GEORGIA INSTITUTE OF TECHNOLOGY

Title: ISyE6785 Interim Project 2

Author(s): Weifeng Lyu
Instructor: Dr. Shijie Deng

TABLE OF CONTENTS

1. Introduction

1.1 Problem

ISyE 6785: Mini-project 2 (Due on 7/12/2018)
Note: For all the computations, please report the configuration of your computer (e.g. CPU type and speed, size of RAM) and the computing CPU time in seconds.

1. A spread call option on two assets S1, S2 with strike price K pays off max (S1 - S2 - K, 0); a spread put option on two assets S1, S2 with strike price K pays off max (K – (S1 - S2), 0). Suppose the asset price processes are given by the following correlated Geometric Brownian motions.

$$dS_t^1 = (r - \delta_1)S_t^1 dt + \sigma_1 S_t^1 dW_t^1$$

$$dS_t^2 = (r - \delta_2)S_t^2 dt + \rho\sigma_2 S_t^2 dW_t^1 + \sqrt{1-\rho^2}\,\sigma_2 S_t^2 dW_t^2$$

where r = 4.5%, δ1 = 2%, σ1 = 20%, δ2 = 0.5%, σ2 = 25%, ρ = 0.3. The current prices are S1 = $100, S2 =$95.
a. Implement a simulation approach to simulate 100 price paths of (S1, S2) from time 0 to T = 0.5 years.
b. Implement the Longstaff and Schwartz (RFS, 2001) algorithm to price a standard American-style put option on S1 with strike price K = 90 and maturity time T = 0.5 years.
c. (Bonus question, optional) Price an American-style spread call option with strike price K = 15 and the maturity time 0.5 years.

1.2 Computer Configuration

Manufacturer: Dell
Model: Inspiron 7559 Signature Edition
Processor: Intel® Core™ i7-6700HQ CPU @ 2.60GHz 2.60GHz
Installed Memory (RAM): 8.00GB (7.88 usable)
System Type: 64-bit Operating System, x64-based processor

2. Technology Review

2.1 Algorithm Review

(a) Geometric Brownian motions formula is given by the problem

$$dS_t^1 = (r - \delta_1)S_t^1 dt + \sigma_1 S_t^1 dW_t^1$$

$$dS_t^2 = (r - \delta_2)S_t^2 dt + \rho\sigma_2 S_t^2 dW_t^1 + \sqrt{1-\rho^2}\,\sigma_2 S_t^2 dW_t^2$$

(b) Antithetic variates: Equipped with a basis for evaluating potential efficiency improvements, we can now consider specific variance reduction techniques.

Consider the problem of computing the Black-Scholes price of a European call option on a no-dividend stock.

$$S_T^{(i)} = S_0 e^{(r-(1/2)\sigma^2)T + \sigma\sqrt{T}Z_i}, \quad i = 1, \ldots, n,$$

Based on n replications, an unbiased estimator of the price of an option with strike K is given by

$$\hat{C} = \frac{1}{n}\sum_{i=1}^{n} C_i \equiv \frac{1}{n}\sum_{i=1}^{n} e^{-rT}\max\{0, S_T^{(i)} - K\}.$$

the method of antithetic variates4 is based on the observation that if Zi has a standard normal distribution, then so does -Zi, The price obtained from (2) with Zi replaced by -Zi is thus a valid sample from the terminal stock price distribution. Similarly, each

$$\tilde{C}_i = e^{-rT}\max\{0, \tilde{S}_T^{(i)} - K\}$$

is an unbiased estimator of the option price, as is, therefore,

$$\hat{C}_{AV} = \frac{1}{n}\sum_{i=1}^{n} \frac{C_i + \tilde{C}_i}{2}.$$

(c) The LSM approach uses least squares to approximate the conditional expectation function at t. We work backwards since the path of cash flows C(w,s;t,T) generated by the option is defined recursively.

$$dS = rSdt + \sigma s dZ,$$

As a set of basis functions, we use a constant and the first three Laguerre polynomials as given in the followings:

$$L_0(X) = \exp(-X/2),$$

$$L_1(X) = \exp(-X/2)\ (1 - X),$$

$$L_2(X) = \exp(-X/2)\ (1 - 2X + X^2/2),$$

Thus we regress discounted realized cash flows on a constant and three nonlinear functions of the stock price. Since we use linear regression to estimate the conditional expectation function, it is straightforward to add additional basis functions as explanatory variables in the regression as needed. Using more than three basis functions are sufficient to obtain effective convergence of the algorithm in this example.

$$L_n(X) = \exp(-X/2)\frac{e^X}{n!}\frac{d^n}{dX^n}(X^n e^{-X}).$$

The partial differential equation satisfied by the price P(S,t) is

$$(\sigma^2 S^2/2)P_{SS} + rSP_S - rP + P_T = 0,$$

2.2 Algorithm Parameter

1. Black-Scholes Model

S0 : int or float, initial asset value
K : int or float, strike
T : int or float, time to expiration as a fraction of one year
r : int or float, continuously compounded risk free rate, annualized
sigma : int or float, continuously compounded standard deviation of returns
kind : str, {'call', 'put'}, default 'call', type of option

2. Monte Carlo Methods

S0 : int or float, initial asset value
K : int or float, strike
T : int or float, time to expiration as a fraction of one year
M : int, grid or granularity for time (in number of total points)
r : int or float, continuously compounded risk free rate, annualized
i : int, number of simulations
sigma : int or float, continuously compounded standard deviation of returns
delta : int or float
rho: int or float
kind : str, {'call', 'put'}, default 'call', type of option
seed: int, random seed to generate simulations

3. Project Architecture

3.1 Python Implementation

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import scipy.stats
import time
import math

class Option(object):
    """Compute European option value, greeks, and implied
volatility.

    Parameters
    ==========
    S0 : int or float
        initial asset value
    K : int or float
        strike
    T : int or float
        time to expiration as a fraction of one year
    M : int
        grid or granularity for time (in number of total points)
    r : int or float
        continuously compounded risk free rate, annualized
    i : int
        number of simulations
    sigma : int or float
        continuously compounded standard deviation of returns
    delta : int or float
    rho: int or float
    kind : str, {'call', 'put'}, default 'call'
        type of option

    Resources
    =========

http://www.thomasho.com/mainpages/?download=&act=model&file=256
    """

    def __init__(self, S0, K, T, M, r, delta, sigma, i,
kind='call'):
        if kind.istitle():
            kind = kind.lower()
```

```
        if kind not in ['call', 'put']:
            raise ValueError('Option type must be \'call\' or
\'put\'')

        self.S0 = S0
        self.K = K
        self.T = T
        self.M = int(M)
        self.r = r
        self.delta = delta
        self.sigma = sigma
        self.i = int(i)
        self.kind = kind
        self.time_unit = self.T / float(self.M)
        self.discount = np.exp(-self.r * self.time_unit)

        self.d1 = ((np.log(self.S0 / self.K)
                + (self.r + 0.5 * self.sigma ** 2) * self.T)
                / (self.sigma * np.sqrt(self.T)))
        self.d2 = ((np.log(self.S0 / self.K)
                + (self.r - 0.5 * self.sigma ** 2) * self.T)
                / (self.sigma * np.sqrt(self.T)))

        # Several greeks use negated terms dependent on option
type
        # For example, delta of call is N(d1) and delta put is
N(d1) - 1
        self.sub = {'call' : [0, 1, -1], 'put' : [-1, -1, 1]}

    def value(self):
        """Compute option value."""
        return (self.sub[self.kind][1] * self.S0
                * scipy.stats.norm.cdf(self.sub[self.kind][1] *
self.d1)
                + self.sub[self.kind][2] * self.K * np.exp(-
self.r * self.T)
                * scipy.stats.norm.cdf(self.sub[self.kind][1] *
self.d2))

    def AmericanPutPrice(self, seed):
        """ Returns Monte Carlo price matrix rows: time columns:
price-path simulation """
        np.random.seed(seed)
        path = np.zeros((self.M + 1, self.i), dtype=np.float64)
        path[0] = self.S0
        for t in range(1, self.M + 1):
            rand = np.random.standard_normal(int(self.i / 2))
```

```
            rand = np.concatenate((rand, -rand))
            path[t] = (path[t - 1] * np.exp((self.r -
self.delta) * self.time_unit + self.sigma *
np.sqrt(self.time_unit) * rand))

        """ Returns the inner-value of American Option """
        if self.kind == 'call':
            payoff = np.maximum(path - self.K, np.zeros((self.M
+ 1, self.i), dtype=np.float64))
        else:
            payoff = np.maximum(self.K - path, np.zeros((self.M
+ 1, self.i), dtype=np.float64))

        value = np.zeros_like(payoff)
        value[-1] = payoff[-1]
        for t in range(self.M - 1, 0, -1):
            regression = np.polyfit(path[t], value[t + 1] *
self.discount, 5)
            continuation_value = np.polyval(regression, path[t])
            value[t] = np.where(payoff[t] > continuation_value,
payoff[t], value[t + 1] * self.discount)

        return np.sum(value[1] * self.discount) / float(self.i)

# This is a function simulating the price path for a Geometric
Brownian Motion price model
def gen_paths(S0_1, S0_2, r, delta_1, delta_2, sigma_1, sigma_2,
rho, T, M, I):
    dt = float(T) / M
    path_1 = np.zeros((M + 1, I), np.float64)
    path_2 = np.zeros((M + 1, I), np.float64)
    path_1[0] = S0_1
    path_2[0] = S0_2
    for t in range(1, M + 1):
        rand_1 = np.random.standard_normal(I)
        rand_2 = np.random.standard_normal(I)
        path_1[t] = path_1[t - 1] * np.exp((r - delta_1) * dt +
                                           sigma_1 * np.sqrt(dt)
* rand_1)
        path_2[t] = path_2[t - 1] * np.exp((r - delta_2) * dt +
                                           rho * sigma_2 *
np.sqrt(dt) * rand_1 +
                                           np.sqrt(1-rho**2) *
sigma_2 * np.sqrt(dt) * rand_2)
    return [path_1,path_2]

def gen_paths_antithetic(S0_1, S0_2, r, delta_1, delta_2,
```

```
sigma_1, sigma_2, rho, T, M, I):
    dt = float(T) / M
    path_1 = np.zeros((2*M + 1, I), np.float64)
    path_2 = np.zeros((2*M + 1, I), np.float64)
    path_1[0] = S0_1
    path_2[0] = S0_2
    for t in range(1, M + 1):
        rand_1 = np.random.standard_normal(I)
        rand_2 = np.random.standard_normal(I)
        rand_anti_1 = -1.0 * rand_1  # antithetic variates
        rand_anti_2 = -1.0 * rand_2  # antithetic variates
        path_1[2 * t] = path_1[2 * (t - 1)] * np.exp((r -
delta_1) * dt +
                                                     sigma_1 *
np.sqrt(dt) * rand_1)
        path_1[2 * t - 1] = path_1[max(2 * t - 3, 0)] *
np.exp((r - delta_1) * dt +

sigma_1 * np.sqrt(dt) * rand_anti_1)
        path_2[2*t] = path_2[2*(t - 1)] * np.exp((r - delta_2) *
dt +
                                                 rho * sigma_2 *
np.sqrt(dt) * rand_1 +
                                                 np.sqrt(1 - rho
** 2) * sigma_2 * np.sqrt(dt) * rand_2)
        path_2[2*t-1] = path_2[max(2*t - 3, 0)] * np.exp((r -
delta_2) * dt +
                                                       rho *
sigma_2 * np.sqrt(dt) * rand_anti_1 +
                                                       np.sqrt(1
- rho ** 2) * sigma_2 * np.sqrt(dt) * rand_anti_2)
    return [path_1,path_2]

def hist_comp(dist1, dist2, lgnd, bin_num):
    hist_start = min(min(dist1), min(dist2))
    hist_end = max(max(dist1), max(dist2))
    bin_vec = np.linspace(hist_start, hist_end, bin_num)
    plt.hist([dist1, dist2], color=['r','g'],
label=[lgnd[0],lgnd[1]], alpha=0.8, bins=bin_vec)
    plt.legend(loc='upper right')
    plt.show()

#1
S0_1 = 100.0
S0_2 = 95.0
K = 90.0
r = 0.045
```

```
delta_1 = 0.02
sigma_1 = 0.2
delta_2 = 0.005
sigma_2 = 0.25
rho = 0.3
T = 0.5
M = 200 #252
i = 100
discount_factor = np.exp(-r * T)

seed = 84
## Closed-form option price
start_time = time.time()
call_option_1 = Option(S0_1, K, T, M, r, delta_1, sigma_1, i,
'call')
call_option_2 = Option(S0_2, K, T, M, r, delta_2, sigma_2, i,
'call')
print('B-S price for Asset 1: %f, time used: %f.' %
(call_option_1.value(), time.time()-start_time))
print('B-S price for Asset 2: %f, time used: %f.' %
(call_option_2.value(), time.time()-start_time))

## Set seed for a random number generator
start_time = time.time()
np.random.seed(seed)
[path1,path2] = gen_paths(S0_1, S0_2, r, delta_1, delta_2,
sigma_1, sigma_2, rho, T, M, i)
duration = time.time()-start_time

## Plot all sample paths
pd.DataFrame(path1).plot()
plt.xlabel('time')
plt.ylabel('price')
plt.title("Simulate 100 price paths of S1")
pd.DataFrame(path2).plot()
plt.xlabel('time')
plt.ylabel('price')
plt.title("Simulate 100 price paths of S2")
plt.show()


# Compute the value of a Call option
CallPayoffAverage_1 = np.average(np.maximum(0, path1[-1] - K))
CallPayoff_1 = discount_factor * CallPayoffAverage_1
print('MC estimator for Asset 1: %f, time used: %f.' %
(CallPayoff_1, duration))
CallPayoffAverage_2 = np.average(np.maximum(0, path2[-1] - K))
```

```
CallPayoff_2 = discount_factor * CallPayoffAverage_2
print('MC estimator for Asset 2: %f, time used: %f.' %
(CallPayoff_2, duration))

## Antithetic variate estimator
start_time = time.time()
[path_1_anti,path_2_anti] = gen_paths_antithetic(S0_1, S0_2, r,
delta_1, delta_2, sigma_1, sigma_2, rho, T, M, i)
CallPayoffAverage_1 = np.average(np.maximum(0, path_1_anti[-1] -
K))
CallPayoffAverage_tilda_1 = np.average(np.maximum(0,
path_1_anti[-2] - K))
CallPayoff_anti_1 = discount_factor *
(CallPayoffAverage_1+CallPayoffAverage_tilda_1)/2.0
CallPayoffAverage_2 = np.average(np.maximum(0, path_2_anti[-1] -
K))
CallPayoffAverage_tilda_2 = np.average(np.maximum(0,
path_2_anti[-2] - K))
CallPayoff_anti_2 = discount_factor *
(CallPayoffAverage_2+CallPayoffAverage_tilda_2)/2.0
mcav_time = time.time() - start_time
print('Antithetic variate estimator for path1: %f, Antithetic
variate time used: %f.' % (CallPayoff_anti_1,mcav_time))
print('Antithetic variate estimator for path2: %f, Antithetic
variate time used: %f.' % (CallPayoff_anti_2,mcav_time))

M = 100  # number of Monte Carlo estimators
MC_vec_1 = []
MCAV_vec_1 = []
MC_vec_2 = []
MCAV_vec_2 = []
best_j1 = 0
best_j2 = 0
best_diff = math.inf

#test different seed for Monte Carlo estimators
for j in range(M):
    np.random.seed(j+1)
    [path1,path2] = gen_paths(S0_1, S0_2, r, delta_1, delta_2,
sigma_1, sigma_2, rho, T, M, i)
    CallPayoffAverage_1 = np.average(np.maximum(0, path1[-1] -
K))
    CallPayoff_1 = discount_factor * CallPayoffAverage_1
    CallPayoffAverage_2 = np.average(np.maximum(0, path2[-1] -
K))
    CallPayoff_2 = discount_factor * CallPayoffAverage_2
    MC_vec_1.append(CallPayoff_1)
```

```
    MC_vec_2.append(CallPayoff_2)
    [path_1_anti, path_2_anti] = gen_paths_antithetic(S0_1,
S0_2, r, delta_1, delta_2, sigma_1, sigma_2, rho, T, M, i)
    CallPayoffAverage_1 = np.average(np.maximum(0, path_1_anti[-
1] - K))
    CallPayoffAverage_tilda_1 = np.average(np.maximum(0,
path_1_anti[-2] - K))
    CallPayoff_anti_1 = discount_factor * (CallPayoffAverage_1 +
CallPayoffAverage_tilda_1) / 2.0
    CallPayoffAverage_2 = np.average(np.maximum(0, path_1_anti[-
1] - K))
    CallPayoffAverage_tilda_2 = np.average(np.maximum(0,
path_2_anti[-2] - K))
    CallPayoff_anti_2 = discount_factor * (CallPayoffAverage_2 +
CallPayoffAverage_tilda_2) / 2.0
    MCAV_vec_1.append(CallPayoff_anti_1)
    MCAV_vec_2.append(CallPayoff_anti_2)
    diff1 = abs(CallPayoff_anti_1 - call_option_1.value())
    diff2 = abs(CallPayoff_anti_2 - call_option_2.value())
    if diff1 + diff2 < best_diff:
        best_diff = diff1 + diff2
        best_j = j

print('Best Seed for Asset S1 and S2:', best_j)

MC_mean_1 = np.average(MC_vec_1)
MC_std_1 = np.sqrt(np.var(MC_vec_1))
MC_mean_2 = np.average(MC_vec_2)
MC_std_2 = np.sqrt(np.var(MC_vec_2))

print('Naive MC estimator for Asset 1 mean: %f, standard
dev: %f.' % (MC_mean_1, MC_std_1))
print('Antithetic Variates MC estimator for Asset 1 mean: %f,
standard dev: %f.' % (np.average(MCAV_vec_1),
np.sqrt(np.var(MCAV_vec_1))))

print('Naive MC estimator for Asset 2 mean: %f, standard
dev: %f.' % (MC_mean_2, MC_std_2))
print('Antithetic Variates MC estimator for Asset 2 mean: %f,
standard dev: %f.' % (np.average(MCAV_vec_2),
np.sqrt(np.var(MCAV_vec_2))))

### Plot the histogram of the Monte Carlo estimators and the
Antithetic Variate estimators
hist_start = min(min(MC_vec_1), min(MCAV_vec_1))
hist_end = max(max(MC_vec_1), max(MCAV_vec_1))
bin_num = 40
```

```
bin_vec = np.linspace(hist_start, hist_end, bin_num)
plt.hist([MC_vec_1, MCAV_vec_1], color=['r','g'],
label=['MC','MCAV'], alpha=0.8, bins=bin_vec)
plt.xlabel('price')
plt.ylabel('number of sample')
plt.legend(loc='upper right')
plt.title("Price Sample Distribution of Naive MC estimator Vs.
Antithetic variate MC estimator of S1")
plt.show()
hist_start = min(min(MC_vec_2), min(MCAV_vec_2))
hist_end = max(max(MC_vec_2), max(MCAV_vec_2))
bin_num = 40
bin_vec = np.linspace(hist_start, hist_end, bin_num)
plt.hist([MC_vec_2, MCAV_vec_2], color=['r','g'],
label=['MC','MCAV'], alpha=0.8, bins=bin_vec)
plt.xlabel('price')
plt.ylabel('number of sample')
plt.legend(loc='upper right')
plt.title("Price Sample Distribution of Naive MC estimator Vs.
Antithetic variate MC estimator of S2")
plt.show()

#2 Price American Put Option
put_option_1 = Option(S0_1, K, T, M, r, delta_1, sigma_1, i,
'put')
mc_price = []
bs_price = []
best_diff = math.inf
for j in range(M):
    mc_put_option_1 = Option(S0_1, K, T, M, r, delta_1, sigma_1,
i, 'put').AmericanPutPrice(j)
    mc_price.append(mc_put_option_1)
    bs_price.append(put_option_1.value())
    diff = abs(mc_put_option_1 - put_option_1.value())
    if diff < best_diff:
        best_diff = diff
        best_seed = j

plt.plot(mc_price,label='MC')
plt.plot(bs_price,label='BS')
plt.xlabel('seed')
plt.ylabel('price')
plt.legend(loc='upper right')
plt.title("Black Scholes Price Vs. Monte Carlo Price Simulation
in Different Seed")
plt.show()
```

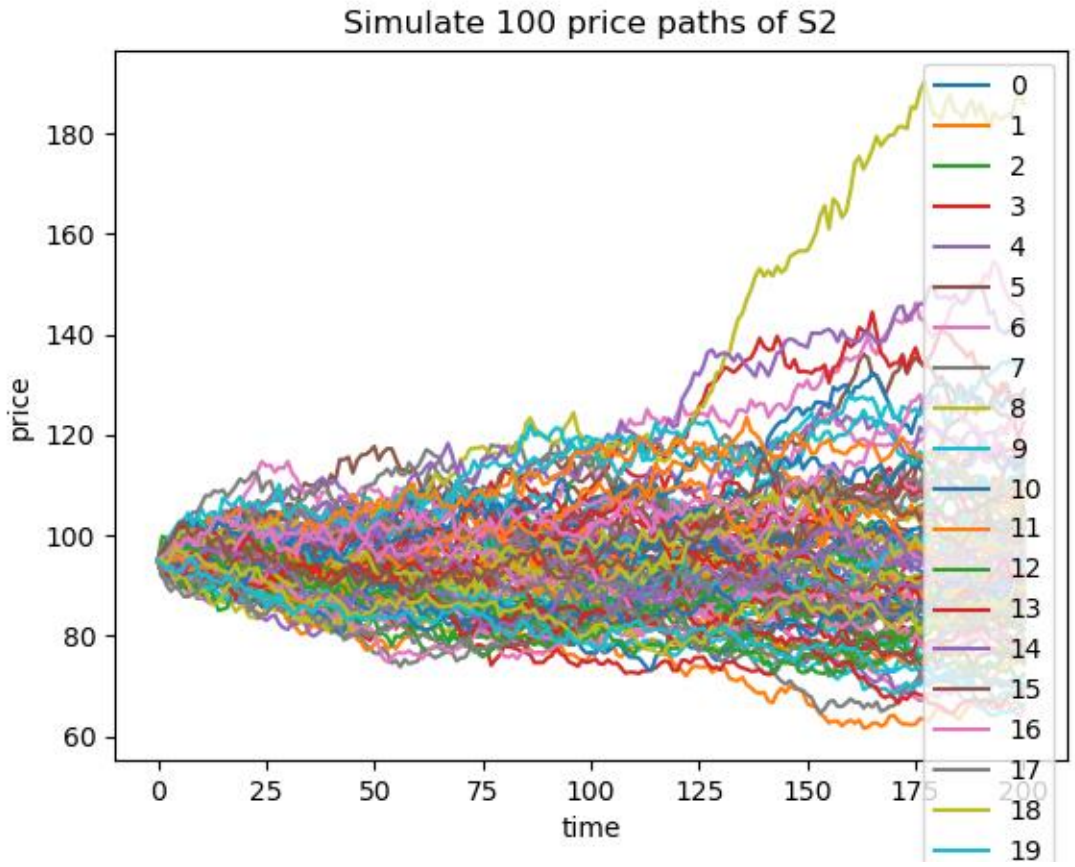
```
print('Best Seed of Put Option for Asset S1:', best_seed)
seed = best_seed
print('American Put Option Price (MC) for S1: %f' % Option(S0_1,
K, T, M, r, delta_1, sigma_1, i, 'put').AmericanPutPrice(seed))
print('American Put Option Price (BS) for S1: %f' %
put_option_1.value())

#3 Price American Call Option
K = 15
call_option_1 = Option(S0_1, K, T, M, r, delta_1, sigma_1, i,
'call')
mc_price = []
bs_price = []
best_diff = math.inf
for j in range(M):
    mc_call_option_1 = Option(S0_1, K, T, M, r, delta_1,
sigma_1, i, 'call').AmericanPutPrice(j)
    mc_price.append(mc_call_option_1)
    bs_price.append(call_option_1.value())
    diff = abs(mc_call_option_1 - call_option_1.value())
    if diff < best_diff:
        best_diff = diff
        best_seed = j

plt.plot(mc_price,label='MC')
plt.plot(bs_price,label='BS')
plt.xlabel('seed')
plt.ylabel('price')
plt.legend(loc='upper right')
plt.title("Black Scholes Price Vs. Monte Carlo Price Simulation
in Different Seed")
plt.show()

print('Best Seed of Call Option for Asset S1:', best_seed)
seed = best_seed
print ('American Call Option Price (MC): ', Option(S0_1, K, T,
M, r, delta_1, sigma_1, i, 'call').AmericanPutPrice(seed))
print ('American Call Option Price (BS): ', Option(S0_1, K, T,
M, r, delta_1, sigma_1, i, 'call').value())
```

3.2 Result of Sample Data and Discussion

a. Implement a simulation approach to simulate 100 price paths of (S1, S2) from time 0 to T = 0.5 years.

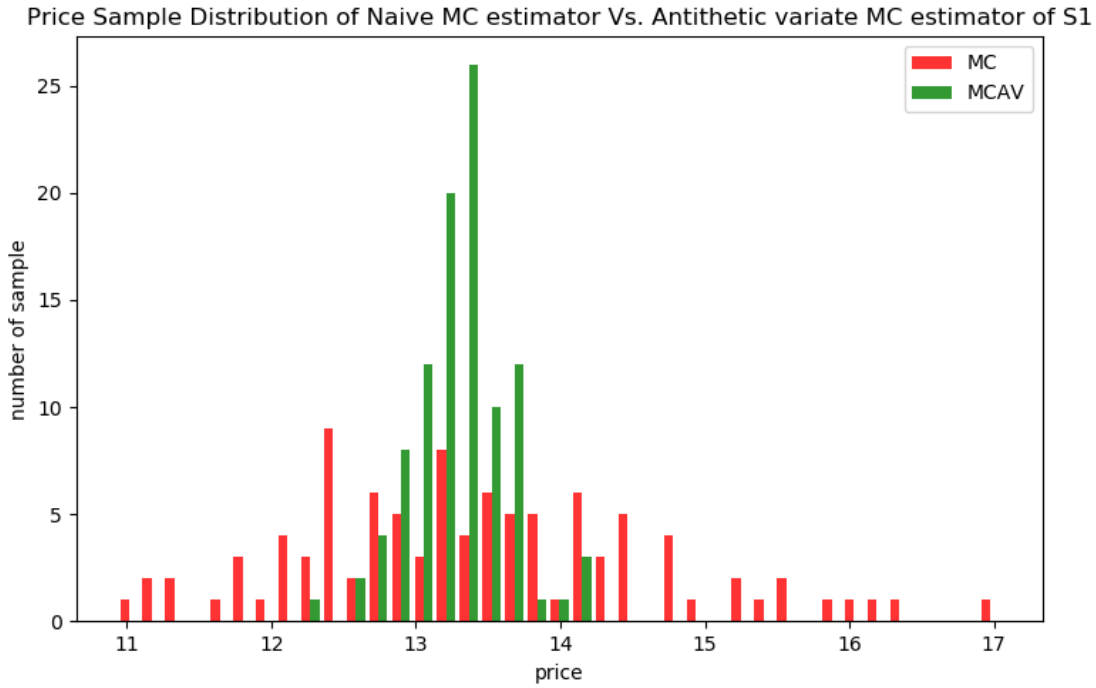
B-S price for Asset 1: 13.322705, time used: 0.000000.
B-S price for Asset 2: 10.538943, time used: 0.000000.
MC estimator for Asset 1: 12.967337, time used: 0.005231.
MC estimator for Asset 2: 9.597588, time used: 0.005231.
Antithetic variate estimator for path1: 13.028943, Antithetic variate time used: 0.013088.
Antithetic variate estimator for path2: 10.863420, Antithetic variate time used: 0.013088.
Best Seed for Asset S1 and S2: 84
price paths of S1:

price paths of S2:

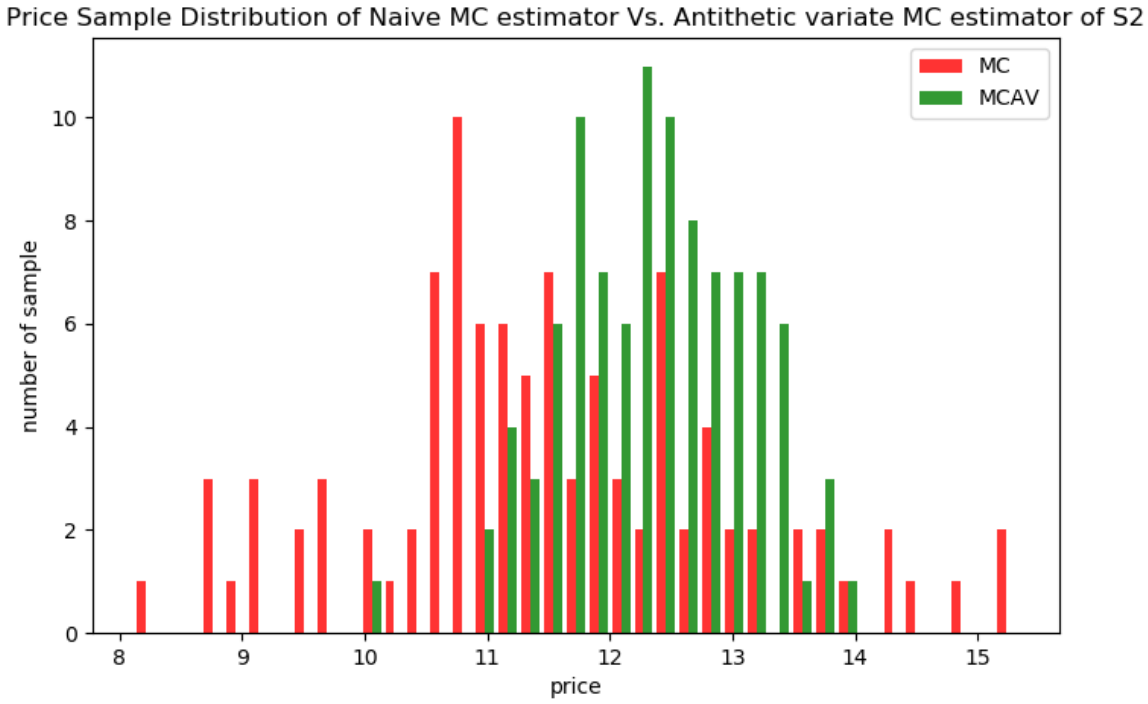


Naive MC estimator for Asset 1 mean: 13.429590, standard dev: 1.221526.
Antithetic Variates MC estimator for Asset 1 mean: 13.295097, standard dev: 0.330655.
Naive MC estimator for Asset 2 mean: 11.517873, standard dev: 1.455184.
Antithetic Variates MC estimator for Asset 2 mean: 12.360179, standard dev: 0.742329.

Price Sample Distribution of Naïve Monte Carlo estimator Vs. Antithetic variate Monte Carlo estimator of S1:

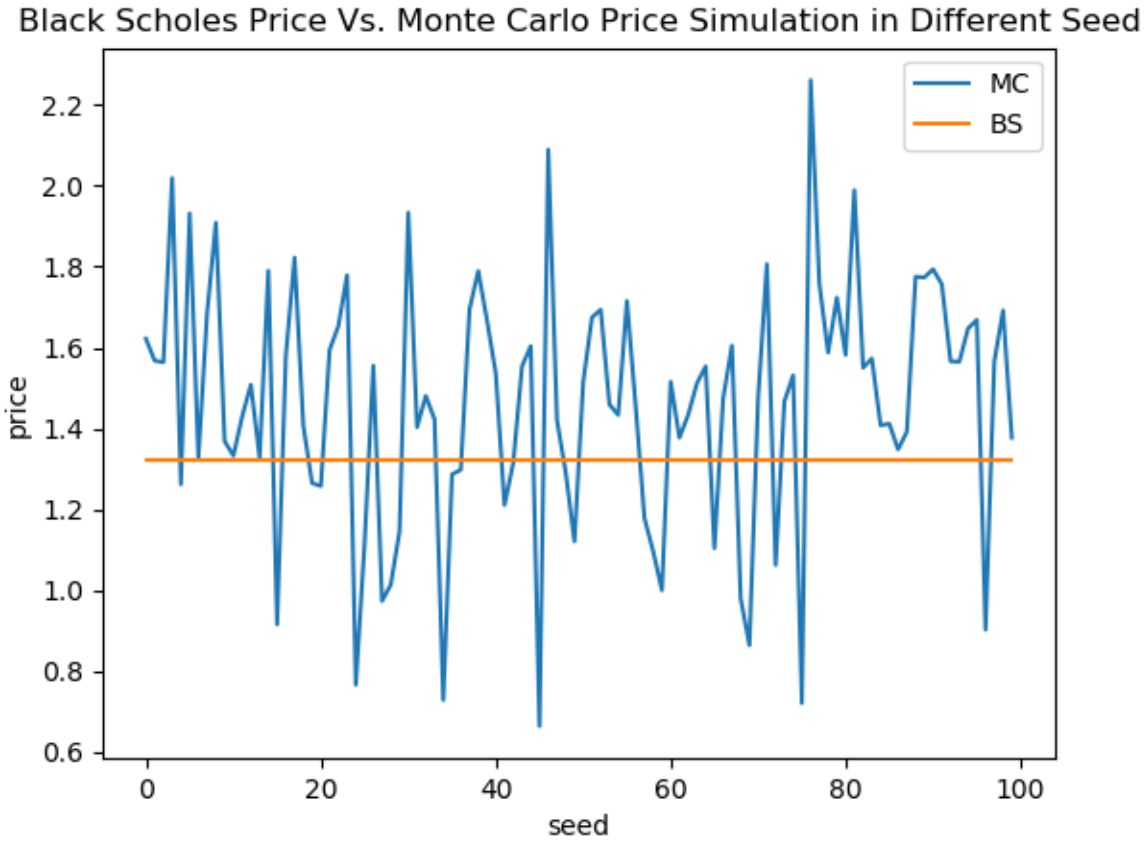


Price Sample Distribution of Naïve Monte Carlo estimator Vs. Antithetic variate Monte Carlo estimator of S2:



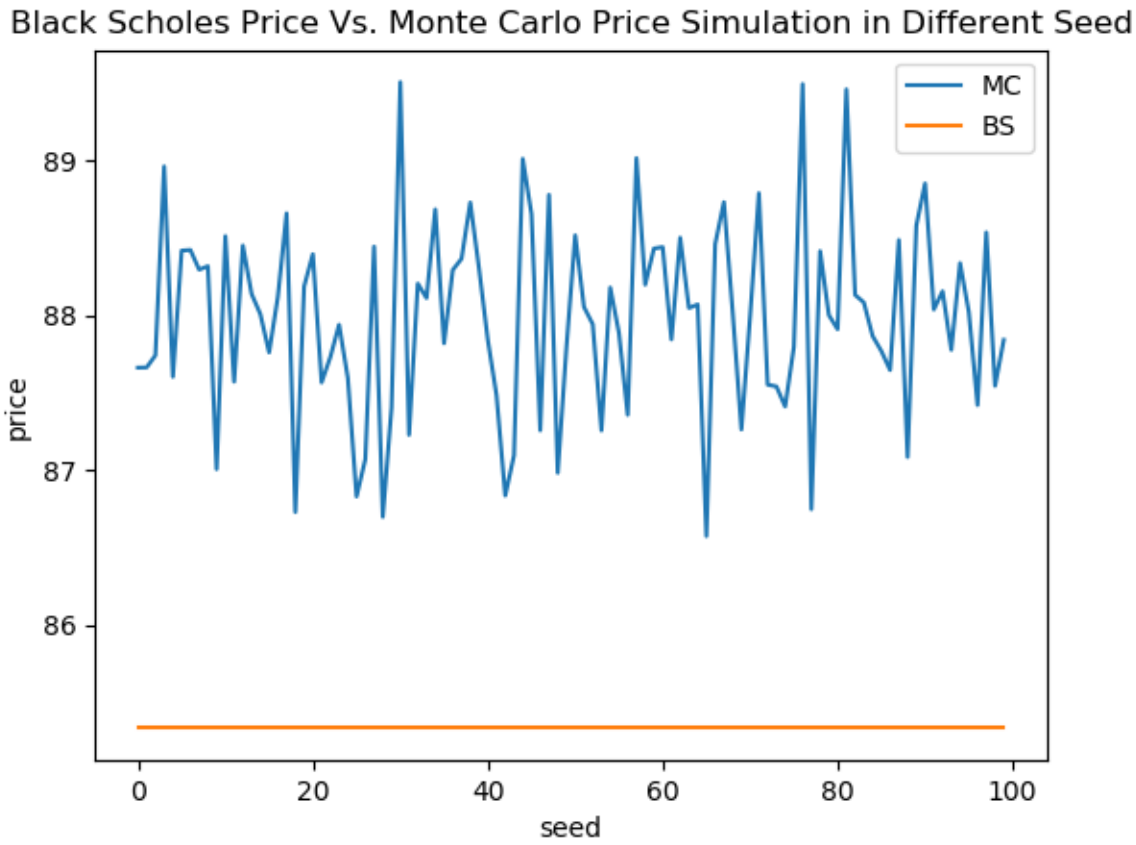
b. Implement the Longstaff and Schwartz (RFS, 2001) algorithm to price a standard American-style put option on S1 with strike price K = 90 and maturity time T = 0.5 years.

Best Seed of Put Option for Asset S1: 42
American Put Option Price (MC) for S1: 1.314627
American Put Option Price (BS) for S1: 1.320316



c. (Bonus question, optional) Price an American-style spread call option with strike price K = 15 and the maturity time 0.5 years.
Best Seed of Call Option for Asset S1: 65
American Call Option Price (MC): 86.57488792405864

American Call Option Price (BS): 85.33373144209996



4. Improvement

In this project, the algorithm used for calculating the call and put option in Monte Carlo Methods should be optimized, especially call option. The call and put option price value generated from different seeds are slightly off theoretical value (Black-Scholes). My python implementation simply compares the Monte Carlo Simulation price value for 100 seeds and get the closest value to Black-Scholes price, then obtain the best value by Monte Carlo Simulation. Antithetic variate Monte Carlo estimator can exclude unimportant samples, which is relatively much off large sample and generate more accurate results by getting positive and negative random variables. The call and put option results can be optimized by selecting better seeds, more effective algorithm and increasing number of simulations and timestep.

5. Conclusions

In this project, I learned how to implement Monte Carlo Methods to price American Put and Call Option based on different parameters: number of time steps, barrier option, current price level, delta, sigma, number of simulations. All the implementations are based on solid understandings of American call option financial theorem.
Computing methods are derived from risk-neutral probability setup and parameters such as current pricing, strike pricing, delta, sigma, time length, risk-free interest rate, dividend rate given by the problem.
The Monte Carlo Simulation are easy to implement because the algorithm just generates normal random zero and one, then run significant amount of simulations to get the price path, then finalize the option price. This implementation of Monte Carlo Simulation performance is not as expensive as Trinomial Tree. The complexity of this algorithm is just O(N) better compared to O(N^2) for Trinomial Tree. The computing methods also generate as accurate pricing values as Trinomial Tree and Adaptive Mesh Model.
When the computing method comes to least-squares method invented by Francis A.Longstaff, it bases on Monte Carlo Methods used in the first questions for generating certain number of price paths. Then the algorithm gets either call option or put option using np.maximum(path - self.K, np.zeros((self.M + 1, self.i), dtype=np.float64)) or np.maximum(self.K - path, np.zeros((self.M + 1, self.i), dtype=np.float64)). The function is analogous to the intermediate cash-flow matrices used in the path generation. The objective of the least-squares method algorithm is to provide a pathwise approximation to the optimal stopping rule that maximizes the value of the American option. The least-sqaures approach uses least squares to approximate the conditional expectation function at tk-1, tk-2, ..., t1. The algorithm work backwards since the path of cash flows C(w,s;t,T) generated by the option is defined recursively; C(w, s;tk, T) can be differ from C(w,s;tk+1,T) since it may be optimal to stop at time tk+1, thereby changing all subsequent cash flows along a realized path w. This is done by regression = np.polyfit(path[t], value[t + 1] * self.discount, 5), continuation_value = np.polyval(regression, path[t]). Ultimately the algorithm sums up all values greater than continuation value, uses that final value to time discount factor divided by number of simulations. The least-squares method algorithm can be used to approximate the value of these options by taking K to be sufficiently large. At time tk, the cash flow from

immediate exercise is known to the investor, and the value of immediate exercise simply equals this cash flow.
Most importantly, this project involves significant amount of mathematics random statistical logic and formula to construct the Monte Carlo method to price American option model using Python, implementing in Python code enhances my understandings of the algorithm. It takes me to learn and practice many powerful python libraries such as math, numpy, pandas, matplotlib, which establishes Monte Carlo simulation pricing path movement across defined time step. I believe this project experience is a valuable add-on to my programming skill, financial knowledge about American call option, and the implementation of random statistical model.

Reference:
[1] “Mark Rubinstein", “Implied Binomial Trees”, The Journal of Finance, Vol. 49, No.3, Papers and Proceedings Fifty-Fourth Annual Meeting of the American Finance Association, Boston, Massachusetts, January 3-5, 1994 (Jul., 1994), 771-818
[2] “Phelim Boyle”**, Mark Broadieb, Paul Glassermanb”, Monte Carlo methods for security pricing, “School of Accountancy, University of Waterloo. Waterloo. Ont. Canada NZL 3GI Graduate School of Business, Columbia University, New York, NY, 10027, USA
[3] “Francis A. Longstaff, Eduardo S.Schwartz”, “Valuing American Options by Simulation: A Simple Least-Squares Approach”, UCLA

Appendices

Code:
The python program code file has been attached with the submission. With Pycharm and libraries installed, the python code will be executable.