A Fast and Robust Fourier Option Pricing Method: Defining a Truncation Range Formula Explicitly for Various Advanced Stock Price Models

Tom Haynes - UTS - Master of Quantitative Finance

January 2023

Contents

Software

- `Python 3.10.0` [1]: Implementation, research and testing conduction in Python using Jupyter Notebooks.
- `numpy 1.22.3` [2]: Matrix algebra, vectorised code in the COS and COS Speedy methods, other mathematical operations and constants.
- `time` [3]: Speedtesting different approaches.
- `warnings` [4]: Suppression of numpy complex number deprecation warning.
- `scipy 1.8.0` [5]: Various distributions of random variables, numerical integration via quadrature, other mathematical operations.
- `math` [6]: Factorial function.
- `matplotlib 3.5.3` [7]: Plotting results.
- `Wolfram Mathematica Student Edition 13.1` [8]: Analytic computation of partial derivatives.

---

1 Introduction

Being able to price options and multiple strikes at high speed is known to be a very important facet of computational finance:

> *"...stock price models are typically calibrated to gives prices of liquid call and put options by minimizing the mean-square-error between model prices and given market prices. During the optimization routine, model prices of call and put options need to be evaluated very often for different model parameters."[9]*

The COS Method is a Fourier option pricing technique that allows for options to be priced extremely quickly. It is in fact the fastest known Fourier transform technique. It was first documented in Fang and Osterlee's paper; "A Novel Pricing Method for European Options Based on Fourier-Cosine Series Expansions"[10], Fang's Phd thesis.

There are some outstanding issues with this method pertaining mainly to **divergence in some advanced stock price models, namely Lévy processes with jumps, and major option pricing errors in extreme moneyness options.** Here I examine these issues and implement solutions I have found in my research. The aim is to formulate a more robust COS-method for pricing stock options at a very wide range of strikes across numerous advanced stock price models whilst attempting to maintain the computational efficiency of the COS method.

1.1 The COS Method

The problem the COS method solves is fairly straightforward. Starting with the basic idea, we are trying to solve the following integral:

$$\int_{\mathbb{R}} v(x)f(x)dx \tag{1}$$

Given the following conditions:

- The stochastic component (underlying price) is modelled by a random variable X where in which the probability density function f is unknown.
- But, we do have the characteristic function of f explicitly, denoted φ.
- We have an option payoff function $v(X)$.

The COS method details how we can use the COS Fourier expansion to estimate this f from φ. Let $N \in \mathbb{N}$, we approximate $f(y)$. Let:

$$\bar{F}_k := \frac{2}{b-a} \operatorname{Re}\left\{\phi_X\left(\frac{k\pi}{b-a}\right) \cdot \exp\left(-i\frac{ka\pi}{b-a}\right)\right\} \tag{2}$$

Then we approximate $f_x(y)$ by:

$$\hat{f}_X(y) \approx \sum_{k=0}^{N-1} \bar{F}_k \cos\left(k\pi\frac{y-a}{b-a}\right) \tag{3}$$

1.1.1 Exponential Convergence of the COS Method

The primary reason this is such a computationally efficient solution is via the exponential convergence of the Fourier COS expansion. Demonstrating this in figure 1 below using the characteristic function of the normal distribution:

$$X \sim \mathcal{N}(0,1), \quad f_X(y) = \frac{1}{\sqrt{2\pi}} \mathrm{e}^{-\frac{1}{2}y^2}, \quad \varphi_X(u) = \mathrm{e}^{-\frac{1}{2}u^2}, \tag{4}$$

Generally, with most distributions and sufficiently large N we get errors that are in the bounds of machine precision.

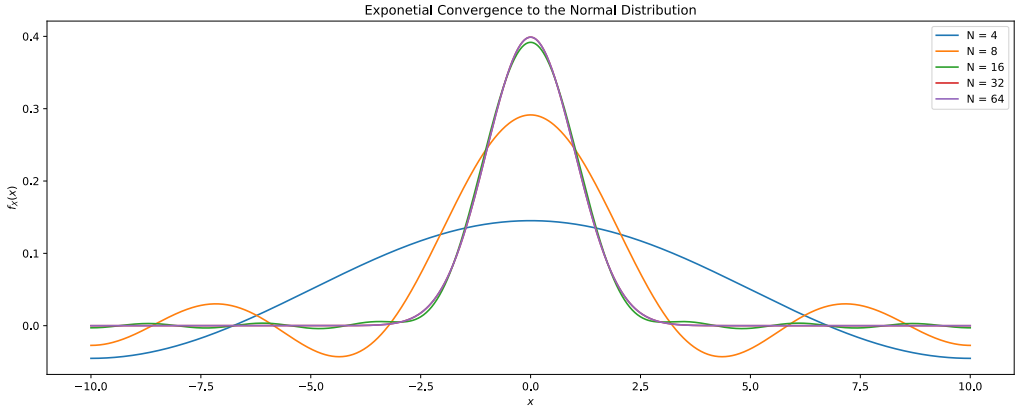

Figure 1: Exponential Convergence to the Normal Distribution

1.1.2 Pricing European Options with the COS Method

Implementation of 1.1: Fang and Osterlee detail how we can use this method to price European options. Below I revisit the highlights of the derivation. First, we start with replacing the European style options density function by the COS formula. We know density decay's to zero as $y \to \infty$, given this, it is reasonable to choose a truncation range $[a, b]$ on $\mathbb{R}$, this leads us the approximation of v_1.

$$v\left(x, t_0\right) = e^{-r\Delta t} \int_a^b v(y, T) f(y \mid x) dy \tag{5}$$

Now, from what we can use the Fourier-Cosine expansion to replace the conditional density function that is unknown.

$$f(y|x) = \sum_{k=0}^{\infty}{}' A_k(x) \cos\left(k\pi \frac{y-a}{b-a}\right) \tag{6}$$

$$A_k(x) = \frac{2}{b-a} \int_a^b f(y \mid x) \cos\left(k\pi \frac{y-a}{b-a}\right) dy \tag{7}$$

$$\text{then,} \quad v\left(x, t_0\right) = e^{-r\Delta t} \int_a^b v(y, T) \sum_{k=0}^{\infty}{}' A_k(x) \cos\left(k\pi \frac{y-a}{b-a}\right) dy \tag{8}$$

Now, define V_k as the following:

$$V_k = \frac{2}{b-a}\int_a^b v(y,T)\cos\left(k\pi\frac{y-a}{b-a}\right)dy \tag{9}$$

$$\text{then,}\quad v\left(x,t_0\right) = \frac{1}{2}(b-a)e^{-r\Delta t}\cdot{\sum_{k=0}^{\infty}}' A_k(x)V_k \tag{10}$$

$$= \frac{1}{2}(b-a)e^{-r\Delta t}\cdot{\sum_{k=0}^{N-1}}' A_k(x)V_k$$

$$= e^{-r\Delta t}{\sum_{k=0}^{N-1}}' \operatorname{Re}\left\{\phi\left(\frac{k\pi}{b-a};x\right)e^{-ik\pi\frac{a}{b-a}}\right\}V_k$$

Obtaining V_k in 10 for Vanilla Puts and Calls: Here V_k must be obtained analytically for European options in Heston and Lévy processes. In the original work, Fang and Osterlee detail the derivation for digital and gap option prices as well. With these two payoff functions we can derive the prices for a wide range of payoffs. *This is outside the scope of this paper.* So, given we know φ; the characteristic function of the natural logarithm of asset prices, denoted below as:

$$x = \ln\left(\frac{S_0}{K}\right),\quad y = \ln\left(\frac{S_T}{K}\right) \tag{11}$$

Then, for the payoff function for Vanilla options:

$$v(y,T) = \max\{\alpha\cdot K(e^y-1),\ 0\},\quad \alpha = \begin{cases}1 & \text{Call}\\ -1 & \text{Put}\end{cases} \tag{12}$$

Some calculus shows that,

$$V_k^{\text{Call}} = \frac{2}{b-a}\int_0^b K(e^y-1)\cos\left(\frac{y-a}{b-a}\right)dy \tag{13}$$

$$= \frac{2}{b-a}K(\chi_k(0,b)-\psi_k(0,b))$$

$$\text{similarly,}\quad V_k^{\text{Put}} = \frac{2}{b-a}K(\psi_k(a,0)-\chi_k(a,0)) \tag{14}$$

Where,

$$\chi_k(c,d) = \int_c^d e^y\cos\left(k\pi\frac{y-a}{b-a}\right)dy \tag{15}$$

$$= \frac{1}{1+\left(\frac{k\pi}{b-a}\right)^2}\left[\cos\left(k\pi\frac{d-a}{b-a}\right)\mathrm{e}^d - \cos\left(k\pi\frac{c-a}{b-a}\right)\mathrm{e}^c\right.$$

$$\left.+\frac{k\pi}{b-a}\sin\left(k\pi\frac{d-a}{b-a}\right)\mathrm{e}^d - \frac{k\pi}{b-a}\sin\left(k\pi\frac{c-a}{b-a}\right)\mathrm{e}^c\right]$$

And,

$$\begin{aligned}\psi_k(c,d) &= \int_c^d \cos\left(k\pi\frac{y-a}{b-a}\right)dy \\ &= \begin{cases}\left[\sin\left(k\pi\frac{d-a}{b-a}\right) - \sin\left(k\pi\frac{c-a}{b-a}\right)\right]\frac{b-a}{k\pi} & k \neq 0 \\ (d-c) & k = 0\end{cases}\end{aligned} \tag{16}$$

This concludes the the highlights from Fang and Osterlee's COS method derivation. A method that is fairly straightforward to employ in a computational context. In the following section, section 2, I detail the the choice of $[a,b]$ in Fang and Osterlee's seminal work, and describe situations where in which the intuition behind the range can become problematic.

2 Problems with the COS Method

2.1 Divergence in the Merton Jump Diffusion (MJD) Model

This issue is borne from the Fang and Osterlee's definition of the truncation ranges $[a,b]$. The choice of the correct truncation range is essential. A truncation range too large is going to require larger N to ensure convergence. A truncation range too small will result in truncation error. The truncation range originally proposed was the cumulants based truncation range. They also proposed rather quick-and-dirty range based on option maturity.

$$\text{Cumulants Based Range:} \quad [a,b] = \begin{cases}\left[c_1 - L\cdot\sqrt{c_2},\quad c_1 + L\cdot\sqrt{c_2}\right], & L = 12 \\ \left[c_1 - L\cdot\sqrt{c_2+\sqrt{c_4}},\quad c_1 + L\cdot\sqrt{c_2+\sqrt{c_4}}\right] & L = 10\end{cases} \tag{17}$$

$$\text{Maturity Range:} \quad [a,b] = \left[-L\cdot\sqrt{T},\quad L\cdot\sqrt{T}\right] \tag{18}$$

The intuition behind the cumulants based range is two fold: The n^{th} cumulant is going to be proportional to the n^{th} power of the standard deviation. The inclusion of a 4^{th} culumant in the generalised formula above is due to options with very short maturities T. These have probability distribution functions with sharp peaks and heavy tails.

The intuition behind the second range in 18 less sound: it is not going to be fit to $\ln\frac{S(t)}{K}$, but has the advantage that the range does not depend on the cumulants or the strike - and so when cumulants are difficult to compute, and when we are trying to calculate an option price for a multitude of strikes, it is quite a nice solution. L must often be chosen heuristically here. This is not entirely robust.

Ambiguity Surrounding the Truncation Ranges

General ambiguity surrounding the truncation range is widespread. I have come across a variety of different representations in various research papers of this same formula in 17, often sometimes by the same authors of the original work, Fang and Osterlee. These are often accompanied with little explanation regarding the intuition behind the change in formula. The wider sentiment among

quantitative and computational finance message boards as well as other 3rd party resources; that of general confusion, is something I hope to try and combat here. Some users claim to use it as their "work horse" for a variety of stock price models, whilst also claiming they use fixed parameters for the truncation range denoted L, and N, usually $N = 2^8$. I don't believe this is entirely "robust". **More often than not users with issues revert back to Carr-Madan or Saddle approaches, or even where applicable closed and semi-closed form solutions - missing out on the primary benefit of the COS method: speed.**

It must be noted that in the original paper they do suggest potentially using 6 cumulants (see 19), and that the above, that with 4 cumulants, is simply a rule of thumb. It is mentioned that in some cases you may need a 6^{th} cumulant for options with extremely short maturities, i.e. $T = 0.001$, but this is often challenging to derive in many advanced stock price models [10]. Below I detail why the above may not be a great rule of thumb in regards to the use of the 4^{th} cumulant anyways.

$$\text{The } 6^{\text{th}} \text{ cumulant:} \quad [a,b] = \left[c_1 - L \cdot \sqrt{c_2 + \sqrt{c_4 + \sqrt{c_6}}}, \quad c_1 + L \cdot \sqrt{c_2 + \sqrt{c_4 + \sqrt{c_6}}} \right] \tag{19}$$

Because of the fact that the 6^{th} cumulant is often too difficult to derive in some advanced stock price models, I do not use it unless stated explicitly otherwise. Also, please note in the analysis further I for the most part ignore the range in 18 and focus primarily on the the cumumlants based range.

2.1.1 Divergence of the COS Density

The problem here is the focus of Junike and Pankrashkins work [9], documented here is a series of numerical experiments with parameters mirrored in Junike and Parnkraskin's paper.

For Levy models with jumps, there are instances where the expected size of the jump is going to exist outside of the truncation range, or the truncation range is simply going to be insufficient to ensure the correct convergence. A prime example of this is the aforementioned MJD model, the subject of this section. **Taking model parameters, namely T, that are within Fang and Osterlee's original suggestions [10], we can show divergence in this model, using;**

$$T = 0.1, \quad \sigma = 0.1, \quad \lambda = 0.001, \quad \kappa = -0.5, \quad \delta = 0.2, \quad \left(\text{i.e.} \quad \mu_j = \ln(1+\kappa) - \frac{1}{2}\delta^2 \right) \tag{20}$$

With $N = 1,000$, $q = 0$, $r = 0$, $S = 0$ and $K = 100$ (ATM call option):

$$\varphi_X(u,t) = \exp\left(iu\mu t - \frac{1}{2}\sigma^2 u^2 t \right) \cdot \varphi_{\text{Merton}}(u,t) \tag{21}$$

$$\varphi_{\text{Merton}}(u,t) = \exp\left[\lambda t \left(\exp\left(iu\mu_j - \frac{1}{2}\delta^2 u^2 \right) - 1 \right) \right]$$

$$\mu := r - \frac{1}{2}\sigma^2 - q - \bar{\omega}$$

$$\bar{\omega} = \lambda \left(\exp\left(\frac{1}{2}\delta^2 + \mu_j \right) - 1 \right)$$

And the following functions for the cumulants:

$$c_1 = T\left(r - q - \bar{\omega} - \frac{1}{2}\sigma^2 + \lambda\mu_j\right) \tag{22}$$

$$c_2 = T\left(\sigma^2 + \lambda\mu_j^2 + \delta^2\lambda\right) \tag{23}$$

$$c_4 = T\lambda\left(\mu_j^4 + 6\delta^2\mu_j^2 + 3\delta^4\lambda\right) \tag{24}$$

Using this, we get the integration range $[a, b] \approx [-0.85, 0.85]$ for the cumulants based, and $[a, b] \approx [-3.16, 3.16]$ for the maturity based range. In 1 below, these results are compared against the actual probability density function - that defined by use of the 6^{th} cumulant, that which is verified by results in Junike and Pankasharin's results for the probability density function with these parameters.

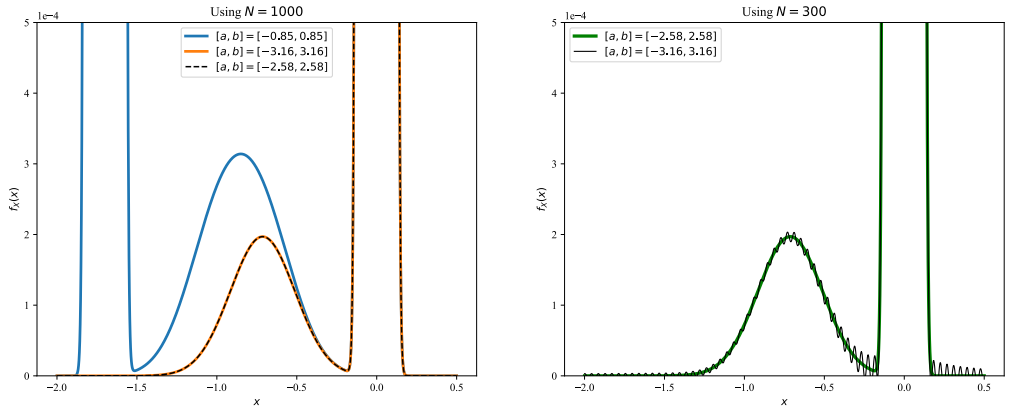

Figure 2: Convergence of the MJD PDF Using COS Method

In figure 2 we have two panels. The first uses $N = 1,000$ as described above. You can clearly see that the cumulants based range produces a significant error - this is the truncation error that arises from having a truncation range that is too small. Here we also have the real distribution for this MJD model, and the distribution defined by the T based truncation range, with $L = 10$. It converges correctly - hence inclusion of the second panel in figure 2. This is the problem that then arises from using this range with a fixed N parameter. It is now too wide, and N is needs to be larger than the range defined by the use of the 6^{th} cumulant. So, the obvious problems are The choice of N, and the choice of what cumulant range we should use. In the context of option pricing, these scenarios may lead to both minor pricing problems and major mis-pricing errors. Focusing on the cumulants based range going forward, it follows that we detail these errors exactly with some further numerical experiments.

2.1.2 The Affect on COS Method Prices

An ill-defined probability density function is going to have a significant affect on the price of the option. Ignoring the aforementioned problem of extreme moneyness, can we display this error occurring in a normal ATM option calculation using the closed form solution for a call option in a jump-diffusion process for comparison:

$$\text{Defining the BS Equation as:} \quad M(S,T;K,\sigma^2,r) = S\mathcal{N}(d_1) - Ke^{-rT}\mathcal{N}(d_2) \tag{25}$$

$$\text{Where:} \quad d_1 = \frac{\ln(S/K) + (r+\sigma^2/2)T}{\sigma\sqrt{T}}, \quad d_2 = d_1 - \sigma\sqrt{T}$$

And then the MJD option pricing formula [11], truncating the sum at 170, the highest factorial python can calculate:

$$f(S,\tau) = \sum_{n=0}^{170} \frac{e^{-\lambda' T}(\lambda' T)^n}{n!} M_n(S,T) \tag{26}$$

$$\text{where:} \quad M_n(S,T) = M(S,T;K,v_n^2,r_n), \quad v_n^2 = \sigma^2 + \frac{n\delta^2}{T}, \quad r_n = r - \lambda k + \frac{n\gamma}{T}$$

$$\gamma = \ln(1+k), \quad \lambda' = \lambda(1+k)$$

Truncation Range Error Leading to Major Mispricing: Using the MJD model with parameters described in 20, noting the choice of maturity $T = 0.1$ here, this is a European option that expires in a month, i.e. not an extremely short maturity. Choosing the first cumulant range described in 17, we get $[a,b] \approx [-0.39, 0.39]$ we get the following pricing issues.

$$\begin{aligned} V_{\text{Closed Form}} &= 1.2639205902147466 \\ V_{\text{COS}}^{N=300} &= 1.2644795849040216 \\ V_{\text{COS}}^{N=1000} &= 1.2644795849040216 \end{aligned}$$

The COS method has converged to the wrong price early on in the series - the figure below us shows that convergence is not possible with any value of N:

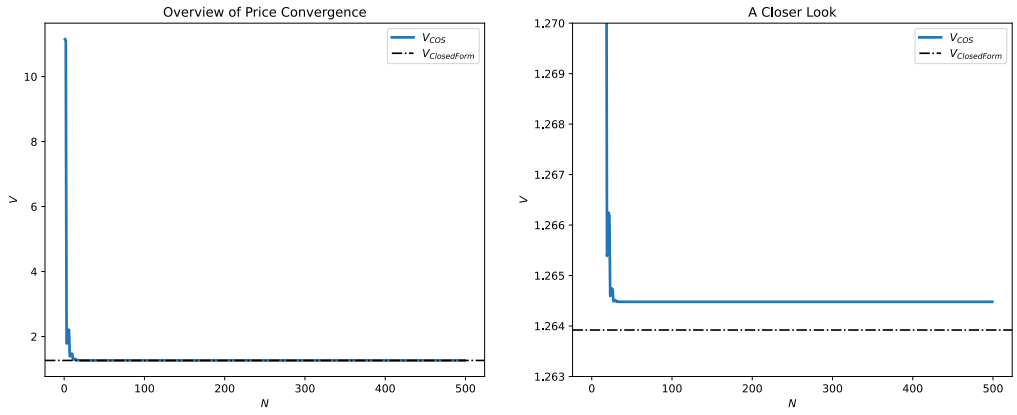

Figure 3: Convergence of Price, COS vs. Closed Form

Smaller Errors Using the 4^{th} Cumulant: It is still important to note than even when including the 4^{th} cumulant, in models with jumps we can still get some noticeable error. Changing the parameters in 27 to:

$$T = 0.01, \quad \sigma = 0.1, \quad \lambda = 0.00001, \quad \kappa = e^{-6.98} - 1, \quad \delta = 0.2, \quad \left(\mu_j = \ln(1+\kappa) - \frac{1}{2}\delta^2\right) \tag{27}$$

With the truncation range $[a, b] \approx [-1.25, 1.25]$, and $N = [300,\ 1,000]$ for testing, we get a COS price defined below:

$$\begin{aligned} V_{\text{Closed Form}} &= 0.3989455935507185 \\ V_{\text{COS}}^{N=300} &= 0.3989490908545711 \\ V_{\text{COS}}^{N=1000} &= 0.39894559786454303 \end{aligned}$$

We arrive at an acceptable answer with large enough N: **what, if any, indication do we have for the choice of N - and to what extent is it proportional to the choice of $[a, b]$.**

2.2 Extreme Moneyness Options

We know there is a problem with this option pricing technique in many advanced stock price models when considering the case of extreme moneyness. This is documented extensively in Hirsa's book "Computational Methods in Finance" [12]. I have used some of these results to verify my own in the Variance-Gamma model, see appendix 6 and section 4.4.4. This is a problem considering that this method in practise can be used to price a range of options with different strikes very quickly. It is known that moneyness does have intuitive relationship to liquidity, and option pricing models are calibrated primarily using liquid options. With that being said - providing a technique exhibits failure at ill-defined moneyness ranges is not exactly robust; hence the search for a solution.

2.2.1 The MJD Model

Using our previous MJD model, with parameters described in 20, and a truncation range that is verified to work, that defined by the 6^{th} cumulant, defined by Junike and Pankrashkin: $[a, b] \approx [-2.8, 2.8]$ and $N = 2^8$. Here we can examine this issue in detail. Looking at the call option price as before:

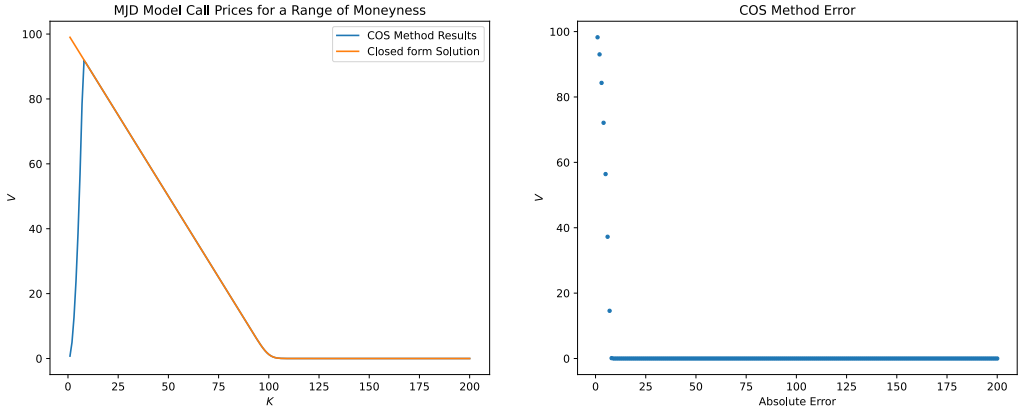

Figure 4: MJD COS Method Error for ITM/OTM Call Options

Around the $K = 10$ moneyness mark we get this failure. Past that we get errors seen to be reasonably close to 0. The analysis naturally continues to investigate other stock price models to see if this persists - starting with the simplest stock price model, the Black-Scholes model.

2.2.2 Geometric Brownian Motion

For the closed form solution, we simply use the Black-Scholes formula in 25.

Repeating the above, using the COS method with the second cumulant range described in 17, i.e. $[a, b] \approx [-0.8, 0.8]$, using the following parameters (with $q = 0$ still - this choice is consistent in all the following tests):

$$r = 0.05, \quad T = \frac{1}{12}, \quad S = 100, \quad \sigma = 0.25 \tag{28}$$

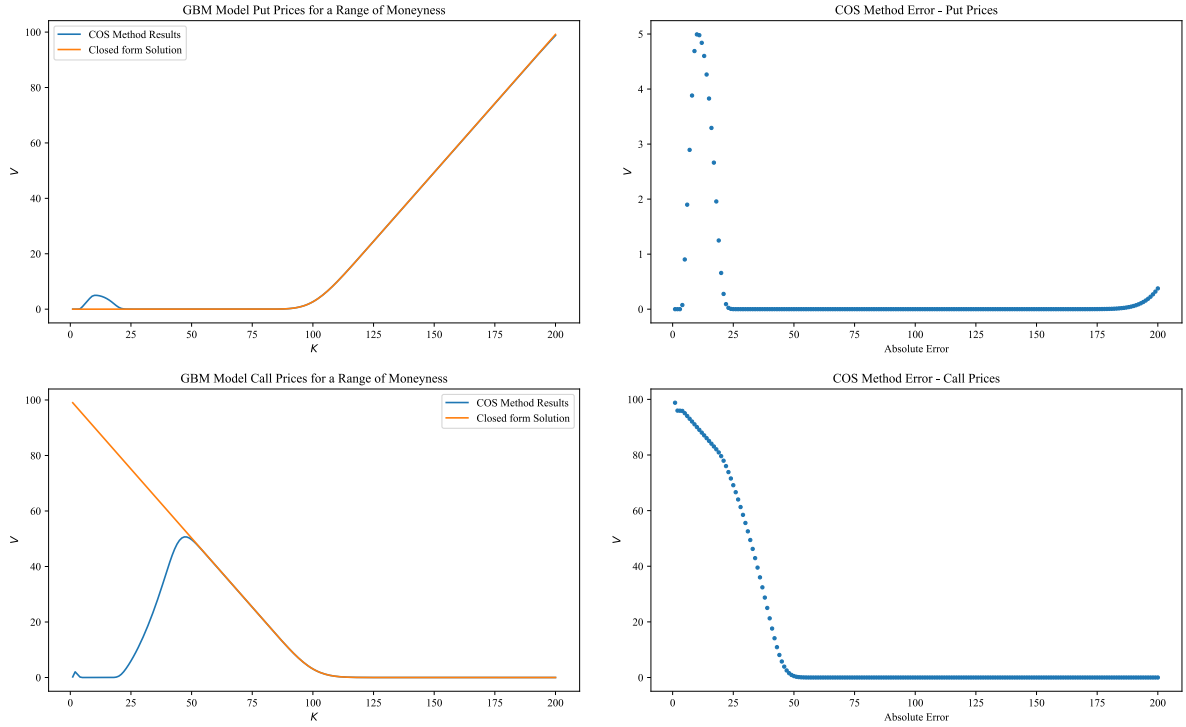

Figure 5: COS Method Error for in the GBM Model

It has been reported that this is a significant problem across numerous other advanced stock price models [13][12], and was also described by Fang and Osterlee in their original paper [10]. This issue is endemic in the derivation, unlike the issues described in section 2.1.

Revisiting Fang and Osterlee's Definition of V_k, the Call and Put Coefficients [14]: Taking another look at how payoff coefficients are derived gives us some context surrounding why this issue is going to be persistent in a all advanced stock price models. Call and Put Coefficients, denoted V_k^{Call} and V_k^{Put} in equations 13 and 14 respectively are defined relative to the strike price. I.e.,

$$y(T) = \ln\left(\frac{S(T))}{K}\right) \tag{29}$$

Where:

$$V(T,y) = [\alpha \cdot K(e^y - 1)]^+, \quad \alpha = \begin{cases} 1 & \text{Calls} \\ -1 & \text{Puts} \end{cases} \tag{30}$$

Looking at the calculation derivation of cumulants of the characteristic functions:

$$c_n(X) = \frac{1}{i^n} \cdot \frac{\partial^n(\ln(\varphi_X(u)))}{\partial u^n}\bigg|_{u=0} \tag{31}$$

Here, $\varphi_X(u)$ is computed relative to the spot price, not relative to the strike price of the option

giving rise to this issue.

3 Solutions

3.1 Junike and Pankrashkin's Moments

Junike and Pankrashkin offer a solution to the serious mis-pricing issues that may be caused by mis-selection of the truncation range [9]. They formulate a new proof of the COS method using the Markov inequality, coming up with a solution that allows us to define a truncation range subject to a pre-defined error tolerance.

The Markov Inequality essentially allows us to define an error bound for non-negative functions. Here it is used to estimate the tail sum of the n^{th} moment of f.

Their proof is extensive, and I do not think it, in its entirety, is relevant to the scope of this paper. Here is a summary of how they hypothesise we can define a new truncation range.

We are trying to find a new value for N, L and M such that:

$$\left| \int_{\mathbb{R}} f(x)v(x)dx - {\sum_{k=0}^{N}}' c_k^L c_k^M \right| \leq \varepsilon \tag{32}$$

Where:

$${\sum_{K=0}^{N}}' c_k^L v_k^M \tag{33}$$

For notational simplicity they change the truncation range notation in the original COS method: i.e. changing the $[a, b]$ to $[-L, L]$ and also changing the $[c, d]$ in the original COS method to $[-M, M]$, both symmetric ranges about 0. Where c_k^L is the COS method with truncation range L that defines the function analogous to f, and v_k^M represent the call and put coefficients in the original COS method, with $0 < M \leq L$. They suggest the following truncation range, whilst proving that results do fall within the predefined error bound ϵ. Let the n^{th} moment of the characteristic function be denoted: μ_n, and that this moment exists for $2 \geq n$, i.e,

$$\mu_n = \int_{\mathbb{R}} x^n f(x)dx < \infty \tag{34}$$

Then, define the new $[-M, M]$ range to be, for $\varepsilon > 0$:

$$M = \sqrt[n]{\frac{2K\mu_n}{\varepsilon}} \tag{35}$$

$$L = \max\left\{ M, \quad \left(-\frac{\sigma}{2\sqrt{2}} \ln\left(\frac{\sqrt{2}\sigma\varepsilon^2}{72MK^2} \frac{12}{\pi^2} \left(\frac{\sigma^2}{M^2} + \frac{2\sqrt{2}\sigma}{M} + 4 \right)^{-1} \right) \right) \right. \tag{36}$$
$$\left. , \quad -\frac{\sigma}{2\sqrt{2}} \log\left(\frac{2\sqrt{2}\sigma\varepsilon^2}{72MK^2} \right) \right\}$$

They go on to prove that these functions are indeed inline with assumptions regarding convergence of the COS method, and that the resulting errors ϵ for sufficiently large N are $= \epsilon < \varepsilon$.

Conclusions | Correcting notational differences between Fang and Osterlee, and Junike and Pankrashkin:

For sufficiently small ε (here we use 10^{-7}) $L = M$. This is because the M is of the order $\varepsilon^{-1/n}$, and the other terms in equation 36, are of the order $\ln(\varepsilon)$. Meaning, importantly, we can make minor changes to the original COS method program implemented in previous numerical experiments.

The original COS method defines the call and put coefficients v_K^M as follows:

$$v_K^M = \int_{-M}^{M} v(x)\cos\left(k\pi\frac{x+L}{2L}\right)dx \tag{37}$$

Where $x = \ln(S(T)/K)$. This corresponds with Osterlee's notation, call and put coefficients are separated, and defining a χ and ψ functions, such that computational implementation is considerably easier for both calls and puts. Switching back to the H_k notation now, for $a < 0 < b$ with $y = \ln(S(T)/K)$.

$$\begin{aligned} H_k^{\text{Call}} &= \frac{2}{b-a}\int_0^b K(e^y - 1)\cos\left(\frac{y-a}{b-a}\right)dy \\ &= \frac{2}{b-a}K(\chi_k(0,b) - \psi_k(0,b)) \\ \text{similarly,}\quad H_k^{\text{Put}} &= \frac{2}{b-a}K(\psi_k(a,0) - \chi_k(a,0)) \end{aligned} \tag{38}$$

Where:

$$\begin{aligned} \chi_k(c,d) &= \int_c^d e^y\cos\left(k\pi\frac{y-a}{b-a}\right)dy \\ &= \frac{1}{1+\left(\frac{k\pi}{b-a}\right)^2}\left[\cos\left(k\pi\frac{d-a}{b-a}\right)\mathrm{e}^d - \cos\left(k\pi\frac{c-a}{b-a}\right)\mathrm{e}^c\right. \\ &\left. + \frac{k\pi}{b-a}\sin\left(k\pi\frac{d-a}{b-a}\right)\mathrm{e}^d - \frac{k\pi}{b-a}\sin\left(k\pi\frac{c-a}{b-a}\right)\mathrm{e}^c\right] \end{aligned} \tag{39}$$

And:

$$\begin{aligned} \psi_k(c,d) &= \int_c^d \cos\left(k\pi\frac{y-a}{b-a}\right)dy \\ &= \begin{cases} \left[\sin\left(k\pi\frac{d-a}{b-a}\right) - \sin\left(k\pi\frac{c-a}{b-a}\right)\right]\frac{b-a}{k\pi} & k \neq 0 \\ (d-c) & k = 0 \end{cases} \end{aligned} \tag{40}$$

In the COS method we only compute put prices, and rely on put and call parity to find the value for the calls. I have explained this reasoning further in 3.2.

For $[c,d] \subset [a,b]$. Using all the the above, it follows that the new payoff coefficients, given this truncation range, and using $\varepsilon = 10^{-7}$, that I will apply in my computation is:

$$V_{\text{COS}}(M,N) = e^{-rT} \sum_{k=0}^{N-1}{}' \text{Re}\left\{ \varphi_X \left(\frac{k\pi}{2M} e^{2^{-1}(ik\pi)} \right) \right\} \cdot H_k \tag{41}$$

$$\text{where:} \quad H_k = -\frac{1}{M} L(\psi_k(-M,0) - \chi_k(-M,0)) \tag{42}$$

Which is simply the original COS method, but with symmetric $a = -b$ around 0, where $b = M$, the new moments defined range. Here we should see, for sufficiently large N:

$$\left| V_{\text{COS}}(M,N) - \int_{\mathbb{R}} v(x) f(x) dx \right| = \epsilon \leq \varepsilon \tag{43}$$

3.1.1 Solving the problem in the MJD model

The results of implementing this new formula to see if errors have been extracted in the first extreme case divergence seen in 2.1.2, where at no point does the answer converge to the real answer, as well as in the second case in 2.1.2, where a suitably correct answer is achieved after numerous iterations.

Moments of the MJD Characteristic Function

Using 44 we can derive the moments of the characteristic function and use it to define a truncation range in both sets of parameters. It is mentioned in Junike and Pankraskin's experiments that the 8^{th} moment is enough [9], after which we do not see enough change in the value to change the truncation range to a level that would hinder results. Let each moment of the characteristic function in question be μ_n, then:

$$\mu_n = \frac{1}{i^n} \cdot \frac{d^n \varphi_X(u)}{du^n}\bigg|_{u=0}, \quad \mu_8 = \frac{1}{i^8} \cdot \frac{d^8 \varphi_X(u)}{du^8}\bigg|_{u=0} \tag{44}$$

In testing these solutions I use Wolfram Mathematica to calculate this μ_8.

$$\mu_8 \approx 2.20433 \cdot 10^{-8}, \quad M = \sqrt[8]{\frac{4.40866 \cdot 10^{-8}}{10^{-7}}} = 0.902689 \cdot \sqrt[8]{K} \tag{45}$$

Going back and implementing this function into the equation, with $\mu_8 \approx 3.58$.

MJD Results

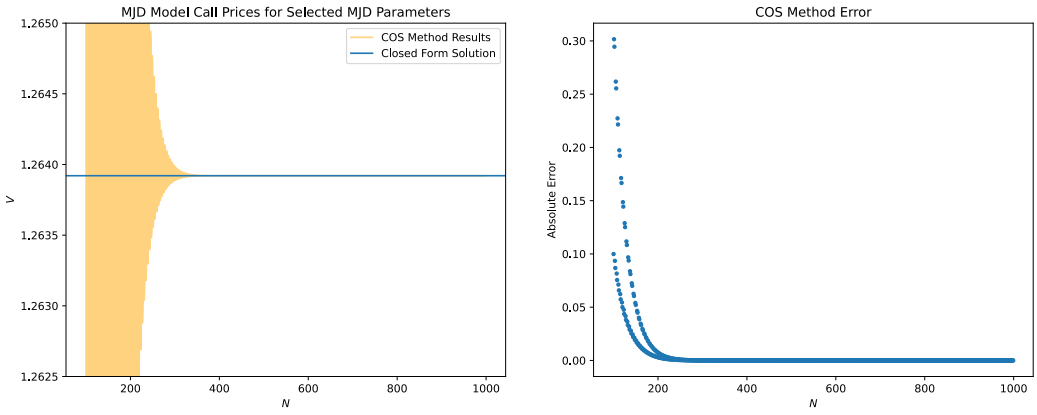

Figure 6: COS Method Error for in the MJD Model Using Moments Truncation Range

The option price converges to the correct result. Although taking large N here is not ideal, it is obviously better than serious mis-pricing. *We are at the point of obscuring the goal of attempting to maintain the efficiency, borne from exponential convergence of the COS method.* Here, even with this large N, it is still quicker to price options under MJD via the COS method as opposed to the closed - form solution where the looping summation for the Jump Diffusion Model using Python's `math` package to calculate factorials in each iteration puts strain on the efficiency of the method.

Now, moving on to the very small errors seen in the second set of parameters in 20. Can we also extrapolate these errors for the same ATM option? Here we see the value converge to well within our desired error bounds. Here we have this same problem of large N.

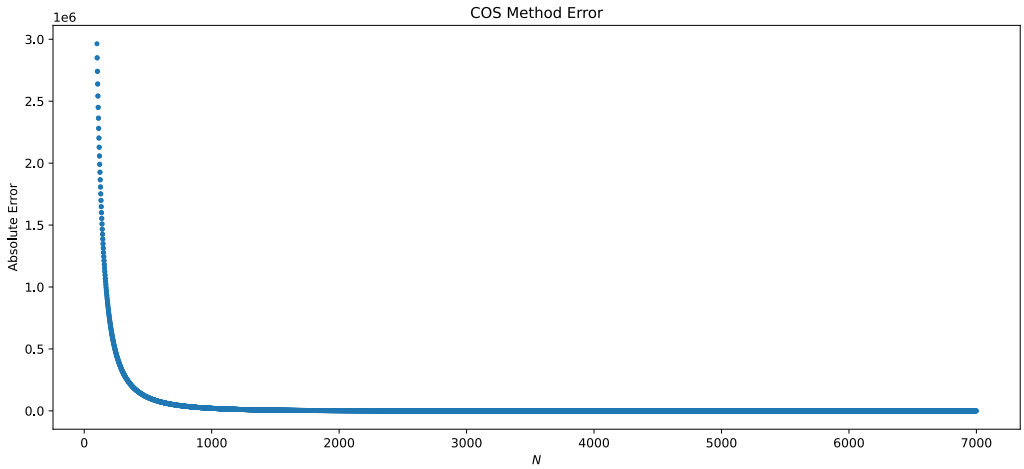

Figure 7: COS Method Error for MJD - With New Markov Ranges

$$V_{\text{Closed Form}} = 0.3989455935507185 \tag{46}$$
$$V_{\text{COS}}^{N=300} = 1.2737040423912387 \tag{47}$$
$$V_{\text{COS}}^{N=1000} = 0.49848290396371775 \tag{48}$$
$$V_{\text{COS}}^{N=6000} = 0.3989455966818699 \tag{49}$$
$$V_{\text{COS}}^{N=20000} = 0.3989455935507349 \tag{50}$$

Because of the still apparent speed of using more N in vectorised code, what we should be able to do is essentially find an approximate relationship between the size of the truncation range and the N required for convergence. In experimentation, I find this to in the range of $\{z \in \mathbb{Z} \mid 500 \leq z \leq 1,000\}$, where $N = \lfloor z \cdot M \rfloor$. The trade-off is obviously going to be that between speed and accuracy, as the choice of N is what is directly linked to the computational efficiency of method.

3.2 Le Floch's Forward Prices

In Le Floc'h's paper "More Robust Pricing of European Options Based on Fourier Cosine Series Expansions" a Heston Stochastic Volatility model is described where in which divergence is exhibited in the extreme moneyness cases [13].

Le Floc'h derives a new derivation of the put price formula, but relative to the stock no arbitrage forward price $F = S_0 e^{-rT}$. He concludes that the original COS method can be used, but the truncation range for Put prices must be shifted relative to $h = \ln\frac{K}{F}$, i.e. $[a_{\text{LF}}, b_{\text{LF}}] = [a - h, b - h]$. When K is outside of the truncation, i.e. when $h > b$, we should use the intrinsic value of the option as the pricing formula is no longer relevant: $e^{-rT}[K - F]^+$. This also goes for when $h < a$, in such a scenario, we the value of the put option should be set to 0.

It is noted again here that in general we should use the put option pricing formula and rely on no arbitrage put call parity in order to calculate the call option price. This is due to absolute value of the cosine coefficients in the formula increase exponentially with time to maturity, whereas in the case of the put formula, it remains constant.

3.2.1 GBM Results

Below I the above aforementioned solutions in the GBM model, using the parameters outlined in 28, and using the cumulant based range (there is only two cumulants calculable here in the GBM model, so the range is stuck at $[a, b] = [-0.86, 0.86]$), using the cumulant equations described below:

$$c_1 = rt - \frac{1}{2}\sigma^2 t, \quad c_2 = \sigma^2 t, \quad c_4 = 0 \tag{51}$$

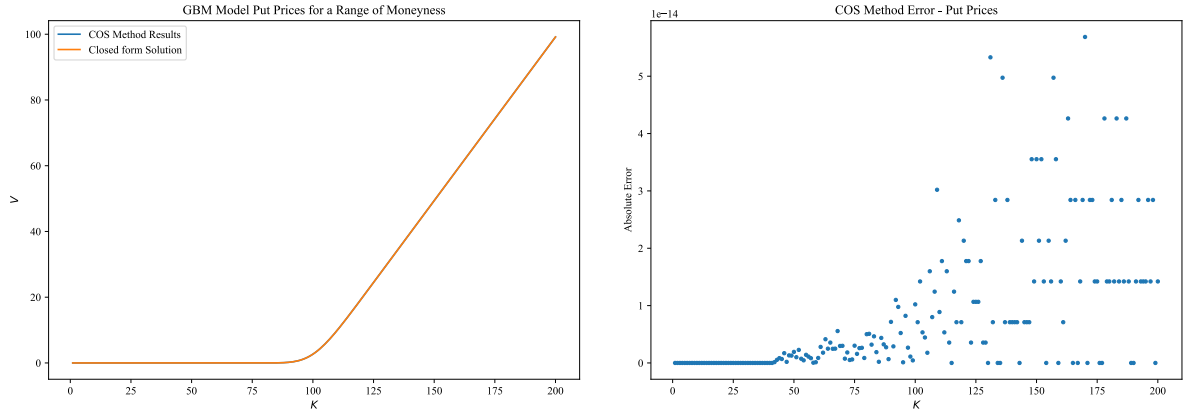

Figure 8: New COS Black-Scholes Method Error - Put Prices

Results here have errors that are at maximum $\approx 6e$-14, which is well within reason. Now, employing put call parity to find the call prices with this same spot price and the same moneyness range, it should follow our results are also correct:

$$V_t^{\text{Call}} - V_t^{\text{Put}} = S_t - Ke^{-rT} \tag{52}$$

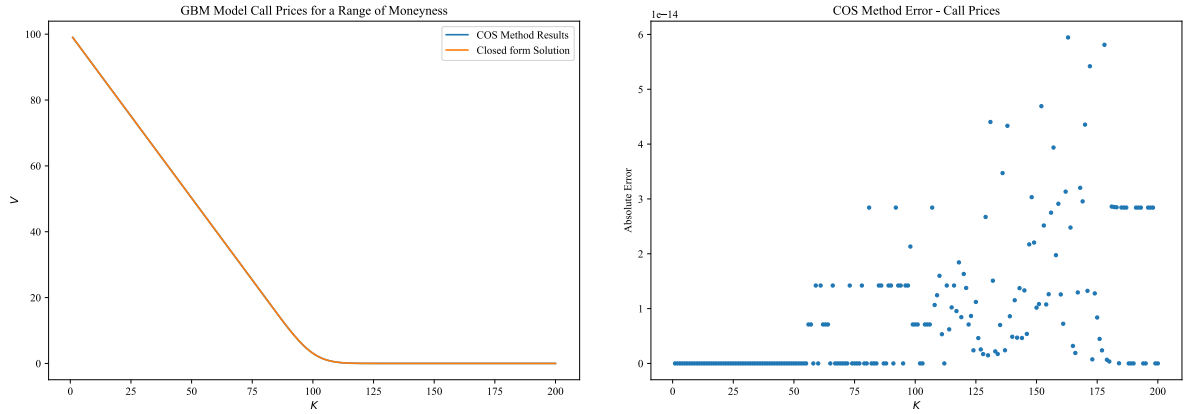

Figure 9: New COS Black-Scholes Method Error - Call Prices

We have the desired results. Errors that are at a maximum $\approx 5e$-14 for this range of K.

3.2.2 Applying this to Merton Jump Diffusion

Going ahead and using the Markov Range. Note here that the truncation range widens with strike, using the n^{th} moment at a rate of $\sqrt[n]{K}$. This rate is for reasonable choices of n and K, and so

should not be a problem. In the below we use the parameters in 20, but use the cumulant range to define where put valuation may be 0. Here, using $N = \lfloor 1000 \cdot M \rfloor$ to maximise accuracy. First, a look at what happens to put options very far out of the money with the Markov cumulants range method: i.e. using $[a, b] = [-0.86, 0.86]$. Note, as opposed to what was discussed in section 2.2, we are only interested in put prices, and seek to rely on put call parity to define the call prices. See results below:

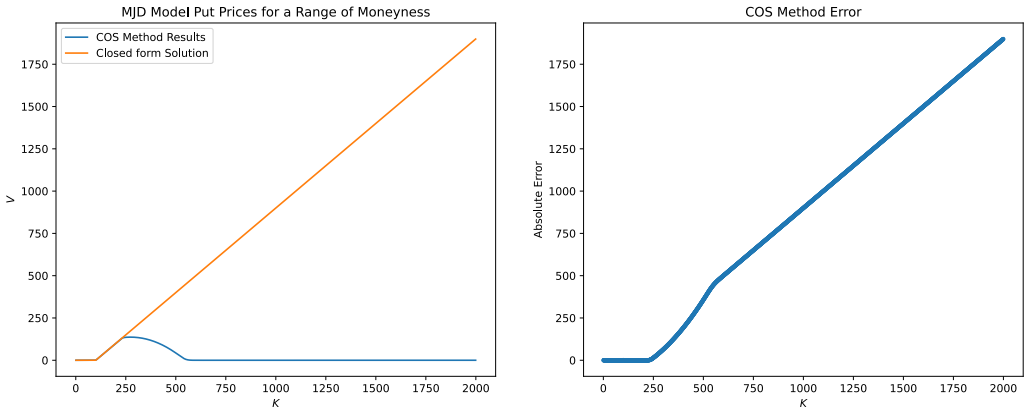

Figure 10: COS Method Error for in the MJD Model

Evidence suggests the COS method as originally constructed completely fails to price ITM put options accurately. Employing Le Floc'h's solution:

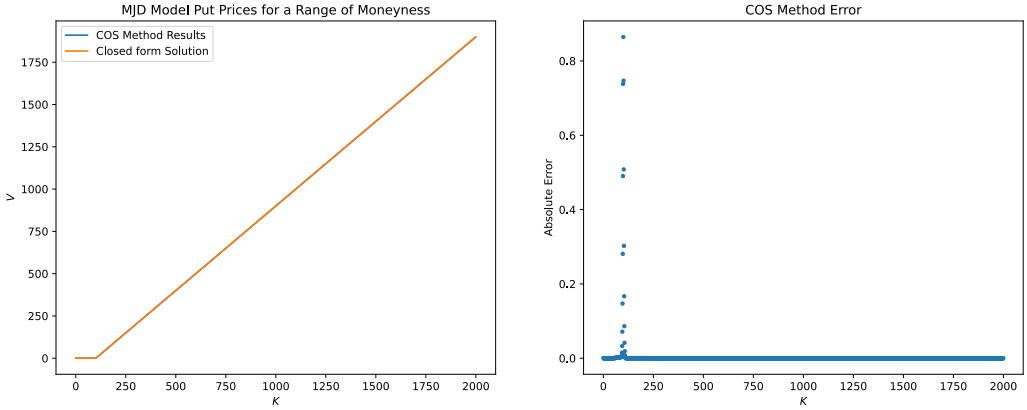

Figure 11: COS Method Error for in the MJD Model with Le Floc'h's Augmentations

A new error arises at the in the money options, this is essentially where Le Floc'h's formula is erroneously pricing the put 0. It is clear that, because the cumulants based range is tighter than the moments range, the formula is being constrained incorrectly - the COS method formula is indeed

unstable still at this level of truncation. And so, employing Junike and Pankrashkins moments for the following range of strikes.

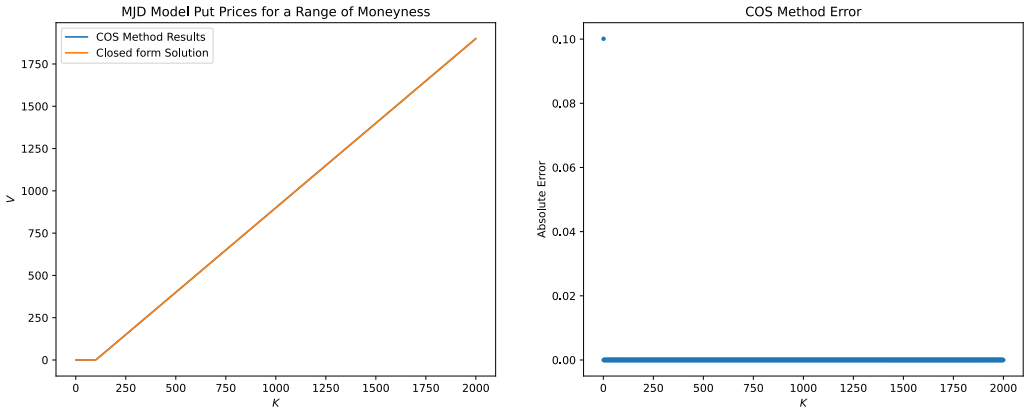

Figure 12: COS Method Error for in the MJD Model using Junike and Pankrashkin's Solutions

Where we have one small error at the first option. It would follow from Le Floc'h's work that we should use a new parameter in the $a < \ln\frac{K}{F}$. Using $-M < \ln\frac{K}{F} \Rightarrow V = 0$. Employing this part of Le Floc'h's solution bears the following results:

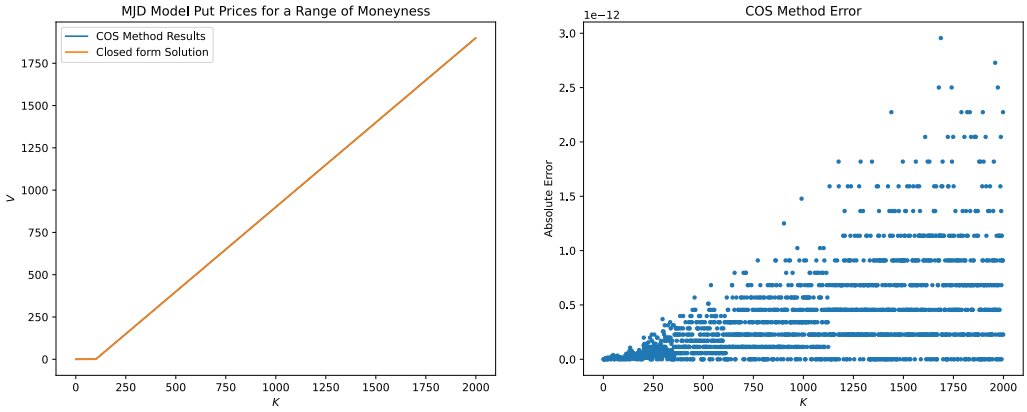

Figure 13: COS Method Error for in the MJD Model with both Solutions

In Le Floc'h's augmentation, we have the truncation range shifting left on the in $\mathbb{R}$. Shifts in the $[a, b]$ range relative to the strike price depending on the sign of the natural logarithm of K/F are going to be congruent to what is encapsulated in the moments based range. We can demonstrate this with some arbitrary moments based and cumulant ranges; noting that the moments based range is generally wider than that of the cumulants based range.

Taking, for example, an $[a, b] = [-2, 2]$ range. For ease, lets call $F = 100$ and $K = \{K \in \mathbb{Z} \mid 0 \leq K \leq 200\}$. Now for M. Recall that:

$$M = \sqrt[8]{\frac{2K\mu_8}{\varepsilon}} = \sqrt[8]{\frac{2\mu_8}{\varepsilon}} \cdot \sqrt[8]{K}$$

Denote

$$p = \sqrt[8]{\frac{2\mu_8}{\varepsilon}}$$

Because $[-p, p]$ is usually by definition, with $n = 8$ and $\varepsilon = 10^{-7}$, going to be wider in magnitude than $[a, b]$. Again, for demonstrative purposes let $p = 2.5$, i.e. $[-p, p] = [-2.5, 2.5]$. I.e.,

$$[-M, M] = [-p \cdot \sqrt[8]{K}, p \cdot \sqrt[8]{K}]$$

Using this:

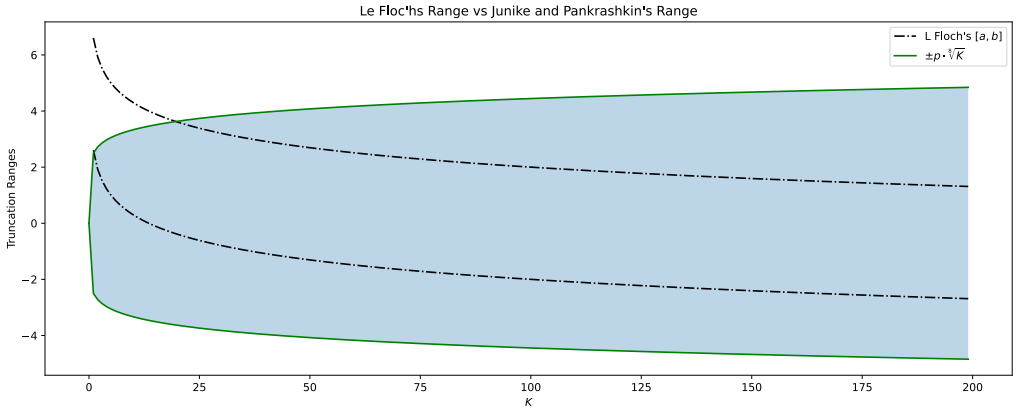

Figure 14: Changing Range Sizes against K

The above gives light as to why Junike and Pankrashkin's moments based range may also bare a solution for the issues surrounding extreme moneyeness of these vanilla options, the truncation range is wide enough and strike dependent. In computational experiments this tends to always be the case, especially as M often outstrips $[a, b]$ cumulants based ranges by a wide margin. With N set to be proportional to M, it follows that there will be convergence in the models discussed.

4 Implementation

The above research and testing leads us in the direction of a clear answer in regards to a robust and efficient option pricing formula, using a combination work from Fang and Osterlee's original COS method, Le Floc'h's follow up solution for extreme moneyness and most importantly Junike and Pankrashin's moments solution based solution for Lévy processes with jumps, which can also be applied to Heston Models as well.

4.1 Computing Partial Derivatives in Python

The computational intensity of computing these derivatives is a major hurdle for this approach. For my experiments and results in the work completed to this point I have used Junike and Pankrashkin's results for the moments that have been verified by computing them analytically in Wolfram Mathematica. This is a very extensive process, and should not be considered a reasonable solution for an efficient computational implementation that is closed at both ends.

Junike and Pankrashkin note this to be a an outstanding issue, and in order to maintain some efficiency of the COS method a random forest machine learning approach could be adopted to solve this partials problem. I propose an alternative solution using the connection between the central moments of a characteristic function and their respective cumulants [15]. Cumulants considerably simpler to calculate, and we can derive central moments from these cumulants. Let the n^{th} moment of a real-value random variable be:

$$\mu_n = \mathbb{E}\left(X^n\right) = \int_{-\infty}^{\infty} x^n f(x) dx \tag{53}$$

For $n \in \mathbb{Z}_{>0}$, provided the Taylor expansion about the origin is of course the moment generating function:

$$M(\xi) = \mathbb{E}\left(e^{\xi X}\right) = \mathbb{E}\left(1 + \xi X + \cdots + \xi^n X^n/n! + \cdots\right) \tag{54}$$

$$= \sum_{n=0}^{\infty} \mu_n \xi^n / n! \tag{55}$$

Similarly, cumulants c_n are also defined by the Taylor expansion of the cumulant generating function $K(\xi)$:

$$K(\xi) = \log M(\xi) = \sum_n c_n \xi^n / n! \tag{56}$$

We can then find the relationship between the moments and the cumulants by extracting the coefficients from the expansion [15]:

$$\begin{aligned} \mu_2 &= c_2 + c_1^2 \\ \mu_3 &= c_3 + 3c_2c_1 + c_1^3 \\ \mu_4 &= c_4 + 4c_3c_1 + 3c_2^2 + 6c_2c_1^2 + c_1^4 \end{aligned} \tag{57}$$

This allows us to use the far easier to compute cumulants to find a value for the n^{th} moment. Cumulants for most distributions tend to have fairly straight forward formulas to implement, another positive for closed computation. Going forward we use:

$$\mu_4 = c_4 + 4c_3c_1 + 3c_2^2 + 5c_2c_1^2 + c_1^4 \tag{58}$$

4.2 The Mathematics

Given the characteristic function of a stochastic process $\varphi(u)$

$$\mu_4 = \frac{\partial^4 \varphi_X(u)}{\partial u^4}\bigg|_{u=0}, \quad M = \sqrt[4]{\frac{2\mu_4}{10^{-7}}} \cdot \sqrt[4]{K} = -a = b, \quad F = S_0 e^{-rT} \tag{59}$$

$$\mu_4 = c_4 + 4c_3c_1 + 3c_2^2 + 5c_2c_1^2 + c_1^4 \tag{60}$$

$$N = \lfloor z \cdot M \rfloor, \quad \{z \in \mathbb{Z} \mid 500 \leq z \leq 1,000\} \tag{61}$$

$$V_{\text{COS}}^{\text{Put}}(M,N,T) = \begin{cases} e^{-rT} \sum_{k=0}^{N-1}{}' \text{Re}\left\{ \varphi_X\left(\frac{k\pi}{2M}\right) \exp\left(\frac{ik}{M}\right) \right\} \cdot H_k, & \ln\left(\frac{K}{F}\right) \geq M \\ 0, & \ln\left(\frac{K}{F}\right) < M \end{cases} \tag{62}$$

$$V_{\text{COS}}^{\text{Call}} = V_{\text{COS}}^{\text{Put}}(M,N,T) + S - Ke^{rT} \tag{63}$$

$$H_k = \frac{K}{M}\left(\chi_k(M,0) - \psi_k(M,0)\right) \tag{64}$$

Where χ_k and ψ_k functions are detailed in 39 and 40 respectively.

4.3 The COS Method in Python

The final delivery comes as a `.ipynb` notebook, where I have included most of my research and testing. Here I have annotated throughout with using Markdown - this is what is discussed below.

- First, I go ahead and import the required packages, and suppress the numpy deprecation warning, this is for their complex numbers. This is more to remove a nuisance than to provide any useful functionality. Note here, also output software versions. Check the versions are congruent with those described in **Software**.

```
# Python Version
from platform import python_version
print(str('Python Version: ') + python_version())

# Vectorise code using numpy
import numpy as np
print(str('Numpy Version: ') + np.version.version)

# Time the COS Method
import time
# time is built in to python - version is the same as the python instillation

# Remove numpy complex number deprecation warnings
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
# warnings is built in to python - version is the same as the python instillation
```

```
# Scipy intergrate for HSV Model
from scipy.integrate import quad

# For Variance Gamma
import scipy.stats as ss
import scipy as scp

# Norm function from Scipy States - normal cdf
from scipy.stats import norm
print(str('Scipy Version: ') + scp.__version__)

# Math for factorial function
import math
# Math is built in to python - version is the same as the python instillation

# Plotting testing
import matplotlib
import matplotlib.pyplot as plt
print(str('Matplotlib Version: ') + matplotlib.__version__)
```

- The first two formulas are for the χ and ψ components, defined in 39 and 40 respectively These functions takes the values for a, b, c, d and k.

```
    def Psi(a,b,c,d,k):

        # Formula for all the psi values
        psi = np.sin(k * np.pi * (d - a) / (b - a)) - np.sin(k * np.pi * (c - a)/(b
            - a))

        # When k not equal to 0
        psi[1:] = psi[1:] * (b - a) / (k[1:] * np.pi)

        # When k = 0
        psi[0] = d - c

        # Return all psi's
        return -psi

    def Chi(a,b,c,d,k):

        # Outside of the square brackets
        chi_out = 1.0 / (1.0 + np.power((k * np.pi / (b - a)) , 2.0))

        # First half inside the square brackets
        chi_in_1 = np.cos(k * np.pi * (d - a)/(b - a)) * np.exp(d) - np.cos(k * np.pi
                    * (c - a) / (b - a)) * np.exp(c)

        # Second half inside the square brackets
        chi_in_2 = k * np.pi / (b - a) * np.sin(k * np.pi *
                          (d - a) / (b - a)) - k * np.pi / (b - a) * np.sin(k
                          * np.pi * (c - a) / (b - a)) * np.exp(c)
```

```
    # Bringing them all together
    chi = chi_out * (chi_in_1 + chi_in_2)

    # Return all chi's
    return -chi
```

- The next formula is a function of the H_k's, the put coefficients, defined in equation 64, this takes a, b and k.

```
def Hk(a,b,k):

    # On the interval [-M,0]
    c = a
    d = 0.0

    # Get a chi_k and psi_k value from the chi and psi function based on
        this M
    Chi_k = Chi(a,b,c,d,k)
    Psi_k = Psi(a,b,c,d,k)

    # define the H_ks
    H_k  = 2.0 / (b - a) * -(Psi_k - Chi_k)

    # Return the H_k's
    return H_k
```

- The final formula brings it all together. I have added many annotations to this, it follows from the original COS method, but with some essential differences.
  - The formula takes an additional variable, `mu_4`, this is the 4^{th} moment of the characteristic function in question.
  - The next change is N. For formula no longer takes N, but uses $1000 \cdot M$ to find the N to ensure convergence, as in 61
  - The final changes come at the end, where we are calculating the no arbitrage forward price, finding $\ln(K/F)$, and determining whether the put price should be 0. Then, we take the `call/put` variable, which is of string type, and is either `call` or `put`, if it is a call we are pricing, it uses the put call parity formula to price the option. As in 63

```
def COS(CP,cf,S0,r,T,K,mu_4):

    # Define M
    M = np.power((2*K*mu_4)/(10**-7), (1/4))

    # Define N as z*M. I have chose 1,000 here to ensure accuracy in
        most cases
    N = int(1_000 * M)
```

```
# Setting some constants for the summation
i = np.complex(0.0,1.0)
y = np.log(S0 / K)

# Setting the truncation domain [-M,M]
a = - M
b = M

# Create a column vector of the (in fang and osterlee's notation)
    v_{k}^{M}
k = np.linspace(0,N-1,N).reshape([N,1])
u = k * np.pi / (b - a);

# Grab the put coefficients based on this truncation domain
H_k = Hk(a,b,k)

# Now doing the sum
sumation = cf(u) * H_k
    # Halfing the first value (sum ')
sumation[0] = 0.5 * sumation[0]

# Outer product
outerproduct = np.exp(i * np.outer((y - a) , u))

# Final result, discount and *K * real part of the dot product of
    the summation and the outerproduct variable
output = np.exp(-r * T) * K * np.real(outerproduct.dot(sumation))

# Implenting the condition for the forward
Forwards = float(S0 * np.exp(-r*T))
ftest = np.log(K/Forwards)

if ftest < a:
    out = 0
else:
    out = float(output[:,0][0])

if CP == "put":
    price = out
elif CP == "call":
    price = out + S0 - K*np.exp(-r*T)

# Return this output
return price
```

What follows is testing for specific advanced stock price models. In each model characteristic functions, moments and closed form solutions are going to be model specific. For the Variance-Gamma (VG) model I have used a simple Monte-Carlo program to verify price.

4.4 Specific Advanced Stock Price Models

4.4.1 Geometric Brownian Motion

Closed Form Solution [11]:

$$M\left(S,T;K,\sigma^2,r\right) = S\mathcal{N}\left(d_1\right) - Ke^{-rT}\mathcal{N}\left(d_2\right) \tag{65}$$

Where,

$$d_1 = \frac{\ln(S/K) + \left(r + \sigma^2/2\right)T}{\sigma\sqrt{T}}, \quad d_2 = d_1 - \sigma\sqrt{T}$$

Characteristic Function [14]:

$$\varphi_X(u,t) = \exp\left(iu\mu t - \frac{1}{2}\sigma^2u^2t\right) \tag{66}$$

$$\mu = r - \frac{1}{2}\sigma^2$$

Cumulants:

$$c_1 = rt - \frac{1}{2}\sigma^2t, \quad c_2 = \sigma^2t, \quad c_3, \quad c_4 = 0 \tag{67}$$

Parameters:

$$r = 0.05, \quad T = \frac{1}{12}, \quad S = 100, \quad \sigma = 0.25 \quad z = 1,000 \tag{68}$$

Accuracy:

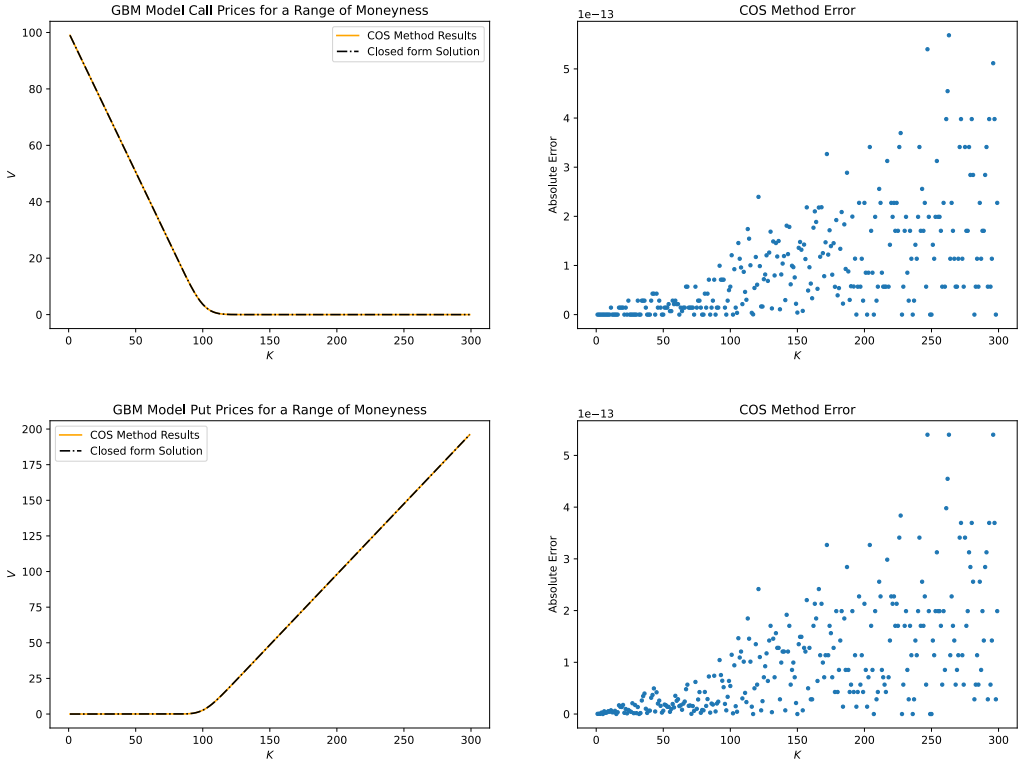

Figure 15: GBM Results

Speed-test

COS Method: 0.9427030086517334s, Closed Form: 0.04162478446960449s

Python Code: See Appendix 6

4.4.2 Merton Jump Diffusion

Closed Form Solution [11]:

$$V(S,T) = \sum_{n=0}^{170} \frac{e^{-\lambda' T}(\lambda' T)^n}{n!} M_n(S,T) \tag{69}$$

$$M_n(S,T) = M\left(S,T;K,v_n^2,r_n\right), \quad v_n^2 = \sigma^2 + \frac{n\delta^2}{T}, \quad r_n = r - \lambda k + \frac{n\gamma}{T}$$
$$\gamma = \ln(1+k), \quad \lambda' = \lambda(1+k)$$

Characteristic Function [14]:

$$\varphi_X(u,t) = \exp\left(iu\mu t - \frac{1}{2}\sigma^2 u^2 t\right) \cdot \varphi_{\text{Merton}}(u,t) \tag{70}$$

$$\varphi_{\text{Merton}}(u,t) = \exp\left[\lambda t\left(\exp\left(iu\mu_j - \frac{1}{2}\delta^2 u^2\right) - 1\right)\right]$$

$$\mu := r - \frac{1}{2}\sigma^2 - q - \bar{\omega}$$

$$\bar{\omega} = \lambda\left(\exp\left(\frac{1}{2}\delta^2 + \mu_j\right) - 1\right)$$

Cumulants:

$$c_1 = T\left(r - q - \bar{\omega} - \frac{1}{2}\sigma^2 + \lambda\mu_j\right) \tag{71}$$

$$c_2 = T\left(\sigma^2 + \lambda\mu_j^2 + \delta^2\lambda\right)$$

$$c_3 = T\lambda\left(3\mu_j\delta^2 + \mu_j^3\right)$$

$$c_4 = T\lambda\left(\mu_j^4 + 6\delta^2\mu_j^2 + 3\delta^4\lambda\right)$$

Parameters:

$$T = 0.1, \quad \sigma = 0.1, \quad \lambda = 0.001, \quad \kappa = -0.5, \quad \delta = 0.2, \quad \left(\text{ i.e. } \quad \mu_j = \ln(1+\kappa) - \frac{1}{2}\delta^2\right), \quad S = 100 \tag{72}$$

Accuracy:

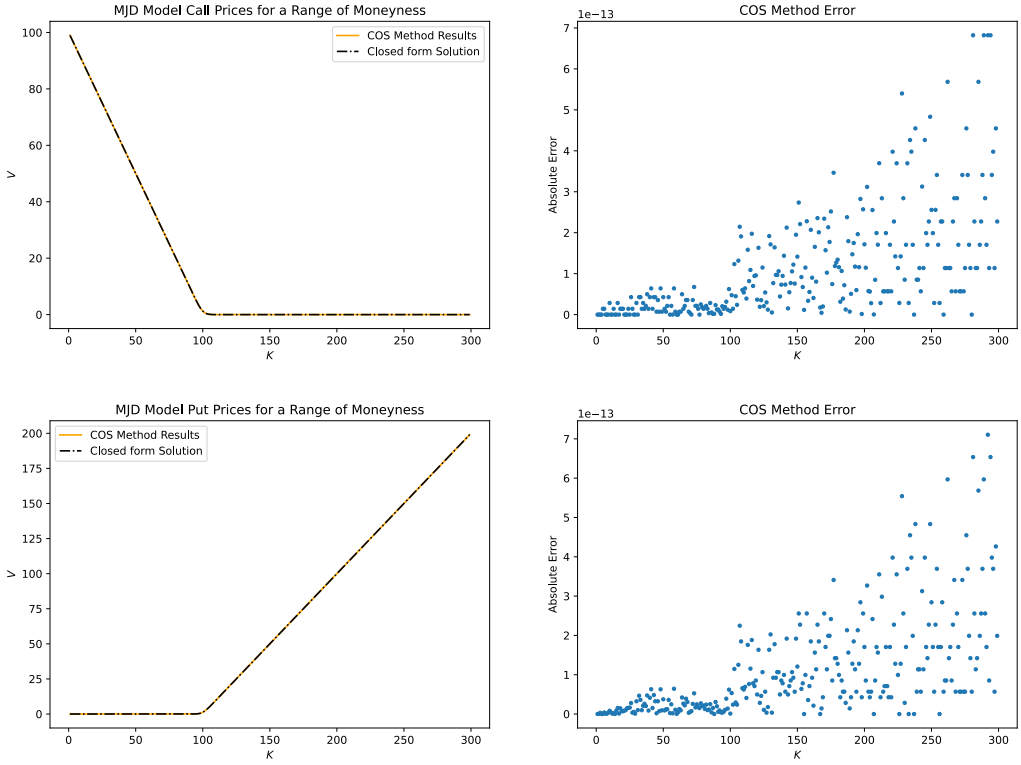

Figure 16: MJD Results

Speedtest:

COS Method: $0.7268922328948975s$ Closed Form: $4.095110893249512s$

Python Code: See Appendix 6

4.4.3 Heston Stochastic Volatility

Heston's Semi-Closed Form Solution [16]: Calculating Call prices, and using put-call parity to define put prices 52. With,

$$x := \ln\left(\frac{F_{t,T}}{K}\right), \quad F_{t,T} = \frac{S(t)}{B(t,T)} \tag{73}$$

We want to solve:

$$\hat{C}(x,v,\tau) = \frac{C(x,v,\tau)}{B(t,T)} \quad \tau := T - t \tag{74}$$

$$\hat{C}(x,v,\tau) = K\left(e^{x}P_1(x,v,\tau) - P_0(x,v,\tau)\right) \tag{75}$$

Defining the following variables:

$$\alpha = -\frac{u^2}{2} - \frac{iu}{2} + iju, \quad \beta = k - \rho\eta j - \rho\eta iu, \quad \gamma = \frac{n^2}{2} \tag{76}$$

$$r_\pm = \frac{\beta \pm \sqrt{\beta^2 - 4\alpha\gamma}}{2\gamma} =: \frac{\beta \pm d}{\eta^2}$$

$$D(u,\tau) = r_- \frac{1 - e^{-d\tau}}{1 - ge^{-d\tau}}$$

$$d = \sqrt{(k - \eta\rho_{x,v}iu)^2 + (u^2 + iu)\,\eta^2}$$

$$C(u,\tau) = k\left(r_-\tau - \frac{2}{\eta^2}\ln\left(\frac{1 - ge^{-d\tau}}{1 - g}\right)\right), \quad g := \frac{r_-}{r_+}$$

$$P_j(x,v,\tau) = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \mathrm{Re}\left[\frac{\exp\{C_j(u,\tau)\theta + D_j(u,\tau)v + iux\}}{iu}\right] du$$

Characteristic Function [16]:

$$\varphi(u) = \exp\left(C(u,\tau)\cdot\theta + D(u,\tau)\cdot v + iux\right) \tag{77}$$

Cumulants:

In [17] they define the Heston Stochastic Volatility function with the two following stochastic differential equations:

$$dS(t) = \mu S(t)dt + \sqrt{v(t)}S(t)dW_1(t) \tag{78}$$

$$dv(t) = \alpha\left(\sigma^2 - v(t)\right)dt + k\sqrt{v(t)}dW_2(t) \tag{79}$$

Congruent to the notation in :

$$dS(t) = r(t)S(t)dt + \sqrt{v(t)}S(t)dZ_1(t) \tag{80}$$

$$dv(t) = k(\theta - v(t))dt + \eta\sqrt{v(t)}dZ_2(t) \tag{81}$$

The cumulants are derived as follows:

$$c_1 = -\frac{1}{2}\sigma^2 t \tag{82}$$

$$c_2 = \frac{\sigma^2}{8\alpha^3}\left[-k^2\mathrm{e}^{-2\alpha t} + 4k\mathrm{e}^{-\alpha t}(k - 2\alpha\rho) + 2\alpha t\left(4\alpha^2 + k^2 - 4\alpha k\rho\right) + k(8\alpha\rho - 3k)\right] \tag{83}$$

$$c_3 = \frac{k\sigma^2}{8\alpha^5}\left\{k^3\mathrm{e}^{-3\alpha t} - 3\alpha k\mathrm{e}^{-2\alpha t}[-kt(k - 2\alpha\rho) - 2(\alpha - k\rho)] - 3\mathrm{e}^{-\alpha t}\left[2\alpha kt(k - 2\alpha\rho)^2 + 3...\right.\right. \tag{84}$$

$$c_4 = \frac{3k^2\sigma^2}{64\alpha^7}\left\{-3k^4\mathrm{e}^{-4\alpha t} - 8k^2\mathrm{e}^{-3\alpha t}\left[2\alpha kt(k - 2\alpha\rho) + 4\alpha^2 + k^2 - 6\alpha k\rho\right] - 4\mathrm{e}^{-2\alpha}...\right. \tag{85}$$

I include this here to detail the extensive nature of the implementation of this method - we need to explore a more terse description of cumulants, or moments for that matter. The lack of testing data surrounding the implementation of these functions explicitly for any set of parameters renders

this method useless - and so, we explore the much more tractable moments definition in the Heston Stochastic Volatility model. Using the method of moments, and a simple discretisation of the Heston model described by Dunn et al in "Estimating Option Prices with Heston's Stochastic Volatility Model"[18]. Here an estimation of the moments via their "Method of Moments" technique is discussed. Most notably here the 4^{th} moment is estimated:

$$\begin{aligned}\mu_4 =& \frac{1}{k(k-2)}\left(k^2r^4 + 4k^2r^3 + 6k^2r^2\theta - 2kr^4 + 6k^2r^2 + 12k^2r\theta\right. \\ &+ 3k^2\theta^2 - 8kr^3 - 12kr^2\theta + 4k^2r + 6k^2\theta - 12kr^2 - 24kr\theta \\ &\left.-6k\theta^2 - 3\sigma^2\theta + k^2 - 8kr - 12k\theta - 2k\right)\end{aligned} \tag{86}$$

Note, here, the notation in the derivation aligns as in Heston's derivation [19] and almost Professor Erik Schlogl's Lecture [16], where volatility of volatility: $\sigma = \nu$. Also, the method of moments does provide a derivation up to the seventh moment - with that, the value of ρ does not have huge affects on the value of call and put option prices. Here, we can assume this is going to provide an efficient solution for μ_4 in our model. With this, we only need to make minor changes to 4.2, specifically equation 60.

Parameters:

$$S = 100, \quad v_0 = 0.1^2, \quad r = \ln(1.0005), \quad k = 2.0, \quad \eta = 0.3, \quad \rho = 0.2, \quad \tau = 0.5 \tag{87}$$

Accuracy:

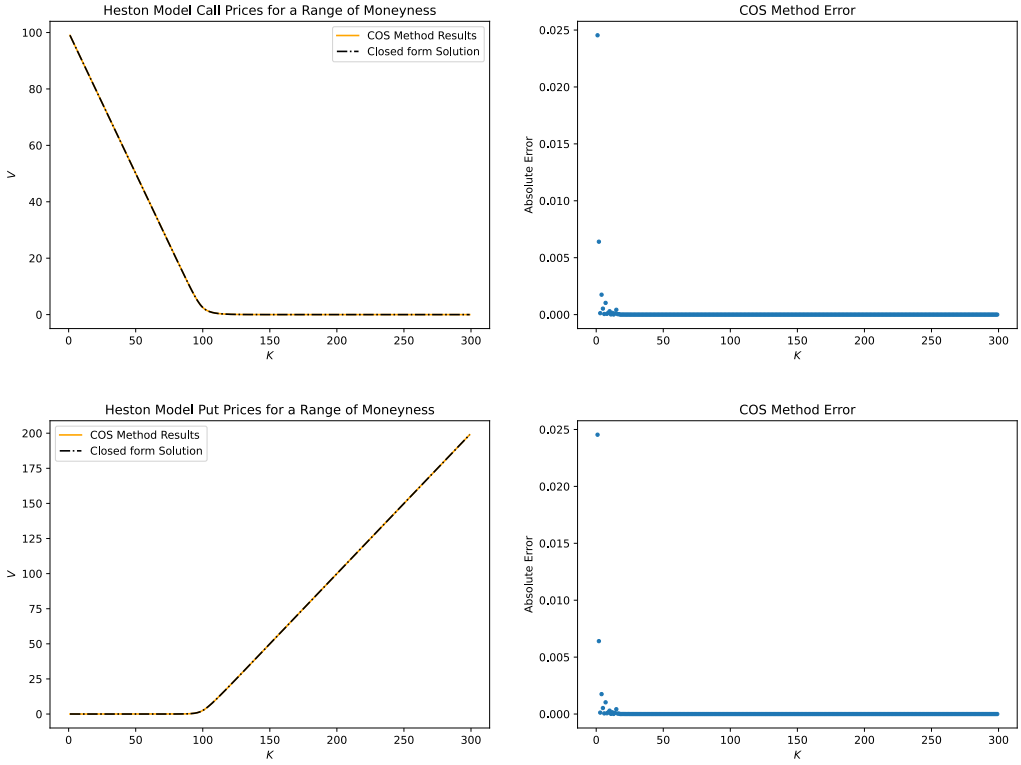

Figure 17: COS Method Heston Results

Speedtest:

$$\text{COS Method: } 27.074234008789062s \quad \text{Closed Form: } 19.883678197860718$$

Python Code: Please see Appendix 6

4.4.4 Variance Gamma

Carr, Madan and Chang's Semi-Closed Form Solution:

$$C(S(0);K,T) = S(0)\Psi\left[d\sqrt{\frac{1-c_1}{v}}, (\varsigma s+s)\sqrt{\left(\frac{1-c_1}{v}\right)^{-1}}, \frac{t}{v}\right] \tag{88}$$

$$-K\exp(-rt)\Psi\left[d\sqrt{\frac{1-c_2}{v}}, (\varsigma s+s)\sqrt{\left(\frac{1-c_2}{v}\right)^{-1}}, \frac{t}{v}\right]$$

With the following definitions:

$$\xi = \frac{\theta}{\sigma^2}, \quad s = \sigma \cdot \left(\sqrt{1 + \frac{\frac{\theta^2 v}{\sigma^2}}{2}} \right)^{-1}, \quad d = \frac{1}{s} \left[\ln \left(\frac{S(0)}{K} \right) + rt + \frac{t}{v} \ln \left(\frac{1 - c_1}{1 - c_2} \right) \right]$$

$$c_1 = \frac{v(\xi s + s)^2}{2}, \quad c_2 = \frac{v(\xi s)^2}{2}$$

Where Ψ is the modified Bessel function:

$$\Psi(\alpha, \beta, \gamma) = \int_0^{+\infty} N \left(\frac{\alpha}{\sqrt{x}} + \beta \sqrt{x} \right) \frac{x^{\gamma - 1} e^{-x}}{\Gamma(\gamma)} dx \tag{89}$$

It is important to note on implementation of this solution I experienced some serious negative bias in the results. I have used the model results from Hirsa [12] to confirm these results: see appendix 6. With this in mind, I refer to a Monte-Carlo solution with $N = 1,000,000$ simulations in order to derive call and put prices in the Variance-Gamma model; with an eye to the convergence of monte-carlo solutions to test accuracy:

$$\left[V_{\text{MC}} - \sqrt{\frac{\sigma^2}{N}},\ V_{\text{MC}} + \sqrt{\frac{\sigma^2}{N}} \right] \tag{90}$$

The Monte-Carlo simulation simulated terminal prices based on:

$$X_t = \theta t + \sigma W_t, \quad T_t \sim \Gamma(t, vt) \quad \Rightarrow \quad X_{T_t} = \theta T_t + \sigma W_{T_t} \tag{91}$$

Where σ : volatility of the brownian motion, v : variance of the gamma process and θ : the drift of the brownian motion as in [20]. Then for simulation of terminal prices with n simulations is:

$$\gamma = \frac{1}{v}, \quad \bar{w} = \gamma \cdot \left(-\ln \left(1 - \theta v - \frac{v}{2} \sigma^2 \right) \right), \quad G^{(n)} = v \Gamma(\gamma T)^{(n)} \tag{92}$$

Then let $Z^{(n)} \sim \mathcal{N}(0, 1)$ be the vector of random normal variables, then simulation of n terminal prices is as follows:

$$S_T^{(n)} = S_0 \text{exp} \left((r - \bar{w})T + \theta G^{(n)} + \sigma \sqrt{G^{(n)}} Z^{(n)} \right)$$

And with then the simulation of call prices is:

$$\frac{1}{n} \sum_n \max\{S_T^{(n)} - K, 0\}$$

Characteristic Function [14]:

$$\varphi_X(u, t) = \exp(iu\mu t) \cdot \varphi_{\text{VG}}(u, t) \tag{93}$$

$$\varphi_{\text{VG}}(u, t) = \left(1 - iu\theta v + \frac{1}{2} \sigma^2 v u^2 \right)^{-t/v}$$

$$\mu = r - q + \bar{\omega}$$

$$\bar{\omega} = (1/v) \cdot \ln\left(1 - \theta v - \frac{1}{2}\sigma^2 v\right)$$

Cumulants:

$$c_1 = (r - q - \bar{\omega} + \theta)t \tag{94}$$

$$c_2 = \left(\sigma^2 + v\theta^2\right) t$$

$$c_3 = t\left(2\theta^3 v^2 + 3\sigma^2\theta v\right)$$

$$c_4 = 3\left(\sigma^4 v + 2\theta^4 v^3 + 4\sigma^2\theta^2 v^2\right) t$$

Parameters:

$$S = 100, \quad T = 1/12, \quad r = 0.1, \quad v = 0.2, \quad \sigma = 0.12, \quad \theta = -0.14 \tag{95}$$

Accuracy:

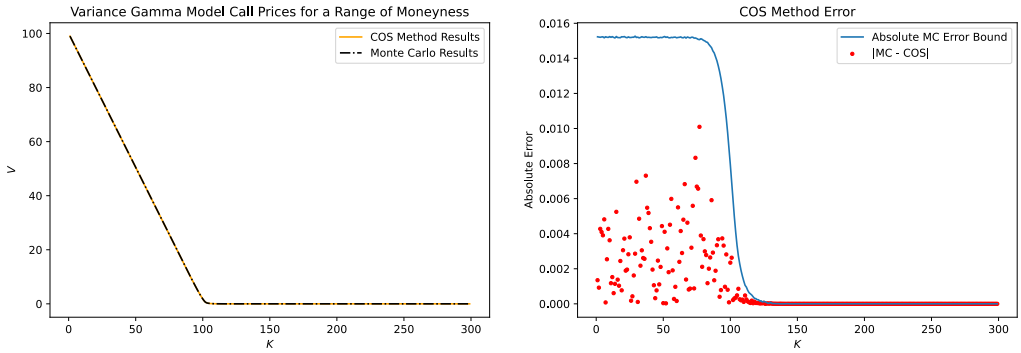

Figure 18: COS Method VG Results

In figure 18 above, we only display the call prices. This is because in my MC procedure I also use put call parity to define put prices. Meaning the relationship with errors in put price approximations is going to be identical. With this being said, it is clear that the COS method implementation is sufficient (using $1,000,000$ simulations in Monte-Carlo), with errors lying in the error bound of the Monte-Carlo Simulation.

Speedtest:

COS Method: $1.0007538795471191s$

I do not report a speedtest for the Monte-Carlo simulation, as for obvious reasons it is going to be considerably larger than the COS method approximation.

Python Code: Please see Appendix 6

5 The COS_Speedy Function

The results from the speedtest's above in 4.4 for each stock price model lend itself to a new computational implementation of the COS method - I denote this the `COS_speedy` function. The aim of this function is to use the COS method in these same models, but make a real attempt at reducing the speed, attempting to make a reasonable trade-off between speed and accuracy. To do this we employ the following solution: remove z in 61. Set N, the number of iterations to be fixed initially at 2^8, we can then increase this value heuristically. Then vectorise the the strikes K parameter using `numpy`. What this means is that we will have to call the function only once. The obvious outstanding problem is then the truncation range, the magnitude of which grows at a rate of $\sqrt[4]{K}$. The solution to this is straightforward, what we do is we take the highest value of K and define the truncation range using this maximum K.

The expected results are divergence in the ATM and ITM options where the range is less sufficient, in particular, too wide to ensure convergence. The idea is that we can increase N such that the we get some "acceptable" level of convergence. I hypothesise that although this is a less robust approach, the vecotorisation of the strikes parameter is going to be fast enough such that the increasing iterations provides a more favourable speed trade-off that of calling the function for each strike.

Following this, we want to remove information from the calculation that could be calculated prior to the initiation of the COS method; namely μ_4, M and $K_{\max}$, as well as the put/call input function. Here we assume the calculation of call prices post use of the `COS_Speedy` method to be a straightforward operation, which can also be done quickly using vectorised code. `COS_Speedy` function:

```
def COS_speedy(cf,S0,r,T,K,M,N):

    # Setting some constants for the summation
    i = np.complex(0.0,1.0)
    y = np.log(S0 / K)

    # Setting the truncation domain [-M,M]
    a = - M
    b = M

    # Create a column vector of the (in fang and osterlee's notation) v_{k}^{M}
    k = np.linspace(0,N-1,N).reshape([N,1])
    u = k * np.pi / (b - a);

    # Grab the put coefficients based on this truncation domain
    H_k = Hk(a,b,k)

    # Now doing the sum
    sumation = cf(u) * H_k
        # Halfing the first value (sum ')
    sumation[0] = 0.5 * sumation[0]

    # Outer product
```

```
outerproduct = np.exp(i * np.outer((y - a) , u))

# Final result, discount and *K * real part of the dot product of the summation and
    the outerproduct variable
output = np.exp(-r * T) * K * np.real(outerproduct.dot(sumation))

# Implenting the condition for the forward
Forwards = float(S0 * np.exp(-r*T))
ftest = np.log(K/Forwards)

test = np.where(ftest < a, 0, output)

# Return this output
return test
```

5.1 Results

Using the same parameters as described in 4.4 I complete testing across the relevant advanced stock price models.

5.1.1 Geometric Brownian Motion

Accuracy:

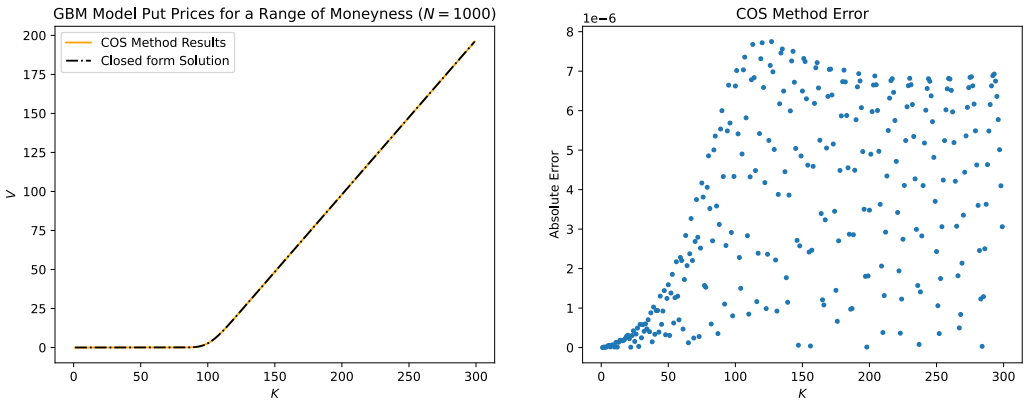

Figure 19: GBM `COS_speedy` Results

In the above, it took $N = 1,000$ to achieve an acceptable level or error.

Speedtest:

COS Speedy Method: $0.029228925704956055s$, Closed Form: $0.048547983169555664s$

Python Code: Please see Appendix 6

5.1.2 Merton Jump Diffusion

Accuracy:

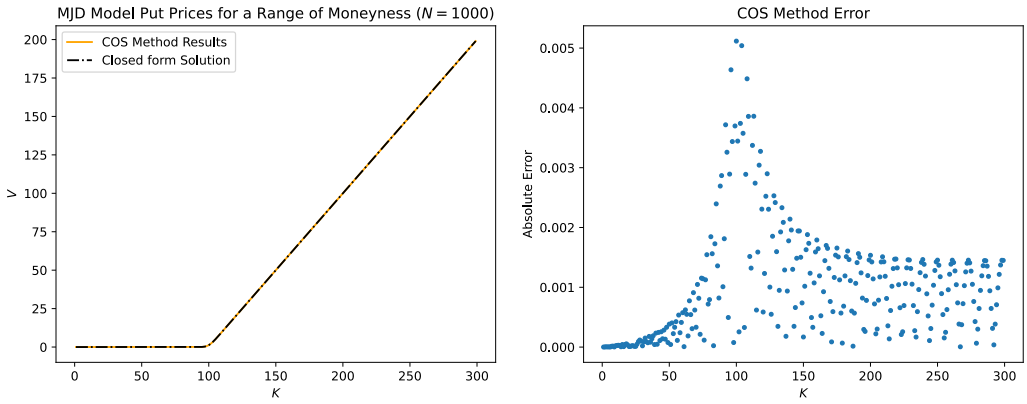

Figure 20: MJD `COS_speedy` Results

Again here, using $N = 1000$ to find "acceptable" error. Admittedly here this is not conservative.

Speedtest:

COS Speedy Method: $0.01833510398864746s$ Closed Form: $4.238032817840576s$

Python Code: Please See Appendix 6

5.1.3 Heston Stochastic Volatility

Accuracy:

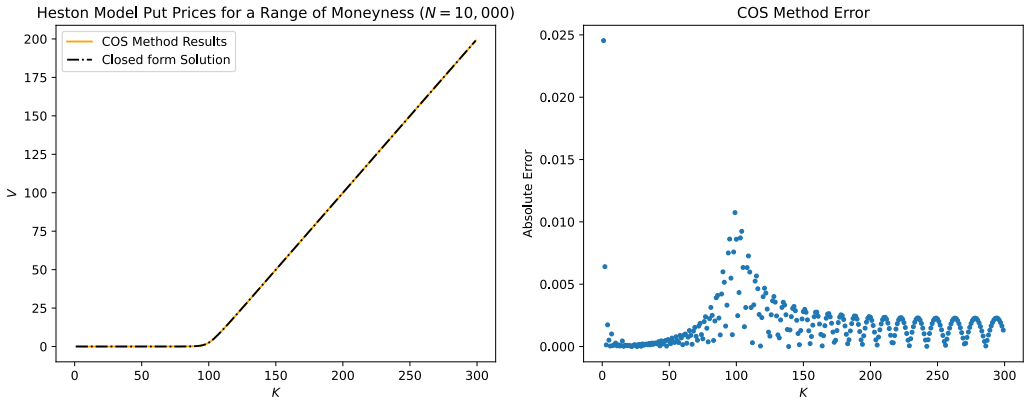

Figure 21: Heston `COS_speedy` Results

Note, it is considerably harder to obtain convergence here - choosing $N = 10,000$ does the job.

Speedtest:

COS Speedy Method: $0.10463833808898926s$ Closed Form: $19.8371798992157s$

5.1.4 Variance Gamma:

Python Code: Please See Appendix 6 **Accuracy:**

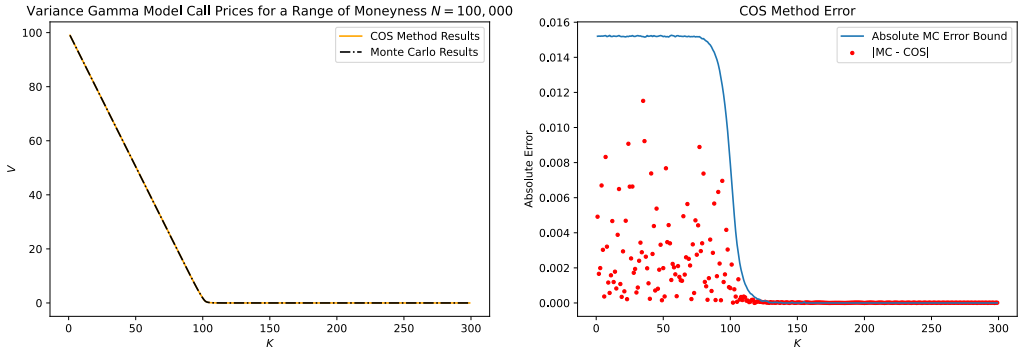

Figure 22: Variance Gamma `COS_speedy` Results

This experiment required $N = 100,000$ in order for suitable convergence, i.e. below the Monte Carlo absolute error bound. Even though there are still large errors here I discuss the root of these errors in the discussion in section 6. Here, we may be getting the correct price, but are hamstrung by the choice of Monte-Carlo method to verify price.

Speedtest:

COS Speedy Method: $1.540647029876709s$

Python Code: Please See Appendix 6

6 Conclusion

For the robust method, the COS function that takes just one strike at a time and calls the function incrementally, our results for all models were positive, providing results that were well within acceptable levels of error. With the Geometric Brownian Motion and Merton Jump Diffusion models, I don't believe this requires further discussion.

This is not the case with Heston Stochastic Volatility and Variance Gamma models. In Variance-Gamma we could only use the Monte-Carlo simulation with the number of simulations set to $n = 1,000,000$. Anymore than this meant looping through the strikes in python would take too long. We know that the convergence of MC simulations of this type have a convergence rate $\frac{1}{\sqrt{N}}$,

so here $1,000,000$ is a reasonable choice. But with that, results were achieved that existed within the error bounds, I have also verified these results with the work of Hirsa [12], see appendix 6. Even with this it does not give as much credence to our COS method results as was desired - the introduction of variance reduction techniques and potentially some PIDE approaches to price options in Variance-Gamma such as in [20] was regretfully not explored here. Moving to Heston Stochastic Volatility, we do get an error arising right at the bottom end of the strikes. A quick investigation into this error gives us the following results from the for the semi-closed form solution:

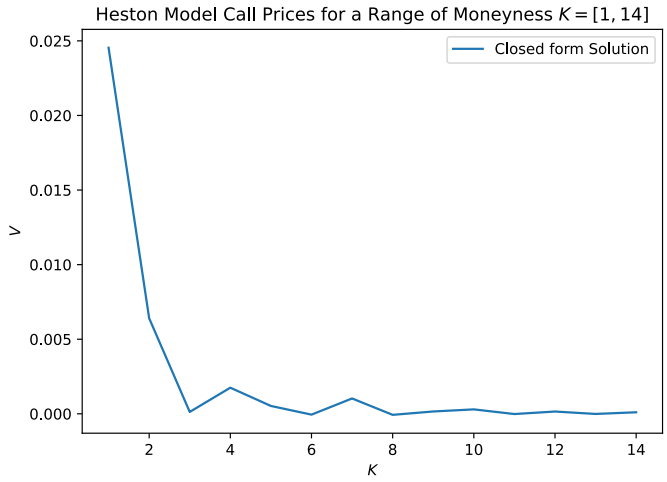

Figure 23: Heston Closed Put Prices at Very OTM Strikes

Above, in figure 23 you can see the closed form solution becomes unstable at these low strikes, providing further credence for the correct solution being obtained via the robust COS method. With this, I do believe the primary goal of this research piece has been achieved, a robust COS method that fixes at the money divergence in Levy processes with jumps, and fixes the problem of pricing options across all levels of moneyness.

The secondary goal was to attempt to maintain one of the primary benefits of the COS method, its speed. In doing so I found that vectorising the strikes parameter did indeed speed up the method enough such that we could simply choose the number of iterations heuristically for the model, and using: $\varepsilon = 0$, $\lim_{N\to\infty}$, choosing extremely large values of N, i.e. $10,000 - 20,000$, it was still quicker than calling the formula separately each time. Here, N is inversely proportional to individual error tolerance.

It follows that our error is going to spike at the options closer to the money here, as we have picked the truncation range to be as far in the money as possible. Because these options are typically the most liquid, and thus' most commonly used for model calibration, this method should be used with additional prudence in regards to selection of N. With that being said, in model calibration using these liquid options, the `max_strike` variable is going to be a lot closer to the spot price, and so this error will not be nearly as large here. I have demonstrated this below in figure

24 in the Heston Stochastic Volatility model - where the robust COS and COS speedy methods appeared to be slowest. With strikes $K = [S - 15, S + 15]$ and other parameters as stated in 87; for $N = 30,000$ the results where as follows:

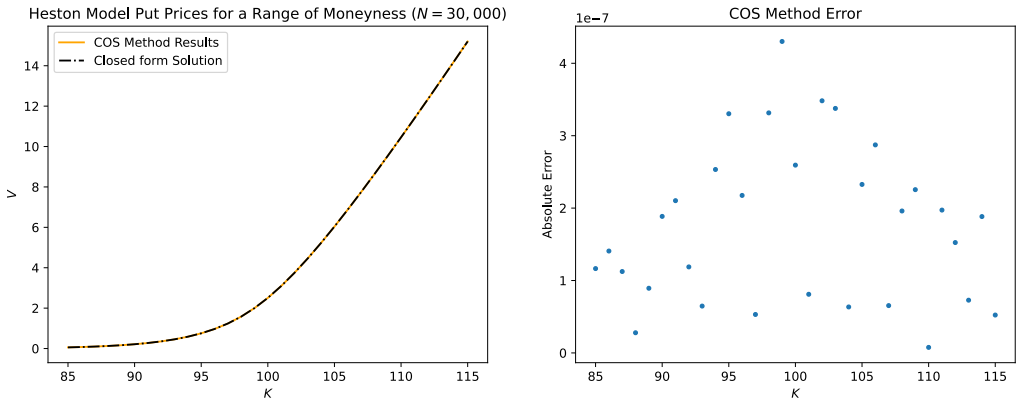

Figure 24: COS Speedy Error with Lower `max_strike`

Here we have errors in $1e^{-7}$, fairly suitable for a very fast solution. The speedtest results for this level of N were:

$$\text{COS Speedy Method: } 0.055632829666137695s \quad \text{Closed Form: } 0.6702420711517334s$$

Note, it took $N \approx 400,000$ for the speed to be close to that of the Closed Form Solution. Here the errors are ≈ 0 for all strikes bar $K = 94$ where we received an error of $\approx 2.75e^{-10}$ for reasons I am yet to investigate fully. In the Variance-Gamma COS speedy test it took a very large N to ensure convergence to a result that was within the Monte-Carlo error bounds. Note, this was a shortfall of my method of testing, and not the COS speedy method itself. It can be said that in all the above experiments we have implemented the standard solution, we haven't used vectorised code in the COS method and COS speedy method. This does limit comparability, but that being said; overall I believe showing the additional speed of vectorisation of the strikes did provide an applicable solution to maintaining the speed and accuracy.

References

[1] "3.10.10 Documentation."

[2] "NumPy documentation — NumPy v1.22 Manual."

[3] "time — Time access and conversions — Python 3.10.10 documentation."

[4] "warnings — Warning control — Python 3.10.10 documentation."

[5] "SciPy 1.8.0 Release Notes — SciPy v1.10.0 Manual."

[6] "math — Mathematical functions — Python 3.10.10 documentation."

[7] "What's new in Matplotlib 3.5.0 (Nov 15, 2021) — Matplotlib 3.7.0 documentation."

[8] "Summary of New Features in 13.1—Wolfram Language Documentation."

[9] G. Junike and K. Pankrashkin, "Precise option pricing by the COS method–How to choose the truncation range," *Applied Mathematics and Computation*, vol. 421, p. 126935, May 2022. arXiv:2109.01030 [q-fin].

[10] F. Fang and C. W. Oosterlee, "A Novel Pricing Method for European Options Based on Fourier-Cosine Series Expansions," *SIAM Journal on Scientific Computing*, vol. 31, pp. 826–848, Jan. 2009.

[11] S. Erik, "Chapters 12 & 13: Jump-Diffusion Processes."

[12] A. Hirsa, *Computational methods in finance*. Chapman & Hall/CRC Financial mathematics series, Boca Raton, FL: CRC Press, pp 54-78, 2013.

[13] F. L. Floc'h, "More Robust Pricing of European Options Based on Fourier Cosine Series Expansions," June 2020. arXiv:2005.13248 [q-fin].

[14] C. W. Oosterlee and L. A. Grzelak, *Mathematical modeling and computation in finance: with exercises and python and matlab computer codes*. Hackensack: World Scientific Publishing Co. Pte. Ltd, 1 ed., pp. 163-182, 2019.

[15] "Cumulants," July 2017.

[16] S. Erik, "Chapter 15: Stochastic Volatility."

[17] G. Bormetti, V. Cazzola, G. Livan, G. Montagna, and O. Nicrosini, "A generalized Fourier transform approach to risk measures," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2010, p. P01005, Jan. 2010.

[18] R. Dunn, P. Hauser, T. Seibold, and H. Gong, "Estimating Option Prices with Heston's Stochastic Volatility Model," article, Kenyon College, The College of New Jersey, Western Kentucky University, Valparaiso University.

[19] S. L. Heston, "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options," *Review of Financial Studies*, vol. 6, pp. 327–343, Apr. 1993.

[20] N. Cantarutti, "An Introduction to Levy processes and PIDE's," Mar. 2019.

Appendix

Hirsa's Variance Gamma Results [12]

K	Analytical	COS	FrFFT	FFT	SP
10	90.0832	95.1859	90.0830	90.0983	90.0460
20	80.1660	80.8759	80.1660	80.1687	80.0835
30	70.2490	70.2490	70.2490	70.2388	70.1157
40	60.3320	60.3319	60.3319	60.3303	60.1560
50	50.4149	50.4149	50.4149	50.4010	50.2120
60	40.4979	40.4979	40.4979	40.4869	40.2881
70	30.5813	30.5813	30.5813	30.5969	30.3879
80	20.6702	20.6704	20.6704	20.6617	20.5189
90	10.8289	10.8289	10.8289	10.7983	10.7156
100	1.8150	1.8150	1.8151	1.7913	1.5406
110	0.0195	$1.94e-02$	$1.95e-02$	$5.29e-02$	$2.26e-02$
120	$6.9339e-04$	$5.38e-04$	$5.83e-04$	$1.15e-02$	$6.57e-04$
130	$2.7159e-05$	$-1.08e-06$	$2.56e-05$	$-3.43e-03$	$2.85e-05$
140	$5.7237e-06$	$-1.25e-04$	$1.89e-07$	$-6.44e-03$	$1.63e-06$
150	$3.90e-08$	$-3.71e-04$	$-1.73e-06$	$2.86e-03$	$1.16e-07$
160	$2.14e-09$	$-1.53e-03$	$1.45e-06$	$1.47e-03$	$1.00e-08$
170	$0.00e+00$	$-5.19e-03$	$1.37e-06$	$-2.84e-03$	$1.01e-09$
180	$0.00e+00$	$-1.73e-02$	$-3.27e-07$	$2.61e-03$	$1.16e-10$
190	$0.00e+00$	$-5.47e-02$	$7.88e-07$	$-2.05e-03$	$1.51e-11$
200	$0.00e+00$	$-1.68e-01$	$-7.31e-07$	$1.65e-03$	$2.19e-12$

COS Method Python Implementation for Various Stock Price Models

Geometric Brownian Motion

```
# Closed form solution for Black Scholes put and call prices

N = norm.cdf

def BS_CALL(S, K, T, r, sigma):
    d1 = (np.log(S/K) + (r + sigma**2/2)*T) / (sigma*np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    return S * N(d1) - K * np.exp(-r*T)* N(d2)

def BS_PUT(S, K, T, r, sigma):
    d1 = (np.log(S/K) + (r + sigma**2/2)*T) / (sigma*np.sqrt(T))
    d2 = d1 - sigma* np.sqrt(T)
    return K*np.exp(-r*T)*N(-d2) - S*N(-d1)

# Cumulants in GBM

def cumulants_GBM(r,sgm,T):
    return r - 0.5 * ( sgm**2 ) * T, ( sgm**2 ) * T, 0, 0
```

```
# Moments in GBM

def moment_func_GBM(r,sgm,T):
    mus = cumulants_GBM(r,sgm,T)
    return mus[3] + 4 * mus[2] * mus[0] + 3 * ( mus[1]**2 ) + 5 * mus[1] * ( mus[0]**2 )
        + ( mus[3]**4 )

# Implementing the COS Method in GBM

# Set i to be complex 1 real 0
i=np.complex(0.0,1.0)

# Set testing parameters
S0 = 100.0
r = 0.1
T = 0.1
sgm = 0.25
CP = "put"

# Define characteristic function
cf = lambda u: np.exp((r - 0.5 * np.power(sgm,2.0)) * i * u * T - 0.5
                      * np.power(sgm, 2.0) * np.power(u, 2.0) * T)

cos_vals_puts = []
bs_vals_puts = []
errors_puts = []

# Loop accross a range of strikes
for K_ in range(1,300):
    # Find the 4th moment
    mu_4 = moment_func_GBM(r,T,sgm)

    # Value using COS
    cosPrice = COS(CP,cf,S0,r,T,K_,mu_4)
    cos_vals_puts.append(cosPrice)

    # Valueing via black and scholes
    bsPrice = BS_PUT(S0,K_,T,r,sgm)
    bs_vals_puts.append(bsPrice)

    # Find the error
    error = abs(bsPrice - cosPrice)
    errors_puts.append(error)

# Redefine CP
CP = "call"

cos_vals_calls = []
bs_vals_calls = []
errors_calls = []
```

```
# Loop accross a range of strikes
for K_ in range(1,300):
    # Find the 4th moment
    mu_4 = moment_func_GBM(r,T,sgm)

    # Value using COS
    cosPrice = COS(CP,cf,S0,r,T,K_,mu_4)
    cos_vals_calls.append(cosPrice)

    # Valueing via black and scholes
    bsPrice = BS_CALL(S0,K_,T,r,sgm)
    bs_vals_calls.append(bsPrice)

    # Find the error
    error = abs(bsPrice - cosPrice)
    errors_calls.append(error)
```

Merton Jump Diffusion

```
# Closed form solution for Puts and Calls

def mjd_PUT(S,K,r,sgm,k,v,lam,T):
    lamd = lam*(1+k)
    gamma = np.log(1 + k)
    p = 0
    for n in range(0,99):
        coef = (np.exp(-lamd*T) * ((lamd*T)**n)) / math.factorial(n)
        v_n_2 = np.sqrt((sgm**2) + (n*(v**2)/T))
        r_n = r - lam*(k) + (n * gamma)/T
        BS = BS_PUT(S,K,T,r_n, v_n_2)
        p += coef*BS
    return p

def mjd_CALL(S,K,r,sgm,k,v,lam,T):
    lamd = lam*(1+k)
    gamma = np.log(1 + k)
    p = 0
    for n in range(0,99):
        coef = (np.exp(-lamd*T) * ((lamd*T)**n)) / math.factorial(n)
        v_n_2 = np.sqrt((sgm**2) + (n*(v**2)/T))
        r_n = r - lam*(k) + (n * gamma)/T
        BS = BS_CALL(S,K,T,r_n, v_n_2)
        p += coef*BS
    return p

# Cumulants in MJD

def cumulants_MJD(k,sgm,lam,r,T,delta):
    muj = np.log(k + 1) - 0.5 * ( delta**2 )
```

```
    w_hat = lam * ( np.exp ( 0.5 * ( delta**2 ) + mu_j ) -1 )
    mu = r - ( sgm**2 ) - w_hat
    c1 = T * (r - w_hat - 0.5*(sgm**2) + lam*mu_j)
    c2 = T * (sgm**2 + lam * ( delta**2 ) + lam * ( delta**2 ) )
    c3 = T * lam * (3*mu_j*(delta**2) + (mu_j**3))
    c4 = T * lam * ( ( mu_j**4 ) + 6 * ( delta**2 ) * ( mu_j**2 ) + 3 * ( delta**4 ) *
        lam )
    return c1, c2, c3, c4

# Moments in MJD

def moment_func_MJD(k,sgm,lam,r,T,sgm_j):
    mus = cumulants_MJD(k,sgm,lam,r,T,sgm_j)
    return mus[3] + 4 * mus[2] * mus[0] + 3 *( mus[1]**2 ) + 5 * mus[1] * ( mus[0]**2 )
        + ( mus[3]**4 )

## Implementing the COS method in MJD

CP = "put"
S0 = 100
r = 0
T = 0.1
sgm = 0.1
lam = 0.001
kappa = -0.5
delta = 0.2

# Find model variables
mu_j = np.log(kappa + 1) - 0.5*(delta**2)
# Mu used in charteristic function below
mu_c = r - ((sgm**2)/2) - lam*(np.exp(0.5*(delta**2) + mu_j) -1)

# Characteristic function
cf = lambda u: np.exp((i * u * mu_c * T) - (0.5 * (sgm**2) * (u**2) * T)) * \
           np.exp(lam * T * (np.exp((i * mu_j * u) - (0.5* (delta**2) * (u**2))) -1))

# Find the 4th moment
mu_4 = moment_func_MJD(kappa,sgm,lam,r,T,delta)

cos_vals_puts = []
bs_vals_puts = []
errors_puts = []

# Loop accross a range of strikes
for K_ in range(1,300):
    # Find the 4th moment
    mu_4 = moment_func_MJD(kappa,sgm,lam,r,T,delta)

    # Value using COS
    cosPrice = COS("put",cf,S0,r,T,K_,mu_4)
    cos_vals_puts.append(cosPrice)
```

```
    # Valueing via black and scholes
    bsPrice = mjd_PUT(S0,K_,r,sgm,kappa,delta,lam,T)
    bs_vals_puts.append(bsPrice)

    # Find the error
    error = abs(bsPrice - cosPrice)
    errors_puts.append(error)

# Redefine CP
CP = "call"

cos_vals_calls = []
bs_vals_calls = []
errors_calls = []

# Loop accross a range of strikes
for K_ in range(1,300):
    # Find the 4th moment
    mu_4 = moment_func_MJD(kappa,sgm,lam,r,T,delta)

    # Value using COS
    cosPrice = COS("call",cf,S0,r,T,K_,mu_4)
    cos_vals_calls.append(cosPrice)

    # Valueing via black and scholes
    bsPrice = mjd_CALL(S0,K_,r,sgm,kappa,delta,lam,T)
    bs_vals_calls.append(bsPrice)

    # Find the error
    error = abs(bsPrice - cosPrice)
    errors_calls.append(error)
```

Heston Stochastic Volatility

Closed form solution from [16] associated code. Characteristic function from [14].

```
# Heston Closed Form Solution

class Heston:
    def __init__(self,S0,v0,r,k,theta,eta,rho):
        self.S0 = S0
        self.v0 = v0
        self.r = r
        self.k = k
        self.theta = theta
        self.eta = eta
        self.rho = rho
        self.gamma = eta**2/2.0
    def P(self,j,x,tau):
```

```
        return 0.5+1.0/np.pi*(quad(self.integrand,0.0,np.inf,args=(j,x,self.v0,tau)))[0]
    def integrand(self,u,j,x,v,tau):
        return np.real(np.exp(self.C(j,u,tau)*self.theta+self.D(j,u,tau)*v+1j*u*x)/(u*1j))
    def C(self,j,u,tau):
        g = self.rminus(j,u)/self.rplus(j,u)
        return
            self.k*(self.rminus(j,u)*tau-2.0/self.eta**2*np.log((1.0-g*np.exp(-self.d(j,u)*tau))/(1.0-g)
    def d(self,j,u):
        return np.sqrt(self.beta(j,u)**2-4*self.alpha(j,u)*self.gamma)
    def rminus(self,j,u):
        return (self.beta(j,u) - self.d(j,u))/(2*self.gamma)
    def rplus(self,j,u):
        return (self.beta(j,u) + self.d(j,u))/(2*self.gamma)
    def beta(self,j,u):
        return self.k-self.rho*self.eta*j-self.rho*self.eta*u*1j
    def alpha(self,j,u):
        return -u**2/2-u*1j/2+j*u*1j
    def D(self,j,u,tau):
        g = self.rminus(j,u)/self.rplus(j,u)
        return
            self.rminus(j,u)*(1.0-np.exp(-self.d(j,u)*tau))/(1.0-g*np.exp(-self.d(j,u)*tau))
    def callprice(self,strike,tau):
        B = np.exp(-self.r*tau)
        F = self.S0/B
        x = np.log(F/strike)
        return B*(F*self.P(1,x,tau)-strike*self.P(0,x,tau))

def ChFHestonModel(r,tau,kappa,gamma,vbar,v0,rho):
    i = np.complex(0.0,1.0)
    D1 = lambda u: np.sqrt(np.power(kappa-gamma*rho*i*u,2)+(u*u+i*u)*gamma*gamma)
    g = lambda u: (kappa-gamma*rho*i*u-D1(u))/(kappa-gamma*rho*i*u+D1(u))
    C = lambda u: (1.0-np.exp(-D1(u)*tau))/(gamma*gamma*(1.0-g(u)*np.exp(-D1(u)*tau)))\
                  *(kappa-gamma*rho*i*u-D1(u))
    A = lambda u: r * i*u *tau + kappa*vbar*tau/gamma/gamma *(kappa-gamma*rho*i*u-D1(u))\
              - 2*kappa*vbar/gamma/gamma*np.log((1.0-g(u)*np.exp(-D1(u)*tau))/(1.0-g(u)))
    # Characteristic function for the Heston model
    cf = lambda u: np.exp(A(u) + C(u)*v0)
    return cf

# Defining the 4th moment in Heston

def Heston_mu4(k,r,theta,eta):
    first = 1/(k*(k-2.0))
    second = (k**2)*(r**4) + 4*(k**3)*(r**3) + 6*(k**2)*(r**2)*theta - 2*(k**2)*(r**2) +
        12*(k**2)*r*theta
    third = 3*(k**2)*(theta*2) - 8*(k**1)*(r**3) - 12*(k**1)*(r**2)*theta + 4*(k**2)*r +
        6*(k**2)*theta - 12*(k**1)*(r**2) - 24*k*r*theta
    fourth = -6*(k**1)*(theta**2) - 3*(eta**2)*(theta**1) + (k**2) - 8*(k**1)*(r**1) -
        12*(k**1)*(theta) - 2*(k**1)
    return first*(second + third + fourth)
```

```
# The COS Method for Heston Stochastic Volatility

# Set i to be complex 1 real 0
i=np.complex(0.0,1.0)

S0 = 100.0
v0 = 0.1**2
theta = 0.1**2
r = np.log(1.0005)
k = 1.0
eta = 0.3
rho = 0.2
K = [100]
tau = 0.5
CP = "call"

# Define characteristic function
cf = ChFHestonModel(r,tau,k,eta,theta,v0,rho)

# Find the 4th moment
mu_4 = Heston_mu4(k,r,theta,eta)

# Define Heston Closed Form
hestonmodel = Heston(S0,v0,r,k,theta,eta,rho)

cos_vals_puts = []
heston_vals_puts = []
errors_puts = []

# Loop accross a range of strikes
for K_ in range(1,300):
    # Find the 4th moment

    # Value using COS
    cosPrice = COS("put",cf,S0,r,tau,K_,mu_4)
    cos_vals_puts.append(cosPrice)

    # Valueing via Heston
    hs_price_put = hestonmodel.callprice(K_,tau) + K_*np.exp(-r*tau) - S0
    heston_vals_puts.append(hs_price_put)

    # Find the error
    error = abs(hs_price_put - cosPrice)
    errors_puts.append(error)

# Redefine CP
CP = "call"

cos_vals_calls = []
heston_vals_calls = []
errors_calls = []
```

```
# Loop accross a range of strikes
for K_ in range(1,300):

    # Value using COS
    cosPrice = COS("call",cf,S0,r,tau,K_,mu_4)
    cos_vals_calls.append(cosPrice)

    # Valueing via Heston
    hs_price_call = hestonmodel.callprice(K_,tau)
    heston_vals_calls.append(hs_price_call)

    # Find the error
    error = abs(hs_price_call - cosPrice)
    errors_calls.append(error)
```

Variance Gamma

```
# Cumulants in Variance Gamma

def VG_cumulants(r,t,v,theta,sigma):
    wbar = 1/v * np.log(1 - theta*v * - 0.5*(sigma**2)*v)
    c1 = (r - wbar + theta)*t
    c2 = t*((sigma**2) + kappa*(theta**2))
    c3 = t*(2*(theta**3)*(v**2) + 3*(sigma**2)*theta*v)
    c4 = 3*( (sigma**4)*v + 2*(theta**4)*(v**3) + 4*(sigma**2)*(theta**2)*(v**2) )*t
    return c1, c2, c3, c4

# Moments function in Variance Gamma

def VG_mu4(r,t,v,theta,sigma):
    mus = VG_cumulants(r,t,v,theta,sigma)
    return mus[3] + 4 * mus[2] * mus[0] + 3 *( mus[1]**2 ) + 5 * mus[1] * ( mus[0]**2 )
        + ( mus[3]**4 )

# Variance Gamma Characteristic Function

def ChFVG(t,r,v,theta,sigma,S0):
    i = np.complex(0.0,1.0)
    omega = 1/v * np.log(1.0-v*theta-0.5*v*sigma*sigma)
    mu = r + omega
    cf = lambda u: np.exp(i*u*mu*t)*np.power(1.0-i*v*theta*u+0.5*v*sigma*sigma*u*u,-t/v)
    return cf

def VG_MonteCarlo(N,S0,K,v,theta,sigma,r,T):
    y = 1 / v
    w = -np.log(1 - (theta * v) - (v/2 * sigma**2)) /v
    G = ss.gamma(y * T).rvs(N) / y
    normRVs = ss.norm.rvs(0,1,N)
    S_T = S0 * np.exp( ((r-w)*T) + (theta * G) + (sigma * np.sqrt(G) * normRVs))
```

```
    call_price = np.exp(-r*T) * scp.mean( np.maximum(S_T-K,0) )
    bound_C = 2*(scp.std(np.maximum(S_T - K, 0)))/np.sqrt(N)
    return call_price, bound_C

# Implementing the COS Method

S0 = 100
T = 1/12
r = 0.1
kappa = 0.2
sigma = 0.12
theta = -0.14

# Characteristic function
cf = ChFVG(T,r,kappa,theta,sigma,S0)

# Find the 4th moment
mu_4 = VG_mu4(r,T,kappa,theta,sigma)

# Create a set of empty lists
MC_Price = []
lower_c = []
upper_c = []
cosPriceVG_call = []

# Loop over the strikes, find COS Method Price and MC Price in given params above for
    call prices only
for K_ in range(1,300):
    c, bc = VG_MonteCarlo(1_000_000,100,K_,kappa,theta,sigma,r,T)
    MC_Price.append(c)
    lower_c.append(c - bc)
    upper_c.append(c + bc)
    cosPriceVG_call.append(COS("call",cf,S0,r,T,K_,mu_4))
```

COS Speedy Implementation in Various Stock Price Models

Geometric Brownain Motion

```
# COS Speedy in GBM

# Set i to be complex 1 real 0
i=np.complex(0.0,1.0)

# Set testing parameters
S0 = 100.0
r = 0.1
T = 0.1
sgm = 0.25
CP = "put"
```

```
# Define characteristic function
cf = lambda u: np.exp((r - 0.5 * np.power(sgm,2.0)) * i * u * T - 0.5
                      * np.power(sgm, 2.0) * np.power(u, 2.0) * T)

# Taking out pre-calculatable variables from the cos method
K = range(1,300)
K_ = np.array(K).reshape([len(K),1])
max_strike = 299
mu4 = moment_func_GBM(r,T,sgm)
M = np.power((2*max_strike*mu_4)/(10**-7), (1/4))

# Speedtest
start = time.time()
x = (COS_speedy(cf,S0,r,T,K_, M, int(1000)))
end = time.time()
print(end-start)

test = []
start = time.time()
for K_ in range(1,300):
    test.append(BS_PUT(S0,K_,T,r,sgm))
end = time.time()
print(end-start)
```

Merton Jump Diffusion

```
## Implementing the COS speedy method in MJD

i = np.complex(0.0, 1.0)

S0 = 100
r = 0
T = 0.1
sgm = 0.1
lam = 0.001
kappa = -0.5
delta = 0.2

# Find model variables
mu_j = np.log(kappa + 1) - 0.5*(delta**2)
# Mu used in charteristic function below
mu_c = r - ((sgm**2)/2) - lam*(np.exp(0.5*(delta**2) + mu_j) -1)

# Characteristic function
cf = lambda u: np.exp((i * u * mu_c * T) - (0.5 * (sgm**2) * (u**2) * T)) * \
          np.exp(lam * T * (np.exp((i * mu_j * u) - (0.5* (delta**2) * (u**2))) -1))

# Taking out pre-calculatable variables from the cos method
K = range(1,300)
K_ = np.array(K).reshape([len(K),1])
```

```
max_strike = 299
mu_4 = moment_func_MJD(kappa,sgm,lam,r,T,delta)
M = np.power((2*max_strike*mu_4)/(10**-7), (1/4))

# Speedtest
start = time.time()
x = (COS_speedy(cf,S0,r,T,K_, M, int(1000)))
end = time.time()
print(end-start)

test = []
start = time.time()
for K_ in range(1,300):
    test.append(mjd_PUT(S0,K_,r,sgm,kappa,delta,lam,T))
end = time.time()
print(end-start)
```

Heston Stochastic Volatility

```
## Implementing the COS speedy method in MJD

i = np.complex(0.0, 1.0)

S0 = 100
r = 0
T = 0.1
sgm = 0.1
lam = 0.001
kappa = -0.5
delta = 0.2

# Find model variables
mu_j = np.log(kappa + 1) - 0.5*(delta**2)
# Mu used in charteristic function below
mu_c = r - ((sgm**2)/2) - lam*(np.exp(0.5*(delta**2) + mu_j) -1)

# Characteristic function
cf = lambda u: np.exp((i * u * mu_c * T) - (0.5 * (sgm**2) * (u**2) * T)) * \
           np.exp(lam * T * (np.exp((i * mu_j * u) - (0.5* (delta**2) * (u**2))) -1))

# Taking out pre-calculatable variables from the cos method
K = range(1,300)
K_ = np.array(K).reshape([len(K),1])
max_strike = 299
mu_4 = moment_func_MJD(kappa,sgm,lam,r,T,delta)
M = np.power((2*max_strike*mu_4)/(10**-7), (1/4))

# Speedtest
start = time.time()
x = (COS_speedy(cf,S0,r,T,K_, M, int(1000)))
```

```
end = time.time()
print(end-start)

test = []
start = time.time()
for K_ in range(1,300):
    test.append(mjd_PUT(S0,K_,r,sgm,kappa,delta,lam,T))
end = time.time()
print(end-start)
```

Variance Gamma

```
# Implementing the COS speedy method in variance gamma

S0 = 100
T = 1/12
r = 0.1
kappa = 0.2
sigma = 0.12
theta = -0.14
K = range(1,300)
K_ = np.array(K).reshape([len(K),1])
max_strike = 299
mu4 = VG_mu4(r,T,kappa,theta,sigma)
M = np.power((2*max_strike*mu_4)/(10**-7), (1/4))

# Characteristic function
cf = ChFVG(T,r,kappa,theta,sigma,S0)


start = time.time()
y = (COS_speedy(cf,S0,r,T,K_, M, int(100_000)))
end = time.time()
print(end-start)

# call put parity
COSVGSpeedy = []
for i in range(0,len(y)):
    COSVGSpeedy.append(y[i] - ((i+1) * np.exp(-r*T)) + S0)

# Empty lists
MC_Price = []
lower_c = []
upper_c = []

# Find MC Result
for K_ in range(1,300):
    c, bc = VG_MonteCarlo(1_000_000,100,K_,kappa,theta,sigma,r,T)
    MC_Price.append(c)
    lower_c.append(c - bc)
```

```
    upper_c.append(c + bc)

# Find the errors
VG_errors = []
for i in range(0,len(y)):
    VG_errors.append(abs(MC_Price[i] - COSVGSpeedy[i]))

# Absolute MC error bound
ebs_c = []

for i in range(0,len(lower_c)):
    ebs_c.append(abs(lower_c[i] - upper_c[i]))
```