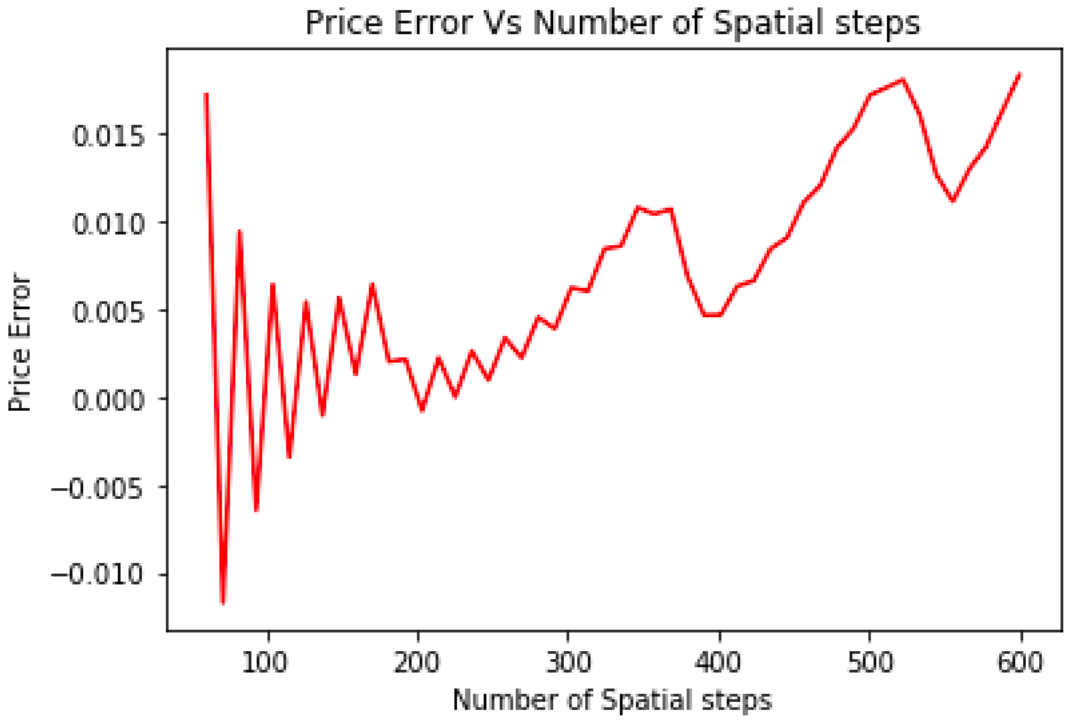
Option Pricing Analysis Report
Name: Rajat Dua

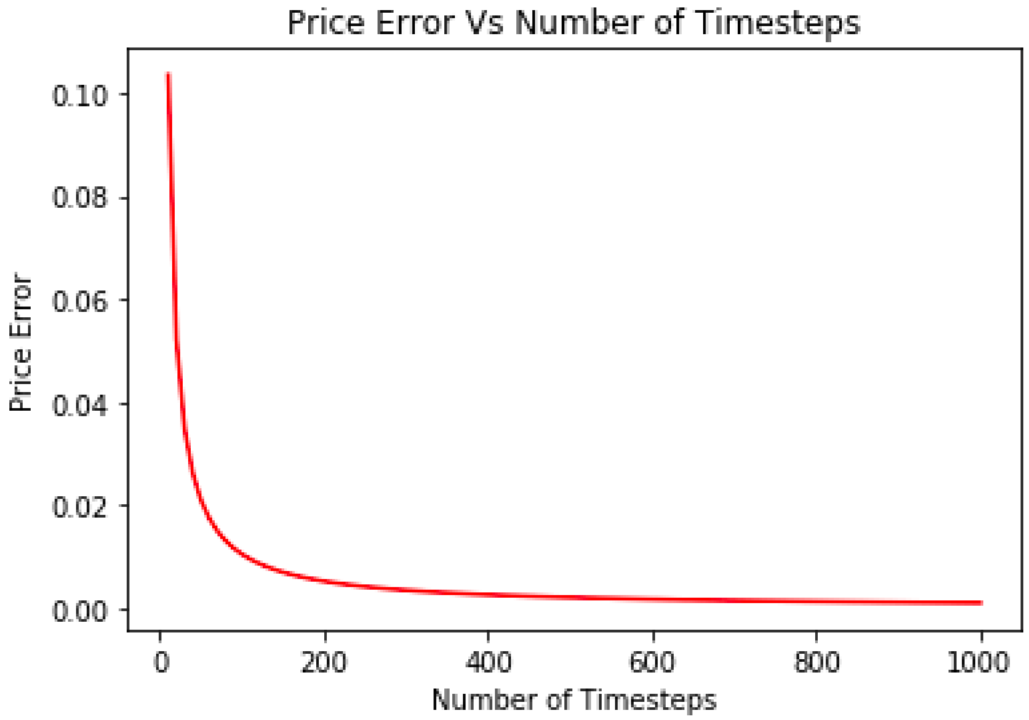
In this folder, I have incorporated some of my projects involving pricing of financial derivative products such as option pricing. I went through the implementation of Option Pricing Algorithms using Binomial Tree lattice methods, Finite Difference, Monte Carlo simulation, and Black Scholes option pricing formula. In addition, I have included an algorithm which will find roots of the given function using either the Bisect Method or the Newton's Method. Some of the projects which I have included in the folder are as follows:

1. FiniteDifferenceAmerican.py: In this problem, I have defined a function fdAmerican, to price American option value using the Gauss-Seidel method to solve a Crank-Nicolson formulation of the finite difference scheme, with input parameters: callput to determine whether the option is call or put, spot price (S0), strike price (K), yield to maturity (r), Time to Maturity (T) (in years), volatility (sigma), dividend yield rate (q), M+1 is the number of spatial steps in the uniform grid N+1 is the number of time steps in the uniform grid, Smax is the value where the upper spatial boundary condition is applied, and tol is the tolerance which is the stopping criteria for the Gauss-Seidel iteration method.
The ratio of dt (=T/N) and dS^2 (=(Smax/M)^2) should be less than 1 in order to ensure that the finite difference scheme is stable or else the price output of the function will be too high (positive) or too low (negative). The output of this function is a current American option value corresponding to the current stock price, and the difference between option value calculated from black-scholes formula and Crank-Nicolson formulation of the finite difference scheme as price error. I have compared the final value with the output generated from bsformula for the option value and found out that the final value from fdAmerican is very close to the output generated by the bsformula. I created a plot comparing the variation of price Error between bsformula and fdAmerican with Number of spatial steps(M), as following:



---

2. BinomialAmerican.py: In this problem, I have defined a function binomialAmerican to find the value of American options with input values spot price (S0), strike price (Ks), yield to maturity (r), Time to Maturity (T) (in years), volatility (sigma), dividend yield rate (q), callputs to determine whether the option is Call or put, and the number of timesteps in binomial tree (M). I have used these input variables to extract a tuple of 2 values, that is, payoff of the option at timestep t = 0 (option value now) and also the difference between option value calculated from black-scholes formula and binomial tree as price error. I have compared the final value with the output generated bsformula for the option value and found out that the final value from binomialAmerican is very close to the output generated by the bsformula. I created a plot comparing the variation of price Error between bsformula and binomialAmerican with Number of timesteps(M), as following:

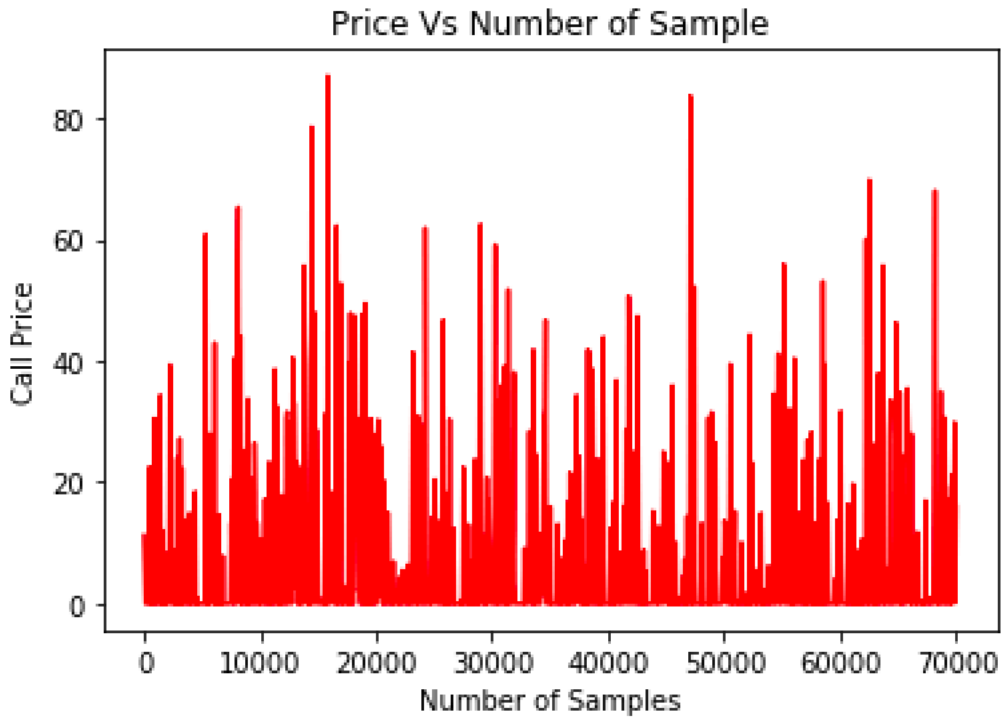


This chart clearly shows that the price error decreases drastically with increase in the number of timesteps.

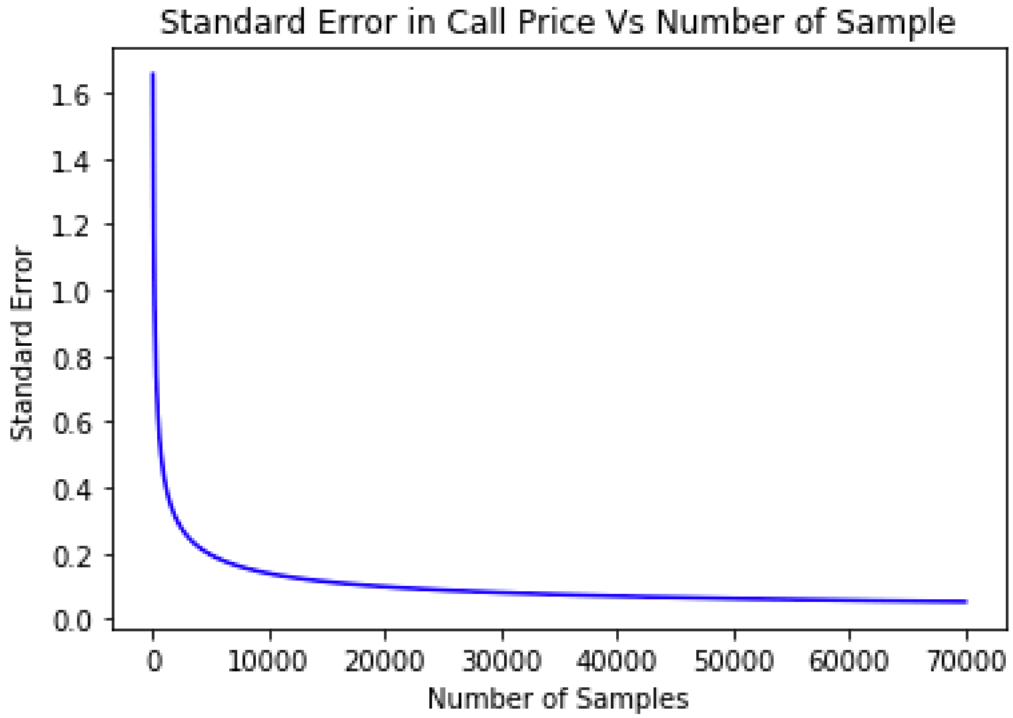
---

3. MCOptionPrices.py: In this problem , I have defined a function MCOptionPrices, with following inputs: S0, K, T, rateCurve, sigma, t, checkpoints, samples, and integrator. The output of the function is a dictionary with 4 key value pairs, that is, an array of running means at each checkpoint with a key 'Means'; an array of running standard deviations at each checkpoint with key 'StdDevs'; an array of running standard error at each checkpoint with key 'StdErrs'; and a floating point variable that represents the final value or the mean at the last checkpoint, represented by key 'TVs'. I have plotted graphs for each of the methods to compare the convergence properties of standard error with increase in number samples or sample paths(M).

Output using standard method of sample generation:

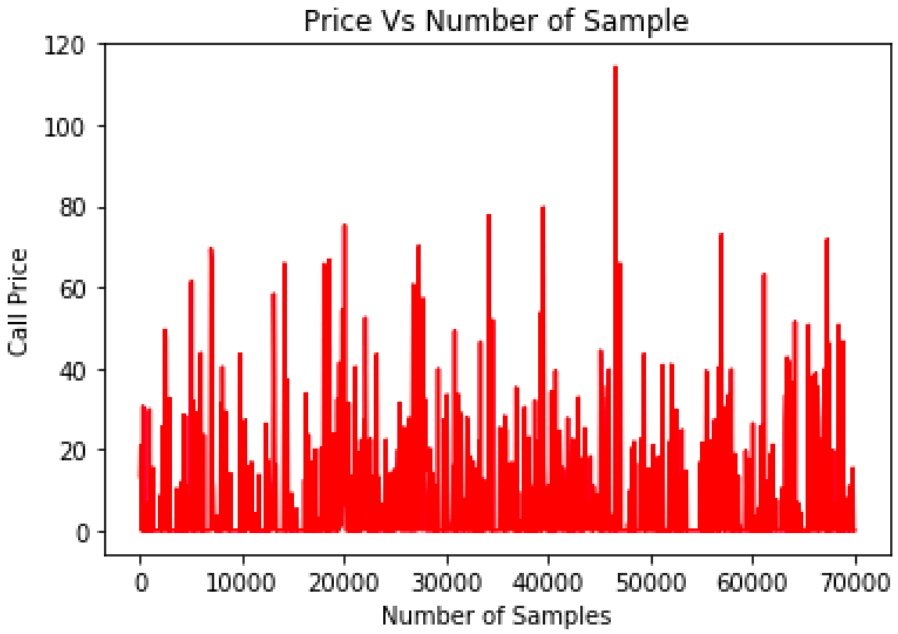


In this graph I have plotted the value of call price generated using standard method of sample generation for Monte Carlo Simulation against the number of sample paths.

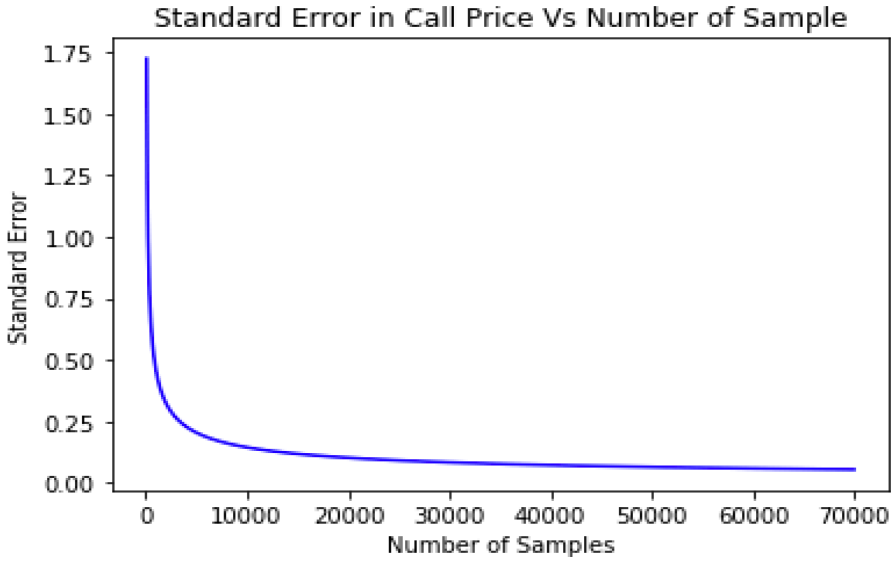


In this graph, I have plotted standard error of call price against number of samples using standard method of sample generation. It can be shown that the standard error decreases drastically with increase in the number of sample size, confirming the 1/sqrt(M) theoretical relationship between the standard error and the number of samples.

Output using 'Euler' method of sample generation:



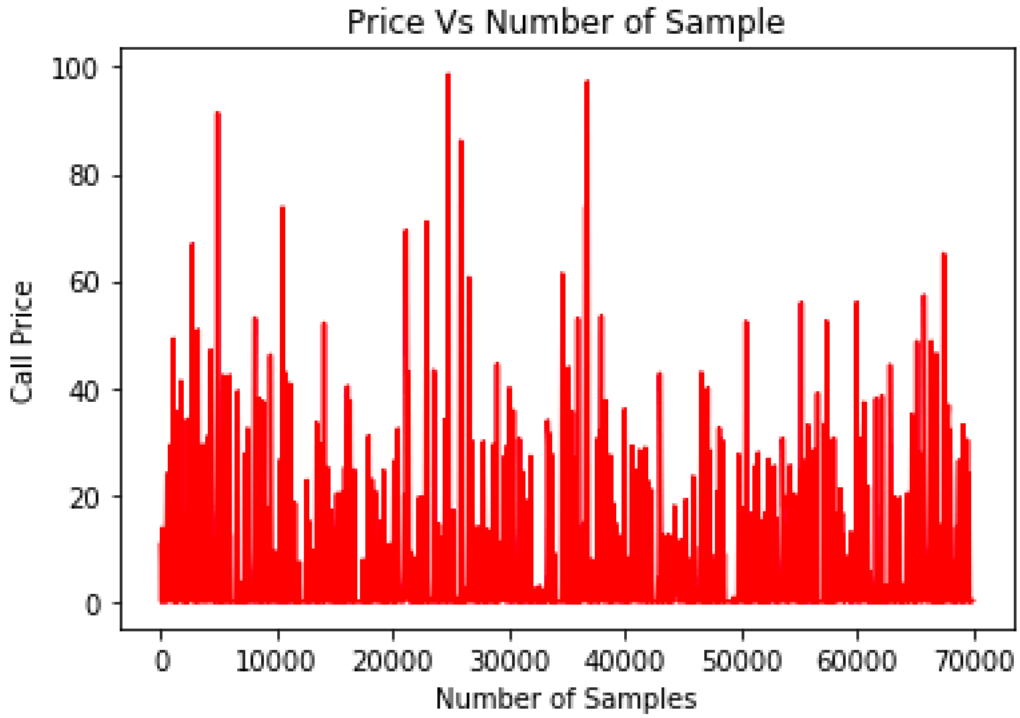
In this graph I have plotted the value of call price generated using 'Euler' method of sample generation for Monte Carlo Simulation against the number of sample paths.



In this graph, I have plotted standard error of call price against number of samples using 'Euler' method of sample generation. It can be shown that the standard error decreases drastically with

increase in the number of sample size, confirming the 1/sqrt(M) theoretical relationship between the standard error and the number of samples.

Output using 'Milstein' method of sample generation:



In this graph I have plotted the value of call price generated using 'Milstein' method of sample generation for Monte Carlo Simulation against the number of sample paths.

---

4. BSImpVol.py: In this problem, I have defined a function bsimpvol which returns the implied volatility using Black Scholes Formula, bsformula. It uses iterative methods of newton or bisection to compute the Implied Volatility. I have used partial function to pass the input variables in the bsformula to find option value and vega as a function of sigma, that is, volatility. For both the methods, I have used the difference between optionValue and price as the function variable. For the newton method, I have used vega as the derivative function. I validated the output using the online implied volatility calculator. I have tested the code by incorporating all the necessary conditions that were mentioned in the problem.
After comparing the convergence properties of Newton's method against the bisection method, I conclude that Newton Method does a far better job in computing the implied volatility as compared to the bisection method since the number of iterations required for the newton's method is far less than the bisection method. For out of the money (OTM) European option, the performance difference between newton and bisect method is more noticeable.

---

5. Bisect.py: In this problem, I have defined 2 functions newton and bisect which are iterative methods to find the roots of the given function. For newton method, I have provided a security

wherein if the start value is zero, then it will ask the user to enter the value of lower bound and upper bound. The iteration process will then start from the mid-point of upper and lower bound. In addition, if the average causes the derivative function to come out to be zero, I have incorporated another security wherein it will add some small random uniform variable between 0 and 0.01 to the derivative function in order to begin the iteration process. I have defined a list, xval, which captures all the values of iterative roots until the function root is captured, or the value of x comes within acceptable tolerance limit, defined by the tolerance input variable. Fdiff captures list of difference between each function output of each newton iteration and target value (generally, zero) until the function comes with in given tolerance input level.
For bisect method, I have provided security that if start value is None, then the function will start iteration from the mid-point of upper and lower bound.

---

6. BS.py: In this problem, I defined a function bsformula with input values callput [-1, 1], Spot Price (S0), Strike Price (K), interest rate (r), Time to maturity (T) (in years), volatility (sigma), and continuous dividend rate or foreign interest rate (q). I have used these inputs to compute the values of the Option Price, its delta and its vega. The function bsformula returns a 3-tuple optionValue, delta and vega.
