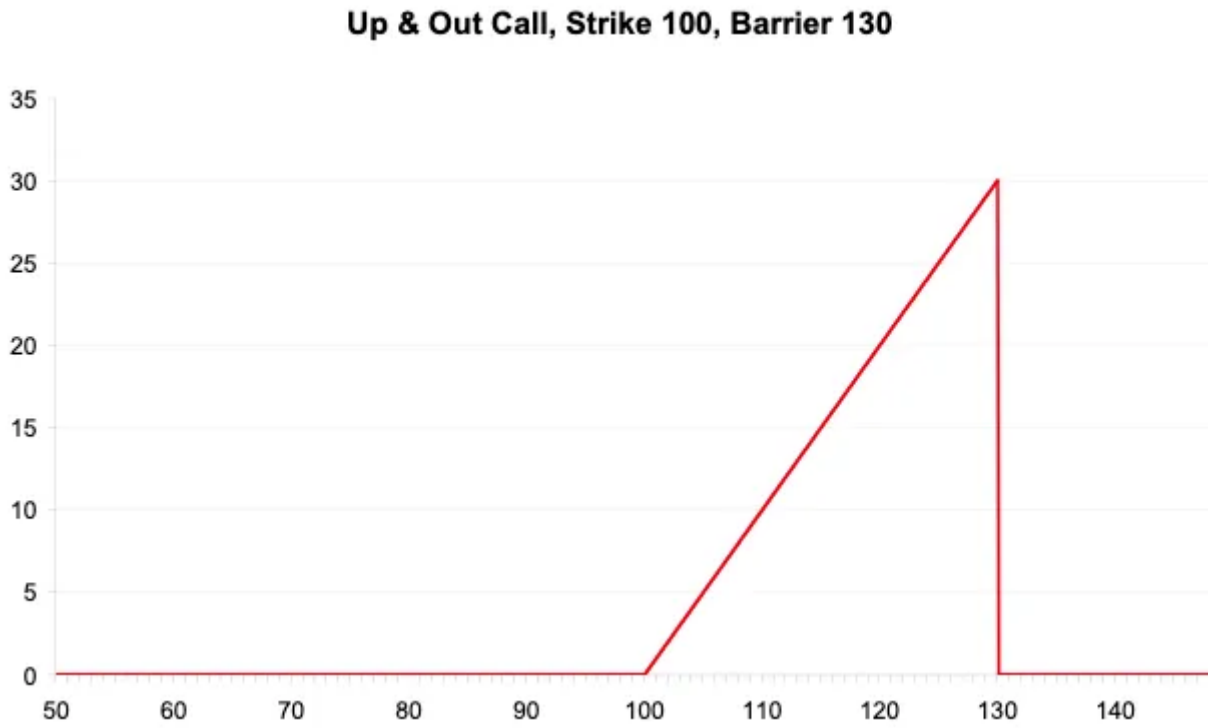
Published in The Quant Journey

Andrea Chello Follow

May 16, 2022 · 6 min read · ✨ · Listen

Save

Pricing Barrier Options using Monte Carlo Simulation in Python



Modelling Exotic Options

When modelling exotic options, one has to make a fundamental decision very early in the process: should you model the option in a continuous-time, Black-Scholes type of model, or in a binomial model.

- Generally many exotic options are initially priced via a binomial model, and then at some point traders figure out a closed-form pricing model.
- Sometimes, it turns out that no closed form solution is ever found.
- Indeed, for certain highly path-dependent options, one cannot even work backwards in a lattice, instead one must use a Monte Carlo method to value the option.

1. Barrier Options

Barrier options are options that have a **payout** that is **dependent** not only on the **terminal stock price**, but also **depend upon whether the stock attains some "barrier" during the life of the option**.

If the price of the underlying does not rise above the barrier level, the option acts like any other option — it gives the holder the right but not the obligation to exercise their call or put option at the strike price on or before the expiration date specified in the contract.

Two general kinds: Knock-out and Knock-in options

1.1 Knock-out options

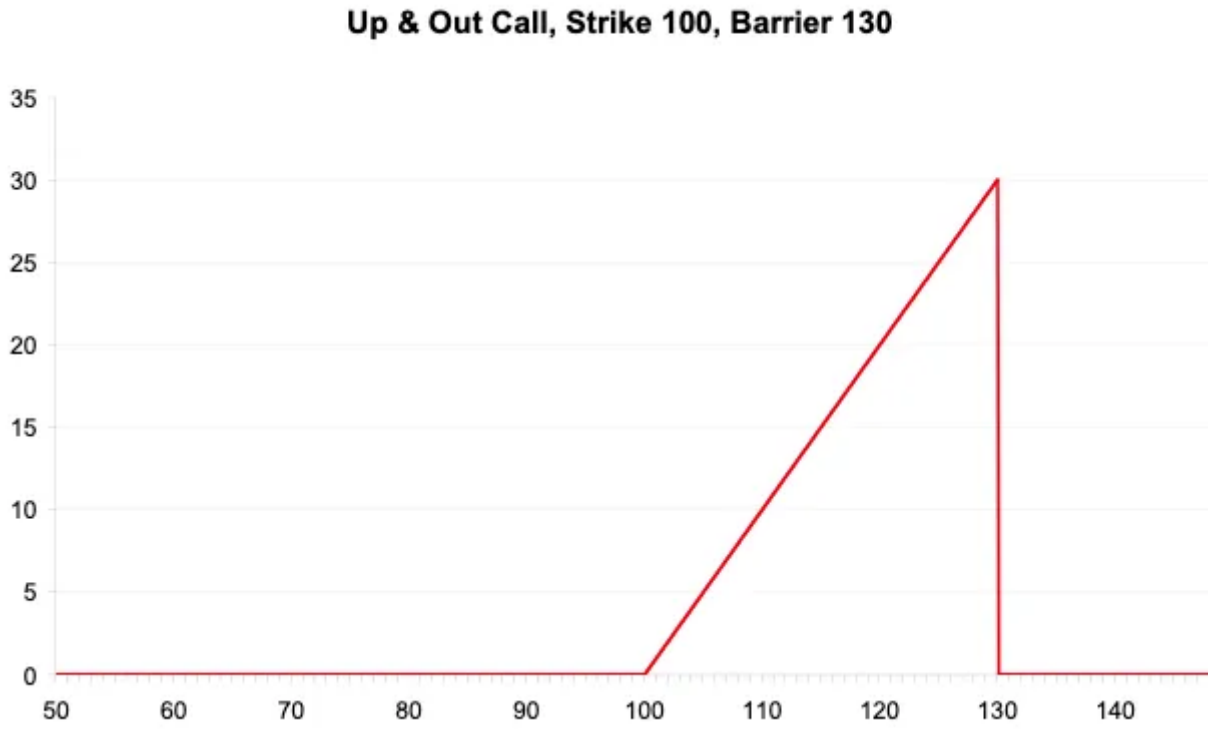
The **option ceases to exist if the stock reaches a given barrier** during the options life.

There are four types of Knock-out options.

Let, K be the strike of option, B the barrier level and St the share price at time t.

a. Up-and-out Call

$$C = (S_T - K)^+ \mathbb{I}_{\left\{ \max_{t\in[0,T]} S_t<B \right\}} = \begin{cases} S_T - K, & \text{if } \max_{t\in[0,T]} S_t \leq B \\ 0, & \text{if } \max_{t\in[0,T]} S_t > B \end{cases}$$



b. Up-and-out Put

$$P = (K - S_T)^+ \mathbb{I}_{\left\{ \max_{t\in[0,T]} S_t<B \right\}} = \begin{cases} K - S_T, & \text{if } \max_{t\in[0,T]} S_t < B \\ 0, & \text{if } \max_{t\in[0,T]} S_t \geq B \end{cases}$$

c. Down-and-out Call

$$C = (S_T - K)^+ \mathbb{I}_{\left\{ \min_{t\in[0,T]} S_t>B \right\}} = \begin{cases} S_T - K, & \text{if } \min_{t\in[0,T]} S_t > B \\ 0, & \text{if } \min_{t\in[0,T]} S_t \leq B \end{cases}$$

d. Down-and-out Put

$$P = (K - S_T)^+ \mathbb{I}_{\left\{ \min_{t\in[0,T]} S_t > B \right\}} = \begin{cases} K - S_T, & \text{if } \min_{t\in[0,T]} S_t \geq B \\ 0, & \text{if } \min_{t\in[0,T]} S_t < B \end{cases}$$

1.2 Knock-in Options

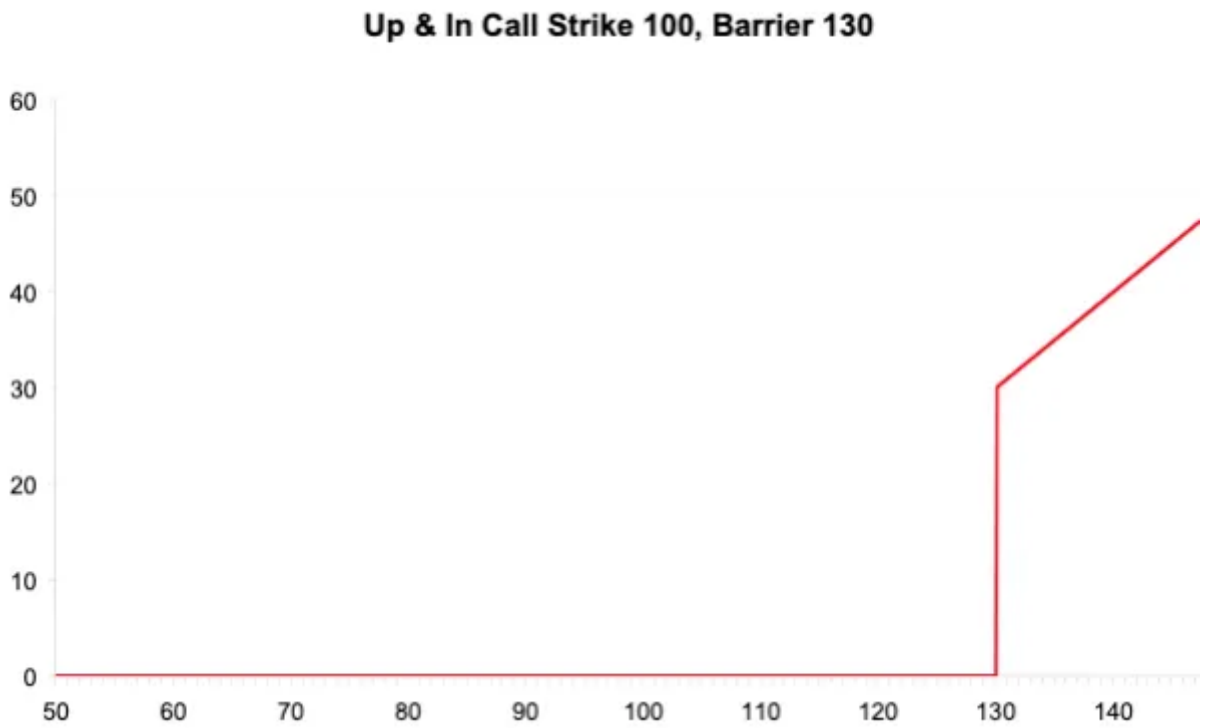
The **option comes into being only if the stock reaches a given barrier** during its life.

There are four types of Knock-in options

Let, *K* be the strike of option, *B* the barrier level and *St* the share price at time t.

a. Up-and-in Call

$$C = (S_T - K)^+ \mathbb{I}_{\left\{ \max_{t\in[0,T]} S_t > B \right\}} = \begin{cases} S_T - K, & \text{if } \max_{t\in[0,T]} S_t > B \\ 0, & \text{if } \max_{t\in[0,T]} S_t \leq B \end{cases}$$



b. Up-and-in Put

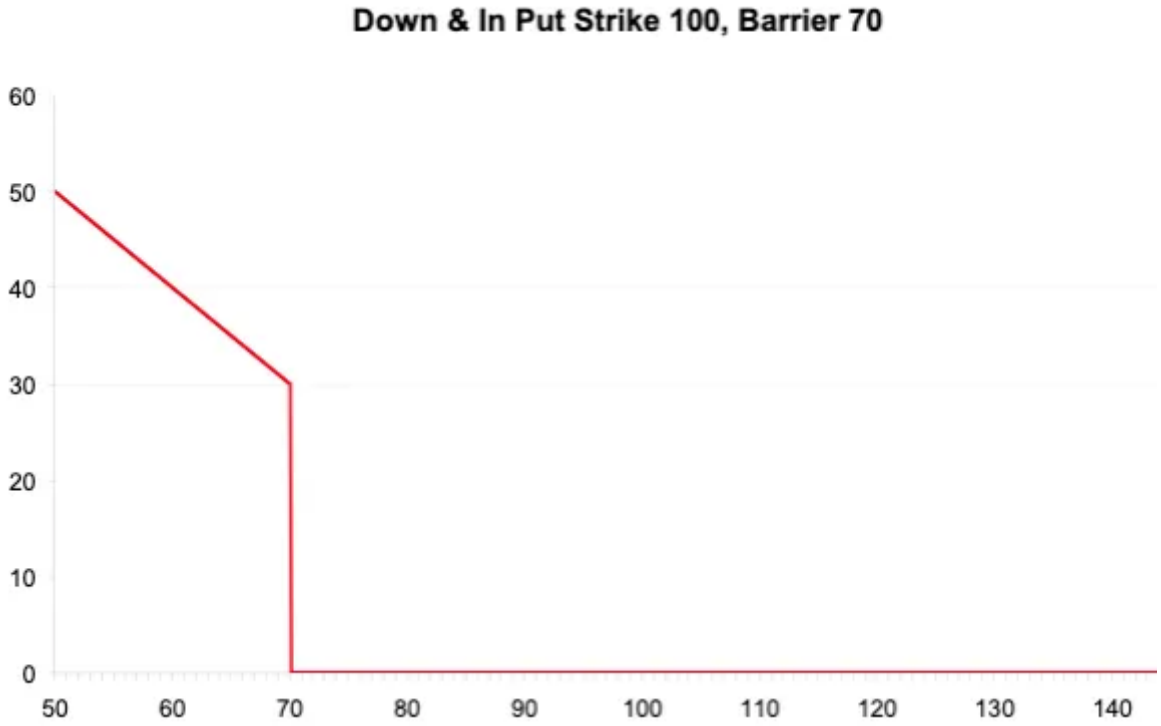
$$P = (K - S_T)^+ \mathbb{I}_{\left\{ \max_{t\in[0,T]} S_t > B \right\}} = \begin{cases} K - S_T, & \text{if } \max_{t\in[0,T]} S_t \geq B \\ 0, & \text{if } \max_{t\in[0,T]} S_t < B \end{cases}$$

c. Down-and-in Call

$$C = (S_T - K)^+ \mathbb{I}_{\left\{ \min_{t\in[0,T]} S_t < B \right\}} = \begin{cases} S_T - K, & \text{if } \min_{t\in[0,T]} S_t \leq B \\ 0, & \text{if } \min_{t\in[0,T]} S_t > B \end{cases}$$

d. Down-and-in Put

$$P = (K - S_T)^+ \mathbb{I}_{\left\{ \min_{t\in[0,T]} S_t < B \right\}} = \begin{cases} K - S_T, & \text{if } \min_{t\in[0,T]} S_t < B \\ 0, & \text{if } \min_{t\in[0,T]} S_t \geq B \end{cases}$$



2. Pricing a European up-and-out Barrier Call Option

As a barrier option is path-dependent we have to simulate the paths for the underlying share and for the couterparty's firm value.

Payoff of a European up-and-out call option:

$$v(S_T) = (S_T - K)^+ \quad \max_{t \in [0,T]} S_t < B$$

The option payoff is dependent on the history of the share price, and not just on its terminal value. Therefore, the entire share price paths need to be simulated to estimated the price of this option.

a. Assumptions

```python
import numpy as np
import pandas as pd
from scipy.stats import norm
import matplotlib.pyplot as plt
import math as m
import random as r

#Market information
risk_free = 0.08

#share specific information
S_0 = 100
sigma = 0.3
strike = 100
dT=1/12
current_time=0

#European up and out call option information
T = 1
barrier = 150

#firm specific information
V_0 = 200
sigma_firm = 0.25
debt = 175
correlation = 0.2  #correlation between counter party and stock
recovery_rate = 0.25
```

barrier.py hosted with ❤ by GitHub view raw

b. Defining the Functions

```python
def terminal_shareprice(S_0, risk_free_rate, sigma, Z, T):
    """
    Generates the terminal share price given some random normal values, z
    """
    # It returns an array of terminal stock prices.
    return S_0*np.exp((risk_free_rate-sigma**2/2)*T+sigma*np.sqrt(T)*Z)

def discounted_call_payoff(S_T, K, risk_free_rate, T):
    """
    Function for evaluating the discounted payoff of a call option
    in the Monte Carlo Estimation
    """
    # It returns an array which has the value of the call for each terminal stock price
    return np.exp(-risk_free_rate*T)*np.maximum(S_T - K, 0)
```

barrier1.py hosted with ❤ by GitHub view raw

c. Simulating Paths

Simulate paths for the underlying share and for the counterparty's firm value using sample sizes of 1000, 2000, …, 50000. We will do monthly simulations for the lifetime of the option.

```python
np.random.seed(0)
num_simulations = 1000
# the number of steps represent the times we simulate the process, with each step compr
# 1000*i steps so we get simulations from 1000 to 50000
num_steps = 50
num_of_months = 13
# terminal price is an array of size 13 to account for the 12 months plus initial value
# it is timed by the number of steps
term_val = [[None]*num_of_months]*num_steps

# initialise the monte carlo value, estimates and std as empty array of size number of
mbarrier_val = [None]*num_steps
mbarrier_estimates = [None]*num_steps
mbarrier_std = [None]*num_steps
```

barrier2.py hosted with ❤ by GitHub view raw

d. Monte Carlo Simulation

Given that:

$$\max_{t\in[0,T]} S_t < B$$

```python
for i in range(1,num_steps+1):
    # fill out the first value with our initial stock price
    term_val[i-1][0] = np.full((num_simulations*i), S_0)

    for j in range (1,num_of_months):
        # update current month to reflect the monthly simulation we are currently in
        current_month = (j-1)/12
        norm_array = norm.rvs(size = num_simulations*i)
        term_val[i-1][j] = terminal_shareprice(term_val[i-1][j-1],risk_free,sigma,norm_

    # Compute discounted barrier Price of the option
    mbarrier_val[i-1] = discounted_call_payoff(term_val[i-1][12],strike,risk_free,T-cur

    # use the above formula to calculate the values of the barrier option
    ## get array of booleans for when stock is knocked out or not
    knock_out_array = (np.max(term_val[i-1],axis = 0) < barrier)
    ## times it by the value of the previously calculated barrier option
    mbarrier_val[i-1] = mbarrier_val[i-1] * knock_out_array

    # compute mean and standard deviation of entire path
    mbarrier_estimates[i-1] = np.mean(mbarrier_val[i-1])
    mbarrier_std[i-1] = np.std(mbarrier_val[i-1]/np.sqrt(i*num_simulations))
```

barrier3.py hosted with ❤ by GitHub view raw

```
The Monte Carlo Price of the Barrier Option is: 6.714095415887313
```

3. Pricing a European up-and-in Barrier Call Option

The conditions of an up and in barrier are the opposite of those of an up and out, namely if the price touches the barrier the payoff is the vanilla call option, else the payoff is zero.

We will use the same assumptions as above.

$$C = (S_T - K)^+ \mathbb{I}_{\left\{ \max_{t\in[0,T]} S_t > B \right\}} = \begin{cases} S_T - K, & \text{if } \max_{t\in[0,T]} S_t > B \\ 0, & \text{if } \max_{t\in[0,T]} S_t \leq B \end{cases}$$

a. Simulating Paths

```
np.random.seed(0)
num_simulations = 1000
# the number of steps represent the times we simulate the process, with each step compr
# 1000*i steps so we get simulations from 1000 to 50000
num_steps = 50
num_of_months = 13
# terminal price is an array of size 13 to account for the 12 months plus initial value
# it is timed by the number of steps
term_val = [[None]*num_of_months]*num_steps

# initialise the monte carlo value, estimates and std as empty array of size number of
mbarrier_val = [None]*num_steps
mbarrier_estimates = [None]*num_steps
mbarrier_std = [None]*num_steps
```

barrier4.py hosted with ❤ by GitHub view raw

b. Monte Carlo Simulation

Given that:

$$\max_{t\in[0,T]} S_t > B$$

```python
for i in range(1,num_steps+1):
    # fill out the first value with our initial stock price
    term_val[i-1][0] = np.full((num_simulations*i), S_0)

    for j in range (1,num_of_months):
        # update current month to reflect the monthly simulation we are currently in
        current_month = (j-1)/12
        norm_array = norm.rvs(size = num_simulations*i)
        term_val[i-1][j] = terminal_shareprice(term_val[i-1][j-1],risk_free,sigma,norm_

    # Compute discounted barrier Price of the option
    mbarrier_val[i-1] = discounted_call_payoff(term_val[i-1][12],strike,risk_free,T-cur

    # use the above formula to calculate the values of the barrier option
    ## get array of booleans for when stock is knocked out or not
    knock_in_array = (np.max(term_val[i-1],axis = 0) > barrier)
    ## times it by the value of the previously calculated barrier option
    mbarrier_val[i-1] = mbarrier_val[i-1] * knock_in_array

    # compute mean and standard deviation of entire path
    mbarrier_estimates[i-1] = np.mean(mbarrier_val[i-1])
    mbarrier_std[i-1] = np.std(mbarrier_val[i-1]/np.sqrt(i*num_simulations))
```

barrier5.py hosted with ❤ by GitHub view raw

```
The Monte Carlo Price of the Barrier Option is: 8.97919712295608
```

4. Pricing a European up-and-out Barrier Call Option using Different Strike Prices

We can look at how different strike prices affect the price of our up and out call option.

We will use the same assumptions as above.

a. Define Strike Prices

```
strike_1, strike_2, strike_3, strike_4, strike_5, strike_6 = 85, 90,
95, 105, 110, 115
```

b. Monte Carlo Simulation for each Strike Price

```
def get_barrier_price(strike_num):
    for i in range(1,num_steps+1):
        # fill out the first value with our initial stock price
        term_val[i-1][0] = np.full((num_simulations*i), S_0)

        for j in range (1,num_of_months):
            # update current month to reflect the monthly simulation we are currently i
            current_month = (j-1)/12
            norm_array = norm.rvs(size = num_simulations*i)
            term_val[i-1][j] = terminal_shareprice(term_val[i-1][j-1],risk_free,sigma,r

        # Compute discounted barrier Price of the option
        mbarrier_val[i-1] = discounted_call_payoff(term_val[i-1][12],strike,risk_free,T

        # use the above formula to calculate the values of the barrier option
        ## get array of booleans for when stock is knocked out or not
        knock_out_array = (np.max(term_val[i-1],axis = 0) < barrier)
        ## times it by the value of the previously calculated barrier option
        mbarrier_val[i-1] = mbarrier_val[i-1] * knock_out_array

        # compute mean and standard deviation of entire path
        mbarrier_estimates[i-1] = np.mean(mbarrier_val[i-1])
        mbarrier_std[i-1] = np.std(mbarrier_val[i-1]/np.sqrt(i*num_simulations))

    return np.mean(mbarrier_estimates)
```

barrier6.py hosted with ❤ by GitHub view raw

```
strike_1_price, strike_2_price, strike_3_price, strike_4_price,
strike_5_price, strike_6_price =\
get_barrier_price(strike_1), get_barrier_price(strike_2),
get_barrier_price(strike_3),\
get_barrier_price(strike_4), get_barrier_price(strike_5),
get_barrier_price(strike_6)

result = pd.Series({
    "Strike":"Option Price",
    strike_1:strike_1_price,
    strike_2:strike_2_price,
    strike_3:strike_3_price,
    strike_4:strike_4_price,
    strike_5:strike_5_price,
    strike_6:strike_6_price,
})
```

```
result
```

Strike	Option Price
85	6.69435
90	6.71506
95	6.70997
105	6.69369
110	6.6998
115	6.71404

5. Determine Monte Carlo estimates of both the default-free value of the option and the Credit Valuation Adjustment (CVA)

This will be done for the European up and out Call Option

5.1 Monte Carlo estimates of the default-free value a European up and out call option

a. Simulating Paths

```
np.random.seed(0)
num_simulations = 1000
# the number of steps represent the times we simulate the process, with each step compr
# 1000*i steps so we get simulations from 1000 to 50000
num_steps = 50
num_of_months = 13
# terminal price is an array of size 13 to account for the 12 months plus initial value
# it is timed by the number of steps
term_val = [[None]*num_of_months]*num_steps

# initialise the monte carlo value, estimates and std as empty array of size number of
mbarrier_val = [None]*num_steps
mbarrier_estimates = [None]*num_steps
mbarrier_std = [None]*num_steps
```

barrier7.py hosted with ❤ by GitHub view raw

b. Monte Carlo Simulation

```python
for i in range(1,num_steps+1):
    # fill out the first value with our initial stock price
    term_val[i-1][0] = np.full((num_simulations*i), S_0)

    for j in range (1,num_of_months):
        # update current month to reflect the monthly simulation we are currently in
        current_month = (j-1)/12
        norm_array = norm.rvs(size = num_simulations*i)
        term_val[i-1][j] = terminal_shareprice(term_val[i-1][j-1],risk_free,sigma,norm_

    # Compute discounted barrier Price of the option
    mbarrier_val[i-1] = discounted_call_payoff(term_val[i-1][12],strike,risk_free,T-cur

    # use the above formula to calculate the values of the barrier option
    ## get array of booleans for when stock is knocked out or not
    knock_out_array = (np.max(term_val[i-1],axis = 0) < barrier)

    ## times it by the value of the previously calculated barrier option
    mbarrier_val[i-1] = mbarrier_val[i-1] * knock_out_array

    # compute mean and standard deviation of entire path
    mbarrier_estimates[i-1] = np.mean(mbarrier_val[i-1])
    mbarrier_std[i-1] = np.std(mbarrier_val[i-1]/np.sqrt(i*num_simulations))
```

barrier8.py hosted with ❤ by GitHub view raw

The Monte Carlo Barrier estimates will be:

```
[6.952903571459723, 6.921354605151922, 6.428390188772714,
6.796464372056431, 6.888491223681098, 6.728588325311477,
6.851646107411101, 6.685434028989336, 6.819660887328652,
6.792256147585757, 6.647290313106, 6.682728656516728,
6.572349635748065, 6.7120992019035155, 6.785680985387171,
6.859352080154061, 6.728533320841484, 6.745003680283005,
6.821036081733498, 6.6340333850791415, 6.79835458338198,
6.714408806881992, 6.7049032062626415, 6.737164257931305,
6.695890047217236, 6.703102989803265, 6.623574720504978,
6.539047617606773, 6.675675942959946, 6.701703770585718,
6.663319202262758, 6.735782007044407, 6.738659716742189,
6.697518309013903, 6.772972008713773, 6.697788073328364,
6.661770733796724, 6.635861228746139, 6.637641769493973,
6.782577290202915, 6.652177213449349, 6.809159045245774,
6.7282021117635535, 6.634124021795217, 6.707470662573413,
6.690197406334271, 6.701925561714455, 6.785891508531344,
6.68895307453393, 6.718582176378565]
```

5.2 Monte Carlo Estimate of CVA

a. Create Empty Arrays to store Values

```
#empty array to store values
term_stock_val =  [[None]*num_of_months]*num_steps
mbarrier_val2 = [None]*num_steps
term_firm_val = [None]*num_steps
amount_lost = [None]*num_steps
cva_estimates = [None]*num_steps
cva_std = [None]*num_steps
```

b. Monte Carlo estimates

```python
for i in range(1,num_steps+1):
    # fill out the first value with our initial stock price value and firm value
    term_stock_val[i-1][0] = (np.full((num_simulations*i), S_0),np.full((num_simulation

    for j in range (1,num_of_months):
        # update current month to reflect the monthly simulation we are currently in
        current_month = (j-1)/12

        corr_matrix = np.array([[1,correlation],[correlation,1]])
        norm_matrix = norm.rvs(size = np.array([2,num_simulations*i]))
        corr_norm_matrix = np.matmul(np.linalg.cholesky(corr_matrix), norm_matrix)
        term_stock_val[i-1][j] = terminal_shareprice(term_stock_val[i-1][j-1],risk_free

    # Compute discounted barrier Price of the option
    mbarrier_val2[i-1] = discounted_call_payoff(term_stock_val[i-1][12],strike,risk_fre

    #compute terminal firm value
    term_firm_val[i-1] = terminal_shareprice(V_0,risk_free,sigma_firm,corr_norm_matrix|

    #compute amount lost
    amount_lost[i-1] = np.exp(-risk_free*T)*(1-recovery_rate)*(term_firm_val[i-1] < deb

    # get array of booleans for when stock is knocked out or not
    knock_out_array2 = (np.amax(term_stock_val[i-1], axis = 0)< barrier)

    # multiply it by the value of the previously calculated amount lost
    amount_lost[i-1] = amount_lost[i-1] * knock_out_array2


    # compute mean and standard deviation
    cva_estimates[i-1] = np.mean(amount_lost[i-1])
    cva_std[i-1] = np.std(amount_lost[i-1]/np.sqrt(i*num_simulations))
```

barrier9.py hosted with ❤ by GitHub view raw

The final CVA estimates are:

```
[0.8185955907825436, 0.8060917023320449, 0.6555800915794667,
0.6430895286931251, 0.69718661445995, 0.7521798793754507,
0.7379097223289557, 0.6956888323915302, 0.7556984149437391,
0.7925888998861441, 0.757663393083224, 0.7193302788083542,
0.6742925963005665, 0.7551662091354194, 0.7014317754132186,
0.6929955452403777, 0.7047599655316822, 0.7422354965810541,
0.7574585502740152, 0.7324616012000625, 0.7732018357582887,
0.7421355749315399, 0.7268178798326702, 0.7518313524212152,
0.7586650167404645, 0.7612140368880638, 0.7190054332441704,
0.7258404425790702, 0.7271708138706607, 0.7359538276074311,
```

```
0.7306742982804011, 0.7581262654675898, 0.7519503274185494,
0.7270582333402852, 0.7645955507062125, 0.7452306924132879,
0.7187636549643519, 0.7544789244253182, 0.7348785733916363,
0.734532286738978, 0.7300901221377352, 0.7292117411130734,
0.7544530414154849, 0.7513131096862078, 0.7455075178544784,
0.7368038175249791, 0.7766280989684483, 0.7263975400481919,
0.7565373157411249, 0.7418640735729483]
```

6. Calculate the Monte Carlo estimates for the price of the option incorporating counterparty risk

The counterparty risk is given by the default-free price minus the CVA

```
#create arrays for monte carlo estimates of default free value and
CVA
arr1 = np.array(mbarrier_estimates)
arr2 = np.array(cva_estimates)

#find monte carlo estimates for price of option with counter-party
risk
risky_price_estimates = np.subtract(arr1,arr2)

print(risky_price_estimates)
```

```
[6.13430798 6.1152629  5.7728101  6.15337484 6.19130461 5.97640845
 6.11373639 5.9897452  6.06396247 5.99966725 5.88962692 5.96339838
 5.89805704 5.95693299 6.08424921 6.16635653 6.02377336 6.00276818
 6.06357753 5.90157178 6.02515275 5.97227323 5.97808533 5.98533291
 5.93722503 5.94188895 5.90456929 5.81320718 5.94850513 5.96574994
 5.9326449  5.97765574 5.98670939 5.97046008 6.00837646 5.95255738
 5.94300708 5.8813823  5.9027632  6.048045   5.92208709 6.0799473
 5.97374907 5.88281091 5.96196314 5.95339359 5.92529746 6.05949397
 5.93241576 5.9767181 ]
```

The Quant Journey

This Article is published on **The Quant Journey**, an initiative started to take the readers along side this journey towards learning about quantitative finance topics.

Finance Programming Computational Finance Options Python