MA 573 Project

Alexander Shoop, Ming Min, and Yijiang (Chuck) Xu

1 INTRODUCTION

For our team project for the course *Computational Methods in Financial Mathematics*, we were first tasked to extract all call option data on IBM US stock over all time periods and maturities possible. After the data extraction, and after reducing the dataset of the options to only those with high liquidity (ie, volume traded > 10), then we were tasked to use the option data to calibrate the CEV (Constant Elasticity of Variance) model. Upon successful calibration, we were then asked to use the calibrated model and parameters to price discretely monitored barrier options (call options) using Monte-Carlo methods and monthly monitoring of the barrier. We then compared our estimated calculated option values with those retrieved from the Bloomberg Terminal computer.

2 METHODOLOGY

2.1 Data Extraction

In order to stay consistent with the result comparison of our option values, we used the Bloomberg Terminal computers to retrieve and extract the specified call option data on IBM US equity. From the Bloomberg Terminal home screen, we used the function OMON <GO> and then viewed the available 25 different strike call options for different expiration dates. We then used the built-in action of exporting the data to an Excel CSV file. After extraction, our team ended up with eight different Excel CSV files, each with the days to expiration, strikes, bid-ask costs, and volume traded. Our focus was only on options with high liquidity, therefore our Python code reads-in only those option values that have a volume traded greater than 10.

2.2 Model Calibration

Our designated model to calibrate was the CEV model [1]. We have for Brownian motion W_t, $\sigma > 0$, and $\beta \in [-1,0]$, the dynamics were given as:

$$dS_t = rS_t dt + \sigma S_t^{\beta} S_t dW_t$$

Let S be the solution to the above SDE. Then the generator $\mathcal{L}$ of the given SDE was:

$$\mathcal{L} = \frac{1}{2}\sigma^2 S_t^{2\beta} S_t^2 \frac{d^2}{ds^2} + rS_t \frac{d}{ds}$$

Our finite difference approximation of the specified Cauchy problem was calculated as:

$$\begin{cases} -v_t(t,s) + \frac{1}{2}\sigma^2 S_t^{2\beta} S_t^2 v_s(t,s) + rS_t v_{ss}(t,s) = rv(t,s) \\ v(0,s) = (s-K)^+ \end{cases}$$

Where:

$$v_s(t,s) = \frac{v_{i+1}(t) - v_{i-1}(t)}{2\Delta s}$$

$$v_{ss}(t,s) = \frac{v_{i+1}(t) - 2v_i(t) + v_{i-1}(t)}{\Delta s^s}$$

Therefore, our approximating system of ODEs becomes the vector matrix: $\frac{d}{dt}v = Av$, where the matrix A has α_i, β_i, γ_i parameters:

$$\begin{cases} \alpha_i = \frac{1}{2}(\frac{\sigma^2 S_t^{2\beta} S_t^2}{\Delta s^2} - \frac{r}{\Delta s}) \\ \beta_i = \frac{-\sigma^2 S_t^{2\beta} S_t^2}{\Delta s^2} - r \\ \gamma_i = \frac{1}{2}(\frac{\sigma^2 S_t^{2\beta} S_t^2}{\Delta s^2} + \frac{r}{\Delta s}) \end{cases}$$

We decided on 1000 space steps as a reasonable size for the scheme. Furthermore, our team decided to use the Crank-Nicolson scheme for the finite difference estimation. We believe this is more efficient and more accurate than the explicit/implicit schemes [2]. In regards to the time steps, we only focused on work weekdays (ie, excluding the weekends). We also made sure to set the appropriate boundary conditions on the system; explicit boundary condition at 0 and linearity boundary condition at 300.

Our objective was then to calibrate the σ and β parameters of the model, using the European call option pricing routine. To set up the least square fit estimation, we first had to weight the option price values by the inverse of the bid-ask spread. We then ran the curve_fit() [3] optimization function from the Python Scipy Optimize library. We decided to use the curve_fit() function as it fit our needs to use the non-linear least squares approximation to calibrate our parameters σ and β.

2.3 Monte-Carlo Pricing

We have calibrated our CEV model with the appropriate σ and β parameters. In order to price discretely monitored barrier options (eg, Knockin and Knockout) with maturity T = 1 year and with monthly monitoring of the barrier, we had to implement a Monte-Carlo pricing method. Our team decided to use sample size of 1000 to acquire a relatively accurate result.

2.3.1 Stock simulation

Part of the Monte-Carlo pricing method involved stock step simulation. In order to stay consistent with our price estimation, we used 2520 simulated stock steps for ease of calculation. Therefore our dt "timestep size" for our stock simulation was $\frac{1}{2520}$. In regards to the risk-free rate r, we used the money market account rate that the Bloomberg OVME <GO> module listed when pricing Knockin/Knockout options. Note that our code uses the initial IBM US stock price S_0 = 160.28; this is due to our code implementation being worked on and accomplished from over the weekend.

2.3.2 Barrier option pricing

For sake of consistency and simplicity, we focused only on barrier call options. When considering the implementation of our barrier option pricing method, we decided to have the end-user input the specified barrier type that they wish to price. For example, if the end-user specifies the barrier type "down-in," then our program will price a down-and-in barrier option. Through this manner, we have a single pricing method whose input parameters are the 'barrier type,' strike value, and barrier value. This is to mimic the specified input required for Bloomberg's OVME module. Our barrier option pricing method can price the following barrier options: down-and-in, up-and-in, down-and-out, and up-and-out.

3 Implementation & Results

3.1 Calibration

After successful implementation of our calibration function, we calibrated the parameters σ and β of the CEV model as follows:

$$\begin{cases} \sigma = 0.123509451756 \\ \beta = -0.00270567786139 \end{cases}$$

We compared our calibrated parameters with the actual prices of the call options for IBM US stock; in other words we compared our call option pricing result (which used our calibrated parameters) with the actual call option price result – which was calculated as $\frac{(ask\ cost + bid\ cost)}{2}$ retrieved from Bloomberg [4]. Our plots showing the comparisons are as follows:

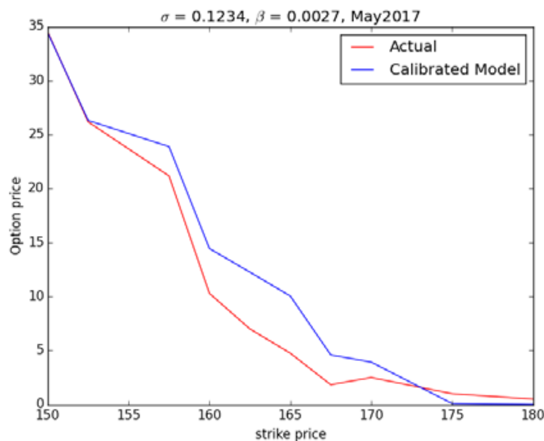

Figure 1 Call option price comparison (May 2017)

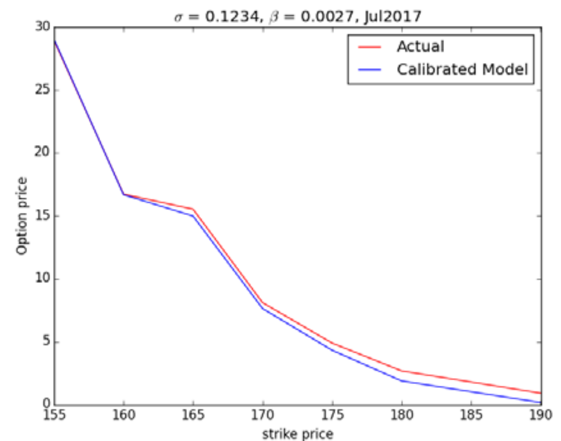

Figure 3 Call option price comparison (July 2017)

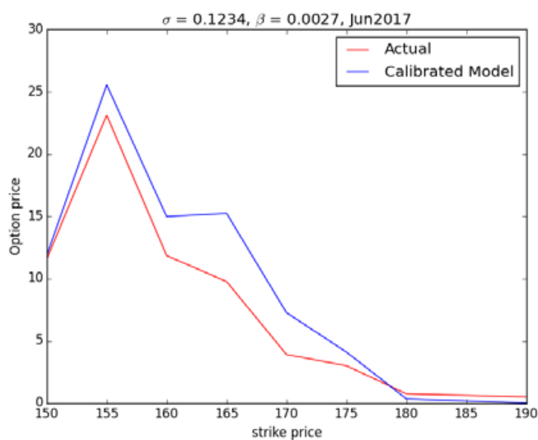

Figure 2 Call option price comparison (June 2017)

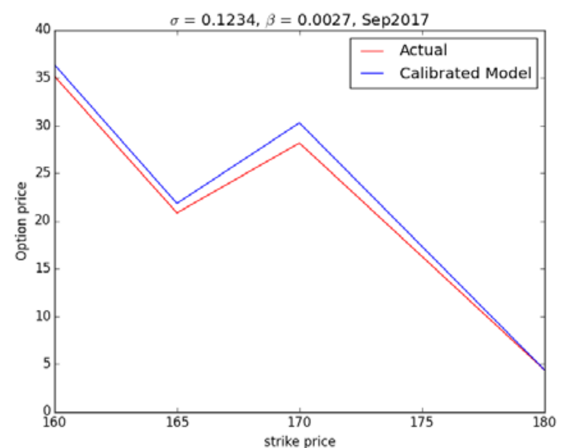

Figure 4 Call option price comparison (Sept 2017)

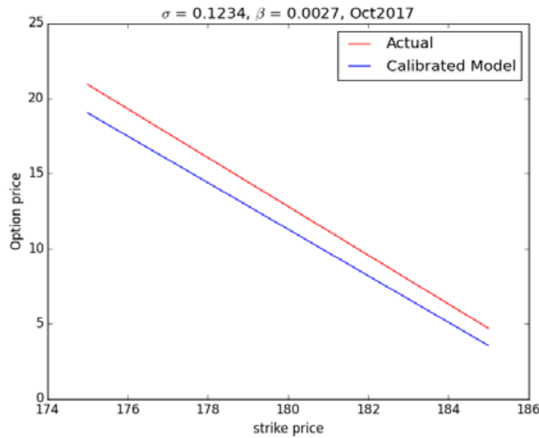

Figure 5 Call option price comparison (Oct 2017)

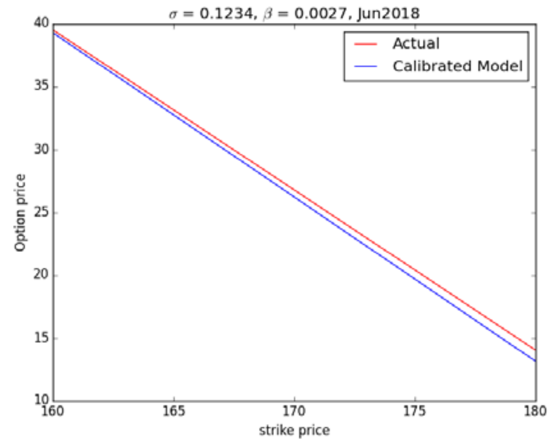

Figure 7 Call option price comparison (June 2018)

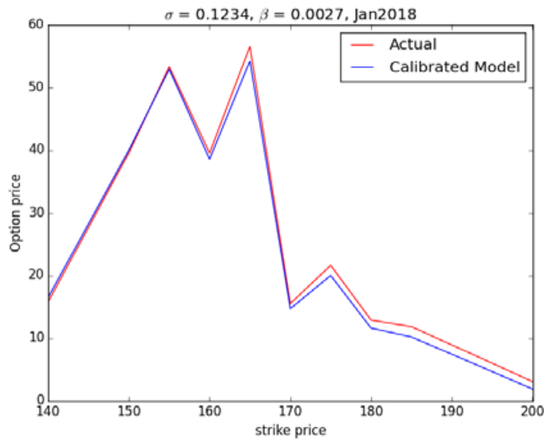

Figure 6 Call option price comparison (Jan 2018)

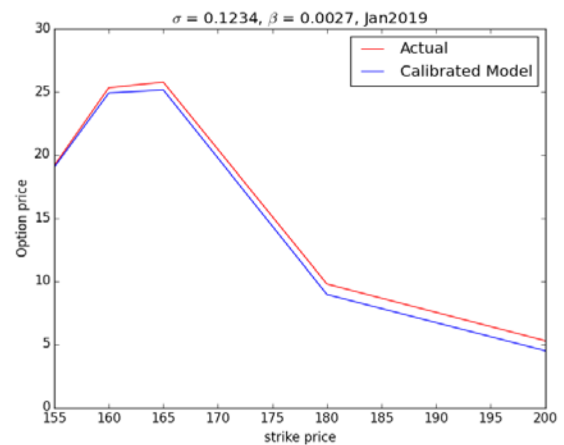

Figure 8 Call option price comparison (Jan 2019)

As clearly seen, our calibrated model is fairly reasonable with the actual results. One clarification that is important to mention is that these option price values are weighted values. Specifically, each actual price and our calculated option prices were all weighted by a factor of $\frac{1}{(ask\ cost - bid\ cost)}$.

To further explain our calibration approach, at first we did individual parameter calibration. In other words we first calibrated the sigma with a fixed beta (eg, $\beta = -1$) and then calibrated beta with a fixed sigma (eg, $\sigma = 0.5$). We discovered – after trial and error – that the best initial guesses for our parameters were $\beta = 0$ and $\sigma = 0.2$.

Due to the large dataset of our CSV files and time complexity of our call option pricing method, our calibration code took some time give us the results. We noticed σ converged to about 0.12 and β actually converged to a small number (10^{-3}).

3.2 Monte-Carlo

Using our calibrated CEV model (with $\sigma = 0.123509451756$ and $\beta = -0.00270567786139$) and our implemented barrier option pricing method, we have the following table of results:

Knockout Barrier price (blue is **UP**-and-out, green is **DOWN**-and-out):

Strike \ Barrier	140	150	160	170	180
150	0	0	0	2.47803296507	5.94641056927
155	0	0	0	0.982371287458	4.61212351608
160	0	0	0	0.683304736495	2.57306242997
165	0	0	0	0.230237151468	1.34757850953
170	0	0	0	0.0976473570919	0.68896997456
175	0	0	0	0.0221989917271	0.308663648371
180	0	0	0	0.0112770980081	0.0821402336922
150	15.6976187282	13.6787985016	5.99147258308	0	0
155	11.2550066186	10.3107281038	5.02576716819	0	0
160	9.14661975315	8.56109903474	3.90293667891	0	0
165	6.55856148052	6.47779342693	3.36377165927	0	0
170	5.24275398701	4.5387507635	2.44281372919	0	0
175	3.08132619119	3.03282416718	1.86487248522	0	0
180	2.66462140169	2.00719912612	1.32916710518	0	0

Knockin Barrier price (blue is **UP**-and-in, green is **DOWN**-and-in):

Strike \ Barrier	140	150	160	170	180
150	14.3820058949	14.5089210853	14.8241338113	13.2890788519	9.84393849695
155	11.2587300564	11.7567893277	12.0056060853	10.8043698454	6.38103429971
160	9.29637556575	9.43584050487	9.40717954225	9.24811571206	7.00356477972
165	6.41518013447	6.89447922008	7.07397367769	5.47225142437	5.03098878208
170	4.85219182396	4.62704921986	5.21322452516	4.54221675682	3.74473982469
175	3.1226207322	3.25053185931	3.44251104896	3.56901236584	3.7714601219
180	2.20310624727	2.35983447996	2.6444677694	2.22590684653	1.92304900808
150	0.118010193747	1.89418642454	9.37834522964	14.6285662172	15.3426337278
155	0.0745086704052	1.15656561728	6.70009610055	11.6590542207	11.7236148895
160	0.0194574822217	0.831406423387	5.02146996848	9.09075526978	8.92369728728
165	0.00241567659375	0.395069991089	3.57641423075	6.44807448356	6.95229790845
170	0.0011832162541	0.154796967095	2.46922941154	4.65969343866	3.90459694603
175	0	0.103181114906	1.25067929067	3.33527418595	3.6048454977
180	0	0.0563878989427	0.860813327007	2.32789741873	2.26111832113

There are a few key points to remark from these results:

Knockout:

The behavior of the knockout option (up or down) is what our team anticipated. For example, clearly for an initial stock value S_0 = 160.28, any up-and-out barrier option with barrier below S_0 will definitely have payoff = 0 because we are already out of the barrier.

Knockin:

In regards to the behavior of the knockin option, the results require more thinking. Note that, for an up-and-in barrier option with a barrier level = 180 and strike = 175, our Monte-Carlo pricing method returns price = 3.77. Although our barrier is quite high in comparison to the initial S_0 = 160.28, thanks to our σ and β parameters for the CEV model, we know that there is some chance that the stock value will actually reach the high barrier level. This results in a small, but not 0, price value. However for a low barrier level for a down-and-in barrier option, we have price results of either 0 or close to 0.

3.3 Bloomberg Comparison

When performing the comparison with Bloomberg's OVME pricing module, we specified the following conditions:

- Pricing Model: Local Volatility
- Dividend: 0%
- Expiration: 365d
- Monitoring Freq: 30

We ignored the BS Discrete and BS Continuous pricing models because our designated CEV model is a local volatility model itself. The results retrieved from Bloomberg's OVME module are in red text [4].

Knockout Barrier price (blue is **UP**-and-out, green is **DOWN**-and-out):

Strike \ Barrier	**140**		**150**		**160**		**170**		**180**	
150	0	0	0	0	0	0	2.47	1.15	5.94	4.59
155	0	0	0	0	0	0	0.98	0.59	4.61	3.06
160	0	0	0	0	0	0	0.68	0.23	2.57	1.84
165	0	0	0	0	0	0	0.23	0.05	1.35	0.92
170	0	0	0	0	0	0	0.09	0	0.69	0.35
175	0	0	0	0	0	0	0.02	0	0.31	0.07
180	0	0	0	0	0	0	0.01	0	0.08	0
150	15.69	17.17	13.67	14.38	5.99	6.26	0	0	0	0
155	11.25	14.13	10.31	12.06	5.02	5.43	0	0	0	0
160	9.14	11.33	8.56	9.87	3.90	4.64	0	0	0	0
165	6.55	8.85	6.47	7.84	3.36	3.84	0	0	0	0
170	5.24	6.73	4.53	6.07	2.44	3.11	0	0	0	0
175	3.08	5.01	3.03	4.58	1.86	2.46	0	0	0	0
180	2.66	3.67	2.00	3.4	1.33	1.91	0	0	0	0

Knockin Barrier price (blue is **UP**-and-in, green is **DOWN**-and-in):

Strike \ Barrier	**140**		**150**		**160**		**170**		**180**	
150	14.38	18.14	14.50	18.14	14.82	18.14	13.29	0.95	9.84	0.07
155	11.25	14.77	11.75	14.77	12.00	14.77	10.80	0.81	6.38	0.06
160	9.29	11.74	9.43	11.74	9.40	11.74	9.25	0.68	7.00	0.05
165	6.41	9.1	6.89	9.1	7.07	9.1	5.47	0.56	5.03	0.04
170	4.85	6.88	4.62	6.88	5.21	6.88	4.54	0.45	3.74	0.04
175	3.12	5.09	3.25	5.09	3.44	5.09	3.56	0.36	3.77	0.03
180	2.20	3.71	2.35	3.71	2.64	3.71	2.23	0.28	1.92	0.03
150	0.11	0.04	1.89	0.39	9.37	6.8	14.62	18.15	15.34	18.15
155	0.07	0.03	1.15	0.3	6.70	5.46	11.66	14.78	11.72	14.78
160	0.02	0.02	0.83	0.22	5.02	4.21	9.09	11.75	8.92	11.75
165	0.002	0.02	0.39	0.16	3.57	3.19	6.45	9.11	6.95	9.11
170	0.001	0.01	0.15	0.11	2.46	2.33	4.66	6.88	3.90	6.88
175	0	0.01	0.10	0.08	1.25	1.65	3.33	5.1	3.60	5.1
180	0	0.01	0.056	0.05	0.86	1.16	2.33	3.72	2.26	3.72

4 Discussion

For our knockout barrier call option (both up and down), everything is as expected and we are satisfied with our implementation for knockout barrier call options. Although there are some discrepencies between our model and Bloomberg's OVME calculated prices, the knockout option allows us to restrict the stock value in a specified range. Thus, this explains why our knockout barrier option values are fairly close to that of Bloomberg.

However, knockin options do not seem as consistent. As seen in the table above, in particular for an up-and-in barrier

option, strike = 150, barrier level = 180, our Monte-Carlo method gave us price = 13.29, whereas Bloomberg returned price = 0.95. For specifically up-and-in barrier options, our price values are an overestimation in comparison to the Bloomberg values.

We performed a volatility comparison analysis between our calibrated σ and the volatility supplied from Bloomberg Barrier option OVME [4]. Our plot is as follows:

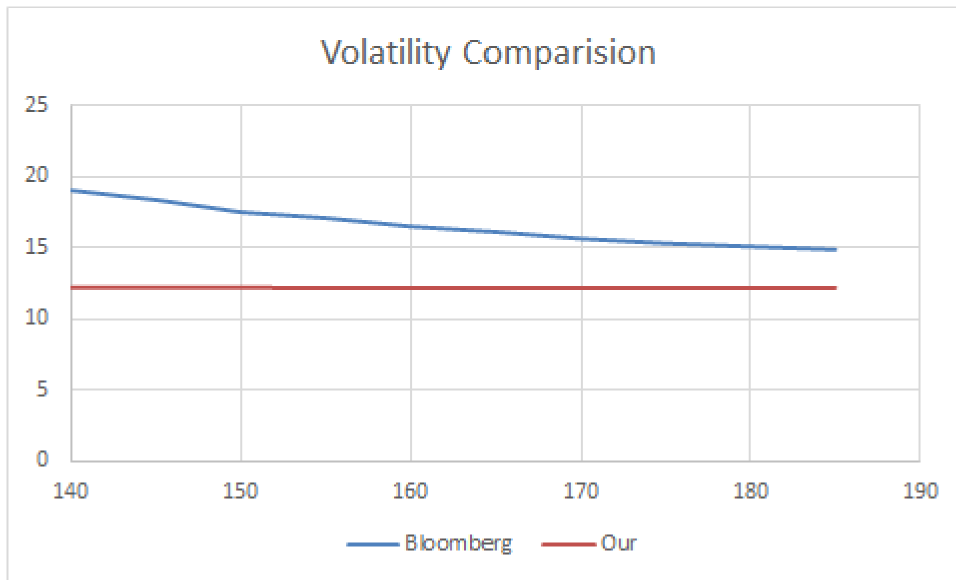

Figure 9 Volatility comparison

This tells us that our calibrated σ stays almost constant for different stock values. But as seen in the figure, Bloomberg's volatility is higher and has almost a "smile" shape [5]. Upon further investigation, we believe that the Bloomberg model uses a different diffusion factor. Our calibrated CEV model has the coefficient of diffusion as "$\sigma S_t^{\beta} S_t$" whereas we do not know what, if any different, coefficient difficusion factor that the Bloomberg model uses. We conclude that Bloomberg may be using an extremely small diffusion factor other than just volatility.

5 Conclusion

The CEV (Constant Elasticity of Variance) model, based on our implementation, is close to the market data and is a valid model to price Barrier options. Our own calibrated CEV model is consistent with the results acquired from Bloomberg's model, except for some knockin option settings. For us to determine which model we would want to use for pricing options, we would require more market data in order to better calibrated the CEV model.

REFERENCES

[1] Paul Glasserman, Monte Carlo Methods in Financial Engineering. Springer 2004. ISBN 978-0-387-00451-3, p. 133.

[2] Ali Hirsa, Computational Methods in Finance. Chapman Hall/CRC 2013. ISBN 978-1-4665-7604-9, p. 72.

[3] Silva, F., (2013). Learning SciPy for numerical and scientific computing (New;1; ed.). Birmingham: Packt Publishing, Limited, p. 83.

[4] Bloomberg L.P. (2017) Option prices for IBM US. Retrieved April 30, 2017 from Bloomberg database.

[5] Shreve, S. E. (2004). Stochastic calculus for finance II: Continuous-time models (Vol. 11). Springer Science & Business Media, p. 125.

APPENDIX

Project_calibr.py:

```
1.  #!/usr/bin/env python2
2.  # -*- coding: utf-8 -*-
3.  """
4.  Created on Fri Apr 28 16:11:17 2017
5.
6.  @author: mmin, akshoop, yxu9
7.  """
8.
9.  import numpy as np
10. import pandas as pd
11. import scipy.optimize as optimization
12. from tridiagonal import solver_modify
13. import matplotlib.pyplot as plt
14.
15. # define the function to give the option price by finite difference scheme
16. def CallOption(par, sigma, beta, r=0.013 ,s_min=0., s_max=300.):
17.
18.     if sigma <= 0:
19.         sigma = 0.1
20.     if beta < -1:
21.         beta = -1
22.     if beta > 0:
23.         beta = 0
24.
25.     '''''set time discretization as 1500, time discretization as 1/252, T=days'''
26.
27.     price = []
28.     for T, k, w in par:
29.         incre = s_max/1000
30.         s = np.array([0.3*i for i in range(1000)])
31.         A = np.array([[0.0]*1000]*1000)
32.         A[0, 0] = -r
33.         A[-1, -1] = r*s_max/incre - r
34.         A[-1, -2] = -r*s_max/incre
35.
36.         for i in range(1, 999):
37.             A[i, i] = -sigma**2*s[i]**(2+2*beta)/incre**2 - r
38.             A[i ,i-1] = 0.5*sigma**2*s[i]**(2+2*beta)/incre**2 - 0.5*r*s[i]/incre
39.             A[i, i+1] = 0.5*sigma**2*s[i]**(2+2*beta)/incre**2 + 0.5*r*s[i]/incre
40.
41.         v = s - float(k)
42.         v[v<0] = 0
43.         # JUST consider the weekdays
44.         weekdays = int(T - 2*(T/7.))
45.         # implement the time discretization until maturity
46.         for i in range(weekdays):
47.             z = np.dot(np.identity(1000)+1/252./2*A, v)
48.             v = solver_modify(np.identity(1000)-1/252./2*A, z)
49.         price.append(v[534]/w)
50.     print sigma, beta
51.     return np.array(price)
52.
53.
54. def calibration(data):
55.     '''''
56.     data is the dataframe contains all data we need, has three columns:"Strike",
57.     "Maturity_Time", "Act_Price"
58.     '''
59.     # set the initial value of beta and sigma
60.     beta = 0
```

```
61.     sigma = 0.2
62.     initial = (sigma, beta)
63.     para = optimization.curve_fit(CallOption,
64.         np.transpose([data['Maturity_Time'].values, data['Strike'].values, data['weight'].val-
    ues]),
65.                         data['Act_Price'], initial)
66.     return para
67.
68.
69. if __name__=="__main__":
70.     names =['Sep2017.csv','Jun2018.csv', 'Jan2018.csv', 'Jan2019.csv',
71.     'Jul2017.csv', 'May2017.csv', 'Jun2017.csv', 'Oct2017.csv']
72.     # combine all needed data together in one dataframe
73.     data = pd.DataFrame(columns = ['Strike','Maturity_Time', 'Act_Price'])
74.     for filename in names:
75.         newdata = pd.DataFrame(columns = ['Strike','Maturity_Time', 'Act_Price', 'weight'])
76.         f = pd.read_csv(filename)
77.         try:
78.             T = float(f['Strike'][0][11:14])
79.         except ValueError:
80.             T = float(f['Strike'][0][11:13])
81.         # use data with more than 10 volume to calculate
82.         f = f[f['Volm']>10]
83.         newdata['Strike'] = np.array(f['Strike'], dtype=float)
84.         newdata['Maturity_Time'] = T
85.         newdata['Act_Price'] = np.array((f['Bid']+f['Ask'])/2, dtype=float)/np.array(f['Ask']-
    f['Bid'], dtype=float)
86.         newdata['weight'] = np.array(f['Ask']-f['Bid'], dtype=float)
87.         data = data.append(newdata, ignore_index = True)
88.     newParams = calibration(data)
89.     print("Our calibrated sigma and beta for the CEV model is:")
90.     print newParams[0]
```

Project_MC.py:

```
1.  #!/usr/bin/env python2
2.  # -*- coding: utf-8 -*-
3.  """
4.  Created on Fri Apr 28 16:11:17 2017
5.
6.  @author: mmin, akshoop, yxu9
7.  """
8.
9.  import numpy as np
10. import pandas as pd
11. import scipy.optimize as optimization
12. from tridiagonal import solver_modify
13. import matplotlib.pyplot as plt
14.
15. # generate stock process
16. def Stock(sigma, beta, r=0.013, s=160.28, T=1.):
17.     stock = [s]
18.     N = 2520
19.     for i in range(N):
20.         stock.append(stock[-1] + r*stock[-1]*(T/N) + sigma*stock[-
    1]**(beta+1)*np.sqrt(T/N)*np.random.standard_normal())
21.     return np.array(stock)
22.
23. # function calculate Barrier option payoff with different type
24. def BarrierOption(stock, k, b, Type, T=1.):
25.     ''''stock: the stock process; b: barrier; Type: option type
26.         Assume all these are call option'''
27.     check = [stock[210*i] for i in range(12)]
28.     if Type=='up-out':
29.         if max(check) >= b:
30.             return 0
```

```
31.         else: payoff = stock[-1]-k
32.
33.     if Type=='up-in':
34.         if max(check) >= b:
35.             payoff = stock[-1]-k
36.         else:
37.             return 0
38.
39.     if Type=='down-out':
40.         if min(check) <= b:
41.             return 0
42.         else: payoff = stock[-1]-k
43.
44.     if Type=='down-in':
45.         if min(check) <= b:
46.             payoff = stock[-1]-k
47.         else: return 0
48.     return payoff if payoff > 0 else 0
49.
50.
51. if __name__=="__main__":
52.     # Part c)
53.     # do Monte Carlo methods
54.     n = 1000
55.     price = []
56.     Type = raw_input("Please input your barrier types: (choose from 'up-out', 'up-in', 'down-
    out', 'down-in') ")
57.     k = input("What is your strike price: ")
58.     b = input("What is your barrier: ")
59.     r = 0.0138
60.     T = 1.
61.     # retrieved from CEV model calibration
62.     sigma = 0.123509451756
63.     beta = -0.00270567786139
64.     for i in range(n):
65.         stock = Stock(sigma, beta)
66.         price.append(np.exp(-r*T)*BarrierOption(stock, k, b, Type))
67.     price = np.array(price)
68.     # print the price(np.mean(price))
69.     print "The price of your Barrier Option is: "+ str(np.mean(price))
70.
```