American Option Pricing with Least Squares Monte Carlo

Final Project: STA 3431

Vanessa Pizante

University of Toronto

Student ID: 1004874022

v.pizante@mail.utoronto.ca

June 8, 2019

1 Introduction

A vanilla option is a contract written on an underlying asset (whose price at time t is denoted S_t) that gives the holder the right, but not the obligation, to sell (for a put) or to buy (for a call) the underlying for some previously specified price (called the strike price - denoted K) over some predetermined period of time. Two of the most well-known forms of vanilla options are European and American options. These two styles of option differ in that a European option can only be exercised at maturity (time $t = T$) while an American option can be exercised anytime between when it is written ($t = 0$) and its maturity. In a Black-Scholes market model, the price of a European option at any point in time has a closed form, while the price of an American option does not and and hence it must be estimated. One of the ways to do this is using a least-squares Monte Carlo method [Longstaff and Schwartz, 2001], which when applied to pricing American options is called the Longstaff-Schwartz algorithm. In this project, we will attempt to implement this algorithm within a Black-Scholes market model to find the price of a contemporary (written November 23, 2018) at-the-money Netflix put option that expires on February 22, 2018. We will also apply the results of more recent studies that attempt to improve upon this algorithm through variance reduction techniques.

2 American Option Valuation in the Black-Scholes Market Model

The Black-Scholes market model consists of a single risky asset, S_t. The market assumptions are as follows:

1. No transaction costs
2. The risk-free interest rate, r, is constant
3. There is no arbitrage opportunities (no strategies lead to a riskless profit)
4. No dividends are paid
5. $(S_t)_t$ follow a Geometric Brownian Motion with constant parameters (drift μ and volatility σ) under the real world probability measure $\mathbb{P}$

Note that 3 and 5 imply that the asset prices can also be modelled via Geometric Brownian Motion under the risk-neutral probability measure $\mathbb{Q}$ as follows:

$$dS_t = rS_t dt + \sigma S_t dW_t$$

where W_t is a standard Brownian motion under $\mathbb{Q}$. Equivalently,

$$S_t \overset{d}{=} S_0 e^{(r-\frac{\sigma^2}{2})t+\sigma\sqrt{t}Z} \tag{1}$$

where $Z \sim N(0,1)$ under $\mathbb{Q}$.

Note that the payoff function for a put option is:

$$G(S) = (K-S)^+ \equiv \max(K-S,0) \tag{2}$$

Furthermore, the price of a European put option is:

$$V^E(S_t) = \mathbb{E}^{\mathbb{Q}}[e^{-r(T-t)}G(S_T)|S_t] \tag{3}$$

This result follows from the fundamental theorem of asset pricing, for more details see [Shreve, 2004]. This value has a closed form - the famed Black-Scholes formula [Black, 1973] - and does not require a Monte Carlo estimation techniques or otherwise to estimate it. However since an American option can be exercised at any time $t \in [0, T]$, its value is given by:

$$V_t = \sup_{t^* \in [t,T]} \mathbb{E}^{\mathbb{Q}}[e^{-r(T-t)}G(S_T)|S_t] \tag{4}$$

We call time t^* the optimal exercise time of the option and it is unknown since it depends on the random price path of the asset. To account for this, we assume the holder of the option will only exercise if the value the option if exercised is larger than the expected value of the option is they continued to hold. The former is simple to calculate for any point in time t, as it is simply $G(S_t)$. The latter is often referred to as the continuation value and is denoted $C(S_t)$. The continuation value can be described in the form of the following conditional expectation:

$$C(S_t) = \mathbb{E}^{\mathbb{Q}}[e^{-r(t^*-t)}G(S_{t^*})|S_t] \tag{5}$$

The intuition behind 5 is that $C(S_t)$ is the amount the holder will receive from exercising the option at some future optimal exercise time t^*, discounted back to time t, given the value of S_t.

Note that at time T, we have no choice to hold the option, so $V_T = G(S_T)$. We can use this to determine V_t over a set of points in time $\{t_0 = 0, t_1, ..., t_N = T\}$ for a stock price path obeying equation 1 using backwards recursion. Specifically, for a given asset price path $(S_t)_{i \in t_0,...,t_N}$:

```
**for** $t = t_{N-1}, ..., t_1$ **do**
    **if** $C(S_t) < G(S_t)$ **then**
        $V_t \leftarrow G(S_t)$    *(exercise)*
    **else**
        $V_t \leftarrow C(S_t)$    *(hold)*
```

Of course, this only applies given a known asset price path $(S_t)_{i \in t_0,...,t_N}$. To estimate V_t properly, we must simulate a large number of asset price paths and repeatedly apply those to the algorithm. Then, we can use these to obtain a Monte Carlo estimate for V_t. However, an additional problem we face is that $C(S_t)$ also has no closed form. To compute $C(S_t)$, we will use least squares-regression with a polynomial basis function. This approach to the problem is known as least-squares Monte Carlo (LSM) or the Longstaff-Schwartz algorithm. In the following section, we will discuss the implementation of this algorithm in more detail.

3 The Longstaff-Schwartz Algorithm

As stated earlier, the Longstaff-Schwartz algorithm is a least-squares Monte Carlo approach to pricing American put options. The least-squares part of the algorithm comes in approximating the continuation value function $C(S_t)$ using a least-squares regression approach. The choice of basis functions in this case is typically polynomial [Hilpisch, 2014]. The choice of polynomial family has been found to have little impact on the accuracy of results, especially when pricing a single claim [Areal et al., 2008]. Furthermore, increasing the number of basis functions past 5 has little impact on the accuracy as well [Areal et al., 2008]. Hence we elect to employ five power family basis functions to price our claim. Note that the power family is selected as it is simple to implement and runs quite quickly in Python using `numpy`'s `polyfit` and `polyval` functions.

The Monte Carlo component of least-squares Monte Carlo is involved in computing a final estimate for the current option price, V_0. Note that although V_t is defined in terms of a conditional expectation in equation 4, V_0 can be regarded as an unconditional expectation since the current asset price S_0 is a known constant. This is what allows us to compute the option price using a classical Monte Carlo-style estimate.

To properly appreciate the algorithm, we will now look at a detailed layout of its implementation. First, we discretize the time interval representing the lifespan of the option, $[0, T]$, into N equally spaced time steps $\{t_0, t_1, ..., t_N\}$, with time step length $\Delta t = t_n - t_{n-1}$ for $n = 1, ..., N$. The Longstaff-Schwartz algorithm is then implemented as follows:

1. Simulate M paths of the stock price, where the time t_n value price of the m^{th} simulation is given by:

$$S_n^m = S_0 \exp\left(\sum_{k=0}^{n}\left((r - \frac{\sigma^2}{2})\Delta t + \sigma \Delta t Z_k^m\right)\right)$$

   where $Z_n^m \sim N(0,1)$, $n = 0, ..., N$ and $m = 0, ..., M$.

2. For each path, find the corresponding option payoff at expiry $V_N^m = G(S_N^m)$.

3. Backwards recursively for $n = N-1, ..., 1$:

   i Define a function $C(s) = \sum_{k=1}^{5} \beta_k s^k$ where $\boldsymbol{\beta}$ minimizes the squared error $\sum_{m=0}^{M}(V_{n+1}^m e^{-rt} - C(S_n^m))^2$. Note that we use `numpy`'s `polyfit` function to get the $\boldsymbol{\beta}$ values and `polyval` to get the resulting $C(S_n^m)$.

   ii For each $m = 0, ..., M$:

   - If $G(S_n^m) > C(S_n^m)$ the option is exercised so $V_n^m = G(S_n^m)$
   - Otherwise we hold onto the option so $V_n^m = e^{-r\Delta t} V_{n+1}^m$

4. Take $V_0^m = V_1 e^{-r\Delta t}$ for each $m = 0, ..., M$

5. We can now obtain a Monte Carlo estimate for the current option price

$$V_0 \approx V_0^{MC} = \frac{1}{M}\sum_{m=1}^{M} V_0^m$$

This estimate for V_0 has been proven to converge to the actual value of V_0 under much more general conditions than a single asset Black-Scholes model (see [Clément et al., 2002] and [?]). Note that the approximate standard error of the estimate is computed simply using the standard unbiased sample variance estimate. Namely,

$$SE_{V_0}^{MC} \approx \frac{1}{\sqrt{M}} \sqrt{\frac{\sum_{m=1}^{M} (V_0^m - V_0^{MC})^2}{M-1}}$$

Furthermore, we are also assured a central limit theorem ([Clément et al., 2002]). The mathematics behind this proof are a bit too complicated for me to understand or convey properly. However, a key condition on having this CLT is the independence of the simulated stock price paths.

Now that we have seen the algorithm in full, we will discuss some methods for improving it through variance reduction techniques.

4 Variance Reduction Techniques

In this section we examine two possible variance reduction techniques to improve the accuracy of our least-sqaures Monte Carlo estimate. Namely, we examine how moment matching and antithetic variates can be used within this problem to better our estimate of the American put Option price in a Black-Scholes market model.

4.1 Antithetic Variates

The idea behind antithetic variates is to reduce standard error of the Monte Carlo estimate by generating a random sample from a symmetric distribution and reflecting it about its centre. We then take both the original and reflected simulations to compute our Monte Carlo estimate. This leads to some form of error cancellation, as the original samples and the reflected samples are negatively correlated.

In our case, the random numbers we generate are the $Z_n^1, ..., Z_n^M \sim N(0,1)$, for each time step $n = 0, ..., N$, to simulate the stock prices. So to use the antithetic variates, we reduce the size of our random sample to $\frac{M}{2}$ so we are left with $Z_n^1, ..., Z_n^{\frac{M}{2}}$ and then take the remaining $\frac{M}{2}$ samples to be $-Z_1, ..., -Z_{\frac{M}{2}}$. We then proceed with the Longstaff-Schwartz algorithm as given in the previous section for each separate sequence of stock prices, one derived using the original values $\boldsymbol{Z}$ and the generated using $\boldsymbol{Z}$'s antithetic counterpart, $\tilde{Z} \equiv --\boldsymbol{Z}$. From this we obtain the resulting simulated option prices $(V_0^0, ..., V_0^{\frac{M}{2}})$ and $(\tilde{V}_0^0, ..., \tilde{V}_0^{\frac{M}{2}})$. It follows that the antithetic Monte Carlo estimate for V_0 is given by: $V_0 \approx V_0^{anti} \frac{1}{M} \sum_{m=1}^{M/2} (V_0^m + \tilde{V}_0^m)$ The approximate standard error of the antithetic estimate is then:

$$SE_{V_0} \approx SE_{V_0}^{anti} = \frac{1}{\sqrt{M}} \sqrt{\frac{\sum_{m=1}^{M} ((V_0^m + \tilde{V}_0^m) - V_0^{anti})^2}{M-1}}$$

Note that a CLT is maintained using these modified estimates for the mean and standard error, since the samples are only pairwise dependent, meaning each pair of samples is independent from every other pair of samples [Owen, 2018].

Moment Matching

Moment matching is another technique to reduce the variability of Monte Carlo estimates. This method involves transforming the simulated random samples $Z = (Z_n^1, ..., Z_n^M)$ at each $n = 0, ..., N$ so that the sample mean and sample standard deviation match the theoretical mean and standard deviation. Specifically:

$$\tilde{Z}_n^m = (Z_n^m - \overline{Z}_n)\frac{\sigma_Z}{\sigma_{Z_n}} + \mu_Z$$

where $\overline{Z}_n = \sum_m = 1^M Z_n^m$ and $\overline{\sigma_{Z_n}}^2 = \frac{1}{M-1}\sum_{m=1}^M (Z_n^m - \overline{Z}_n)$.

In our case, the Z_n^m are from a standard normal distribution so $\mu_Z = 0$ and $\sigma_Z = 1$, so our moment matching transformation of each Z_n^m is:

$$\tilde{Z}_n^m = \frac{Z_n^m - \overline{Z}_n}{\overline{\sigma_{Z_n}}}$$

We then simply use $\tilde{Z}_n^m$ to compute the simulated stock prices and proceed with the Longstaff-Schwartz algorithm as usual.

The upside of this method is that we are able to match both the mean and variance of the samples to the distribution we are trying to simulate. In this sense, it offers an advantage that antithetic variates do not, as antithetic variates only assure the sample mean matches the theoretical mean. The downside however is that by using $\overline{Z}_n$ and $\overline{\sigma}_{Z_n}$ to compute $\tilde{Z}_n^m$, the stock price paths are no longer independent from one another. This lack of independence cannot be remedied via a simple adjusted through a simple change in the form of the Monte Carlo estimate as we did for antithetic variates. This causes us to lose the central limit theorem property of the Longstaff-Schwartz estimate for V_0. The estimate for the option price will also be slightly biased, as the transformed $\tilde{Z}_n^m$ are no longer standard normally distributed. In the case of our simple model and claim, with a large enough number of simulations M this bias is quite small [Boyle et al., 1997].

To get some idea of the standard error to compare the accuracy of the moment matching method to antithetic and straight least-squares Monte Carlo, we must employ batching. Namely, we run the least squares Monte-Carlo moment matching algorithm for $M = ab$ iterations $a, b \in \mathbb{N}$, where a is the number of batches and b is the number of simulations per batch. The estimate for V_0, denoted V_0^{MM}, is the usual least-squares Monte Carlo estimate from the previous section. To estimate the variance, for $i = 1, ..., a$ let v_i be the Monte Carlo estimate for V_0 based on the i^{th} batch of simulations; meaning:

$$v_i = \sum_{m=b(i-1)+1}^{b \cdot i} \tilde{Z}_n^m$$

Now assuming b is large enough (approximately $\sqrt{M}$ [Boyle et al., 1997]), we can treat the v_i as independent samples. Hence their sample variance is:

$$\sigma_v^2 = \frac{b}{a-1}\sum_{i=1}^{a}(v_i - V_0^{MM})^2$$

So we now approximate the standard error of V_0^{MM} to be: $SE_{V_0}^{MM} = \frac{\sigma_v}{N}$ We can now apply the CLT to our Monte Carlo estimate using our batched standard error to obtain confidence intervals for the estimate.

Now that we have completed thoroughly discussed the theory surrounding our least squares Monte Carlo estimate and the variance reduction techniques we plan to apply, we will now look at the results using the aforementioned Netflix put option.

5 Implementation and Results

In the following section we look closely at the results achieved from implementing the Longstaff-Schwartz algorithm in Python to price an American put option written on Netflix stock. To begin, we will briefly discuss the model parameters we have selected and how we have chosen them.

5.1 Obtaining the Model Parameters

In order to implement least-squares Monte Carlo to price a real-world option, we must first approximate our model parameters. In a Black-Scholes model, recall the parameters we need to set are K, σ, S_0, T and r. As mentioned in the introduction, we are interested in an option that was written on November 23, 2018 with maturity February 22, 2018. We can use the `datetime` module in Python to use these values to obtain T. We also will be using historical data from the last 30 days to compute σ, so we set this date in Python as well.

```
# start date for old data
dayHist=datetime.date(2018,10,23)
# time 0: when option was written
day0=datetime.date(2018,11,23)
# time T: when option expires
dayT=datetime.date(2019,2,22)
```

Note that all time units will be in terms of days. We now use the web-scraping module `fix yahoo finance` to find the current (S_0) and historical Netflix stock prices. Also, since we are interested in an at-the-money option, we set $K = S_0$.

```
import fix_yahoo_finance as yf
fix_yahoo_finance
# symbol of underlying stock
sym='NFLX'
# get historical data to choose model parameters
dataHist = yf.download(sym,dayHist,day0)
# get historical prices of stock
sHist=dataHist['Open']
# number of historical prices
tHist=len(sHist)
sHist=sHist.values.reshape((tHist,))
# time 0 stick price is last historical value
S0=sHist[-1]
```

```
# strike price for at-the-money option
K=S0
# daily market risk-free rate of return
r=0.024/90
```

Note that the choice of r was found simply by taking the 3-month US treasury yields and dividing by 90 days [Bloomburg, 2018].

We obtain the historical prices of the stock to obtain a reasonable estimate for the volatility σ. Specifically, we take σ to be the standard deviation of the log asset returns over the previous 30 days

```
# daily log rate of return on asset
X=np.log(sHist[1:]/sHist[0])
# asset volatility estimate
sigma=np.std(X)
```

The resulting parameter values are

$$T = 64,\ \sigma = 0.067,\ r = 0.00027,\ S_0 = 260.11 \text{ and } K = 260.11$$

Note that we also take $N = T$, so each time step corresponds to one day. Now that we have our parameter values, we can implement and examine the Longstaff-Schwartz algorithm.

5.2 Results

The implementation of the Longstaff-Schwartz algorithm with and without moment-matching or antithetic variates in Python can be found in the appendix. The algorithms directly follow what was detailed in the previous sections.

To begin our analysis, we will run each version of the Longstaff-Schwartz algorithm with $M = 10,000$, $M = 100,000$ and $M = 1,000,000$. Note that for moment-matching we set $a = 100$ and $b = 100$ when $M = 10,000$, $a = 250$ and $b = 400$ when $M = 100,000$ and $a = 1,000$ and $b = 1,000$ when $M = 1,000,000$. Note that corresponding 95% confidence intervals are also given in table 4 in the Appendix.

LSM Estimate of V_0			
Algorithm	$M = 10,000$	$M = 100,000$	$M = 1,000,000$
Straight LSM	51.972986	51.682607	51.683480
LSM with Antithetic Variates	52.104953	51.748844	51.679378
LSM with Moment Matching	51.610329	51.551762	51.692436

Table 1: V_0 estimates using LSM algorithms.

Approximate Standard Error of LSM Estimate of V_0			
Algorithm	$M = 10,000$	$M = 100,000$	$M = 1,000,000$
Straight LSM	0.490557	0.152529	0.048352
LSM with Antithetic Variates	0.415633	0.129810	0.041018
LSM with Moment Matching	0.450873	0.151757	0.047829

Table 2: Approximated standard error of V_0 estimates using LSM algorithms.

As expected, we see a gradual decline in the standard error of the estimate as M increases. Another expected consequence of a larger M is a longer runtime of the algorithm. Specifically, one run all three algorithms with $M = 10,000$ takes less than a second to run, while the runtime for $M = 1,000,000$ is over 20 seconds. A runtime of 20 seconds is certainly manageable when computing a single value, however if we were interested in running these algorithms for, say, a variety of strike prices and exercise times, we would begin to notice the slowing effect. On the other hand, with $M = 1,000,000$ the estimates V_0 across all three algorithms differ only by 0.03, as can be seen in the last column of table 5.2 and the standard errors are 10 times smaller than those corresponding to $M = 10,000$. There is also clear evidence of diminishing returns in increasing the value of M, as can be seen in comparing the relative difference in the approximate standard errors of the estimates from $M = 10,000$ to $M = 100,000$ to $M = 100,000$ to $M = 1,000,000$ in table 2. This indicates that perhaps a choice of $M = 100,000$ will yield reasonable results without too much of a computational cost. In sum, the choice of M for the algorithm depends on what we will use it for and how long we are able to wait for results. If possible, a choice of will yield more accurate results. This accuracy can be very important in practice when dealing with large sums of money. However, smaller choices of M can still lead to reasonably accurate results and will be necessary if we are interested in applying the algorithm to anything more complicated than a single claim with all its parameters fixed in the Black-Scholes market model.

We now turn our attention to the results corresponding to the different versions of LSM. Upon first glance of table 2, it appears that the moment matching variance reduction technique resulted in little change to the standard error, while the antithetic variates offered a noticeable reduction in the size of the standard error. Furthermore, it seems the relative decrease in the standard errors due to increasing M are reasonably similar across all versions of the algorithm. This indicates that neither of the variance reduction techniques have much of an effect on the way the number of simulations, M, impacts the accuracy of the estimate.

Although these tabulated results seem to indicate a more accurate estimate arising from applying antithetic variates, we will look at several runs of the algorithm to confirm these suspicions. We elect to repeatedly run the algorithms with $M = 10,000$ for computational efficiency, since although the resulting estimates of V_0 will be less accurate than those produced with a larger M, this makes little difference in comparing the algorithms, as previously noted.

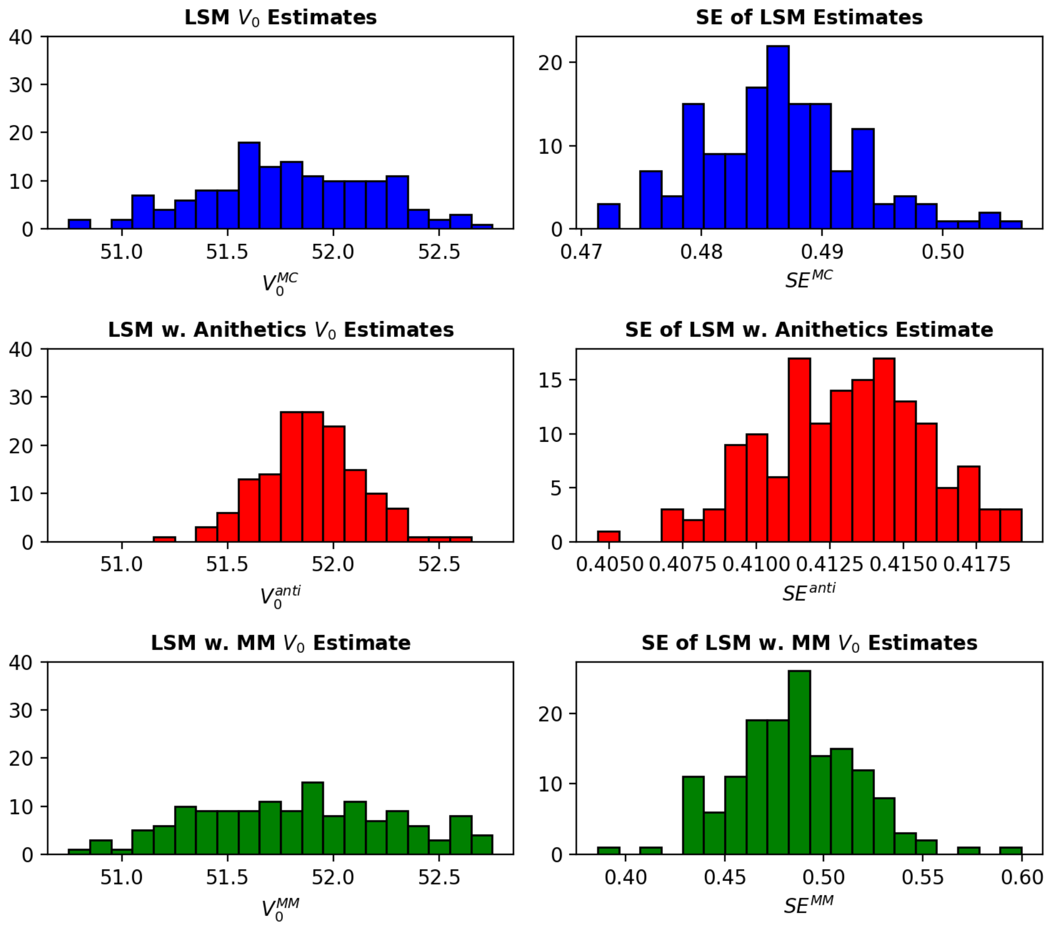

Figure 1: Histograms (each with 20 bins) from 150 LSM estimates (with $M = 10,000$ simulations) of V_0 and the corresponding approximate standard error across all three versions of LSM.

Averages over 150 runs of LSM with $M = 10,000$		
Algorithm	V_0 Estimate	Approx. SE of V_0 Estimate
Straight LSM	51.80924	0.487087
LSM with Antithetic Variates	51.9169	0.4131
LSM with Moment Matching	51.794296	0.489716

Table 3: Average V_0 estimate and average approximate standard error of the estimates for LSM algorithms.

The resulting histograms confirm our conjectures based upon a single run each algorithm. Namely, the algorithm which employs antithetic variates yields the most accurate results. This is evident in examining both histograms in red in figure 1. The antithetic estimates for V_0 are far more consistent, as their standard error estimates are much smaller (all within a range of 0.4 and 0.43) which is reflected in the tighter range of the corresponding histogram for the V_0 estimates. Furthermore, recall in table 5.2, the estimates for V_0 corresponding to $M = 1,000,000$ across all three versions of LSM ranged in value between 51.6 and 51.8. Looking at the histogram for V_0^{anti}, this range corresponds to then highest frequency bins. This reflects the antithetic variates technique's accuracy and efficiency. To further speak to its efficiency, recall from section 3 that we only require $\frac{M}{2}$ random samples from $N(0,1)$ to simulate the stock price (as the other $\frac{M}{2}$ samples are just the originals reflected in the y-axis). Although the remainder of the algorithm still involves working with M samples (as we must compute the stock and option price paths corresponding to both the $\boldsymbol{Z}$ and $-\boldsymbol{Z}$), it is still worth noting that the antithetic variates method requires half as many initial random samples and achieves more accurate results.

On the other hand, the moment matching estimates seemed to provide little to no improvement to the straight LSM algorithm in computing V_0 on average. As can be seen in table 5.2, the standard error of the method of moments estimates is very similar to that from using straight LSM. The approximate standard error of the estimates are also similar for moment matching and straight LSM for larger values of M, as shown in table 2. This phenomenon is unlikely due to the error in approximating the standard error for moment matching via batching, as the histograms for V_0^{MC} and V_0^{MM} have a similar spread, which is noticeable wider the spread for V_0^{anti}. However, the cause of the larger variation in the standard error estimates, as seen in histogram of SE^{MM}, is unclear and could be (but is not necessarily) due to batching.

The results found via antithetic variates seem to be the most accurate. Of course, using a larger number of simulations ($M = 1,000,000$ in our case) also leads to more accurate results. So the best estimate for the price of the American put option written on Netflix stock is \$51.69. Note that this price is unlikely to match the market price of this specific option in reality. This is not due to an issue with implementation as the Python code was tested using the toy example from Longstaff and Schwartz's original paper (see [Longstaff and Schwartz, 2001]), but due to problems inherent in the Black-Scholes market model. We will discuss this and possible improvements to the algorithm in our concluding remarks.

6 Conclusion

In sum, we found that employing antithetic variates to reduce the variance of the LSM estimate results in more consistent estimates with lower standard error when compared to the estimates derived via straight LSM and LSM with moment matching. These are by no means the only variance reduction techniques that can be applied to the Longstaff-Schwartz algorithm. In fact, there is a wide array of literature that explores these different variance reduction techniques being applied to LSM to estimate the price of American options (see [Lemieux and La, 2005], [Areal et al., 2008] and [Hilpisch, 2014]) varying degrees of success.

As noted at the end of the results section, this does not mean our final estimate of a price of \$51.69 for the option will match the reality of the market. This is in part attributed to the flaws of the the Black-Scholes market model, as it simplifies the reality of the market. For instance, its assumptions of a constant risk-free interest rate and constant volatility of the underlying asset intuitively do not match reality of the ever-changing market. Nevertheless, the Longstaff-Schwartz algorithm does not only apply in a Black-Scholes market model. The results of Longstaff and Schwartz's initial paper [Longstaff and Schwartz, 2001] are general enough to be applied to many different market models which vary in their complexity. However, these more complicated algorithms often require increasingly more convoluted numerical schemes and methodology to even obtain an estimate for the stock price. This leads to several potential sources of error, making it more difficult to specifically study the least-squares Monte Carlo approach and compare the variance reduction techniques. Hence why for this project we elected to stick to a simple Black-Scholes model, as a more complicated model would force the project's focus away from just Monte Carlo-based analysis. The Longstaff-Schwartz algorithm can also be applied to price a multitude of financial derivatives. The algorithm specializes in pricing path-dependent options in continuous market models. Similarly to the vanilla American option, the price of these securities often have no closed-form, so Monte-Carlo simulation is required to value them.

Ultimately, although our estimate for the price of a Netflix American put option may not be the most accurate, it allowed us to explore how least-squares Monte Carlo can be applied to option pricing via the Longstaff-Schwartz algorithm. Furthermore, flexibility of the Longstaff-Schwartz algorithm allows us to improve upon the estimate through different variance reduction techniques and the employment of more complex market models. This flexibility also allows us to apply the algorithm to more complex claims as well, making the Longstaff-Schwartz algorithm an important tool in derivative pricing.

7 Appendix

7.1 Tables

Approximate 95% CI for LSM Estimate of V_0			
Algorithm	$M = 10,000$	$M = 100,000$	$M = 1,000,000$
Straight LSM	$(51.011493, 52.934478)$	$(51.511482, 52.109395)$	$(51.588710, 51.778251)$
LSM with Antithetic Variates	$(51.290312, 52.919594)$	$(51.492010, 52.000867)$	$(51.598983, 51.759773)$
LSM with Moment Matching	$(52.494041, 50.726618)$	$(51.254319, 51.849205)$	$(51.598691, 51.786181)$

Table 4: Approximate 95% confidence intervals for V_0 estimates using LSM algorithms.

7.2 Pyhton Code

```
### Options Module

import numpy as np


# Payoff of american put option

def payoff(S,K):

    return(np.maximum(K-S,0))




def GBM(So,dt,mu,sig,n,m):


    # get (n+1 x m/2) iid standard normal random samples

    Z=np.random.standard_normal((n+1,m))


    #Transform to simulate stoc prices in BS model

    s=So*np.exp(np.cumsum((mu-0.5*sig**2)*dt+sig*np.sqrt(dt)*Z, axis=0))

    return(s)




def GBManti(So,dt,mu,sig,n,m):

    # get (n+1 x m/2) iid standard normal random samples

    Z=np.random.standard_normal((n+1,int(m/2)))


    # s1 is stock prices corresponding to Z

    s1=So*np.exp(np.cumsum((mu-0.5*sig**2)*dt+sig*np.sqrt(dt)*Z, axis=0))
    # s2 is stock prices corresponding to -Z
    s2=So*np.exp(np.cumsum((mu-0.5*sig**2)*dt+sig*np.sqrt(dt)*(-1*Z), axis=0))
    return(s1,s2)


def GBMmm(So,dt,mu,sig,n,m):
    Z1=np.random.standard_normal((n+1,m))

    # Obtain Sample mean and variance of Z
    Zbar=np.mean(Z1, axis=1)
    sigZ=np.std(Z1, axis=1)

    ZbarM=np.ones((n+1,m))
    sigZM=np.ones((n+1,m))

    for i in range(0,n+1):
        ZbarM[i,:]=np.ones(m)*Zbar[i]
        sigZM[i,:]=np.ones(m)*sigZ[i]

    #transformed Z
    Z=(Z1-ZbarM)/sigZM

    # get corresponding stock prices
    s=So*np.exp(np.cumsum((mu-0.5*sig**2)*dt+sig*np.sqrt(dt)*(Z+ZbarM), axis=0))
    return(s)
############################################################
# LSM Module
import numpy as np
from Options import GBM, payoff, GBManti, GBMmm


def LSMsimple(S0,T,r,sigma,K,N,M):
    # time step size
    dt=T/N

    # get stock prices
    S=GBM(S0,dt,r,sigma,N,M)
```

```
# V is matrix of simulated option prices
V=np.zeros((N+1,M))

# Final row of V contains time T option value,
# which is just the payoff of the option
V[-1]=payoff(S[-1],K)

# get option prices using backwards recursive LSM algorithm
for n in range(N-1, 0, -1):
    # get beta: the coefficients of least-squares regression model with
    # polynomial basis functions
    beta=np.polyfit(S[n],V[n+1]*np.exp(-r*dt),5)

    # Approximate value of option if we do not exercise using
    # polynomial regression coefficents we just obtained
    Cont=np.polyval(beta,S[n])

    # Time n value of option
    V[n]=np.where(payoff(S[n],K)>Cont, payoff(S[n],K),V[n+1]*np.exp(-r*dt))

# Discount time 1 option value to time 0
V[0]=V[1]*np.exp(-r*dt)

# Monte Carlo estimate of option price at all discretized
# points in time
Vmc=np.mean(V[0])

# Standard error of Monte Carlo estiamte
se=(1/np.sqrt(M))*np.std(V[0])

# Upper and lower 95% confidence interval bounds for MC estiamte
lower=Vmc-se*1.96
upper=Vmc+se*1.96

return(Vmc,se,lower,upper)
```

```
def LSManti(S0,T,r,sigma,K,N,M):
    dt=T/N # time step length

    # get antithetic variates stock prices
    S=GBManti(S0,dt,r,sigma,N,M)
    S=np.array(S)
    # V is matrix of simulated option prices
    V=np.zeros((2,N+1,int(M/2)))

    # Final row of V contains time T option value,
    # which is just the payoff of the option
    V[0,-1,:]=payoff(S[0,-1,:],K)
    V[1,-1,:]=payoff(S[1,-1,:],K)

    for n in range(N-1, 0, -1):
        # get beta: the coefficients of least-squares regression model with
        # polynomial basis functions
        beta0=np.polyfit(S[0,n,:],V[0,n+1,:]*np.exp(-r*dt),5)
        beta1=np.polyfit(S[1,n,:],V[1,n+1,:]*np.exp(-r*dt),5)

        # Approximate value of option if we do not exercise using
        # polynomial regression coefficents we just obtained
        Cont0=np.polyval(beta0,S[0,n,:])
        Cont1=np.polyval(beta1,S[1,n,:])

        # Time n value of option
        V[0,n,:]=np.where(payoff(S[0,n,:],K)>Cont0, payoff(S[0,n,:],K),V[0,n+1,:]*np.exp(-r*dt))
        V[1,n,:]=np.where(payoff(S[1,n,:],K)>Cont1, payoff(S[1,n,:],K),V[1,n+1,:]*np.exp(-r*dt))


    # Discount time 1 option value to time 0
    V[0,0,:]=V[0,1,:]*np.exp(-r*dt)
    V[1,0,:]=V[1,1,:]*np.exp(-r*dt)

    V0=V[0,0,:]
    V1=V[1,0,:]
    # antithetic MC estimate of V0
    y=(V0+V1)
    Vmc=(1/M)*np.sum(y)

    # estimate se of V0
    var=(1/(M-1))*np.sum((y-Vmc)**2)
    se=np.sqrt(1/M)*np.sqrt(var)

    # Upper and lower 95% confidence interval bounds for estiamte
    lower=Vmc-se*1.96
    upper=Vmc+se*1.96

    return(Vmc,se,lower,upper)

def LSMmm(S0,T,r,sigma,K,N,M,a,b):
    dt=T/N # time step length



    S=GBMmm(S0,dt,r,sigma,N,M)



    V=np.zeros((N+1,M))

    V[-1]=payoff(S[-1],K)

    for n in range(N-1, 0, -1):
        # get beta: the coefficients of least-squares regression model with
        # polynomial basis functions
        beta=np.polyfit(S[n],V[n+1]*np.exp(-r*dt),5)

        # Approximate value of option if we do not exercise using
        # polynomial regression coefficents we just obtained
        Cont=np.polyval(beta,S[n])

        # Time n value of option
        V[n]=np.where(payoff(S[n],K)>Cont, payoff(S[n],K),V[n+1]*np.exp(-r*dt))
    # Discount time 1 option value to time 0
    V[0]=V[1]*np.exp(-r*dt)
    Vo=V[0]

    # moment matching estimate for V0
    Vmc=np.mean(Vo)

    # get batch means
    vbatch=np.zeros(a)
    for i in range(1,a+1):
        vbatch[i-1]=np.mean(Vo[b*(i-1):b*i])

    # get se estimate for V0 using se of batches
    var=(b/(a-1))*np.sum((vbatch - Vmc)**2)
    se=(1/np.sqrt(M))*np.sqrt(var)

    # Upper and lower 95% confidence interval bounds for estimate
    lower=Vmc-se*1.96
    upper=Vmc+se*1.96

    return(Vmc,se,lower,upper)
############################################################
import numpy as np
import datetime
import matplotlib.pyplot as plt
import fix_yahoo_finance as yf
import pandas as pd
from LSM import LSMsimple, LSManti, LSMmm

np.random.seed(12345)
# start date for old data
dayHist=datetime.date(2018,10,23)

# time 0: when option was written
day0=datetime.date(2018,11,23)

# time T: when option expires
```

```
dayT=datetime.date(2019,2,22)

# symbol of underlying stock
sym='NFLX'

# daily market risk-free rate of return
# found via https://www.bloomberg.com/markets/rates-bonds/government-bonds/us
r=0.024/90

#%%
# get historical data to choose model parameters
dataHist = yf.download(sym,dayHist,day0)
#%%
# time to expiry in days
T=len(pd.bdate_range(day0, dayT))-2




# get historical prices of stock
sHist=dataHist['Open']
# number of historical prices
tHist=len(sHist)
sHist=sHist.values.reshape((tHist,))

# daily log rate of return on asset
X=np.log(sHist[1:]/sHist[0])

# asset volatility estimate
sigma=np.std(X)

# time 0 stick price is last historical value
S0=sHist[-1]

# strike price for at-the-money option
K=S0
```

```
 # number of simulations

M=10000

# number of time steps=number of days to expriy

N=T


# Output parameter values

print('T:', '%3f'%T)

print('sigma:', '%2f'%sigma)

print('r:', '%2f'%r)

print('So:', '%2f'%S0)

print('K:', '%2f'%K)



# time step length

dt=T/N


#%%

# simple LSM

mc,ste,LB,UB=LSMsimple(S0,T,r,sigma,K,N,M)


# LSM with anithetic variates

mc2,ste2,LB2,UB2=LSManti(S0,T,r,sigma,K,N,M)



# LSM with moment-matching


# number of batches to get se

a=100

# number of simulations per batch to get se

b=100

mc3,ste3,UB3,LB3=LSMmm(S0,T,r,sigma,K,N,M, a, b)


#%%

# Print Results

print('LSM with M = 10000: \n', 'Estimate of Vo: ', '%2f'%mc,

      '\n', 'Approximate standard error of estimate: ', '%2f'%ste, '\n',

      'Approximate 95 % for estimate of Vo: (', '%2f'%LB, ',', '%2f'%UB, ')','\n','\n' )
```

```
print('LSM using Antithetic Variates with M = 10000: \n', 'Estimate of Vo: ', '%2f'%mc2,
      '\n', 'Approximate standard error of estimate: ', '%2f'%ste2, '\n',
      'Approximate 95 % for estimate of Vo: (', '%2f'%LB2, ',', '%2f'%UB2, ')','\n','\n' )

print('LSM using Moment Matching with M = 10000: \n', 'Estimate of Vo: ', '%2f'%mc3,
      '\n', 'Approximate standard error of estimate: ', '%2f'%ste3, '\n',
      'Approximate 95 % for estimate of Vo: (', '%2f'%LB3, ',', '%2f'%UB3, ')'  )

#%%
mcV=[]
seV=[]
mcV2=[]
seV2=[]
mcV3=[]
seV3=[]
for i in range(0,150):
    temp=LSMsimple(S0,T,r,sigma,K,N,M)
    mcV.append(temp[0])
    seV.append(temp[1])

    temp2=LSManti(S0,T,r,sigma,K,N,M)
    mcV2.append(temp2[0])
    seV2.append(temp2[1])

    temp3=LSMmm(S0,T,r,sigma,K,N,M,a,b)
    mcV3.append(temp3[0])
    seV3.append(temp3[1])

#%%
print('Using LSM over 150 iterations the average: \n', 'Estimate of Vo is: ', np.mean(mcV), '\n',
      'Approximate standard error of estimate is: ', np.mean(seV), '\n', '\n')

print('Using LSM with antithetic variates over 150 iterations the average: \n', 'Estimate of Vo is: ', np.mean(mcV2),
      '\n','Approximate standard error of estimate is: ', np.mean(seV2), '\n', '\n')

print('Using LSM with moment matching over 150 iterations the average: \n', 'Estimate of Vo is: ', np.mean(mcV3),'\n'
      'Approximate standard error of estimate is: ', np.mean(seV3), '\n', '\n')
#%%
fig=plt.figure(figsize=(8, 7))
ax = fig.add_subplot(111)
#st = fig.suptitle("Histograms over 50 LSM Estimates with M=10,000 Simulations",fontweight='bold')

sub1 = fig.add_subplot(321)
sub1.set_title('LSM $V_0$ Estimates', fontsize=10,fontweight='bold')
sub1.hist(mcV, edgecolor='black', bins=20, color='b', range=(50.75,52.75))
sub1.set_ylim([0, 40])
sub1.set_xlabel('$V_0^{MC}$')

sub2 = fig.add_subplot(322)
sub2.set_title('SE of LSM Estimates', fontsize=10,fontweight='bold')
sub2.hist(seV, edgecolor='black', bins=20, color='b')
sub2.set_xlabel('$SE^{MC}$')

sub3 = fig.add_subplot(323)
sub3.set_title('LSM w. Anithetics $V_0$ Estimates', fontsize=10,fontweight='bold')
sub3.hist(mcV2, edgecolor='black', bins=20, color='r', range=(50.75,52.75))
sub3.set_ylim([0, 40])
sub3.set_xlabel('$V_0^{anti}$')

sub4 = fig.add_subplot(324)
sub4.set_title('SE of LSM w. Anithetics Estimate', fontsize=10,fontweight='bold')
sub4.hist(seV2, edgecolor='black', bins=20, color='r')
sub4.set_xlabel('$SE^{anti}$')

sub5 = fig.add_subplot(325)
sub5.set_title('LSM w. MM $V_0$ Estimate', fontsize=10,fontweight='bold')
sub5.hist(mcV3, edgecolor='black', bins=20, color='g', range=(50.75,52.75))
sub5.set_ylim([0, 40])
sub5.set_xlabel('$V_0^{MM}$')

sub6 = fig.add_subplot(326)
sub6.set_title('SE of LSM w. MM $V_0$ Estimates', fontsize=10,fontweight='bold')
sub6.hist(seV3, edgecolor='black', bins=20, color='g')
```

```
sub6.set_xlabel('$V_0^{MM}$')



ax.spines['top'].set_color('none')
ax.spines['bottom'].set_color('none')
ax.spines['left'].set_color('none')
ax.spines['right'].set_color('none')
ax.tick_params(labelcolor='w', top='off', bottom='off', left='off', right='off')
plt.tight_layout()
fig.show()
#%%
Mi=100000
ai=250
bi=400
mci,stei,LBi,UBi=LSMsimple(S0,T,r,sigma,K,N,Mi)

mc2i,ste2i,LB2i,UB2i=LSManti(S0,T,r,sigma,K,N,Mi)

mc3i,ste3i,LB3i,UB3i=LSMmm(S0,T,r,sigma,K,N,Mi, a, b)

#%%
print('LSM with M = 100000: \n', 'Estimate of Vo: ', '%2f'%mci,
      '\n', 'Approximate standard error of estimate: ', '%2f'%stei, '\n',
      'Approximate 95 % for estimate of Vo: (', '%2f'%LBi, ',', '%2f'%UBi, ')','\n','\n' )

print('LSM using Antithetic Variates with M = 100000: \n', 'Estimate of Vo: ', '%2f'%mc2i,
      '\n', 'Approximate standard error of estimate: ', '%2f'%ste2i, '\n',
      'Approximate 95 % for estimate of Vo: (', '%2f'%LB2i, ',', '%2f'%UB2i, ')','\n','\n' )

print('LSM using Moment Matching with M = 100000: \n', 'Estimate of Vo: ', '%2f'%mc3i,
      '\n', 'Approximate standard error of estimate: ', '%2f'%ste3i, '\n',
      'Approximate 95 % for estimate of Vo: (', '%2f'%LB3i, ',', '%2f'%UB3i, ')'  )
#%%

Mii=1000000
aii=1000
bii=1000
```

```
mcii,steii,LBii,UBii=LSMsimple(S0,T,r,sigma,K,N,Mii)

mc2ii,ste2ii,LB2ii,UB2ii=LSManti(S0,T,r,sigma,K,N,Mii)

mc3ii,ste3ii,LB3ii,UB3ii=LSMmm(S0,T,r,sigma,K,N,Mii, aii, bii)

#%%
print('LSM with M = 1000000: \n', 'Estimate of Vo: ', '%2f'%mcii,
      '\n', 'Approximate standard error of estimate: ', '%2f'%steii, '\n',
      'Approximate 95 % for estimate of Vo: (', '%2f'%LBii, ',', '%2f'%UBii, ')','\n','\n' )

print('LSM using Antithetic Variates with M = 1000000: \n', 'Estimate of Vo: ', '%2f'%mc2ii,
      '\n', 'Approximate standard error of estimate: ', '%2f'%ste2ii, '\n',
      'Approximate 95 % for estimate of Vo: (', '%2f'%LB2ii, ',', '%2f'%UB2ii, ')','\n','\n' )

print('LSM using Moment Matching with M = 1000000: \n', 'Estimate of Vo: ', '%2f'%mc3ii,
      '\n', 'Approximate standard error of estimate: ', '%2f'%ste3ii, '\n',
      'Approximate 95 % for estimate of Vo: (', '%2f'%LB3ii, ',', '%2f'%UB3ii, ')'  )
```

7.3 Output from Print Statements

```
# Parameter Values
T: 64.000000
sigma: 0.066839
r: 0.000267
So: 260.109985
K: 260.109985
#####################################################

LSM with M = 10000:
 Estimate of Vo:  51.972986
 Approximate standard error of estimate:  0.490557
 Approximate 95 % for estimate of Vo: ( 51.011493 , 52.934478 )



LSM using Antithetic Variates with M = 10000:
 Estimate of Vo:  52.104953
 Approximate standard error of estimate:  0.415633
 Approximate 95 % for estimate of Vo: ( 51.290312 , 52.919594 )



LSM using Moment Matching with M = 10000:

 Estimate of Vo:  51.610329

 Approximate standard error of estimate:  0.450873

 Approximate 95 % for estimate of Vo: ( 52.494041 , 50.726618 )

 #####################################################

 LSM with M = 100000:

 Estimate of Vo:  51.810438

 Approximate standard error of estimate:  0.152529

 Approximate 95 % for estimate of Vo: ( 51.511482 , 52.109395 )



LSM using Antithetic Variates with M = 100000:

 Estimate of Vo:  51.746439

 Approximate standard error of estimate:  0.129810

 Approximate 95 % for estimate of Vo: ( 51.492010 , 52.000867 )



LSM using Moment Matching with M = 100000:

 Estimate of Vo:  51.551762

 Approximate standard error of estimate:  0.151757

 Approximate 95 % for estimate of Vo: ( 51.254319 , 51.849205 )

 #####################################################

 LSM with M = 1000000:

 Estimate of Vo:  51.683480

 Approximate standard error of estimate:  0.048352

 Approximate 95 % CI for estimate of Vo: ( 51.588710 , 51.778251 )



LSM using Antithetic Variates with M = 1000000:

 Estimate of Vo:  51.679378

 Approximate standard error of estimate:  0.041018

 Approximate 95 % CI for estimate of Vo: ( 51.598983 , 51.759773 )
```

```
LSM using Moment Matching with M = 1000000:

 Estimate of Vo:  51.692436

 Approximate standard error of estimate:  0.047829

 Approximate 95 % for estimate of Vo: ( 51.598691 , 51.786181 )


 ####################################################


 Using LSM over 150 iterations the average:

 Estimate of Vo is:  51.809240217906904

 Approximate standard error of estimate is:  0.4870872768465016




Using LSM with antithetic variates over 150 iterations the average:

 Estimate of Vo is:  51.916937479563266

 Approximate standard error of estimate is:  0.4131222285922192




Using LSM with moment matching over 150 iterations the average:

 Estimate of Vo is:  51.79429626740034

 Approximate standard error of estimate is:  0.4897163736765682
```

8 Bibliography

References

[Areal et al., 2008] Areal, N., Rodrigues, A., and Armada, M. (2008). On improving the least squares monte carlo option valuation method. *Review of Derivatives Research*, 11:119–151.

[Black, 1973] Black, Fischer; Scholes, M. (1973). The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654.

[Bloomburg, 2018] Bloomburg (2018). United states rates bonds.

[Boyle et al., 1997] Boyle, P., Broadie, M., and Glasserman, P. (1997). Monte carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 21(8):1267 – 1321. Computational financial modelling.

[Clément et al., 2002] Clément, E., Lamberton, D., and Protter, P. (2002). An analysis of a least squares regression method for american option pricing. *Finance and Stochastics*, 6(4):449–471.

[Flegal and Jones, 2010] Flegal, J. M. and Jones, G. L. (2010). Batch means and spectral variance estimators in markov chain monte carlo. *Ann. Statist.*, 38(2):1034–1070.

[Hilpisch, 2014] Hilpisch, Y. (2014). *Python for Finance: Analyze Big Financial Data*. O'Reilly Media, Inc., 1st edition.

[Lemieux and La, 2005] Lemieux, C. and La, J. (2005). A study of variance reduction techniques for american option pricing. In *Proceedings of the Winter Simulation Conference, 2005.*, pages 8 pp.–.

[Longstaff and Schwartz, 2001] Longstaff, F. A. and Schwartz, E. S. (2001). Valuing american options by simulation: A simple least-squares approach. *Review of Financial Studies*, pages 113–147.

[Owen, 2018] Owen, A. (2009-2013, 2018). Variance reduction.

[Shreve, 2004] Shreve, S. E. (2004). *Stochastic calculus for finance 2, Continuous-time models*. Springer, New York, NY; Heidelberg.

[Stentoft, 2004] Stentoft, L. (2004). Convergence of the least squares monte carlo approach to american option valuation. *Management Science*, 50:1193–1203.

[Ware, 2017] Ware, T. (2017). Monte carlo methods: Stat 583/683.