12-2016

Options Pricing Through Computational Methods

Robert Petty

OPTIONS PRICING THROUGH COMPUTATIONAL METHODS

by

Robert Petty

A paper submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

FINANCIAL ECONOMICS

Approved:

Dr. Tyler Brough Major Professor	Dr. Jason Smith Committee Member

Dr. Jared DeLisle
Committee Member

Utah State University
Logan, Utah

2016

Abstract

The purpose of this paper is to show the practical application of computational methods to price options. Emphasis is especially given to the use of the Longstaff-Schwartz method for pricing American and exotic options. An implementation of these pricing methods in a computer program are demonstrated. The advantages of using object-oriented programming and design patterns to make pricing programs more flexible and useful is also discussed.

Introduction

In academics, we tend to focus our work on European options because we have closed form solutions, such as the formula developed by Black and Scholes, which make calculating prices easy. In reality, unfortunately, the vast majority of options are American, which have the ability to be exercised before expiration. This makes it important to have tools that we can use for real world applications so that we can move beyond the theoretical environment or enhance our capabilities in the theoretical environment. In doing this we want to strive for algorithms that first accurate and second fast. The speed helps it to be more useful for real world applications especially for those like high speed traders that need fast computation to be profitable. The first step in this, is the binomial tree method to calculate options prices. This method, although relatively simple, still requires large amounts of repeated calculations. So even the simplest method for solving American options requires computational methods. The binomial tree method is only capable of pricing simple vanilla options though. To price more complicated American options or exotic ones like Asian options or look-back options, whose calculations depend on the specific price path the asset follows, we need a more powerful algorithm like that created by Longstaff and Schwartz. I have created a computer program to demonstrate the practical use of computational methods in pricing these types of options.

Algorithms

Here I will explain the algorithms used in my program to price options. I will begin with the simpler binomial method and move onto the more complex Longstaff-Schwartz method.

Binomial Method

The binomial pricing method is a simple and computationally fast method for pricing vanilla options. With this method, you create a tree of possible price outcomes of the underlying asset (Cox). To do this you start with the current spot price of the underlying asset. From this, you create two branches, one with the expected price in the next period in an upstate and the other for the expected price in a downstate. These states are calculated by multiplying the current state by u or d where u is the up-state factor and d is the down-state factor (McDonald):

$$u = e^{[(rate-dividend)*dt+volatility*\sqrt{dt}]}$$

$$d = e^{[(rate-dividend)*dt-volatility*\sqrt{dt}]}$$

From each of these branches you will create two more branches by the same process. This process is repeated until the options expiry is reached, creating a price tree (as can be seen in diagram examples below).

To price a European option, a payoff function needs to be applied to each leaf node. For example, with most simple options the payoff function would be:

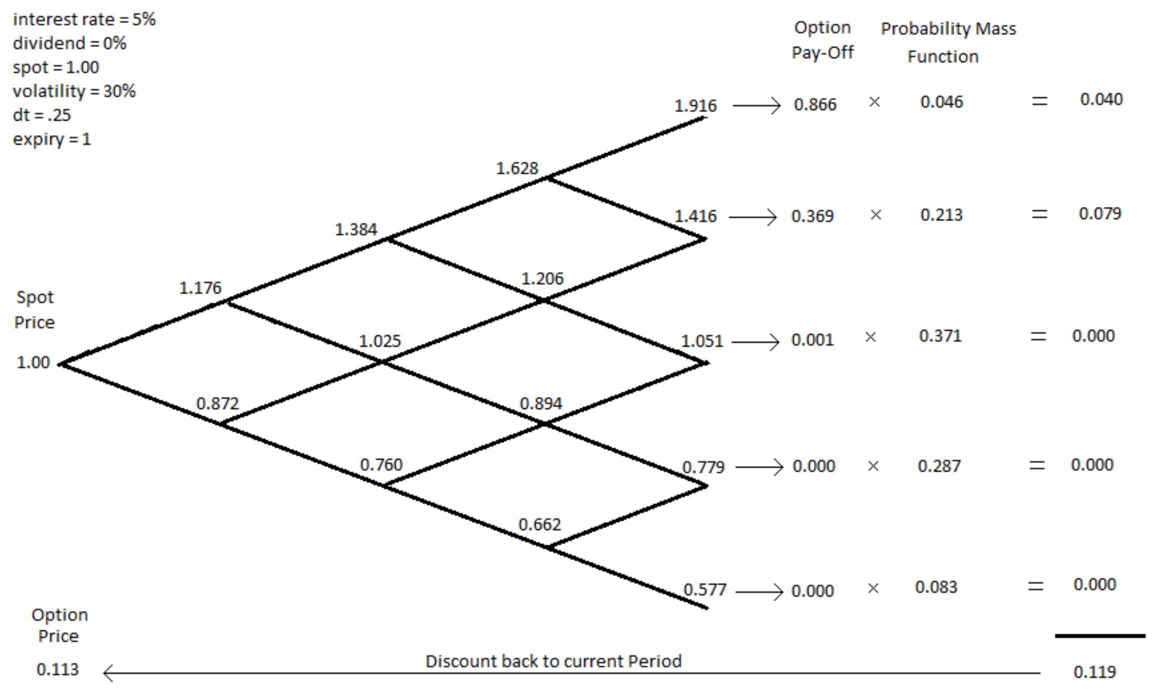
$$Payoff_{put} = \max(Strike - Price_f\ , 0)$$

$$Payoff_{Call} = \max(Price_f - Strike\ , 0)$$

but other payoffs could be used for different types of options. These payoffs must then weighted by the probability of that branch occurring. We can calculate these weights this by determining the probability of an up-state(p_u) using the equations below combined with the probability mass function of a binomial distribution (McDonald).

$$p_u = \frac{e^{(rate-dividend\ yield)*dt} - d}{u - d} \qquad p_d = 1 - p_u$$

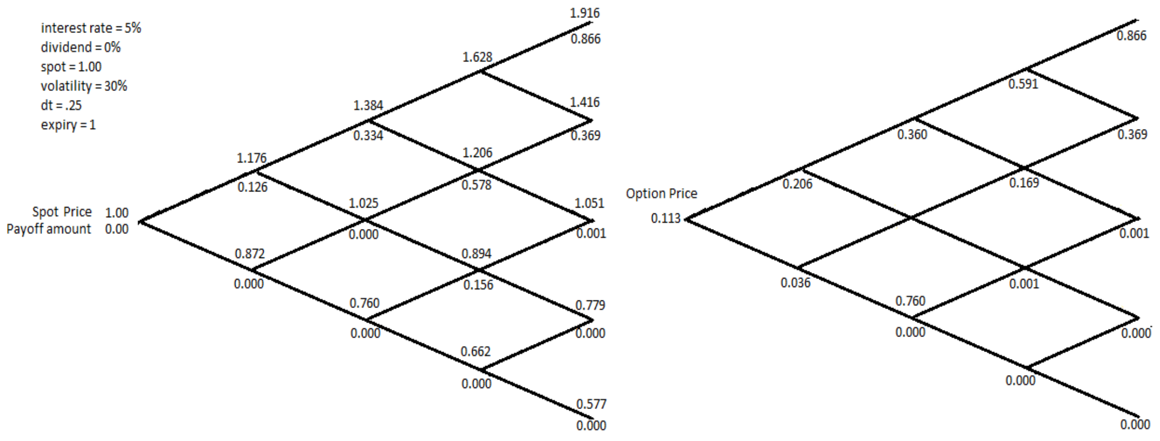
Then the weighted payoffs are summed and discounted back to the starting period for the final option price. A graphical demostraction of this process can be seen below.



American options are a bit more complicated. For American options, when the price tree has been created, payoffs need to be calculated at all nodes, and an option price determined for each node starting from the end branches then working back the present . The option price at any parent node can be found by the following equation:

$$parent\ node = \max(Payoff_t, [(b_u\ * p_u) + (b_u * p_u)] * e^{-rate})$$

This equation show us that we take the greater of either the payoff of exercising at that point or the weighted sum of its branches discounted back one period. This is repeated until you return to the final node at the present time. This final node is the price of your options contract.



Attempting to build a tree with more than three periods by hand becomes a tedious and cluttered process, helping us realize the importance of computerizing the algorithm. Even with thousands of steps this process can be done almost instantly with use of a couple of “for” loops by a computer. This can be seen in the pseudo code below that was used in making the program:

```
int steps = 4;
double expiry = 1;
double spot = 1.0;
double rate = .05;
double volatility = .3;
double dividend = .0;
double strike = 1.05;
int nodes = steps + 1;

double dt = expiry / steps;
double u = exp(((rate - dividend) * dt) + volatility * sqrt(dt));
double d = exp(((rate - dividend) * dt) - volatility * sqrt(dt));
double pu = (exp((rate - dividend) * dt) - d) / (u -d);
double disc = exp(-rate * expiry);
double spotT = 0.0;
double payoffT =0.0;
BinomialDistribution pmf = new BinomialDistribution(steps,pu);

for(int i = 0; i < nodes; i++) {
        spotT = spot * u^(steps-i)*d^i;
        payoffT += max(strike-spotT,0) * pmf.probability(steps - i);
}
double price = disc * payoffT;
```

You will notice that in the code the process is simplified by skipping the tree building steps and skipping to solving for payoffs at the end of each brand and summing them together as it goes. This simplification is what makes this process so fast. The American Binomial method uses a similar code:

```
int steps = 4;
double expiry = 1;
double spot = 1.0;
double rate = .05;
double volatility = .3;
double dividend = .0;
double strike = 1.05;
int nodes = steps + 1;

int nodes = steps + 1;
double dt = expiry / steps;
double u = exp(((rate - dividend) * dt) + volatility * sqrt(dt));
double d = exp(((rate - dividend) * dt) - volatility * sqrt(dt));
double pu = (exp((rate - dividend) * dt) - d) / (u -d);
double pd = 1 - pu;
double disc = exp(-rate * dt);
double dpu = disc * pu;
double dpd = disc * pd;

double[] ct = new double[nodes];
double[] st = new double[nodes];

for (int i=0; i < nodes; i++) {
        st[i] = spot * u^(steps-i)*d^i;
        ct[i] = max(strike- st[i],0);
}

for (int i=steps-1; i >= 0; i--) {
        for (int j=0; j <= i; j++ ) {
                ct[j]  = dpu * ct[j] + dpd * ct[j+1];
                st[j] = st[j] / u;
                ct[j] = max(ct[j], max(strike-st[j],0));
        }
}

return ct[0];
```

You can see that another for loop is needed so for the American option to compare the immediate payoff to the discounted option price of its weighted branches. Although this is slower that the European option code it still solves in nearly an instant.

When using binomial trees, errors may occur if only a few periods are used. This is because a binomial tree will only produce one more leaf node (or potential outcome) than the number of periods used, when infinite outcomes are actually possible. By increasing the number of periods, we create more possible outcomes that are distributed more continuously. For situations where options can only be exercised at certain intervals, this method is not practical.

Another downfall to the binomial method is that does not handle path dependent options well. Exotic options like Asian, barrier, and lookback options may need the asset's entire price path to calculate the proper payoff. The binomial method focuses only on individual points in time. For these type of options that look at the whole path, for a price certain types of Monte Carlo pricing methods are preferred.

<u>Monte Carlo Method</u>

Monte Carlo simulation is a process of generating many randomized outcomes and taking an average to find an answer. For options pricing, this means we are generating random paths that the underlying asset's price may follow. The in my program I make use of Geometric Brownian Motion to arrive at a price. Geometric Brownian Motion creates a path where the direction of each step is random but normally distributed. I use this to create a possible path that the stock price might follow. This is done by applying the equation (hull)

$$S_{t+1} = S_t * e^{[(rate-dividend-0.5*volatility^2)*dt+epsilon*volatility*\sqrt{dt}]}$$

to the current spot price of the option's underlying asset where epsilon is a random normal sample generated by the computer.

This gives us a hypothetical price for the following period. We then apply the same equation again to the price in the next period until we reach the option's expiry. This generated path looks similar to what is normally seen on a typical stock chart. A single path does not provide much information, so this process needs to be repeated many times. When this is done the payoff function is taken on the ending price of each period. The average of these payoffs are discounted back to the present time to arrive at a final options price for European options. As can be seen from the pseudo code below, generating these random paths is quite simple.

```
int steps = 4;
int paths = 10;
double expiry = 1;
double spot = 1.0;
double rate = .05;
double volatility = .3;
double dividend = .0;
double strike = 1.05;
double dt = expiry/steps;
double drift = (rate - dividend - 0.5 * volatility * volatility) * dt;
double diffusion = volatility * sqrt(dt);

double sum_CT = 0;
double[][] spot_T = new double[paths][steps+1];

for (int i=0; i<paths; i++) {
        spot_T[i][0]= spot;
        for (int j=1; j<=steps; j++) {
                double epsilon = randNorm.sample();
                spot_T[i][j] = spot_T[i][j-1] * exp(drift + diffusion * epsilon);
        }
}

for (int i = 0; i < paths; i++) {
        spot_T[i][steps] = max(0,strike - spot_T[i][steps]));
        sum_CT += spot_T[i][steps];
}

double price = sum_CT/paths * exp(-rate * expiry);
double stderr = stdev(spot_T)/sqrt(paths);
return price;
```

Normality in the sampling distribution is critical for arriving at accurate answers. The end prices should appear normally distributed if the sampling distribution is normal and many paths are used. This normality is dependent on the quality of the random number generation. Computers cannot generate truly random numbers but can produce numbers that seem random that usually will suffice. Key to this randomness is the periodicity of the generator, or how many numbers it can produce before it repeats the sequence. Our answers are dependent on the quality of the generator used. So it is important to use a generator with high periodicity.

To get the price to converge, many thousands of paths are needed. Because of the large number of paths needed, using this naïve type of Monte Carlo can be inefficient and slow, even for a computer. To compensate for this, we can adjust the way we take and use our random samples to converge on a price more quickly. The following are ways we can reduce the number of samples needed to converge on a price.

Antithetic

Antithetic sampling is a very simple way to improve the computing speed of the Monte Carlo method. It is done by creating two paths from a single sample. Just like a naïve Monte Carlo, we build a random walk from the equation above. The difference is that we will simultaneously build another path with a negative epsilon. This can be done be done because an epsilon mirrored across zero is equally likely to happen. This method will double the paths created from the same amount of random samples. This only gives a small boost in computational speeds. In the snippet of code below you can see that there is only a small change required in the for loop as compared the naïve Monte Carlo previously presented.

```
for (int i=0; i<paths; i++) {
    spot_T[i][0]= spot;
    spot_T[i+paths][0] = spot;
    for (int j=1; j<=steps; j++) {
        double epsilon = randNorm.sample();
        spot_T[i][j] = spot_T[i][j-1] * exp(drift + diffusion * epsilon);
        spot_T[i+paths][j] = spot_T[i+paths][j-1] * exp(drift - diffusion * epsilon);
    }
}
```

Stratified

Stratified sampling method is another way to reduce the number of samples needed to converge on a price, thus reducing calculation time. This method forces our sample to be taken from different percentile buckets. For example, if we were to take samples using five different percentile buckets, we would take a uniform random sample between 0.0 and 0.2 and between 0.2 and 0.4 and so on until 1. We then take these stratified uniform samples and convert them to a normally distributed samples using the cumulative distribution function. Sampling in this way will help our sampling appear normally distributed much faster. We can also increase the number of buckets to converge more quickly. This method can give a large increase in computational speed.

This method is simple to implement for European Options because only the end point is needed for calculations. American and some exotic options on the other hand need an extra step because they are path dependent. For these options we need to build a Brownian bridge. A Brownian bridge is a random path as we used before except we have known beginning and end points and the randomness is generated in-between. We find our stratified end points first using the same method as in European Options. Then with one of these end prices and our initial spot price we find another point halfway between the two using linear interpolation. This is the point that would be on our path if our path moved in a straight line from the time 0 to expiry. We need to add randomness to this point by using the following equation (Brandimarte):

$$S_{t_{mid}} = .5 * \left(S_{t_0} + S_{t_{final}}\right) + .5 * epsilon * \sqrt{t_{final} - t_0}$$

We now have a line moving from time 0 to the midpoint and the midpoint to expiry. We then bisect each of these lines the same as before and repeat the process until the desired amount of periods are created. The repeated bisection requires that the number of periods be a base 2 integer. A sample of this methods code can be seen in the file labeled "StratifiedMonteCarlo.java". It can be found with the entire program at the link given later in the paper.

Control Variate

There are other methods that can be used such as the control variate. The control variate makes use of known formulas such as Black Scholes to nudge the results in the correct direction. This will significantly reduce the number of samples needed but in my experience takes more computational time per path. So the efficiency of the code does determine the effectiveness of this method. I will not be using control Variate in my program because the use of a control variate on an American Option is beyond the scope of this paper.

Parallelization

Another very simple way to increase the program speed is through parallelization. Parallelization is the method of splitting up the computing task into different threads that can be run simultaneously (in parallel) on different CPUs. Monte Carlo simulations do very well in parallelization because the number of desired paths can easily be divided by the number of available processors and run separately on different threads. When all threads have calculated a price, the average can be taken to get the final price. The speed is obviously heavily dependent on the hardware used. The more processors available the faster it will run.

Longstaff-Schwartz

As previously mentioned, American options that are not plain vanilla can be difficult to price because there is not a closed form equation to solve them like the Black-Scholes formula for European options. American and many exotic options are often are too complex for a binomial tree. Also, a naïve Monte Carlo will not work because of these options ability to exercise before their expiry date, which means the asset’s whole price path, not just the final price, is needed to determine the option price. This problem of pricing options that are path dependent, is what also makes pricing Asian or many other exotic options difficult. To price more complicated options we need a more robust method.

Francis Longstaff and Eduardo Schwartz provide a method that accommodates path dependencies of these option types in their 2001 paper “Valuing American Options by Simulation: A Simple Least-Squares Approach”. This method is a combination of Monte Carlo simulation and regression analysis. It builds random price paths, as with other methods discussed earlier, but also performs a simple ordinary-least squares regression at each time step to determine when would be the statistically optimal time to exercise the option on each path. Their method although efficient requires many processes that must be repeated thousands of times, showing the need for computing power to be worth-while.

To conceptualize this method I will use the simplified example given by Longstaff and Schwartz (Longstaff). It values a put option on a non-dividend paying stock with a strike price of $1.10 and the possibility to exercise at times 1, 2, and 3. We are also assuming a risk-free rate of 6%.

Spot Price Paths

Path	$t = 0$	$t = 1$	$t = 2$	$t = 3$
1	1.00	1.09	1.08	1.34
2	1.00	1.16	1.26	1.54
3	1.00	1.22	1.07	1.03
4	1.00	.93	.97	.92
5	1.00	1.11	1.56	1.52
6	1.00	.76	.77	.90
7	1.00	.92	.84	1.01
8	1.00	.88	1.22	1.34

Their method begins the same as valuing a European option by Monte Carlo. You simulate the spot price through many price paths by using Geometric Brownian Motion. This creates a price matrix as seen above. We then create a cash-flow matrix by filling an array with null or zero values and then determine cash-flows at the expiry based on the options payoff formula. Being a put option, the payoff is the asset’s price at expiry less the strike price or, if this is negative, the pay-off is 0. The sample payoff matrix is seen below on the left.

Cash-Flow matrix at time 3

Path	$t = 0$	$t = 1$	$t = 2$	$t = 3$
1	-	-	-	.00
2	-	-	-	.00
3	-	-	-	.07
4	-	-	-	.18
5	-	-	-	.00
6	-	-	-	.20
7	-	-	-	.09
8	-	-	-	.00

Regression at time 2

Path	Y	X
1	$.00 \times .94176$	1.08
2	-	-
3	$.07 \times .94176$	1.07
4	$.18 \times .94176$	.97
5	-	-
6	$.20 \times .94176$	.77
7	$.09 \times .94176$	.84
8	-	-

.94176 is the discount rate for a single period

We must then move backwards and find the payoff at each previous period. In periods before expiry, the individual must determine if it is optimal to exercise the option early or hold onto it. To do this, we can use OLS to regress Y on X and X^2 where Y is the payoff in the following period discounted back one period using the risk-free rate and X is the spot price at that time. So, in our example we regress the payoff at t = 3 discounted by one period against the assets spot price at t = 2. Paths where the payoff is zero in the period in question should be removed from the regression because we know they will not be exercised. This also has the benefit of speeding up the program because there are fewer items to regress.

Cash-Flow matrix at time 2

Path	$t = 0$	$t = 1$	$t = 2$	$t = 3$
1	-	-	.00	.00
2	-	-	.00	.00
3	-	-	.00	.07
4	-	-	.13	.00
5	-	-	.00	.00
6	-	-	.33	.00
7	-	-	.26	.00
8	-	-	.00	.00

Final cash-flow matrix

Path	$t = 0$	$t = 1$	$t = 2$	$t = 3$
1	0.00	0.00	.00	.00
2	0.00	0.00	.00	.00
3	0.00	0.00	.00	.07
4	.17	0.00	.00	.00
5	0.00	0.00	.00	.00
6	.34	0.00	.00	.00
7	.18	0.00	.00	.00
8	.22	0.00	.00	.00

This regression gives us a conditional expectation function that can be used to determine the expected value of continuing to hold the option. In our example case, the equation is E[Y / X] = -1.070 + 2.983X - 1 .813x^2. We then compare the value of immediate exercise to the expected value of continuation. If the payoff value of immediate exercise is greater, this value is placed into the cash-flow matrix otherwise the cash-flow matrix entry is set to 0. If a path does exercise early all entries after it must be set to zero because the options cannot be exercised again so there will be no following cash-flows. This continued building of the cash-flow matrix is seen in the chart above on the left.

These steps are then repeated for each previous period until you reach time 0 (chart above on the right). You then discount the values in the cash-flow matrix back to time zero and take the average of the cash-flows over every path. This average is the final resulting price of the option.

This algorithm can take advantage of the Monte Carlo methods like antithetic, stratified, and control variate mentioned earlier. The stratified method is a bit more complicated but still possible and effective.

To view the algorithms code see the file named “LongstaffSchwarts.java” in the link in the next section.

An important part of using these different algorithms is using the right tool for the right job. If your option is plain vanilla and continuously exercisable, the binomial method will be the fastest by far. Moving to Monte Carlo methods will slow things down but provide accuracy to European options that have more complexity to them. The Longstaff-Schwartz method will be the slowest (still reasonably fast though) but can handle everything others can and much more. Anything with complexity and American or exotic style payoffs will want it’s power.

Application

To be useful these algorithms need computer power because of the huge amount of repetitions needed to solve them. I chose to implement these algorithms using the Java language. I chose Java because I was in the process of learning it for other projects I was working on. Luckily this language comes with some useful benefits. The main benefit of using java is that it is fast. Java is a compiled language, meaning the computer translates my code into binary code (computer language) before running so it runs much faster than an interpreted language that executes code step by step (like the Python or Matlab). Since this program requires many thousands of iterations speed is an important benefit. Also, once compiled, Java applications will run on most operating systems, so it can easily be transferred between computers. Lastly, Java is an object-oriented programming language. While most modern languages are object-oriented, in Java everything is an object and its syntax forces you to use OOP concepts. The use of OOP will be discussed more later. To view and use the application I created in Java, visit https://github.com/roblpetty/Plan-B-Java. Another simpler version is available in the python language at https://github.com/roblpetty/Plan-B-Python but will not be discussed in this paper.

I have created this code so it can be used in two ways. One is through a GUI (Graphical User Interface). The other requires the creation of a simple file, called a client file, to direct the program. The GUI implementation is useful because it doesn't require the user to have any programming knowledge. It appears as a simple calculator with text fields for inputs. The user enters the option parameters (option type, payoff, calculation method) and market data into the input field and just presses a button and the program does all the work to return an answer. While this makes the program user-friendly it greatly restricts some of the potential uses for the program.

The program can also be run from a simple client file. The client file is a short file with very little code that will direct the rest of the program. It holds the same options and entries as the GUI but in code form. It does require the users to have some basic programing knowledge but not much. Those willing to make their own client file and edit a few lines of code will find much more flexibility in the program and its uses. For example, the client file could easily be altered to run two or more options at once for comparison or can be altered so the options pricer can be integrated into a completely different program.

Object Oriented Programming (OOP)

In designing this program, I used object oriented programming which gives it more flexibility and makes it easier to upgrade or alter. In programming an “object” is an element that can contain attributes and methods which are analogous to variables and functions. They are created using “classes” which act as blue prints for the object. What makes objects and classes so special is that by building a class you can create multiple objects of the same type. They may be given different attributes but they are still the same type. For example I could create multiple objects of “dog” type (or class) and each has a “bark” method. One dog may have a color attribute “brown” but the other “black”. So each object is the same type but can have separate attributes and work independently of another.

Another important aspect of classes and objects is the concept of inheritance. Some classes depend on parent classes. Subclasses are able to inherit specified methods and attributes. For example, I could create an “animal” class with “eat” method. I could then create the “dog” class again with the “bark” method and have the dog class inherit from the animal class so that the dog can both eat and bark. Since all dogs are animals they can do everything the animal does. Not all animals are dogs, though, so not all animals can bark. Although these animal examples are silly they show well how object-oriented programming is used.

This concept of classes and objects is important to this program. I have created a simple “Option” class that has an expiry time, strike price, and a payoff function. I also have classes that inherit from this class such as vanilla and exotic Options. I also have an “Engine” class that holds functions and attributes common to all pricing engines but then created subclasses like “BinomialPricingEngine” class and “MonteCarloPricingEngine” class which hold functions and attributes specific to that type of engine but also inherit the attributes of its parent “Engine” class.

The use and importance of OOP is confusing at first but is very powerful. The use of objects make the program flexible and prevents the need for repetitious code. The program will require general type objects like an “Engine” object but can be given any object that inherit from “Engine” like “BinomialPricingEngine”. Objects can also work independently of other objects. This means that if we want to change the type of engine we don’t need to re-write our whole code. We just make a new engine object and swap it in. This makes upgrades to a program much easier because it can be edited object by object without fear of breaking the program as a whole.

Design Patterns

Design patterns are patterns in programming that make the code more functional. They are blueprints that can be used in many different programs and across many programming languages. I’ll explain a couple that I used and why they are useful.

Facade Pattern

The facade pattern is a way of hiding complex code behind a simple interface. I do this through the use of client files which I have previously mentioned. This way someone that wants to run my program doesn’t need to sift through the many files I created to make changes. This file for the most part is just a list of user defined variables that are needed to solve the pricing problem and then calls the needed methods to run the rest program. It is just a single short file that is easier for the user to digest and understand but gives full access to the power of the entire program. This makes the program much more accessible to those who are not very familiar with the program or coding in general.

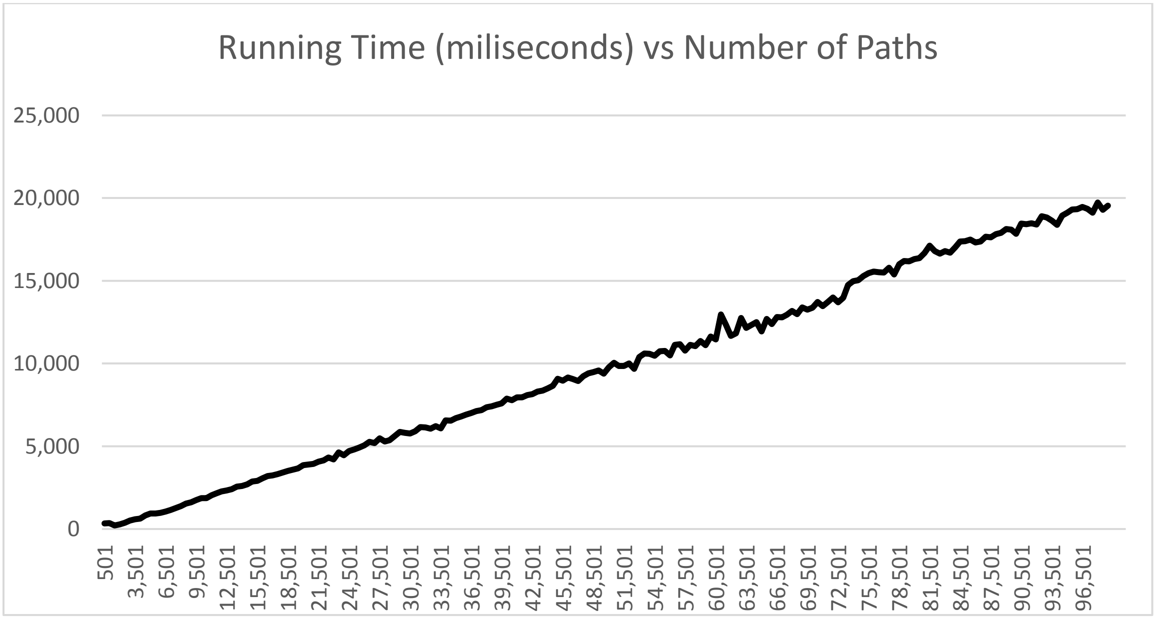
Strategy Pattern

The strategy pattern is a way of passing functions to objects. This allows objects of the same type to do different things. This is known as polymorphism. By making these objects polymorphic, it is easy to replace the pricing method. So for a Monte Carlo pricing method, we can easily switch between naïve, antithetic, or stratified. Also, it allows us to pass different payoff methods to our “option” object. Thus keeping the same “option” object but makes it act differently by passing it different functions to perform.

Program Results

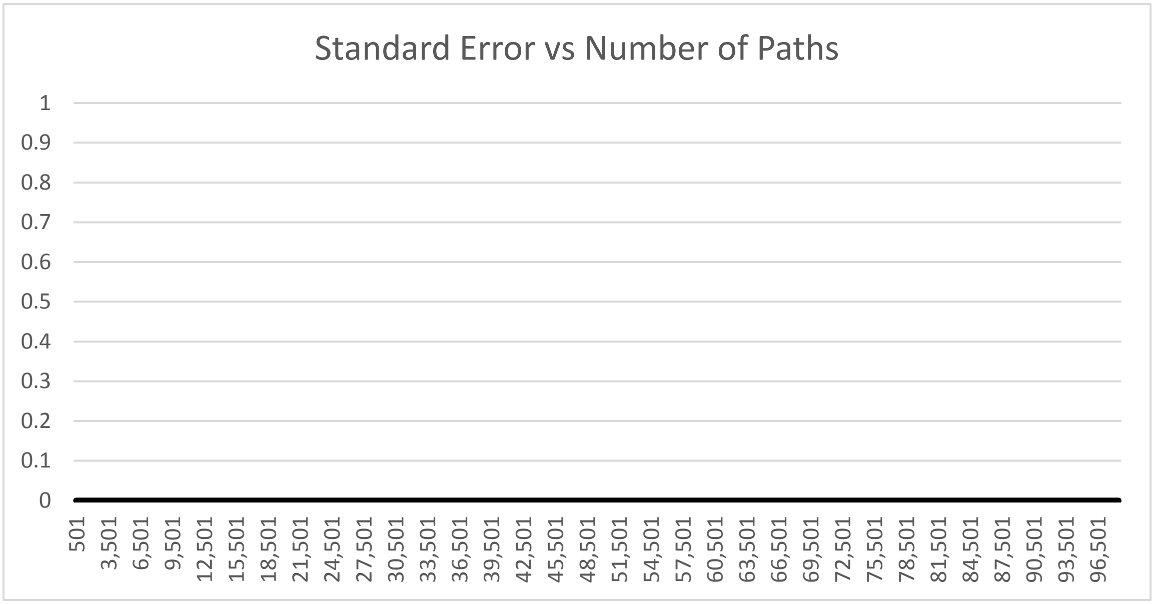
For this program to be useful to people in real world application accuracy and speed are essential factors. I have recorded the price, standard error, and running time while pricing an option at differing number of paths used in a Monte Carlo. I chose to price an American put option with a strike price of $40 and 1 year to maturity. I am assuming the underlying asset has a spot price of $41, 8% risk-free rate, 30% volatility, and no dividend. I ran pricing scenarios from 500 paths up to 99500 paths at 500 path intervals.

figure 1



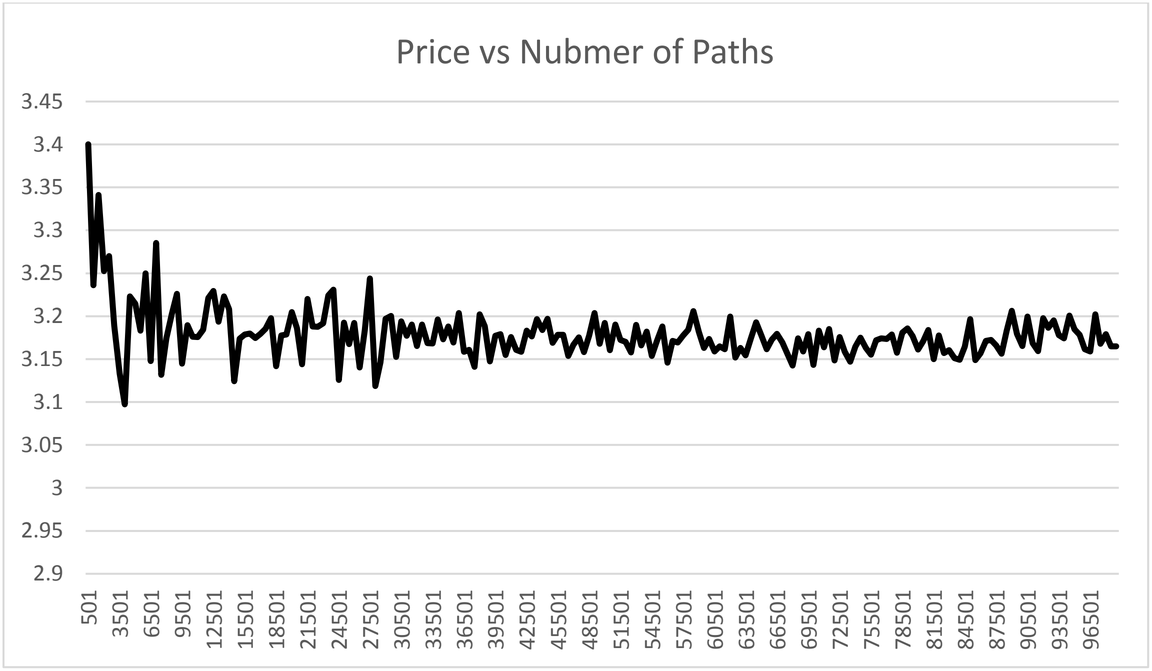
In figure 1, it is easy to see that time increases at a fairly linear rate as the number of paths increase. Also in figure 2, standard error decreases at a decreasing rate as paths increase. This means increasing running time is returning less and less improvements in standard error.

figure 2



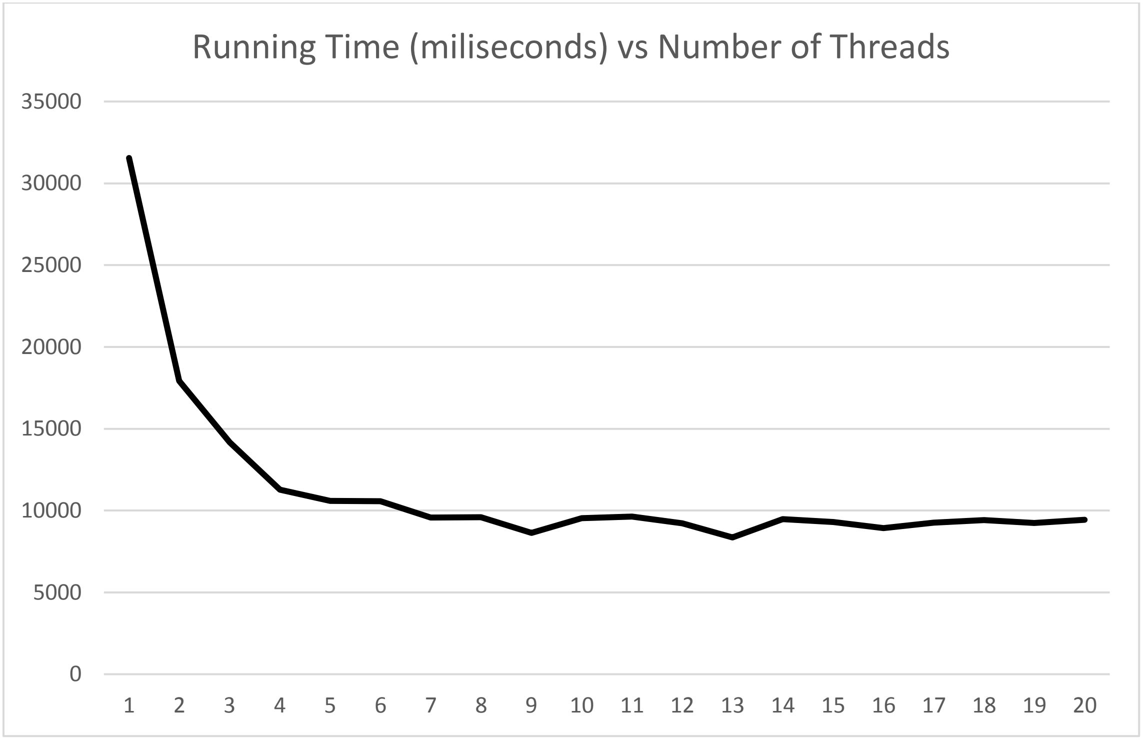
In figure 3, we again see that accuracy can rapidly improve by adding paths when paths are low and the standard error quickly converges. Unfortunately, even with near 100,000 paths there is about a 5 cent variation in prices. This is likely due to a low quality random number generator in the program. Meaning the numbers generated have short periodicity making them less useful in appearing truly random. This randomness is key to accurate results when using Monte Carlo simulations. In the future, it would be beneficial to find a new random number generator with longer periodicity.

figure 3



I also tested the effectiveness of using parallelization. I ran the program on a computer with four processing cores. I ran the same option as before but with 100,000 paths starting with 1 thread and moving up to 20 threads (figure 4). Steep reductions in running time can be seen when moving from 1 thread to 4. This is expected with four cores. We don't see large increases above four threads because each core can only process one task at a time. So if there are more threads than processing cores the threads have to take turns passing through it. There does seem to be a slight decrease in time by adding more threads than there are cores but it is not much and I would make a guess that it is because the program is able to process the regressions a little faster when there are fewer samples on each thread.

figure 4



Conclusion

Here I have demonstrated an effective tool for pricing options that allows the user access to more options scenarios. This tool lets people move beyond the simple European option environment that is usually taught in school. They can now price more complex European, American, and many exotic options that require path dependencies because of the Longstaff-Schwartz method. Also using proper computational methods allows faster pricing times which allows it to be more practical for real world applications.

References

Cox, Ross, Rubenstein. Option Pricing: a Simplified Approach. Journal of Financial Economics, Sep. 1979

Brandimarte, Paolo. Handbook in Monte Carlo Simulation. John Wiley & Sons, Inc., 2014.

Hull, John Options Futures, and Other Derivatives. 8th ed., Prentice Hall. 2012.

McDonald, Robert. Derivative Markets. Pearson Education. 2003.