Alexander Badran
11156890

Option Pricing with Stochastic Volatility and Jumps

Introduction

It is well known that the classical Black-Scholes model does not fully reflect the stochastic nature of financial markets. Consequently, there is a need for more realistic models that better reflect random market movements. One such formulation, which captures many features of the observed volatility surface, is a model with stochastic volatility and jumps in the underlying (SVJ). This project required the implementation of the pricing formula for European options under the SVJ model, where a combination of Heston stochastic volatility (SV) and a Merton-style lognormally-distributed jump process is assumed. To do so, the formulae derived in Gatheral [3] were used as a guide.

This report is set out as follows: following the introduction, a more detailed mathematical description of the problem is given. Next, an outline of the implementation algorithm is provided. The method used to test the correctness of the solution and difficulties encountered in the implementation are then presented, followed by a brief conclusion.

Formulation

More formally, the project required the implementation of the pricing formula for a European call option, where the following stochastic differential equations (SDEs) govern the underlying process:

$$dS = \mu S dt + \sqrt{v} S dZ_1 + (e^{\alpha+\delta\epsilon} - 1) S dq \tag{1}$$

$$dv = -\lambda(v - \overline{v})dt + \eta\sqrt{v} dZ_2 \tag{2}$$

with

$$\langle dZ_1 dZ_2 \rangle = \rho dt$$

where μ is the deterministic instantaneous drift of stock price returns, η is the volatility of the volatility process, $\overline{v}$ is the long-run volatility, λ is the speed of reversion to the long-run volatility, ρ is the correlation between the random stock price returns and the changes in the volatility, α is the mean log-jump and δ is the standard deviation. dZ_1

and dZ_2 are the Wiener processes, $\epsilon \sim N(0,1)$ and

$$dq = \begin{cases} 0, \text{ with probability } 1-\lambda_J \\ 1, \text{ with probability } \lambda_J \end{cases}$$

is the Poisson process driving the jumps, with jump intensity λ_J. Note that Gatheral assumes zero interest rates and dividend payments.

The pricing method adopted by Gatheral is a characteristic function approach. Once the characteristic function of the process is obtained, the following formula is used to price the option

$$C(S,K,T) = S - \sqrt{SK}\frac{1}{\pi}\int_0^\infty \frac{du}{u^2+\frac{1}{4}}\text{Re}\left[e^{-iuk}\phi_T(u-i/2)\right] \tag{3}$$

where $\phi(u)$ is the characteristic function of the underlying process and $k = \log(K/S)$. Gatheral provides the characteristic function for the SVJ process, demonstrating that it is simply the product of the well-known Heston and jump characteristic functions. Specifically,

$$\phi_T(u) = e^{C(u,T)\overline{v}+D(u,T)v}\, e^{\psi(u)T} \tag{4}$$

with

$$\begin{aligned} C(u,\tau) &= \lambda\left\{r_-\tau - \frac{2}{\eta^2}\log\left(\frac{1-ge^{-d\tau}}{1-g}\right)\right\} \\ D(u,\tau) &= r_-\frac{1-e^{-d\tau}}{1-ge^{-d\tau}} \\ \psi(u) &= -\lambda_J iu\left(e^{\alpha+\delta^2/2}-1\right) + \lambda_J\left(e^{iu\alpha-u^2\delta^2/2}-1\right) \end{aligned}$$

where we define

$$\alpha = -\frac{u^2}{2} - \frac{iu}{2}, \quad \beta = \lambda - \rho\eta iu, \quad \gamma = \frac{\eta^2}{2}$$
$$d = \sqrt{\beta^2 - 4\alpha\gamma}, \quad r_\pm = \frac{\beta \pm d}{\eta^2}, \quad g = \frac{r_-}{r_+}$$

The first exponential term in Equation (4) is the characteristic function of the SV component, while the second is that of the jump component. It is worth noting that the definition of $C(u,\tau)$ provided by Gatheral differs from Heston's original definition and those provided in most other papers on the subject. In Gatheral's definition, the argument of the complex logarithm never cross the negative real axis. Consequently, the principle value of the complex logarithm in $C(u,\tau)$ is continuous with respect to u. This offers a substantial advantage in comparison to alternate definitions in which the principle value of the log function in $C(u,\tau)$ is discontinuous as u crosses the negative real axis. Ordinarily, when implementing the solution care must be taken to ensure that the correct value of the multivalued log function is chosen. As we are no longer concerned with the multivalued nature of the complex logarithm, standard numerical integration routines can be used.

Implementation

The implementation of the solution was done in two parts. First, the SV component was implemented and tested. Once the correctness of the SV component was confirmed, jumps were added by amending the characteristic function. The implementation of the pricing formula was performed in C++. The main component of the solution is the class `Euro_Call_Option`, which is designed to represent a European call option. The parameters given above are private members of `Euro_Call_Option`, and below are the headers of the four member functions (three public and one private) of the class.

- `Euro_Call_Option::Euro_Call_Option(double xK, double xv_bar, double xeta, double xrho, double xl, double xalpha_J=0, double xdelta_J=0, double xl_J=0)`

  The constructor for the class assigns parameter values to an instance of the object upon declaration. If no jump parameters are given the underlying process used for pricing is the standard Heston model.

- The public member function
  `dcomp phi(double u, double complex_shift, double v, double T)` returns the value of characteristic function, as defined in Equation (4), at the point `u-i*complex_shift` with `T` time to expiration and current volatility `v`. The reason this function is declared publicly is for future use.

- The private member function
  `double Integrand(double u, double x, double v, double T)` returns the value of the integrand in Equation (3) at `u`, where `v` is the current volatility and `x` and `T` are defined in Equation (3).

- The public member function `double Value(double S, double v, double T)` returns the value of a European call option with stock price `S`, current volatility `v` and `T` time to expiration. The function implements Equation (3) by numerically integrating the member function `Integrand(double,double,double,double)`.

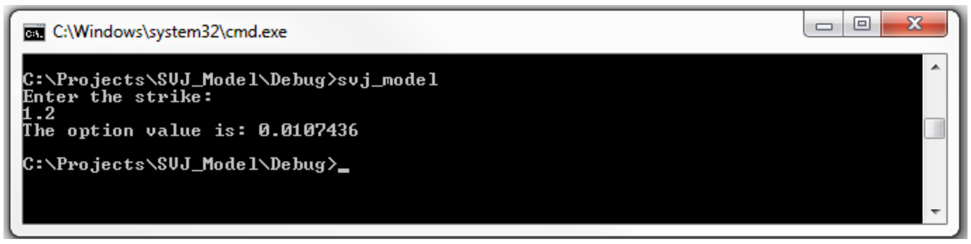
The only member function that is model dependent is the characteristic function
`dcomp phi(double v, double T, double u, double complex_shift)`. Consequently, this implementation offers a user flexibility. If a user is required to price a European option with a different underlying process only the characteristic function needs to be redefined as the rest of the implementation operates independently. The level of abstraction could be taken further, however, for the purpose of this project the level of abstraction is suffice.

A vast number of numerical integration schemes are available to numerically approximate the integral in Equation (3). As efficiency is not a concern for the project, the numerical integration routine chosen was the straight forward composite Simpson's rule,

$$\int_a^b f(u)\,du \approx \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right] \tag{5}$$

with n subintervals (n is even), $h = (b-a)/n$, $x_0 = a$, $x_j = a + jh$ and $x_n = b$. The composite Simpson's rule is implemented in the function `double SimpsonsRule(boost::function<double(double u)>f,double a,double b,int n)`, which returns the numerical integral of the boost function `f(u)` where `a`, `b`, and `n` are as above. Due to the decaying nature of the integrand in Equation (3), the numerical integration is performed between 0 and 1 with 100,000 subintervals, to ensure that the component that has the largest contribution is well sampled, and then from 1 to 100,000 with 5,000,000 subintervals. The choices of `n` and the upper limit were made after sampling the option price (for a given set of parameters) to confirm a stable solution. For future research, a more robust numerical integration routine will be implement.

The main function of the program is relatively simple. First, the user is prompted for a strike price as all the other parameters and variables are hard-wired in `int main(int argc, char *argv[])`. Once received, the function returns the option value. Figure 1 is a screen shot of an example output of the program.



```
C:\Projects\SVJ_Model\Debug>svj_model
Enter the strike:
1.2
The option value is: 0.0107436

C:\Projects\SVJ_Model\Debug>_
```

Figure 1: An example of the output of the program

Testing

Once the solution had been implemented the validity of the output was tested. First, the implementation of the ordinary Heston model (without jumps) was tested against a Monte-Carlo simulation[1]. It is important to note that there are three types of errors to be expected in testing this method: the numerical error associated with the calculation of the semi-infinite integral in Equation (3), the statistical error, which is associated with the calculation of the expected value of a random process via the simulation of independent realisations of the process, and the discretisation error, which is the error associated with the discretisation of the SDEs (1) and (2). The numerical error is relatively negligible due to the use of a large number of subintervals. The statistical error is alleviated by simulating a large number of sample paths.

[1]The implementation of the Monte-Carlo simulation used for testing is a modification of code obtained from Gatheral's website: *http://www.math.nyu.edu/fellows_fin_math/gatheral/h2.c*.

The discretisation error, however, is not trivial for this process and care must be taken upon implementation. Initially, the Monte-Carlo simulation used a simple Euler discretisation for the variance and stock price process. It is well known, however, that discretisation errors are encountered when simulating SV models due to the non-zero probability of obtaining negative variances (for the continuous-time case the variance cannot reach zero as long as the Feller condition is satisfied, which is the case for all examples considered in this project). To deal with the problem of negative variances, an absorbing assumption is imposed: if $v < 0$ then $v = 0$. Despite taking a large number of time steps in an attempt to minimise the error, a discretisation error persisted. To alleviate the problem, the Euler discretisation of the volatility SDE was changed to a Milstein discretisation and the Monte-Carlo simulation was repeated, i.e. Equation (2) is discretised as

$$\Delta v_{i+1} = -\lambda(v_i - \overline{v})\Delta t + \eta\sqrt{v_i \Delta t}Z + \frac{\eta^2}{4}\Delta t(Z^2 - 1)$$

with $Z \sim N(0, 1)$.

To test our implementation of the SV component, we considered an option over a range of strike prices with $S = 1$, $v = 0.04$, $T = 1$, $\overline{v} = 0.04$, $\rho = -0.64$, $\eta = 0.39$, $\lambda = 1.15$ and $\alpha = \delta = \lambda_J = 0$. These parameters were taken from a calibrated option provided on page 71 of Gatheral [3]. In each case, 1,000,000 paths and their antithetic paths were simulated with 10,000 time steps. As efficiency is not a concern in this project, the larger the number of time steps and sample paths, the better. These values were chosen as they are substantially large, yet are computationally feasible. The results are shown for both an Euler and a Milstein discretisation in Tables 1 and 2 respectively. Table 1 illustrates that for $K = 1.2$ our option value falls outside the confidence interval obtained from the Monte-Carlo simulation, despite having a relative error when compared to the upper and lower confidence limit of less than 1.2%. Table 2 illustrates that the results agree more closely when a Milstein discretisation is used for the volatility SDE. Further analysis could be performed on the results obtained from the Monte-Carlo simulation, however, the purpose of this project is not to investigate Monte-Carlo methods for the SVJ process, rather, to implement the closed-form solution for the SVJ process. As these results demonstrate that the implementation of the SV component is valid, testing of the SV component stopped here.

Strike	Our Price	MC Price	Std. Dev.	Confidence Interval
0.8	0.217837	0.217862	0.059275	(0.217744, 0.217981)
0.9	0.137427	0.137465	0.053557	(0.137358, 0.137573)
1.0	0.072398	0.072449	0.050418	(0.072348, 0.072550)
1.1	0.029465	0.029516	0.041025	(0.029434, 0.029598)
1.2	0.009342	0.009398	0.026009	(0.009346, 0.009450)

Table 1: Comparison of our implementation of Equation (3) to a Monte-Carlo simulation. $S = 1$, $v = 0.04$, $T = 1$, $\overline{v} = 0.04$, $\rho = -0.64$, $\eta = 0.39$, $\lambda = 1.15$, $\alpha = \delta = \lambda_J = 0$ and SV is approximated with a Euler discretisation.

Strike	Our Price	MC Price	Std. Dev.	Confidence Interval
0.8	0.217837	0.217851	0.059444	(0.217733, 0.217970)
0.9	0.137427	0.137465	0.053580	(0.137390, 0.137572)
1.0	0.072398	0.072447	0.050301	(0.072388, 0.072547)
1.1	0.029465	0.029485	0.040874	(0.029404, 0.029567)
1.2	0.009342	0.009332	0.025880	(0.009281, 0.009384)

Table 2: Comparison of our implementation of Equation (3) to a Monte-Carlo simulation. $S = 1$, $v = 0.04$, $T = 1$, $\overline{v} = 0.04$, $\rho = -0.64$, $\eta = 0.39$, $\lambda = 1.15$, $\alpha = \delta = \lambda_J = 0$ and SV is approximated with a Milstein discretisation.

Since the correctness of the implementation of the SV component had been verified, the jump component was added to the process. To do so, the SV characteristic function was multiplied by the jump characteristic function. This was performed in the public member function `dcomp phi(double u, double complex_shift, double v, double T)`. As the amendment only required an addition two lines of code, minimal implementation difficulties were expected to be encountered.

To test the implementation the Monte-Carlo algorithm was altered to include jumps. To do so, the following algorithm obtained from page 22 of Broadie [1] was implemented:

1. Disregard the jump component and simulate the stock price S_T under the SV model.
2. Generate a Poisson random variable with mean $\lambda_J T$ to determine the number of jumps, denoted by J, that have occurred in the time horizon.
3. Generate independent jump sizes ξ_i, for $i = 1, ..., J$, from a lognormal distribution with mean α_J and variance δ_J^2.
4. Calculate the adjusted stock price by multiplying the price from Step 1 with the jump sizes, i.e. $\tilde{S_T} = S_T \prod_{i=1}^{i=J} \xi_i$.
5. Compute the derivative payoff at $\tilde{S_T}$.

The parameters that were used to test the SV implementation are again used for the SVJ model, except with the following choices for the jump parameters: $\alpha_J = -0.1151$, $\delta_J = 0.0967$ and $\lambda_J = 0.1308$. Again, these parameters were taken from a calibrated option provided by Gatheral. 1,000,000 paths and their antithetic paths were simulated with 10,000 time steps (the antithetic paths were only simulated for the SV component and a standard Monte-Carlo simulation was performed for the jumps). The Milstein discretisation in Equation (2) was used for the volatility SDE. Unfortunately, as the results in Table 3 illustrate, the sets of option prices obtained from each implementation do not agree. Clearly there must have been an error in the implementation of the closed-form solution, the implementation of the Monte-Carlo algorithm with jumps, or in both implementations. Interestingly enough, however, after thorough testing, no errors were found in any of the implementations. The "error" was in fact a conceptual error, as the Monte-Carlo algorithm was not adjusted to impose that the risk-neutral expectation of the stock price is the forward price. Furthermore, the conceptual misunderstanding was not resolved until thorough testing of the implementation had been performed. Consequently, much time was spent trying to find an implementation error that was not actually there.

Strike	Our Price	MC Price	Std. Dev.	Confidence Interval
0.8	0.217837	0.206782	0.065097	(0.206652, 0.206913)
0.9	0.137427	0.129043	0.057777	(0.128928, 0.129159)
1.0	0.072398	0.067433	0.051524	(0.067330, 0.067536)
1.1	0.029465	0.027387	0.040222	(0.027306, 0.027467)
1.2	0.009342	0.008752	0.025075	(0.008702, 0.008802)

Table 3: Initial comparison of our implementation of Equation (3) to a Monte-Carlo simulation. $S = 1$, $v = 0.04$, $T = 1$, $\overline{v} = 0.04$, $\rho = -0.64$, $\eta = 0.39$, $\lambda = 1.15$, $\alpha = -0.1151$, $\delta = 0.0967$, $\lambda_J = 0.1308$ and SV is approximated with a Milstein discretisation.

As the only difference between the implementation of the Heston model and the SVJ model is in the inclusion of the jump characteristic function, checking the implementation of this function was a logical place to begin a search for any errors. After close inspection, however, it appeared as though there were no coding errors in the implementation. To perform a more thorough analysis, the real and imaginary components of the jump characteristic function were evaluated at the point $u_r - i/2$ (where u_r is a real number), i.e.

$$\begin{aligned}
\phi_T(u_r - i/2) &= -\lambda_J i\left(u_r - \frac{i}{2}\right)(e^{\alpha+\delta^2/2} - 1) + \lambda_J[e^{i(u_r-i/2)\alpha-(u_r-i/2)^2\delta^2/2} - 1] \\
&= -\frac{\lambda_J(e^{\alpha+\delta^2/2} + 1)}{2} - \lambda_J i u_r(e^{\alpha+\delta^2/2} - 1) + \lambda_J e^{\alpha/2-u_r^2\delta^2/2+\delta^2/8+iu_r(\alpha+\delta^2/2)} \\
&= -\frac{\lambda_J(e^{\alpha+\delta^2/2} + 1)}{2} + \lambda_J e^{\alpha/2-u_r^2\delta^2/2+\delta^2/8}\cos\left[u_r(\alpha+\delta^2/2)\right] \\
&\quad - \lambda_J i\left\{u_r(e^{\alpha+\delta^2/2} - 1) + e^{\alpha/2-u_r^2\delta^2/2+\delta^2/8}\sin[u_r(\alpha+\delta^2/2)]\right\} \\
&= \Re[\phi_T(u_r - i/2)] + i\Im[\phi_T(u_r - i/2)]
\end{aligned}$$

where $\Re(\cdot)$ and $\Im(\cdot)$ denote the real and imaginary part of a complex number. The resulting formula was input into an Excel spreadsheet (this Excel spreadsheet has been submitted with the project). The implementation of the jump characteristic function was checked against the version derived by hand for certain values of u_r. Both expressions returned the same value for each of the tested points, indicating that the error was not located in the implementation of the characteristic function.

Since there were no errors in the implementation of the jump characteristic function, the expression provided by Gatheral was rederived to ensure that there were no errors in the text. Upon doing so (in fact, it was after the derivation was revisited for a second time), it became apparent that the drift parameter in the Monte-Carlo simulation had not been adjusted. The derivation of the jump characteristic function performed for testing is included for completeness. Consider the Lévy-Khintchine representation of a Merton-style jump-diffusion process (with $\sigma = 0$),

$$\phi(u) = \exp\left\{iu\omega T + T\int \left[e^{iu\chi} - 1\right]\mu(\chi)d\chi\right\}$$

where

$$\mu(\chi) = \frac{\lambda_J}{\sqrt{2\pi\delta^2}}\exp\left[-\frac{(\chi-\alpha)^2}{2\delta^2}\right]$$

Simplifying this expression yields,

$$\begin{aligned}
\phi(u) &= \exp\left\{iu\omega T + T\int \left[e^{iu\chi} - 1\right]\frac{\lambda_J}{\sqrt{2\pi\delta^2}}\exp\left\{-\frac{(\chi-\alpha)^2}{2\delta^2}\right\}d\chi\right\}\\
&= \exp\left\{iu\omega T + \frac{\lambda_J T}{\sqrt{2\pi\delta^2}}\int_{-\infty}^{\infty}\exp\left[iu\chi - \frac{(\chi-\alpha)^2}{2\delta^2}\right]d\chi - \lambda_J T\right\}\\
&= \exp\left\{iu\omega T + \frac{\lambda_J T e^{iu\alpha}}{\sqrt{2\pi\delta^2}}\int_{-\infty}^{\infty}\exp\left[iu(\chi-\alpha) - \frac{(\chi-\alpha)^2}{2\delta^2}\right]d\chi - \lambda_J T\right\}\\
&= \exp\left\{iu\omega T + \frac{\lambda_J T e^{iu\alpha}}{\sqrt{2\pi\delta^2}}\int_{-\infty}^{\infty}\exp\left\{-\frac{1}{2\delta^2}\left[(\chi-\alpha)^2 - 2iu\delta^2(\chi-\alpha)\right]\right\}d\chi - \lambda_J T\right\}\\
&= \exp\left\{iu\omega T + \frac{\lambda_J T e^{iu\alpha}}{\sqrt{2\pi\delta^2}}\int_{-\infty}^{\infty}\exp\left\{-\frac{1}{2\delta^2}\left[(\chi-\alpha-iu\delta^2)^2 + u^2\delta^4\right]\right\}d\chi - \lambda_J T\right\}\\
&= \exp\left\{iu\omega T + \frac{\lambda_J T e^{iu\alpha-u^2\delta^2/2}}{\sqrt{2\pi\delta^2}}\int_{-\infty}^{\infty}\exp\left[-\frac{(\chi-\alpha-iu\delta^2)^2}{2\delta^2}\right]d\chi - \lambda_J T\right\}\\
&= \exp\left\{iu\omega T + \lambda_J T e^{iu\alpha-u^2\delta^2/2} - \lambda_J T\right\}\\
&= \exp\left\{iu\omega T + \lambda_J T\left[e^{iu\alpha-u^2\delta^2/2} - 1\right]\right\}
\end{aligned}$$

Imposing the condition that the risk-neutral expectation of the stock price is the forward price allows us to solve for the drift, i.e.

$$\phi(-i) = E[e^{x_T}] = 1 \implies \omega T + \lambda_J T(e^{\alpha+\delta^2/2} - 1) = 0 \implies \omega = -\lambda_J(e^{\alpha+\delta^2/2} - 1)$$

and hence

$$\phi_T(u) = e^{\psi(u)T}$$

with

$$\psi(u) = -\lambda_J iu(e^{\alpha+\delta^2/2} - 1) + \lambda_J(e^{iu\alpha-u^2\delta^2/2} - 1)$$

which agrees with the jump characteristic function as provided by Gatheral.

As the jump characteristic function provided by Gatheral was proven to be correct, other testing was performed in an attempt to find a discrepancy between the implementation and the Monte-Carlo prices. The next method of testing performed was to validate the function `double Integrand(double u, double x, double v, double T)`. This was done by implementing the formula for the integrand in an Excel worksheet (located in the same Excel worksheet as the jump characteristic function) and testing the output for certain values of u. As was the case with the jump characteristic function, the values calculated in Excel agreed with those obtained from the implemented function. This result was not very surprising as the implementation functioned correctly for the SV model without jumps. As the further testing was performed, it became more apparent that there was no implementation error in the solution. This lead to the realisation that the error must be conceptual rather than in the implementation. Eventually, the conceptual error was discovered and resolved.

Once the drift parameter in the Monte-Carlo algorithm had been adjusted to impose the risk-neutral pricing assumption, the simulation was repeated for the parameters $S = 1$, $v = 0.04$, $T = 1$, $\overline{v} = 0.04$, $\rho = -0.64$, $\eta = 0.39$, $\lambda = 1.15$, $\alpha = -0.1151$, $\delta = 0.0967$ and $\lambda_J = 0.1308$. Once again, 1,000,000 paths and their antithetic paths were simulated with 10,000 time steps (the antithetic paths were only simulated for the SV component and a standard Monte-Carlo simulation was performed for the jumps). The results from the simulation are provided in Table 4.

The closed-form option prices and those obtained from the Monte-Carlo simulation are relatively close; the largest relative error between the bounds of the confidence interval and the closed-form option prices is approximately 2%. The closed-form option prices, however, lie outside the confidence interval for each strike price. The discrepancies were assumed to be caused by the statistical error involved with the Monte-Carlo simulation, not by an implementation error. The reason for such an assumption was that the values were too close for the error to be

Strike	Our Price	MC Price	Std. Dev.	Confidence Interval
0.8	0.219081	0.218695	0.066917	(0.218561, 0.218829)
0.9	0.139729	0.139341	0.059288	(0.139222, 0.139460)
1.0	0.075586	0.075209	0.052951	(0.075103, 0.075315)
1.1	0.032256	0.031917	0.042657	(0.031831, 0.032002)
1.2	0.010744	0.010576	0.027447	(0.010521, 0.010631)

Table 4: Comparison of our implementation of Equation (3) to a Monte-Carlo simulation. $S = 1$, $v = 0.04$, $T = 1$, $\overline{v} = 0.04$, $\rho = -0.64$, $\eta = 0.39$, $\lambda = 1.15$, $\alpha = -0.1151$, $\delta = 0.0967$, $\lambda_J = 0.1308$ and SV is approximated with a Milstein discretisation.

Strike	Our Price	MC Price	Std. Dev.	Confidence Interval
0.8	0.200740	0.200736	0.032065	(0.200730, 0.200742)
0.9	0.104401	0.104937	0.022552	(0.104393, 0.104402)
1.0	0.012987	0.012986	0.006418	(0.012984, 0.012987)
1.1	0.000102	0.000102	0.001989	(0.000102, 0.000103)
1.2	0.000007	0.000007	0.000502	(0.000007, 0.000007)

Table 5: Comparison of our implementation of Equation (3) to a Monte-Carlo simulation. $S = 1$, $v = 0$, $T = 1$, $\overline{v} = \rho = \eta = \lambda = 0$, $\alpha = -0.1151$, $\delta = 0.0967$ and $\lambda_J = 0.1308$.

implementation related, and as the jump component is not path dependant, there should be no further discretisation error added by the jump process. Furthermore, the numerical integration was producing stable results to seven significant figures. To illustrate that the error was in fact in the Monte-Carlo prices, the simulation could be repeated with an increased number of time steps and paths, however, this turned out to be computationally infeasible, as the simulation did not return results after several days. Consequently, to test the correctness of the implementation a different approach needed to be adopted. An option on an underlying that follows a pure-jump process was considered, where $S = 1$, $v = 0$, $T = 1$, $\overline{v} = \rho = \eta = \lambda = 0$, $\alpha = -0.1151$, $\delta = 0.0967$ and $\lambda_J = 0.1308$. To test the pricing of this option, 100,000,000 paths were simulated, with only 1 time step to reduce the computation time. As the jumps are not path dependant, taking only 1 time step will not impose any discretisation errors. The results are presented in Table 5. For the pure-jump process, the Monte-Carlo prices agree with our closed-form solution illustrating that our implementation of the jump process is valid. It would be interesting to study the Monte-Carlo simulation of the SVJ process more closely, however, as it is not the primary concern of the project, the analysis of the Monte-Carlo algorithm was stopped here.

Conclusion

For this project, an option pricing model with stochastic volatility with jumps in the underlying was implemented in C++. First, a Monte-Carlo algorithm with a Milstein discretisation was used to test the implementation of the stochastic volatility component. Once the stochastic volatility component was validated, jumps were added to the underlying process. There were not many implementation difficulties, however, conceptual difficulties were encountered. Specifically, the drift parameter was not adjusted in the Monte-Carlo algorithm to impose a risk-neutral pricing assumption when jumps were included. Consequently, much time was spent searching for errors in the implementation that did not exist. Once the drift was adjusted, the implementation of the SVJ model was tested against the Monte-Carlo simulation. Despite obtaining small confidence intervals that were relatively close to our option values, our option values fell outside the acceptable range. When more paths were taken, it was shown that option prices obtained from the Monte-Carlo simulation agree with the implemented solution. As this project is not concerned with the Monte-Carlo simulation of option prices under the SVJ model, no more time was spent refining the algorithm. Finally, the solution created in this project will be adapted and used in future research to fit volatility surfaces.

References

[1] M. Broadie and . Kaya. Exact simulation of stochastic volatility and other affine jump diffusion processes. *Operations Research*, 54:217–231.

[2] J. Gatheral. Official heston monte carlo c - code. `http://www.math.nyu.edu/fellows_fin_math/gatheral/h2.c`, 2002.

[3] J. Gatheral. *The Volatility Surface: A Practitioner's Guide.* John Wiley & Sons Ltd., 2006.