Group Assingment
Numerical Valuation of Options with Jumps in the Underlying - Finite Difference Method for Option Pricing under the Merton Jump-Diffusion Model

Joshua Felix Jablonowski - 60849
Tornike Kikacheishvili - 60762
Rodrigo Orozco Perez - 60348

May 2024

Contents

1 Introduction

Various alternative models have been developed in the financial literature to address issues like stochastic volatility, deterministic local volatility functions, jump-diffusion models, and Lévy models. Jump-diffusion and Lévy models are particularly attractive due to their ability to explain stock jump patterns and realistic pricing of options close to maturity.

The paper focuses on the numerical valuation of European Vanilla options using the jump-diffusion approach with constant coefficients. It explores models by Merton and Kou, utilizing analytical formulas for solutions. The research aims to lay the groundwork for solving more complex financial products like American options and path-dependent options.

The numerical valuation of jump-diffusion processes involves various techniques such as the ADI finite difference method combined with the fast Fourier transform and a finite element method. The paper discusses the discretization process, the solver methodology, and the use of the fast Fourier transform to enhance computational efficiency. Numerical tests are conducted to verify the accuracy of the numerical schemes.

This project aims to develop and implement numerical methods for valuing European Vanilla options under the jump-diffusion framework with constant coefficients, specifically focusing on the Merton.

While the paper explores analytical solutions, the project emphasizes creating numerical implementations using the techniques described in sections 3 and 4. This can involve:

Reproducing the techniques: The project can replicate the numerical methods presented in section 4, such as the BDF2 finite difference method combined with the fast Fourier transform (FFT) or a finite element method. Presenting alternative methods: The project can explore alternative numerical approaches besides those mentioned in the paper for valuing the options. Section 7 likely provides reference results obtained using the paper's methods. The project should compare the results obtained from the implemented numerical methods with these reference values to assess their accuracy and efficiency.

In essence, the project aims to bridge the gap between theoretical models and practical applications by providing functional numerical tools for option valuation under jump-diffusion models

2 Numerical Approach

We now present the Merton numerical scheme function. This function serves as a numerical implementation of the finite difference method for option pricing under the Merton jump-diffusion mode. It takes some parameters such as time to maturity ($T = 1$), truncation point ($x^* = 4$), variance of the jumps ($\sigma_j = 0.5$), intensity of the jumps ($\lambda = 0.1$), and strike price ($K = 1$).

```
def merton_numerical_scheme(T, x_star, n, q, sigma, r, lambda_,
    zeta, K, mu_J=0, sigma_j=0.5):
    # Discretize space and time
    h = 2 * x_star / (n - 1)
    k = 0.2
    price = np.linspace(-x_star, x_star, n)
```

We utilize a two-dimensional array, u, to store the option prices across the discretized time and space. This array is initialized with zeros, having q rows (representing time steps) and n columns (representing grid points in the underlying asset's space).

The payoff function, defines the option's value at expiry. For a European call option, this payoff is not negative, logarithm of stock price (S) minus the strike price (K), price should be take range from -4 to 4 as it given in paper. Therefore, the payoff function becomes exp(x) - K.

This approach aligns with the finite difference method's backward-in-time solution approach, where we begin with the final condition (payoff at expiry) and iteratively solve backwards to reach the initial time.

```
    u = np.zeros((q, n))
    for i in range(n):
        if math.exp(price[i]) - K >= 0:
            u[0, i] = math.exp(price[i]) - K
```

The epsilon function is used to approximate the integral term in the partial differential equation of the Merton model, which represents the expected payoff from jumps in the underlying asset price.

$$\epsilon(\tau, x, x^*) = e^{x+\sigma_j^2}\phi(\frac{x - x^* + \sigma_j^2}{\sigma_j} - Ke^{-r\tau}\phi(\frac{x - x^*}{\sigma_j})$$

where:

$$\phi(y) := \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{y} e^{-\frac{x^2}{2}}\, dx$$

We should consider a uniform mesh in space and time. Let $x_i = -x^* + (i-1)h$ for $i = 1, \ldots, n$ and $\tau_m = (m-1)k$ for $(m = 1, \ldots, q)$. Let $u_i^m \approx u(\tau_m, x_i)$ and $f_{ij} := f(x_j - x_i)$.

from this formula $\tau_m = (m-1)k$ for $(m = 1, \ldots, q)$, we can define q, which is equal to $\frac{\tau_m}{k} + 1$

Code version of the epsilon function is:

```
def epsilon(tau_m, x_i, x_star):
    term1 = np.exp(x_i + 0.5 * sigma_j**2) * norm.cdf((x_i - x_star + sigma_j**2) / sigma_j)
    term2 = K * np.exp(-r * tau_m) * norm.cdf((x_i - x_star) / sigma_j)
    return term1 - term2
```

Also, classical Merton's model Y_i are normally distributed, with mean μ_J and standard deviation σ_J. That is, $dF(x) = f_m(x)\,dx$, where

$$f_{\mathrm{m}}(x) = \frac{1}{\sqrt{2\pi}\sigma_J} e^{-(x-\mu_J)^2/2\sigma_{\mathrm{j}}^2]}$$

This function is represented in code as a f function:

```
def f(x):
    return np.exp(-(x - mu_J)**2 / (2 * sigma_j**2)) / np.sqrt(2 * np.pi * sigma_j**2)
```

After applying all time, space discretization to the system of equations which give by: (eq. 22) the problem can be reduced to a problem of solving a linear system of equations (eq. 32)

$$\begin{cases} u_\tau - \frac{1}{2}\sigma^2 u_{xx} - (r - \frac{1}{2}\sigma^2 - \lambda\zeta)u_x + (r+\lambda)u - \lambda \int_R u(\tau, x+y)\,dF(y) = 0, & \forall(\tau, x) \in [0,T) \times R, \\ u(0,x) = \tilde{g}(x), & \forall x \in R. \end{cases}$$

$$(\omega_0 \mathbf{I} + \mathbf{C} + \mathbf{D})u^m = \mathbf{b}^m$$

where:

$$\omega_0 = \begin{cases} 1 & \text{if} \quad m = 1, \\ 3/2 & \text{if} \quad m \geq 2 \end{cases}$$

I is the matrix and matrices $C := [c_{ij}]_{i,j=1}^n$ and $D := [d_{ij}]_{i,j=1}^n$ are given by

$$c_{ij} = \begin{cases} -k\sigma^2/2h^2 + k(r - \sigma^2/2 - \lambda\zeta) & \text{if} \quad i = j-1, 2 \leq i \leq n-1, \\ k\sigma^2/2h^2 + (r+\lambda)k & \text{if} \quad i = j, 2 \leq i \leq n-1, \\ -k\sigma^2/2h^2 - k(r - \sigma^2/2 - \lambda\zeta)/2h & \text{if} \quad i = j+1, 2 \leq i \leq n-1, \\ 0 & otherwise; \end{cases}$$

$$d_{ij} = \begin{cases} -kh_{ij}/2 & \text{if} \quad 2 \leq i \leq n-1 \text{ and } j = 1, n, \\ -kh_{ij} & \text{if} \quad 2 \leq i \leq n-1 \text{ and } 2 \leq j \leq n-1, \\ 0 & otherwise; \end{cases}$$

Finally, the right side $b^m := (b_1, b_2, \cdot, b_n - 1, b_n)^T$ is given component-wise by:

$$b_i = k\lambda\epsilon(\tau_m, x_i, x^*) + \omega_1 u_i^m - 2, \quad \text{for } i = 2, \ldots, n-1.$$

where:

$$\omega_1 = \begin{cases} 1 & \text{if} \quad m = 1, \\ 2 & \text{if} \quad m \geq 2, \end{cases}$$

$$\omega_1 = \begin{cases} 0 & \text{if} \quad m = 1, \\ -1/2 & \text{if} \quad m \geq 2, \end{cases}$$

and from the boundary conditions $b_1 = 0, \quad b_n = \omega_0(e^{x^*} - \mathbf{K}e^{-r\tau_m})$

For our implementation we chose to follow the authors approach by using the Crank-Nichelson (BDF2) method and using the composite trapezoidal method for the integral discretization. The core computational routine of the code leverages the finite difference method to address the partial differential equation (PDE) governing the Merton model. This numerical approach entails discretizing the PDE into a well-defined system of linear equations. Subsequently, the code employs a suitable linear solver to determine the option prices at each grid point within the discretized spatial and temporal domain.

```
        tau_m = (m-1)*k
        C = np.zeros((n, n))
        D = np.zeros((n, n))
        b = np.zeros(n)
        omega0 = 3/2 if m > 1 else 1
        omega1 = 2 if m > 1 else 1
        omega2 = -1/2 if m > 1 else 0

        for i in range(1, n - 1):
            C[i, i - 1] = -k * sigma**2 / (2 * h**2) + k * (r - 0.5
    * sigma**2 - lambda_ * zeta) / (2 * h)
            C[i, i] = k * sigma**2 / h**2 + k * (r + lambda_)
            C[i, i + 1] = -k * sigma**2 / (2 * h**2) - k * (r - 0.5
    * sigma**2 - lambda_ * zeta) / (2 * h)
            D[i, 0] = -k * h * lambda_ * f(price[0] - price[i]) / 2
            D[i, -1] = -k * h * lambda_ * f(price[-1] - price[i]) /
    2

            for j in range(1, n - 1):
                D[i, j] = -k * h * lambda_ * f(price[j] - price[i])

            b[i] = k * lambda_ * epsilon(tau_m, price[i], x_star) +
    omega1 * u[m - 1, i] + omega2 * u[m - 2, i]

        C += omega0 * np.eye(n) + D
        C[0, 0], C[-1, -1] = 1, 1
```

```
        b[0], b[-1] = 0, omega0 * (np.exp(x_star) - K * np.exp(-r *
    tau_m))
        u[m, :] = np.linalg.solve(C, b)

    return u

Our solutions is:
\[
\begin{bmatrix}
0.00000000e+00 & 1.58183680e-11 & 3.40896415e-11 & 6.84265741e-11
    \\
1.36842046e-10 & 2.73966123e-10 & 5.48727979e-10 & 1.09774309e-09
    \\
2.18931154e-09 & 4.34525669e-09 & 8.57266924e-09 & 1.68098998e-08
    \\
3.28082136e-08 & 6.39422067e-08 & 1.25071491e-07 & 2.46984630e-07
    \\
4.94927047e-07 & 1.00850973e-06 & 2.08456462e-06 & 4.34066784e-06
    \\
9.01874425e-06 & 1.85108224e-05 & 3.72013305e-05 & 7.27043170e-05
    \\
1.37512410e-04 & 2.50977513e-04 & 4.41546945e-04 & 7.50005648e-04
    \\
1.23991609e-03 & 2.05210663e-03 & 3.70962041e-03 & 8.81747367e-03
    \\
3.14949039e-02 & 1.49642779e-01 & 2.98843800e-01 & 4.70599021e-01
    \\
6.65818006e-01 & 8.87251679e-01 & 1.13829900e+00 & 1.42286098e+00
    \\
1.74537122e+00 & 2.11086133e+00 & 2.52503916e+00 & 2.99437755e+00
    \\
3.52621497e+00 & 4.12886945e+00 & 4.81176805e+00 & 5.58559374e+00
    \\
6.46245195e+00 & 7.45605930e+00 & 8.58195740e+00 & 9.85775493e+00
    \\
1.13034021e+01 & 1.29415019e+01 & 1.47976645e+01 & 1.69009100e+01
    \\
1.92841260e+01 & 2.19845869e+01 & 2.50445340e+01 & 2.85117991e+01
    \\
3.24403385e+01 & 3.68899101e+01 & 4.19205884e+01 & 4.75582052e+01
    \\
5.35981500e+01 & & &
\end{bmatrix}
\]
```

3 Analytical Solution

The following section of this assignment will be concerned with the explicit solution to the Merton Special Case and quantifying the difference in accuracy of the numerical method developed to estimate the PIDE. The exact solution

to the Merton case is given by

$$w(t,s) = \sum_{m=0}^{\infty} \frac{e^{-\lambda'\tau} \cdot (\lambda'\tau)^m}{m!} (C_B S)(\tau, s, K, r_m, \sigma_m)$$

where: $\tau := T - t$, and recalling that $\zeta = e^{\frac{\sigma_J^2}{2}-1}$, the rest of the parameters are given by

$$\lambda' = \lambda(1+\zeta), \qquad \sigma_m^2 = \sigma^2 + \frac{m\sigma_j^2}{\tau}, \qquad r_m = r - \lambda\zeta + \frac{m\ln(1+\zeta)}{\tau}$$

and $C_B S$ denotes the Black-Scholes value of a call:

$$(C_B S)(\tau, s, K, r, \sigma) = s\phi(d_1) - \mathbf{K}e^{-r\tau}\phi(d_2)$$

where:

$$d_1 = \frac{log(\frac{s}{k}) + (r + \frac{1}{2}\sigma^2)}{\sigma\sqrt{\tau}}, \; d_2 = d_1 - \sigma\sqrt{tau}$$

and Φ is the normal cumulative distribution function.

Based on this formulas we crate two functions in code, where m is arbitrarily chosen and $m = 5$. Other variables such as: volatility, strike price, time to maturity, interest rate are given in paper. In the code we have create separately function which is calculating call option price.

```
i = 5
lambda_ = 0.1
sigma = 0.2
sigma_j = 0.5
K = 1
T = 1
t = 0.8
tau_ = 0.2
r = 0
n = 65
zeta = math.exp((sigma_j ** 2) / 2) - 1
lambda_der = lambda_ * (1 + zeta)

def w(t, s, K, tau_, r, sigma, sigma_j, i):
    total_sum = 0
    for m in range(0, i-1):
        sigma_m = sigma ** 2 + (m * sigma_j ** 2) / tau_
        r_sub_m = r - lambda_ * zeta + (m * math.log(1 + zeta)) / tau_
        total = ((math.exp(-lambda_der * tau_) * (lambda_der * tau_) ** m) / math.factorial(m)) * black_scholes(s, K, tau_, r_sub_m, sigma_m)
        total_sum += total
    return total_sum

# Define the Black-Scholes function
def black_scholes(s, K, tau_, r, sigma_m, option_type='call'):
```

```
    d1 = (np.log(s / K) + (r + 0.5 * sigma_m ** 2) * tau_) / (np.
    sqrt(sigma_m) * np.sqrt(tau_))
    d2 = d1 - np.sqrt(sigma_m) * np.sqrt(tau_)

    if option_type == 'call':
        option_price = s * norm.cdf(d1) - K * np.exp(-r * tau_) *
    norm.cdf(d2)

    return option_price
```

Finally we got array where logarithm prices are in range from -4 to 4, which gives us analytical solutions for each time and each price.

```
x_star = 4
s = np.exp(np.linspace(-x_star, x_star, n))
analytical_solution = []

for price in s:
    each_price = w(t, price, K, tau_, r, sigma, sigma_j, i)
    analytical_solution.append(each_price)

print(analytical_solution)
```

Following the derivation of an analytical solution for the Merton model's partial differential equation (PDE), a crucial step involves comparing it to the numerical solution obtained via the finite difference method. This comparison serves to quantify the discretization errors inherent in the numerical approach. To ensure transparency and allow for direct comparison with prior research, we utilize the identical variable notation employed by the authors, where number of spatial points, time and the time step is the same. Right side is our result.

n	**k**	**Error at** x_K
65	0.2	0.00442717

n	**k**	**Error at** x_K
65	0.2	0.00775888

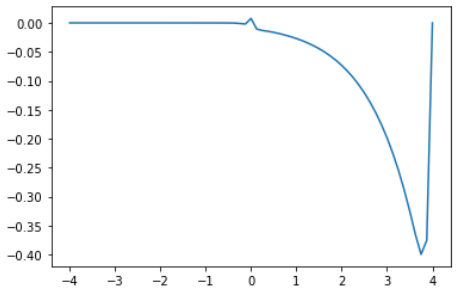

Figure 1: Error

4 Conclusions

This work investigates numerical solutions for valuing European options in a market with a finite number of jumps. We leverage our a-priori knowledge of the underlying jump process through its Lévy-Khintchine representation. The option value is typically obtained by solving a partial integro-differential equation. We focus on the classical Merton model and tried to implement code.

5 Bibliography

1. Cornelis W. Oosterlee b. - Numerical valuation of options with jumps in the underlying Ariel Almendral
2.Duniel j. Duffy - Finite Difference Methods in Financial Engineering