Problem Set 1
Root Finding and Implied Volatility

- This homework consists of three questions, each carrying equal marks. Your code will be graded by a Python script which compares your result against the baseline result. It is therefore important that your Python stack is identical to the baseline so that you avoid losing marks even though your code is correct.
- Submit a 7z compressed archive containing four modules, `BS.py`, `test_BS.py`, `Bisect.py`, `BSImplVol.py` and a report in PDF format.
- Be sure to include your name as part of the archive filename, and at the top of your report.

Problem 1. *Write a Python module* `BS.py` *with the Black-Scholes value, delta, and vega (sensitivity to vol) implemented as*

```
def bsformula( callput, S0, K, r, T, sigma, q=0.):
```

This function returns a 3-tuple `optionValue, delta, vega`. *Here,* `callput` *is 1 for a call and -1 for a put, and* `q` *is a continuous return rate on the underlying, for example a foreign interest rate or a dividend rate. Check it against another Black-Scholes pricer (such as* `blsprice` *in MatLab, or any of numerous websites) that you can trust. Use its output in creating the Python module* `test_BS.py` *with unit tests for your* `bsformula` *function.*

Problem 2. *Write a python module* `Bisect.py` *containing an implementation of the bisection method for finding roots of one-dimensional equations. Your program should first recognize when the given bounds do not contain a solution by checking the minimum and maximum value of the function on the bounded domain, and raise an exception. If no bounds are supplied, carefully seek left and right until a root has been bounded. Raise an exception if this is not possible.*

Return the entire series of x-values tested by your method as a `numpy` *array. Generally the final item of your array shall be the solution, and of course the length of the array tells how many calls were made to the function. Raise an exception if the maximum iteration count is reached.*

```
def bisect(target, targetfunction, start=None,
           bounds=None, tols=[0.001,0.010], maxiter=1000)
```

where

- `target` *is the target value for the function f*
- `function` *is the handle for the function f.*
- `start` *is the x-value to start looking at. If* `None`*, the mean of the upper and lower bounds shall be used. Subsequent steps shall*

always use the mean of the active bounds. This input is used only when the initial bounds have not been supplied.

- `bounds` *is the upper and lower bound beyond which* x *shall not exceed.*
- `tols` *are the stopping criteria, the distance between successive* x*-values that indicates success and the difference between target and the* y*-value that indicates success.*
- `maxiter` *is the maximum iteration count the solver shall note exceed.*

Problem 3. *Create a file* `BSImplVol.py` *capable of using either your bisective root finder or Newton-Raphson to get implied volatility*

```
def bsimpvol( callput, S0, K, r, T, price, q=0.,
     priceTolerance=0.01, method='bisect' , reportCalls=False ):
```

The `method` *input may be* `method='newton'` *in order to force use of Newton's method (Hint: this use of Newton's method is why* `bsformula` *returns vega).* `method='bisect'` *forces use of your* `bisect` *function from* `Bisect.py`*.*

This function usually just returns the volatility for an option given its price within the given price tolerance. If no volatility can be found (because an input is `NaN` *or the option value is less than intrinsic) then it should return* `NaN`*. If* `reportCalls` *is* `True` *then the function must return a 2-tuple consisting of the volatility found (or* `NaN` *if applicable), and the number of times it made calls to the function* `bsformula`*.*

Compare the convergence properties of Newton's method against the bisection method for finding the implied volatility. For which kinds of European options (i,e. ATM, ITM, OTM) is the performance difference more noticeable?