WORCESTER POLYTECHNIC INSTITUTE

COMPUTATIONAL METHODS OF FINANCIAL MATHEMATICS

Project report

Author:
Sahil SHAHANI
Fangzheng SUN
Weixi LIU

Instructor:
Professor Stephan Sturm

May 2, 2017

Contents

List of Figures

List of Abbreviations

B	**B**arrier **L**evel
BS	**B**lack-**S**choles
FDS	**F**inite **D**ifference **S**cheme
ODE	**O**rdinary **D**ifference **E**quatiion
SDE	**S**tochastic **D**ifference **E**quatiion
CEV	**C**onstant **E**lasticity of **V**ariance model
IBM	**I**nternational **B**usiness **M**achines

Physical Constants

Risk-free interest rate:	$r = 0.0375$ (Yield at 04/28/2017)
Underlying price:	$s = 160.29$ (Data: 04/28/2017)

List of Symbols

r	risk-free interest rate
s	underlying price
v	option price
H	payoff function
$\mathcal{L}$	generator
A	matrix

Chapter 1

Introduction

In this project, we justify the need of computational methods used for pricing options and do a stochastic volatility (CEV) model calibration with market data of call option prices on IBM stock over all times and maturities to find the optimal parameters where CEV model price is equal to the Black-Scholes model. Then we develop this pricing routine using finite difference scheme and a least square fit model and consider options which are sufficiently liquid in the market. We further use this calibrated model to price a discretely monitored Barrier option with monthly monitoring intervals. We make use of Milstein scheme to generate the stock path and Monte Carlo methods to price this option.

Chapter 2

Data Collection

2.1 European Call option

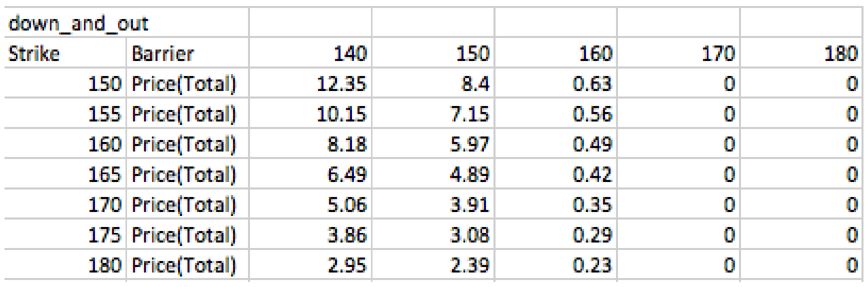
Through Bloomberg Terminal, we found **8** maturity dates options over span of **next 2 years** and created a program which extracted all data for each of these maturity dates. Python Pandas package helps us to read excel file into data frame (Shown in figure 2.1). We extracted the bid price, ask price, strike price, volume and transferred them into eight 2-dimesional arrays regarding to 8 options respectively. And they would be accessed in part (b) of the project to determine the liquidity of the option.

As shown in the figure 2.1,each **maturity object** has 25 strike prices.

2.2 Barrier Option

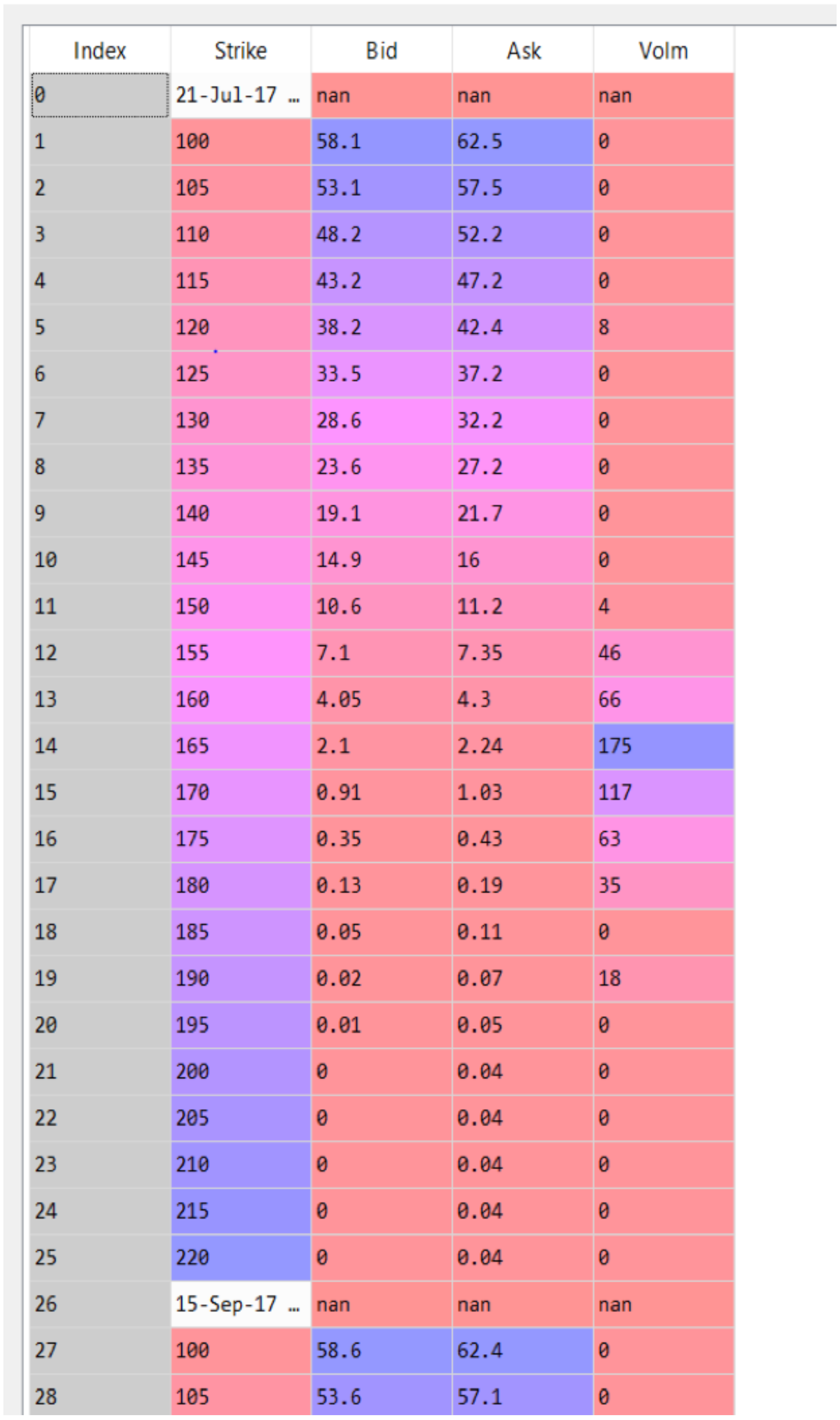
For each model (BS continuous, BS discrete, Local volatility, Trinomial), and for each option type, we collect option prices when $K = 150, 155, 160, 165, 170, 175, 180$, and $B = 140, 150, 160, 170, 180$ from Bloomberg Terminal.

For example, figure 2.2 as below is down-and-out option prices calculated by BS continuous model.



down_and_out						
Strike	Barrier	140	150	160	170	180
150	Price(Total)	12.35	8.4	0.63	0	0
155	Price(Total)	10.15	7.15	0.56	0	0
160	Price(Total)	8.18	5.97	0.49	0	0
165	Price(Total)	6.49	4.89	0.42	0	0
170	Price(Total)	5.06	3.91	0.35	0	0
175	Price(Total)	3.86	3.08	0.29	0	0
180	Price(Total)	2.95	2.39	0.23	0	0

FIGURE 2.2: Down-and-out option prices calculated by BS continuous model.



Index	Strike	Bid	Ask	Volm
0	21-Jul-17 …	nan	nan	nan
1	100	58.1	62.5	0
2	105	53.1	57.5	0
3	110	48.2	52.2	0
4	115	43.2	47.2	0
5	120	38.2	42.4	8
6	125	33.5	37.2	0
7	130	28.6	32.2	0
8	135	23.6	27.2	0
9	140	19.1	21.7	0
10	145	14.9	16	0
11	150	10.6	11.2	4
12	155	7.1	7.35	46
13	160	4.05	4.3	66
14	165	2.1	2.24	175
15	170	0.91	1.03	117
16	175	0.35	0.43	63
17	180	0.13	0.19	35
18	185	0.05	0.11	0
19	190	0.02	0.07	18
20	195	0.01	0.05	0
21	200	0	0.04	0
22	205	0	0.04	0
23	210	0	0.04	0
24	215	0	0.04	0
25	220	0	0.04	0
26	15-Sep-17 …	nan	nan	nan
27	100	58.6	62.4	0
28	105	53.6	57.1	0

FIGURE 2.1: Example of one Pandas data frame.

Chapter 3

Methodology

3.1 Feynman-Kac Formula

Theorem 1. *Let X be the solution of an SDE with generator $\mathcal{L}$. Then the function*

$$v(t,x) = E^Q[e^{-\int_t^T g(X_s)ds} H(X_T) | X_t = x]$$

is the (unique) solution to the Cauchy problem

$$v_t(t,x) + \mathcal{L}v(t,x) = g(x)v(t,x)$$

$$V(T,x) = h(x)$$

Here the first line describes a partial differential equation (PDE) and the second gives a terminal condition.

3.2 Crank-Nicolson Scheme

When used a centered finite difference shceme on an equidistributed grid for the spatial derivatives, we get an approximation by a system of ODEs.

This can easily be written in vector-matrix form

$$\frac{dv}{dt} = Av$$

where

$$A = \begin{pmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \underbrace{0\cdots0}_{i-3} & \alpha_{i-1} & \beta_{i-1} & \gamma_{i-1} & \underbrace{0\cdots0}_{I-i} & & \\ & \underbrace{0\cdots0}_{i-2} & \alpha_i & \beta_i & \gamma_i & \underbrace{0\cdots0}_{I-i-1} & \\ & & \underbrace{0\cdots0}_{i-1} & \alpha_{i+1} & \beta_{i+1} & \gamma_{i+1} & \underbrace{0\cdots0}_{I-i-2} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix} \tag{3.1}$$

with $\alpha_i = \frac{1}{2}(\frac{\sigma_i^2}{\Delta x^2} - \frac{b_i}{\Delta x}), \beta_i = -\frac{\sigma_i^2}{\Delta x^2} - g_i, \gamma_i = \frac{1}{2}(\frac{\sigma_i^2}{\Delta x^2} + \frac{b_i}{\Delta x})$

Thus we have transformed the Cauchy problem into solving a system of ODEs,

$$\frac{dv}{dt} = Av$$

Using the grid $\{t_0, \cdots, t_n, \cdots, t_N\}$ with $t_{n+1} - t_n = \Delta t$, and the notation $v_i^n = v_i(t_n) = v(t_n, x_i)$, we can wirte it as

$$\frac{v_i^{n+1} - v_i^{n-1}}{2\Delta t} = (Av^n)_i$$

Combined with

$$\frac{v_i^{n+1} - v_i^{n}}{2\Delta t} = Av^{n+1}$$

We have

$$(I_N - \frac{1}{2}\Delta t A)v_i^{n+1} = (I_N + \frac{1}{2}\Delta t A)v_i^n$$

Hence

$$v_i^{n+1} = (I_N - \frac{1}{2}\Delta t A)^{-1}(I_N + \frac{1}{2}\Delta t A)v_i^n$$

This is Crank-Nicolson Scheme.

3.3 Milstein Scheme

- To approximate

$$dX_t = b(X_t)dt + \sigma(X_t)dW_t, X_0 = x$$

- we write

$$X_t = x + \sum_{t_i \le t} \int_{t_i}^{t_{i+1}} b(X_s)ds + \sum_{t_i \le t} \int_{ti}^{t_{i+1}} \sigma(X_s)dW_s$$

- By Ito's formula, we have

$$\sigma(X_s) = \sigma(X_t) + \int_s^t \sigma^{'}(X_u)\sigma(X_u)dW_u + \int_s^t \sigma^{'}(X_u)b(X_u)du + \frac{1}{2}\int_s^t \sigma^{''}(X_u)\sigma^2(X_u)du$$

$\Rightarrow$

$$\sigma(X_s) = \sigma(X_t) + \sigma^{'}(X_t)\sigma(X_t)(W_s - W_t)$$

- Thus for small h,

$$\int_t^{t+h} \sigma(X_s)ds \approx \int_t^{t+h} \sigma(X_t) + \sigma^{'}(X_t)(W_s - W_t)dW_s$$
$$= \sigma(X_t)(W_{t+h} - W_t) + \sigma^{'}(X_t)\sigma(X_t)(\frac{1}{2}(W_{t+h} - W_t)^2 - \frac{1}{2}t)$$

- Finally, we have

$$X_{t_{i+1}} = X_{t_i} + b(X_t)\frac{T}{N} + \sigma(X_t)\sqrt{\frac{T}{N}}Z_i + \frac{1}{2}\sigma^{'}(X_t)\sigma(X_t)(\frac{T}{N}Z_i^2 - \frac{T}{N})$$

3.4 Calibration

Definition
Calibration is a procedure for finding the parameters from the market data.

- One tries to set up the calibration in a way that thecalculated prices are as close as possible to the actual market data
- Denote the parameter set $\Theta = \{\theta_1, \theta_2, \cdots, \theta_n\}$, the model prices by C_i^{Θ} and the true price by C_i
- Try to determine the optimal parameters by

  $$\underset{\Theta}{\operatorname{argmin}} \sum_i w_i |C_i^{\Theta} - C_i|^2$$

  Where w_i is weights, at here, we use bid-ask spread as weights
- we done such a procedure via the scipy optimize package, we learnt the package from https://python4mpia.github.io/fitting_data/least-squares-fitting.html.

Chapter 4

Implementation

4.1 Calibrate the CEV Model

4.1.1 Assumptions

1. Determining the actual price of the stock by averaging the bid and ask price.
2. Set minimum volume to 10 for liquidity purpose.

4.1.2 Procedure and Result

We calibrated the CEV model with the market data on IBM Option against the actual price from Bloomberg and found the optimal parameters (sigma and beta) for the SDE of CEV model for all maturities. Here we considered European Call (Vanilla) options with (bid-minus-ask spread) less than 1 and volume greater than 10, as this defines their liquidity in the market.

We first created an array of all strikes and actual option price plus CEV and BS model functions for calibration and solved for the CEV

$$dS_t = rS_t dt + \sigma S_t^{\beta} S_t dW_t, S_0 = s$$

We have

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

Thus, by Feynman-Kac Formula and Theorem 1 we know

$$v(t,s) = \mathbb{E}^{\mathbb{Q}}[e^{-r(T-t)}(S_T - K)^+ | S_{T-t} = x]$$

is the unique solution to the Cauchy problem:

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

Applying the following Crank Nicolson Finite Difference Scheme (FDS) to solve the problem, since be testing, this scheme can have a reasonable result conparing with option price calculated by BS formula.

$$V_i^{n+1} = (I_N - \frac{1}{2}\Delta t A)^{-1}(I_N + \frac{1}{2}\Delta t A)v_i^n$$

Then we created **A matrix** from the coefficients in the FDS ODE, with boundary condition and linearity boundary conditions:

- For $i = 0$:

$$A[0][0] = -\frac{\sigma^2 S_i^{2\beta+2}}{\Delta s^2} - r$$

$$A[0][1] = \frac{1}{2}\frac{\sigma^2 S_i^{2\beta+2}}{\Delta s^2} + \frac{rS_i}{\Delta s}$$

- For $i = 1, 2, \cdots, n-1$:

$$A[i][i-1] = \frac{1}{2}\frac{\sigma^2 S_i^{2\beta+2}}{\Delta s^2} - \frac{rS_i}{\Delta s}$$

$$A[i][i] = -\frac{\sigma^2 S_i^{2\beta+2}}{\Delta s^2} - r$$

$$A[i][i+1] = \frac{1}{2}\frac{\sigma^2 S_i^{2\beta+2}}{\Delta s^2} + \frac{rS_i}{\Delta s}$$

- For $i = n$:

$$A[-1][-2] = -\frac{rS_{max}}{\Delta s}$$

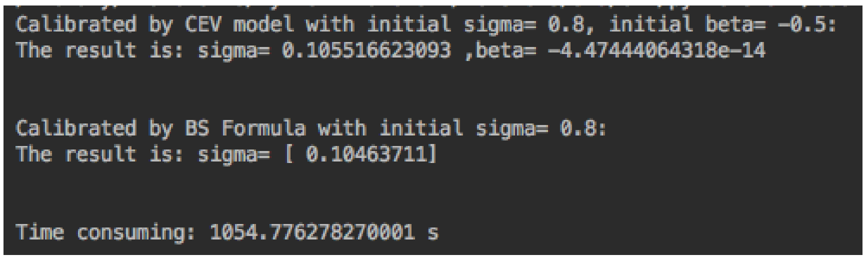
$$A[-1][-2] = \frac{rS_{max}}{\Delta s} - r$$

Finally, we obtained the vector-matix form and our option grid.

The least square fit API (optimization.curve_fit) from scipy library are used in calibration of the CEV model and Black Scholes model using, by forwarding the actual price array vs. price array found by CEV and BS functions respectively.

Consequently, we found that the optimal values for beta and sigma(volatility) are $-4.47444 * 10^{-14}$ and 0.10552 respectively.

Result from Python as below:



```
Calibrated by CEV model with initial sigma= 0.8, initial beta= -0.5:
The result is: sigma= 0.105516623093 ,beta= -4.47444064318e-14


Calibrated by BS Formula with initial sigma= 0.8:
The result is: sigma= [ 0.10463711]


Time consuming: 1054.776278270001 s
```

FIGURE 4.1: Calibration result.

By verifying our calibration by comparing the option price matrix output by both functions (Black Scholes and CEV), we concluded that the price approximations were similar for both models.

4.2 Price Barrier Options by Monte-Carlo Methods

4.2.1 Assumptions

1. 360 days as our year duration for convenience
2. 30 days a month as the monitoring period

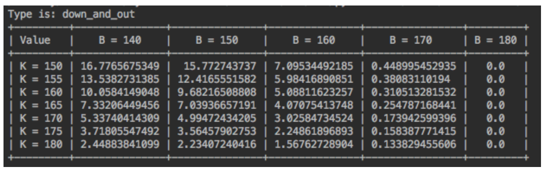
4.2.2 Procedure and Result

We focused on the price of different discretely monitored barrier options such as (Knock-Out/Knock-In option types) by Monte-Carlo methods. We generated the stock path using the following Milstein Scheme and did monthly monitoring by comparing the stock prices to the barrier at monthly interval.

$$S_{t_{i+1}} = S_{t_i} + rS_{t_i}\frac{T}{N} + \sigma S_{t_i}^{\beta} S_{t_i}\sqrt{\frac{T}{N}}Z_i + \frac{1}{2}\sigma(\beta+1)S_{t_i}^{\beta}\sigma S_{t_i}^{\beta+1}(\frac{T}{N}Z_i^2 - \frac{T}{N})$$

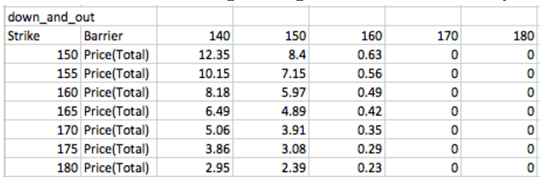
We passed initial stock price, barrier value, strike price and option type as function parameters to the CEV function in MonteCarlo.py file. This function ran a switch on the type parameter to determine the option type and sets a flag if barrier is hit. At every 30th day, we monitored the stock price to the barrier and set a flag to determine if the option is in-the-money or out-of-money depending on the option type [up-and-in, up-and-out, down-and-in, down-and-out]. We checked the flag value and got the prices of the options accordingly.

By tring different models in Bloomberg, we found option price calculated by BS-continuous model is lower than our model. Just putting down-and-out type barrier option as an example, here is the comparison:



```
Type is: down_and_out
+---------+---------------+---------------+---------------+----------------+---------+
| Value   |    B = 140    |    B = 150    |    B = 160    |    B = 170     | B = 180 |
+---------+---------------+---------------+---------------+----------------+---------+
| K = 150 | 16.7765675349 |  15.772743737 | 7.09534492185 | 0.448995452935 |   0.0   |
| K = 155 | 13.5382731385 | 12.4165551582 | 5.98416890851 | 0.38083110194  |   0.0   |
| K = 160 | 10.0584149048 | 9.68216508808 | 5.08811623257 | 0.310513281532 |   0.0   |
| K = 165 | 7.33206449456 | 7.03936657191 | 4.07075413748 | 0.254787168441 |   0.0   |
| K = 170 | 5.33740414309 | 4.99472434205 | 3.02584734524 | 0.173942599396 |   0.0   |
| K = 175 | 3.71805547492 | 3.56457902753 | 2.24861896893 | 0.158387771415 |   0.0   |
| K = 180 | 2.44883841099 | 2.23407240416 | 1.56762728904 | 0.133829455606 |   0.0   |
+---------+---------------+---------------+---------------+----------------+---------+
```

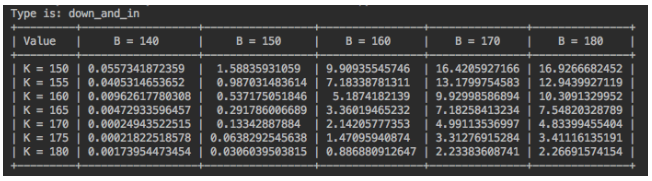
FIGURE 4.2: down and out option price calculated by our model.



down_and_out						
Strike	Barrier	140	150	160	170	180
150	Price(Total)	12.35	8.4	0.63	0	0
155	Price(Total)	10.15	7.15	0.56	0	0
160	Price(Total)	8.18	5.97	0.49	0	0
165	Price(Total)	6.49	4.89	0.42	0	0
170	Price(Total)	5.06	3.91	0.35	0	0
175	Price(Total)	3.86	3.08	0.29	0	0
180	Price(Total)	2.95	2.39	0.23	0	0

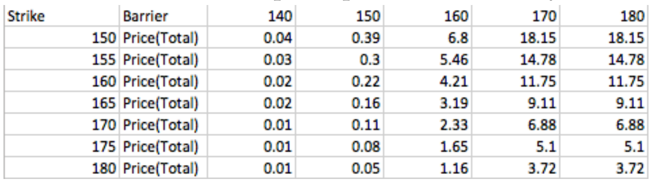
FIGURE 4.3: down and out option price collected from bloomberg terminal with BS continuous model.

For local volatility model, we surprised found that those data comes from Bloomberg Terminal are very close with our result. Below are comparisons:



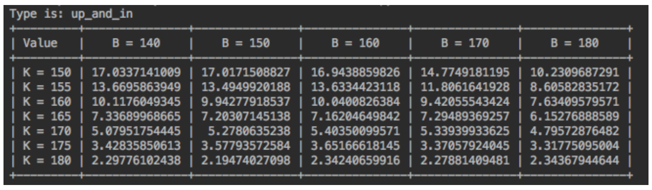
```
Type is: down_and_in
+---------+------------------+------------------+------------------+-----------------+-----------------+
| Value   |     B = 140      |     B = 150      |     B = 160      |    B = 170      |    B = 180      |
+---------+------------------+------------------+------------------+-----------------+-----------------+
| K = 150 | 0.0557341872359  |  1.58835931059   | 9.90935545746    | 16.4205927166   | 16.9266682452   |
| K = 155 | 0.0405314653652  |  0.987031483614  | 7.18338781311    | 13.1799754583   | 12.9439927119   |
| K = 160 | 0.00962617780308 |  0.537175051846  |  5.1874182139    | 9.92998586894   | 10.3091329952   |
| K = 165 | 0.00472933596457 |  0.291786006689  | 3.36019465232    | 7.18258413234   | 7.54820328789   |
| K = 170 | 0.00024943522515 |  0.13342887884   | 2.14205777353    | 4.99113536997   | 4.83399455404   |
| K = 175 | 0.00021822518578 | 0.0638292545638  | 1.47095940874    | 3.31276915284   | 3.41116135191   |
| K = 180 | 0.00173954473454 | 0.0306039503815  | 0.886880912647   | 2.23383608741   | 2.26691574154   |
+---------+------------------+------------------+------------------+-----------------+-----------------+
```

FIGURE 4.4: down and in option price calculated by our model.



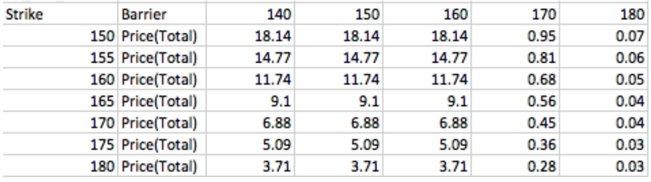
Strike	Barrier	140	150	160	170	180
150	Price(Total)	0.04	0.39	6.8	18.15	18.15
155	Price(Total)	0.03	0.3	5.46	14.78	14.78
160	Price(Total)	0.02	0.22	4.21	11.75	11.75
165	Price(Total)	0.02	0.16	3.19	9.11	9.11
170	Price(Total)	0.01	0.11	2.33	6.88	6.88
175	Price(Total)	0.01	0.08	1.65	5.1	5.1
180	Price(Total)	0.01	0.05	1.16	3.72	3.72

FIGURE 4.5: down and in option price collected from bloomberg terminal with local volatility model.



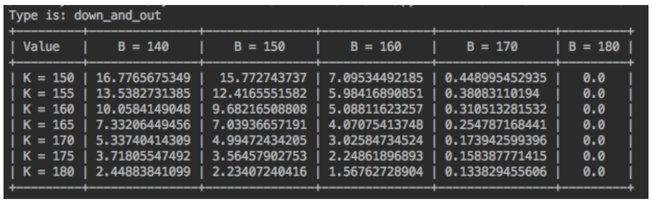
```
Type is: up_and_in
+---------+---------------+---------------+---------------+---------------+---------------+
| Value   |   B = 140     |   B = 150     |   B = 160     |   B = 170     |   B = 180     |
+---------+---------------+---------------+---------------+---------------+---------------+
| K = 150 | 17.0337141009 | 17.0171508827 | 16.9438859826 | 14.7749181195 | 10.2309687291 |
| K = 155 | 13.6695863949 | 13.4949920188 | 13.6334423118 | 11.8061641928 | 8.60582835172 |
| K = 160 | 10.1176049345 | 9.94277918537 | 10.0400826384 | 9.42055543424 | 7.63409579571 |
| K = 165 | 7.33689968665 | 7.20307145138 | 7.16204649842 | 7.29489369257 | 6.15276888589 |
| K = 170 | 5.07951754445 |  5.2780635238 | 5.40350099571 | 5.33939933625 | 4.79572876482 |
| K = 175 | 3.42835850613 | 3.57793572584 | 3.65166618145 | 3.37057924045 | 3.31775095004 |
| K = 180 | 2.29776102438 | 2.19474027098 | 2.34240659916 | 2.27881409481 | 2.34367944644 |
+---------+---------------+---------------+---------------+---------------+---------------+
```

FIGURE 4.6: up and in option price calculated by our model.



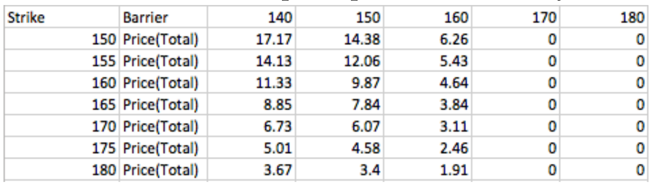
Strike	Barrier	140	150	160	170	180
150	Price(Total)	18.14	18.14	18.14	0.95	0.07
155	Price(Total)	14.77	14.77	14.77	0.81	0.06
160	Price(Total)	11.74	11.74	11.74	0.68	0.05
165	Price(Total)	9.1	9.1	9.1	0.56	0.04
170	Price(Total)	6.88	6.88	6.88	0.45	0.04
175	Price(Total)	5.09	5.09	5.09	0.36	0.03
180	Price(Total)	3.71	3.71	3.71	0.28	0.03

FIGURE 4.7: up and in option price collected from bloomberg terminal with local volatility model.



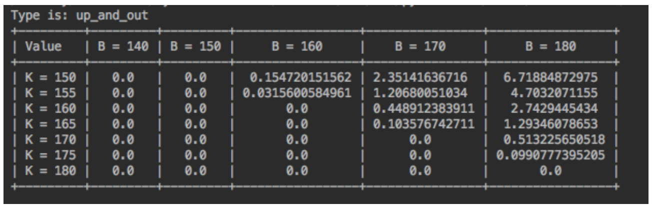
```
Type is: down_and_out
+------------+----------------+----------------+----------------+----------------+-----------+
| Value      |    B = 140     |    B = 150     |    B = 160     |    B = 170     | B = 180 |
+------------+----------------+----------------+----------------+----------------+-----------+
| K = 150 | 16.7765675349 |  15.772743737 | 7.09534492185 | 0.448995452935 |    0.0    |
| K = 155 | 13.5382731385 | 12.4165551582 | 5.98416890851 | 0.38083110194  |    0.0    |
| K = 160 | 10.0584149048 | 9.68216508808 | 5.08811623257 | 0.310513281532 |    0.0    |
| K = 165 | 7.33206449456 | 7.03936657191 | 4.07075413748 | 0.254787168441 |    0.0    |
| K = 170 | 5.33740414309 | 4.99472434205 | 3.02584734524 | 0.173942599396 |    0.0    |
| K = 175 | 3.71805547492 | 3.56457902753 | 2.24861896893 | 0.158387771415 |    0.0    |
| K = 180 | 2.44883841099 | 2.23407240416 | 1.56762728904 | 0.133829455606 |    0.0    |
+------------+----------------+----------------+----------------+----------------+-----------+
```

FIGURE 4.8: down and out option price calculated by our model.



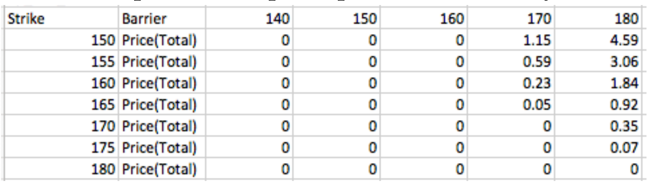
Strike	Barrier	140	150	160	170	180
150	Price(Total)	17.17	14.38	6.26	0	0
155	Price(Total)	14.13	12.06	5.43	0	0
160	Price(Total)	11.33	9.87	4.64	0	0
165	Price(Total)	8.85	7.84	3.84	0	0
170	Price(Total)	6.73	6.07	3.11	0	0
175	Price(Total)	5.01	4.58	2.46	0	0
180	Price(Total)	3.67	3.4	1.91	0	0

FIGURE 4.9: down and out option price collected from bloomberg terminal with local volatility model.



```
Type is: up_and_out
+------------+---------+---------+------------------+------------------+------------------+
| Value      | B = 140 | B = 150 |     B = 160      |     B = 170      |     B = 180      |
+------------+---------+---------+------------------+------------------+------------------+
| K = 150 |   0.0   |   0.0   |  0.154720151562 | 2.35141636716    |  6.71884872975   |
| K = 155 |   0.0   |   0.0   | 0.0315600584961 | 1.20680051034    |   4.7032071155   |
| K = 160 |   0.0   |   0.0   |       0.0        | 0.448912383911 |   2.7429445434   |
| K = 165 |   0.0   |   0.0   |       0.0        | 0.103576742711 |  1.29346078653   |
| K = 170 |   0.0   |   0.0   |       0.0        |       0.0        |  0.513225650518 |
| K = 175 |   0.0   |   0.0   |       0.0        |       0.0        | 0.0990777395205 |
| K = 180 |   0.0   |   0.0   |       0.0        |       0.0        |       0.0        |
+------------+---------+---------+------------------+------------------+------------------+
```

FIGURE 4.10: up and out option price calculated by our model.



Strike	Barrier	140	150	160	170	180
150	Price(Total)	0	0	0	1.15	4.59
155	Price(Total)	0	0	0	0.59	3.06
160	Price(Total)	0	0	0	0.23	1.84
165	Price(Total)	0	0	0	0.05	0.92
170	Price(Total)	0	0	0	0	0.35
175	Price(Total)	0	0	0	0	0.07
180	Price(Total)	0	0	0	0	0

FIGURE 4.11: up and out option price collected from bloomberg terminal with local volatility model.

Appendix A

Code

A.1 DataCollect

```
import pandas as pd

row_12 = pd.read_excel('data1,2.xlsx')
row_12.drop(['Ticker', 'Last', 'IVM'], axis=1, inplace=True)
July_21_17 = row_12[1:26].as_matrix()
Sept_15_17 = row_12[27:].as_matrix()

row_34 = pd.read_excel('data3,4.xlsx')
row_34.drop(['Ticker', 'Last', 'IVM'], axis=1, inplace=True)
June_15_18 = row_34[1:26].as_matrix()
Jan_18_19 = row_34[27:].as_matrix()

row_56 = pd.read_excel('data5,6.xlsx')
row_56.drop(['Ticker', 'Last', 'IVM'], axis=1, inplace=True)
May_19_17 = row_56[1:26].as_matrix()
June_16_17 = row_56[27:].as_matrix()

row_78 = pd.read_excel('data7,8.xlsx')
row_78.drop(['Ticker', 'Last', 'IVM'], axis=1, inplace=True)
Oct_20_17 = row_78[1:26].as_matrix()
Jan_19_18 = row_78[27:].as_matrix()
```

A.2 Calibration

```
import numpy as np
from numpy.linalg import inv
import scipy.optimize as optimization
from scipy.stats import norm
from CollectData28 import July_21_17
from CollectData28 import Sept_15_17
from CollectData28 import June_15_18
from CollectData28 import Jan_18_19
from CollectData28 import May_19_17
from CollectData28 import June_16_17
from CollectData28 import Oct_20_17
from CollectData28 import Jan_19_18
import timeit
start_BM_Ite = timeit.default_timer()
```

```
## Sifting
T = []
xdata = []
K = []
price_actual = []
weight = []

for data in range(8):
if data == 0:
for i in range(len(July_21_17)):
if abs(July_21_17[i][1]-July_21_17[i][2]) <= 1 and July_21_17[i][3] >= 10:
K.append(July_21_17[i][0])
weight.append(1.0/abs(July_21_17[i][1]-July_21_17[i][2]))
#weight.append(1.0)
price_actual.append(0.5*(July_21_17[i][1] + July_21_17[i][2]))
T.append(84)

if data == 1:
for i in range(len(Sept_15_17)):
if abs(Sept_15_17[i][1]-Sept_15_17[i][2]) <= 1 and Sept_15_17[i][3] >= 10:
K.append(Sept_15_17[i][0])
weight.append(1.0 / abs(Sept_15_17[i][1]-Sept_15_17[i][2]))
#weight.append(1.0)
price_actual.append(0.5 * (Sept_15_17[i][1] + Sept_15_17[i][2]))
T.append(140)

if data == 2:
for i in range(len(June_15_18)):
if abs(June_15_18[i][1]-June_15_18[i][2]) <= 1 and June_15_18[i][3] >= 10:
K.append(June_15_18[i][0])
weight.append(1.0 / abs(June_15_18[i][1]-June_15_18[i][2]))
#weight.append(1.0)
price_actual.append(0.5 * (June_15_18[i][1] + June_15_18[i][2]))
T.append(413)

if data == 3:
for i in range(len(Jan_18_19)):
if abs(Jan_18_19[i][1]-Jan_18_19[i][2]) <= 1 and Jan_18_19[i][3] >= 10:
K.append(Jan_18_19[i][0])
weight.append(1.0 / abs(Jan_18_19[i][1]-Jan_18_19[i][2]))
#weight.append(1.0)
price_actual.append(0.5 * (Jan_18_19[i][1] + Jan_18_19[i][2]))
T.append(630)

if data == 4:
for i in range(len(May_19_17)):
if abs(May_19_17[i][1]-May_19_17[i][2]) <= 1 and May_19_17[i][3] >= 10:
K.append(May_19_17[i][0])
weight.append(1.0 / abs(May_19_17[i][1]-May_19_17[i][2]))
#weight.append(1.0)
price_actual.append(0.5 * (May_19_17[i][1] + May_19_17[i][2]))
T.append(21)

if data == 5:
for i in range(len(June_16_17)):
```

```
if abs(June_16_17[i][1]-June_16_17[i][2]) <= 1 and June_16_17[i][3] >= 10:
K.append(June_16_17[i][0])
weight.append(1.0 / abs(June_16_17[i][1]-June_16_17[i][2]))
#weight.append(1.0)
price_actual.append(0.5 * (June_16_17[i][1] + June_16_17[i][2]))
T.append(49)

if data == 6:
for i in range(len(Oct_20_17)):
if abs(Oct_20_17[i][1]-Oct_20_17[i][2]) <= 1 and Oct_20_17[i][3] >= 10:
K.append(Oct_20_17[i][0])
weight.append(1.0 / abs(Oct_20_17[i][1]-Oct_20_17[i][2]))
#weight.append(1.0)
price_actual.append(0.5 * (Oct_20_17[i][1] + Oct_20_17[i][2]))
T.append(175)

if data == 7:
for i in range(len(Jan_19_18)):
if abs(Jan_19_18[i][1]-Jan_19_18[i][2]) <= 1 and Jan_19_18[i][3] >= 10:
K.append(Jan_19_18[i][0])
weight.append(1.0 / abs(Jan_19_18[i][1]-Jan_19_18[i][2]))
#weight.append(1.0)
price_actual.append(0.5 * (Jan_19_18[i][1] + Jan_19_18[i][2]))
T.append(266)

initial_parameters = np.array([0.8, -0.5])
K = np.asarray(K)
T = np.asarray(T)
price_actual = np.asarray(price_actual)
weight = np.asarray(weight)
xdata.append(T)
xdata.append(K)
xdata = np.asarray(xdata)

####
def BS_func(xdata, sigma):
r = 0.0375
S0 = 160.29
K = xdata[1]
T = xdata[0]*1.0/360
d1 = (np.log(S0 * 1.0 / K) + (r + 0.5 * sigma ** 2) * T) * 1.0 / (sigma * np.sqrt(T))
d2 = d1 - sigma * np.sqrt(T)
C_model = S0 * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
#print(C_model)
return C_model

####
def optionPrice1(xdata, sigma, betaPara):
s = 160.29  ## The price of Apple Stock at time: 3:00 pm, date: 04/28/2017
r = 0.0375  ## Yield at 04/28/2017
T = xdata[0] / 360
K = xdata[1]
n = 100
m = 1000
s_max = 300
ds = s_max * 1.0 / m
position = round(s / ds)
```

```
##Construct A_star which is a m*m matrix
A = np.zeros((m, m))
si = np.linspace(1, m, m) * ds

for i in range(1, len(A) - 1):
A[i][i - 1] = 0.5 * ((sigma ** 2) * (si[i] ** (2 * betaPara + 2))
/ (ds ** 2) - (r * si[i]) / ds)
A[i][i] = -1.0 * (sigma ** 2) * (si[i] ** (2 * betaPara + 2))
/ (ds ** 2) - r
A[i][i + 1] = 0.5 * ((sigma ** 2) * (si[i] ** (2 * betaPara + 2))
/ (ds ** 2) + (r * si[i]) / ds)
A[0][0] = -1.0 * (sigma ** 2) * (si[0] ** (2 * betaPara + 2))
/ (ds ** 2) - r
A[0][1] = 0.5 * ((sigma ** 2) * (si[0] ** (2 * betaPara + 2))
/ (ds ** 2) + (r * si[0]) / ds)
A[-1][-2] = - r * s_max * 1.0 / ds
A[-1][-1] = r * s_max * 1.0 / ds - r


model_value = []
for times in range(len(K)):
dt = T[times] * 1.0 / n
## CrankNicolson Scheme
A_star_1 = inv(np.eye(len(A)) - 0.5*dt*A)
A_star_2 = np.eye(len(A)) + 0.5*dt*A
A_star = np.dot(A_star_1, A_star_2)

## Implicit Scheme
#A_star = inv(np.eye(len(A)) - dt*A)

## Explicit Scheme
#A_star = np.eye(len(A)) + dt*A

# Option price V
V = np.zeros((n+1, m+1))

# Set initial data
for i in range(m + 1):
V[0][i] = max(i * ds - K[times], 0)

# Set interior data and higher boundary data
for i in range(1, n + 1):
V[i][1:] = np.dot(A_star, V[i - 1][1:])
model_value.append(V[n][position])
#print(model_value)
model_value = np.asarray(model_value)
return model_value


parameters1 = optimization.curve_fit(optionPrice1, xdata, price_actual,
 initial_parameters, 1.0/weight,
bounds=([0., -1], [np.inf, 0.]))[0]
parameters2 = optimization.curve_fit(BS_func, xdata, price_actual,
initial_parameters[0], 1.0/weight,  bounds=(0., np.inf))[0]
stop_BM_Ite = timeit.default_timer()
print("Calibrated by CEV model with initial sigma= "+str(initial_parameters[0])
+", initial beta= "+str(initial_parameters[1])
+":"
+"\nThe result is: sigma= "+str(parameters1[0]) + " ,
beta= "+str(parameters1[1]))
print("\n")
print("Calibrated by BS Formula with initial sigma= "
+str(initial_parameters[0])+": "
+"\nThe result is: sigma= "+str(parameters2))
print("\n")
print("Time consuming: "+str(stop_BM_Ite-start_BM_Ite)+" s")

#print(optionPrice1(xdata,0.2,0))
#print(BS_func(xdata, 0.2))
#print(K)
```

A.3 Barrier Option

```
import numpy as np
from prettytable import PrettyTable

def CEV(s,K,B,type):
sigma = 0.105516623093
beta = -4.47444*np.power(10, -14)
r = 0.0375
n = 5000
T = 1
dt = 1.0*T/ 360
payoff_array = []

if type == "up_and_out":
for j in range(n):
hit_barrier = False
St = []
St.append(s)
## Normal variable
Z = np.random.standard_normal(size=360-1)
for i in range(1, 360):
St_i = St[i-1] + r*St[i-1]*dt + sigma*np.sqrt(dt)*Z[i-1]*(St[i-1]**(beta + 1)) \
+ 0.5*(sigma**2)*(beta + 1)*(dt*(Z[i-1]**2) - dt)*(St[i-1]**(2*beta+1))
if (i+1)%30==0:  # Monthly Monitor
if St_i > B: ## Up and out option
hit_barrier = True
break  # If St goes above the barrier, stop by setting time
St.append(St_i)
if hit_barrier == False:
payoff = max(St[-1] - K, 0)*np.exp(-r*T)
payoff_array.append(payoff)
else:
payoff_array.append(0)
```

```
price = sum(payoff_array) * 1.0 / n

elif type == "up_and_in":
for j in range(n):
hit_barrier = True
St = []
St.append(s)
## Normal variable
Z = np.random.standard_normal(size=360-1)
for i in range(1, 360):
St_i = St[i-1] + r*St[i-1]*dt + sigma*np.sqrt(dt)*Z[i-1]*(St[i-1]**(beta + 1)) \
+ 0.5*(sigma**2)*(beta + 1)*(dt*(Z[i-1]**2) - dt)*(St[i-1]**(2*beta+1))
if (i+1)%30==0:  # Monthly Monitor
if St_i >= B: ## Up and in option
hit_barrier = False
St.append(St_i)
if hit_barrier == False:
payoff = max(St[-1] - K, 0)*np.exp(-r*T)
payoff_array.append(payoff)
else:
payoff_array.append(0)
price = sum(payoff_array) * 1.0 / n

elif type == "down_and_out":
for j in range(n):
hit_barrier = False
St = []
St.append(s)
## Normal variable
Z = np.random.standard_normal(size=360-1)
for i in range(1, 360):
St_i = St[i-1] + r*St[i-1]*dt + sigma*np.sqrt(dt)*Z[i-1]*(St[i-1]**(beta + 1)) \
+ 0.5*(sigma**2)*(beta + 1)*(dt*(Z[i-1]**2) - dt)*(St[i-1]**(2*beta+1))
if (i+1)%30==0:  # Monthly Monitor
if St_i < B: ## Down and out option
hit_barrier = True
break  # If St goes above the barrier, stop by setting time
St.append(St_i)
if hit_barrier == False:
payoff = max(St[-1] - K, 0)*np.exp(-r*T)
payoff_array.append(payoff)
else:
payoff_array.append(0)
price = sum(payoff_array) * 1.0 / n

elif type == "down_and_in":
for j in range(n):
hit_barrier = True
St = []
St.append(s)
## Normal variable
Z = np.random.standard_normal(size=360-1)
for i in range(1, 360):
St_i = St[i-1] + r*St[i-1]*dt + sigma*np.sqrt(dt)*Z[i-1]*(St[i-1]**(beta + 1)) \
+ 0.5*(sigma**2)*(beta + 1)*(dt*(Z[i-1]**2) - dt)*(St[i-1]**(2*beta+1))
if (i+1)%30==0:  # Monthly Monitor
if St_i <= B: ## Down and in option
```

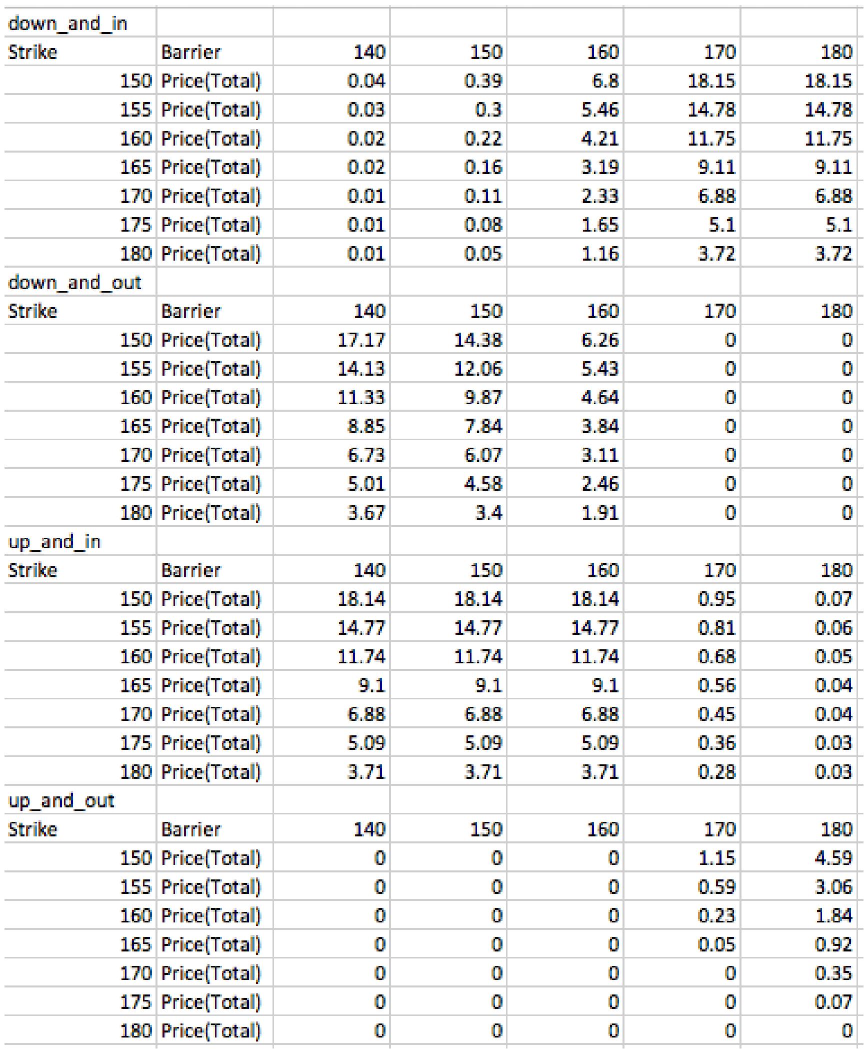
```
hit_barrier = False
St.append(St_i)
if hit_barrier == False:
payoff = max(St[-1] - K, 0)*np.exp(-r*T)
payoff_array.append(payoff)
else:
payoff_array.append(0)
price = sum(payoff_array) * 1.0 / n
return price

def optionType(type):
K = np.array([150,155,160,165,170,175,180])
B = np.array([140,150,160,170,180])
print("Type is: "+ type)
x = PrettyTable(["Value", "B = 140", "B = 150", "B = 160", "B = 170", "B = 180"])
x.align["Value"] = "l"
x.padding_width = 1
for i in range(len(K)):
x.add_row(["K = " + str(K[i]), CEV(160.3,K[i],B[0],type), CEV(160.3,K[i],B[1],type),
CEV(160.3, K[i], B[2], type), CEV(160.3,K[i],B[3],type), CEV(160.3,K[i],B[4],type)])
print(x)
optionType("down_and_out")
```

Appendix B

Bloomberg Data

B.1 Model: Local Volatility



down_and_in						
Strike	Barrier	140	150	160	170	180
150	Price(Total)	0.04	0.39	6.8	18.15	18.15
155	Price(Total)	0.03	0.3	5.46	14.78	14.78
160	Price(Total)	0.02	0.22	4.21	11.75	11.75
165	Price(Total)	0.02	0.16	3.19	9.11	9.11
170	Price(Total)	0.01	0.11	2.33	6.88	6.88
175	Price(Total)	0.01	0.08	1.65	5.1	5.1
180	Price(Total)	0.01	0.05	1.16	3.72	3.72
down_and_out						
Strike	Barrier	140	150	160	170	180
150	Price(Total)	17.17	14.38	6.26	0	0
155	Price(Total)	14.13	12.06	5.43	0	0
160	Price(Total)	11.33	9.87	4.64	0	0
165	Price(Total)	8.85	7.84	3.84	0	0
170	Price(Total)	6.73	6.07	3.11	0	0
175	Price(Total)	5.01	4.58	2.46	0	0
180	Price(Total)	3.67	3.4	1.91	0	0
up_and_in						
Strike	Barrier	140	150	160	170	180
150	Price(Total)	18.14	18.14	18.14	0.95	0.07
155	Price(Total)	14.77	14.77	14.77	0.81	0.06
160	Price(Total)	11.74	11.74	11.74	0.68	0.05
165	Price(Total)	9.1	9.1	9.1	0.56	0.04
170	Price(Total)	6.88	6.88	6.88	0.45	0.04
175	Price(Total)	5.09	5.09	5.09	0.36	0.03
180	Price(Total)	3.71	3.71	3.71	0.28	0.03
up_and_out						
Strike	Barrier	140	150	160	170	180
150	Price(Total)	0	0	0	1.15	4.59
155	Price(Total)	0	0	0	0.59	3.06
160	Price(Total)	0	0	0	0.23	1.84
165	Price(Total)	0	0	0	0.05	0.92
170	Price(Total)	0	0	0	0	0.35
175	Price(Total)	0	0	0	0	0.07
180	Price(Total)	0	0	0	0	0

FIGURE B.1: LocalVolatility data from Blommberg.

Bibliography

Sturm, Stephan (2017). "COMPUTATIONAL METHODS OF FINANCIAL MATHEMATICS". In: URL: https://piazza.com/wpi/spring2017/ma573/resources.