Asset Pricing - Homework 6

November 28, 2021

1 Asset Pricing - Homework 6

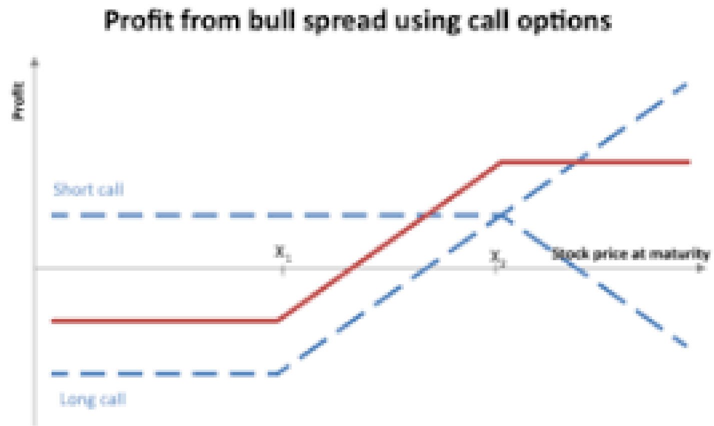
Group: José Barretto, Daniel Deutsch, Ziyad Bekkaoui.

[1]:
```
import numpy as np
import pandas as pd
from scipy.stats import norm
from scipy.stats import describe
import matplotlib.pyplot as plt
import numpy as np
import statistics
import math
```

[2]:
```
# Matplotlib styles
plt.style.use('ggplot')
plt.rcParams.update({
    'figure.figsize': (15, 4),
    'axes.prop_cycle': plt.cycler(color=["#4C72B0", "#C44E52", "#55A868",␣
 ↪"#8172B2", "#CCB974", "#64B5CD"]),
    'axes.facecolor': "#EAEAF2"
})
```

2 Question 1

The derivatives we choosed to work is a bull a spread. The bull spread is derivates wich allow you at maturity to earn the maximum of the minimum between the difference underlying price minus the second strike and zero for the maximum comparaison. The payoff is ilustrated in the picture below.



This derivatives can be replicated as a long call low strike and a short call position on the high strike. To have a closed pricing formula start from the Black & Scholes option formula and modify in order to have a closed formula for the bull spread.

Bellow we have B&S equation for european option call.

$$\mathrm{C(S,t)} = \mathrm{N(d_1)S} - \mathrm{N(d_2)Ke}^{-rt}$$

$$\mathrm{d_1} = \frac{1}{\sigma\sqrt{\mathrm{t}}}\left[\ln\left(\frac{S}{K}\right) + t\left(r + \frac{\sigma^2}{2}\right)\right]$$

$$\mathrm{d_2} = \frac{1}{\sigma\sqrt{\mathrm{t}}}\left[\ln\left(\frac{S}{K}\right) + t\left(r - \frac{\sigma^2}{2}\right)\right]$$

$$N(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} \mathrm{e}^{-\frac{1}{2}z^2}dz$$

The interest of the call spread is that it's a cheaper derivatives than a vanilla call option, financially speaking, because your selling another option, which allow you to earn the prime but to also to be totally hedged, as the EDP, all the greeks can be summed which have the effect to neutralise themself at maturity.

Mathematicaly speaking, the call spread is cheaper because your expected payoff is lower. We can constat this on the distribution curve below where we represent the expected value of a stock in one year, with a volatility of 25 % with Price = 100 at t = 0, a low strike = 100 and a high strike = 120

We can constat that the integer between of the distribution function is lower between 100 and 120 than between 100 and infinity.

To conclude this first part, we developed a call spread pricer, using a closed formula.

```
[3]: def bull_spread(S, K1,K2, T, r, sigma):
    """
    INPUTS:
      - S: spot price
      - K1: strike price low
      - K2: strike price high
      - T: time to maturity
      - r: interest rate
      - sigma: volatility of underlying asset

    OUTPUT:
      - bull_spread
    """

    d1 = (np.log(S / K1) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = (np.log(S / K1) + (r - 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))

    long_call = (S * norm.cdf(d1, 0.0, 1.0) - K1 * np.exp(-r * T) * norm.
 ↪cdf(d2, 0.0, 1.0))

    d1 = (np.log(S / K2) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = (np.log(S / K2) + (r - 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))

    short_call = (S * norm.cdf(d1, 0.0, 1.0) - K2 * np.exp(-r * T) * norm.
 ↪cdf(d2, 0.0, 1.0))


    bull_spread = long_call - short_call
    return bull_spread

bull_spread(100,100,120,1,0.01,0.20)
```

```
[3]: 6.092669293471825
```

3 Question 2

In this part we work on the pricing of a best of option. This exotic option pay the same payoff than a vanilla option but on the best performing underlying on the basket. in this type of option there is one more sensitivity that enter into account for the pricing, the correlation betwwen the basket's stock. In our simulation we will not take this parameter into acount and assume that there is no correlation in between the underlyings. In fact, being long best of option make you short correlation because this position make you taking advantages of one underlying that will escape the group to maximize the performance.

[4]:
```python
def simulate_price(P0, r, sigma, n, T, N):
    """
    INPUTS:
      - P0: initial price
      - r: interest rate
      - sigma: volatility
      - n: number of timesteps per simulation
      - T: total runtime
      - N: total number of simulations

    OUTPUTS:
      - prices: dataframe with dimension (N, n). Each row contains a simulation␣
↪of the price evolution
    """

    # generate empty dataframe
    prices = pd.DataFrame(columns=[k*T/n for k in range(0, n+1)], index=np.
↪arange(N))

    # store initial value in first column
    prices[0] = P0

    # run N simulations
    for i in range(N):

        # run N timesteps
        for k in range(1, n+1):

            # calculate timestep size
            t = k*T/n

            # draw u_k from random distribution
            u = norm.rvs((r-0.5*(sigma**2))*T/n, np.sqrt((sigma**2)*T/n))

            # calculate and store estimated price based on last estimation
            prices.loc[i, k*T/n] = prices.loc[i, (k-1)*T/n]*np.exp(u)

    return prices
```

4 Estimate Fair Price

4.1 Simulates Prices

In this section we simulate prices for three different options. For that, we shall first set some global simulation parameters.

```
[5]: # Global Simulation parameter to modify
n = 100
T = 1
N = 100
P0 = 1000
r = 0.01

# define volatilities
sigma1 = 0.15
sigma2 = 0.2
sigma3 = 0.05
```

4.1.1 Option 1

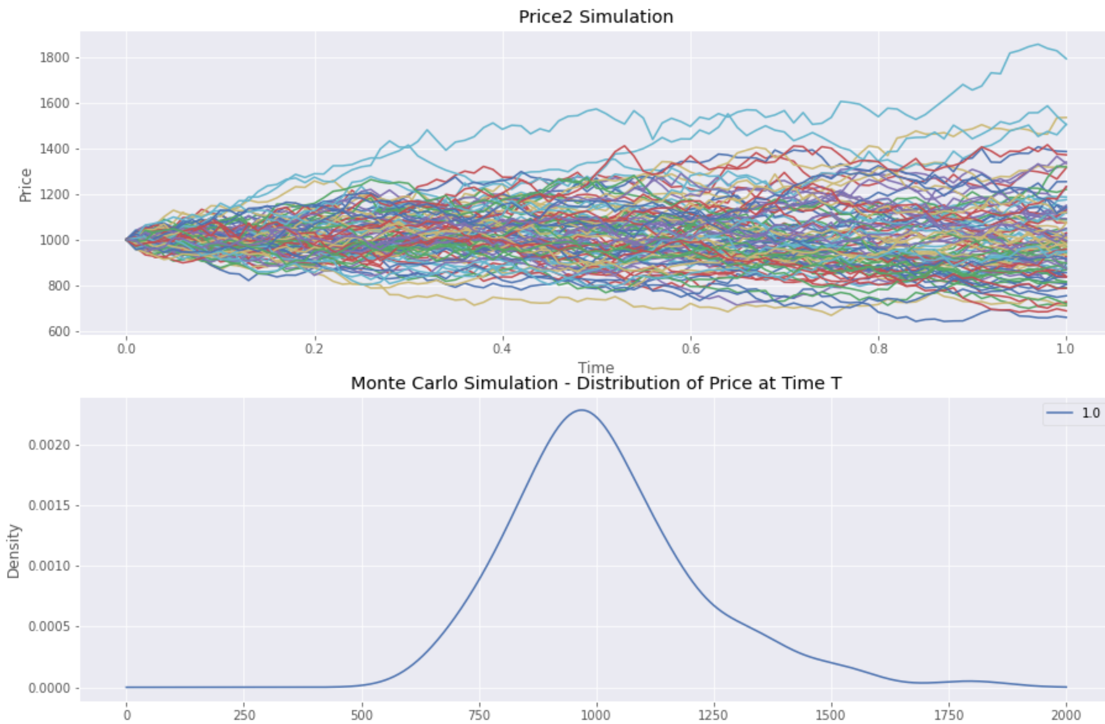
```
[6]: # Simulates prices for and calculates its payoff
prices1 = simulate_price(P0, r, sigma2, n, T, N)
prices1['payoff'] = prices1.iloc[:, -1] - prices1.iloc[:, 0]
prices1['payoff'] = prices1['payoff'].clip(lower=0)

# Creates subplots
fig, axs = plt.subplots(2, 1, figsize=(15, 10))

# Plots the simulated prices
for i, row in prices1.iterrows():
  axs[0].plot(prices1.columns[:-1], row[:-1].astype(float))
axs[0].set_xlabel("Time")
axs[0].set_ylabel("Price")
axs[0].set_title("Price2 Simulation")

# Plots the results distribution
prices1.iloc[:, -2].plot(kind='density', legend = True, ind=np.linspace(0,␣
↪2000, 1000))
plt.title('Monte Carlo Simulation - Distribution of Price at Time T')
plt.show()

plt.show()
```



4.1.2 Option 2

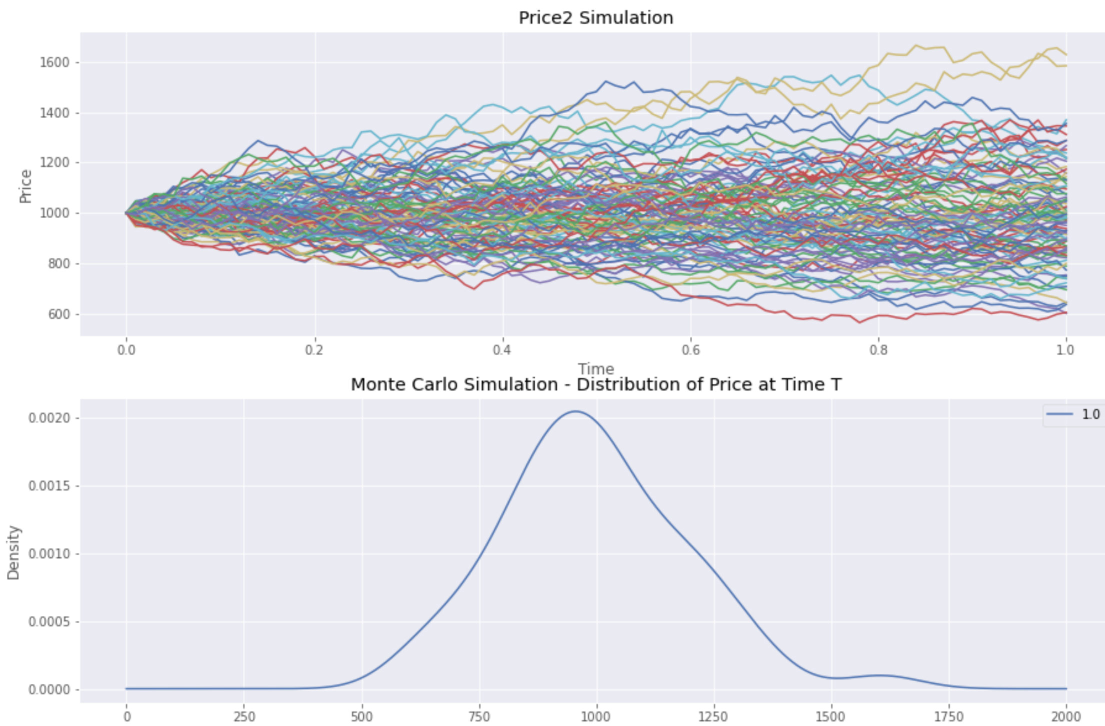
[7]:
```python
# Simulates prices for and calculates its payoff
prices2 = simulate_price(P0, r, sigma2, n, T, N)
prices2['payoff'] = prices2.iloc[:, -1] - prices2.iloc[:, 0]
prices2['payoff'] = prices2['payoff'].clip(lower=0)

# Creates subplots
fig, axs = plt.subplots(2, 1, figsize=(15, 10))

# Plots the simulated prices
for i, row in prices2.iterrows():
  axs[0].plot(prices2.columns[:-1], row[:-1].astype(float))
axs[0].set_xlabel("Time")
axs[0].set_ylabel("Price")
axs[0].set_title("Price2 Simulation")

# Plots the results distribution
prices2.iloc[:, -2].plot(kind='density',legend = True,ind=np.linspace(0, 2000, 1000))
plt.title('Monte Carlo Simulation - Distribution of Price at Time T')
plt.show()
```

```
plt.show()
```



4.1.3 Option 3

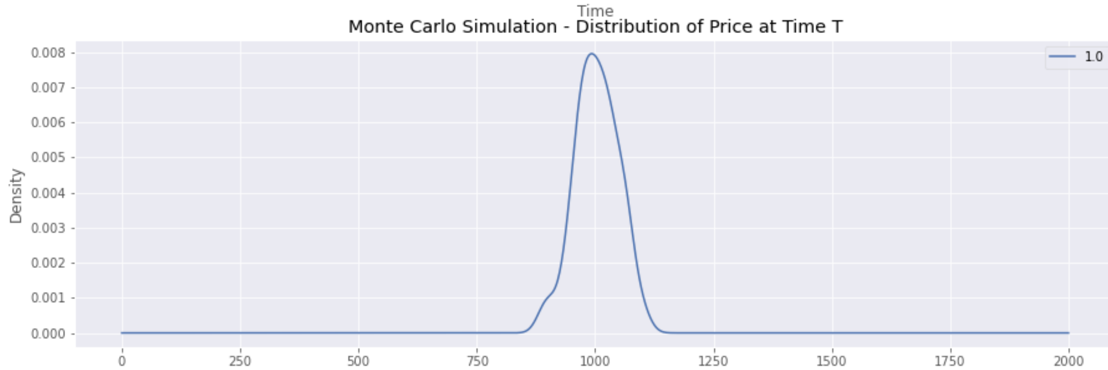
[8]:
```python
# Simulates prices for and calculates its payoff
prices3 = simulate_price(P0, r, sigma3, n, T, N)
prices3['payoff'] = prices3.iloc[:, -1] - prices3.iloc[:, 0]
prices3['payoff'] = prices3['payoff'].clip(lower=0)

# Creates subplots
fig, axs = plt.subplots(2, 1, figsize=(15, 10))

# Plots the simulated prices
for i, row in prices3.iterrows():
  axs[0].plot(prices3.columns[:-1], row[:-1].astype(float))
axs[0].set_xlabel("Time")
axs[0].set_ylabel("Price")
axs[0].set_title("Price3 Simulation")

# Plots payoff
# Plots the results distribution
```

```python
prices3.iloc[:, -2].plot(kind='density',legend = True,ind=np.linspace(0, 2000,␣
 ↪1000))
plt.title('Monte Carlo Simulation - Distribution of Price at Time T')
plt.show()
```



[9]:
```python
prices3.iloc[:,-2:]
```

[9]:

	1.0	payoff
0	1033.043895	33.043895
1	1024.117705	24.117705
2	994.02309	0
3	974.330656	0
4	1015.142862	15.142862
..	…	…
95	1076.789802	76.789802
96	987.885066	0
97	1022.501157	22.501157
98	974.18739	0
99	903.224278	0

[100 rows x 2 columns]

4.2 Obtains the Fair Price

The fair price is obtained via the mean of the max payoff between the simulations.

[10]:

```
# Obtains the fair price
payoffs = pd.concat([prices1['payoff'], prices2['payoff'], prices3['payoff']],
 ↪axis=1, names=['prices1', 'prices2', 'prices3'])
payoffs['max_payoff'] = payoffs.max(axis=1)
fair_price = payoffs['max_payoff'].mean()

# We actualise the price by the risk free rate
fair_price *= np.exp(-r*T)
fair_price
```

[10]:

```
148.57582838099736
```

As ususal in derivatives, in work in term of percentages, that mean that the nominal on which we will apply the payoff in the nominal of the option, all our Price at t = 0 are 100 in order to get payoff and a simulation in percentage. The price obtained is therefore a percentage of the nominal.