Computational Finance
Take Home Exam 3

Anita Mezzetti (307498)

BM $W = (W_1, W_2)$

$$dV_t = K(\theta - V_t)dt + \sigma\sqrt{Q(V_t)}\, dW_{1t}$$
$$dX_t = (r - V_t/2)dt + \rho\sqrt{Q(V_t)}\, dW_{1t} + \sqrt{V_t - \rho^2 Q(V_t)}\, dW_{2t}$$

$$Q(v) \overset{*}{=} \frac{(v - v_{min})(v_{max} - v)}{(\sqrt{v_{max}} - \sqrt{v_{min}})^2}$$

$$\pi_f := \mathbb{E}[f(X_T)] \overset{*}{=} \sum_{n \geq 0} f_n \ell_n \quad \leftarrow \text{European Option Price}$$

a) Basis vector of $Pol_N(\mathbb{R}^2)$:

$$B_N(v, x) = (1, v, H_1(x), v^2, v H_1(x), H_2(x), \dots, v^n, v^{n-1} H_n(x) \dots, H_N(x))$$

Generalised Hermite polynomials: $H_n(x) = \frac{1}{\sqrt{n!}} \mathcal{H}_n\left(\frac{x - \mu_w}{\sigma_w}\right) \quad n \geq 1$

↳ $\mathcal{H}_n$: standard Hermite polynomials.

Write a MATLAB function HermiteMoments.m that compute the first N Hermite moments

$$\ell_n \overset{*}{=} B_N(V_0, X_0)\, e^{G_N T} e_{\pi(0,n)} \qquad 0 \leq n \leq N$$

G_N : matrix repesentation of the generator $\mathcal{G}$ of (V_t, X_t) restricted to $Pol_N(\mathbb{R}^2)$

$\pi : \mathcal{E} \longrightarrow \{1 \dots M\}$ where $M = \frac{(N+2)(N+1)}{2}$

↳ $\mathcal{E} = \{(m, n) : m, n \geq 0 \,;\, m + n \leq M\}$

Solution's steps:

- Definition of the function Ind which, given (m, n), calculates $\pi(m, n) := \frac{(m+n+1)(m+n)}{2} + n - 1$ as it has been done in Homework 6 ex 2a
- Definition of $M = \frac{(N+1)(N+2)}{2}$
- Hermite polynomials:
  We do two loops to create the basis vector B_N and we call it BN:
  - In order to do that we find also the generalised Hermite polynomials Hn, named H_n needed to find B_N.
  - Note that in the first loop m goes from 0 to N, while in the second one n goes from 0 to N-m. This is because in the domain of π we have (m, n) such that $m, n \geq 0$ and $m + n \leq N$
- Matrix of the generator:
  After the initialisation of the matrix G, we follow the formulas on slide 18 of lecture 6 to fill it correctly, considering the conditions on m and n

- Approximation option price:
We create the matrix L as: $L = BN \cdot e^{G \cdot T}$
Then, we create a loop to obtain a vector e from this matrix:

$$e_n = L(0,n)$$

following *.

This e is the result

b) Write a function SimSDEJacobi.m that simulates N_{sim} paths of the process X_t $0 \le t \le T$ via Euler discretization with time intervals N_{time}.

Solution's steps:
(For this part we follow slide 5 of lecture 6)

- Discretization of the time interval in N_{time} periods $[t_k, t_{k+1}] \Rightarrow$ we define the time step $dt = \frac{T}{N_{time}}$
- Definition of the function $Q(v)$ as specified in *
- Creation of a time loop that goes from 1 to N_{time} to simulate the path of the process. In particular, at each step we update X_t adding the contribution dX_t.
In order to do that, we need to create at each step two randoming vectors for the BM and update also V_t.
dV_t and dX_t are given by:

$$dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{Q(V_t)}\, dW_{1t}$$
$$dX_t = (r - V_t/2)\, dt + \rho\sqrt{Q(V_t)}\, dW_{1t} + \sqrt{V_t + \rho^2 (V_t)}\, dW_{2t}$$

$\Rightarrow$ at each step t: $V_t = V_{t-1} + dV_t$
$X_t = X_{t-1} + dX_t$

We return the vector X.

c) Consider $f(x) := e^{-rT}(e^x - e^k)^+$

Getting back to *, the Fourier coefficients can be computed as

$$f_0 = e^{-rT+\mu_w} I_0\left(\frac{k-\mu_w}{\sigma_w}; \sigma_w\right) - e^{-rT+k}\, \Phi\left(\frac{\mu_w - k}{\sigma_w}\right)$$

$n \ge 1$ $$f_n = e^{-rT+\mu_w} \frac{1}{\sqrt{n!}} \sigma_w I_{n-1}\left(\frac{k-\mu_w}{\sigma_w}; w\right)$$

where $I_n(\mu;\nu)$ are defined by $I_0(\mu;\nu) = e^{-\nu^2/2}\, \Phi(\nu - \mu)$
$n \ge 1$ $I_n(\mu;\nu) = H_{n-1}(\mu)\, e^{\nu\mu}\, \phi(\mu) + \nu I_{n-1}(\mu;\nu)$

Compute the first N coefficients following the above recursion.

Solution's steps:

- Initialization of the f vector, the output.
- Creation of the function $H(n,x)$ to compute the probabilistic standar polynomial $H(n,x) = H_n(x)$
- Computation of first coefficient $f(1) = f_0$ and $I = I_0$

When we will upload I, we will overwrite it because we do not need to keep track of it

- Creation of a loop for n from 1 to N in which we find $f_{(n)} = f_n$ and we update I. Note that we first calculate f_n and, then, we update I, because f_n is based on I_{n-1}.

We return f.

d) Write a function PriceApprox.m that computes the approximation of the call option price in the Jacobi model arising from cutting the sum:

$$\Pi f^{(N)} := \sum_{u=0}^{N} f_u l_u$$

Solution's steps:

- We use the function HermiteMoments.m of part a) which is in charge of calculating the vector $l = (l_u)_{u=0,\dots,N}$

- Initialization of the output price, which corresponds to $\Pi f^{(N)}$. This is a vector because we have one value for each strike K.

- Creation of the loop in charge of calculating the approximation of $\Pi f^{(N)}$. At each step we find $\Pi f^{(N)}$ for a certain K. Note that we always use the same l found before, because it does not depend on K. While the fourier coefficients depend on the strike, so we need to recalculate them every time

$$\Pi f^{N}_{K(i)} = l \cdot f_{K(i)} \quad \text{for each } K(i) \text{ in the strike vector } K$$

(· : dot product)

We return the price vector.

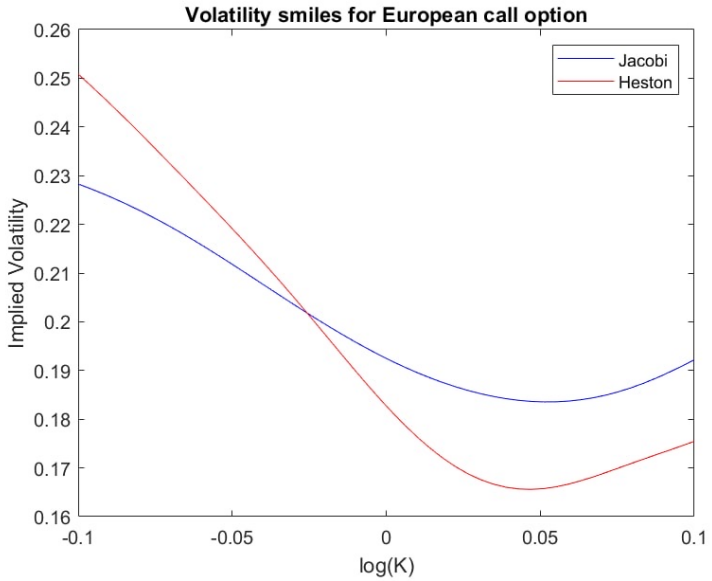
e) In this part, we need to set numerical values for the parameters in order to create some plots.
Specifically we want to compare the Jacobi and the Heston model.
We build 2 plots

- The implied volatility smile
  - In the Jacobi model using the PriceApprox function to find the the approximated prices
  - In the Heston model using the CallHeston function* to find the approximated prices

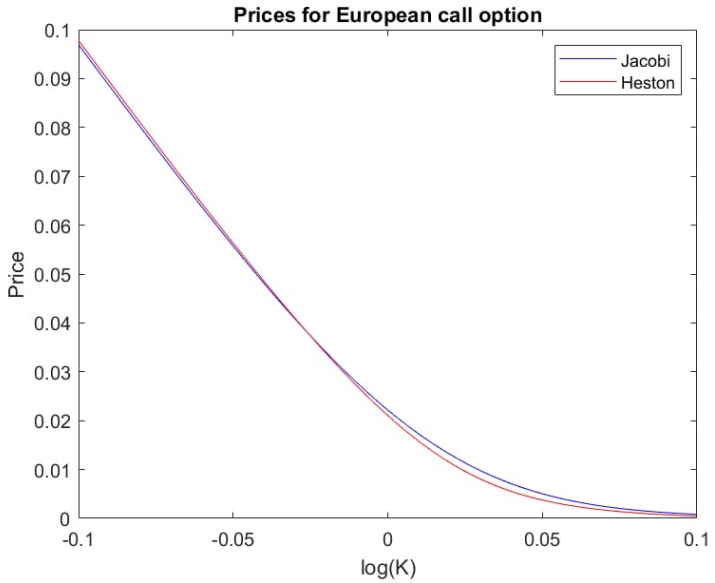
In both cases, from the prices we obtain the IV using blsimpv.

* We define the CallHeston function at the end of the script. This method finds the model characteristic function phi though the formula at slide 27 of lecture 3. Thanks to this phi, we use the Carr-Madan formula to get the approximated prices for the Heston model.

The result is the following:



For the sake of completeness, we plot also the Jacobi and Heston European prices:



f) This point asks to set some parameters and compute the corresponding European call option price using

- the polynomial expansion approach
- the standard MC approach

We need to compute the absolute error and say what we can conclude.

Solution's step:

- We set the parameters and the initial conditions.
- We find μ_W and σ_W, where $\mu_W = \mathbb{E}(X_t)$ and $\sigma_W = \mathrm{Var}(X_t)$, In order to do that we create the function GenJacobiCanonic, which builds the matrix representation of the Jacobi model generator with canonic polynomial basis up to second degree. Once we have this matrix G, we find the moments in M. Note that M(3) corresponds to X_0 and M(6) to X_0^2 $\Rightarrow$ $\mu_W = M(3)$, $\sigma_W = \sqrt{M(6) - M(3)^2}$

- The polynomial expansion price can be easily found using PriceApprox
- We compute the Monte Carlo simulation using SimSDEJacobi, which simulates the paths of X_t. From X_t, we can extract the MC pricing finding the present value:

$$price\_mc = e^{-rT}\, \mathbb{E}[\max(0, \underbrace{e^x - e^k}_{S_t = e^{X_t}\ \forall t})]$$

- Once we have the prices, the absolute error can be computed as:

$$abs(price\_polynomial - price\_montecarlo)$$

In order to have a deeper view of the two approaches, we display also the time needed for each of them.
We also tried to change the truncation level for the polynomial approach and the number of simulations for MC. Please find more details in the comment.

Result:

Polynomial expansion price:
0.0969
Monte Carlo simulation price:
0.0965
Absolute Error between polinomial al Monte Carlo price:
4.1434e-04
Time for polynomail : 42.5601
Time for Monte Carlo: 9.4674

What can you conclude?

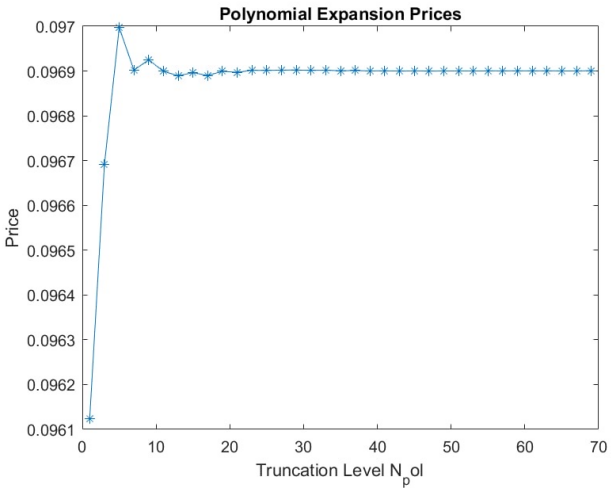
We can conclude that the European call prices computed using the polynomial expansion approach and the Monte Carlo simulation are quite similar for the given parameters. As a matter of fact, the absolute error between the two is in the order of 10^-4. Therefore, the choice of the method does not have much impact on the resulting price in this case.
However, if we compute the time spent by the two methods, our results show that the Monte Carlo approach is remarkably faster (42.5 seconds for the polynomial expansion and 9.4 seconds for the Monte Carlo). Therefore, in this case, maybe it is better to use the Monte Carlo method: this approach gives similar results to the polynomial approach, but it is significantly faster.
However, we would expect the polynomial expansion to be faster. Therefore, we try to change the number N_pol at which we truncate the polynomial expansion and the number N_sim of simulations for the Monte Carlo method, in order to understand how this impacts the results. The outcomes of this trials are saved as PDF files in the “additional experiments point f” folder.

- Decreasing N_pol from 50 to 10 makes the process remarkably faster (3.6 seconds), while the price remains almost unchanged. So, if we are given N_pol=10, we would conclude that, as for 50, the absolute error is in the order of 10^-4 but, in this case, the polynomial approach would be the best in terms of runtime.

- We also try to increase N_pol from 50 to 70. This does not affect the price much either and, therefore, it makes no sense: the process is remarkably slower, and the precision does not improve.

- For the Monte Carlo approach, increasing N_sim to 1e8, the outputs are similar: obviously the runtime remarkably rises, but the price is almost the same.

We conclude that, for the polynomial expansion approach, we get diminishing returns for larger N_pol with a sweet spot that we expect to be in the range of 10. We show that plotting the polynomial expansion price depending on the truncation level:



For the Monte Carlo approach instead, it is not convenient to increase the number of simulations.