Report: Least Squares Monte Carlo Python Implementation

Zhonglin Yang

The report presents a Python implementation of Longstaff and Schwartz's least squares Monte Carlo approach of valuing American options by simulation (from [1]).

1 Longstaff and Schwartz's Method

1.1 Monte-Carlo simulation

One of the most common ways to value financial products such as options is using Monte Carlo simulation. The basic idea of the approach is creating a large sample of paths of random processes for the underlying asset such stock prices, and then explicitly computing the expected value of the discounted payoff using these paths.

1.2 Least squares regression

Longstaff and Schwartz proposed a least squares Monte Carlo approach to evaluate American-style options in 2001. To understand thee intuition behind the approach, we need to recall that the holder of an American option optimally compare the immediate exercise payoff with the expected continuation payoff at any exercise time, and choose to exercise the option only if the immediate exercise payoff is higher than the expected continuation payoff. The key point to evaluate American options by Monte Carlo simulation is to be able to approximate the conditional expectation at any exercise time. Longstaff and Schwartz proposed to use least squares regression to directly find the conditional expectation function at each exercise time. By using the conditional expectation functions, a complete optimal exercise strategy can be found along each path. And with the complete optimal exercise strategy, American options can be evaluated accurately.

1.3 High level of practical implementation

Several articles also addressed valuing American options by simulation around the time that Longstaff and Schwartz published their article. However, none of these articles introduced an approach as practical as Longstaff and Schwartz's approach. Longstaff and Schwartz only include paths for which the option is in the money in doing least squares regression. It increases the efficiency of the approach.

2 LSM Approach Algorithm

2.1 The overall algorithm

The overall algorithm for Longstaff and Schwartz's least squares Monte Carlo approach is given as follows.

Algorithm 1: Valuing American Options by LSM approach

```
generate stock price paths(m, n);
for ( i = n; i > 0; i -- ) {
    do least square regressions(i);
    determine cashflows for each path(i);
}
calculate the option price;
```

2.2 Generate stock price paths

To simulate stock prices and generate paths, we suggest that stock prices follow a geometric Brownian motion.

$$dS_t = rS_t dt + \sigma S_t dW_t \tag{1}$$

Thus, we have $\Delta S = Srdt + S\sigma dW\sqrt{dt}$ where dW is the random variable which follows a normal distribution. We random generates $m \times n$ random vairables dW, create a $m \times n$ matrix S, and store $S[i,t]$ for all $i \in [1,m]$ and all $t \in [1,n]$where $S[i,t]$ represents the stock price of the ith path at timestamp t.

2.3 Do least squares regression

We use the first seven Legendre polynomial functions as basis functions for least squares regression and they are $L_0(X), L_1(X)$ ... $L_6(X)$.

When we have generate m stock price paths, we can do least squares regression for each timestamp t for all $t \in [1,n]$ backwards. Suppose we want to do least squares regression at timestamp k, since we already know $S[i,k]$ for all $i \in [1,m]$, we can collect all the pairs of $S[i,k]$ and discounted $cashflow[i,k+1]$ whose $S[i,k]$ is in the money. Then, Suppose l pairs are collected, according to the l pairs values, we can construct a $l \times 7$ martix ϕ and a $l \times 1$ matrix Y where Y is filled with the l discounted $cashflow$ values and each column ϕ stores the corresponding S values' Legendre polynomials. For example, the first pair is $(S_1, cashflow_1)$, then $Y[1]$ stores $cashflow_1$ and $\phi[1]$ is $[L_0(S_1), L_1(S_1)$ ... $, L_6(S_1)]$. Next, we construct $A = \phi^T \cdot \phi$, and $b = \phi^T \cdot Y$. Lastly, we solve the linear system $Ax = b$. x stores seven coefficients $x[0], x[1]$ ... $, x[6]$. Then the conditional expectation function $E(X)$ is $x[0] \cdot L_0(X) + x[1] \cdot L_1(X)$ ... $+x[6] \cdot L_6(X)$.

2.4 Determine cashflows and evaluate the American option

After getting the conditional expectation functions, we can determine whether exercise the American option at each timestamp or not. For $S[i,t]$, if $payoff(S[i,t])$ ¡ $E(S[i,t])$, then we shouldn't exercise the option at timestamp t in the ith path. After determining the optimal exercise time for each path, the value of the American option can be computed based on simulation results. It is

$$\sum_{n=1}^{m} payoff(S[n, opt(n)]) \cdot discount(opt(n)).$$

3 Python Implementation Code

3.1 Legendre polynomials

We choose the first seven Laguerre polynomials as basis functions for least-squares regression. And to handle the extreme behaviors, we multiply each polynomial by an additional exponential term. The following code snippet is the implementation of Legendre polynomial functions.

```
import math

def l_0(val_arg):
    return math.exp(-val_arg/2)

def l_1(val_arg):
    return math.exp(-val_arg/2)*(1-val_arg)

def l_2(val_arg):
    return math.exp(-val_arg/2)*(1-2*val_arg+1/2*(val_arg**2))

def l_3(val_arg):
    return math.exp(-val_arg/2)*(1-3*val_arg+3/2*(val_arg**2)-1/6*(val_arg ** 3))

def l_4(val_arg):
    return math.exp(-val_arg/2)*(1-4*val_arg+3*(val_arg**2)
                                 -3/4*(val_arg ** 3)+1/24*(val_arg**4))

def l_5(val_arg):
    return math.exp(-val_arg/2)*(1-5*val_arg+5*(val_arg**2)-10/6*(val_arg**3)
                                + 25/120*(val_arg**4)-1/120*(val_arg**5))

def l_6(val_arg):
    return math.exp(-val_arg/2)*(1-6*val_arg+540/72*(val_arg**2)
                                   -240/72*(val_arg**3)+45/72*(val_arg**4)
                                   -1/20*(val_arg**5)+1/720*(val_arg**6))
```

3.2 Determination of cashflow

We use three helper functions to decide whether the American option should be exercised or not at a given timestamp.

```
# check whether to exercise or continue
def whether_exercise(S,K,k,coefficients,test_functions):
    continue_val = 0
    n = len(test_functions)
    for i in range(0, n):
        continue_val+= coefficients[k][i]*test_functions[i](S)
    return max(K - S,0.0) > continue_val

# evaluate the payoff
def payoff(exercise_val,path,i,K,m):
    for j in range(0, m):
        exercise_val[j] = max(0.0,K-path[j][i])

# compare the continuation value and the current payoff
def determine(continue_val,exercise_val,cashflow,m):
    for i in range(0, m):
        if continue_val[i] < exercise_val[i]:
            cashflow[i] = exercise_val[i]
```

3.3 Least-squares regression

The following code snippet is the least-squares regression function where the coefficients are calculated according to the given basis functions.

```
def regression(x_regression,y_regression,size_regression,coefficients,k,test_functions):
    size_1 = size_regression
    size_2 = len(test_functions)
    y_regression_2 = np.zeros(size_regression)

    # construct the matrix phi
    d_phi = np.zeros((size_1, size_2))
    for i in range(0, size_1):
        y_regression_2[i] = y_regression[i]
        for j in range(0, size_2):
            d_phi[i][j] = test_functions[j](x_regression[i])

    # construct A = phi^T*phi, b = phi^T*Y
    d_phi_transpose = d_phi.transpose()
    A = np.dot(d_phi_transpose, d_phi)
    b = np.dot(d_phi_transpose, y_regression_2)

    # solve linear equations Ax = b
    try:
        x = np.linalg.solve(A, b)
```

```
        for i in range(0, size_2):
            coefficients[k][i] = x[i]
    except np.linalg.LinAlgError as err:
        raise np.linalg.LinAlgError()
```

3.4 Projection of continuation value

The projection function is used to evaluate continue values of each path at a given timestamp. The coefficients are calculated by the least-squares regression function.

```
def projection(continue_val,coefficients,k,test_functions,path,m):
    n = len(test_functions)
    for i in range(0, m):
        continue_val[i] = 0
        for j in range(0, n):
            continue_val[i]+= coefficients[k][j] * test_functions[j](path[i][k])
```

3.5 Path generations and regression steps

In the euler_path function, firstly, all pilot paths are generated. Then the regression steps are done backwards. Lastly, according to the results from the regression steps, exercise boundaries are determined and the value of the American option are calculated.

```
def euler_path(m,n,S0,sigma,r,K,T,dW,coefficients,test_functions):
    path = np.zeros((m, n))
    dt = T/n
    S = S0

    # generate all paths
    for j in range(0, m):
        for i in range(0, n):
            S+= r*S*dt + sigma[i]*S*math.sqrt(dt)*dW[j][i]
            path[j][i] = S
        S = S0

    # do the regression steps backward
    continue_val = np.zeros(m)
    exercise_val = np.zeros(m)
    cashflow = np.zeros(m)

    for i in range(n - 1, 0, -1):
        projection(continue_val,coefficients,i,test_functions,path,m)
        payoff(exercise_val,path,i,K,m)
        determine(continue_val,exercise_val,cashflow,m)

        x_regression = np.zeros(m)
        y_regression = np.zeros(m)
        size_regression = 0
```

```
        for j in range(0, m):
            cashflow[j]* = math.exp(-r*dt)
            if path[j][i] < K:
                x_regression[size_regression] = path[j][i - 1]
                y_regression[size_regression] = cashflow[j]
                size_regression+ = 1

        regression(x_regression,y_regression,size_regression,
                   coefficients,i - 1,test_functions)

    projection(continue_val,coefficients,0,test_functions,path,m)
    payoff(exercise_val, path,0,K,m)
    determine(continue_val,exercise_val,cashflow,m)
    V = 0

    # evaluate the option
    for j in range(0, m):
        V+ = math.exp(-r*dt) * cashflow[j]

    return V/m
```

3.6 American option class and tangent method for pricing

To evaluate the American option by using the tangent method. We create an AmericanOption Class and define the american_price function where the value and greek letters of the American option are calculated by using tangent method.

```
class AmericanOption(object):

    def __init__(self,m,n):
        self.m = m
        self.n = n
        self.S0 = 1
        self.T = 1
        self.K = 1
        self.r = 0.04
        self.sigma = np.full(n, 0.2)
        self.vega = np.full(n, 0)
        self.vanna = np.full(n, 0)
        self.dW = np.zeros((m, n))
        self.S = self.S0
        self.delta = 1
        self.gamma = 0
        self.vegaBS = 0
        self.vannaBS = 0
        self.V = 0

    # the pricing function using tangent method
```

```
def american_price(self,coefficients,test_functions):
    sum, sumt, sumd, sumtt, sumtd = 0, 0, 0, 0, 0
    S0 = self.S
    delta0, vegaBS0, gamma0, vannaBS0 = self.delta, self.vegaBS, self.gamma,\
                                        self.vannaBS
    dt = self.T/self.n

    for j in range(0, self.m):
        t = 0
        for i in range(0, self.n):
            self.vannaBS += self.r*self.vannaBS*dt\
                            + self.sigma[i]*self.vannaBS*math.sqrt(dt)*self.dW[j][i]\
                            + self.delta*math.sqrt(dt)*self.dW[j][i]
            self.vegaBS += self.r*self.vegaBS*dt\
                            + self.sigma[i]*self.vegaBS*math.sqrt(dt)*self.dW[j][i]\
                            + self.S*math.sqrt(dt)*self.dW[j][i]
            self.delta += self.r*self.delta*dt\
                            + self.sigma[i]*self.delta*math.sqrt(dt)*self.dW[j][i]
            self.S += self.r*self.S*dt\
                            + self.sigma[i]*self.S*math.sqrt(dt)*self.dW[j][i]
            t = dt*(i+1)
            if whether_exercise(self.S, self.K, i, coefficients, test_functions):
                sum+ = max(self.K - self.S, 0.0)*math.exp(- self.r*t)
                sumt+ = smooth_digit(self.S, self.K)*math.exp(- self.r*t)*self.delta
                sumd+ = smooth_digit(self.S, self.K)*math.exp(- self.r*t)*self.vegaBS
                sumtt+ = (self.delta**2)*math.exp(-self.r*self.T)\
                         *smooth_diac(self.S,self.K)
                sumtd+ = math.exp(-self.r*t)*smooth_diac(self.S, self.K)*self.delta\
                         *self.vegaBS + smooth_digit(self.S, self.K)\
                         *math.exp(-self.r*t)*self.vannaBS
                break

        self.S = S0
        self.delta = delta0
        self.gamma = gamma0
        self.vegaBS = vegaBS0
        self.vannaBS = vannaBS0

    self.S = sum/self.m
    self.delta = sumt/self.m
    self.gamma = sumtt/self.m
    self.vegaBS = sumd/self.m
    self.vannaBS = sumtd/self.m
```

3.7 Main function

Lastly, we define the main function to initialize an AmericanOption object and use its attributes to call euler_path function so that we can get the coefficients of basis functions.

Then, we pass the coefficients to the american_price function to calculate the price and greek letters of the American option.

```
def main(m, n):
    ao = AmericanOption(m, n)

    np.random.seed(0)
    ao.dW = np.random.normal(0,1,(m,n))

    test_functions = [l_0, l_1, l_2, l_3, l_4, l_5, l_6]
    coefficients = np.full((n, len(test_functions)), 0.0)

    # generate the paths and calculate the regression coefficients
    ao.V = euler_path(m,n,ao.S0,ao.sigma,ao.r,ao.K,ao.T,ao.dW,\
                      coefficients,test_functions)

    print(coefficients)

    np.random.seed(1)
    ao = AmericanOption(2*m, n)
    ao.dW = np.random.normal(0,1,(2*m,n))

    # calculate the price of the American option
    ao.american_price(coefficients,test_functions)

    # print out the calculated price and greek letters of the American option
    print('V = {}'.format(ao.V))
    print('S = {}'.format(ao.S))
    print('delta = {}'.format(ao.delta))
    print('gamma = {}'.format(ao.gamma))
    print('vegaBS = {}'.format(ao.vegaBS))
    print('vannaBS = {}'.format(ao.vannaBS))


t0 = time()
main(100000, 100)
t1 = time()
d1 = t1 - t0
print (d1)
```

3.8 Miscellaneous functions

Two approximation functions which are used in the american_price function.

```
def smooth_digit(S, K):
    return -1.0 / (1.0 + math.exp(-(K - S) / 0.005))

def smooth_diac(S, K):
    return math.exp(-(K-S)/0.005)/0.005/
                    ((1.0+math.exp(-(K-S)/0.005))*(1.0+math.exp(-(K-S)/0.005)))
```

4 Numerical Results

With $S_0 = 1, r = 0.04, \sigma = 0.2, T = 1$, and $K = 1$, Geeske and Johnson got the analytical results (price, delta, vega, gamma, and vanna) of the American option in [2]. Pass the same attributes with $m = 100,000, n = 100$, and three various random variable seeds to our program, we get our tangent method results and compare our results with the analytical results in the following table.

	Analytical Results	Seed = 0	Seed = 1	Seed = 2
Price	0.0640	0.05969	0.05925	0.0630
Delta	-0.4160	-0.3810	-0.3685	-0.4069
Vega	0.3833	0.3472	0.3463	0.3719
Gamma	1.1069	0.4947	0.5064	0.5853
Vanna	0.4247	0.1837	0.1793	0.1787

References

[1] Francis A. Longstaff and Eduardo S. Schwartz. Valuing American Options by Simulation: A Simple Least-Squares Approach. *The Review of Financial Studies*, 14(1):113–147, January 2001.

[2] Robert Geske and H. E. Johnson. The American Put Option Valued Analytically. *The Journal of Finance*, 39(5):1511–1524, 1984.