Pricing American Option with Least Squares Monte Carlo

October 2023

Abstract

Pricing American options is inherently more complex than pricing European options, often requiring the use of numerical methods. Gustafsson conducted an evaluation of American option pricing using the Longstaff-Schwartz method, implementing specific enhancements designed to minimize memory usage. Our objective is to reproduce his findings and results of the American put option price within this framework.

1 Introduction

Gustafsson numerically priced American put options as outlined in [Gus15]. He employed the Least Squares Monte Carlo (LSM) method originally developed by Longstaff and Schwartz, as described in [LS01]. Notably, Gustafsson enhanced the algorithm to reduce memory usage significantly. This improvement involved a reduction in the number of slices of GBM (Geometric Brownian Motion) paths stored at each time point.

Monte Carlo methods are employed to estimate both the continuation value and the option price. Gustafsson's improvement introduces the use of the Brownian Bridge method. This innovation allows for the storage of values of the Geometric Brownian Motion (GBM) at the current step only, significantly reducing memory requirements.

1.1 Notations

Consider the following variables:

Time: $t \in [0, T],$ where T represents the expiration date.

Asset Price: $S = S(t),$ denoting the price of the underlying asset.

Option Value: $u = u(S, t),$ representing the value of the option.

Risk-free Rate: r.

Volatility: σ.

Strike Price: K.

1.2 Backgrounds

1.2.1 American put options

Recall that the Black-Scholes formula for American options is expressed as an inequality:

$$\frac{\partial u}{\partial t} + rS\frac{\partial u}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} - ru \leq 0$$

with the following boundary conditions:

- At maturity, $t = T$:

$$u(S, T) = \max(K - S, 0)$$

- Throughout the option's life, $t < T$, especially for American put options:

$$u(S, t) \geq \max(K - S, 0)$$

These equations describe the dynamics and boundary conditions for American options in the Black-Scholes framework.

1.2.2 GBM

A *Geometric Brownian motion* (GBM) is a process $\{S(t)\}$ that satisfies the following Stochastic Differential Equation (SDE):

$$dS(t) = \mu S(t)dt + \sigma S(t)dW(t)$$

where μ and σ are constants, and $W(t)$ is the standard Brownian motion.

By Itô's Lemma, we can solve the above SDE. Its solution is given by:

$$S(t_1) = S(t_0)e^{(\mu - \frac{1}{2}\sigma^2)(t_1 - t_0) + \sigma(W(t_1) - W(t_0))}$$

1.2.3 Monte Carlo

Monte Carlo methods primarily serve as tools for estimating means. If we have a function, $f(x)$, that can be factored as $f(x) = h(x)p(x)$, where $p(x)$ represents some density function, we can express the integral as an expected value:

$$\mu = E[h(x)] = \int h(X)p(x)dx = \int f(X)dx$$

With a large number of samples, x_i, generated, we can estimate the expectation as follows:

$$\hat{\mu} = \frac{1}{n} \sum h(x_i)$$

Furthermore, it is important to note that:

$$\frac{\sqrt{n}(\hat{\mu} - \mu)}{\sigma} \sim \mathcal{N}(0, 1)$$

This result highlights the convergence of the estimate to a normal distribution with mean zero and variance one.

1.3 Methods and Algorithms

1.3.1 Simulate GBM

While a Geometric Brownian Motion (GBM) cannot be exactly modeled, it can be simulated discretely when considering a finite number of steps. GBM can be simulated as follows: Take an equidistant discretization of the time interval $[0, T]$ with time points $0 = t_0, t_1, \ldots, t_N = T$, and obtain $Z_1, Z_2, \ldots, Z_N$, which are independent and identically distributed (i.i.d.) as $N(0, 1)$. Set an initial value, $S(t_0) = S_0$. Then, using the following equation:

$$S(t_{i+1}) = S(t_i) \exp\left(\mu - \frac{1}{2}\sigma^2\right) \Delta t + \sigma\sqrt{\Delta t} Z_{i+1}$$

1.3.2 Pricing American put option

Considering the American put option, the payoff at time t is given by:

$$g(S(t)) = \max(K - S(t), 0),$$

where the underlying asset $S(t)$ follows the Black-Scholes model. We can express the value of the American put option $u(S, t)$ at time $t = 0$ as:

$$u(S, 0) = \sup_{t \in [0,T]} E\left[e^{-rt} g(S(t))\right],$$

To estimate the value of the option using Monte Carlo methods, we need to first find the optimal exercise time t^* and then estimate the expected value as:

$$u(S, 0) = E\left[e^{-t^r} g(S(t^)))\right],$$

To find the optimal exercise time t^*, we can use dynamic programming. Specifically, we consider an equidistant discretization of the time interval $[0, T]$, where $0 = t_0 < t_1 < \ldots < t_N = T$. We denote the value of holding onto the option at time t_i as the "continuation value," denoted by $C(S(t_i))$, and the value of exercising at time t_i as the payoff, which is $g(S(t_i))$. The optimal exercise time is found by working backward from t_N to t_0, following Algorithm 1 as described in Gustafsson's article.

But in the previous argument, the computation of the continuation value, denoted as $C(S(t_i))$, can be described as a conditional expectation:

$$C(S(t_i)) = E\left[e^{-r(t^-t)} g(S(t_i)), |, S(t_i)\right],$$

Here, t^* represents the optimal exercise time within the interval $[t_{i+1}, \ldots, t_N]$. Since dynamic programming is employed from t_N to t_0, this definition is well-suited for the problem.

Gustafsson introduced an auxiliary variable $P = P(S(t_i))$, referred to as the "current payoff," to express the continuation value:

$$C(S(t_i)) = E\left[e^{-r\Delta t} P, |, S(t_i)\right],$$

This formulation makes it convenient to calculate P recursively, simplifying the process of finding the continuation value.

To estimate the conditional expectation mentioned earlier, we employ the LSM method (Least Squares Regression). This conditional expectation can be viewed as a functional in the L^2-spaces. However, it's worth noting that the L^2-space in this context is infinite-dimensional. To handle this, we must select a finite orthonormal basis. Gustafsson chose to use the Laguerre polynomials for this purpose. In contrast to the original LSM method implemented by Longstaff and Schwartz, Gustafsson opted for the use of weighted Laguerre polynomials as they provided more accurate numerical results.

In summary, to estimate the value of the American put option, we simulate M paths of the GBM, denoted as $S_j(t)$, with time steps $t = 0, t_1, \ldots, t_N$. Following the previous discussion, we utilize this data to estimate the continuation value and determine t_j^*, the optimal exercise time for each path $S_j(t)$. Finally, we apply the Monte Carlo method to estimate the option's value using the following formula:

$$\hat{u} = \frac{1}{M} \sum_{j=1}^{M} e^{-rt^j g(S_j(t_j^)).(1)}$$

These steps correspond to Algorithm 2 presented in Gustafsson's paper, constituting the original LSM algorithm.

Gustafsson pointed out that the original LSM algorithm requires generating all time steps of all GBM paths at the outset. This is because the standard Brownian motion (BM) is constructed from time $t = 0$ to $t = T$. However, in the LSM algorithm, we only require information about each GBM path at discrete times $t = t_i$ and then work backward. To reduce memory consumption and computational complexity, Gustafsson introduced the concept of the "Brownian bridge," a method to simplify the LSM algorithm.

First, we note that the GBM $S(t)$ is defined as:

$$S(t) = S(0)e^{(\mu t + \sigma W(t))},$$

Hence, we only need to simulate:

$$X(t) = \mu t + \sigma W(t),$$

where $\mu = r - \frac{1}{2}\sigma$. Since $X(t)$ is normally distributed for each t, we can leverage a theorem regarding the conditional distribution of two normally distributed random variables to generate $X(t_i)$ backward as follows:

$$X(t_i) = \frac{t_i X(t_{i+1})}{t_{i+1}} + \sigma \sqrt{\frac{t_i \Delta t}{t_{i+1}}} Z,$$

where $Z \sim \mathcal{N}(0,1)$. Algorithm 3 in the article represents an improvement over Algorithm 2 by incorporating the aforementioned approach for generating GBM, thereby reducing memory usage in the computation.

1.3.3 Numerically finding the early exercise boundary

To estimate the exercise boundary, we apply either Algorithm 2 or Algorithm 3 to M paths of GBM, denoted as $S_j(t)$. For each path, we find the optimal exercise time t_j and exclude paths where $t_j^{=}T$. This process yields a set of tuples $(t_j^{,} S_j(t^j))$.

Various methods can be employed to estimate the exercise boundary from this point. Some suggest taking the infimum, while others recommend using the mean. In our case, we utilize monotonic (isotonic) regression to fit the data and obtain $\hat{b}(t)$

1.3.4 Suggested Improvement

I believe that this algorithm can be further improved by considering variate control, as described in [Lid07]. In Monte Carlo methods, the estimator $\bar{X} = \frac{1}{M}\sum_{i=1}^{M} X_i$ is commonly used to estimate the expectation of X. However, this estimator can have a large variance. To address this issue, we can employ the variate control estimator with another random variable Y having a mean μ_Y:

$$\bar{X}_{CV}(\beta) = \bar{X} - \beta(\bar{Y} - \mu_Y).$$

This approach can help reduce the variance of our estimates.

2 Replicate

We aim to replicate the results using Python, employing essential libraries and packages for computation. Specifically, we will make use of NumPy for numerical operations, the 'random' module for generating random numbers, and the 'math' module for mathematical functions. Additionally, we will utilize the 'sklearn' library for Isotonic Regression, and the 'matplotlib' library for data visualization and plotting.

2.1 Basis function

The basis functions used in [Gus15] are the Laguerre polynomials, which have convenient recursive definitions:

$$\begin{aligned} L_0(x) &= 1 \\ L_1(x) &= 1 - x \\ L_{n+1}(x) &= \frac{(2n+1-x) \cdot L_n(x) - n \cdot L_{n-1}(x)}{n+1} \end{aligned}$$

These recursive definitions lead to the following Python function:

```
def laguerre(n, x):
    if n < 0:
        return None  # The order should be > 0.
    if n == 0:
        return 1
    elif n == 1:
        return 1 - x
    else:
        k = n - 1
        return ((2*k + 1 - x) * laguerre(k, x) - k * laguerre(k - 1, x)) / (n + 1)
```

To verify the correctness of this function, we can compute some sample points, such as:

```
laguerre(2, 2)
laguerre(3, 4)
```

The first call returns $-\frac{1}{2}$, and the second call returns $\frac{7}{3}$.

2.2 GBM

We simulate the Geometric Brownian Motion (GBM) as suggested in Section 1.3.1.

```
import random
import math

# T: Expiration date
# n: Number of steps
# r: Risk-free rate
# sigma: Volatility
# S0: Initial asset price
def GBM(T, n, r, sigma, S_0):
    delta_t = T / n
    list_time = [i * delta_t for i in range(0, n + 1)]
    list_price = [S_0]
    for i in range(1, n + 1):
        W = sigma * math.sqrt(delta_t) * random.gauss(0, 1)
        X = (r - 0.5 * (sigma**2)) * delta_t + W
        S_i = list_price[i - 1] * math.exp(X)
        list_price.append(S_i)
    return list_time, list_price
```

We verify this function indeed outputs GBM by plotting the output as shown in Figure 1.

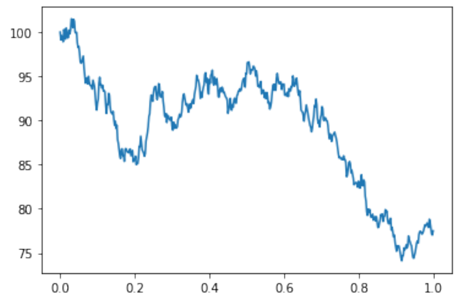

Figure 1: Example of GBM(1, 500, 0.03, 0.15, 100)

2.3 LSM

We follow Algorithm 2 from [Gus15] to price the American put option using the Longstaff-Schwartz (LSM) method:

```
import numpy as np
import math

# M: number of paths
# K: Strike price
# k: number of basis functions in function space
def LSM(T, n, r, sigma, S_0, K, M, k):
    import numpy as np
    paths = []
    dt = T / n
    for i in range(M):
        path = GBM(T, n, r, sigma, S_0)
        paths.append(path[1])
```

```
P = []
for path in paths:
    P.append(max((K - path[n]), 0))

for j in range(n - 1):
    t = n - 1 - j
    itm_paths = []
    for i in range(M):
        if paths[i][t] < K:
            itm_paths.append(i)

    x = []
    y = []

    for i in itm_paths:
        x.append(paths[i][t])
        y.append(math.exp(-r * dt) * P[i])

    L = []
    for i in range(len(x)):
        row = []
        for j1 in range(k):
            row.append(laguerre(j1, x[i]))
        L.append(row)

    L_array = np.array(L)
    y_array = np.array(y)

    Ly_transpose = np.transpose(np.dot(y_array, L_array))
    inverse_L_transpose_L = np.linalg.inv(np.dot(np.transpose(L_array), L_array))
    beta_array = np.transpose(np.dot(inverse_L_transpose_L, Ly_transpose))

    C_array = np.transpose(np.dot(L_array, beta_array))
    C = list(C_array)

    for i in M:
        P[i] = math.exp(-r * dt) * P[i]

    for i in range(len(itm_paths)):
        if K - paths[itm_paths[i]][t] > C[i]:
            P[itm_paths[i]] = K - paths[itm_paths[i]][t]

price = math.exp(-r * dt) / M * sum(P)

return price
```

We have imported the Numpy package for matrix calculations and called the GBM and Laguerre functions.

Please note that in the code, we simplified the matrix multiplication to find $\hat{\beta}_i$ as follows:

$$(\hat{\beta}_0, \ldots, \hat{\beta}_k)^T = (L^T L)^{-1} L^T (y_1, \ldots, y_n)^T$$
$$(\hat{\beta}_0, \ldots, \hat{\beta}_k)^T = (L^T L)^{-1} ((y_1, \ldots, y_n) L)^T$$

The computation result with input values $S_0 = 90$, $K = 100$, $r = 0.03$, $\sigma = 0.15$, $T = 1$ with 200 steps and 100000 samples yields 10.720525865515926, which is close to the true price given in [Gus15] with the same input. This verifies the validity of our code.

2.4 LSM with Brownian Bridge

We implement Algorithm 3 as described in [Gus15] to reduce memory usage when computing option prices. The following code represents the LSM with Brownian Bridge (LSM-BB):

```
import numpy as np
import math
import random

# LSMBB: LSM with Brownian Bridge
# M: number of paths
# K: Strike price
# k: number of basis functions in function space
def LSMBB(T, n, r, sigma, S_0, K, M, k):
    X_i_t = []
    S_i_t = []
    dt = T / n

    # Simulate terminal asset prices
    for i in range(M):
        X_T = random.gauss(r - 0.5 * sigma**2, sigma)
        S_T = S_0 * math.exp(X_T)
        X_i_t.append(X_T)
        S_i_t.append(S_T)

    P = []

    # Calculate option values at expiration
    for S_T in S_i_t:
        P.append(max((K - S_T), 0)
for j in range(n - 1):
    t = n - 1 - j

    for i in range(M):
        X_i_t[i] = t / (t + 1) * X_i_t[i] + sigma * math.sqrt(t / (t + 1) * dt) * randon
        S_i_t[i] = S_0 * math.exp(X_i_t[i])

    itm_paths = []

    for i in range(M):
        if S_i_t[i] < K:
            itm_paths.append(i)

    x, y = []

    for i in itm_paths:
        x.append(S_i_t[i])
        y.append(math.exp(-r * dt) * P[i])

    L = []

    for i in range(len(x)):
        row = []

        for j1 in range(k):
            row.append(laguerre(j1, x[i]))

        L.append(row)

    L_array = np.array(L)
    y_array = np.array(y)

    Ly_transpose = np.transpose(np.dot(y_array, L_array))
    inverse_L_transpose_L = np.linalg.inv(np.dot(np.transpose(L_array), L_array))
    beta_array = np.transpose(np.dot(inverse_L_transpose_L, Ly_transpose))

    C_array = np.transpose(np.dot(L_array, beta_array))
    C = list(C_array)

    for i in range(M):
        P[i] = math.exp(-r * dt) * P[i]

    for i in range(len(itm_paths)):
        if K - S_i_t[itm_paths[i]] > C[i]:
            P[itm_paths[i]] = K - S_i_t[itm_paths[i]]
    price = math.exp(-r * dt) / M * sum(P)

    return price
```

In this LSM-BB code, we do not need to call the GBM function since we calculate asset prices backward at each iteration step, eliminating the need to store entire paths of simulated GBM.

Once again, the computation result with input values $S_0 = 90$, $K = 100$, $r = 0.03$, $\sigma = 0.15$, $T = 1$, using 200 steps and $100,000$ samples, yields a result close to the true price given in [Gus15]. Specifically, it returns 10.726711533063417, verifying the validity of our code.

2.5 Early exercise boundary

As described in Section 1.3.3, we employed the LSM method to compute the early exercise boundary with isotonic regression. Below is the Python code for the LSMNEB (LSM and exercise boundary) function:

```
import numpy as np
import math
import random
# LSMNEB: LSM and exercise boundary
# M: number of paths
# K: Strike price
# k: number of basis of function space
def LSMNEB(T,n,r,sigma,S_0,K,M,k):
    import numpy as np
    paths = []
    dt = T/n
    for i in range(M):
        path = GBM(T,n,r,sigma,S_0)
        paths.append(path[1])
    P = []
    for path in paths:
        P.append(max((K-path[n]),0))

    t_opt = [n]*M


    for j in range(n-1):
        t = n-1-j
        itm_paths = []
        for i in range(M):
            if paths[i][t] < K:
                itm_paths.append(i)

        x = []
    y = []

    for i in itm_paths:
        x.append(paths[i][t])
        y.append(math.exp(-r*dt)*P[i])

    L = []
    for i in range(len(x)):
        row = []
        for j1 in range(k):
            row.append(laguerre(j1,x[i]))
        L.append(row)

    L_array = np.array(L)
    y_array = np.array(y)

    Ly_transpose = np.transpose(np.dot(y_array,L_array))
    inverse_L_transpose_L = np.linalg.inv(np.dot(np.transpose(L_array),L_array))
    beta_array = np.transpose(np.dot(inverse_L_transpose_L,Ly_transpose))

    C_array = np.transpose(np.dot(L_array,beta_array))
    C = list(C_array)

    for i in range(M):
        P[i] = math.exp(-r*dt)*P[i]

    for i in range(len(itm_paths)):
        if K - paths[itm_paths[i]][t] > C[i] :
            P[itm_paths[i]] = K - paths[itm_paths[i]][t]
            t_opt[itm_paths[i]]=t

t_opt_n = []

S_t_opt = []

for i in range(M):
    if t_opt[i] != n:
        t_opt_n.append(t_opt[i])
        S_t_opt.append(paths[i][t_opt[i]])


combined_list = list(zip(t_opt_n, S_t_opt))
sorted_list = sorted(combined_list, key=lambda x: x[0])

x, y = zip(*sorted_list)
```

```
    model = IsotonicRegression()
    y_pred = model.fit_transform(x, y)

    # Plot the original data and the isotonic regression line
    plt.scatter(x, y, label='Original Data')
    plt.plot(x, y_pred, label='Isotonic Regression Line', color='red')
    plt.legend()
    plt.xlabel('t')
    plt.ylabel('Stock Price')
    plt.title('Eearly Exercise Boundary')
    plt.show()

    price = math.exp(-r*dt)/M*sum(P)

    return price
```

We used the scikit-learn package for isotonic regression. The following is an example of an estimated early exercise boundary for an American put option:

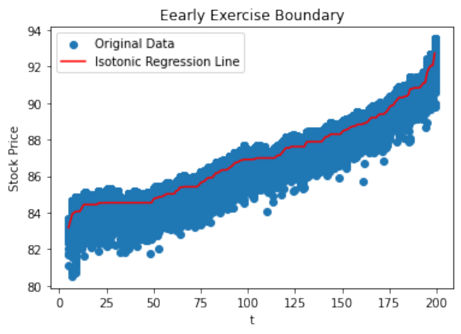

Figure 2: Example of LSMNEB(1,200,0.03,0.15,90,100,100000,4)

3 Result

Due to the limited computational power of the machine, we've reduced the simulated sample size of the GBM. However, in some cases, matrix operations may encounter issues due to low dimensions. This situation is often a result of an insufficient number of in-the-money paths. For instance, when we consider $S_0 = 110$, the difference $K - S_0 = -20$ puts us in an out-of-the-money scenario but no sufficient simulated path will be in-the-money. To avoid this problem, we use $S_0 = 101$ for the out-of-the-money case.

As presented in [Gus15], we calculate the relative error, e_r, using the following formula:

$$e_r = \left| \frac{\hat{\mu} - \mu}{\mu} \right|$$

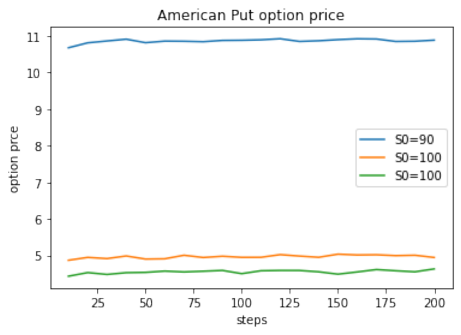

Figure 3: Average result from 20 runs with parameters: K=100, r=0.03, σ=0.15, using 10,000 Geometric Brownian Motion (GBM) simulations, and employing 4 basis functions

to demonstrate the convergence as the number of steps increases.

The true option price corresponding to:

- Volatility $\sigma = 0.15$
- Risk-free interest rate $r = 0.03$
- Strike price $K = 100$
- Time to expiry $T = 1$

is provided in the table below:

Table 1: Option prices for different S_0 values

S_0	u
90	10.726486710094511
100	4.820608184813253
101	4.083019798593495

The diagram below depicts the convergence of the Least Squares Monte Carlo (LSM) method for option pricing, utilizing four basis functions.

4 Conclusion

We have successfully replicated Gustaffon's results, demonstrating the effectiveness of our least square Monte Carlo methods for pricing American put options. However, it's important to note that due to computational limitations, we encountered challenges when dealing with the out-of-the-money case for $S_0 = 110$. In such cases, where the strike price exceeds the initial stock price, the option price cannot be accurately computed using the standard method. Despite

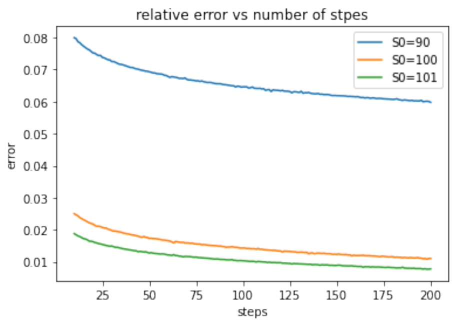

Figure 4: Relative errors from 20 runs with parameters: K=100, r=0.03, σ=0.15, 10,000 GBM simulations, and 4 basis functions.

this limitation, we were able to perform a preliminary analysis and observe the behavior of the option price as we increased the number of time steps in our simulations.

One notable result from our analysis is the convergence of the option price estimates as we increased the number of steps in our simulations. This convergence is a crucial indicator of the reliability of our Monte Carlo methods, suggesting that our estimations are becoming more accurate with a finer time granularity.

Furthermore, our implementation allowed us to estimate the early exercise boundary of the American put option. This boundary provides valuable insights into when it is optimal for an option holder to exercise their right to sell the underlying asset. By applying the least squares regression method, we were able to visualize this boundary, aiding in decision-making for option holders.

In summary, while we faced computational limitations and constraints in certain scenarios, our efforts have provided a solid foundation for pricing American put options and studying their behavior as we vary the number of time steps. We anticipate further improvements and optimizations as computational power continues to advance, enabling us to explore more complex scenarios and enhance the accuracy of our estimates.

References

[Gus15] William Gustafsson. Evaluating the longstaff-schwartz method for pricing of american options, 2015.

[Lid07] Thomas Lidebrandt. *Variance reduction three approaches to control variates*. Matematisk statistik, Stockholms universitet, 2007.

[LS01] 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, 2001.