Simulate Asset Price Evolutions and Reprice Risky up-and-out Call Option

Ng, Joe Hoong
ng_joehoong@hotmail.com

Nguyen, Dang Duy Nghia
nghia002@e.ntu.edu.sg

Ansari, Zain Us Sami Ahmed
zainussami@gmail.com

Thorne, Dylan
dylan.thorne@gmail.com

May. 10, 2020

Keywords: European Options, Non-constant interest rate, Local volatility, LIBOR forward rates, Zero-coupon bond and Stochastic Volatility.

Abstract

This report presents the results of the simulation of a European up-and-out call option over twelve months, but the difference between this submission and submission 1 is the implementation of a non-constant interest rate and local volatility. We use LIBOR forward rate model to simulate interest rates. We also use discount factor to value the option without default risk and then the value of the option with counterparty default risk.

1 Introduction

In this paper, we go beyond the constant risk-free continuously-compounded rate. There are several models to implement stochastic interest rates. The short rate models which describe instantaneous continuously-compounded interest rate at time t, r_t include the Vasicek model introduced in [1], Mamon in [2] advances this model by presenting approaches in obtaining the closed-form solution using the Vasicek model. Under the Vasicek model the short rate dynamics is given by,

$$dr_t = \alpha(b - r_t)dt + \sigma dW_t$$

Where W_t is Brownian motion, b is the level to which the short rate will tend in the long run, α is the rate at which the short rate will tend towards b and σ is the volatility of the short rate. The stochastic differential equation can be solved to give r_t

$$r_t = e^{-\alpha t}[r_0 + b(e^{\alpha t} - 1) + \int_0^t \sigma e^{\alpha s} dW_s]$$

Cox, Ingersoll and Ross also presented a stochastic differential equation for short term rates in [4] given by

$$dr_t = \alpha(b - r_t)dt + \sigma\sqrt{r_t}dW_t$$

This model prevents the short from becoming 0 but offers no closed form solution

The Hull-White model described in [3] also presents stochastic differential equation for the short term rate,

$$dr_t = (\theta(t) - \alpha(t)r_t)dt + \sigma(t)dW_t$$

where $\theta(t) = \alpha b$ which is a non constant term, allowing the mean reversion level to vary.

The model we implement in this assignment is the LIBOR forward rate model to simulate interest rates. The initial values for the LIBOR forward rates need to be calibrated to the market forward rates which can be deduced through the market zero-coupon bond prices. This continuously compounded interest rate is given by,

$$e^{r_{ti}(t_{i+1}-t_i)} = 1 + L(t_i, t_{i+1})(t_{i+1} - t_i)$$

Most of code implemented in this submission is derived from Module 6 [12] and Module 7 [13] of the course.

We initialize most variables as given by the question.

- Option maturity is one year
- The option is struck at-the-money
- The current share price is $100
- The up-and-out barrier for the option is $150
- The risk-free continuously compounded interest rate is 8%
- The volatility for the underlying share is 30%
- The volatility for counterparty's firm value is 25%
- The counterparty's debt, due in one year, is $175
- The current firm value for the counterparty is $200
- The correlation between the counterparty and the stock is constant at 0.2
- The recovery rate with the counterparty is 25%

2 LIBOR Forward Rates, Stock Paths, and Counterparty Firm Values

In this part, we use a sample size of 100000, jointly simulate LIBOR forward rates, stock paths, and counterparty firm values. We assume that the counterparty firm and stock values are uncorrelated with LIBOR forward rates.

2.1 Calibrate LIBOR forward rate model from zero coupon bond prices

We initialize the given zero-coupon bond prices:

```
[4]: t = np.linspace(0,1,13)

     market_zcb_prices = np.array([1.0, 0.9938, 0.9876, 0.9815, 0.9754, 0.9694, 0.
      ↪9634, 0.9574, 0.9516,
            0.9457, 0.9399, 0.9342, 0.9285])
```

We next create functions to calculate the simulated bond prices from the Vasicek model (as well as helper functions A and D). We also define function F which is the differences between the bond prices calculated by our model and actual market zero-coupon bond prices:

```
[5]: def A(t1, t2, alpha):
         return (1-np.exp(-alpha*(t2-t1)))/alpha
     def D(t1, t2, alpha, b, sigma):
         val1 = (t2-t1-A(t1,t2,alpha))*(sigma**2/(2*alpha**2)-b)
         val2 = sigma**2*A(t1,t2,alpha)**2/(4*alpha)
         return val1-val2

     def bond_price_fun(r,t,T, alpha, b, sigma):
         return np.exp(-A(t,T,alpha)*r+D(t,T,alpha,b,sigma))



     def F(x):
         alpha = x[0]
         b = x[1]
         sigma = x[2]
         r0 = x[3]
         return sum(np.abs(bond_price_fun(r0,0,t,alpha,b,sigma)-market_zcb_prices))
```

We use the fmin_slsqp function from scipy to calculate the optimal model parameters, with a minimum value of F

```
[6]: #minimizing F
     bnds = ((0,1),(0,0.2),(0,0.2), (0.00,0.10))
     opt_val = scipy.optimize.fmin_slsqp(F, (0.3, 0.05, 0.03, 0.05), bounds=bnds)
     opt_alpha = opt_val[0]
     opt_b = opt_val[1]
     opt_sigma = opt_val[2]
     opt_r0 = opt_val[3]
```

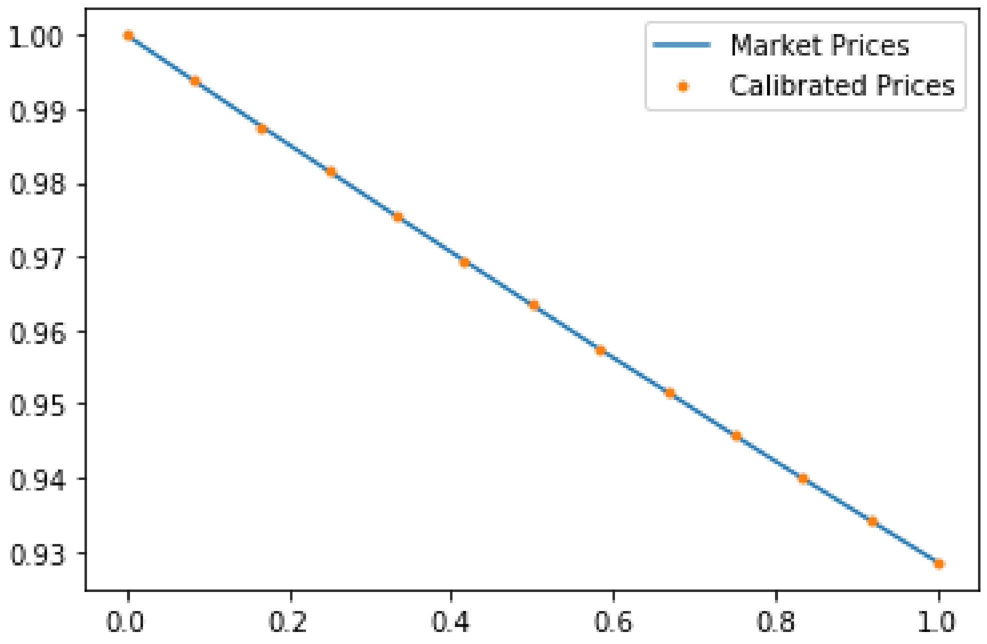
```
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 0.00025649918134296l8
            Iterations: 10
            Function evaluations: 74
            Gradient evaluations: 10
```

[7]:
```
print("Optimal alpha: {:.3f}".format(opt_val[0]))
print("Optimal b: {:.3f}".format(opt_val[1]))
print("Optimal sigma {:.3f}".format(opt_val[2]))
print("Optimal r0: {:.3f}".format(opt_val[3]))
```

```
Optimal alpha: 0.273
Optimal b: 0.069
Optimal sigma 0.028
Optimal r0: 0.075
```

We plot the actual market bond prices, with model-derived bond prices, and they look like a close fit.

[8]:
```
model_prices = bond_price_fun(opt_r0,0,t, opt_alpha, opt_b, opt_sigma)
model_yield = -np.log(model_prices)/t

plt.plot(t,market_zcb_prices, label='Market Prices')
plt.plot(t, model_prices, '.', label='Calibrated Prices')
plt.legend()
plt.show()
```



2.2 Simulate LIBOR rate paths

We first initialize the parameter σ_j

[9]:
```
sigmaj = 0.2
```

We use paramters we obtained above to recreate the Vasicek bond prices:

[10]:
```
def A(t1, t2):
    return (1-np.exp(-opt_alpha*(t2-t1)))/opt_alpha

def C(t1, t2):
    val1 = (t2-t1-A(t1,t2))*(opt_sigma**2/(2*opt_alpha**2)-opt_b)
    val2 = opt_sigma**2*A(t1,t2)**2/(4*opt_alpha)
    return val1 - val2

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

vasi_bond = bond_price(opt_r0, 0, t)
```

The prices calculated from the Vasicek model are close to the ZCB prices given by the assignment:

[11]:
```
print(vasi_bond)
```

```
[1.         0.99377572 0.98760087 0.98147516 0.97539831 0.96936998
 0.96338983 0.95745752 0.95157266 0.94573489 0.93994381 0.93419901
 0.9285001 ]
```

We now initialize the matrices we will use to store the Monte Carlo simulations, for both basic Monte Carlo and Predictor-Corrector method.

[12]:
```
n_simulations = sample_size
n_steps = len(t)

mc_forward = np.ones([n_simulations, n_steps-1])*(vasi_bond[:-1]-vasi_bond[1:])/
 ↪(vasi_bond[1:])
predcorr_forward = np.ones([n_simulations, n_steps-1])*(vasi_bond[:
 ↪-1]-vasi_bond[1:])/(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])
```

We now run the Monte Carlo simulation for each time step:

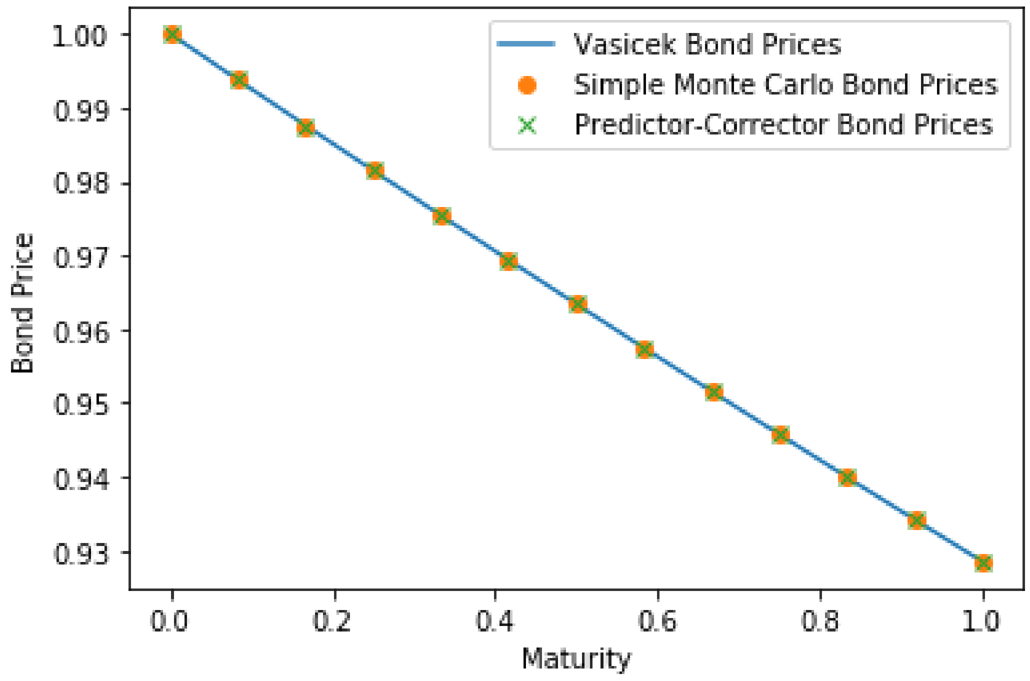
[13]:
```
for i in range(1, n_steps):
    Z = norm.rvs(size=[n_simulations,1])

    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)
```

From our Monte Carlo simulation, we now calculate the capitalization factors and bond prices, and plot them to compare them with the Vasicek bond prices.

[14]:
```
mc_capfac[:,1:] = np.cumprod(1+mc_forward, axis=1)
predcorr_capfac[:,1:] = np.cumprod(1+predcorr_forward, axis=1)

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)
```

[15]:
```
plt.xlabel("Maturity")
plt.ylabel("Bond Price")
plt.plot(t,vasi_bond, label="Vasicek Bond Prices")

plt.plot(t, mc_final, 'o', label="Simple Monte Carlo Bond Prices")
plt.plot(t, predcorr_final, 'x', label="Predictor-Corrector Bond Prices")
plt.legend()
plt.show()
```



From our simulation of forward rates (we take the Predictor-Corrector method, we use the formula $e^{r_{t_i}(t_{i+1}-t_i)} = 1 + L(t_i, t_{i+1})(t_{i+1} - t_i)$ to obtain the continuous compounded interest rates:

[16]:
```
r_sim = np.log(1 + predcorr_forward*(delta))/delta
```

We also calculate an annualized form of the interest rates:

[17]:
```
r_sim_annualized = pd.DataFrame(r_sim/delta)
```

[18]:
```
r_sim_annualized
```

[18]:
```
               0         1         2         3         4         5         6  \
0        0.07514  0.066032  0.073336  0.075253  0.065992  0.066157  0.066794
1        0.07514  0.072083  0.076464  0.081468  0.079424  0.082719  0.085049
2        0.07514  0.078051  0.079510  0.077283  0.080866  0.078631  0.085738
3        0.07514  0.067973  0.066249  0.068189  0.061341  0.061618  0.058422
4        0.07514  0.068828  0.067538  0.068650  0.064811  0.064151  0.067009
...          ...       ...       ...       ...       ...       ...       ...
999995   0.07514  0.080051  0.081460  0.080036  0.078720  0.085394  0.090455
999996   0.07514  0.075648  0.076774  0.074305  0.081762  0.077945  0.075159
999997   0.07514  0.080705  0.077654  0.076033  0.068839  0.066677  0.062133
999998   0.07514  0.085071  0.085031  0.082831  0.083125  0.087771  0.077675
999999   0.07514  0.079983  0.080474  0.078617  0.073621  0.071960  0.072492

                7         8         9        10        11
```

```
0        0.071269  0.076366  0.074026  0.080224  0.076966
1        0.085452  0.094079  0.106641  0.120261  0.116014
2        0.082785  0.072790  0.076300  0.066587  0.063343
3        0.061475  0.058621  0.057793  0.058849  0.054900
4        0.063756  0.062794  0.065755  0.066556  0.072365
...           ...       ...       ...       ...       ...
999995   0.091226  0.086187  0.086976  0.081274  0.075437
999996   0.079487  0.069895  0.067660  0.062574  0.059106
999997   0.059384  0.056089  0.055484  0.056012  0.054232
999998   0.075775  0.075541  0.073908  0.078261  0.073692
999999   0.068380  0.066112  0.071977  0.071854  0.065560

[1000000 rows x 12 columns]
```

2.3 Generate stock and firm values

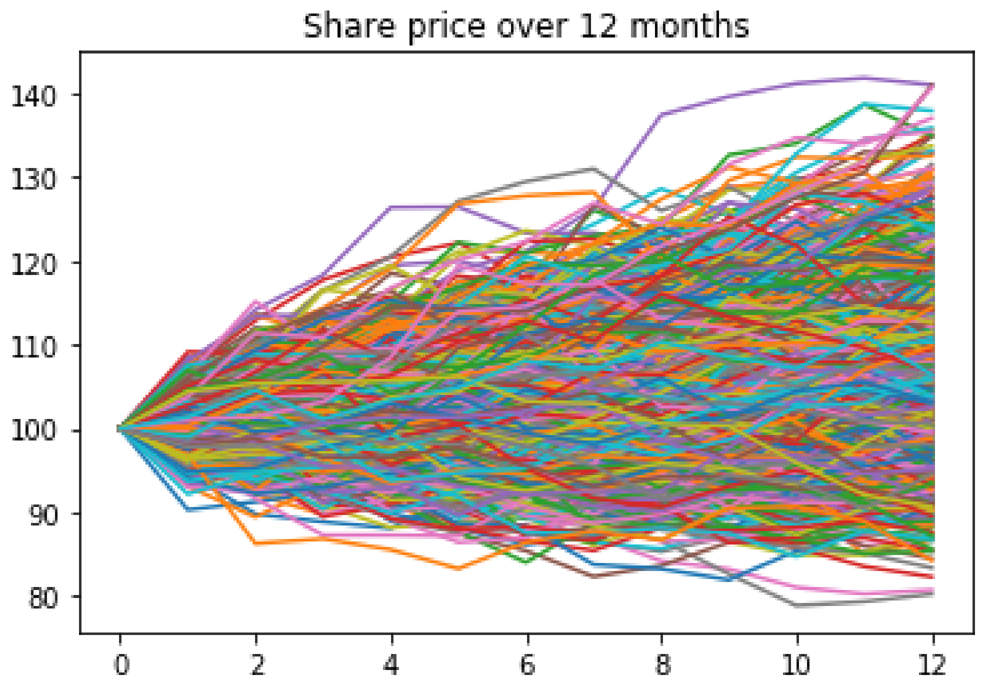
Similar to the first groupwork assignment, we use a Cholesky decomposition to generate the correlated price paths

[19]:
```python
def next_share_price(prev_price, r, dT, sigma_const, gamma, sample_size, Z,␣
 ↪varying_vol = True):

    if varying_vol:
        sigma = sigma_const*(prev_price)**(gamma-1)
    else:
        sigma = sigma_const*(S0)**(gamma-1)

    return prev_price*np.exp(np.cumsum((r-(sigma**2)/2)*(dT)+(sigma)*(np.
 ↪sqrt(dT))*Z,1))

def generate_share_and_firm_price(S0, v_0, r_sim, sigma_const, gamma, corr, T,␣
 ↪sample_size, timesteps = 12):
    corr_matrix = np.array([[1, corr], [corr, 1]])
    norm_matrix = stats.norm.rvs(size = np.array([sample_size, 2, timesteps]))
    corr_norm_matrix = np.matmul(np.linalg.cholesky(corr_matrix), norm_matrix)


    share_price_path = pd.DataFrame(next_share_price(S0, r_sim, 1/timesteps,␣
 ↪sigma_const, gamma, sample_size, Z=corr_norm_matrix[:,0,]))
    share_price_path = share_price_path.transpose()

    first_row = pd.DataFrame([S0]*sample_size)
    first_row = first_row.transpose()
    share_price_path = pd.concat([first_row, share_price_path])
    share_price_path = share_price_path.reset_index(drop=True)
    firm_price_path = pd.DataFrame(next_share_price(v_0, r_sim, 1/timesteps,␣
↪sigma_const, gamma, sample_size, Z=corr_norm_matrix[:,1,]))
    firm_price_path = firm_price_path.transpose()

    first_row = pd.DataFrame([v_0]*sample_size)
    first_row = first_row.transpose()
    firm_price_path = pd.concat([first_row, firm_price_path])
    firm_price_path = firm_price_path.reset_index(drop=True)

    return [share_price_path,firm_price_path]
```

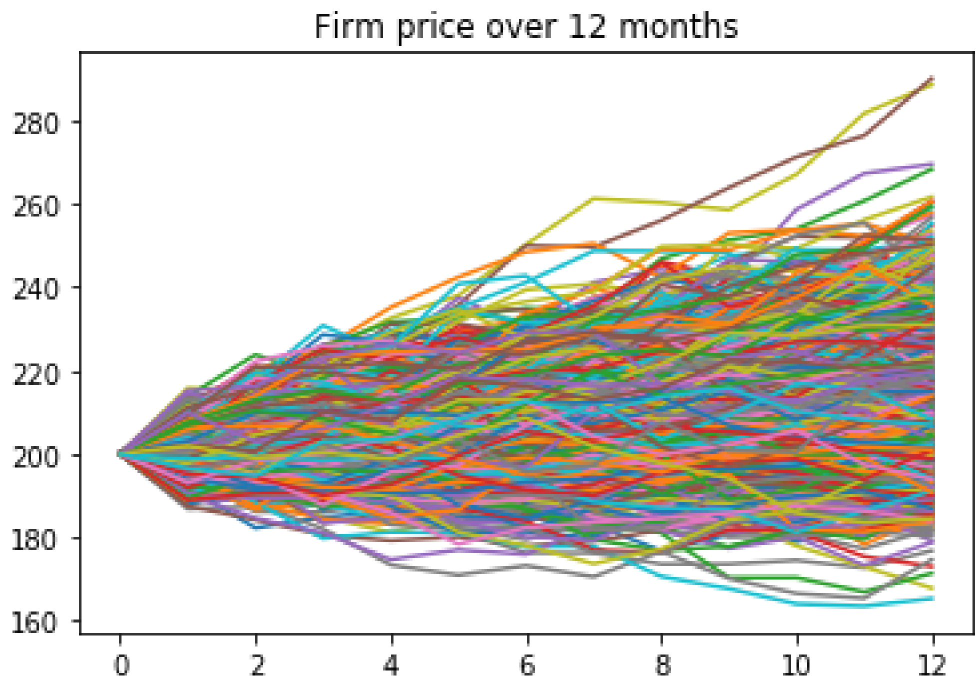
```
[20]: share_prices, firm_prices = generate_share_and_firm_price(S0, v_0,␣
↪r_sim_annualized, sigma_const, gamma, corr, T, sample_size, timesteps = 12)
```

We also plot the first 1000 stock price and firm value paths simulated:

```
[23]: share_prices.iloc[:,0:1000].plot(title='Share price over 12 months',␣
↪legend=False);
```



```
[24]: firm_prices.iloc[:,0:1000].plot(title='Firm price over 12 months', legend=False);
```



3 Discount Factor and Value of the Up-and-Out Call Option

In this section we use the capitalisation factor calculated in the section aboove to 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.

We first calculated the one year discount factor, by inverting the capitalisation factor. The capitalisation factor is calculated by taking the cumulative product of the interest rate between each timestep.

[25]:
```
one_year_disc_fac = 1/np.cumprod(1+r_sim,1)[:,-1]
one_year_disc_fac
```

[25]:
```
array([0.93045792, 0.91464088, 0.92665426, ..., 0.93658426, 0.92383486,
       0.92979433])
```

Next, we calculate the Default-Free Option Value. One difference is that we multiple the payoff by the one year discount factor, instead of multiplying with e^{rT}

[26]:
```
# define payoff for up-and-out call option
def payoff(S_t, K, L):
    stopped_S = S_t.iloc[-1].where((S_t < L).all(), 0)
    return np.maximum(stopped_S - K, 0).to_numpy()
```

[27]:
```
# Estimate the default-free value of the option:
option_estimate = []
option_std = []


payoffs = payoff(share_prices, K, L)
option_price = one_year_disc_fac*payoffs
option_estimate = option_price.mean()
option_std = option_price.std()/np.sqrt(sample_size)
```

[28]:
```
print("Default-free option price {:.3f}".format(option_estimate))
print("Default-free option price standard deviation {:.3f}".format(option_std))
```

```
Default-free option price 8.296
Default-free option price standard deviation 0.008
```

Next, we incorporate the CVA Adjustment similar to the first submission.

[29]:
```
payoffs = payoff(share_prices, K, L)
term_firm_vals = firm_prices.iloc[-1].to_numpy()
amount_lost = one_year_disc_fac*(1-recovery_rate)*(term_firm_vals < debt)*payoffs
cva_estimate = amount_lost.mean()
cva_std = amount_lost.std()/np.sqrt(sample_size)

option_cva_price = option_price - amount_lost
option_cva_adjusted_prices = option_cva_price.mean()
option_cva_adjusted_std = option_cva_price.std()/np.sqrt(sample_size)
```

[30]:
```
print("Credit value adjustment {:.3f}".format(cva_estimate))
print("Credit value adjustment standard deviation {:.3f}".format(cva_std))

print("CVA-adjusted option price {:.3f}".format(option_cva_adjusted_prices))
print("CVA-adjusted option price standard deviation {:.3f}".
 ↪format(option_cva_adjusted_std))
```

```
Credit value adjustment 0.018
Credit value adjustment standard deviation 0.000
CVA-adjusted option price 8.278
CVA-adjusted option price standard deviation 0.008
```

4 Conclusion

In this paper, we simulated correlated firm and share price paths. From this, we priced an up-and-out call option at 8.296 with a default-free risk profile, and at 8.278 for the CVA-adjusted price. Considering that the same parameters are used as the first submission, where the Black-Scholes-Merton model was used to arrive at a price of 5.697, we observe that both the default risk, the variable interest rate and local volatility have all added a premium to the option price. These conditions, which are more aligned to observed market conditions, gave a total increase of 45% over the previously calculated price, with the default risk accounting for only 0.2 of the difference. These results underline the importance of choosing the correct model and performing accurate calibration in order to calculate instrument prices.

References

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

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

[3] Hull, J. and White, A. (2001). The general hull-white model and supercalibration, Financial Analysts Journal pp. 34-43.

[4] Cox, J. C., Ingersoll Jr, J. E. and Ross, S. A. (1985). An intertemporal general equilibrium model of asset prices, Econometrica: Journal of the Econometric Society pp. 363-384.

[5] Merton, Robert C. "Theory of rational option pricing." The Bell Journal of economics and management science (1973): 141-183.

[6] Cox, John. "Notes on option pricing I: Constant elasticity of variance diffusions." Unpublished note, Stanford University, Graduate School of Business (1975).

[7] Rich, Don R. "The mathematical foundations of barrier option-pricing theory." Advances in futures and options research 7 (1994).

[8] Wong, Hoi Ying, and YueâĂŘKuen Kwok. "Multi-asset barrier options and occupation time derivatives." Applied Mathematical Finance 10.3 (2003): 245-266.

[9] Black, Fisher, and Myron Scholes. "The pricing and Corportate Liabilities." Journal of Political Economy 81 (1973).

[10] Yousuf, M. "A fourth-order smoothing scheme for pricing barrier options under stochastic volatility." International Journal of Computer Mathematics 86.6 (2009): 1054-1067.

[11] MScFE630 Computational Finance Module 5: Monte Carlo Methods for Risk Management

[12] MScFE630 Computational Finance Module 6: Pricing Interest Rate Options

[13] MScFE630 Computational Finance Module 7: Calibration