is possible to import Put and Call functions from the module BSformulas, but you have to say to Python where BSformulas.py is located. This can be done by importing the `sys` library (by writing `import sys` in your file) and using the command `sys.path.append("path_to_the_directory")` before trying to import Put and Call from BSformulas. Try that out: open a new editor window, type in lines

```
import numpy as np
import sys
#instead of h:/compfin write the correct path!
sys.path.append("h:/compfin")
import BSformulas as bs
print("The price of Call:",bs.Call(S=100,E=100,T=0.5,sigma=0.5,r=0.05,D=0.01))
print("The price of Put:", bs.Put(S=100,E=100,T=0.5,sigma=0.5,r=0.05,D=0.01))
```

and run the code.

When we apply mathematics for solving real word problems, we describe some quantities of interest as functions of some variables (for example, we may say, that option price is a function of stock price and time). In computational mathematics we often can not find a formula for the function but only a table of values of the function of interest for a certain number of arguments and we have to know how to relate the tables (given in vectors and matrices) to the values of the function for given values of the arguments. Additionally, we have to know how to get a reasonable value for the function in the case when we do not have the value of the function for given arguments in the table (for example, if the values of the function u are computed only for $x = 0, 1, 2, \ldots, 10$, but we want to know the value $u(0.7)$).

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 a 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 $x_i = x$. The result can be transformed to integers with the function `np.int64()`.
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 $u(x) = x^2$.

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 between the known values: 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 of the function 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.

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 `numpy` package in the case $x = 0.9$, $U = (1.5, 0.8, 1, 4)$, $X = (-1, 0.5, 1.7, 3)$.