GROUP PROJECT SUBMISSION 3

Group Members

Theophilus Addae
Williams Botchway
Moses E. Tommie
Lucia Sarkwa

INTRODUCTION

In this peer review, we will try to simulate the evolution of an asset price and price a simple (i.e. a risky up and out call option) option using python. In order to price this knock out option with non-constant interest rate and local volatility, we use the LIBOR forward rate model to simulate interest rates. The initial values for the LIBOR forward rates are then calibrated to the market forward rates which are deduced through a given zero-coupon bond market prices. First we present a definitive view of the approach used in this project and all its assumptions.

OUR MODEL AND ITS ASSUMPTIONS

- Option maturity is one year
- The option is struck at-the-money
- The up-and-out barrier for the option is \$150
- The current share price is \$100
- The current firm value for the counterparty is \$200
- The counterparty's debt, due in one year, is \$175
- The correlation between the counterparty and the stock is constant at 0.2
- The recovery rate with the counterparty is 25%.

The local volatility function for the stock is given by

$$\sigma_s(t_i,\ t_{i+1}) = 0.3(S_{ti})^{\gamma-1}. \qquad (1)$$

The local volatility function for the counterparty is also given by

$$\sigma_v(t_i,\ t_{i+1}) = 0.3(V_{ti})^{\gamma-1}\text{ , where } \gamma = 0.75. \qquad (2)$$

The share price path is modeled by the formula;

$$S_{t_{i+1}} = S_{t_i} e^{\left(r-\frac{\sigma^2(t_i,t_{i+1})}{2}\right)(t_{i+1}-t_i) + \sigma(t_i,t_{i+1})\sqrt{t_{i+1}-t_i}Z} \qquad (3)$$

Where S_{t_i} is the share price at time t_i, $\sigma(t_i, t_{i+1})$ is the volatility for the period $[t_i, t_{i+1}]$, r_{t_i} is the risk-free interest rate, and *Z~N* (0, 1). The counterparty firm values also follow the same dynamics.

We observe the following zero-coupon bond prices (per \$100 nominal) in the market.

Maturity	Price
1 month	\$99.38

2 months	$98.76
3 months	$98.15
4 months	$97.54
5 months	$96.94
6 months	$96.34
7 months	$95.74
8 months	$95.16
9 months	$94.57
10 months	$93.99
11 months	$93.42
12 months	$92.85

USING MARKET DATA TO CALIBRATE VASICEK MODEL TO FIND OPTIMAL VALUES FOR r, alpha, b, and sigma

Here we use the given zero coupon bond market data to implied values for r, alpha, b, and sigma as follows.

```
In [1]:
#Importing Libraries
import numpy as np
from scipy.stats import norm
import scipy.optimize
import matplotlib.pyplot as plt
```

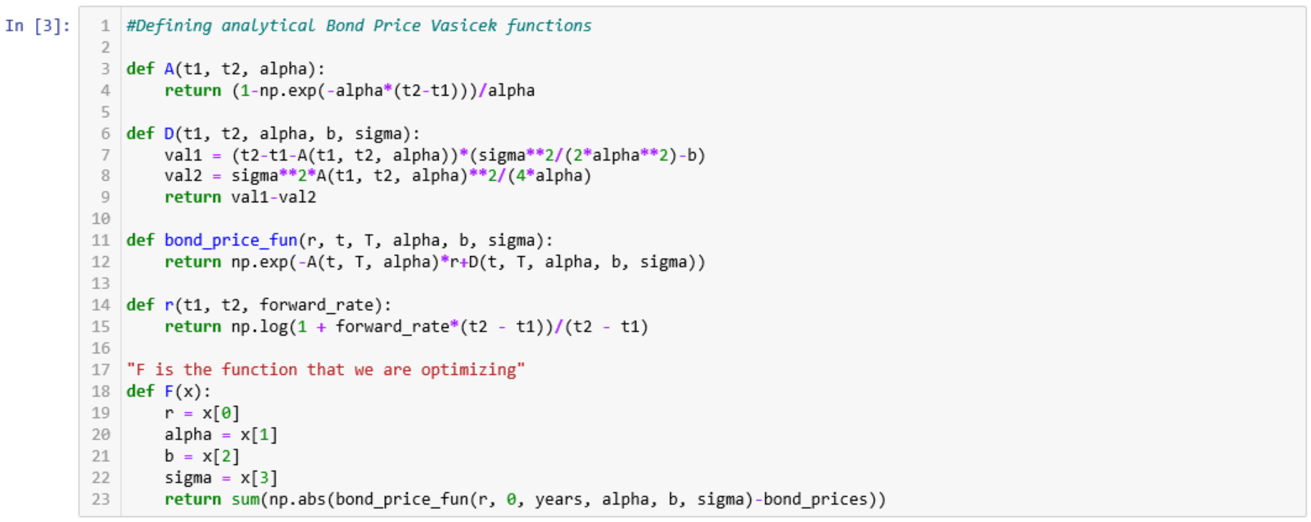
The above imports relevant libraries needed for implying values for r, alpha, b, and sigma using the Vasicek model which was proposed by Vasicek (1977). Under this model the zero coupon bond price is given by

$$B(t,T) = e^{-A(t,T)r_t + D(t,T)} \tag{4}$$

Where $A(t,T) = \frac{1- e^{-\alpha(T-t)}}{\alpha}$ and $D(t,T) = \left(b - \frac{\sigma^2}{2\alpha^2}\right)[A(t,T) - (T-t)] - \frac{\sigma^2 A(t,T)^2}{4\alpha}$

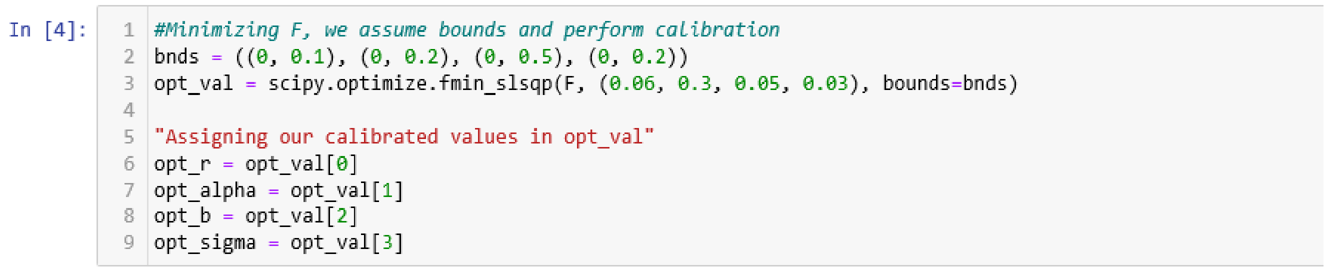
```
In [2]:
#Using zero coupon bond price from the given table
years = np.linspace(0, 1, 13)
delta_t = 1/12
"Bond prices deduced from market, starting one month apart, upto one year maturity"
bond_prices = np.array([1.00, 0.9938, 0.9876, 0.9815, 0.9754, 0.9694, 0.9634, 0.9574, 0.9516, 0.9457, 0.9399,
                        0.9342, 0.9285 ])
```

The above specify the bond prices transformed into a normal distribution across the interval [0,1]



```
In [3]:
 1 #Defining analytical Bond Price Vasicek functions
 2 
 3 def A(t1, t2, alpha):
 4     return (1-np.exp(-alpha*(t2-t1)))/alpha
 5 
 6 def D(t1, t2, alpha, b, sigma):
 7     val1 = (t2-t1-A(t1, t2, alpha))*(sigma**2/(2*alpha**2)-b)
 8     val2 = sigma**2*A(t1, t2, alpha)**2/(4*alpha)
 9     return val1-val2
10 
11 def bond_price_fun(r, t, T, alpha, b, sigma):
12     return np.exp(-A(t, T, alpha)*r+D(t, T, alpha, b, sigma))
13 
14 def r(t1, t2, forward_rate):
15     return np.log(1 + forward_rate*(t2 - t1))/(t2 - t1)
16 
17 "F is the function that we are optimizing"
18 def F(x):
19     r = x[0]
20     alpha = x[1]
21     b = x[2]
22     sigma = x[3]
23     return sum(np.abs(bond_price_fun(r, 0, years, alpha, b, sigma)-bond_prices))
```

The functions A, D, and `bond_price_fun` calculate model implied prices for given α, b, and σ parameters (as well as time parameters, but we are not optimizing over these). Line 11 above calculates the bond price derived from the Vasicek model as shown in equation (4). F takes in a vector x, where the first element of x is r, the second is α, and the third is b and the fourth is σ. It returns the sum of the absolute differences between the model implied bond prices, and the implied bond prices from the table above.

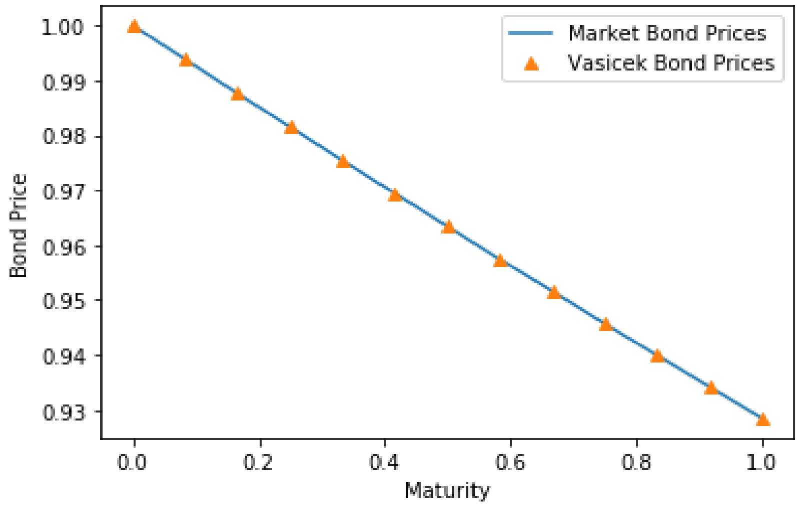


```
In [4]:
1 #Minimizing F, we assume bounds and perform calibration
2 bnds = ((0, 0.1), (0, 0.2), (0, 0.5), (0, 0.2))
3 opt_val = scipy.optimize.fmin_slsqp(F, (0.06, 0.3, 0.05, 0.03), bounds=bnds)
4 
5 "Assigning our calibrated values in opt_val"
6 opt_r = opt_val[0]
7 opt_alpha = opt_val[1]
8 opt_b = opt_val[2]
9 opt_sigma = opt_val[3]
```

The variable bounds in line 2 define our lower and upper bounds. This means our bounds are given by : $0 \leq r \leq 0.1$, $0 \leq \alpha \leq 0.2$, $0 \leq b \leq 0.5$,$0 \leq \sigma \leq 0.2$. The lower bounds are all 0, since negative values would not make sense for these variables. The upper bounds are based on realistic expectations of how large these variables could be. As an inputs to the function `scipy.optimize.fmin_slsqp`, we provided the function to be minimized (F), an initial guess (r$= 0.06$, $\alpha= 0.3$, $b = 0.05$, $\sigma= 0.03$), and the bounds for these variables being optimized over. In lines 6, 7, 8, and 9, we extract the r, α, b, and σ values from the optimized vector.

```python
1 #Calculating our Vasicek Bond prices
2 model_prices = bond_price_fun(opt_r, 0, years, opt_alpha, opt_b, opt_sigma)
3 #Plotting to show quality of fit
4 plt.xlabel("Maturity")
5 plt.ylabel("Bond Price")
6 plt.plot(years, bond_prices, label = "Market Bond Prices")
7 plt.plot(years, model_prices, '^', label = "Vasicek Bond Prices")
8 plt.legend()
9 plt.show()
```

In line 2 we find the model implied bond prices with the optimized parameters. We then plot these model implied values versus the market values given by the table above. We plot these model implied values versus the market values.

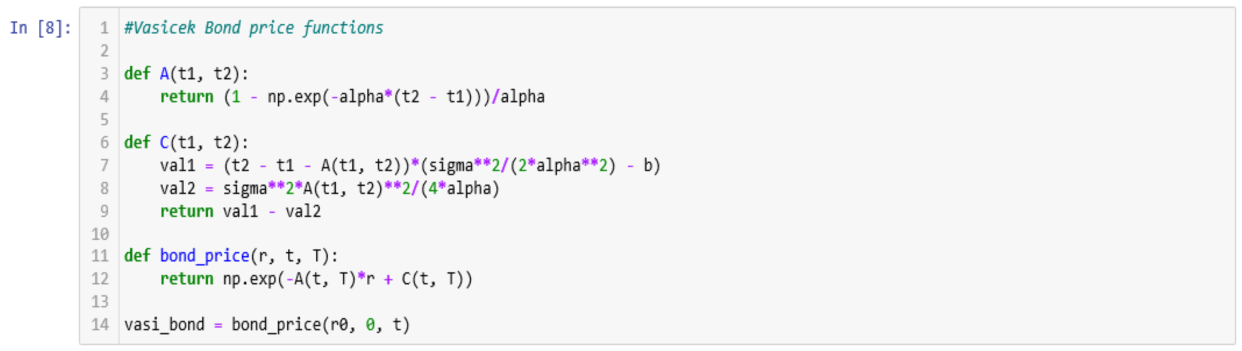


1. USING A SAMPLE SIZE OF 100000 TO SIMULATE LIBOR FORWARD RATES, STOCK PATHS AND COUNTERPARTY FIRM VALUES

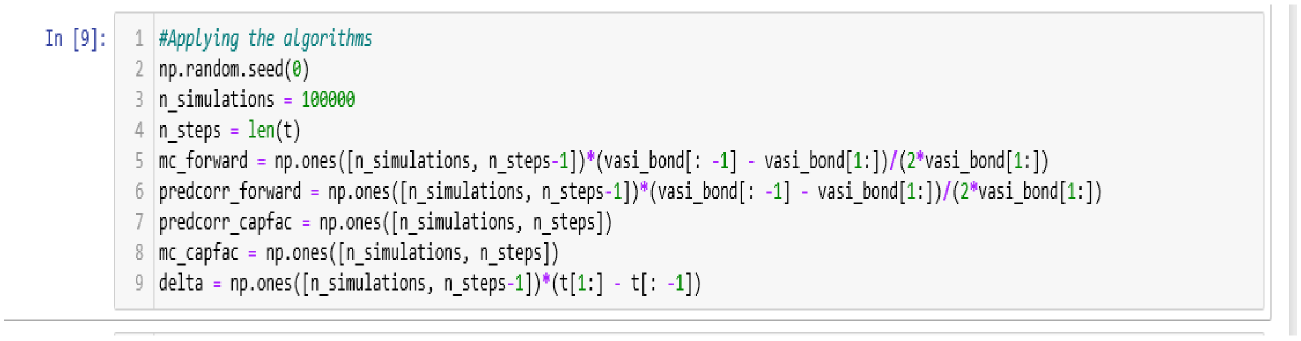
We now simulate the uncorrelated forward rates with the calibrated r, alpha, b, and sigma using monthly market data.

```python
In [7]:
#Assigning calibrated parameters
r0 = opt_r
alpha = opt_alpha
b = opt_b
sigma = opt_sigma
T = 1
t = np.linspace(0, 1, 13)
sigmaj = 0.2
```

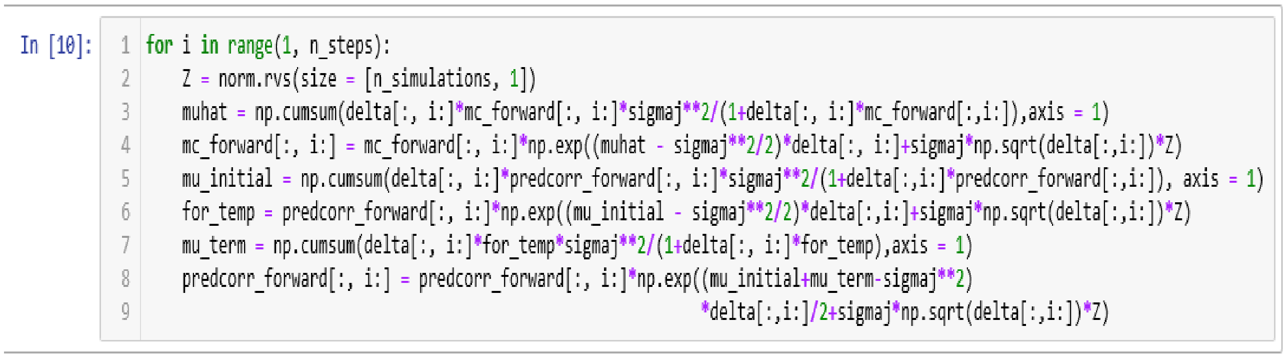
Here we set parameters with the calibrated r, alpha, b, and sigma for a period of one year.


In [8]:
```python
#Vasicek Bond price functions

def A(t1, t2):
    return (1 - np.exp(-alpha*(t2 - t1)))/alpha

def C(t1, t2):
    val1 = (t2 - t1 - A(t1, t2))*(sigma**2/(2*alpha**2) - b)
    val2 = sigma**2*A(t1, t2)**2/(4*alpha)
    return val1 - val2

def bond_price(r, t, T):
    return np.exp(-A(t, T)*r + C(t, T))

vasi_bond = bond_price(r0, 0, t)
```

We generate our own bond prices using Vasicek dynamics. The portion of code above generates synthetic bond prices that can be use for our simulations.

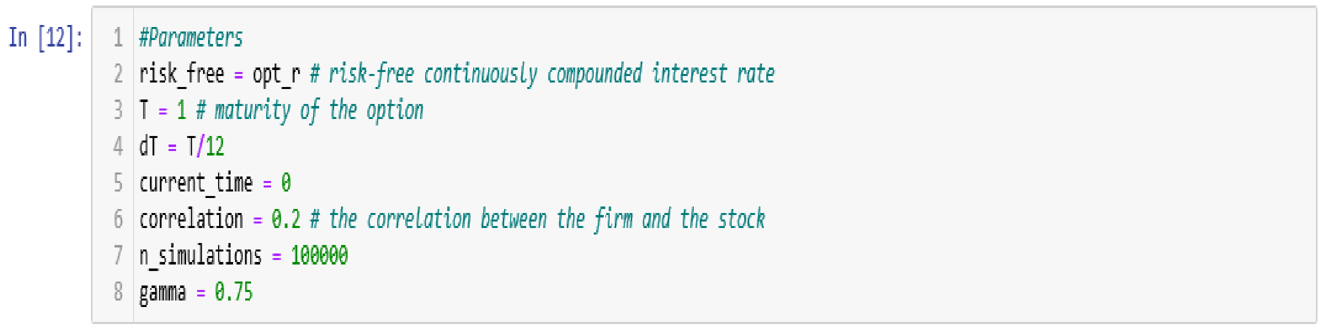

In [9]:
```python
#Applying the algorithms
np.random.seed(0)
n_simulations = 100000
n_steps = len(t)
mc_forward = np.ones([n_simulations, n_steps-1])*(vasi_bond[: -1] - vasi_bond[1:])/(2*vasi_bond[1:])
predcorr_forward = np.ones([n_simulations, n_steps-1])*(vasi_bond[: -1] - vasi_bond[1:])/(2*vasi_bond[1:])
predcorr_capfac = np.ones([n_simulations, n_steps])
mc_capfac = np.ones([n_simulations, n_steps])
delta = np.ones([n_simulations, n_steps-1])*(t[1:] - t[: -1])
```

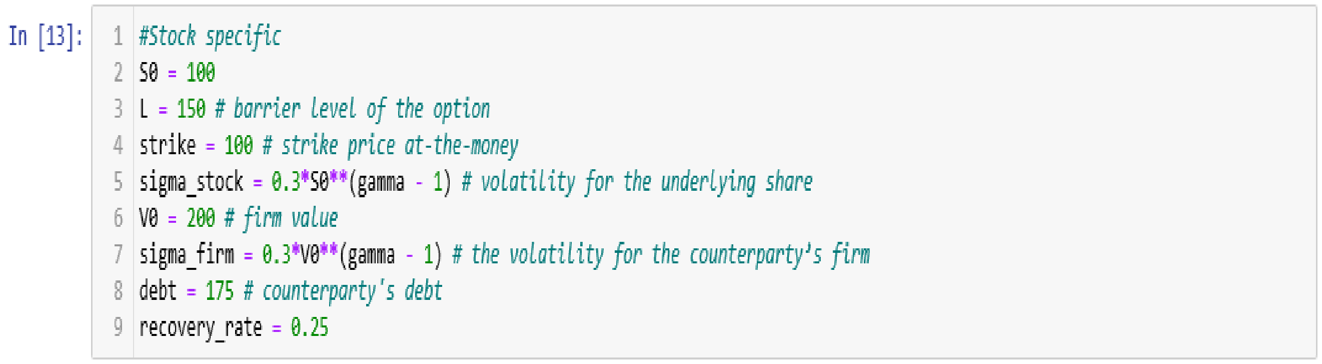
Now we can get to the actual simulation. The portion of the code above sets the seed and creates a few new parameters. The n simulations variable is the number of Monte Carlo simulations we are going to do at each time point. The n steps variable stores the number of time steps we are simulating over. Finally, we pre-allocate space for our for-ward rates under both methods (i.e., using Approximation Method and the Predictor-Corrector method), as well as the final capitalization factors.


In [10]:
```python
for i in range(1, n_steps):
    Z = norm.rvs(size = [n_simulations, 1])
    muhat = np.cumsum(delta[:, i:]*mc_forward[:, i:]*sigmaj**2/(1+delta[:, i:]*mc_forward[:,i:]),axis = 1)
    mc_forward[:, i:] = mc_forward[:, i:]*np.exp((muhat - sigmaj**2/2)*delta[:, i:]+sigmaj*np.sqrt(delta[:,i:])*Z)
    mu_initial = np.cumsum(delta[:, i:]*predcorr_forward[:, i:]*sigmaj**2/(1+delta[:,i:]*predcorr_forward[:,i:]), axis = 1)
    for_temp = predcorr_forward[:, i:]*np.exp((mu_initial - sigmaj**2/2)*delta[:,i:]+sigmaj*np.sqrt(delta[:,i:])*Z)
    mu_term = np.cumsum(delta[:, i:]*for_temp*sigmaj**2/(1+delta[:, i:]*for_temp),axis = 1)
    predcorr_forward[:, i:] = predcorr_forward[:, i:]*np.exp((mu_initial+mu_term-sigmaj**2)
                                                  *delta[:,i:]/2+sigmaj*np.sqrt(delta[:,i:])*Z)
```

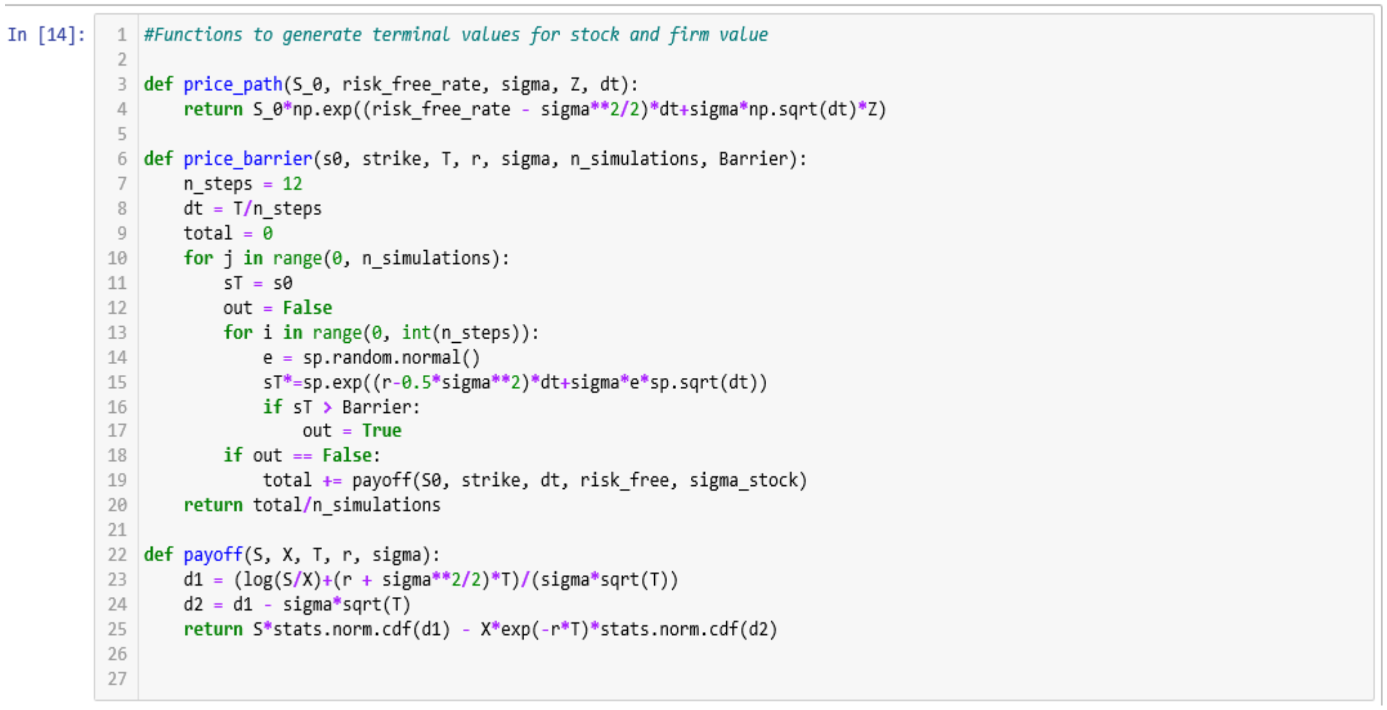
The above portion of the code simulates the forward rates.

1B. JOINTLY SIMULATING CORRELATED STOCK PATH AND COUNTERPARTY FIRM VALUES


In [12]:
```python
#Parameters
risk_free = opt_r # risk-free continuously compounded interest rate
T = 1 # maturity of the option
dT = T/12
current_time = 0
correlation = 0.2 # the correlation between the firm and the stock
n_simulations = 100000
gamma = 0.75
```


In [13]:
```python
#Stock specific
S0 = 100
L = 150 # barrier level of the option
strike = 100 # strike price at-the-money
sigma_stock = 0.3*S0**(gamma - 1) # volatility for the underlying share
V0 = 200 # firm value
sigma_firm = 0.3*V0**(gamma - 1) # the volatility for the counterparty's firm
debt = 175 # counterparty's debt
recovery_rate = 0.25
```

The above portion of the code sets the parameters and stock specific values for simulation.


In [14]:
```python
#Functions to generate terminal values for stock and firm value

def price_path(S_0, risk_free_rate, sigma, Z, dt):
    return S_0*np.exp((risk_free_rate - sigma**2/2)*dt+sigma*np.sqrt(dt)*Z)

def price_barrier(s0, strike, T, r, sigma, n_simulations, Barrier):
    n_steps = 12
    dt = T/n_steps
    total = 0
    for j in range(0, n_simulations):
        sT = s0
        out = False
        for i in range(0, int(n_steps)):
            e = sp.random.normal()
            sT*=sp.exp((r-0.5*sigma**2)*dt+sigma*e*sp.sqrt(dt))
            if sT > Barrier:
                out = True
        if out == False:
            total += payoff(S0, strike, dt, risk_free, sigma_stock)
    return total/n_simulations

def payoff(S, X, T, r, sigma):
    d1 = (log(S/X)+(r + sigma**2/2)*T)/(sigma*sqrt(T))
    d2 = d1 - sigma*sqrt(T)
    return S*stats.norm.cdf(d1) - X*exp(-r*T)*stats.norm.cdf(d2)


```

The function price_path simulate the path of the stock and the function price_barrier calculates the price of the barrier option. The payoff function calculates the payoff of the option.

In [15]:
```
#Using sample sizes of 100000 to simulate paths
corr_matrix = np.array([[1, correlation], [correlation,1]])
norm_matrix = norm.rvs(size = np.array([2, n_simulations]))
corr_norm_matrix = np.matmul(np.linalg.cholesky(corr_matrix), norm_matrix)
"Simulating paths for the underlying stock and counter party firm"
term_stock_val = price_path(S0, risk_free, sigma_firm, corr_norm_matrix[0,], dT)
term_firm_val = price_path(V0, risk_free, sigma_firm, corr_norm_matrix[1,], dT)
```

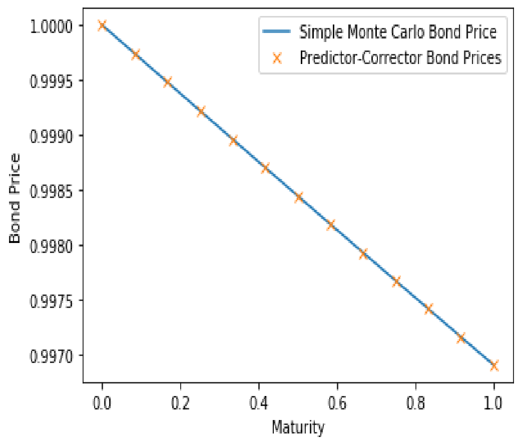
The above code simulates paths for the underlying stock and counter party firm

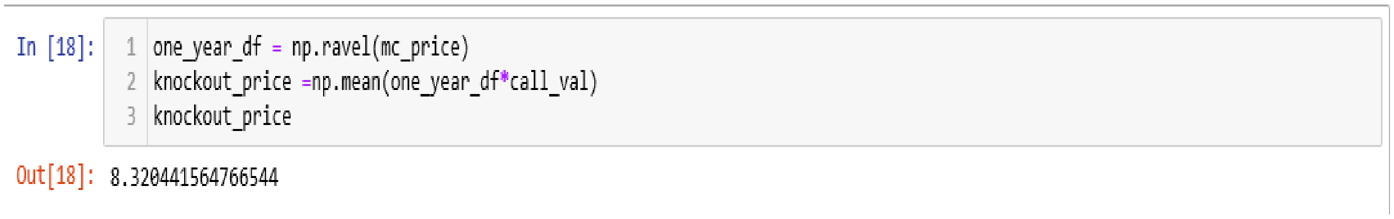
2. REPRISING KNOCKOUT OPTION USING ONE YEAR DISCOUNT FACTOR AND MONTE CARLO TECHNIQUE

In [16]:
```
"Using Monte Carlo pricing technique to estimate the payoff for the barrier call"
call_val = payoff(S0, strike, T, risk_free, sigma_stock)
mc_capfac[:, 1:] = np.cumprod(1+delta*mc_forward, axis = 1)
predcorr_capfac[:, 1:] = np.cumprod(1+delta*predcorr_forward, axis = 1)
"discount factors for the simulated paths"
mc_price = mc_capfac**(-1)
predcorr_price = predcorr_capfac**(-1)
mc_final = np.mean(mc_price, axis = 0)
predcorr_final = np.mean(predcorr_price, axis = 0)
```

Now we use the simulated forward rates to imply capitalisation factors. This can be done using the following equation: $C(t_0, t_n) = \prod_{k=1}^{n} \left(1 + \delta_k F_k(t_k)\right)$

In [17]:
```
plt.xlabel("Maturity")
plt.ylabel("Bond Price")
plt.plot(t, mc_final,  label = "Simple Monte Carlo Bond Price")
plt.plot(t, predcorr_final, 'x', label = "Predictor-Corrector Bond Prices")
plt.legend()
plt.show()
```

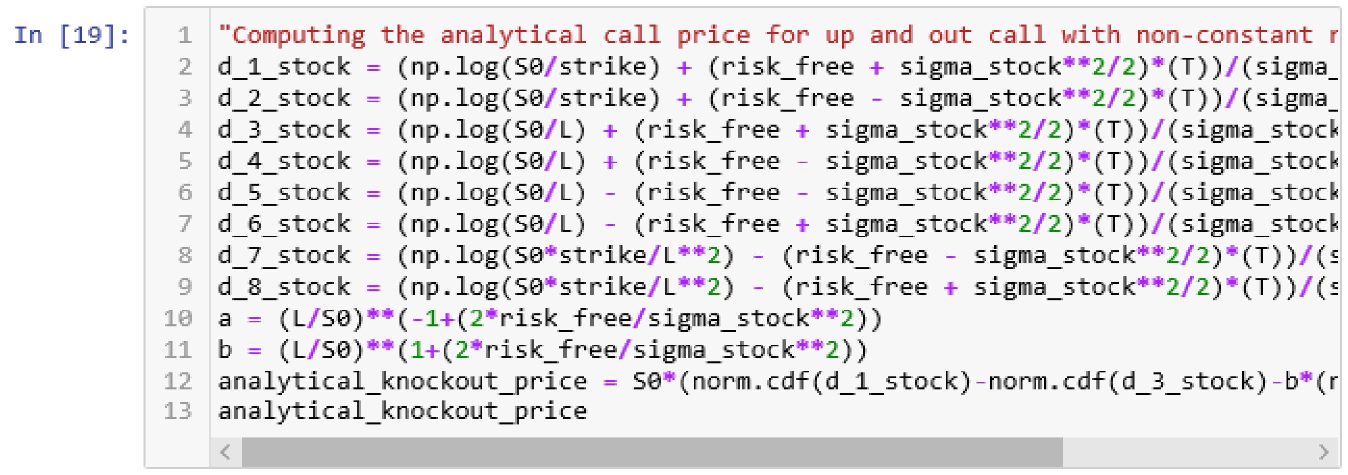




```
In [18]: one_year_df = np.ravel(mc_price)
         knockout_price =np.mean(one_year_df*call_val)
         knockout_price

Out[18]: 8.320441564766544
```

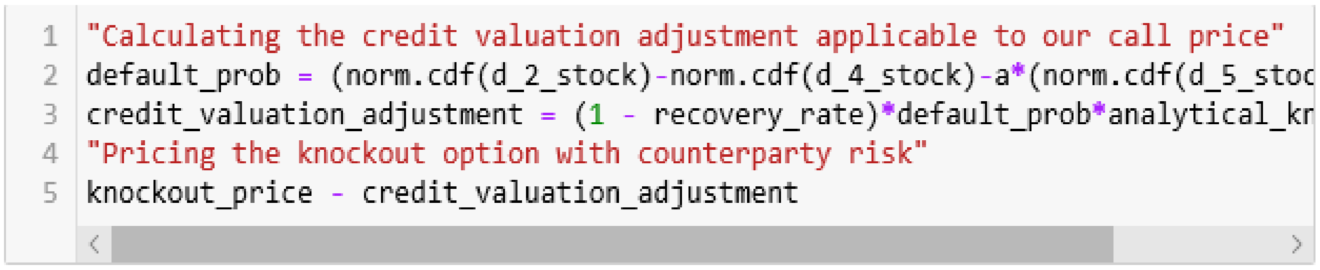
The above calculate the knockou price of the option.

2B. VALUE OF KNOCKOUT OPTION WITH COUNTERPARTY DEFAULT RISK



```
In [19]: "Computing the analytical call price for up and out call with non-constant r
         d_1_stock = (np.log(S0/strike) + (risk_free + sigma_stock**2/2)*(T))/(sigma_
         d_2_stock = (np.log(S0/strike) + (risk_free - sigma_stock**2/2)*(T))/(sigma_
         d_3_stock = (np.log(S0/L) + (risk_free + sigma_stock**2/2)*(T))/(sigma_stock
         d_4_stock = (np.log(S0/L) + (risk_free - sigma_stock**2/2)*(T))/(sigma_stock
         d_5_stock = (np.log(S0/L) - (risk_free - sigma_stock**2/2)*(T))/(sigma_stock
         d_6_stock = (np.log(S0/L) - (risk_free + sigma_stock**2/2)*(T))/(sigma_stock
         d_7_stock = (np.log(S0*strike/L**2) - (risk_free - sigma_stock**2/2)*(T))/(s
         d_8_stock = (np.log(S0*strike/L**2) - (risk_free + sigma_stock**2/2)*(T))/(s
         a = (L/S0)**(-1+(2*risk_free/sigma_stock**2))
         b = (L/S0)**(1+(2*risk_free/sigma_stock**2))
         analytical_knockout_price = S0*(norm.cdf(d_1_stock)-norm.cdf(d_3_stock)-b*(r
         analytical_knockout_price
```

```
8.316543614112419
```



```
"Calculating the credit valuation adjustment applicable to our call price"
default_prob = (norm.cdf(d_2_stock)-norm.cdf(d_4_stock)-a*(norm.cdf(d_5_stoc
credit_valuation_adjustment = (1 - recovery_rate)*default_prob*analytical_kr
"Pricing the knockout option with counterparty risk"
knockout_price - credit_valuation_adjustment
```

```
3.514060796390269
```

CONCLUSION

Using a sample size of 100000 we jointly simulate LIBOR forward rates, stock paths, and counterparty firm values using monthly values. We assume that the counterparty firm and stock values are uncorrelated with LIBOR forward rates. We then calculate the one-year discount factor which applies for each simulation, and use this to find first the value of the option for the jointly simulated stock and firm paths with no default risk, and then the value of the option with counterparty default risk. The figure generated is the result which shows the model prices as the yellow points, with the blue line showing the market prices. You can see that, the model does perfectly match the market. It means the model has been able to capture almost all intricacies present in the market. The error margin is definitely low with this model. This can be attributed to the fact that the number of simulation was huge, 100000, hence that low error bounds. If you are calibrating a model to price a certain portfolio or derivative, and you know the maturity time of that portfolio, you might only use market values with similar maturity values as your derivative to calibrate your model.

REFERENCES

Hilpisch, Y. (2015). Derivatives analytics with Python: data analysis, models, simulation, calibration and hedging. John Wiley & Sons.

Mamon, R. S. (2004). Three ways to solve for bond prices in the vasicek model, Advances in Decision Sciences 8(1): 1–14.

Vasicek, O. (1977). An equilibrium characterization of the term structure, Journal of financial economics 5(2): 177–188.