Q3_main

2024-10-30

3. Least-Squares Monte Carlo Algorithm (LSM) on Put Option

3.1 Objective and Motivation

In this section, we implement the Least Squares Monte Carlo (LSM) algorithm based on the method outlined in the paper "Valuing American Options by Simulation: A Simple Least-Squares Approach" by Longstaff and Schwartz. The LSM approach is highly effective in pricing American options, as it estimates the conditional expectation of continuation payoffs, which is crucial for determining the optimal exercise strategy. This aligns with the fundamental asset pricing theorem, where the conditional expectation under the risk-neutral measure ensures that the stock price paths $S_t^* = S_t e^{-rt}$ are modeled as martingales, reflecting no-arbitrage conditions. By leveraging the martingale property and performing cross-sectional regression on simulated paths, LSM accurately captures the fair price of the option's continuation payoff.

It is important to note that the goal of this report is not to summarize or restate the details of the LSM algorithm as presented by Longstaff and Schwartz. Rather, our primary objective is to apply the LSM method iteratively to calculate the expected value of a put option across various settings. We experiment with different combinations of parameters, such as the number of stock paths (n), the number of regressors (k), and the number of time steps (m). We will present our findings and explore how variations in these parameters affect the outcomes of the simulation. Throughout the report, we will also discuss the application of variance reduction techniques and our own data processing choices where appropriate.

One potential drawback of the LSM algorithm, if implemented naively, is its computational expense. As both the number of paths and time steps increase, the computational cost grows significantly due to the need for performing least-squares regression at each recursive step. We will also address these challenges by presenting our efforts to improve the computational efficiency of the algorithm and the resulting performance enhancements.

3.2 Generating Stock Path

Each stock path follows a geometric brownian motion under the risk-neutral measure, complying to no-arbitrage and ensuring the discounted stock S^* paths are martingales. We define our stock process for each discrete time step Δt as

$$\Delta S_t = (r - q)S_t\Delta t + \sigma S_t\sqrt{\Delta t} * \phi_t$$

where S_t is our stock value, r and q is our constant risk-free rate and continuous dividend rate, and ϕ is random standard normal variable.

Variance Reduction 1: We generate our stocks paths by creating an antithetic counterpart to ϕ_t^i for each ϕ_t^i generaated. Specifically for each time step, if we require n paths, we only need to generate $n/2$ values of $\phi_t^1, \phi_t^1, ...\phi_t^{n/2}$, and take the negative for the other half. By introducing negative correlation between paired outcomes, antithetic variates balance fluctuations in opposite directions, reducing variance and enabling the Monte Carlo estimator to converge more efficiently to the true value without increasing the sample size.

The implementation of the stock path simulation is shown in Snippet 3.1.

Snippet 3.1: Generating stock path matrix

```
simulate_S_paths <- function(S0,r,q,sigma,n,m,delta_t) {
  set.seed(3)
  S = rep(S0,n)
  S_mat = matrix(0, nrow = n, ncol = m)
  for (i in 1:m) {
    # + antithetic paths
    z_temp = rnorm(n/2)
    z = c(z_temp, -z_temp)
    delta_S =  S * (r-q) * delta_t + sigma * S * sqrt(delta_t) * z
    S = S + delta_S
    S_mat[,i] = S
  }
  return(S_mat)
}
```

3.3 Choice of Basis Function and Scaling Stock Prices

In our experiment, we ran our least squares method by representing our regressors as basis functions calculated from our underlying stock prices $S_t = S_t^1, S_t^2, ..., S_t^n$ for each time step t. Although our primary analysis will be based on using Chebyshev polynomials as our basis function, we did run simulations on Laguerre polynomials (as suggested in the paper) as well. We concluded that the converged put option values are very similar to each other, and the discrepancy between the two choices of basis functions is very small. Both polynomials are orthogonal, which helps reduce collinearity between the regressors. We omit the use of regular polynomials since they are prone to collinearity. Using Chebyshev and Laguerre polynomials has proven to offer numerical stability in performing least squares regression. For interested readers, we include our LSM results using Laguerre polynomials in Appendix 3.2.

Before applying Chebyshev polynomial conversions on the stock prices, we apply min-max scaling with boundary $[-1, 1]$ since Chebyshev polynomials are orthogonal between $[-1, 1]$ and minimize the approximation error within these boundaries. For Laguerre polynomials, we apply min-max scaling with boundary $[0, 1]$, since they are designed to take positive values $[0, \infty]$, but because they consist of exponential terms, we cap our normalized value within 1, ensuring numerical stability.

We demonstrate our implementations of our basis functions and scaling functions in Snippet 3.2.

Snippet 3.2: Basis Functions and min-max scaler

```
mm_scaler <- function(x) {

  xmin = min(x)
  xmax = max(x)
  a = 2 / (xmax - xmin)
  b = 1 - a * xmax
  return( a*x+b)
}

laguerre_basis <- function(k, x) {
  laguerre_pi <- function(i, x) {
    if (i == 0) return(exp(-x / 2))
    coeff = sapply(0:i, function(j) ((-1)^j * choose(i, j) * x^j) / factorial(j))
    return(exp(-x / 2) * rowSums(coeff))
  }
  # Use sapply to compute Laguerre polynomials and convert result to a matrix
  result = sapply(0:(k - 1), function(i) laguerre_pi(i, x))
```

```
  return(as.matrix(result))  # Convert to matrix
}

chebyshev_basis <- function(k, x) {

  n = length(x)
  X = matrix(0, nrow = n, ncol = k + 1)
  X[, 1] = 1
  if (k > 0) {
    X[, 2] = x
  }
  if (k > 1) {
    for (i in 3:(k + 1)) {
      X[, i] = 2 * x * X[, i - 1] - X[, i - 2]
    }
  }
  return(X[, -1])
}
```

3.4 Least Squares Monte Carlo Algorithm

In this section, we briefly go through our implementation of the LSM algorithm. We start by generating our n stock paths with m time steps and pre-calculate our payoff matrix. We apply backward recursive steps starting from the $m-1$th step. As for American options, we are concerned with stock paths that give us the opportunity to early exercise; paths that are in-the-money (ITM) at any time step. The idea is we want to compare the these ITM paths' payoffs with the conditional expectation of (discounted) continuation payoffs ($\hat{V}^t_{cont.}\mathbb{E}[V^t_{cont.}|X_{S_t}]$), and decide whether or not to exercise at at current time step t or at the future $t+a$ step, where $t+a$ will depend on our tracked stopping time for each stock path . We generate our conditional expectations by running least squares on our transformed S_t's in the form of Chebyshev Polynomials. The parameter k will determine the number of regressors, and hence will specify the order of the polynomials. Through out our report, we denote k as the number of regressors not including the constant term, so k specifically is the number of polynomial regressors. We also define a vector "exercise_times" to keep track of our stopping times for each stock path. At any time step and stock path, if the current payoff exceeds $\hat{V}^t_{cont.}$, then we update our stopping time by replacing stopping time $t+a$ with t. Once all time steps have been recursively traversed, we discount all the payoffs of every stock path with reference to our tracked stopping times and compute the following measures: the expectation and standard error of the discounted payoffs, percentage of stock paths that have been early exercised, and exercise times. The code snippet of our implementation can be viewed in Snippet 3.

Variance Reduction 2: Although we can use the entire n stock paths as our input to the least squares algorithm, we only used samples that are ITM. Using only ITM stock paths in LSM improves the accuracy of the regression by focusing on paths where the option holder might exercise. OTM paths contribute no useful information since their payoffs are always zero, adding unnecessary noise to the estimation. Limiting the regression to ITM paths reduces variance, leading to more precise estimates of the continuation value and a better exercise strategy. This approach also improves computational efficiency by decreasing the number of data points used in the regression.

Snippet 3.3: LSM Algorithm Implementation

```
LSM_put <- function(S0,K,r,q,sigma,t,n,m, k_regressors, basis_func) {
  run_ols <- function(X, Y) { # function to generate condtional expectation values using least squares
    beta <- solve(t(X) %*% X, t(X) %*% Y)
    return(X %*% beta)
  }
  # Function to run the Least-Squares Monte Carlo
  delta_t = t/m # time steps
  S_mat = simulate_S_paths(S0,r,q,sigma,n,m,delta_t)  # simulate stock paths
  exercise_times = rep(m,n) # intialize stopping times at expiration

  # create payoff dataframe for all discrete times
  payoff_mat = pmax(K-S_mat,0)

  # Recursively loop backwards to apply LSM
  for (i in (m-1):1) {
    itm_idx = payoff_mat[,i] > 0 # find ITM idx

    # get future payoffs according to current stopping times
    future_cashflows = payoff_mat[cbind(1:n, exercise_times)][itm_idx]

    # get times to discount according to current stopping times
    discount_times = delta_t * (exercise_times - i)

    # define target as present value of future payoffs
    Y = future_cashflows * exp(-r*discount_times[itm_idx])

    # filter only ITM underlying stock prices
    S_itm = S_mat[,i][itm_idx]

    # Create Laguerre polynomial regressors matrix
    X = basis_func(k_regressors, mm_scaler(S_itm))

    # Run OLS and calculate conditional expectation of Y|X
    cond_exp_Y = run_ols(cbind(1, X), Y)

    # If current payoff exceeds E[Y|X], then exercise now, if not in the future
    # To implement this logic, we update our stopping times
    current_itm_payoff = payoff_mat[,i][itm_idx]
    exercise_times[itm_idx] = ifelse(current_itm_payoff > cond_exp_Y, i, exercise_times[itm_idx])
  }
  # get future payoffs according to final stopping times, and discount them
  payoff_decisions = payoff_mat[cbind(1:n, exercise_times)]
  discount_times = delta_t * (exercise_times - i)
  option_path_values = payoff_decisions * exp(-r*discount_times)

  # option value is the mean of all option present values from each path
  option_value = mean(option_path_values)
  se = sd(option_path_values) / sqrt(n)

  # % of paths that are early exercised
  early_exercise_portion = mean(exercise_times < m)
  return(list(value = option_value, se=se, early_portion=early_exercise_portion,
```

```
                ee_times=exercise_times))}
```

3.5 Computational Efficiency Improvements

In this section, we explain our implementations aimed at increasing the computational efficiency and speed of our LSM algorithm. Our original implementation of the algorithm relied primarily on manipulating dataframe objects and passing our regressors' dataframe into R's "lm" least squares package. This method proved to be underwhelming in terms of computational speed and allocated unnecessary memory overhead in R (or, in fact, in any scientific programming language). For interested readers, the original implementation of the algorithm, including the basis function implementations, can be found in Appendix 3.1. However, we argue that starting the prototyping of the algorithm with dataframes is preferred, as it is easier to debug and more intuitive for the developer. For instance, it allows us to check the least squares results to verify whether numerical instability persists and whether the coefficients produced are stable.

After ensuring that our algorithm computations were stable and the results sound, we shifted our approach from working with dataframes to matrices, as seen in Snippets 3.2 and 3.3. In this updated approach, the entire code now consists solely of matrix computations. In particular, we discarded the "lm" package and implemented our own least squares function, solving for the coefficients using the R function "solve" and computing the dot product between the Chebyshev polynomials and coefficients. This change resulted in a significant improvement in computational efficiency.

We compare the computational speed between the "dataframe" and "matrix" implementations with three regressors in Table 3.1 and Figure 3.1 by using our most precise LSM pricer(n=256000, k=3). Table 3.1 shows the speed in terms of option prices evaluated per second as the number of m time steps increase. Figure 3.1 illustrates this in log10 scaling. As an example, for $m = 80$, the speed increased from 0.01 to 0.31 options prices per second. This means that for $n = 256000$, the corresponding runtime will have decreased from around 187.1 to 3.2 seconds. As we will see later, for $n = 256000$, the standard error of the options price is very small and hence at 3.2 seconds runtime. We compare the runtimes for $n = 256000$ and three regressors in Table 3.2.

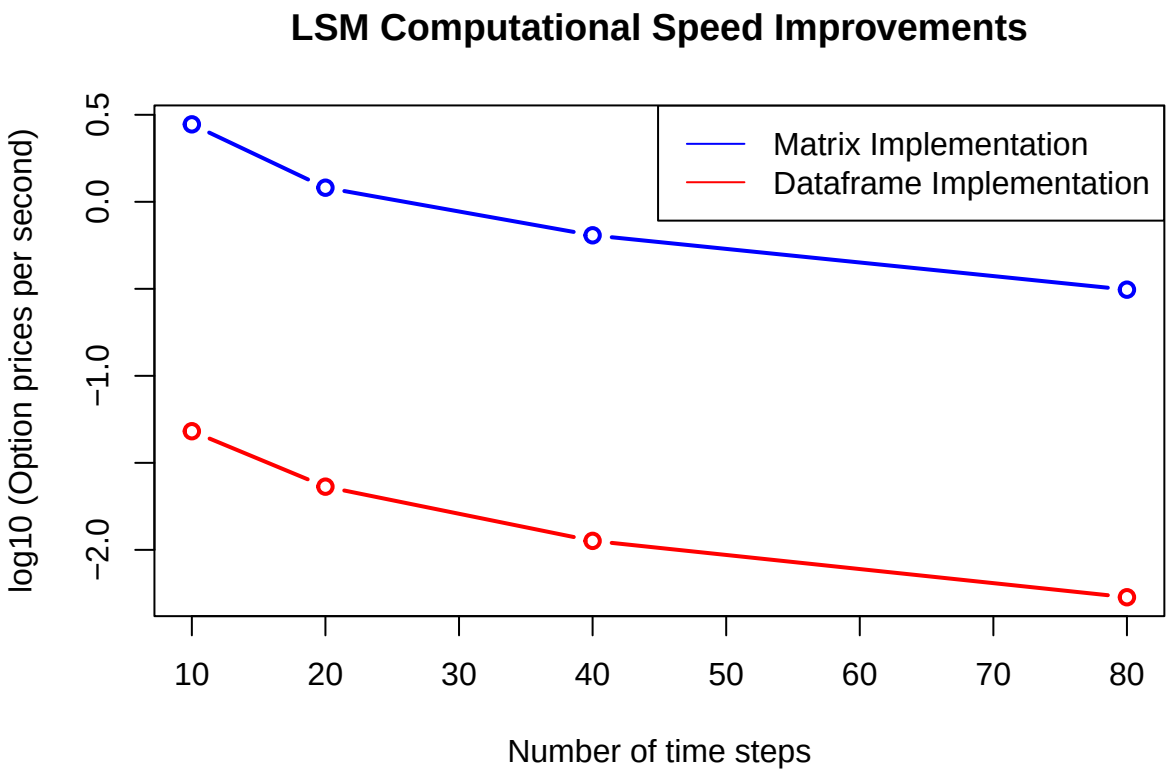
Table 3.1: Computational Speed Comparison (in units of prices/second) for n=256000 and k=3

	m=10	m=20	m=40	m=80
dataframe implementation	0.05	0.02	0.01	0.01
matrix implementation	2.79	1.21	0.64	0.31

Table 3.2: Runtime Comparison for n=256000 and k=3

	m=10	m=20	m=40	m=80
dataframe implementation	20.81	43.33	88.76	187.14
matrix implementation	0.36	0.83	1.56	3.20

Figure 3.1: Computational Speed Improvements



3.6 Experiment Implementation

We run our LSM simulation based on the following sets of n's, m's, and k's: $n : \{1000, 4000, 16000, 64000, 256000\}$, $m : \{10, 20, 40, 80\}$, $k : \{1, 2, 3\}$. For our stock parameters: $S_0 = K = 100$, $T = 1/12$, $r = 0.04$, $q = 0.02$, and $\sigma = 0.2$, corresponding to underlying stock price at $t = 0$, strike price, time to maturity, risk-free rate, and continuous dividend rate, respectively. Similar to results shown in the original paper, we also compute the early exercise value by taking the difference between the LSM American option and Black Scholes Merton closed-form (BSM) values. By definition any stock with dividends should lead to a positive early exercise value. As such we define our BSM function in Snippet 3.4. Our iterative implementation of LSM on different combinations of n's, m's, and k's is shown in Snippet 3.5.

Snippet 3.4: BSM Closed Form Function

```
# Closed form BSM solution for European put
european_put_BSM <- function(S,K,r,q,sigma,t) {

  d1 = (log(S / K) + (r - q + 0.5 * sigma^2) * t) / (sigma * sqrt(t))
  d2 = d1 - sigma * sqrt(t)

  put_price = K * exp(-r * t) * pnorm(-d2) - S0 * exp(-q * t) * pnorm(-d1)
  return(put_price)
}
```

Snippet 3.5: Experiment Implementation

```r
# main code to iterate LSM for list of m steps, n paths, and k regressors using Chebyshev Basis Functio
# output two dataframes: option value and se, early exercise value and % of early exercise paths.
# Early exercise value = American (LSM) value - European value
S0 = K = 100
t = 1/12
r = 0.04
q = 0.02
sigma = 0.2

european_put_value = european_put_BSM(S0,K,r,q,sigma,t)

m_list = c(10,20,40,80)
n_list = c(1000,1000*4, 1000*4**2, 1000*4**3, 1000*4**4)
k_regressors_list = c(1,2,3)

put_LSM_results = data.frame()
put_LSM_results2 = data.frame()

for (k_regressors in k_regressors_list) {
  temp_results = data.frame(matrix(ncol = 0, nrow = length(n_list)))
  temp_results2 = data.frame(matrix(ncol = 0, nrow = length(n_list)))

  for (m in m_list) {
    value_list = c()
    se_list = c()
    rt_list = c()

    early_portion_list = c()
    early_value_list = c()
    for (n in n_list) {
      start_time = Sys.time()
      option_LSM_res = LSM_put(S0,K,r,q,sigma,t,n,m, k_regressors, chebyshev_basis)
      end_time = Sys.time()
      runtime = as.numeric(end_time - start_time, units = "secs")
      value_list = c(value_list, round(option_LSM_res$value,4))
      se_list = c(se_list, round(option_LSM_res$se,4))
      rt_list = c(rt_list, round(runtime, 4))

      early_value_list = c(early_value_list, round(option_LSM_res$value - european_put_value,4))
      early_portion_list = c(early_portion_list, round(option_LSM_res$early_portion,3))

    }
    temp_results[[paste0("value_m",m)]] = value_list
    temp_results[[paste0("se_m",m)]] = se_list
    temp_results[[paste0("rt_m",m)]] = rt_list

    temp_results2[[paste0("EE_value_m",m)]] = early_value_list
    temp_results2[[paste0("Pct_EE_m",m)]] = early_portion_list
  }
  n_names = c()
  for (n in n_list) {n_names = c(n_names, paste0("k",k_regressors,",","n",n))}
  rownames(temp_results) = n_names
  rownames(temp_results2) = n_names
```

```
  put_LSM_results = rbind(put_LSM_results, temp_results)
  put_LSM_results2 = rbind(put_LSM_results2, temp_results2)
}

# write.csv(put_LSM_results, file = "put_LSM_cpf_res_20241127.csv")
# write.csv(put_LSM_results2, file = "put_LSM_cpf_res2_20241127.csv")
```

3.7 Result Analysis

Table 3.3 shows the LSM simulations results for every combination of n's, m's, and k's stated in section 3.6 and includes put option values (value), standard errors (se), and run times (rt) and Table 3.4 shows the results for early exercise values (EE_value) and percentage of options that were early exercised (Pct_EE). We observed that $\forall k$, the standard error of the put value decreases as we increase n. Since we choose n such that for every iteration n increases by 4 times, the standard error of decreases by 2 times, as standard error converges at a rate of order $\frac{1}{\sqrt{n}}$. Although not so obvious, as m increases, we observe that the standard error also decreases but by a very marginal amount. Run time for the simulations also increase as m, n, k increases. In particular, $\forall n \in [4000, 16000]$, we were able to compute put values that have standard errors approximately between 4 to 2 cents, which is certainly admissible for American Options. Figure 3.2 shows the convergence path of the put values as n increases for difference m parameters fixing $k = 3$. We can see that for $n < 64000$ the convergence path seems to be oscillating quite significantly. For $n > 64000$, we get a very smooth converging path. For $m = 40, 80$, the both path converges to a very similar value, which is in line with the fact that as m increases, we get a more accurate result. We will discuss about accuracy of our put value in 3A where we also compute the put option value via Binomial Black-Schoels with Richarson Exprapolation.

Moving on to Table 3.4, we see the effects of including higher order polynomials (increasing k). All else equal, both the early exercise value and percentage of early exercise tends to increase. Increasing the order of polynomial basis in LSM improves the accuracy of continuation value estimation, allowing for better identification of early exercise opportunities and capturing complex, nonlinear relationships between stock price and payoffs. This leads to higher early exercise value and percentage of early exercised options. This is also inline with the fact that the early exercise value should positive for dividend paying stocks. Increasing m also increases the percentage of early exercise which comes from the increasing granularity of capturing accurate continuation payoffs. Fixing m and k, we see that increasing n reduces the variability in early exercise value and percentage of early exercise. The values stabilize and tend toward convergence as n becomes large.

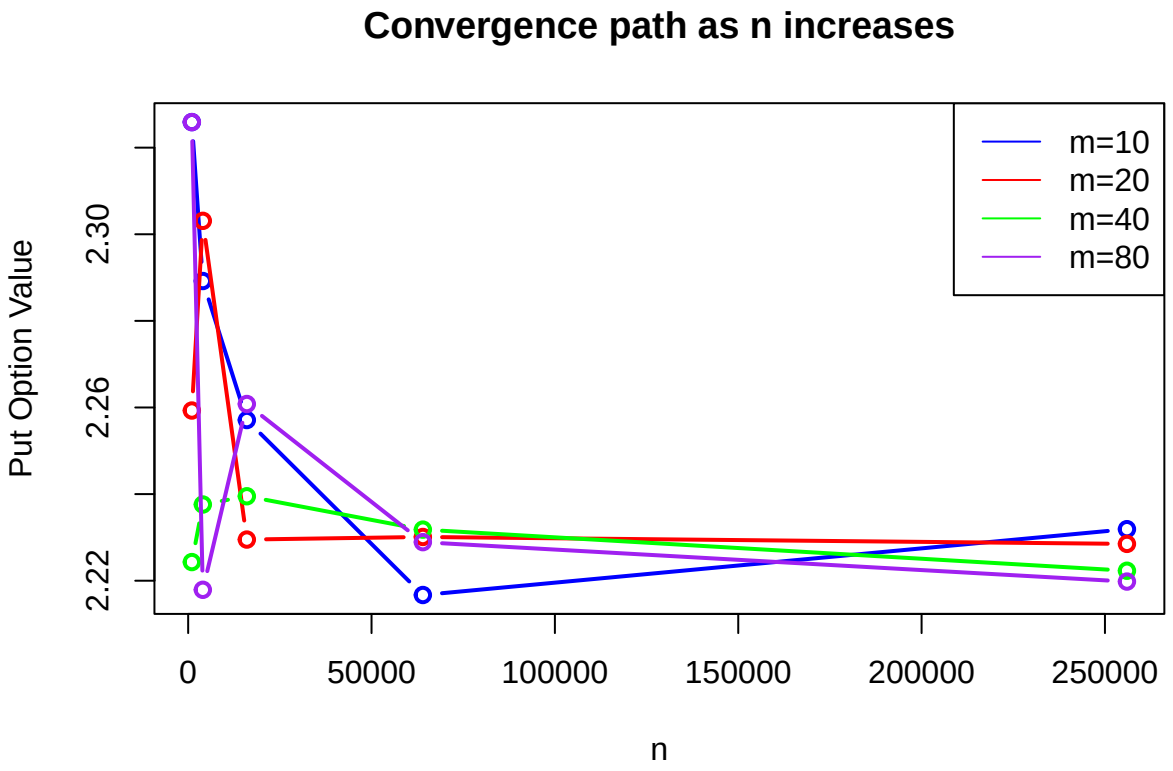
Table 3.3: Put Option Value, Standard Error, and Run time

	value_m10	se_m10	rt_m10	value_m20	se_m20	rt_m20	value_m40	se_m40	rt_m40	value_m80	se_m80	rt_m80
k1,n1000	2.2873	0.0914	0.0128	2.2247	0.0879	0.0024	2.1553	0.0842	0.0053	2.2102	0.0889	0.0103
k1,n4000	2.2846	0.0477	0.0046	2.2644	0.0451	0.0088	2.2332	0.0455	0.0234	2.2222	0.0437	0.0404
k1,n16000	2.2311	0.0233	0.0176	2.2169	0.0227	0.0909	2.2190	0.0228	0.0826	2.2327	0.0225	0.1683
k1,n64000	2.2149	0.0115	0.0839	2.2190	0.0114	0.1600	2.2125	0.0111	0.3388	2.2152	0.0111	0.7099
k1,n256000	2.2237	0.0058	0.3679	2.2205	0.0057	0.7610	2.2104	0.0056	1.5137	2.2083	0.0055	2.8193
k2,n1000	2.3399	0.0941	0.0013	2.2425	0.0858	0.0027	2.2603	0.0917	0.0054	2.3217	0.0989	0.0109
k2,n4000	2.2822	0.0474	0.0045	2.2750	0.0447	0.0100	2.2194	0.0422	0.0203	2.2445	0.0429	0.0405
k2,n16000	2.2566	0.0233	0.0194	2.2232	0.0220	0.0450	2.2291	0.0222	0.0812	2.2458	0.0216	0.1750
k2,n64000	2.2168	0.0111	0.0850	2.2249	0.0110	0.1678	2.2266	0.0109	0.3530	2.2253	0.0110	0.7242
k2,n256000	2.2291	0.0057	0.3400	2.2271	0.0055	0.7623	2.2190	0.0054	1.5714	2.2151	0.0053	3.0637
k3,n1000	2.3259	0.0927	0.0014	2.2593	0.0862	0.0030	2.2243	0.0868	0.0056	2.3259	0.0980	0.0115
k3,n4000	2.2892	0.0479	0.0049	2.3031	0.0457	0.0102	2.2376	0.0452	0.0211	2.2179	0.0403	0.0469
k3,n16000	2.2571	0.0231	0.0198	2.2295	0.0220	0.0410	2.2395	0.0225	0.0945	2.2608	0.0230	0.1856
k3,n64000	2.2167	0.0111	0.0902	2.2301	0.0113	0.1865	2.2318	0.0109	0.3511	2.2289	0.0112	0.7842
k3,n256000	2.2319	0.0057	0.3588	2.2285	0.0055	0.8297	2.2223	0.0055	1.5572	2.2198	0.0054	3.1976

Table 3.4: Early Exercise Value, Percentage of Early Exercise

	EE_value_m10	Pct_EE_m10	EE_value_m20	Pct_EE_m20	EE_value_m40	Pct_EE_m40	EE_value_m80	Pct_EE_m80
k1,n1000	0.0723	0.236	0.0096	0.235	-0.0598	0.197	-0.0048	0.515
k1,n4000	0.0695	0.142	0.0494	0.261	0.0181	0.191	0.0071	0.264
k1,n16000	0.0160	0.170	0.0018	0.176	0.0040	0.197	0.0177	0.206
k1,n64000	-0.0001	0.158	0.0040	0.169	-0.0026	0.199	0.0002	0.207
k1,n256000	0.0087	0.155	0.0054	0.182	-0.0046	0.201	-0.0068	0.209
k2,n1000	0.1249	0.276	0.0275	0.339	0.0453	0.255	0.1066	0.433
k2,n4000	0.0671	0.184	0.0600	0.334	0.0043	0.342	0.0295	0.338
k2,n16000	0.0415	0.264	0.0081	0.296	0.0140	0.281	0.0307	0.326
k2,n64000	0.0018	0.240	0.0099	0.264	0.0116	0.303	0.0102	0.297
k2,n256000	0.0140	0.240	0.0120	0.275	0.0040	0.301	0.0001	0.313
k3,n1000	0.1108	0.340	0.0443	0.359	0.0092	0.334	0.1108	0.500
k3,n4000	0.0741	0.178	0.0880	0.346	0.0226	0.371	0.0028	0.402
k3,n16000	0.0420	0.294	0.0145	0.333	0.0244	0.350	0.0458	0.367
k3,n64000	0.0017	0.279	0.0151	0.300	0.0168	0.354	0.0139	0.350
k3,n256000	0.0168	0.274	0.0135	0.319	0.0072	0.348	0.0047	0.359

Figure 3.2: Options Convergence path when k=3



3A. Put Option Pricing via Binomial Black-Scholes with Richardson Extrapolation

3A.1 Objective

In this section, we implement the Binomial Black-Scholes with Richardson Extrapolation (BBSR) to compute the put option value. This allows us to compare the accuracy of the put value between using BBSR and LSM. As suggested in the paper "Broadie and Detemple, 1996, American Option Valuation: New Bounds,

Approximations, and a Comparison of Existing Methods", we will also define our benchmark for the put option value using a standard Binomial Tree pricer with $m = 15000$ time steps. Through this section, we refer to this benchmark as the "True Value". We will also compare computational speeds and convergence behavior with respect to the true put option value.

3A.2 BBSR Implementation

The BBSR algorithm derived from the standard Bimomial Tree algorithm but with some modifications. The first modification is on the $m-1$th time step, where the option continuation value at every node is replaced with the Black-Scholes option value. The binomial tree combined with the first modification is called the Binomial Black-Scholes (BBS). The second modification is applying Richardson extrapolation on the computed option value. In our implementation, we use "two-point" Richardson extrapolation. Referencing to the paper, we use the "two-point" Richardson extrapolation as the authors suggest that higher order extrapolation does not add value in terms of accuracy. Hence as an example, if were to compute the BBSR price V for $m = 40$, the Richardson extrapolation formula used to apply on the BBS price is $V = 2 \cdot V1 - V0$, where $V1$ and $V0$ are the BBS prices on $m_1, m_0 = 40, 20$, respectively. The BBSR code snippet can be found in Snippet 3.6. In computing the option put value, we use the same stock parameters as defined in Section 3. We ran our algorithm using $m : \{10, 20, 40, 80, 160, 320\}$. We also record the run times for m cases. For interested readers, we include our implementation of the standard Binomial Tree algorithm with $m = 15000$ in Appendix 3.3.

Snippet 3.6: Experiment Implementation

```
# Binomial Black-Scholes Algorithm
BBS <- function(S0,r,q,sigma,m,t) {
  dt = t/m
  u = exp(sigma*sqrt(dt))
  d = 1/u
  p = (exp((r-q)*dt) - d)/ (u-d)
  df = exp(-r*dt)

  S = matrix(0, nrow=m+1, ncol=m+1)
  V = matrix(0, nrow=m+1, ncol=m+1)

  for (j in 1:(m+1)) {
    for (i in 1:j) {
      S[i,j] = S0 * u**(j-i) * d**(i-1)
    }
  }
  # discard option value at time step m+1, because we can calculate in from indexing at node m
  for (j in (m):1) {

    for (i in 1:j) {
      if (j == m) {
        v_continuation = european_put_BSM(S[i,j], K, r, q, sigma, dt)

        V[i, j] = max(v_continuation, max(K - S[i,j], 0))
      }

      else {
        v_continuation = df * (p* V[i,j+1] + (1-p) *V[i+1,j+1])
        if (j==1) {

          V[i,j] = v_continuation
        }
```

```
        else {

          V[i,j] = max(v_continuation, max(K - S[i,j], 0))
        }

      }
    }
  }
  return(V[1,1])
}

# Richardson Extrapolation Function
richardson_extrap <- function(S0,r,q,sigma,t,m) {

  step_ratio = 2
  V0 = BBS(S0, r, q, sigma, m/step_ratio, t)

  V1 = BBS(S0, r, q, sigma, m, t)
  V = 2*V1 - V0

  return(V)
}
S0 = K = 100
t = 1/12
r = 0.04
q = 0.02
sigma = 0.2
m_list = c(10,20,40,80)
V_bbsr = c()
V_bbsr_rt = c()
for (m in m_list) {
  start_time = Sys.time()
  V_bbsr = c(V_bbsr, c(richardson_extrap(S0,r,q,sigma,t,m)))
  end_time = Sys.time()
  run_time = as.numeric(end_time - start_time, units = "secs")
  V_bbsr_rt = c(V_bbsr_rt, round(run_time,4))
}
```

3A.3 Discussion on Accuracy, Path Convergence, and Computational Efficiency

Accuracy

In this section we compare accuracy, convergence behavior and computational speed of LSM($k = 3, n = 64000$), LSM($k = 3, n = 256000$), and BBSR methods with varying m steps where $m : \{10, 20, 40, 80\}$. We purposely chose LSM($k = 3, n = 256000$) to reflect the most precise and method among all LSM runs and include LSM($k = 3, n = 64000$) to sacrifice some precision for speed. This will provide us with a more meaningful comparison once we include the BBSR method into the picture. We choose the absolute error metric as our basis for evaluating accuracy.

LSM price accuracy evaluation using BSSR price as benchmark

We now evaluate our LSM model's accuracy using BSSR model as the benchmark. The absolute error values are shown in Table 3.5, where we measure the difference between BSSR prices and LSM prices by varying m steps. Overall, LSM prices are considered to be accurate. For $m \geq 20$, we achieve absolute error values of under 0.01, which is quite remarkable. From our results, we can see that we do not require as much as

256000 stock paths to get an accurate price, as the LSM model with 64000 stock paths are enough to achieve sufficient accuracy.

Table 3.5: LSM price accuracy with BSSR price as benchmark

	BBSR	LSM(n=64000)	Abs. Error	LSM(n=256000)	Abs. Error
m=10	2.292439	2.2167	0.0757395	2.2319	0.0605395
m=20	2.227690	2.2301	0.0024101	2.2285	0.0008101
m=40	2.226110	2.2318	0.0056905	2.2223	0.0038095
m=80	2.225939	2.2289	0.0029614	2.2198	0.0061386

Compare LSM and BSSR prices with the True Value as benchmark

Here, we include the true value (binomial(n=15000)) in our comparison and use the true value as the benchmark for the BSSR and LSM prices. The results are shown in Table 3.6. We can see that the BBSR method is very accurate and superior to the LSM method at every m. At $m = 80$, the error of the BSSR value is small and negligible in practice. This is not to say that the LSM algorithm did a poor job, because we can see that the absolute error is of order 10^{-3}, which is certainly less than a cent.

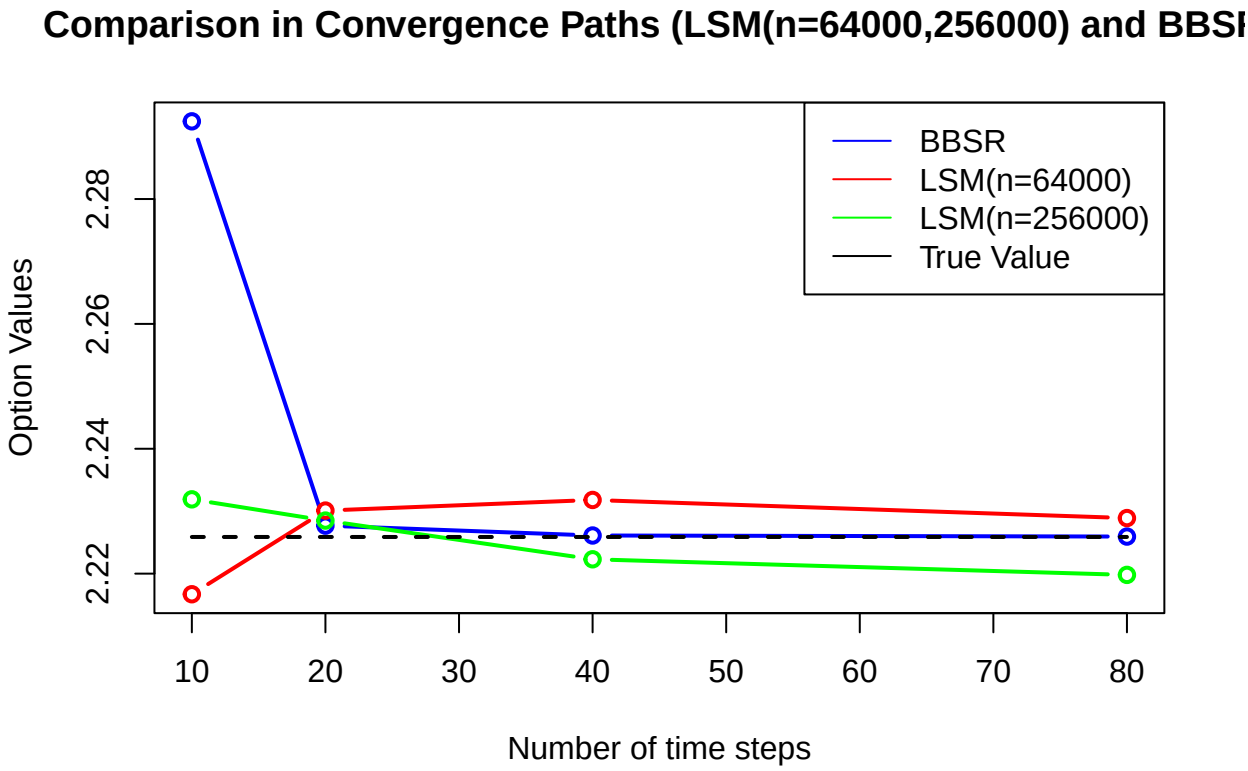
Table 3.6: Accuracy Comparison with True Value as benchmark

	True Value	LSM(n=64000)	Abs. Error	LSM(n=256000)	Abs. Error	BBSR	Abs. Error
m=10	2.225874	2.2167	0.009174	2.2319	0.006026	2.292439	0.0665655
m=20	2.225874	2.2301	0.004226	2.2285	0.002626	2.227690	0.0018159
m=40	2.225874	2.2318	0.005926	2.2223	0.003574	2.226110	0.0002355
m=80	2.225874	2.2289	0.003026	2.2198	0.006074	2.225939	0.0000646

Convergence Behavior

Here, we dicuss the convergence behaviors of the algorithms and Figure 3.3 shows the plot each algorithm's convergence path. In terms of converging behavior, the LSM algorithm started off with a relatively low discrepancy compared to the true value and converges more smoothly compared to the BSSR. As comparison, the BSSR algorithm started of with a relatively high discrepancy and descends with a rapid rate and converges to the true value beautifully at around $m = 40$. This implies that the LSM algorithm converges with a slower rate than the BSSR and might require more time steps to produce a more accurate result. The result shows that the BSSR is very powerful in that it is able to converge to the true price by using a very low computational setting since it is already accurate for a few time steps (i.e. $m = 40$).

Figure 3.3: Options Convergence Path with True Value as benchmark



Computational Speed

Table 3.7 shows the run times and computational speed. Computational speed is calculated as option prices per second by taking the inverse of the run time (sec). In particular, we observed that the computational efficiency results for BSSR method is not linear in that for smaller time steps, it takes longer time to run the algorithm. The increased runtime for smaller time steps (m) in the implementation is likely due to fixed overheads in initializing and updating matrices, which dominate when m is small and the computations are less intensive. Nevertheless, if we compare the speeds and run time, BSSR is fast and efficient. At $m = 80$, using the observed speed of 344 prices per second, we can say that it is certainly viable in practice. Compared to LSM ($n = 64000$), we can compute only 3 prices per second.

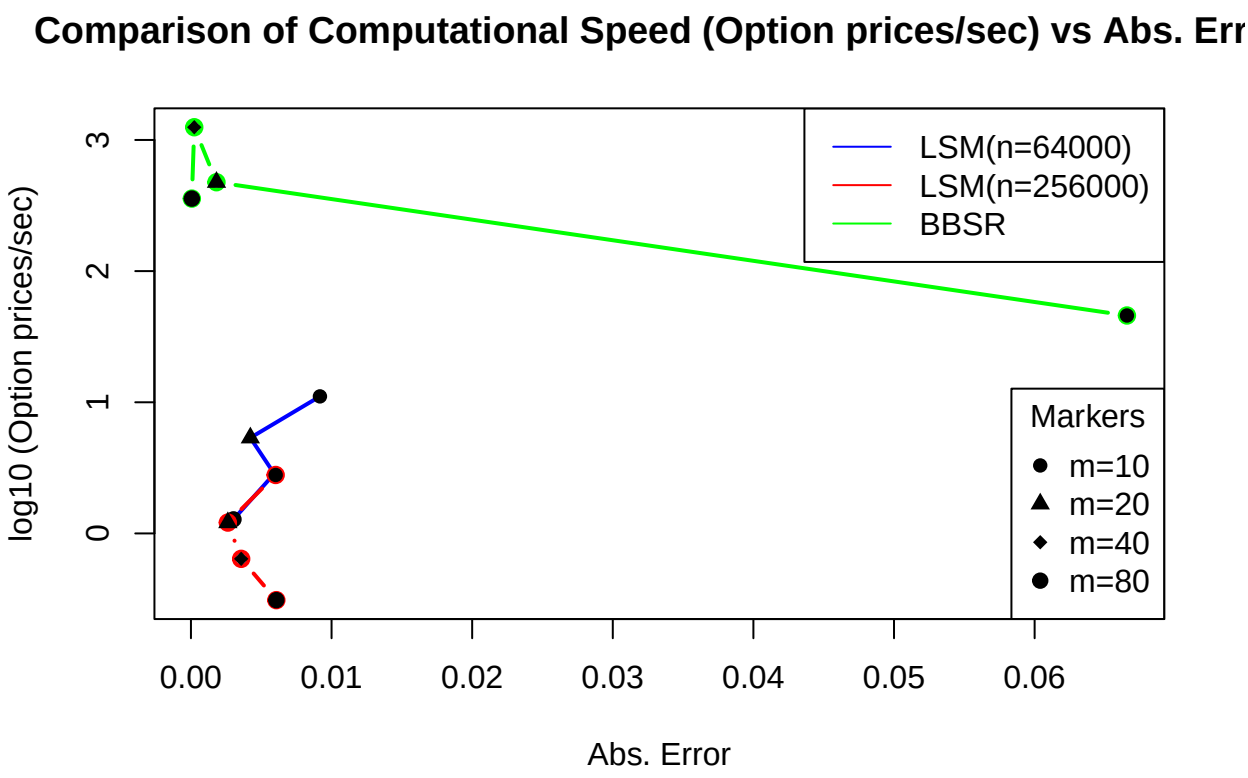
Table 3.7: Run Times (seconds/price) and Speed Comparison (prices/sec)

	LSM(n=64000) Run Time	LSM(n=64000) Speed	LSM(n=256000) Run Time	LSM(n=256000) Speed	BBSR Run Time	BBSR Speed
m=10	0.0902	11.09	0.3588	2.79	0.0218	45.87
m=20	0.1865	5.36	0.8297	1.21	0.0021	476.19
m=40	0.3511	2.85	1.5572	0.64	0.0008	1250.00
m=80	0.7842	1.28	3.1976	0.31	0.0028	357.14

3A.4 Summing it all up

We conclude that in our specific case, the BSSR is superior to LSM in that it requires very small computational effort to compute a very accurate option price. BBSR wins in both the race for accuracy and speed. Figure 3.4 illustrates the plot that examines the accuracy and speed of each algorithm with varying time step. In the plot, we scale the speed by taking the log_{10} for better representation for comparison. In this plot, the algorithm preference would be at the top left area, which points that lie in that area exhibits relatively high speeds and low errors. Although the plot shows that the accuracy between BSSR and LSM is very close, the computational speed is the winning factor and we conclude that BSSR certainly practical, accurate, and simple to implement.

Figure 3.4: Speed vs Abs. Error Plot Comparison



Appendix

3.1: Non-vectorized version of LSM Algorithm using DataFrames

```
simulate_S_paths0 <- function(S0,r,q,sigma,n,m,delta_t) {
  #Function for simulating discrete-time stock paths of m steps with n paths based on the
  # BSM risk neutral measure
  set.seed(3)
  S = rep(S0,n)
  S_df = data.frame(matrix(ncol = 0, nrow = n))
  for (i in 1:m) {
    # + antithetic paths
    z_temp = rnorm(n/2)
    z = c(z_temp, -z_temp)
    delta_S =  S * (r-q) * delta_t + sigma * S * sqrt(delta_t) * z
    S = S + delta_S
    S_df[[paste0("S",i)]] = S
  }
  return(S_df)
}

laguerre_basis0 <- function(k, x) {
  laguerre_pi <- function(i, x) {
    if (i == 0) return(exp(-x / 2))
    coeff <- sapply(0:i, function(j) ((-1)^j * choose(i, j) * x^j) / factorial(j))
    return(exp(-x / 2) * rowSums(coeff))
  }
  return(as.data.frame(sapply(0:(k-1), function(i) laguerre_pi(i, x))))
}

chebyshev_basis0 <- function(k,x) {
  X = data.frame(matrix(ncol = 0, nrow = length(x)))
  X[[paste0("CB",0)]] = rep(1, length(x))
  X[[paste0("CB",1)]] = x
  if (k > 1) {
    for (i in 2:k) {
      X[[paste0("CB",i)]] = 2 * x * X[[paste0("CB",i-1)]] - X[[paste0("CB",i-2)]]
    }
  }
  return(subset(X, select = -CB0))
}
```

```
LSM_put0 <- function(S0,K,r,q,sigma,t,n,m, k_regressors, basis_func) {
  # Function to run the Least-Squares Monte Carlo
  delta_t = t/m # time steps
  S_df = simulate_S_paths(S0,r,q,sigma,n,m,delta_t) # simulate stock paths
  exercise_times = rep(m,n) # intialize stopping times at expiration
  # create payoff dataframe for all discrete times
  payoff_df = data.frame(apply(K - S_df, c(1, 2), function(x) max(x, 0)))

  # scaling underlying stock prices to prevent numerical issues with polynomials
  S_scaled_df = S_df
  # Recursively loop backwards to apply LSM
```

```
  for (i in (m-1):1) {
    itm_idx = payoff_df[,i] > 0 # find ITM idx
    # get future payoffs according to current stopping times
    future_cashflows = mapply(function(row, col) payoff_df[row, col], row = 1:nrow(payoff_df), col = ex
    # get times to discount according to current stopping times
    discount_times = delta_t * (exercise_times - i)
    # define target as present value of future payoffs
    Y = future_cashflows * exp(-r*discount_times[itm_idx])
    # filter only ITM underlying stock prices
    S_itm = S_scaled_df[,i][itm_idx]
    # Create Laguerre polynomial regressors matrix
    X = basis_func(k_regressors, mm_scaler(S_itm))
    # Run OLS and calculate conditional expectation of Y|X
    model = lm(Y ~ ., data=X)
    cond_exp_Y = predict(model, newdata = data.frame(X))
    names(cond_exp_Y) = NULL
    # If current payoff exceeds E[Y|X], then exercise now, if not in the future
    # To implement this logic, we update our stopping times
    current_itm_payoff = payoff_df[,i][itm_idx]
    exercise_times[itm_idx] = ifelse(current_itm_payoff > cond_exp_Y, i, exercise_times[itm_idx])
  }
  # get future payoffs according to final stopping times, and discount them
  payoff_decisions = mapply(function(row, col) payoff_df[row, col], row = 1:nrow(payoff_df), col = exer
  discount_times = delta_t * (exercise_times - i)
  option_path_values = payoff_decisions * exp(-r*discount_times)
  # option value is the mean of all option present values from each path
  option_value = mean(option_path_values)
  se = sd(option_path_values) / sqrt(n)
  # % of paths that are early exercised
  early_exercise_portion = mean(exercise_times < m)
  return(list(value = option_value, se=se, early_portion=early_exercise_portion, ee_times=exercise_time
}
```

3.2: LSM Results using Laguerre Basis Functions

	value_m10	se_m10	rt_m10	value_m20	se_m20	rt_m20	value_m40	se_m40	rt_m40	value_m80	se_m80	rt_m80
k1,n1000	2.3032	0.0898	0.0098	2.1934	0.0810	0.0032	2.2247	0.0859	0.0054	2.1483	0.0717	0.0113
k1,n4000	2.2466	0.0438	0.0130	2.2793	0.0443	0.0111	2.2236	0.0410	0.0191	2.2447	0.0418	0.0399
k1,n16000	2.2483	0.0227	0.0235	2.2251	0.0214	0.0372	2.2137	0.0210	0.0865	2.2305	0.0202	0.1756
k1,n64000	2.2153	0.0108	0.1276	2.2201	0.0106	0.1755	2.2241	0.0105	0.3117	2.2041	0.0101	0.6608
k1,n256000	2.2301	0.0055	0.3479	2.2246	0.0053	0.7727	2.2151	0.0052	1.3900	2.2071	0.0051	2.8284
k2,n1000	2.3490	0.0943	0.0017	2.2395	0.0853	0.0032	2.2467	0.0903	0.0075	2.2967	0.0965	0.0171
k2,n4000	2.2861	0.0475	0.0054	2.2737	0.0449	0.0111	2.2332	0.0424	0.0234	2.2369	0.0424	0.0511
k2,n16000	2.2506	0.0230	0.0217	2.2254	0.0221	0.0466	2.2257	0.0221	0.0918	2.2491	0.0216	0.2209
k2,n64000	2.2194	0.0112	0.0881	2.2252	0.0110	0.1796	2.2275	0.0109	0.3977	2.2196	0.0108	0.7909
k2,n256000	2.2276	0.0056	0.3798	2.2269	0.0055	0.8221	2.2196	0.0054	1.5772	2.2155	0.0053	3.3583
k3,n1000	2.3351	0.0934	0.0020	2.2448	0.0858	0.0041	2.2155	0.0859	0.0080	2.3165	0.0994	0.0363
k3,n4000	2.2879	0.0480	0.0065	2.2800	0.0441	0.0126	2.2578	0.0451	0.0253	2.2592	0.0431	0.0522
k3,n16000	2.2540	0.0229	0.0249	2.2264	0.0219	0.0535	2.2362	0.0225	0.1077	2.2547	0.0225	0.2351
k3,n64000	2.2158	0.0111	0.0995	2.2264	0.0111	0.2162	2.2295	0.0109	0.4092	2.2252	0.0111	0.8643
k3,n256000	2.2316	0.0056	0.4085	2.2271	0.0056	0.8942	2.2206	0.0054	1.8166	2.2188	0.0054	3.7954

	EE_value_m10	Pct_EE_m10	EE_value_m20	Pct_EE_m20	EE_value_m40	Pct_EE_m40	EE_value_m80	Pct_EE_m80
k1,n1000	0.0882	0.317	-0.0217	0.377	0.0097	0.393	-0.0668	0.459
k1,n4000	0.0316	0.296	0.0643	0.365	0.0085	0.396	0.0297	0.399
k1,n16000	0.0333	0.304	0.0101	0.354	-0.0013	0.379	0.0154	0.410
k1,n64000	0.0002	0.306	0.0050	0.344	0.0090	0.378	-0.0110	0.401
k1,n256000	0.0151	0.299	0.0096	0.346	0.0001	0.380	-0.0079	0.397
k2,n1000	0.1339	0.274	0.0244	0.341	0.0316	0.255	0.0816	0.505
k2,n4000	0.0710	0.182	0.0586	0.338	0.0182	0.371	0.0218	0.374
k2,n16000	0.0355	0.266	0.0104	0.296	0.0106	0.313	0.0341	0.375
k2,n64000	0.0043	0.234	0.0101	0.263	0.0125	0.319	0.0045	0.349
k2,n256000	0.0126	0.237	0.0119	0.277	0.0045	0.319	0.0005	0.348
k3,n1000	0.1200	0.329	0.0297	0.357	0.0004	0.335	0.1014	0.493
k3,n4000	0.0729	0.176	0.0650	0.346	0.0428	0.372	0.0442	0.380
k3,n16000	0.0390	0.286	0.0113	0.319	0.0212	0.336	0.0397	0.365
k3,n64000	0.0007	0.272	0.0113	0.295	0.0145	0.347	0.0101	0.341
k3,n256000	0.0166	0.270	0.0120	0.307	0.0056	0.340	0.0037	0.350

3.3: Code Snippet of Binomial Model with n=15000 time steps

```
BinomTree <- function(S0,r,q,sigma,m,t) {
  dt = t/m
  u = exp(sigma*sqrt(dt))
  d = 1/u

  p = (exp((r-q)*dt) - d)/ (u-d)

  df = exp(-r*dt)


  S = matrix(0, nrow=m+1, ncol=m+1)
  V = matrix(0, nrow=m+1, ncol=m+1)

  for (j in 1:(m+1)) {

    for (i in 1:j) {
      S[i,j] = S0 * u**(j-i) * d**(i-1)

    }
  }
  # discard option value at time step m+1, only
  for (j in (m):1) {

    for (i in 1:j) {
      if (j == m) {
        v_continuation = df * (p*  max(K - S[i,j+1], 0) + (1-p) * max(K - S[i,j+1], 0))

        V[i, j] = max(v_continuation, max(K - S[i,j], 0))

      }

      else {

        if (j==1) {

          V[i,j] = v_continuation
        }
        else {

          V[i,j] = max(v_continuation, max(K - S[i,j], 0))
        }
      }

    }

  }
  return(V[1,1])
}
m=15000
V_binom = BinomTree(S0,r,q,sigma,m,t)
```