Monte Carlo Pricing of a European Fixed Strike Lookback Call Option with Stochastic Volatility Using Black–Scholes Delta-, Gamma-, and Vega – Control Variates

By: Kyle Gregerson, Garrett Jones, Colby Granstrom

Lookback Option

A lookback option is an exotic option which lets investors to "look back" at the underlying prices that occur over the life of the option. The investors can then exercise based on the underlying asset's optimal value. Lookback options reduce uncertainties that are associated with when an investor enters the market. There are two types of lookback options, floating and fixed.

A floating lookback option is where the option's strike price is fixed at maturity. The fixed point is different for calls and puts. In the case of a floating call lookback option, the strike price is fixed at the lowest price point that is reached during the life of the option. In the case of a floating put lookback option, the strike price is fixed at the highest price point that is reached during the life of the option.

A fixed lookback option is where the option's strike price is fixed at the purchase of the option. The option is not exercised at the market price, however. In our case, a fixed call lookback option, the investor can choose to exercise the option at the point where the underlying asset was priced at the highest point over the life of the option. In the case of a fixed put lookback option, the investor can choose to exercise the option at the point where the underlying asset was priced at the lowest point over the life of the option.

Lookback Option Payoffs		
	Call	**Put**
Fixed	Max(M-K, 0)	Max(K-m, 0)
Floating	(St-m)	(M-St)

Lookback options are appealing to investors because the payoffs depend on the minimum or maximum price of the underlying asset that was attained during the life of the option or a certain period of the life of the option. Lookback options are also path dependent which is why using a Monte Carlo simulation to examine what could happen to lookback option's payoff over time is essential to estimating the option's value. A Monte Carlo simulation allows an investor to see all the possible outcomes of a particular decision and then assess the impact of risk. This allows for better decision making under uncertainty. The simulation gives the investor a range of possible outcomes.

Monte Carlo Simulation

The Monte Carlo simulation relies on Brownian motion to calculate the possible outcomes of a random event. A Brownian motion is a continuous stochastic process that models random behavior that evolves over time. It can be applied to pricing options or stock price fluctuations because it can simulate a limited random walk. This means that for every time step there can be many different outcomes and a Brownian motion can simulate these using a mathematical formula. The Brownian motion is also called the Wiener process, named after Norbert Wiener, who developed mathematical theories that ultimately proved the existence of Brownian motions.

In a Monte Carlo simulation, you must first define the variables that are not random. This is important because the model uses the parameters defined in the simulation to specify the function outcomes of the random variable. The series of random variables create the different price paths that an option can take. The formula to find the different price paths is:

$$S(t + \Delta t) = S(t)e^{\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma\varepsilon\sqrt{\Delta t}}$$

This formula samples one specific price movement; the results are recorded and then the formula is repeated for each step in the price path. After a price path is mapped the program repeats the simulation, the process can be repeated hundreds or thousands of times. This is to get a large enough sample of price paths so the investor can make a more informed decision. In a Monte Carlo simulation for lookback options, each ending payoff is collected and then averaged. After the average payoff is calculated, that value is then discounted back to its present value in order to find the actual value of the option today.

Monte Carlo Model Expansions

Monte Carlo simulations rely on repetition to increase accuracy of estimation. However, there are additional techniques that can be applied to the Monte Carlo simulation to further increase the accuracy of estimation while also decrease the time and power required for computation. Methods include: antithetic sampling, stratified sampling, control variates, and stochastic volatility. For our project we have implemented the control variate variance reduction technique as well as stochastic volatility.

Control Variates

The control variate method, as used as a variance reduction technique within the Monte Carlo simulation, uses given information to improve model efficiency through reducing sampling error. We are comparing a known value with the model estimated value to adjust the objective unbiased estimator - to receive a more accurate estimation that reduces the sampling error. In this model, we are comparing Black – Scholes Greeks with Monte Carlo estimated Greeks in order to improve the accuracy of price estimates. We used the analytical solutions for Black - Scholes Delta, Gamma, and Vega compared to Delta, Gamma, and Vega estimated from the price simulation to adjust the premium of each price path.

Stochastic Volatility

Stochastic volatility is a more realistic representation of price fluctuations and is implemented into the model to increase the external validity of our Monte Carlo simulation estimates. In standard Binomial, Monte Carlo, and Black - Scholes models, volatility is assumed to be constant, this is one of the limiting factors of neo-classical pricing models. Our model allows for volatility to change over time, due to random shocks from a standard normal distribution, to adjust the path of variance over time. These random shocks are independent of the random shocks that determine changes in the price path.

Source Code

Object Oriented Implementation

Due to Python's class and module structure, the software engineer can implement design patterns and strategies, including encapsulation, inheritance, polymorphism, and the façade. These allow the end – user to have total functionality of the program, without needing the underlying knowledge or access to the underlying code. Likewise, changes or additions to the functionality of the code can be implemented without requiring the end – user's functionality to change, allowing for more fluid updates with new code. Within this project, we implemented this philosophy to update our object - oriented pricing engine to include the control variate Monte Carlo pricing method and lookback option payoff.

Pricing Engine. Within our module named Dylan, we implemented our control variate engine class, which inherits from our control variate Monte Carlo pricing program, which then inherits the pricing variables such as steps, replications, alpha, Vbar, and Xi, in order to run. The pricing technique is implemented as follows:

```
def ControlVariateMonteCarloPricer(pricing_engine, option, data):
    expiry = option.expiry
    strike = option.strike
    alpha = pricing_engine.alpha
    xi = pricing_engine.xi
    Vbar = pricing_engine.Vbar
    (spot, rate, volatility, dividend) = data.get_data()
    dt = expiry / pricing_engine.time_steps
    #nudt = (rate - dividend - 0.5 * volatility * volatility) * dt
    xisdt = xi * np.sqrt(dt)
    erddt = np.exp((rate - dividend) * dt)
    egam1 = np.exp(2*(rate-dividend)*dt)
    egam2 = -2*erddt+1
    eveg1 = np.exp(-alpha*dt)
    eveg2 = Vbar - Vbar *eveg1

    beta1 = -1
    beta2 = 1
    beta3 = 1

    payoff = np.zeros((pricing_engine.replications, ))
    price = 0.0

    for j in range(pricing_engine.replications):

        cv1 = 0
        cv2 = 0
        cv3 = 0

        s = np.zeros(pricing_engine.time_steps)
        v = np.zeros(pricing_engine.time_steps)
        s[0] = spot
        v[0] = Vbar
        z1 = np.random.normal(size=int(pricing_engine.time_steps))
        z2 = np.random.normal(size=int(pricing_engine.time_steps))

        for i in range(int(pricing_engine.time_steps)):
            t = (i-1) * dt
            delta = BlackScholesDelta(s[i], t, strike, expiry, volatility, rate, dividend)
            gamma = BlackScholesGamma(s[i], t, strike, expiry, volatility, rate, dividend)
            vega = BlackScholesVega(s[i], t, strike, expiry, volatility, rate, dividend)

            ##### Evolve Variance #####
            v[i] = v[i-1] + alpha * dt + (Vbar - v[i-1]) + xisdt * z1[i]
            if v[i] < 0.0:
                v[i] = 0.0

            #####Evolve Asset Price #####
            s[i] = s[i-1] * np.exp((rate - 0.5 * v[i-1]) * dt + np.sqrt(v[i-1]) * np.sqrt(dt) * z2[i])

            ##### Accumulate Control Variates #####
            cv1 = cv1 + delta * (s[i] - s[i-1] * erddt)
            cv2 = cv2 + gamma * ((s[i] - s[i-1]) * (s[i] - s[i-1]) - s[i-1] * s[i-1] * (egam1 * np.exp(v[i-1]*dt) + egam2))
            cv3 = cv3 + vega * ((v[i]- v[i-1])-(v[i] * v[i] * eveg1 + eveg2 - v[i-1]))

        strike = option.strike
        payoff[j] = option.payoff(strike, s) + beta1 * cv1 + beta2 * cv3 + beta3 * cv3


    price = np.exp(-rate * expiry) * payoff.mean()
    #stderr = payoff.std() / np.sqrt(engine.replications)
    return price
```

Payoff. Within Dylan we created an abstract class – Payoff – that is used as an option payoff interface, particularly for our class exotic payoff – which then allows our lookback call option payoff function to run. As follows:

```
class Payoff(object, metaclass=abc.ABCMeta):
    """
    An option payoff interface.

    """

    @property
    @abc.abstractmethod
    def expiry(self):
        pass

    @expiry.setter
    @abc.abstractmethod
    def expiry(self):
        pass

    @abc.abstractmethod
    def payoff(self):
        pass

class ExoticPayoff(Payoff):
    def __init__(self, expiry, strike, payoff):
        self.__expiry = expiry
        self.__strike = strike
        self.__payoff = payoff

    @property
    def expiry(self):
        return self.__expiry

    @expiry.setter
    def expiry(self, new_expiry):
        self.__expiry = new_expiry

    @property
    def strike(self):
        return self.__strike

    @strike.setter
    def strike(self, new_strike):
        self.__strike = new_strike

    def payoff(self, spot):
        return self.__payoff(self, spot)

def Lookback_Call_Payoff(strike, spot):
    """
    The payoff function for a European lookback call option.

    Args:
        option:     the self variable from the Payoff class that aggregates the function.
        spot (array): the price path of the underlying asset
    """

    return maximum(np.amax(spot) - strike, 0.0)
```

Conclusion

The design pattern within Python allows for greater malleability for the software engineer, and an easier user interface for the professionals using the program and its functions. As the finance industry becomes more efficient in its use of data analytics and collection, it becomes important to have fast and accurate computational techniques, that are dynamic with different assets. With the design pattern, Python allows the implementation of new classes and functions that run fluidly with the previous code, giving the end user flexibility without the possibility they break the programmer's code. As computational techniques become more efficient, the ability to optimize these techniques becomes a very valuable skillset for finance professionals, and will continue to advance at a speed equal or great than the current advancement of financial markets.