Stevens Institute of Technology

Financial Engineering

Effect of Jump Diffusion Price Dynamics on European S&P 500 Index Options

Authors:
Zach Townley-Smith
Evan Ganning
Scott Slawson

Faculty Advisor:
Dr. Khaldoun Khashanah

May 14, 2022

Abstract

This paper investigates the suitability of jump diffusion models as a representation of the S&P 500 ETF (SPY) price and what response in SPX option prices is seen following a SPY price jump. Log-Normal and Double-Exponential jump diffusion models are calibrated daily to match market returns by maximum likelihood estimation and then used to price standard maturity, European SPX options using Fourier Transform methods. Market SPY ETF and SPX option price data is higher frequency, at a minute resolution, covering the years 2010-2012. SPY jumps are detected by comparing the ratio of the log-return to local volatility. The modeled and market option time series around the time a jump is detected are compared by cross correlation to determine lead/lag time for an option response to an ETF jump. This study provides coverage for calibration implementation with higher frequency price observations and also couples jump detection and calibration methodologies for more stable model parameter estimation. We find that Jump Diffusion Models are a better match to market returns than Black Scholes Models, though the theoretical adjustment in the option space is a small correction. Additionally, it is found that most response times occur within one minute. The extension of this analysis to high frequency data is left for further study.

Keywords: Option Pricing, Black Scholes, Jump Diffusion, Implementation
JEL Classification: G120, G130

1 Introduction

Though the modeling of stock price movements as a Geometric Brownian Motion (GBM) and pricing options using the analytic equations proposed by Black and Scholes (1973) revolutionized quantitative finance, this view of the market was limited. In practice, the assumption that asset log returns are normally distributed is demonstrably false, with market returns displaying "fatter tails" that a normal distribution does not capture Officer (1972). Another known deviation in market data from the Black Scholes Model (BSM) is the existence of a "Volatility Smile" in implied volatilities. This is to say that, using market prices and risk-free rates, the calculated volatility that solves the Black Scholes pricing equation is not constant with strike price as it is if the asset follows geometric Brownian motion.

One attempt to improve upon BSM was to add the potential for discrete, independent jumps to stock prices as noted in Merton (1976). In Merton's Jump Diffusion model (MJD), the arrival times of these pricing jumps are given by an independent Poisson process and the magnitude of the jump is given by an independent, log-normal random variable. Another popular jump diffusion model is given by Kou (2002) which modifies MJD by having jump magnitudes follow a double exponential distribution instead of a log-normal one. Both MJD and Kou's Double Exponential Jump Diffusion Model (DEJD) allow for heavy tails and a leptokurtic shape in their return distributions, which could better match market returns and also lead to implied volatility smiles in priced options.

While jump diffusion models improve upon GBM, they are not themselves without limitations. As Kou (2002) points out, a main weakness of jump diffusion models is an inability to capture volatility clusters which could be characterized in a stochastic volatility model. A model that combines stochastic volatility, discrete price jumps, and volatility jumps offers a more robust representation of asset prices, but with added complexity comes computational challenges in both estimating model parameters and pricing options. Jump diffusion models strike a balance between model complexity and usability which is why they continue to persist in practice.

The goal of this project is to investigate the adequacy of the MJD and DEJD models as a representation of the S&P 500 ETF (SPY) price and to explore the effect that SPY price jumps have on European SPX options. Our scope includes ETF and option pricing data, at a one-minute resolution, from January 2010 through December 2012. Jump diffusion models are calibrated daily to match the distribution of recent market SPY returns and then used to price standard maturity SPX options. The main results of this study include a determination of the lead/lag time in SPX options responding to a SPY price jump and a comparison of how well these jump diffusion models match market option prices versus BSM.

The remainder of the report is structured as follows: Section 2 covers relevant prior work, Section 3 discusses jump detection, model calibration, and option pricing methodologies employed, Section 4 covers our design of experiment, Section 5 is a discussion of results, and Section 6 concludes the study.

2 Literature Review

2.1 Model Background and Initial Option Pricing

Merton (1976) demonstrates that introducing discrete price jumps from an independent Poisson process introduces a risk that cannot be perfectly hedged in a portfolio of stocks and options. As a result, the no arbitrage arguments used by Black and Scholes (1973) to develop option pricing equations are not applicable to jump diffusion models of the stock price. In his paper, Merton works through the Ito calculus to describe the dynamics of a European option price under jump diffusion, though he is unable to obtain a closed form solution to his differential equation for the option price. However, Merton is able to approximate the price of an option by assuming that assets are priced under the Capital Asset Pricing Model (CAPM) and that the jump-component of an asset's return is uncorrelated with the market.

Kou proposes a jump diffusion model in his paper, whose jump magnitude is given by a double exponential distribution. The double exponential distribution has a fundamental explanation in that investors may respond differently to positive or negative news about a company. Beyond having an economic meaning, the double exponential distribution also exhibits a memory-less property which enables closed form pricing equations for some path dependent options including American, barrier, and lookback options. Both MJD and DEJD allow for leptokurtic returns, heavy tails compared to GBM, and implied volatility smiles while portraying a complete market that does not permit arbitrage. The ability to derive closed form pricing equations for path dependent options is the main advantage of DEJD over MJD Kou (2002).

2.2 Fourier Methods for Pricing Options

The mathematical challenges seen in Merton (1976) are greatly simplified through use of a Fourier Transformation. Scott (1997) and Bakshi and Madan (2000) show that when the characteristic function of the terminal stock price is known, the probability that an option will finish in the money ρ and the delta of the option δ can be directly computed. This allows for direct pricing of a European option whose underlying does not pay dividends: $Call = S\delta - K\rho e^{-rT}$. Though this is a tractable formulation, and the required characteristic functions are generally known, the computation time for pricing options via this method can be improved upon.

Carr and Madan (1999) provide two methods for obtaining the price of a European option for processes with known characteristic functions that are compatible with the fast fourier transform (FFT) algorithm which is known for its computational advantages. The first modifies the call option price by multiplying by an exponential term to ensure that the value for the option as a function of it's log strike is square integrable, which enables FFT to be used. The second method corrects oscillatory behavior seen in solving pricing integrals for out-of-the-money options near maturity by introducing a hyperbolic sine smoothing factor. Carr and Madan find that these FFT methods are 20-50 times faster than the δ, ρ approach discussed above.

Lewis (2001) is able to build off of the work done by Carr and Madan (1999) in deriving a pricing equation that holds for any path independent option whose underlying asset's price dynamics are driven by a Levy process. This is a much broader, general result which includes all jump diffusion models (pure JD models without stochastic volatility) and both standard/exotic options with European execution. Lewis' method makes use of the Fourier Transformation for the payoff of the option in addition to the characteristic function for the asset price process. This formulation allows for the price of the option to be expressed as a single complex integral of the product of the characteristic function and transformed option payoff. Depending on the contours chosen in handling the complex integral, and resulting residue calculus, Lewis shows that the price of a European Call option can even be expressed as a single real integral.

2.3 Jump Detection

The project is concerned with how option prices respond around the time of a jump, and so another preliminary requirement is to define what magnitude of a return constitutes being a jump. Lee and Mykland (2008) propose a test statistic for classifying a log return as a jump that uses the ratio of the return to an estimate of local volatility. The approach uses the result that as $\Delta t \to 0$, this ratio approaches infinity if the log return is a jump but approaches a known distribution if the log return arises from diffusion fluctuations alone. The test is non parametric, which is beneficial in that it can be generalized to any type of jump diffusion model, as opposed to being model specific. Their method performs well against prior non-parametric detection methods proposed by Barndorff-Nielsen and Shephard (2006), Jiang and Oomen (2005) both in terms of accuracy and computational demand.

2.4 Model Calibration Techniques

An intuitive technique for calibrating jump diffusion models is to solve for model parameters which minimize the squared pricing error between observed and modeled option prices. Cont and Tankov (2004) demonstrate the limitations of this approach: that multiple, sufficiently different sets of model coefficients can be fit to the same market option data and that the optimization problem is susceptible to local optima due to being non-convex. To transform this minimization to a stable, well posed optimization, Cont and Tankov recommend applying a convex penalty term equal to the difference in relative entropies between the current model parameters and a prior reference set of parameters .

An alternative calibration strategy is to solve for jump diffusion parameters which fit a set of market returns on the underlying asset, as in Hanson and Zhu (2004). Their approach uses second order approximations to the distribution of log-returns for jump diffusion models to solve for model parameters via maximum likelihood estimation. MJD, DEJD, and a log-uniform jump diffusion model are in scope for their calibration study, where model parameters are determined by fitting to daily return data from 1992-2001. To reduce the number of parameters to solve for by 2, Hanson and Zhu impose that the calibrated jump diffusion model must match the first and second moments of the input market return data.

3 Methodology

3.1 Jump Detection

Our paper employs the detection methodology described by Lee and Mykland (2008) who propose a statistic for determining if an observed log return is a jump whose occurrence would not be expected under a diffusion-only process. The motivating idea is that a jump should fall outside the typical noise levels set by the volatility of the underlying process which drives most price fluctuations. Their test statistic L, defined in Equation (1), is able to classify log returns as jumps by comparing the return magnitude to the local volatility $\hat{\sigma}$ at that time. If jumps are infrequent, the volatility of the diffusive process at time t_i can be estimated by the variation of returns in a window leading up to that time.

$$L(i) = \frac{log(S(t_i)) - log(S(t_{i-1}))}{\hat{\sigma}(t_i)} \tag{1}$$

Bipower variation, the absolute value of the product of adjacent log returns, is used as an estimate of local volatility $\hat{\sigma}(t_i)$ and given by Equation (2). Bipower variation is used to estimate volatility instead of power variation, the sum of squared returns, because it is more applicable for jump processes Barndorff-Nielsen and Shephard (2004). K is the number of price observations in the leading window used to calculate $\hat{\sigma}(t_i)$. The square root of the annualized number of observations, depending on the resolution of pricing data that is being worked with, provides a condition for the minimum sufficient window size K. Increasing K above this level increases computational costs without further improving accuracy.

$$\hat{\sigma}(t_i)^2 = \frac{1}{K-2} \sum_{j=i-K+2}^{i-1} \left| log \frac{S(t_j)}{S(t_{j-1})} \right| \left| log \frac{S(t_{j-1})}{S(t_{j-2})} \right| \tag{2}$$

$$K \geq ceiling(\sqrt{252 * n_{obs/d}})$$

Lee and Mykland (2008) show that as $\Delta t \to 0$, $|L(i)| \to \infty$ if the return is a jump and $L(i) \to N(0, \frac{\pi}{2})$ if the return is from diffusive fluctuations. This behavior can then be used to set a maximum value for the statistic above which a candidate log return is classified as a jump. Equation (3) states that as $\Delta t \to 0$, the maximum $L(i)$ tends toward ξ which has a cumulative distribution function $P(\xi \leq x) = exp(-e^{-x})$. The proof of Equation (3) follows from earlier work by Galambos (1978) and Aldous (1989).

$$\frac{max|L(i)| - C_n}{S_n} \to \xi \tag{3}$$

Where n is the number of price observations and constants C_n, S_n and c are given by:

$$C_n = \frac{\sqrt{2log(n)}}{c} - \frac{log(\pi) + log(log(n))}{2c\sqrt{(2log(n))}} \qquad S_n = \frac{1}{\sqrt{c(2log(n)}} \qquad c = \frac{\sqrt{2}}{\sqrt{\pi}}$$

3.2 Jump Diffusion Model Calibration

Our paper employs the methodology described by Hanson and Zhu (2004) who implement a multinomial maximum log likelihood approach to fitting jump diffusion model parameters. The setup is as follows:

1. Bin the vector of market log returns into a histogram and calculate the frequency f^{sp} for every log return bin.
2. Given a choice of model parameters x, calculate the expected frequency f^{jd} in each log return bin B_b, by integrating the density function of jump diffusion model returns ϕ^{jd} over the log-return space of the bin η.

$$f_b^{jd}(x) \equiv ns \int_{B_b} \phi^{jd}(\eta; x) d\eta$$

3. Return the jump diffusion model parameters which minimize the objective function:

$$y(x) \equiv -\sum_{b=1}^{nb} [f_b^{sp} log(f_b^{jd}(x))] \tag{4}$$

In this implementation, the integral of the jump diffusion density function is evaluated using the following second-order, numerical approximations to the distribution function of log returns. Equation (5) gives the return distribution for Merton's log-normal jump diffusion model and Equation (6) for Kou's double exponential jump diffusion model.

$$\Phi_{mjd}(\eta_1, \eta_2) \approx \frac{\sum_{k=0}^{2} p_k(\lambda\Delta t)\Phi_n(\eta_1, \eta_2, \mu + k\mu_j, \sigma^2 + k\sigma_j^2)}{\sum_{k=0}^{2} p_k(\lambda\Delta t)} \tag{5}$$

Where $\Phi_n(\eta_1, \eta_2, \mu, \sigma^2)$ is the normal cumulative distribution function over the interval $[\eta_1, \eta_2]$ having mean μ and variance σ^2. Δt is the time increment in years between each price observation in the sample data. $\mu \equiv \sqrt{\mu_{ld}\Delta t}$ and $\sigma \equiv \sqrt{\sigma_d^2 \Delta t}$.

$$\Phi_{dejd}(\eta_1, \eta_2) \approx \frac{\sum_{k=0}^{2} p_k(\lambda\Delta t)\Phi_{dejd}^{(k)}(\eta_1, \eta_2)}{\sum_{k=0}^{2} p_k(\lambda\Delta t)} \tag{6}$$

Where $\Phi_{dejd}^{(0)}$, $\Phi_{dejd}^{(1)}$ and $\Phi_{dejd}^{(2)}$ are defined as:

$$\Phi_{dejd}^{(0)} \equiv \Phi_n(\eta_1, \eta_2, \mu, \sigma^2)$$

$$\Phi_{dejd}^{(1)} = \Phi_n(\eta_1, \eta_2, \mu, \sigma^2) + p_1(\psi_{\eta_2,\nu_1} - \psi_{\eta_1,\nu_1}) + p_2(\psi_{\eta_1,\nu_2} - \psi_{\eta_2,\nu_2})$$

$$
\begin{aligned}
\Phi^{(2)}_{dejd} = \Phi_n(\eta_1,\eta_2,\mu,\sigma^2) &+ \mu_1\Bigg(\Big(\epsilon_{12} + \epsilon_{11}\big(\mu - \frac{\sigma^2}{\mu_1} + \mu_1 - \eta_2\big)\Big)\psi_{\eta_2,\nu_1} - \Big(\epsilon_{12} + \epsilon_{11}\big(\mu - \frac{\sigma^2}{\mu_1} + \mu_1 - \eta_1\big)\Big)\psi_{\eta_1,\nu_1}\Bigg) \\
&+ \mu_2\Bigg(\Big(\epsilon_{12} - \epsilon_{22}\big(\mu - \frac{\sigma^2}{\mu_2} - \mu_2 - \eta_1\big)\Big)\psi_{\eta_1,\nu_2} - \Big(\epsilon_{12} - \epsilon_{22}\big(\mu - \frac{\sigma^2}{\mu_2} - \mu_2 - \eta_2\big)\Big)\psi_{\eta_2,\nu_2}\Bigg) \\
&+ \frac{\sigma}{\sqrt{2\pi}}(\mu_2\epsilon_{22} - \mu_1\epsilon_{11})(e^{-z_1^2/2} - e^{-z_2^2/2})
\end{aligned}
$$

Using the below variable definitions for ν, ψ_{η_x,ν_y}, z and ϵ terms:

$$
\begin{aligned}
\nu_1 &= \mu - 0.5\sigma^2/\mu_1 & \nu_2 &= \mu + 0.5\sigma^2/\mu_2 \\
\psi_{\eta_2,\nu_1} &= e^{(\eta_2-\nu_1)/\mu_1}\Phi_n(-\eta_2, -\mu + \sigma^2/\mu_1, \sigma^2) & \psi_{\eta_1,\nu_1} &= e^{(\eta_1-\nu_1)/\mu_1}\Phi_n(-\eta_1, -\mu + \sigma^2/\mu_1, \sigma^2) \\
\psi_{\eta_1,\nu_2} &= e^{-(\eta_1-\nu_2)/\mu_2}\Phi_n(\eta_1, \mu + \sigma^2/\mu_2, \sigma^2) & \psi_{\eta_2,\nu_2} &= e^{-(\eta_2-\nu_2)/\mu_2}\Phi_n(\eta_2, \mu + \sigma^2/\mu_2, \sigma^2) \\
z_1 &= (\eta_1 - \mu)/\sigma & z_2 &= (\eta_2 - \mu)/\sigma \\
\epsilon_{11} &= (p_1/\mu_1)^2 & \epsilon_{22} &= (p_2/\mu_2)^2 \\
\epsilon_{12} &= 2p_1p_2/(\mu_1 + \mu_2)
\end{aligned}
$$

Under these formulations, we now have the expected frequency of observations in a bin of log returns f^{jd} as a function of $(\mu_{ld}, \sigma_d, \lambda, \mu_j, \sigma_j)$ parameters for Merton's jump diffusion model and $(\mu_{ld}, \sigma_d, \lambda, \mu_1, \mu_2, p_1)$ parameters for Kou's double exponential jump diffusion model. μ_{ld} is the log-diffusive drift, σ_d is the volatility of the diffusion process, and λ is the jump intensity or the expected number of jumps annually. In the MJD model, μ_j is the expected jump magnitude and σ_j is the variance of jump magnitude. In the DEJD model μ_1 is the expected magnitude of negative jumps, μ_2 is the expected magnitude of positive jumps and p_1 is the probability of having a downward price jump. Note that the probability of a positive price jump $p_2 = 1 - p_1$ and that $\mu_1, \mu_2 > 0$.

The number of model parameters to fit can be reduced by imposing that the first and second moments of the calibrated jump diffusion model (M_1^{jd},M_2^{jd}) are equal to the first and second moments of the sample of log returns used in the calibration (M_1^{sp},M_2^{sp}). This sets the μ_{ld} and σ_d parameters on both diffusion models as:

$$
\begin{aligned}
\mu_{ld} &= (M_1^{sp} - \mu_j\lambda\Delta t)/\Delta t \\
\sigma_d^2 &= \big(M_2^{sp} - (\sigma_j^2 + \mu_j^2)\lambda\Delta t\big)/\Delta t
\end{aligned}
$$

In the case of Merton's Jump diffusion model, we have μ_j and σ_j as direct outputs from the calibration. For Kou's double exponential model, the mean and variance of jump magnitudes can be calculated by the following equations:

$$
\begin{aligned}
\mu_j &= -p_1\mu_1 + p_2\mu_2 \\
\sigma_j^2 &= p_1\big((\mu_j + \mu_1)^2 + \mu_1^2\big) + p_2\big((\mu_j - \mu_2)^2 + \mu_2^2\big)
\end{aligned}
$$

3.3 Option Pricing Under Jump Diffusion

Our paper employs the pricing equations derived by Lewis (2001). In his paper he proves that the value of a path independent option for a jump diffusion process can be found by Equation (7). The method builds off of prior work in pricing options under jump diffusion through Fourier Transformations from Carr and Madan (1999) who were able to derive closed form option pricing equations by applying a Fourier Transformation to the terminal stock price.

Lewis' improvement comes about by also applying a Fourier Transformation to the payoff of the option, which yields more concise pricing equations. In this representation, r is the risk free rate, T is the time to maturity of the option in years, $\phi_T(z) = E[e^{izX_T}]$ is the characteristic function for the Levy process X_T and $\hat{\omega}(z)$ is the Fourier Transform for the payoff of the option $\omega(x)$.

$$V(S_0) = \frac{e^{rT}}{2\pi} \int_{i\nu-\infty}^{i\nu-\infty} e^{izY} \phi_T(-z)\hat{\omega}(z)dz \tag{7}$$

$$Y = log(S_0) + (r-q)T \qquad\qquad z = u + i\nu$$

Our project is concerned with European Call and Put options which have the following payoffs $\omega(x)$ and Fourier Transformation $\hat{\omega}(z) = \mathscr{F}|\omega(x)|$, respectively, per Lewis (2000).

$$\omega(x) = (e^x - K, 0)^+ \qquad\qquad \hat{\omega}(z) = -\frac{K^{iz+1}}{z^2 - iz}$$

$$\omega(x) = (K - e^x, 0)^+ \qquad\qquad \hat{\omega}(z) = -\frac{K^{iz+1}}{z^2 - iz}$$

The characteristic functions for the two jump diffusion processes we are studying are well understood and are given by Equation (8) and Equation (9) below. Derivation of the characteristic function for the log normal jump diffusion model is with credit to Carr and Wu (2003) and the characteristic function for the double exponential jump diffusion model is per Kou and Wang (2004).

$$\phi_{mjd}(z) = exp[iz\theta T - \frac{1}{2}z^2\sigma^2 T + \lambda T(e^{iz\alpha - z^2\delta^2/2} - 1)] \tag{8}$$

$$\phi_{dejd}(z) = exp[iz\theta T - \frac{1}{2}z^2\sigma^2 T + \lambda T(e^{izk}\frac{1-\eta^2}{1+z^2\eta^2} - 1)] \tag{9}$$

Working with the the transform for a European call option payoff $\hat{\omega}(z)$ in Equation (7), Lewis (2001) is able to express the value of the option as an infinite, real integral. This result is one of several variations depending on the contour ν used in solving the complex integral and resulting residue calculus. Equation (10) is the form used in our implementation for pricing European call options, using the characteristic functions in Equations (8) and (9). The price of a put $P(S_0)$ is calculated by the put-call parity.

$$C(S_0) = S_0 - \frac{\sqrt{S_0 K} e^{-rT/2}}{\pi} \int_0^\infty Re[e^{izk} \phi_T(z - i/2, T)] \frac{dz}{z^2 + 1/4} \tag{10}$$

$$P(S_0) = C(S_0) - S_0 + K e^{-rT}$$

From a coding implementation, we leverage Python scripts provided by Hilpisch (2014) for valuing options by Equation (10).

4 Experimental Design

4.1 Data

The scope of our project includes SPX options and SPY exchange traded fund price movements from 2010 - 2012, at a minute resolution. The set of SPX options in the study was chosen by selecting one at-the-money, one in-the-money, and one out-of-the-money option for both calls/puts for a total of six options per month. The definition of in-the-money vs out-of-the-money used in choosing strikes for each month were the nearest strikes K satisfying $K \geq 1.05(S_0)$ and $K \leq 0.95(S_0)$ respectively. Selected options are all standard maturity, expiring on the third Friday of the month and the price of the option is followed from the Monday after the third Friday until expiration.

In terms of data cleaning, our minute SPX option and SPY ETF price datasets were converted to eastern time and subset to only include observations between 9:30 and 16:15 when the Chicago Board Options Exchange is open for trading SPX options. Missing price observations were padded in both data sets to give full coverage over the trading day. The risk free rate was taken to be the yield on a one month treasury bond, updated daily.

4.2 Jump Detection

As described in the jump detection methodology in Section 3, a window size K must be specified in order to calculate the local volatility. In the project's implementation, $K = ceil[\sqrt{24 * 60 * 252}]$ or 603 return observations. Additionally, a 1% tolerance was used in setting the threshold value above which $\frac{|L(i)| - C_n}{S_n}$ classified the log return at time i as being a jump. Considering the exponential distribution of ξ, this translates to a threshold value of $-ln(-ln(0.99)) = 4.6001$. Note that the "opening jump" at 9:30 each morning was discarded from the results

4.3 Jump Diffusion Model Calibration

Model parameters are calibrated daily for the log-normal and double exponential jump diffusion models. Minute resolution log returns from the prior 14 trading days are input into the maximum log likelihood estimation methodology described in Section 3. In their paper, Hanson and Zhu (2004) work with daily log-returns using the adjusted closing price and allow the optimization to calculate jump intensity, among other model parameters. When scaled down to the higher frequency level of minute returns, the optimal jump intensity λ was found to be an unstable prediction. To improve model stability, λ was removed from the optimization and fixed at the amount of jumps detected in the 14 day window (expressed annually) of the return data used in the MMLE calibration. This instability is a finding discussed in Section 5 of this paper but just to clarify that, outside of that discussion, all results from our jump diffusion models were using a fixed λ setup which was updated daily.

4.4 Option Pricing

The full set of SPX options were priced under Black-Scholes-Merton, Merton's log-normal jump diffusion, and Kou's double exponential jump diffusion models.

4.5 Analysis of Results

This project looks to study what the typical lead/lag response time is, in minutes, for an SPX option responding to an SPY jump and also how that varies among jump magnitudes, trading volume, and option types. The lead lag time is something that can be directly calculated after every option was priced at each minute timestamp. For each detected jump time we obtain a market option time series and a model option time series by taking the subset of price observations that are on the minute time interval $[T_{jump} - 60, T_{jump} + 60]$, where it is recognized that the minimum time of 9:30 and maximum of 16:15 cannot be exceeded. The lead lag time is then given by the lag which maximizes the cross correlation between between the market / modeled option time series.

The quality of the jump diffusion model fit is assessed with regard to how well each model matched the "equilibrium price" of the market option price following a jump. The definition of equilibrium priced used is either the price at the end of the current jump window $T_{jump(i)} + 60$ or the price including lag of the following jump $T_{jump(i+1)} - 60 + lag$, whichever is sooner.

5 Results

5.1 Jump Detection

The monthly number of jumps detected using the methodology outlined by Lee and Mykland (2008) on minute resolution SPY returns from 2010-2012 is given by Figure 1 below along with some jump distribution characteristics in Figure 2. In this study, we see that our jump intensity was significantly higher in 2010 compared to the other two years with 239/518 detected jumps occurring in that year.

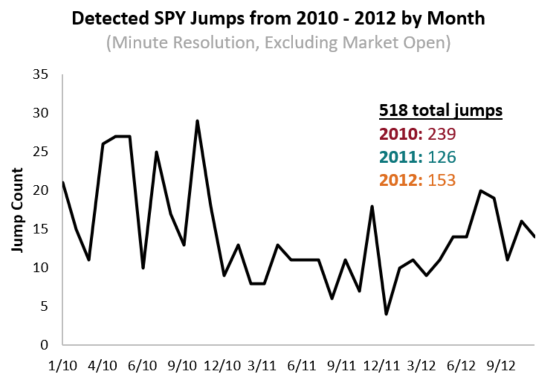
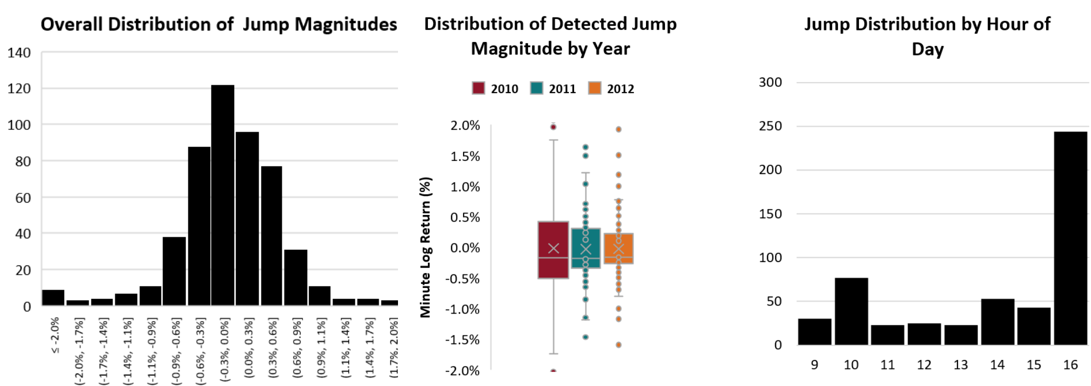


Figure 1: figure caption

This time period is also slightly skewed toward downward pricing jumps, as seen in Figure 2a, with 54.4% of jumps being negative. It was also found that the distribution of Jump magnitudes generally tightened year on year from 2010 - 2012, as shown in Figure 2b. As mentioned in Section 3, the opening log return was removed from the analysis. Before discarding, over 500 opening log returns would have been characterized as jumps. We also note that the majority of detected jumps occur near the end of the trading day, as in Figure 2c.



(a) (b) (c)

Figure 2: Distribution of Detected SPY ETF Price Jumps: (2a), (2b) and (2c)

5.2 Model Calibration

An initial test was performed to ensure that our Python implementation of the MLE jump diffusion calibration technique described by Hanson and Zhu (2004) was fit for use. Our test was to try and replicate the result of Hanson and Zhu's study, by calibrating a log normal and double exponential jump diffusion model to daily SPY returns from 1992-2001. The results of this trial are given in Table 1 below. In this implementation, we are solving for μj, σj, λ in the log-normal jump diffusion model and $\mu 1$, $\mu 2$, $p1$ λ in the double exponential jump diffusion model. Generally we find strong agreement between the calibrated coefficients from our Python implementation and Hanson and Zhu's Matlab results.

Table 1: Comparison of Project vs Hanson & Zhu (HZ) jump diffusion parameters

Model	μd	σd	μj	σj	λ
HZ MJD	0.191	0.088	-7.09E-04	1.19E-2	121
Project MJD	0.191	0.087	-6.92E-04	1.18E-2	123
HZ DEJD	0.17	0.085	-3.21E-04	9.40E-3	202
Project DEJD	0.17	0.084	-3.19E-04	9.32E-3	205

One of the extensions of our overall work is the implementation of these methods to higher frequency data. Our initial approach to obtaining daily MJD and DEJD jump diffusion parameters was to solve for the μj, σj, λ (MJD) and $\mu 1$, $\mu 2$, $p1$ λ (DEJD) which maximized the log likelihood between our jump diffusion approximated distribution function and a sample of minute log returns. While intuitive, this approach was found to be unstable as shown in Figure 3, which looks at 1Q2010 daily calibrations. We found that the optimum jump intensity λ varied widely day by day. Recall that with minute resolution data for the duration the CBOE is open for trading SPX options, we're looking at 405 log returns each day or about 100,000 yearly returns. A few cases of optimum lambda exceeded even the number of observations, which is an unacceptable result.

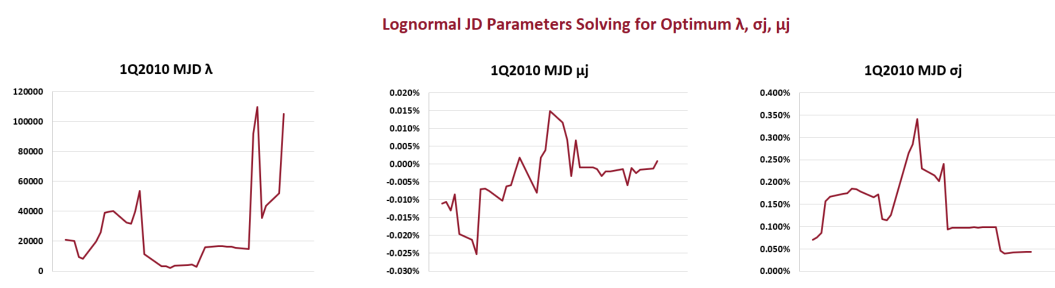


Figure 3: 1Q2010 Optimum MJD Model Parameters via MLE

In order to stabilize the daily calibration, we decided to remove λ from the maximum log likeliness optimization. Because this study was also implementing a jump detection methodology, it was possible to fix lambda to the annualized jump intensity seen detection results of the leading 14 day window of log returns used in the model calibration. The updated 1Q2010 model parameters after making this change are shown in Figure 4 below. This change helped enable consistent model predictions, and is the calibration approach used in the remainder of the paper.

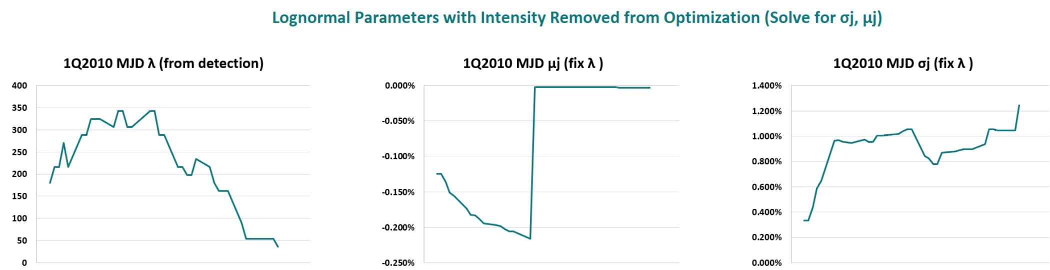


Figure 4: 1Q2010 Optimum MJD Model Parameters via MLE fix λ

While fixing λ in the MLE calibration was a solution to an ill-posed optimization problem, it begs the question with as to what might be causing this behavior. To investigate the problem further, a trial was performed by fitting several fixed λ log-normal jump diffusion models to January 2011 minute SPY returns and comparing the distributions of these models against GBM and the market data. The graphical results of this trial are given in Figure 5 below. A prominent characteristic of the market returns is its leptokurtic peak, slim shoulders and heavy tails compared to the normal distribution. It was seen that even a low intensity jump diffusion model was an improvement over GBM, but higher leptokurtic returns imply a higher λ, which was causing the behavior seen in Figure 3.

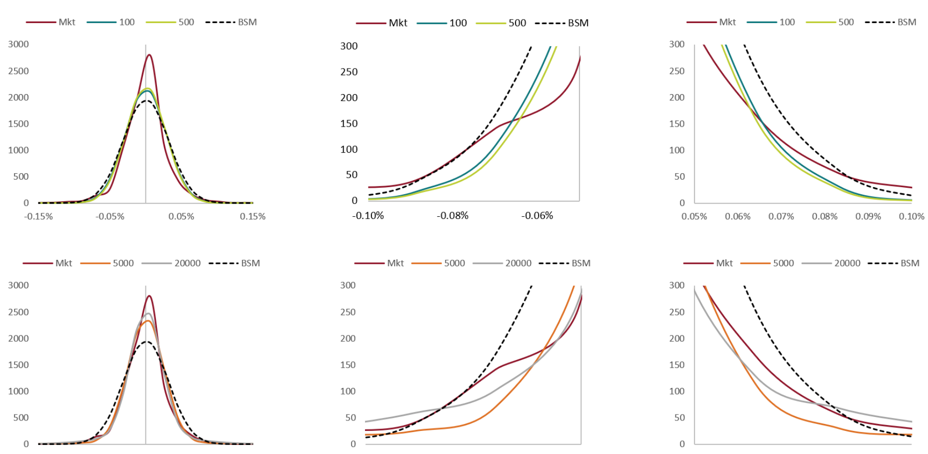


Figure 5: Log-Return PDF for MJD Models of Varying λ (January 2011 Data)

The daily expected jump magnitude and associated standard deviation are given in below Figures 6 and 7 respectively. Generally strong agreement was seen optimum model parameters between the two jump diffusion models in this study. This is not a wholly unexpected result, as Hanson and Zhu (2004) point out both MJD and DEJD have a small probability of producing higher magnitude jumps, which leads to similar tail behavior in the distribution of overall model log-returns.

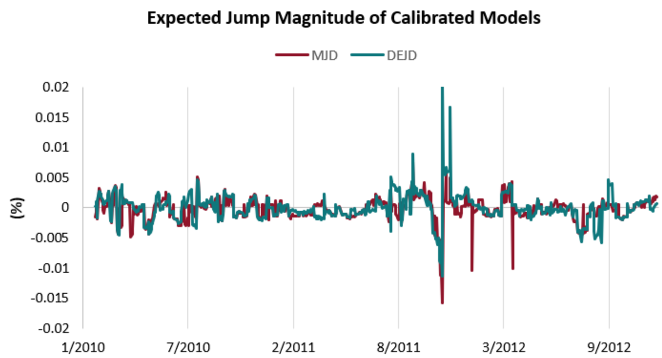


Figure 6: Comparison of Calibrated μj Between MJD and DEJD Models

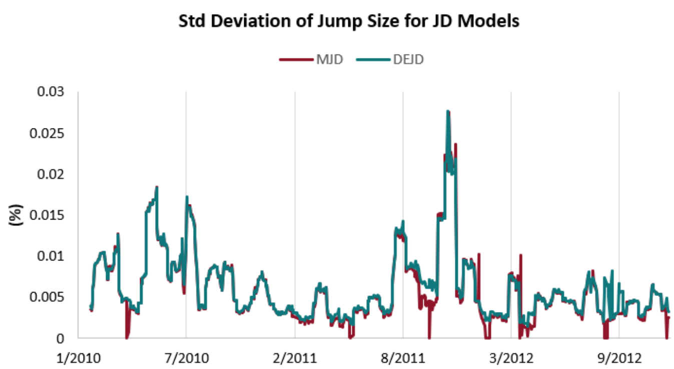


Figure 7: Comparison of Calibrated σj Between MJD and DEJD Models

It's worth pointing out to those wishing to duplicate these results some of the challenges in implementing in Python. Our code relied on the minimize function from the scipy.optimize package and used the Nelder-Mead optimization algorithm, as did Hanson and Zhu. It was found that this algorithm was far superior to some of the other standard optimization algorithms (SLSQP, Powell) though a limitation was that constraint equations were not supported in Scipy's Nelder-Mead function. This presented a real challenge especially in the DEJD calibration whose numerical PDF had exponential terms that were susceptible to overflow limitations depending on choice of $\mu 1$ and $\mu 2$ and also a desired limit that the calculated σ_d^2 by matching market returns be a positive number.

5.3 Option Pricing

5.3.1 Pricing Error Comparison by Option Moneyness

Our calibrated jump diffusion models were to price a set of 6 SPX options each month at a minute resolution. The log normal and double exponential jump diffusion models are benchmarked against BSM to see how they perform in matching the price of the option after a detected SPY pricing jump has occurred. Table 2 provides the average pricing error and Table 3 provides the standard deviation of this error for out-of-the-money, at-the-money, and in-the-money options. In Table 2, we can see that the average error between the models were generally comparable to one another.

The Merton Jump Diffusion Model and the Black Scholes Model performed similarly to one another across all levels of option moneyness, being nearly identical for out-of-the-money options. The DEJD model had mixed results compared to BSM and MJD, outperforming the other models in terms of average error for at-the-money options but under performing for out-of-the-money options. All models were found to have the largest error in pricing in-the-money options.

Table 2: Average Error by Option Moneyness

	MJD	BS	DEJD
OTM	-0.4	-0.4	-2.0
ATM	1.1	1.5	0.0
ITM	3.4	3.0	3.3

Table 3: Error Standard Deviation by Option Moneyness

	MJD	BS	DEJD
OTM	5.6	5.6	5.1
ATM	8.6	8.4	8.7
ITM	8.0	7.9	8.1

5.3.2 Put-Call Comparison

Figure 8 compares the pricing error of each model vs the strike price of the option, separating by call or put options. All three models tended to overestimate call options and underestimate the value of put options. The DEJD model was seen to outperform MJD and BSM in pricing call options more accurately, though it exhibited the largest error in pricing put options. This relationship follows from the put-call parity, but it was interesting to see that these relative pricing errors between models held across the full range of strike prices used. It can also be seen that the pricing errors for each model decrease with increasing strike price for call options and decrease with decreasing strike price for put options which is a consequence described above, where the magnitude of pricing errors is lowest for out of the money options.

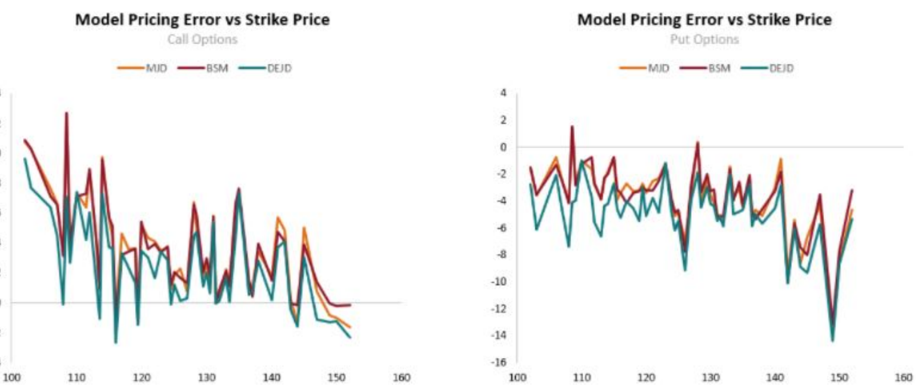


Figure 8: Call / Put Option Comparison

5.3.3 SPX Option Lag Response to SPY Jumps

Using cross correlation to compare the our modeled option time series against market option prices around the time of a jump, we find that this lead/lag response time typically occurs within one minute. We find that 97.9% of the time there was no identified lag between the model and market time series at this resolution. The distribution of non zero lag times is given in Figure 9. When lag was apparent, 92% of the time the SPX price changed after the SPY price did, meaning the SPY price movements effect SPX options more than vice versa. It can see from the distribution below that the lag time is skewed left, showing that the lag was typically a delayed in change in the SPX price with respect to the SPY.

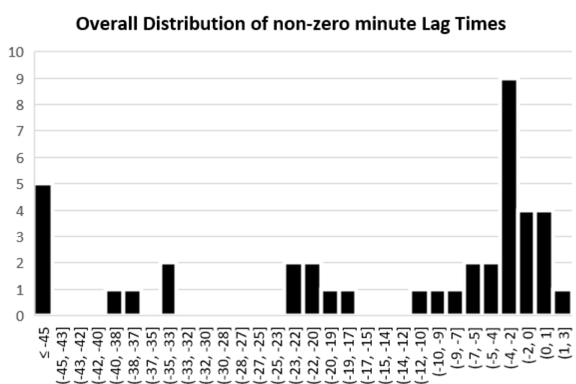


Figure 9: Non-Zero Lag Distribution

6 Conclusions and Further Study

This study finds that Merton's log-normal jump diffusion model and Kou's double-exponential jump diffusion model are improvements compared to Geometric Brownian motion in matching the distribution of market SPY ETF minute resolution log returns. However, the expected adjustments to European SPX option prices after calibrating these jump diffusion models to match the underlying's returns are not as clearly seen in market option data. We find that the majority ($\geq 97\%$) of lead/lag response times between SPY jumps and SPX option price reactions occur within the 1 minute resolution used in this study. Similar to prior work calibrating jump diffusion models to match option prices, we find that the daily calibration of JD models to match the underlying asset's return distribution leads to unstable parameter estimates. This stability can be improved by coupling the max likelihood estimation with an independent jump detection technique though the fact that the jump intensity λ needed to match match the distribution of returns differs from the number of actual jumps detected highlights an incompleteness with pure jump diffusion models.

Increasing the resolution of pricing observations to high frequency ($\leq$ 1 min) such that the lead / lag response between jumps can be more fully characterized is a meaningful extension of this work that can be taken up in a later study. Given more time, we would have expanded the scope of jump diffusion models studied to include self exciting processes such as the Hawkes process which can model jump clustering events and also a log-uniform jump diffusion model which exhibits stronger tail behavior than MJD or DEJD. Further work can be done on conditioning the MLE calibration problem to yield more consistent parameter estimations. The fixed 14-day leading window of log-returns used in the calibration is something that could be optimized and the incorporation of a dissimilarity penalty to prior solutions, such as the relative entropy term proposed by Cont and Tankov (2004), would be notable improvements to this setup.

Our python code for jump detection, calibration, and option pricing can be found in Appendices A, B, and C respectively.

References

Aldous, D. (1989). The heuristic. In *Probability Approximations via the Poisson Clumping Heuristic*, pp. 1–22. Springer.

Bakshi, G. and D. Madan (2000). Spanning and derivative-security valuation. *Journal of financial economics 55*(2), 205–238.

Barndorff-Nielsen, O. E. and N. Shephard (2004). Power and bipower variation with stochastic volatility and jumps. *Journal of financial econometrics 2*(1), 1–37.

Barndorff-Nielsen, O. E. and N. Shephard (2006). Econometrics of testing for jumps in financial economics using bipower variation. *Journal of financial Econometrics 4*(1), 1–30.

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

Carr, P. and D. Madan (1999). Option valuation using the fast fourier transform. *Journal of computational finance 2*(4), 61–73.

Carr, P. and L. Wu (2003). The finite moment log stable process and option pricing. *The journal of finance 58*(2), 753–777.

Cont, R. and P. Tankov (2004). Nonparametric calibration of jump-diffusion option pricing models. *The Journal of Computational Finance 7*, 1–49.

Galambos, J. (1978). The asymptotic theory of extreme order statistics. Technical report.

Hanson, F. B. and Z. Zhu (2004). Comparison of market parameters for jump-diffusion distributions using multinomial maximum likelihood estimation. In *2004 43rd IEEE Conference on Decision and Control (CDC)(IEEE Cat. No. 04CH37601)*, Volume 4, pp. 3919–3924. IEEE.

Hilpisch, Y. (2014). Derivatives analytics with python.

Jiang, G. J. and R. Oomen (2005). A new test for jumps in asset prices. *Preprint*.

Kou, S. G. (2002). A jump-diffusion model for option pricing. *Management science 48*(8), 1086–1101.

Kou, S. G. and H. Wang (2004). Option pricing under a double exponential jump diffusion model. *Management science 50*(9), 1178–1192.

Lee, S. S. and P. A. Mykland (2008). Jumps in financial markets: A new nonparametric test and jump dynamics. *The Review of Financial Studies 21*(6), 2535–2563.

Lewis, A. (2000). Option valuation under stochastic volatility. Technical report, Finance Press.

Lewis, A. L. (2001). A simple option formula for general jump-diffusion and other exponential lévy processes. *Available at SSRN 282110*.

Merton, R. C. (1976). Option pricing when underlying stock returns are discontinuous. *Journal of financial economics 3*(1-2), 125–144.

Officer, R. R. (1972). The distribution of stock returns. *Journal of the american statistical association 67*(340), 807–812.

Scott, L. O. (1997). Pricing stock options in a jump-diffusion model with stochastic volatility and interest rates: Applications of fourier inversion methods. *Mathematical Finance 7*(4), 413–426.

A Jump Detection Python Code

```python
import numpy as np
import pandas as pd
import math

# freq = frequency of price observations in days
# tol = confidence level for jump detection
# price_df = Pandas Dataframe of price observations (TimeStamp, Volume, Price columns)


def jump_detection(freq, tol, price_df):
    K = math.ceil((252*freq)**(1/2))
    prices = price_df['Price']
    time_stamp = price_df['TimeStamp']
    volumes = price_df['Volume']

    # Compute log returns and bi power variation of returns
    log_diffs = np.log(prices / prices.shift(1))
    bi_pwr = np.abs(log_diffs * log_diffs.shift(1))

    # Compute estimated local volatility, using prior K price observations
    bi_pwr_vec = []
    for i in range(K, len(prices) + 1):
        temp = bi_pwr[(i-K+2):i]
        local_bi_pwr = (np.sum(temp) / (K-2)) ** (1/2)
        bi_pwr_vec.append(local_bi_pwr)

    bi_pwr_vec = np.insert(bi_pwr_vec, 0, np.zeros(K-1))

    # Define constants for Mykland test statistic
    n = len(prices)

    c = (2 / math.pi) ** (1/2)

    c_n = ((2*math.log(n))**(1/2)) / c - \
            (math.log(math.pi) + math.log(math.log(n)))/2/c/((2*math.log(n))**(1/2))

    s_n = 1/c/((2*math.log(n))**(1/2))

    l_threshold = -math.log(-math.log(1-tol))

    # Use ratio of log return : local volatility for each timestep to detect jumps by comparing test statistic to
    # the maximum expected value to arise from diffusion alone, at a specificed confidence level.
    l_statistic = []
    for i in range(K, len(prices)):
        temp = (np.abs(log_diffs[i] / bi_pwr_vec[i-1]) - c_n)/s_n
        l_statistic.append(temp)

    l_statistic = np.insert(l_statistic, 0, np.zeros(K))

    jump_detected = np.where(l_statistic > l_threshold, np.sign(log_diffs), 0)

    df = pd.DataFrame({'Time Stamp': time_stamp, 'Asset Price': prices,
                       'Volume': volumes, 'Log Returns': log_diffs, 'Jumps': jump_detected, 'Local vol': bi_pwr_vec})

    return df


if __name__ == "__main__":
    price_df = pd.read_csv('C:/Users/Lori/Grad School/FE800 Project Data and Outputs/Clean_ETF_Data.csv')
    freq = 24*60
    tol = 0.01
    jump_df = jump_detection(freq=24*60, tol=0.01, price_df=price_df)
    print(jump_df)

    jump_df.to_csv('C:/Users/Lori/Grad School/FE800 Project Data and Outputs/ETF_jumps.csv', index=False)
```

B Jump Calibration Python Code

B.1 Jump Diffusion CDF and MMLE Functions

```python
import scipy.stats as sps
import numpy as np
import math


# Define a function for computing the distribution function for a Merton Jump Diffusion model
def log_normal_pdf(u_j, sigma_j, intensity, x1, x2, d_t, u_ld, sigma_d):

    n_0 = math.exp(-intensity*d_t)*(intensity*d_t)**0 / math.factorial(0) * \
          (sps.norm.cdf(x=x2, loc=(u_ld * d_t + 0 * u_j), scale=((d_t * sigma_d ** 2 + 0 * sigma_j ** 2)**(1 / 2))) -
           sps.norm.cdf(x=x1, loc=(u_ld * d_t + 0 * u_j), scale=((d_t * sigma_d ** 2 + 0 * sigma_j ** 2)**(1 / 2))))

    n_1 = math.exp(-intensity * d_t) * (intensity * d_t) ** 1 / math.factorial(1) * \
          (sps.norm.cdf(x=x2, loc=(u_ld * d_t + 1 * u_j), scale=((d_t * sigma_d ** 2 + 1 * sigma_j ** 2) ** (1 / 2))) -
           sps.norm.cdf(x=x1, loc=(u_ld * d_t + 1 * u_j), scale=((d_t * sigma_d ** 2 + 1 * sigma_j ** 2) ** (1 / 2))))

    n_2 = math.exp(-intensity * d_t) * (intensity * d_t) ** 2 / math.factorial(2) * \
          (sps.norm.cdf(x=x2, loc=(u_ld * d_t + 2 * u_j), scale=((d_t * sigma_d ** 2 + 2 * sigma_j ** 2) ** (1 / 2))) -
           sps.norm.cdf(x=x1, loc=(u_ld * d_t + 2 * u_j), scale=((d_t * sigma_d ** 2 + 2 * sigma_j ** 2) ** (1 / 2))))

    denom = math.exp(-intensity * d_t) * (intensity * d_t) ** 0 / math.factorial(0) + \
            math.exp(-intensity * d_t) * (intensity * d_t) ** 1 / math.factorial(1) + \
            math.exp(-intensity * d_t) * (intensity * d_t) ** 2 / math.factorial(2)

    output = (n_0 + n_1 + n_2)/denom
    return output


# Define a function for computing the distribution function for a Kou Double Exponential Jump Diffusion model
def double_exp_pdf(u1, u2, p1, intensity, x1, x2, d_t, u_ld, sigma_d):
    p2 = 1-p1
    u = u_ld * d_t
    sigma = math.sqrt(sigma_d**2 * d_t)

    v1 = u - 0.5 * sigma**2 / u1
    v2 = u + 0.5 * sigma**2 / u2

    p11 = (p1 / u1) ** 2
    p22 = (p2 / u2) ** 2
    p12 = 2 * p1 * p2 / (u1 + u2)

    z1 = (x1 - u) / sigma
    z2 = (x2 - u) / sigma

    # Error handling for exponential terms
    try:
        math.exp((x2 - v1) / u1)
    except OverflowError:
        u1 = p1/100
        v1 = 0.00001
        v2 = 0.00001

    try:
        math.exp((x1 - v1) / u1)
    except OverflowError:
        u1 = p1/100
        v1 = 0.00001
        v2 = 0.00001

    try:
        math.exp(-(x1 - v2) / u2)
    except OverflowError:
        print('overflow')
        u2 = p2/100
        v1 = 0.00001
        v2 = 0.00001

    try:
        math.exp(-(x2 - v2) / u2)
    except OverflowError:
        u2 = p2/100
        v1 = 0.00001
        v2 = 0.00001

    norm_diff = sps.norm.cdf(x=x2, loc=u, scale=sigma) - sps.norm.cdf(x=x1, loc=u, scale=sigma)
    px2_v1 = math.exp((x2 - v1) / u1) * sps.norm.cdf(x=-x2, loc=-u + sigma ** 2 / u1, scale=sigma)
    px1_v1 = math.exp((x1 - v1) / u1) * sps.norm.cdf(x=-x1, loc=-u + sigma ** 2 / u1, scale=sigma)
    px1_v2 = math.exp(-(x1 - v2) / u2) * sps.norm.cdf(x=x1, loc=u + sigma ** 2 / u2, scale=sigma)
```

```
    px2_v2 = math.exp(-(x2 - v2) / u2) * sps.norm.cdf(x=x2, loc=u + sigma ** 2 / u2, scale=sigma)

    n_0 = math.exp(-intensity * d_t) * (intensity * d_t) ** 0 / math.factorial(0) * norm_diff

    n_1 = math.exp(-intensity * d_t) * (intensity * d_t) ** 1 / math.factorial(1) * (
        norm_diff + p1 * (px2_v1 - px1_v1) + p2 * (px1_v2 - px2_v2))

    n_2 = math.exp(-intensity * d_t) * (intensity * d_t) ** 2 / math.factorial(2) * (
        norm_diff + u1*((p12+p11*(u-sigma**2/u1+u1-x2))*px2_v1 - (p12+p11*(u-sigma**2/u1+u1-x1))*px1_v1)+\
        u2*((p12-p22*(u+sigma**2/u2-u2-x1))*px1_v2 - (p12-p22*(u+sigma**2/u2-u2-x2))*px2_v2) +\
        sigma / math.sqrt(2*math.pi)*(u2*p22 - u1*p11)*(math.exp(-(z1**2 / 2)) - math.exp(-(z2**2 / 2))))

    denom = math.exp(-intensity * d_t) * (intensity * d_t) ** 0 / math.factorial(0) + \
            math.exp(-intensity * d_t) * (intensity * d_t) ** 1 / math.factorial(1) + \
            math.exp(-intensity * d_t) * (intensity * d_t) ** 2 / math.factorial(2)

    output = (n_0 + n_1 + n_2) / denom
    return output


# Define function for computing the log-likelihood of a return distribution matching an MJD model given a test set of
# JD model parameters.
def mjd_calibration(parameters, returns, d_t):
    # Split off input parameters from initial guess vector
    u_j = parameters[0]
    sigma_j = parameters[1]
    intensity = parameters[2]
    n = len(returns)

    # Variable Bounding
    if intensity > 20000:
        intensity = 20000

    # Calculate moments of sampled returns
    sam_mean = np.mean(returns)
    sam_sd = np.std(returns)

    # Match 1st and 2nd Moments of JD model to sampled returns
    u_ld = (sam_mean - u_j * intensity * d_t) / d_t
    sigma_d2 = max(((sam_sd ** 2 - (sigma_j ** 2 + u_j ** 2) * intensity * d_t) / d_t), 0.000001)
    sigma_d = sigma_d2 ** (1 / 2)

    # Print statements for troubleshooting during scipy.optimize execution
    print('')
    print('iteration:')

    print('u_j')
    print(u_j)

    print('sigma_j')
    print(sigma_j)

    print('lambda')
    print(intensity)
    print('')

    # Bin Returns into histogram and compute log likelihood across all bins
    ret_hist, ret_bins = np.histogram(returns, bins=100, density=False)
    log_like = 0
    for x in range(len(ret_hist)):
        x1 = ret_bins[x]
        x2 = ret_bins[x + 1]

        pdf_int = log_normal_pdf(u_j=u_j, sigma_j=sigma_j, intensity=intensity, x1=x1, x2=x2,
                                 d_t=d_t, u_ld=u_ld, sigma_d=sigma_d)

        if pdf_int == 0:
            bin_log_like = 0
        else:
            bin_log_like = math.log(pdf_int * n) * ret_hist[x]
        log_like = log_like + bin_log_like

    log_like = (log_like * -1) / n
    return log_like


# Define function for computing the log-likelihood of a return distribution matching an MJD model given a test set of
# JD model parameters.
def mjd_calibration_fix_intensity(parameters, returns, d_t, intensity):
    # Split off input parameters from initial guess vector
    u_j = parameters[0]
    sigma_j = parameters[1]
    n = len(returns)

    # Calculate moments of sampled returns
    sam_mean = np.mean(returns)
    sam_sd = np.std(returns)

    # Match 1st and 2nd Moments of JD model to sampled returns
    u_ld = (sam_mean - u_j * intensity * d_t) / d_t
```

```python
    sigma_d2 = max(((sam_sd ** 2 - (sigma_j ** 2 + u_j ** 2) * intensity * d_t) / d_t), 0.000001)
    sigma_d = sigma_d2 ** (1 / 2)

    # Bin Returns into histogram and compute log likelihood across all bins
    ret_hist, ret_bins = np.histogram(returns, bins=50, density=False)
    log_like = 0
    for x in range(len(ret_hist)):
        x1 = ret_bins[x]
        x2 = ret_bins[x + 1]

        pdf_int = log_normal_pdf(u_j=u_j, sigma_j=sigma_j, intensity=intensity, x1=x1, x2=x2,
                                 d_t=d_t, u_ld=u_ld, sigma_d=sigma_d)

        if pdf_int == 0:
            bin_log_like = 0
        else:
            bin_log_like = math.log(pdf_int * n) * ret_hist[x]
        log_like = log_like + bin_log_like

    log_like = (log_like * -1) / n
    return log_like


# Define function for computing the log-likelihood of a return distribution matching a DEJD model given a test set of
# JD model parameters.
def dejd_calibration(parameters, returns, d_t):

    # Split off input parameters from initial guess vector
    u1 = parameters[0]
    u2 = parameters[1]
    p1 = parameters[2]
    intensity = parameters[3]
    n = len(returns)

    # Variable Bounding
    if u1 < 0:
        u1 = 0.0001
    if u2 < 0:
        u2 = 0.0001
    if p1 > 1:
        p1 = 1
    if p1 < 0:
        p1 = 0
    if intensity > 20000:
        intensity = 20000

    # Calculate moments of sampled returns
    sam_mean = np.mean(returns)
    sam_sd = np.std(returns)

    # Calculate expected Jump size and standard deviation given DEJD parameters
    u_j = -p1*u1 + (1-p1)*u2
    sigma_j = math.sqrt(p1*((u_j + u1)**2 + u1**2) + (1-p1)*((u_j-u2)**2 + u2**2))

    # Match 1st and 2nd Moments of JD model to sampled returns
    u_ld = (sam_mean - u_j * intensity * d_t) / d_t
    sigma_d2 = max(((sam_sd ** 2 - (sigma_j ** 2 + u_j ** 2) * intensity * d_t) / d_t), 0.000001)
    sigma_d = sigma_d2 ** (1 / 2)

    # Print statements for troubleshooting during scipy.optimize execution
    print('')
    print('iteration:')

    print('u_j')
    print(u_j)

    print('sigma_j')
    print(sigma_j)

    print('lambda')
    print(intensity)
    print('')

    # Bin returns into histogram and compute log likelihood across all bins
    ret_hist, ret_bins = np.histogram(returns, bins=100, density=False)
    log_like = 0
    for x in range(len(ret_hist)):
        x1 = ret_bins[x]
        x2 = ret_bins[x + 1]

        pdf_int = double_exp_pdf(u1=u1, u2=u2, p1=p1, intensity=intensity, x1=x1, x2=x2,
                                 d_t=d_t, u_ld=u_ld, sigma_d=sigma_d)
        if pdf_int <= 0:
            bin_log_like = 0
        else:
            bin_log_like = math.log(pdf_int * n) * ret_hist[x]
        log_like = log_like + bin_log_like

    log_like = (log_like * -1) / n
    return log_like

```

```python

def dejd_calibration_fix_intensity(parameters, returns, d_t, intensity):

    # Split off input parameters from initial guess vector
    u1 = parameters[0]
    u2 = parameters[1]
    p1 = parameters[2]
    n = len(returns)

    # Variable Bounding
    if u1 < 0:
        u1 = 0.0001
    if u2 < 0:
        u2 = 0.0001
    if p1 > 1:
        p1 = 1
    if p1 < 0:
        p1 = 0

    # Calculate moments of sampled returns
    sam_mean = np.mean(returns)
    sam_sd = np.std(returns)

    # Calculate expected Jump size and standard deviation given DEJD parameters
    u_j = -p1*u1 + (1-p1)*u2
    sigma_j = math.sqrt(p1*((u_j + u1)**2 + u1**2) + (1-p1)*((u_j-u2)**2 + u2**2))

    # Match 1st and 2nd Moments of JD model to sampled returns
    u_ld = (sam_mean - u_j * intensity * d_t) / d_t
    sigma_d2 = max(((sam_sd ** 2 - (sigma_j ** 2 + u_j ** 2) * intensity * d_t) / d_t), 0.000001)
    sigma_d = sigma_d2 ** (1 / 2)

    # Bin returns into histogram and compute log likelihood across all bins
    ret_hist, ret_bins = np.histogram(returns, bins=50, density=False)
    log_like = 0
    for x in range(len(ret_hist)):
        x1 = ret_bins[x]
        x2 = ret_bins[x + 1]

        pdf_int = double_exp_pdf(u1=u1, u2=u2, p1=p1, intensity=intensity, x1=x1, x2=x2,
                                 d_t=d_t, u_ld=u_ld, sigma_d=sigma_d)
        if pdf_int <= 0:
            bin_log_like = 0
        else:
            bin_log_like = math.log(pdf_int * n) * ret_hist[x]
        log_like = log_like + bin_log_like

    log_like = (log_like * -1) / n

    if sigma_d2 == 0.000001:
        #print('True')
        log_like = log_like + 10000

    return log_like


if __name__ == "__main__":
    # Function Testing
    test1 = log_normal_pdf(u_j=-0.0007, sigma_j=0.012, intensity=121, x1=0,
                           x2=0.01, d_t=1/252, u_ld=0.22113, sigma_d=0.32203)

    # Function Testing
    test2 = double_exp_pdf(u1=0.01, u2=0.01, p1=0.3, intensity=120, x1=0, x2=0.01, d_t=1/252,
                           u_ld=-0.3435711, sigma_d=0.31174753)

    print("MJD PDF Test Result:")
    print(test1)
    print('')
    print("DEJD PDF Test Result:")
    print(test2)
```

B.2 Calibration Implementation Script (Fixed λ)

```python
import pandas as pd
import numpy as np
import math
from Calibration.JumpCalibration import mjd_calibration_fix_intensity, dejd_calibration_fix_intensity
from scipy.optimize import minimize

# Read in csv file with minute SPY ETF price movements, and subset for price moves during the CBOE trading day
test = pd.read_csv('C:/Users/Lori/Grad School/FE800 Project Data and Outputs/ETF_jumps.csv')
test['Time Stamp'] = pd.to_datetime(test['Time Stamp'], format='%m/%d/%Y %H:%M')
days = test['Time Stamp'].dt.date.unique()
test = test.set_index(['Time Stamp'])
test = test.between_time('09:31:00', '16:15:00')

# Define initial guesses for parameter results, and initialize output vectors
mjd_guess = np.array([0.0, 0.012])
dejd_guess = np.array([0.005, 0.005, 0.5])
mjd_u_j = []
mjd_sigma_j = []
mjd_lam = []
mjd_uld = []
mjd_sigma_d = []

uj_bound = (-0.02, 0.02)
sigj_bound = (0, 0.03)
mjd_bnds = (uj_bound, sigj_bound)

dejd_u1 = []
dejd_u2 = []
dejd_p1 = []
dejd_lam = []
dejd_uld = []
dejd_sigma_d = []
date_vec = []

u1_bound = (0, 0.07)
u2_bound = (0, 0.07)
p1_bound = (0, 1)
dejd_bnds = (u1_bound, u2_bound, p1_bound)

d_t = 1/252/6.75/60

for i in range(14, 752):
    start = days[i-14]
    end = days[i]
    subset = test.loc[start:end]
    log_returns = subset['Log Returns']

    # Calculate moments of sampled returns
    sam_mean = np.mean(log_returns)
    sam_sd = np.std(log_returns)

    # Sum up the number of jumps for the prior 14 trading days, fix lambda to that annualized value
    jumps_year = np.sum(np.abs(subset['Jumps']))
    jumps_year = jumps_year * 252/14
    print('')
    print('Iteration:')
    print(i)
    print('Lambda:')
    print(jumps_year)

    # Return u_j and sigma_j which maximize the log likelihood of the 14-day return distribution
    mjd_params = minimize(fun=mjd_calibration_fix_intensity, x0=mjd_guess,
                          args=(log_returns, d_t, jumps_year), method='Nelder-Mead', bounds=mjd_bnds)
    mjd_u_j.append(mjd_params.x[0])
    mjd_sigma_j.append(mjd_params.x[1])
    mjd_lam.append(jumps_year)
    print(mjd_params.x)

    # Match 1st and 2nd Moments of MJD model to sampled returns, append diffusion parameters to results
    mjd_uld_value = (sam_mean - mjd_params.x[0] * jumps_year * d_t) / d_t
    sigma_d2 = max(((sam_sd ** 2 - (mjd_params.x[1] ** 2 + mjd_params.x[0] ** 2) * jumps_year * d_t) / d_t), 0.000001)
    mjd_sigma_d_value = sigma_d2 ** (1 / 2)

    mjd_uld.append(mjd_uld_value)
    mjd_sigma_d.append(mjd_sigma_d_value)

    arguments = (log_returns, d_t, jumps_year)

    # Return u_1, u_2, and p1 which maximize the log likelihood of the 14-day return distribution
    dejd_params = minimize(fun=dejd_calibration_fix_intensity, x0=dejd_guess,
                           args=arguments, method='Nelder-Mead', bounds=dejd_bnds)
    dejd_u1.append(dejd_params.x[0])
    dejd_u2.append(dejd_params.x[1])
    dejd_p1.append(dejd_params.x[2])
    dejd_lam.append(jumps_year)

    # Calculate expected Jump size and standard deviation given DEJD parameters
```

```python
    print(dejd_params.x)
    p1 = dejd_params.x[2]
    if dejd_params.x[2] < 0:
        p1 = 0
    if dejd_params.x[2] > 1:
        p1 = 1
    dejd_u_j = -p1 * dejd_params.x[0] + (1 - p1) * dejd_params.x[1]

    dejd_sigma_j = math.sqrt(p1 * ((dejd_u_j + dejd_params.x[0]) ** 2 + dejd_params.x[0] ** 2) +
                             (1 - p1) * ((dejd_u_j - dejd_params.x[1]) ** 2 + dejd_params.x[1] ** 2))

    # Match 1st and 2nd Moments of DEJD model to sampled returns, append diffusion parameters to results
    dejd_u_ld_value = (sam_mean - dejd_u_j * jumps_year * d_t) / d_t
    sigma_d2 = max(((sam_sd ** 2 - (dejd_sigma_j ** 2 + dejd_u_j ** 2) * jumps_year * d_t) / d_t), 0.000001)
    dejd_sigma_d_value = sigma_d2 ** (1 / 2)

    dejd_uld.append(dejd_u_ld_value)
    dejd_sigma_d.append(dejd_sigma_d_value)

    date_vec.append(end)

# Combine All coefficient vectors to one consolidated DataFrame and output to a csv.
param_df = pd.DataFrame({'Date': date_vec, 'MJD uj': mjd_u_j, 'MJD sigma j': mjd_sigma_j, 'MJD Lambda': mjd_lam,
                         'MJD uld': mjd_uld, 'MJD sigma d': mjd_sigma_d, 'DEJD u1': dejd_u1, 'DEJD u2': dejd_u2,
                         'DEJD p1': dejd_p1, 'DEJD Lambda': dejd_lam, 'DEJD uld': dejd_uld, 'DEJD sigma d': dejd_sigma_d})

param_df.to_csv('C:/Users/Lori/Grad School/FE800 Project Data and Outputs/fix_param_upd_df1.csv', index=False)
print(param_df)
```

C Option Pricing and Results Workup

C.1 MJD and DEJD Pricing Functions

Below script adopted from Hilpisch (2014)

```python
# Valuation of European Call Options
# in Merton's (1976) Jump Diffusion Model
# via Numerical Integration
# 08_m76/M76_valuation_INT.py
#
# (c) Dr. Yves J. Hilpisch
# Derivatives Analytics with Python
#
#
# Valuation by Integration
#
import math
import numpy as np
from scipy.integrate import quad

def M76_integration_function(u, S0, K, T, r, sigma, lamb, mu, delta):
    ''' Valuation of European call option in M76 model via
    Lewis (2001) Fourier-based approach: integration function.
    Parameter definitions see function M76_value_call_INT. '''
    JDCF = M76_characteristic_function(u - 0.5 * 1j, T, r, sigma, lamb, mu, delta)
    value = 1 / (u ** 2 + 0.25) * (np.exp(1j * u * math.log(S0 / K)) * JDCF).real
    return value


def M76_characteristic_function(u, T, r, sigma, lamb, mu, delta):
    ''' Valuation of European call option in M76 model via
    Lewis (2001) Fourier-based approach: characteristic function.
    Parameter definitions see function M76_value_call_INT. '''
    omega = r - 0.5 * sigma ** 2 - lamb * (np.exp(mu + 0.5 * delta ** 2) - 1)
    value = np.exp((1j * u * omega - 0.5 * u ** 2 * sigma ** 2 +
                    lamb * (np.exp(1j * u * mu - u ** 2 * delta ** 2 * 0.5) - 1)) * T)
    return value


def M76_value(S0, K, T, r, sigma, lamb, mu, delta, type):
    ''' Valuation of European call option in M76 model via
    Lewis (2001) Fourier-based approach.
    Parameters
    ==========
    S0: float
    initial stock/index level
    K: float
    strike price
    T: float
    time-to-maturity (for t=0)
    r: float
    constant risk-free short rate
    sigma: float
    volatility factor in diffusion term
    lamb: float
    jump intensity
    mu: float
    expected jump size
    delta: float
    standard deviation of jump
    Returns
    =======
    call_value: float
    European call option present value '''

    int_value = quad(lambda u: M76_integration_function(u, S0, K, T, r,
                                                       sigma, lamb, mu, delta), 0, 50, limit=250)[0]
    call_value = S0 - np.exp(-r * T) * math.sqrt(S0 * K) / math.pi * int_value
    put_value = call_value - S0 + K * np.exp(-r*T)

    if type == 'Call':
        output = call_value
    else:
        output = put_value
    return output


def DEJD_integration_function(u, S0, K, T, r, sigma, lamb, kappa, eta):
    JDCF = DEJD_characteristic_function(u - 0.5 * 1j, T, r, sigma, lamb, kappa, eta)
    value = 1 / (u ** 2 + 0.25) * (np.exp(1j * u * math.log(S0 / K)) * JDCF).real
    return value


def DEJD_characteristic_function(u, T, r, sigma, lamb, kappa, eta):
    omega = r - 0.5 * sigma ** 2 - lamb * (np.exp(kappa) - 1)
    value = np.exp((1j * u * omega - 0.5 * u ** 2 * sigma ** 2 +
                    lamb * (np.exp(1j * u * kappa) * (1-eta**2) / (1 + u**2 * eta**2) - 1)) * T)
```

```
    return value


def DEJD_value(S0, K, T, r, sigma, lamb, kappa, eta, type):
    int_value = quad(lambda u: DEJD_integration_function(u, S0, K, T, r,
                                                        sigma, lamb, kappa, eta), 0, 50, limit=250)[0]
    call_value = S0 - np.exp(-r * T) * math.sqrt(S0 * K) / math.pi * int_value
    put_value = call_value - S0 + K * np.exp(-r * T)

    if type == 'Call':
        output = call_value
    else:
        output = put_value
    return output
```

C.2 Option Pricing Implementation

```python
import pandas as pd
from OptionPricing.Hilpisch_JD_Pricing import M76_value, DEJD_value
from OptionPricing.BSM import bsm_value

# Read in SPY ETF Time Series, Daily JD Calibrated Parameters, and daily 1M Treasury Bill Rate
spy_prices = pd.read_csv('C:/Users/Lori/Grad School/FE800 Project Data and Outputs/Clean_ETF_Data.csv')
calib_params = pd.read_csv('C:/Users/Lori/Grad School/FE800 Project Data and Outputs/fix_param_upd_df1.csv')
rf_rates = pd.read_csv('C:/Users/Lori/Grad School/FE800 Project Data and Outputs/1M_TBill_Rate.csv')

spy_prices.rename(columns={'Price': 'SPY Price', 'Volume': 'SPY Volume'}, inplace=True)

# Read in SPY, Calibrated Parameters, and RF Rate Data. Set Time as index
spy_prices['TimeStamp'] = pd.to_datetime(spy_prices['TimeStamp'], format='%m/%d/%Y %H:%M')
calib_params['Date'] = pd.to_datetime(calib_params['Date'], format='%Y-%m-%d')
rf_rates['Date'] = pd.to_datetime(rf_rates['Date'], format='%m/%d/%Y')
spy_prices = spy_prices.set_index('TimeStamp')
calib_params = calib_params.set_index('Date')
rf_rates = rf_rates.set_index('Date')

# Combine Daily calibrated JD parameters and 1M Treasure Rates for Later use
model_inputs = calib_params.join(rf_rates, how='left')


# Price each option under BSM, MJD, DEJD
for year in [2010, 2011, 2012]:
    for month in range(1, 13):
        year_month = str(year) + '_' + str(month)
        print(year_month)

        input_path = 'C:/Users/Lori/Grad School/FE800 Project Data and Outputs/option_data/' + year_month + '.csv'
        output_path = 'C:/Users/Lori/Grad School/FE800 Project Data and Outputs/final_priced_options/' + year_month + '.csv'

        # Read in option data, convert timestamp to DateTime and strike price to $.
        option_df = pd.read_csv(input_path)
        option_df['DateTime'] = pd.to_datetime(option_df['DateTime'])
        option_df['Date'] = option_df['DateTime'].dt.date
        option_df['Strike'] = option_df['Strike']/100
        option_df = option_df[option_df['DateTime'] > '1-24-2010']

        # Join Daily JD Parameters, RF Rates with minute SPY ETF and SPX Option Data
        option_df = option_df.set_index('DateTime')
        option_df = option_df.join(spy_prices, how='left', sort=False)
        option_df.index.name = 'DateTime'
        option_df = option_df.sort_values(['Type', 'Strike', 'DateTime'], ascending=[True, True, True])

        option_df = option_df.reset_index()
        option_df = option_df.set_index('Date')

        option_df = option_df.join(model_inputs, how='left', sort=False)
        option_df = option_df.sort_values(['Type', 'Strike', 'DateTime'], ascending=[True, True, True])

        option_df['DEJD kappa'] = option_df['DEJD p1'] * option_df['DEJD u1'] * -1 + (1-option_df['DEJD p1']) * option_df['DEJD u2']

        option_df['DEJD eta'] = option_df['DEJD p1'] * ((option_df['DEJD kappa'] + option_df['DEJD u1'])**2 +
                                                        option_df['DEJD u1'] ** 2) + (1-option_df['DEJD p1']) * \
                                    ((option_df['DEJD kappa'] - option_df['DEJD u2']) ** 2 + option_df['DEJD u2'] ** 2)

        option_df['BSM sigma'] = (option_df['MJD sigma d'] ** 2 + option_df['MJD Lambda'] * \
                                  (option_df['MJD sigma j'] ** 2 + option_df['MJD uj'] ** 2)) ** (1/2)

        print('start BSM')
        option_df['BSMvalue'] = option_df.apply(lambda row: bsm_value(S0=row['SPY Price'], K=row['Strike'], T=row['TTM'],
                                                                      r=row['rf_rate'], sigma=row['BSM sigma'], type=row['Type']),
                                                axis=1)

        print('start MJD')
        option_df['MJDvalue'] = option_df.apply(lambda row: M76_value(S0=row['SPY Price'], K=row['Strike'], T=row['TTM'],
                                                                      r=row['rf_rate'], sigma=row['MJD sigma d'],
                                                                      lamb=row['MJD Lambda'], mu=row['MJD uj'],
                                                                      delta=row['MJD sigma j'], type=row['Type']), axis=1)

        print('Start DEJD')
        option_df['DEJDvalue'] = option_df.apply(lambda row: DEJD_value(S0=row['SPY Price'], K=row['Strike'], T=row['TTM'],
                                                                        r=row['rf_rate'], sigma=row['DEJD sigma d'],
```

```
                                                                    lamb=row['DEJD Lambda'], kappa=row['DEJD kappa'],
                                                                    eta=row['DEJD eta'], type=row['Type']), axis=1)
        print('')
        # Write Final DataFrame of priced monthly options to a csv
        if year_month != '2012_12':
            option_df.to_csv(output_path)
        else:
            print('Completed')
```