A mean-reverting model with jumps for energy commodities: calibration, simulation and implementation

Matteo Gardini[*]

October 21, 2024

Abstract

In this document we discuss about a possible version of the model presented in [10, Chapter 3] and how it can be implemented. We discuss about mathematical, numerical and practical issues without pretending to be neither exhaustive nor formal.

Keywords: Stoachastic processes, Monte Carlo simulations, Martingale condition, jump processes.

1 Introduction

The main idea of this study was try to develop a model for prices which exhibits both a diffusive dynamics, a mean-reversion and jumps preserving the mathematical tractability, guarantying a reasonable difficulty in calibration and providing efficient Monte Carlo simulations.

Developing a stochastic process in easy from a theoretical point of view. You can specify the dynamic you prefer, for example if $S = \{S(t); t \geq 0\}$ is the spot price process you can define $X(t) = \log S(t)$ and impose a dynamic for X of the form:

$$dX(t) = \mu(t, X(t))dt + \sigma(t, X(t))dW(t) + f(t, X(t))dN(t), \tag{1}$$

with $\mu(t, x), \sigma(t, x)$ and $f(t, x)$ satisfying some "technical regularity conditions" so that everything is well defined (see Sato [13] and Cont and Voltchkova [6]). Now, once we have defined the process in Equation (1) we have several issues:

- Can we solve Equation (1)? If yes, we can develop a simulation scheme which allows us to exactly simulate process trajectories. If not, we can discretize the equation by using an Euler schema (or a Millstein one) paying a discretization error as explained in Seydel [17]. In any case, simulations seems not to be an issue.

- Can we calibrate the model, namely, can we infer the form of μ, σ and the parameters if the Poisson process $N = \{N(t); t \geq 0\}$? If we are not able to fit parameters on real data, such a model would be completely useless.

[*]Eni Plenitude, Via Ripamonti 85, 20136, Milan, Italy, email matteo.gardini@eniplenitude.com

- Typically in such markets a particular condition is required. Given a spot market S and a forward market where products where $F(t_0, t)$ denotes the future price at time t_0 of energy delivered at time t in the future it is common to require that:

$$\mathbb{E}\left[S(T)|\mathcal{F}(t_0)\right] = F(t_0, T). \tag{2}$$

  This condition simply states that the best estimation for the spot price at time T, given the information at time t_0 (namely the filtration $\mathcal{F}(t_0)$), is the today future price $F(t_0, T)$ which might be reasonable in a risk-neutral world. See Benth et al. [2, Chapter 1]. In order to guarantee that this is true we can proceed as follow.

  1. Consider the spot price modeled as:

  $$S(t) = F(0, t)e^{X(t)+h(t)}, \qquad t \geq 0,$$

     where $h(t)$ is a deterministic function and $F(0, t)$ is the today future price of the commodity at time t (which is known from the market and might be constructed from the quoted futures market as presented for example in Benth et al. [2, Chapter 7]).
  2. Impose the condition in Equation (2):

  $$\mathbb{E}\left[F(0, t)e^{X(t)+h(t)}|\mathcal{F}(0)\right] = F(0, t).$$

     which means that we have to impose that:

  $$h(t) = -\log \phi_{X(t)}(-i),$$

     where $\phi_{X(t)}$ is the characteristic function of X at time t and i is the imaginary unit. So, in order to impose the condition in Equation (2) we must be able to compute the characteristic function of X at time t and this should be done in an analytic way in order to avoid further numerical issues.

  For this reason, in order to have a tractable model, we need to choose a process such that its characteristic function at time t can be computed in analytic way and this somehow restricts the set of process we can select. Note that $h(t)$ can be easily computed numerically. For example, in a very rough way, by using N simulation we can approximate $h(t)$ by:

$$h(t) = -\log \phi_{X(t)}(-i) = -\log \mathbb{E}\left[e^{X(t)}\right] \approx \frac{1}{N}\sum_{i=1}^{N} e^{X_i(t)},$$

  but this requires a huge number of simulations in order to guarantee a good approximation.

The literature we have been inspired consists in the following works: Kluge [10], Cont and Voltchkova [6], Benth et al. [2], Barndorff-Nielsen [1], Sabino and Cufaro-Petroni [12], Gardini et al. [9], Sabino [11], Seifert and Uhrig-Homburg [16], Cartea and Figueroa [5], Schwartz and Smith [15], Schwartz [14], Cufaro Petroni and Sabino [7] and Böerger et al. [4].

2 Mathematical aspects and notes

First of all, we need to recall what do we mean by integration of a deterministic function f with respect to a Poisson process $N = \{N(t); t \geq 0\}$:

$$Y(t) = \int_0^t f(s)dN(s) = \sum_{i=1}^{N(T)} f(T_i)$$

where $\{T_i\}_{i \geq 1}$ enumerates the jumps of the process N.

If we consider a more general jump process J such that its path are right continuous with left limit and we denote by $\Delta J = J(t) - J(t-)$ and let $\Phi(s)$ be an adapted process to a given filtration, we can define the integral withe respect to the jump process as:

$$\int_0^t \Phi(s)dX(s) = \sum_{0<s\leq t} \Phi(s)\Delta J, \tag{3}$$

namely the sum of the function valuated at jump time, times the size of the jump at the jump time s.

For example, following Shreve [18, Chapter 11], if $X(t) = N(t) - \lambda t$ and $\Phi(s) = \Delta N(s)$ (i.e. $N(s) = 1$ if we have a jump at s and zero otherwise) the integral is given by:

$$\int_0^t \Phi(s)dN(s) = \sum_{0<s\leq t} (\Delta N(s))^2 = N(t).$$

Hence, in a very naive way we can think of the stochastic integral with respect a jump process as the sum of something (deterministic or stochastic) with a random number of terms. (Note that this holds only for jump processes with finite number of jumps in a finite time interval, i.e. for Lévy processes with finite activity).

Hence we define a process of the following form:

$$dZ(t) = -\alpha Z(t)dt + \sigma dW(t) + J(t)dN(t),$$

which can be written as:

$$Z(t) - Z(0) = \int_0^t \alpha Z(s)ds + \int_0^t \sigma dW(s) + \sum_{0<s\leq t} J(s)\Delta N(s),$$

with $\Delta N(s) = 1$ and J are jumps with a given jump size distribution (normal, exponential and so on). Does a solution of such an equation exist? In order to find the solution we need a sort of Itōś formula for jump processes.

Theorem 2.1. *(Shreve [18, Theorem 11.5.1]) Let $X(t)$ a jump process and $f(x)$ a function such that $f'(x)$ and $f''(x)$ exist and are continuous. Then:*

$$\begin{aligned} f(X(t)) - f(X(0)) = &\int_0^t f'(X(s))dX^c(s) + \frac{1}{2}\int_0^t f''(X(s))dX^c(s)dX^c(s) \\ &+ \sum_{0<s\leq t} \left[f(X(s)) - f(X(s-))\right], \end{aligned}$$

where $X^c(t)$ represent the continuous part of X.

If the function $f = f(x,t)$ the usual derivative with respect to time appears. Consider the price $Z(t)$ and consider the function $f(t,x) = e^{\alpha t}x$.

$$Z(t) - Z(0) = \int_0^t \alpha Z(s)ds + \int_0^t \sigma dW(s) + \sum_{0<s\leq t} J(s)\Delta N(s),$$

By using the Theorem 2.1 we have that:

$$\begin{aligned}
f(t,X(t)) - f(0,X(0)) &= \int_0^t \alpha e^{\alpha s} Z(s)ds + \int_0^t e^{\alpha s} dZ^c(s) + \sum_{i=1}^{N(t)} \left[f(t,X(\tau_i)) - f(t,X(\tau_i-))\right] \\
&= \int_0^t e^{\alpha s}\sigma dW(s) + \sum_{i=1}^{N(t)} e^{\alpha\tau_i} Z\left(\tau_i\right) - e^{\alpha\tau_i -} Z\left(\tau_i -\right) \\
&= \int_0^t e^{\alpha s}\sigma dW(s) + \sum_{i=1}^{N(t)} e^{\alpha\tau_i} J\left(\tau_i\right).
\end{aligned}$$

Considering $Z(0) = Z_0$ we have that:

$$Z(t) = Z_0 e^{-\alpha t} + \sigma \int_0^t e^{-\alpha(t-s)} dW(s) + \sum_{i=1}^{N(t)} e^{-\alpha(t-\tau_i)} J\left(\tau_i\right),$$

and this can be used to easily simulate the process.

2.1 The model

Assume that $Z(0) = 0$ and call $X(t) = \int_0^t e^{-\alpha(t-s)}\sigma dW(s)$ and $Y(t) = \sum_{i=1}^{N(t)} e^{-\alpha(t-\tau_i)} J\left(\tau_i\right)$. We model the spot price $S = \{S(t); t \geq 0\}$ as:

$$S(t) = F(0,t)e^{Z(t)+h(t)} = F(0,t)e^{X(t)+Y(t)+h(t)},$$

where $F(0,t)$ is today forward price for time t and $h(t)$ is the *drift-corrector* which has to be determined in order to guarantee the condition in Equation (2). Starting from:

$$\mathbb{E}\left[F(0,t)e^{X(t)+Y(t)+h(t)} | \mathcal{F}(0)\right] = F(0,t),$$

we get that:

$$h(t) = -\log\phi_{X(t)}(-i) - \log\phi_{Y(t)}(-i).$$

From the theory we have that $X(t) \sim \mathcal{N}\left(0, \bar{\sigma}^2\right)$ where $\bar{\sigma}^2 = \frac{\sigma^2}{2\alpha}(1 - e^{-2\alpha t})$ whereas $Y(t)$ depends on the distribution we choose for jumps. Anyway, the characteristic function of $Y(t)$ can be computed relying on the following proposition.

Proposition 2.2. *[Kluge [10, Lemma 3.4.2]] The characteristic function of Y at time t is given by:*

$$\phi_{Y(t)}(u) = \exp\left(\lambda \int_0^t \left(\Phi_J(ue^{-\alpha s}) - 1\right) ds\right) \tag{4}$$

where $\Phi_J(u)$ is the characteristic function of jumps and λ is the intensity of the Poisson process N.

Here troubles arise: if we want to compute analytically the function $h(t)$ we need to solve the integral in Equation (4) and this is not always possible. For example, no analytic solution of the integral are known if $J \sim \mathcal{N}\left(\mu_J, \sigma_J^2\right)$. In this case some approximation of the characteristic function for high level of mean-reversion rate α are known (see Kluge [10]). The integral can be computed in case in which the jumps are distributed as an exponential with average jump height μ_J, $J \sim \mathcal{E}\left(1/\mu_J\right)$ and the characteristic function is given by:

$$\phi_{Y(t)}(u) = \left(\frac{1 - iu\mu_J e^{-\alpha t}}{1 - iu\mu_J}\right)$$

with $u\mu_J < 1$ as show in Kluge [10, Example 3.4.3]. But in this case, we are considering only upward jump. It is possible to consider also downward jumps hence considering a double exponential distribution with parameters λ_u, λ_d and p, the rate of upward and downward jumps and the probability of observing an upward jump, respectively. The *pdf* of jump in this case is given by:

$$h(t) = -\log \phi_{X(t)}(-i) - \log \phi_{Y(t)}(-i).$$

$$f_J(x) = p\lambda_u e^{-\lambda_u x}\mathbb{1}_{x \geq 0} + (1-p)\lambda_d e^{\lambda_d x}\mathbb{1}_{x<0} \tag{5}$$

and the jump process is defined as:

$$Y(t) = \sum_{i=1}^{N(t)} e^{-\alpha(t-\tau_i)} J(\tau_i).$$

Following Cufaro Petroni and Sabino [8] we have that the characteristic function of $Y(t)$ is given by:

$$\phi_{Y(t)}(u) = \left(\frac{\lambda_u - iue^{-\alpha t}}{\lambda_u - iu}\right)^{p\frac{\lambda}{\alpha}} \left(\frac{\lambda_d + iue^{-\alpha t}}{\lambda_d + iu}\right)^{(1-p)\frac{\lambda}{\alpha}}$$

Once that $\phi_X(t)(u)$ and $\phi_Y(t)(u)$ are known in closed form the drift corrector $h(t)$ can be computed and hence the condition in Equation (2) is achieved.

Remark 1. Following the idea in Schwartz and Smith [15] we can also add a long-term volatility σ_l to the model, which leads to:

$$Z(t) = Z_0 e^{-\alpha t} + \sigma \int_0^t e^{-\alpha(t-s)} dW(s) + \int_0^t \sigma_l dW_l(t) + \sum_{i=1}^{N(t)} e^{-\alpha(t-\tau_i)} J\left(\tau_i\right). \tag{6}$$

In order to guarantee the condition in Equation (2) we need to modify the drift corrector $h(t)$ by adding a component related to the long-term diffusive part:

$$H(t) = \int_0^t \sigma_l dW_l(t),$$

which characteristic function is given by:

$$\phi_{H(t)}(u) = e^{-\frac{u^2\sigma_l^2 t}{2}}.$$

The drift corrector $h(t)$ is given by:

$$h(t) = -\log \phi_{X(t)}(-i) - \log \phi_{Y(t)}(-i) - \log \phi_{H(t)}(-i).$$

3 Monte Carlo simulations

The starting point for Monte Carlo simulation is to be able to properly simulate the process:

$$dZ(t) = -\alpha Z(t)dt + \sigma dW(t) + J(t)dN(t), \tag{7}$$

which solution is:

$$Z(t) = Z_0 e^{-\alpha t} + \sigma \int_0^t e^{-\alpha(t-s)} dW(s) + \sum_{i=1}^{N(t)} e^{-\alpha(t-\tau_i)} J(\tau_i), \tag{8}$$

Both Equations (7) and (8) can be used to perform Monte Carlo simulations.

Give a time grid on $[0,T]$, $0 = t_0 < ... < t_N = T$ with uniform time step $\Delta t = T/(N+1)$ the Euler schema associated to Equation (7) is:

$$Z(t+\Delta t) = Z(t) - \alpha Z(t)\Delta t + \sigma\sqrt{\Delta t}Z + J(t)\Delta N(t), \tag{9}$$

where $Z \sim \mathcal{N}(0,1)$ and $\Delta N(t) = N(t+\Delta t) - N(t)$, namely the increments in the number of jumps of the Poisson process in the interval Δt. Clearly, by using the Euler method approximation errors arise.

On the other hand, we can also use Equation (8) to simulate the process. We focus only on the jump part, since the simulation of the Gaussian component $\sigma\int_0^t e^{-\alpha(t-s)}dW(s)$ is trivial.

Assume $Z(0) = 0$ and consider the pure jump process:

$$G(t) = \sum_{\tau_i < t} e^{-\alpha(t-\tau_i)} J_{\tau_i},$$

and

$$\begin{aligned} G(t+\Delta t) &= \sum_{\tau_i < t+\Delta t} e^{-\alpha(t+\Delta t-\tau_i)} J_{\tau_i} = e^{-\alpha\Delta t} \sum_{\tau_i < t+\Delta t} e^{-\alpha(t-\tau_i)} J_{\tau_i} \\ &= e^{-\alpha\Delta t}\left(\sum_{\tau_i<t} e^{-\alpha(t-\tau_i)} J_{\tau_i} + \sum_{t\le\tau_i<t+\Delta t} e^{-\alpha(t-\tau_i)} J_{\tau_i}\right) \\ &= e^{-\alpha t}\left(G(t) + \sum_{t\le\tau_i<t+\Delta t} e^{-\alpha(t-\tau_i)} J_{\tau_i}\right). \end{aligned}$$

If a jump occur between $[t, t+\Delta t]$ sum $e^{-\alpha t}e^{-\alpha(t-\tau_*)}J_{\tau_*}$.
By calling:

$$X(t) = \sigma \int_0^t e^{-\alpha(t-s)} dW(s)$$

we can independently simulate X and G and hence the final process $Z(t)$ is given by:

$$Z(t) = Z_0 e^{-\alpha t} + X(t) + G(t).$$

Remark 2. Recall that if $N = \{N(t); t \geq 0\}$ is a Poisson process with intensity λ, the number of jumps o a given time interval $[0, T]$ are uniformly distributed. Since the time grid is discrete, given a sequence of jump times $\tau_i,\ i = 0, \ldots, n$ you have to match them with the points on the discrete time grid. If the time grid is not thick enough multiple jumps may occur is the same interval $[t, t + \Delta t]$.

4 Calibration

From the previous section we can deduce that in this model we have the following vector of parameter to fit: $\boldsymbol{\theta} = (\alpha, \sigma, p, \lambda, \lambda_u, \lambda_d)$ which might be calibrated on historical time series.

Remark 3. Here we do not discuss about calibration on derivatives, which of course can be done in the usual way. Probably some hints can be deduced from [10]. The procedure is the usual one:

- Find a market in which vanilla products $\left\{V_i^{mkt}\right\}_{i=1}^m$, such as European call and put are quoted.
- Fine an efficient way to compute the value of these products (for example by finding closed form solutions or by using a FFT approach).
- Solve the optimization problem given by:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\Theta} \sum_{i=1}^{m} \left(V_i^{\boldsymbol{\theta}} - V_i^{mkt}\right)$$

Remark 4. In case we include the long-term volatility parameter σ_l we can fit this parameter from long-term futures or from options with long maturity. Note that this parameter regulates how much simulations spreads as time goes on.

Now we go back on the calibration procedure, which is inspired by Kluge [10]. We think that the following procedure might be largely improved but here we keep things simple and rough. We focus on calibration on daily power spot prices, but the procedure can be adopted for any commodity.

Assume that a series of daily spot prices $S = \{S(t); t \geq 0\}$ is given. We model the spot price as:

$$S(t) = e^{\Lambda(t)+Z(t)} = e^{\Lambda(t)+X(t)+G(t)},$$

where we assumed $Z(0) = 0$ and where $\Lambda(t)$ is a deterministic seasonal component. Hence:

$$\log S(t) = \Lambda(t) + X(t) + G(t).$$

For the deterministic component many different form can be assumed. For example Benth et al. [2, Chapter 7] assumes

$$\Lambda(t) = a + b\cos\left((t + \omega)\,\frac{2\pi}{365}\right), \tag{10}$$

whereas Seifert and Uhrig-Homburg [16] propose a seasonality which is like:

$$\Lambda(t) = s_1 \sin\left(\frac{2\pi t}{365.25}\right) + s_2 \sin\left(\frac{2\pi t}{365.25}\right) + s_3 \sin\left(\frac{4\pi t}{365.25}\right)$$
$$+ s_4 \sin\left(\frac{4\pi t}{365.25}\right) + \sum_{i \in N_d} \mathbb{1}_i(t) d_i + \mathbb{1}_h(t) w_h + t\mu,$$

where $N_d = \{Mo, Tu - Th, Fr, Sa, Su\}$ is a set of different weekdays accounting for weekly seasonality with indicators $\mathbb{1}_i(t)$. Holiday effects, marked by $\mathbb{1}_h(t)$, can depend on the selected country. In general, different commodities leads to different seasonal component Λ. The set of parameters $\boldsymbol{\eta}$ of the chosen seasonality function $\Lambda(t)$ can be fitted by an least-square technique minimizing the quantity:

$$\arg\min_{\boldsymbol{\eta}} \sum_{t \in [0,T]} \left(\log S(t) - \Lambda(t)\right)^2 .$$

Once that the seasonality has been fitted we can remove it from the observed log-prices series $\log S(t)$ and obtain:

$$X(t) + G(t) = \log S(t) - \Lambda(t).$$

In Figure 1 we show the historical data of log-prices and the fitted seasonality, in the case we assume a functional form for the seasonality $\Lambda(t)$ given by:

$$\Lambda(t) = \sum_{i=1}^{7} a_i \mathbb{1}_i(t) + \sum_{w=1}^{52} b_w \mathbb{1}_w(t) + h\mathbb{1}_h(t)$$

where a_i are related to the day of the week, b_w to the week of the years and h is considered only if the day t is an holiday. In Figure 2 we show the stochastic component $X(t) + G(t)$ we obtain once that the seasonality has been removed.

The next step is to fit the parameter of $X(t)$ and those of $G(t)$. Here the problem is that we observe the sum of two processes and not each of them separately.

In order to fit the parameters we use the following brute force procedure.

- Filters out jumps.
- Fit the distribution of the jump-sizes.
- Remove the jumps from the series and what you get is $X(t),\ t \geq 0$.
- Fit the Ornstein-Uhlembeck process on X to get the final parameter.

In order to remove jumps we use a standard procedure which has been described in Cartea and Figueroa [5]. We remove all values larger that 2.5 standard deviation, hence we recompute the standard deviation and we iterate the procedure until no values have to be removed. Typically this algorithm ends in few iterations. Once that the jumps have been found we can estimate the parameter as follow:

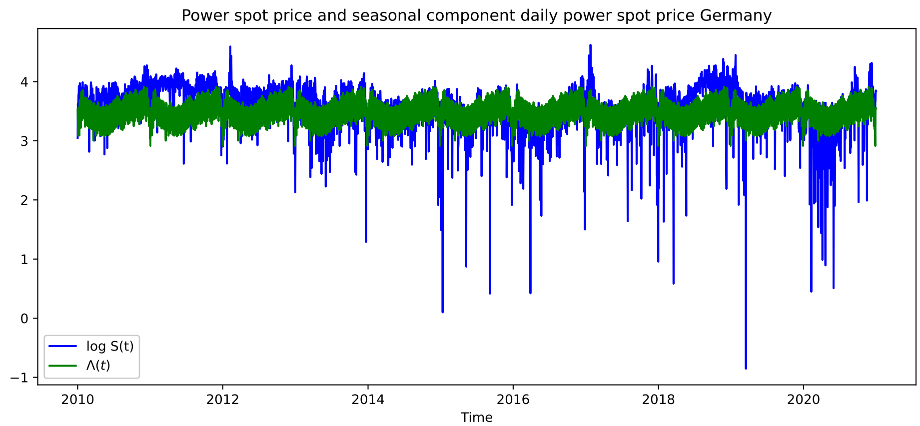

Figure 1: Historical data versus deterministic seasonality component of the $\log S(t)$ series of the German power spot price.

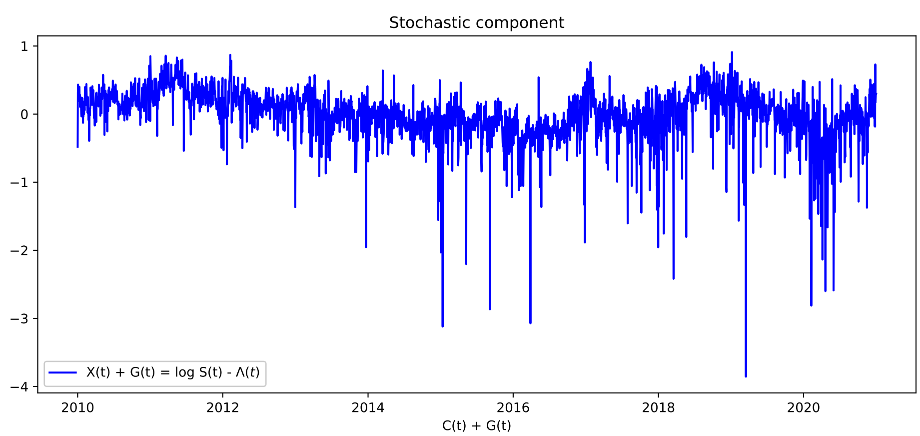

Figure 2: Stochastic component of the $\log S(t)$ series: $X(t) + G(t)$.

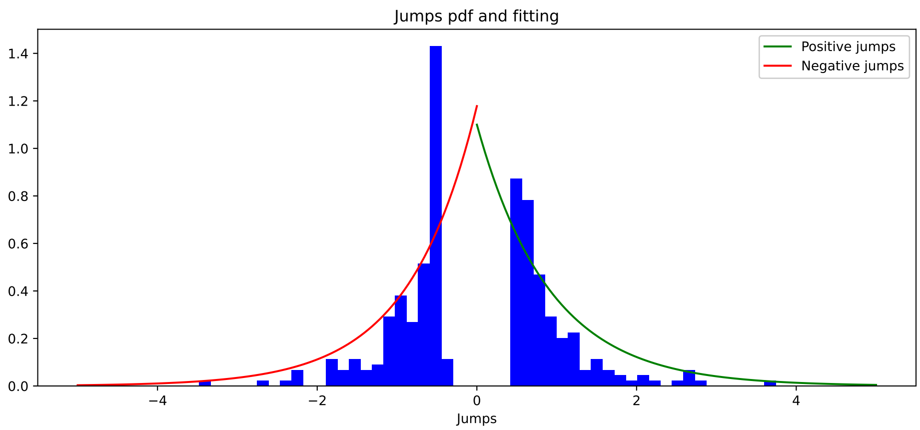

Figure 3: Stochastic component of the $\log S(t)$ series: $X(t) + G(t)$.

1. λ: this is the intensity of the Poisson process. We have the number of jumps and knowing that $\mathbb{E}[N(T)] = \lambda T$ we can estimate λ as:

$$\lambda = \frac{\sum_{\tau_k} \mathbb{1}_{\tau_k} < T}{T}$$

where τ_k are the jumps and T is the number of years in the time series. Hence, we are estimating the frequency of jumps.

2. p: the probability that a jump is positive can be simply estimated by counting the percentage of positive jumps in the data-set:

$$p = \frac{\sum_{\tau_k} \mathbb{1}_{\tau_k < T} \cdot \mathbb{1}_{J_{\tau_k} > 0}}{\sum_{\tau_k} \mathbb{1}_{\tau_k < T}}$$

3. λ_p, λ_d: assuming that jumps are distributed according to a double exponential distribution positive jumps have a *pdf* of the form $f(x) = \lambda_p e^{-\lambda_p x}$ and $\mathbb{E}[J_{\tau_k} | J_{\tau_k} > 0] = \frac{1}{\lambda p}$ the parameter λ_p can be easily estimated by:

$$\frac{1}{\lambda_p} = \frac{\sum_{\tau_k} J_{\tau_k} \mathbb{1}_{\tau_k < T} \cdot \mathbb{1}_{J_{\tau_k} > 0}}{\sum_{\tau_k} \mathbb{1}_{\tau_k < T} \cdot \mathbb{1}_{J_{\tau_k} > 0}}.$$

A very complicated formula to say a simple fact: $\frac{1}{\lambda_p}$ is the average value of the positive jumps. For negative jumps the procedure is the same.

If Figure 3 we show the jumps' distribution fitting.

The last step is the fitting of the Gaussian part, which is characterized by an Orstein-Uhlembeck process. Calibration is based on linear-regression and many methods can be found in literature: a very intuitive method is presented in Brigo et al. [3]. Here two different approaches are possible: we can fit the process on $X(t) + G(t)$ or only on $X(t)$

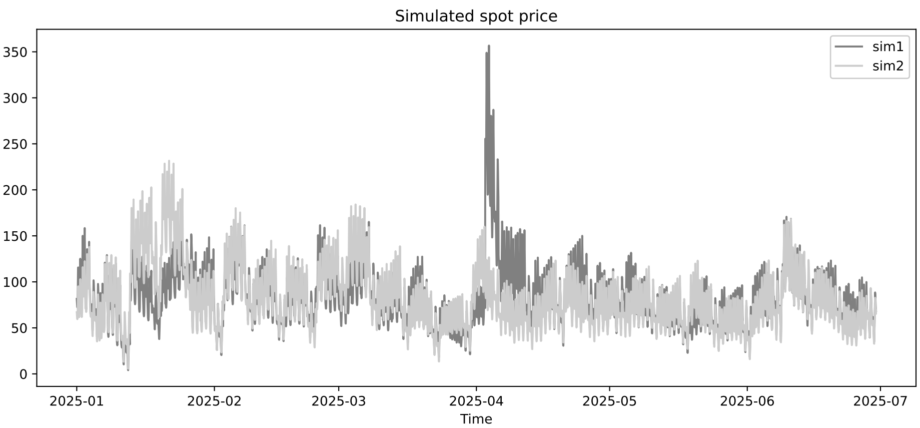

Figure 4: Some simulation of the hourly spot price with double exponential jumps.

(which is the original series once that the seasonality component and jumps are removed). Probably the best choice is to fit on $X(t)$ but the mean-reverting parameter α also acts on jumps: so it is not clear to us what is the best practice. If we fit α on $X(t) + G(t)$ we expect a larger value that fitting only on $X(t)$.

On the other hand, such type of models in many practical situations are subjected to what is called "expert-calibration". This simply means that the trader, the quant analyst or the risk-manager has some sensibility from the market, so that those parameter are chosen such that some market evidence are respected. Let's consider some examples.

- λ: as mentioned before it represents the expected number of jumps for unit of time: $\mathbb{E}[N(T)] = \lambda T$. First of all, it is quite arbitrary to decide what a jump is. Intuitively it is clear, mathematically everything can be a jump! If a trader expects say ten jumps per year the parameter λ will be set equal to ten.

- α: it represents how fast the process reverts to the mean. Indeed if we consider the halving time of jump after Δt^*:

$$\begin{aligned} X(\Delta t^*) &= X(0)e^{-\alpha\Delta t^*}, \\ X(\Delta t^*) &= \frac{X(0)}{2}, \\ \frac{X(0)}{2} &= X(0)e^{-\alpha\Delta t^*}, \end{aligned}$$

  which leads to $\alpha = \log(2)/\Delta t^*$.

- p: if a quant-analyst expects that upward jumps are more frequent that downward jumps, he might be confident to set, for example, $p = 0.75$.

If Figure 4, we show some plots generated by the model.

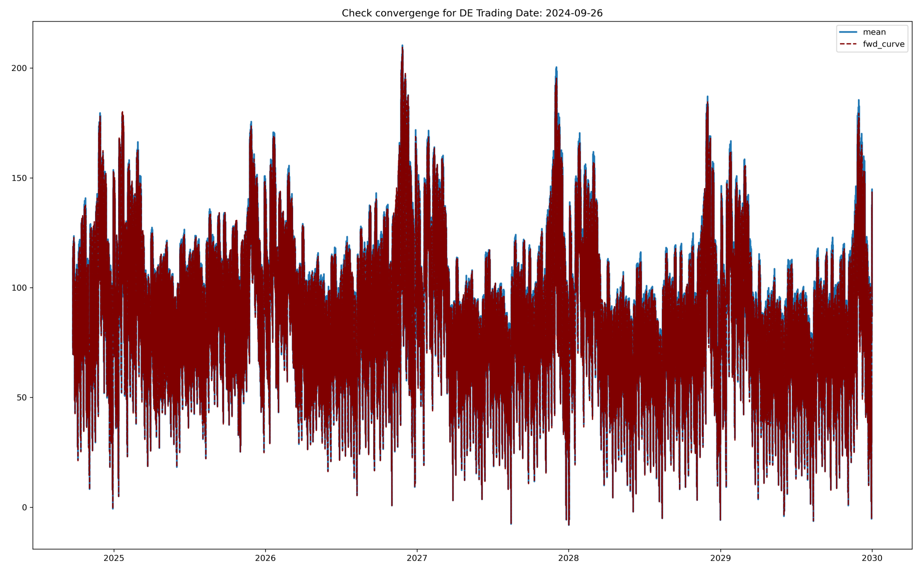

Figure 5: Some simulation of the hourly spot price with double exponential jumps.

5 Considerations and conclusions

The model we propose offers an effective method for incorporating jumps into a mean-reverting process. As with any model, it has both strengths and limitations. It is straightforward to implement, mathematically tractable, and relatively simple to calibrate using historical data. Additionally, when using double exponential jumps, the characteristic function can be derived analytically, allowing us to satisfy the condition:

$$\mathbb{E}\left[S(T)|\mathcal{F}(t)\right] = F(t,T).$$

as shown is Figure 5.

Monte Carlo simulations are computationally efficient, though the challenge lies in evaluating the sum of jumps. It is likely that a more efficient implementation than the one we used could address this. The model is also applicable for simulating forward prices, and calibration under the risk-neutral measure $\mathbb{Q}$ can likely be achieved using standard techniques.

The most challenging aspect of the model, as with many others, lies in the calibration process. As previously mentioned, the calibration of jumps is not the most precise, and fitting the Ornstein-Uhlenbeck process introduces further complications. Specifically, it is unclear whether the mean-reverting parameter should be calibrated on the series without jumps or whether jumps, given their interaction with mean reversion, should be included in the calibration procedure. In some cases, applying the calibration method outlined in Section 4 results in unrealistic parameter values. Consequently, expert calibration is often required, though this can introduce an element of subjectivity into the parameter estimation process.

Furthermore, the fitting of seasonality can also pose difficulties. If the seasonal patterns vary across years, the calibration might be misleading. For example, energy spot prices in Europe during 2021 and 2022 differed significantly from those in previous years due to extraordinary economic and geopolitical factors such as inflation and the war between Ukraine and Russia. One possible solution is to consider different seasonality patterns for each year, or to exclude these outlier years from the calibration, although the latter approach may result in the loss of valuable information.

In conclusion, this model represents a simple and effective approach for incorporating jumps into a mean-reverting process while maintaining mathematical tractability and requiring only a reasonable level of effort for calibration. However, the selection of parameters remains a challenging aspect of the model, particularly in ensuring realistic outcomes.

6 Code

The Python code implementing the model following an object oriented approach.

6.1 utilities module

Some utilities functions we need for the model.

```python
import datetime as dt
import pandas as pd
import numpy as np
import logging
import sys
import matplotlib.pyplot as plt
import os


def get_last_fwd_curve_in_folder(folder_path: str, prefix: str, trading_date: str="") -> str:

    """
    Get the last forward curve in the folder
    :param folder_path: path to the folder
    :param prefix: prefix of the forward curve
    :param trading_date: trading date
    :return: the name of the forward curve and the selected trading date
     """

    if trading_date == "":
        trading_date = max(
            [x.split("_")[-1].split(".")[0] for x in os.listdir(folder_path) if 'all_curves' in x]
        )

        curve_name = f"{prefix + trading_date}.xlsx"

        return curve_name, trading_date
```

```python


def setup_custom_logger(name):
    """
    This function set the logging format
    :param name: name of logger
    :return: logger
    """
    formatter = logging.Formatter(fmt='%(asctime)s %(levelname)-8s %(message)s',
                                  datefmt='%Y-%m-%d %H:%M:%S')
    now_time = dt.datetime.now()
    now_time = now_time.strftime("%Y%m%d%H%M%S")
    log_file_name =  name + "_" + now_time + ".txt"
    handler = logging.FileHandler(log_file_name, mode='w')
    handler.setFormatter(formatter)
    screen_handler = logging.StreamHandler(stream=sys.stdout)
    screen_handler.setFormatter(formatter)
    my_logger = logging.getLogger(name)
    my_logger.setLevel(logging.DEBUG)
    my_logger.addHandler(handler)
    my_logger.addHandler(screen_handler)
    return my_logger


def create_lambda_matrix(daily_calendar, country='IT'):

    """
    Given an daily_calendar this function builds a boolean matrix LAMBDA of size n_hours x 106.
    colums 0: a general intercept
    columns: 24-31 are the days of the week
    columns: 59-106: week of the year
    :param daily_calendar:
    :param country: country. To consider if in a second moment you want also consider holidays
    :return: the LAMBDA MATRIX described above:
    [[1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ...]
     [1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ...]
     [1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ...]
     [1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ...]]
    """

    # Compute the number of hours
    n_days_tot = len(daily_calendar)

    # Create the weekend matrix
```

```
    weekday_matrix = np.zeros((n_days_tot, 6))
    for i in range(6):
        idx_ = daily_calendar.day_of_week == i
        weekday_matrix[idx_, i] = 1

    # Create the week calendar
    week_matrix = np.zeros((n_days_tot, 52))
    week_calendar = daily_calendar.isocalendar().week
    week_calendar = week_calendar.to_numpy()
    # Assume there are always 52 weeks in a year. If more, take the
    modulo operatio
    week_calendar = np.mod(week_calendar, 52)
    # week_calendar = daily_calendar.weekofyear

    for i in range(52):
        idx_ = week_calendar == i + 1
        week_matrix[idx_, i] = 1

    # Create the matrix
    ones_matrix = np.ones((n_days_tot, 1))
    lambda_matrix_seasonality = np.hstack(
        [ones_matrix, weekday_matrix, week_matrix])

    return lambda_matrix_seasonality
```

6.2 market_utilities.py module

Some utilities functions we need to manipulate the market in a proper way.

```python
import pandas as pd
import json
import copy
import numpy as np
import utilities as ut
import os
from scipy.optimize import curve_fit
import sys



def get_last_fwd_curve_file(in_file_path:str, trading_date: str):
    """
    Return the name of the file with the last trading date for a given curve.
    :param in_file_path: path where yoy have to look for
    :return: the file name with the path
    """
    if trading_date == "":
        trading_date = max(
            [x.split("_")[-1].split(".")[0] for x in os.listdir(in_file_path) if x.startswith("all_curves")]
        )


    in_file_name = f"all_curves_{trading_date}.xlsx"
    in_file = os.path.join(in_file_path, in_file_name)


    return in_file, trading_date


class ForwardCurve:
    """
    This class models the Forward curve with different granularity
    """

    def __init__(self, mkt_name: str, granularity: str, trading_dates, values, trading_date=None):
        """

        :param mkt_name: Market name
        :param granularity: granularity of the forward curve: daily, hourly, monthly
        :param trading_dates: trading dates of the forward curve
        :param values: prices of the forward curve
        :param trading_date: trading date of the forward curve
        """
        self.trading_date = trading_date
```

```python
        self.mkt_name = mkt_name
        self.granularity = granularity
        self.trading_dates = trading_dates
        self.values = values

    def aggregate_fwd_curve(self, frequency="D"):
        """
        Given a forward curve, return the aggregate forward curve to a frequency given by "frequency"
        :param frequency: aggregation frequency: "D","H","M"
        :return: a dataframe containing the curve aggregated on "frequency" base
        """

        df_fwd_curve = pd.DataFrame({"trading_dates": self.trading_dates, "values": self.values})
        df_fwd_curve.set_index('trading_dates', inplace=True)
        return df_fwd_curve.groupby(pd.Grouper(freq=frequency)).mean()


class HistoricalSpotCurve:
    """
    This class models the Historical spot curve with different granularity
    """

    def __init__(self, mkt_name: str, granularity: str, trading_dates, values):
        """

        :param mkt_name: Market name
        :param granularity: granularity of the forward curve: daily, hourly, monthly
        :param trading_dates: trading dates of the forward curve
        :param values: prices of the forward curve
        """

        self.mkt_name = mkt_name
        self.granularity = granularity
        self.trading_dates = trading_dates
        self.values = values


    def __repr__(self):
        return f"Market:{self.mkt_name}, granularity {self.granularity}, start_date {self.trading_dates[0]}, end_date {self.trading_dates[-1]}"

    def aggregate_curve(self, frequency="D"):
        """
```

```
        Given a forward curve, return the aggregate forward curve to a frequency given by "frequency"
        :param frequency: aggregation frequency: "D","H","M"
        :return: a dataframe containing the curve aggregated on "frequency" base
        """

        df_fwd_curve = pd.DataFrame({"trading_dates": self.trading_dates, "values": self.values})
        df_fwd_curve.set_index('trading_dates', inplace=True)

        df_curve_aggregate = df_fwd_curve.groupby(pd.Grouper(freq=frequency)).mean()
        self.granularity = frequency
        self.trading_dates = df_curve_aggregate.index
        self.values = df_curve_aggregate.values
        return df_curve_aggregate

    @staticmethod
    def smooth_seasonality(t_data: pd.core.indexes.datetimes.DatetimeIndex,
                           s1: float, s2: float, s3: float, s4: float, s5: float, s6: float):
        """ Form of the smooth seasonality: See Saifert and Uhrig-Homburg: Modelling Jumps in
        Electricity Prices: and empirical evidence
        :param t:
        :param s1:
        :param s2:
        :param s3:
        :param s4:
        :param s5:
        :param s6:
        :return: the valuation of the seasonality
        """

        t_day = t_data.day_of_year
        t_week = t_day.wee


        return s6 + s5*t_day + s1*np.cos(2*np.pi*t_day/365.25) + s2*np.sin(2*np.pi*t_day/365.25) \
               + s3*np.sin(4*np.pi*t_day/365.25) + s4*np.cos(4*np.pi*t_day/365.25)


    def fit_seasonality(self):
        """
        Fit the seasonality over a time series data vector
        :return:
        """
        #TODO: allow the fitting on more than one year of date
```

```python


        t_data = self.trading_dates
        y = self.values
        initial_guess = (0, 0, 0, 0, 0, 0)
        params, covariance = curve_fit(f=self.smooth_seasonality, xdata=t_data, ydata=y.flatten(), p0=initial_guess)


class LogReturns:
    """
    This class models the LogReturns
    """

    def __init__(self, mkt_name: str, granularity, trading_dates, values, prices: pd.DataFrame, T=None):
        """

        :param mkt_name: Market name
        :param granularity: granularity of log-returns: daily, hourly, monthly
        :param trading_dates: trading dates of the log-returns
        :param values: log-returns
        :param prices: prices from which you have computed the log-returns
        """
        self.annualization_factor = np.sqrt(252)
        self.mkt_name = mkt_name
        self.granularity = granularity
        self.trading_dates = trading_dates
        self.values = values
        self.term_structure_volatility = None
        self.fwd_prices = prices
        self.rolling_std_volatility = None

        # Set the time to maturities. Here we have assumed that the time step is a month
        # TODO: allow possible different time to maturity
        if T is None:
            self.T = np.cumsum(1/12.*np.ones((1, np.shape(values)[1])))
        else:
            self.T = T

    def filter_outliers_log_returns(self, threshold: float = 0.11):
        """
        Remove the log-returns bigger that a give theshold
        :return:
        """

        # Compute the rows to keep
```

```
        rows_to_keep = np.all(np.abs(self.values) <= threshold, axis=1)

        # Filter the array based on the condition
        self.values = self.values[rows_to_keep]
        self.trading_dates = self.trading_dates[rows_to_keep]

    def compute_rolling_volatility(self, backward_period: int):
        """
        Compute the rolling volatility of log-returns
        :param backward_period: as integer. Period of computation of the rolling volatility
        :return:
        """
        columns_name = self.fwd_prices.columns
        df_log_returns = pd.DataFrame(data=self.values, columns=columns_name, index=self.trading_dates)
        self.rolling_std_volatility = df_log_returns.rolling(window=backward_period).std()*self.annualization_factor
        self.rolling_std_volatility.dropna(inplace=True)


    def compute_mkt_time_structure(self):
        """
        Compute the market term structure
        :return:
        """

        self.term_structure_volatility = np.std(self.values, axis=0) * self.annualization_factor
        term_structure_vol = self.term_structure_volatility
        return term_structure_vol


class Market:
    """ This is the Market class and contains the market of a given country or products.
    In particular, it contains both monthly and hourly forward curves, the market_code to identify the market, the
    historical log-returns of all the forward products related to the market_code, tells if the market has peak_values
    and it contains the trading date.
    """

    def __init__(self):
        """
        Inizialization: namely the constructor
        """
        self.monthly_fwd_curves = {}
        self.spot_fwd_curves = {}
        self.mkt_code = ""
        self.historical_logrets = {}
```

```
        self.has_peak = True
        self.sub_markets = ["BL", "PK", "OP"]
        self.trading_date = None
        self.historical_spot_prices = {}
        self.spot_granularity = ""



class MarketBuilderExcel:

    def __init__(self):

        # Create a default Market object
        self.market = Market()

    def _get_monthly_fwd_curve(self, mkt_code: str, in_file: str, in_sheet: str, trading_date: str):
        """
        Get the monthly forward curve from Excel
        :param mkt_code: market code, namely the name of the market to process
        :param in_file: input file with path
        :param in_sheet: input sheet containing the forward curve
        :param trading_date: trading date of the forward curve
        :return:
        """

        # Read the monthly forward curve and assign it. Filter the forward curve corresponding to the mkt_code
        dataframe_fwd_curve = pd.read_excel(in_file, index_col="date", sheet_name=in_sheet,
                                            parse_dates=["date"]).filter(regex=mkt_code)

        # Loop over the curves
        n_curves = dataframe_fwd_curve.shape[1]
        # get the calendar
        trading_dates = dataframe_fwd_curve.index
        for product in dataframe_fwd_curve.columns:
            obj_fwd_curve = ForwardCurve(mkt_name=product, granularity="M",
                                         trading_dates=trading_dates, values=dataframe_fwd_curve[product].values,
                                         trading_date=trading_date)

            self.market.monthly_fwd_curves[product] = obj_fwd_curve

    def _get_hourly_fwd_curve(self, mkt_code: str, in_file: str, in_sheet: str, trading_date: str):
        """
        Get the hourly forward curve from Excel
```

```
        :param mkt_code: market code, namely the name of the market to process
        :param in_file: input file with path
        :param in_sheet: input sheet containing the forward curve
        :param trading_date: trading date of the forward curve
        :return:
        """

        # Read the hourly forward curve and assign it. Filter the forward curve corresponding to the mkt_code
        dataframe_fwd_curve = pd.read_excel(in_file, index_col="date", sheet_name=in_sheet,
                                            parse_dates=["date"]).filter(regex=mkt_code)

        # Loop over the curves
        trading_dates = dataframe_fwd_curve.index
        for product in dataframe_fwd_curve.columns:
            obj_fwd_curve = ForwardCurve(mkt_name=product, granularity="H",
                                         trading_dates=trading_dates, values=dataframe_fwd_curve[product].values,
                                         trading_date=trading_date)

            self.market.spot_fwd_curves[product] = obj_fwd_curve

    def _get_log_returns(self, mkt_code: str, in_file: str, in_sheet: str):
        """
        Get the log-returns from Excel
        :param mkt_code: market code, namely the name of the market to process
        :param in_file: input file with path
        :param in_sheet: input sheet containing the forward curve
        :return:
        """

        # Read the log-returns. Filter the forward curve corresponding to the mkt_code
        dataframe_log_returns = pd.read_excel(in_file, index_col="trading_date", sheet_name=in_sheet,
                                              parse_dates=["trading_date"]).filter(regex=mkt_code)
        # get the calendar, namely the trading dates
        trading_dates = dataframe_log_returns.index

        # Always assume that you have PK, OP, BL products and filter the dataframe. If it has not, skip the filling
        for product_type in self.market.sub_markets:
            # Filter OP, PK, BL
            dataframe_log_returns_type = dataframe_log_returns.filter(regex=product_type)
```

```
            # If it is not empty fill it
            if not dataframe_log_returns_type.empty:
                # Create a log-returns object
                mkt_name = self.market.mkt_code + "_" + product_type
                obj_log_returns = LogReturns(mkt_name=mkt_name, granularity="",
                                             trading_dates=trading_dates, values=dataframe_log_returns_type.values)
                # assign to a dictionary
                self.market.historical_logrets[mkt_name] = obj_log_returns

    def _get_historical_spot_prices(self, mkt_code: str, in_file: str, in_sheet: str):
        """

        :param mkt_code:
        :param in_file:
        :param in_sheet:
        :return:
        """

        # Read the hourly forward curve and assign it. Filter the forward curve corresponding to the mkt_code
        dataframe_fwd_curve = pd.read_excel(in_file, index_col="date", sheet_name=in_sheet,
                                            parse_dates=["date"]).filter(regex=mkt_code)

        # Loop over the curves
        trading_dates = dataframe_fwd_curve.index
        for product in dataframe_fwd_curve.columns:
            obj_historical_spot_curves = HistoricalSpotCurve(mkt_name=product, granularity="H",
                                                             trading_dates=trading_dates, values=dataframe_fwd_curve[product].values)

            self.market.historical_spot_prices[product] = obj_historical_spot_curves

    def _compute_log_returns_from_prices(self, mkt_code: str, in_file: str, in_sheet: str):
        """
        Compute the log-returns from prices
        :param mkt_code:
        :param in_file:
        :param in_sheet:
        :return:
        """

```

```
        # Read the log-returns. Filter the forward curve corresponding to the mkt_code
        prices_df = pd.read_excel(in_file, index_col="trading_date", sheet_name=in_sheet,
                                  parse_dates=["trading_date"]).filter(regex=mkt_code)

        # Always assume that you have PK, OP, BL products and filter the dataframe. If it has not, skip the filling
        for product_type in self.market.sub_markets:
            # Filter OP, PK, BL
            prices_df_type = prices_df.filter(regex=product_type)
            # If it is not empty fill it
            if not prices_df_type.empty:

                # remove the overlapping products and compute log-returns
                mkt_code_and_type = mkt_code + "_" + product_type
                prices_df_type, T = ut.prepare_price_dataframe(prices_df=prices_df_type,
                                                               mkt_code_and_type=mkt_code_and_type)
                dataframe_log_returns = ut.prepare_log_returns(df_quotes=prices_df_type)

                trading_dates = dataframe_log_returns.index

                # Create a log-returns object
                mkt_name = self.market.mkt_code + "_" + product_type
                obj_log_returns = LogReturns(mkt_name=mkt_name, granularity="",
                                             trading_dates=trading_dates,
                                             values=dataframe_log_returns.values, prices=prices_df_type, T=T)

                # Remove the values larger that a value date
                obj_log_returns.filter_outliers_log_returns()

                # assign to a dictionary
                self.market.historical_logrets[mkt_name] = obj_log_returns

    def build_market(self, cfg_whole_file, mkt_code: str):
        """

        :param cfg_whole_file: configuration whole file
        :param mkt_code: like DE, TTF, PSV, F7 and so on.
        :return:
        """

```

```python
        TEMP = False

        # Get the dictionary entry corresponding to the given market code
        cfg = cfg_whole_file[mkt_code]

        # get the trading date: if empty look for the most recent file
        trading_date = cfg["trading_date"]

        # set the spot granularity of the spot market
        self.market.spot_granularity = cfg["spot_granularity"]


        # Set the monthly forward curve
        in_file_path = cfg["monthly_fwd_curve"]["in_file_path"]
        in_file, trading_date_str_fwd_curve = get_last_fwd_curve_file(in_file_path, trading_date)
        in_sheet = cfg["monthly_fwd_curve"]["in_sheet"]
        self._get_monthly_fwd_curve(mkt_code=mkt_code, in_file=in_file,
                                    in_sheet=in_sheet, trading_date=trading_date_str_fwd_curve)

        # Set the hourly forward curve
        in_file_path = cfg["hourly_fwd_curve"]["in_file_path"]
        in_file, trading_date_str_spot_curve = get_last_fwd_curve_file(in_file_path, trading_date)
        in_sheet = cfg["hourly_fwd_curve"]["in_sheet"]
        self._get_hourly_fwd_curve(mkt_code=mkt_code, in_file=in_file,
                                   in_sheet=in_sheet, trading_date=trading_date_str_spot_curve)

        # Set the hourly historical spot prices
        in_file = cfg["hourly_historical_spot_curve"]["in_file_path"]
        in_sheet = cfg["hourly_historical_spot_curve"]["in_sheet"]
        self._get_historical_spot_prices(mkt_code=mkt_code, in_file=in_file,
                                         in_sheet=in_sheet)

        assert trading_date_str_spot_curve == trading_date_str_fwd_curve, "Spot and forward trading date must be the same"

        self.market.trading_date = trading_date_str_spot_curve

        # Set the log-returns: from log-returns directly or from prices
        if TEMP:
            in_file = cfg["prices"]["in_file"]
```

```python
            in_sheet = cfg["log_returns"]["in_sheet"]
            self._get_log_returns(mkt_code=mkt_code, in_file=in_file, in_sheet=in_sheet)
        else:
            # Set the log-returns
            in_file = cfg["prices"]["in_file"]
            in_sheet = cfg["prices"]["in_sheet"]
            self._compute_log_returns_from_prices(mkt_code=mkt_code, in_file=in_file, in_sheet=in_sheet)


        # Set if it has a peak or not
        self.market.has_peak = (cfg["has_peak"] == "True")
        if not self.market.has_peak:
            self.market.sub_markets = ["BL"]


class MarketBuilderDirector:

    def __init__(self):
        """
        Constructor
        """
        self.pMarketBuilder = None

    def set_market_builder(self, mb):
        """
        Set the MarketBuilder an object with property mkt_code and build_market
        :param mb:
        :return:
        """
        self.pMarketBuilder = mb

    def build_market(self, cfg_file, mkt_code: str):
        """
        Build the market using the MarketBuilder you have chosen.
        :param cfg_file: configuration file for the Market Builder. This depends on the Market Builder you use
        :param mkt_code: market code to use
        :return:
        """

        self.pMarketBuilder.market.mkt_code = mkt_code
        self.pMarketBuilder.build_market(cfg_file, mkt_code)

        # return the Market object containing all the snapshot of the market
        x = copy.deepcopy(self.pMarketBuilder.market)
        return x


```

```python
if __name__ == "__main__":
    # read the configuration json file

    config_file = r"C:\Users\eid0110204\Documents\coding_projects_test\kiesel_project_git_repo\kiesel_model\config\cfg_excel_builder.json"
    file_object = open(config_file)
    config = json.load(file_object)
    file_object.close()

    mkt_codes = ["DE", "TTF"]
    markets_snapshots = {}

    for mkt_code in mkt_codes:
        # Create the market builder Director
        mbd = MarketBuilderDirector()
        # Create the Market builder Excel
        mb = MarketBuilderExcel()
        # Se the market builder
        mbd.set_market_builder(mb)
        # Create the object Market using the MarketBuilderDirector
        markets_snapshots[mkt_code] = mbd.build_market(config, mkt_code=mkt_code)
```

6.3 kluge_model.py module

This module contains the implementation of the model.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import os
from datetime import datetime
from utilities import setup_custom_logger, create_lambda_matrix
from market_utilities import  HistoricalSpotCurve, ForwardCurve
import time
import json
from sklearn.linear_model import LinearRegression
import statsmodels.api as sm


class Parameter:
    def __init__(self, value=None, locked=False):
        self.value = value
        self.locked = locked

    def __repr__(self):
        return f'value: {self.value}, locked: {self.locked}'


class KlugeModel:

    def __init__(self, builder):
        """
        KlugeModel constructor
        :param builder: KlugeModelBuilder

        """
        self.params = builder.params
        self.historical_spot_prices = builder.historical_spot_prices
        self.forward_curve = builder.forward_curve
        self.trading_date_str = builder.trading_date_str
        self.n_sim = builder.n_sim
        self.market_name = builder.market_name
        self.sim = None
        self.n_days_tot = builder.n_days_tot
        self.time_step = 1/365
        self.granularity = builder.granularity

    def __repr__(self):
        # Get string representation of the parameters
        string_parameters = [f"{k}: {v.value}" for k, v in self.params.items()]
        return f'KlugeModel: market_name{self.market_name} parameters: {string_parameters}, n_sim: {self.n_sim},
```

```
trading_date: {self.trading_date_str}'

    def _replace_negative_values(self, y: np.array, method_replace: str) -> np.array:
        """
        Replace negative values with the minimum positive value
        :param y: np.array. Time series
        :param method_replace: str. Method to replace negative values
        :return: np.array. Time series with negative values replaced
        """
        idx_negative = y < 0

        if method_replace == "min_positive":
            # the minimum available price is the smallest positive value
            minimum_available_price = np.min(y[~idx_negative])
            y[idx_negative] = minimum_available_price
        else:
            raise ValueError("Unknown method to replace negative values")
        return y

    def _remove_seasonality_with_linear_regression(self, y: np.array, lambda_matrix: np.array) -> np.array:
        """
        Remove the seasonality from a time series with linear regression
        :param y: np.array. Time series
        :param lambda_matrix: np.array. Lambda matrix
        :return: np.array. Time series without the seasonal component
        """
        fitted_model = LinearRegression().fit(lambda_matrix, y)
        y_fitting = fitted_model.predict(lambda_matrix)
        return (y - y_fitting).flatten()

    def _deseasonalize_historical_spot(self):
        """
        Remove the seasonality from the historical spot prices
        :return: the time series without the seasonal component of log-prices
        """

        # Aggregate curve on daily basis
        self.historical_spot_prices.aggregate_curve(frequency="D")
        lambda_matrix_seasonality = create_lambda_matrix(daily_calendar=self.historical_spot_prices.trading_dates)

        # Replace negative values with the minimum positive value
```

```python
        y = self._replace_negative_values(y=self.historical_spot_prices.values, method_replace="min_positive")

        # Compute log-prices
        logy = np.log(y)
        # return the signal without seasonal part
        return self._remove_seasonality_with_linear_regression(y=logy, lambda_matrix=lambda_matrix_seasonality)

    def _prepare_OLS_model_data(self, data: np.array):
        """
        Prepare the data for the OLS model
        :param data: np.array. Time series
        :return: np.array. Time series without the seasonal component of log-prices
        """

        # Create the x and y arrays for the linear regression
        regressors = data[:-1]
        response = data[1:]
        regressors = regressors.reshape(1, -1).transpose()

        # Add a constant term for the intercept
        regressors = sm.add_constant(regressors)

        return regressors, response

    def _perform_OLS(self, data: np.array):
        """
        Perform the Ordinary Least Squares (OLS) regression
        :param data: np.array. Time series
        :return: np.array. Time series without the seasonal component of log-prices
        """

        x, y = self._prepare_OLS_model_data(data)

        model = sm.OLS(y, x).fit()

        return model

    def _estimate_ou_parameters(self, model, dt=1/365):
        """
        Estimate the parameters of the Ornstein-Uhlenbeck process
        :param model: sm.OLS. OLS model
        :param dt: float. Time step
        :return: float, float, float. Estimated parameters of the OU process
        """

        a = model.params[0]  # intercept
        b = model.params[1]  # coefficient
```

```

        residuals = model.resid
        mean_of_residuals = np.mean(residuals**2)

        # Map back regression parameters to OU parameters
        theta = (1-b)/dt
        mu = a/(theta*dt)
        sigma = np.sqrt(mean_of_residuals/dt)

        # dictionary of parameters
        return {"mu": mu, "k": theta, "sigma_s": sigma}

    def _assign_parameters(self, params: dict):
        """
        Assign the parameters to the model
        :param params: dict. Parameters
        :return: None
        """

        for key, value in params.items():
            if not self.params[key].locked:
                self.params[key].value = value

    def _calibrate_ou_linear_regression(self, data: np.array, dt=1/365):
        """
        Calibrate the Ornstein-Uhlembeck process via linear-regression
        :return:
        """

        # Perform the OLS regression
        model = self._perform_OLS(data)

        # Estimate the parameters of the OU process
        ou_params = self._estimate_ou_parameters(model, dt)

        # update parameters
        self._assign_parameters(ou_params)

    def _select_jumps_id(self, increments: np.array, threshold: float, method: str="threshold_filter") -> np.array:
        """
        Select the jumps id from the time series
        :param increments: np.array. Time series of increments to filter
        :param threshold: float. Threshold
        :param method: str. Method to detect jumps
        :return: np.array. Jumps id
        """

        if method == "threshold_filter":
```

```
            # Compute the differences in time series
            jumps_id = np.where(np.abs(increments) > threshold)[0]
        else:
            raise ValueError("Unknown method to select jumps")

        return jumps_id

    def _filter_jumps(self, data: np.array):
        """
        Filter the jumps from the time series
        :param data: np.array. Time series to filter
        :return: np.array. Time series without jumps
        """

        #FIXME: return the cleared time series and the jumps sizes

        # Compute the increments
        increments = np.diff(data)

        # Jumps sizes
        jumps_sizes = []

        dev_std_increments = np.std(increments)

        keep_looking_for_jumps = True

        while keep_looking_for_jumps:
            # Look jumps above 2 standard deviations
            jumps_id = self._select_jumps_id(increments=increments,
    threshold=2.8*dev_std_increments, method="threshold_filter")

            if len(jumps_id) == 0:
                keep_looking_for_jumps = False
            else:
                # Store jumps in another array
                jumps_sizes = jumps_sizes + list(increments[
    jumps_id])
                # Remove jumps from the time series
                increments = np.delete(increments, jumps_id)
                # Update the standard deviation
                dev_std_increments = np.std(increments)

        return increments, np.array(jumps_sizes)

    def _calibrate_jump_process_double_exponential(self,
    jumps_sizes: np.array):
        """
        Calibrate the jump process via exponential distribution
        :param jumps_sizes: np.array. Jumps sizes
        :return: None
        """

```

```
        # Compute the number of positive jumps and negative jumps
        n_positive_jumps = len(jumps_sizes[jumps_sizes > 0])

        # prepare the dictionary of parameters
        params = {"lam_P": len(jumps_sizes)/(self.n_days_tot/365),
    "p_up": n_positive_jumps/len(jumps_sizes),
                  "lam_up": np.mean(jumps_sizes[jumps_sizes > 0]),
    "lam_down": -np.mean(jumps_sizes[jumps_sizes < 0])}

        # assign the parameters
        self._assign_parameters(params)

    def calibrate(self):
        """
        Calibrate the Kluge model
        :return: None
        """

        # remove the seasonality the time series of log-prices
        ou_process_to_fit = self._deseasonalize_historical_spot()

        # filter jumps out of the time series
        _, jumps_sizes = self._filter_jumps(ou_process_to_fit)

        # perform the calibration of the Ornstein-Uhlenbeck process
        self._calibrate_ou_linear_regression(ou_process_to_fit)

        # perform the calibration of the jump process
        self._calibrate_jump_process_double_exponential(jumps_sizes
    )

    def _simulate_jumps_number(self, n_jumps: int, time_horizon:
    float) -> np.array:
        """
        Simulate the jumps number according to a Poisson
    distribution of parameter lambda*T
        :param n_jumps: int. Number of jumps
        :time_horizon: float. Time horizon in years

        :return: np.array. Number of jumps, np.array. Cumulative
    sum of jumps
        """
        # generate the number of jumps according to a Poisson
    process
        n_jumps = np.random.poisson(self.params["lam_P"].value *
    time_horizon, self.n_sim)
        cum_sum_jumps = np.cumsum(n_jumps)
        cum_sum_jumps = np.insert(cum_sum_jumps, 0, 0)
        return n_jumps, cum_sum_jumps

    def _simulate_jumps_times(self, n_jumps: np.array, time_horizon
    : float) -> np.array:
```

```python
        """
        simulate the jump times between [0,time_horizon] according to a uniform distribution
        :param n_jumps: np.array. Number of jumps
        :param time_horizon: float. Time horizon in years
        :return: np.array. Jump times
        """


        # generate the jump times (uniformly distributed in the period)
        jump_times = np.random.uniform(0, time_horizon, np.sum(n_jumps))
        return jump_times

    def _simulate_double_exponentially_jumps_sizes(self, n_jumps: np.array) -> np.array:
        """
        Simulate the jump sizes: some of them are up, others are down. According to a double exponential distribution
        :param n_jumps: np.array. Number of jumps
        :return: np.array. Jump sizes
        """

        # generate the jump sizes up and down
        jump_sizes_up = np.random.exponential(scale=self.params["lam_up"].value, size=np.sum(n_jumps))
        jump_sizes_down = np.random.exponential(scale=self.params["lam_down"].value, size=np.sum(n_jumps))
        jump_sizes = np.zeros(np.sum(n_jumps))

        # generate the probability of jump up according to a Uniform distribution
        bool_jumps_up = np.random.uniform(0, 1, np.sum(n_jumps)) < self.params["p_up"].value

        # Set the jump sizes
        jump_sizes[bool_jumps_up] = jump_sizes_up[bool_jumps_up]
        jump_sizes[~bool_jumps_up] = -jump_sizes_down[~bool_jumps_up]

        return jump_sizes

    def _compute_diffusion_ou_increment(self):
        """
        Compute the diffusion increment for the OU process, namely the sigma.
        """
        sigma_ou = np.sqrt(self.params["sigma_s"].value**2.0/(2*self.params["k"].value)*(1-np.exp(-2*self.params["k"].value*self.time_step)))
        return sigma_ou
```

```python

    def _compute_ou_increments(self):
        """
        Compute the increments for the OU process
        """
        # Compute the diffusion increment (sigma_ou)
        sigma_ou = self._compute_diffusion_ou_increment()

        # Increments are normally distributed
        ou_increments = np.random.normal(loc=0, scale=sigma_ou, size=(self.n_sim, self.n_days_tot))
        return ou_increments

    def _compute_diffusion_bm_increment(self):
        """ Generate the BM increment of the long-term diffusion part"""

        # Generate BM increments
        sigma_bm = self.params["sigma_l"].value*np.sqrt(self.time_step)
        bm_increments = np.random.normal(loc=0, scale=sigma_bm, size=(self.n_sim, self.n_days_tot))
        return bm_increments

    def _compute_drift_corrector(self, time_grid: np.array):
        # Drift correction
        sigma_ou_drift = np.sqrt(self.params["sigma_s"].value**2.0/(2*self.params["k"].value)*(1-np.exp(-2*self.params["k"].value*time_grid)))

        drift_corrector = - (0.5*sigma_ou_drift**2 + 0.5*self.params["sigma_l"].value**2*time_grid +
                             self.params["p_up"].value*self.params["lam_P"].value/self.params["k"].value*np.log((1-self.params["lam_up"].value*np.exp(-self.params["k"].value*time_grid))/(1-self.params["lam_up"].value)) +
                             (1-self.params["p_up"].value)*self.params["lam_P"].value/self.params["k"].value*np.log((1+self.params["lam_down"].value*np.exp(-self.params["k"].value*time_grid))/(1+self.params["lam_down"].value)))

        return drift_corrector

    def _simulate_martingale(self):
        """
        Simulate the martingale part of the Kluge model
        :return: np.array. Simulated martingale
        """

        # Time horizon in years
        T = self.n_days_tot/365

```

```python
        # Time grid
        time_grid = np.linspace(0, T, self.n_days_tot)

        # Generate the number of jumps between 0 and T
        n_jumps, cum_sum_jumps = self._simulate_jumps_number(n_jumps=self.n_sim, time_horizon=T)

        # Generate the jump times
        jump_times = self._simulate_jumps_times(n_jumps=n_jumps, time_horizon=T)

        # Generate the jump sizes
        jump_sizes = self._simulate_double_exponentially_jumps_sizes(n_jumps=n_jumps)

        # Generate the increments for the OU process
        ou_increments = self._compute_ou_increments()

        # Generate the increments for the BM process for the long-term diffusion part
        bm_increments = self._compute_diffusion_bm_increment()

        # Simulate the process:
        Z = np.zeros((self.n_sim, self.n_days_tot))


        #FIXME: make a method!!!!!!!!!!!!!!!

        for i in range(self.n_sim):
            # discrete times
            discrete_jump_times = np.sort(self._closest_points(time_grid, jump_times[cum_sum_jumps[i]:cum_sum_jumps[i+1]]))
            jump_sizes_sim_i = jump_sizes[cum_sum_jumps[i]:cum_sum_jumps[i+1]]

            x_jump_sum = np.zeros(self.n_days_tot)
            x_continuous = np.zeros(self.n_days_tot)
            x_bm = np.zeros(self.n_days_tot)

            for j in range(1, self.n_days_tot):
                sum_values = 0

                x_continuous[j] = x_continuous[j-1]*np.exp(-self.params["k"].value*self.time_step) + ou_increments[i,j]
                x_bm[j] = x_bm[j-1] + bm_increments[i,j]
                if n_jumps[i] > 0:
                    for k in range(len(discrete_jump_times)):
                        if discrete_jump_times[k] <= time_grid[j]:
                            sum_values += np.exp(-self.params["k"].value * (time_grid[j] - discrete_jump_times[k])) * jump_sizes_sim_i[k]
                            x_jump_sum[j] = sum_values
```

```
            Z[i,:] =  x_jump_sum + x_continuous + x_bm

        # Compute the drift corrector
        drift_corrector = self._compute_drift_corrector(time_grid=
    time_grid)

        # Create the martingale
        Z = Z + drift_corrector

        return Z

    def _generate_col_names(self):
        """
        Generate the column names for the simulated spot prices
        :return: list. Column names
        """
        n_sim = self.n_sim
        col_names = ["Var" + str(i) for i in range(n_sim)]
        return col_names

    def _apply_hourly_shape_to_sim(self, exp_Z: np.array,
    daily_calendar: np.array):
        """"
        Apply the hourly shape to the simulated spot prices
        :param exp_Z: np.array. Simulated Doleans-Dade process
        :param daily_calendar: np.array. Daily calendar
        :return: None
        """

        # Create hourly shocks (this is a dataframe)
        z_unitary_hourly_shocks_sim, col_names = \
        KlugeModel._replicate_daily_shock_on_hourly_base(
    Z_unit_shocks=exp_Z,

       daily_shock_caledar=daily_calendar,

       hourly_shock_calendar=self.forward_curve.trading_dates,

       n_sim=self.n_sim)

        # get the values of hourly unitary shocks (now shocks are
    hourly)
        Z_unit_shocks = z_unitary_hourly_shocks_sim.values.
    transpose()
        return Z_unit_shocks, col_names

    def _prepare_dataframe_sim(self, simulated_values: np.array):

        # Create the column names
        col_names = self._generate_col_names()

```

```
        # Create and store the dataframe
        self.sim = pd.DataFrame(simulated_values, index=self.forward_curve.trading_dates, columns=col_names)
        self.sim.index.name = "Time"

    def _shock_the_forward_curve(self, Z_unit_shocks: np.array):
        """
        apply the shocks to the forward curve
        :param Z_unit_shocks: np.array. Shocks. They can be daily or hourly
        :return: np.array. Simulated values
        """
        simulated_values = (self.forward_curve.values * Z_unit_shocks).T
        return simulated_values


    @staticmethod
    def _closest_points(grid: np.array, points: np.array):
        """
        Find the closest points in the grid to the given points
        :param grid: np.array. Grid
        :param points: np.array. Points
        :return: np.array. Closest points"""
        closest_points = []
        for p in points:
            # Find the grid point that minimizes the absolute distance
            closest_grid_point = min(grid, key=lambda g: abs(g - p))
            closest_points.append(closest_grid_point)
        return closest_points

    @staticmethod
    def _replicate_daily_shock_on_hourly_base(Z_unit_shocks: np.array, daily_shock_caledar, hourly_shock_calendar, n_sim: int):
        """
        Takes daily shocks and generate hourly ones. Each hourly shock is the same as the daily one
        :param Z_unit_shocks: unitary shoks with daily base
        :param daily_shock_caledar: daily calendar
        :param hourly_shock_calendar: hourly calendar: we need this in order to put the same shcck on each hourd of the day
        :param n_sim: number of simulations
        :return: a dataframe with hourly shocks and each colums is a simulation
        """
        # Transform into DataFrame
        col_names = ["Var" + str(i) for i in range(n_sim)]
        z_unitary_daily_shocks_sim = pd.DataFrame(Z_unit_shocks.transpose(), columns=col_names)
        z_unitary_daily_shocks_sim.index = daily_shock_caledar
```

```python

        # Remove the hours from the format YYYY-MM-DD-HH
        date_without_hours = hourly_shock_calendar.strftime('%Y-%m-%d')
        date_string = z_unitary_daily_shocks_sim.index.strftime('%Y-%m-%d')
        z_unitary_daily_shocks_sim.index = date_string

        # An empty dataframe containing hourly shocks: perform a left join
        z_unitary_hourly_shocks_sim = pd.DataFrame(index=date_without_hours)
        z_unitary_hourly_shocks_sim = z_unitary_hourly_shocks_sim.merge(z_unitary_daily_shocks_sim, left_index=True,
                                                                       right_index=True, how='left')
        return z_unitary_hourly_shocks_sim, col_names

    @staticmethod
    def _create_daily_calendar(start_date: datetime, end_date: datetime):
        daily_calendar = pd.date_range(start=start_date, end=end_date, freq="D")
        return daily_calendar

    @staticmethod
    def _create_doleans_dade_process(Z: np.array):
        """
        Create the Doleans-Dade process
        :param Z: np.array. Simulated martingale
        :return: np.array. Doleans-Dade process
        """

        # Compute the exponential of the martingale
        exp_Z = np.exp(Z)

        return exp_Z


    def simulate(self):
        """
        Simulate the Kluge model
        :return: None
        """

        # Get the forward curve
        forward_curve = self.forward_curve

        # Create a daily calendar between two dates
        daily_calendar = self._create_daily_calendar(start_date=forward_curve.trading_dates[0], end_date=forward_curve.trading_dates[-1])
```

```python

        # Simulate the martingale part
        Z = self._simulate_martingale()

        # Create the Doleans-Dade process
        exp_Z = self._create_doleans_dade_process(Z)

        if self.forward_curve.granularity == "H":
            # Apply the hourly shape to the simulated spot prices
            Z_unit_shocks, _ = self._apply_hourly_shape_to_sim(exp_Z=exp_Z, daily_calendar=daily_calendar)

        #Shock the forward curve
        simulated_values = self._shock_the_forward_curve(Z_unit_shocks=Z_unit_shocks)

        # Create and store the dataframe
        self._prepare_dataframe_sim(simulated_values=simulated_values)


    def to_csv(self, file_name: str):
        """
        Save the simulated spot prices to a csv file
        :param file_name: str. File name
        :return: None
        """

        # If the granularity is not hourly, resample the data
        if self.granularity != "H":
            self.sim.resample('self.granularity').mean()

        # Save to csv
        self.sim.to_csv(file_name, index=True)

    def plot(self, directory_path: str):

        """
        Plot simulations and save into a given path
        : param directory_path: path where plots are saved
        """

        # Get the trading date
        trading_date = self.trading_date_str
        separator = "_"

        # Create name spot files
        name_spot_file = separator.join([self.market_name, "sim", "spot", trading_date]) + ".png"

        path_final_to_save_spot = os.path.join(directory_path, name_spot_file)
```

```
        # If spot simulations are available plot them:
        if not (self.sim is None):
            df_sim = self.sim

            # Aggregate on monthly basis
            df_sim = df_sim.resample('M').mean()

            title_name_plot = self.market_name + " Trading Date: "
 + trading_date
            # Compute the mean
            mean_value = df_sim.mean(axis=1)
            # Compute the percentile data
            percentile_data = df_sim.quantile([0.05, 0.95], axis=1)
            # Get the screen size
            fig = plt.figure(figsize=(15, 10), dpi=100)
            manager = plt.get_current_fig_manager()
            manager.full_screen_toggle()
            plt.plot(mean_value.index, df_sim.values[:, 1:10],
 color=[0.8, 0.8, 0.8])
            plt.plot(mean_value.index, mean_value.values, label="
 mean")
            plt.plot(mean_value.index, percentile_data.values[0,
 :], color=[0, 0, 0.8], label="5th pct")
            plt.plot(mean_value.index, percentile_data.values[1,
 :], color=[0, 0, 0.8], label="95th pct")
            plt.title(title_name_plot)
            plt.legend()
            fig.savefig(path_final_to_save_spot, dpi=300)
            plt.close(fig)

    def check_spot_sim_convergence(self, directory_path: str):
        """
        Check the convergence of the spot simulation to the hourly
 forward curve
        :param directory_path: str. Directory path
        """

        if not (self.sim is None):
            # get simulations
            simulations = self.sim
            mean_sim = np.mean(simulations, axis=1)
            spot_hourly_fwd_curve_name = self.forward_curve

            # Whole plot
            # Get the trading date
            trading_date = self.trading_date_str
            separator = "_"
            name_file = separator.join([self.market_name, "
 check_convergence_spot", trading_date]) + ".png"

            # Final plot name
```

```
            path_final_to_save_spot = os.path.join(directory_path, name_file)

            fig = plt.figure(figsize=(15, 10), dpi=100)
            manager = plt.get_current_fig_manager()
            manager.full_screen_toggle()
            plt.plot(mean_sim.index, mean_sim.values, label="mean", linewidth=2)
            plt.plot(spot_hourly_fwd_curve_name.trading_dates, spot_hourly_fwd_curve_name.values, color=[0.5, 0 ,0],
                     label="fwd_curve", linestyle="--")
            plt.title("Check convergenge for " + self.market_name + " Trading Date: " + trading_date)
            plt.legend()
            fig.savefig(path_final_to_save_spot, dpi=300)

            # zoomed plot
            name_file = separator.join([self.market_name, "check_convergence_spot_zoomed", trading_date]) + ".png"
            path_final_to_save_spot = os.path.join(directory_path, name_file)
            start_date = mean_sim.index[0]
            end_date = start_date + pd.Timedelta(days=120)
            plt.xlim(start_date, end_date)
            fig.savefig(path_final_to_save_spot, dpi=300)

            plt.close(fig)








class KlugeModelBuilder:

    def __init__(self):
        """
        KlugeModelBuilder constructor
        the parametera are initialized with default values
        lam_P: expected number of jumps per year
        lam_up: expected size of the jump up
        lam_down: expected size of the jump down
        p_up: probability of jump up
        sigma_s: volatility of the OU process
        sigma_l: volatility of the BM process
        mu: mean of the OU process
        k: mean reversion speed of the OU process"""

        self.params = {"lam_P": Parameter(value=10.0), "lam_up": Parameter(value=0.3), "lam_down": Parameter(value=0.3),
```

```
                            "p_up": Parameter(value=0.5), "sigma_s": Parameter(value=2.0), "sigma_l": Parameter(value=0.0),
                            "mu": Parameter(value=0.0), "k": Parameter(value=50.0) }
        self.historical_spot_prices = None # HistoricalSpotCurve
        self.forward_curve = None # ForwardCurve
        self.n_sim = None # int number of simulations
        self.n_days_tot = None # int number of days
        self.market_name = None # str market name
        self.trading_date_str = None # str trading date
        self.granularity = None # str granularity: "H" for hourly, "D" for daily and so on

    def set_params(self, params: dict):
        """
        Set the parameters of the Kluge model
        :param params: list. List of parameters
        """

        if not params:
            pass
        else:
            for p in params:
                self.params[p].value = params[p]
                self.params[p].locked = True



        return self

    def set_trading_date(self, trading_date_str: str):
        """
        Set the trading date
        :param trading_date_str: str. Trading date
        """

        self.trading_date_str = trading_date_str
        return self

    def set_n_sim(self, n_sim):
        """
        set the number of simulations
        :param n_sim: int. Number of simulations
        """
        self.n_sim = n_sim
        return self

    def set_granularity(self, granularity="H"):
        """
        Set the granularity of the commodity you want to simulate
        :param granularity: str. Granularity of the commodity
        """
```

```python
        self.granularity = granularity
        return self

    def set_fwd_curve_from_file(self, fwd_curve_file: str, country: str, granularity: str = "H"):
        """
        Set the forward curve from a xlsx file
        :param fwd_curve_file: str. File name containing the forward curve
        :param country: str. Country code
        """
        fwd_curve_df = pd.read_excel(fwd_curve_file, index_col=0, parse_dates=["date"])

        fwd_curve = ForwardCurve(mkt_name=country, granularity="H", trading_dates=fwd_curve_df.index, values=fwd_curve_df[country].values)

        self.forward_curve = fwd_curve

        # Compute the number of days
        self.n_days_tot = len(np.unique(self.forward_curve.trading_dates.date))

    def set_fwd_curve(self, df_fwd_curve: pd.DataFrame, country: str, granularity: str = "H"):
        """
        Set the forward curve contained in a DataFrame
        :param df_fwd_curve: pd.DataFrame. DataFrame containing the forward curve
        :param country: str. Country code
        """

        fwd_curve = ForwardCurve(mkt_name=country, granularity="H", trading_dates=df_fwd_curve.index, values=df_fwd_curve[country].values)

        self.forward_curve = fwd_curve

        # Compute the number of days
        self.n_days_tot = len(np.unique(self.forward_curve.trading_dates.date))

    def set_spot_prices_from_file(self, spot_prices_file: str, country: str):
        """"
        Set the historical spot prices from a xlsx file
        :param spot_prices_file: str. File name containing the historical spot prices
        :param country: str. Country code
        """
```

```python
        spot_prices_df = pd.read_excel(spot_prices_file, index_col=0, parse_dates=["date"])

        self.historical_spot_prices = HistoricalSpotCurve(mkt_name=country, granularity="H",
                                                          trading_dates=spot_prices_df.index, values=spot_prices_df[country].values)

    def set_spot_prices(self, df_spot_prices: pd.DataFrame, country: str):
        """
        Set the historical spot prices contained in a DataFrame
        :param df_spot_prices: pd.DataFrame. DataFrame containing the historical spot prices
        :param country: str. Country code
        """

        self.historical_spot_prices = HistoricalSpotCurve(mkt_name=country, granularity="H",
                                                          trading_dates=df_spot_prices.index, values=df_spot_prices[country].values)

    def set_market_name(self, market_name: str):
        """"
        Set the market name
        :param market_name: str. Market name
        """
        self.market_name = market_name
        return self





if __name__ == "__main__":

    "This is a main test for the code"

    fwd_curve_file = r"C:\Users\EID0110204\Documents\MATLAB\20240925_ou_with_jumps\all_curves_2024-09-25.xlsx"
    spot_prices_file = r"C:\Users\EID0110204\Documents\MATLAB\20240925_ou_with_jumps\historical_spot_prices.xlsx"

    builder = KlugeModelBuilder()
    builder.set_n_sim(500)
    builder.set_market_name("IT")
    builder.set_granularity("H")
    print("Set forward curve")
    builder.set_fwd_curve_from_file(fwd_curve_file, "IT")
    print("Set spot prices")
    builder.set_spot_prices_from_file(spot_prices_file, "IT")
    kluge_model = KlugeModel(builder)
```

```
    kluge_model.calibrate()

    print(kluge_model)

    kluge_model.simulate()



    kluge_model.to_csv("simulated_spot_kluge_prices.csv")

    print("End of the program")
```

6.4 main.py module

This is the main file and show how to run the code. The input are not included in this latex file.

```
import json
import os
import pandas as pd
from utilities import setup_custom_logger, get_last_fwd_curve_in_folder
from kluge_model import KlugeModelBuilder, KlugeModel



if __name__ == "__main__":

    # Configuration file
    config_file = r"config\cfg_main.json"


    logger = setup_custom_logger('log/kluge_model')

    logger.info("||| Starting Kluge model")
    logger.info("||| Reading configuration file")

    # Load configuration file
    file_object = open(config_file)
    config_results = json.load(file_object)
    file_object.close()

    # Forward and spot curve path
    forward_curve_path = config_results['forward_curve_path']
    spot_curve_path = config_results['spot_curve_path']

    # forward and spot curve name
    fwd_curve_prefix = config_results['fwd_curve_prefix']
    spot_prices_file_name = config_results['spot_prices_file_name']

    # trading date
    trading_date = config_results['trading_date']

    # Get the last forward curve in the folder
    selected_fwd_curve, trading_date_fwd = get_last_fwd_curve_in_folder(folder_path=forward_curve_path, prefix=fwd_curve_prefix, trading_date=trading_date)



    # Paths and file
    fwd_curve_path_and_file = os.path.join(forward_curve_path, selected_fwd_curve)
    spot_curve_path_and_file = os.path.join(spot_curve_path, spot_prices_file_name)
```

```python

    logger.info(f"||| Selected forward curve for trading date: {trading_date_fwd}")

    # country_list (a dictionary)
    country_list = config_results['country_list']


    # Output path
    out_sim_path = config_results['out_sim_path']


    # Read the forward curve and the spot prices
    logger.info("||| Reading forward curve from file")
    df_fwd_curve = pd.read_excel(fwd_curve_path_and_file, index_col=0, parse_dates=["date"])

    logger.info("||| Reading spot prices from file")
    df_spot_prices = pd.read_excel(spot_curve_path_and_file, index_col=0, parse_dates=["date"])


    # Loop over the elements of the dictionary
    for key, value in country_list.items():
        logger.info(f"||| Country: {key} in progress")

        # country code:
        country_code = value['country_code']

        # Get the parameters
        parameters = value['parameters']

        # Read number of simulations
        n_sim = value['n_sim']

        # Read the granularity
        granularity = value['granularity']

        # output path sim and file name
        file_out_name = "spot_simulations_" + country_code + ".csv"
        out_sim_path_and_file = os.path.join(out_sim_path, file_out_name)

        # create the object and set the parameters
        builder = KlugeModelBuilder()
        logger.info(f"||| Bulder for {key}")
        builder.set_n_sim(n_sim= n_sim).set_market_name(market_name=country_code).set_granularity(granularity=granularity).set_trading_date(trading_date_str=trading_date_fwd)


        # Set the parameters
```

```python
        logger.info(f"||| Setting parameters for {key}")
        builder.set_params(parameters)

        logger.info(f"||| Setting forward curve for {key}")
        builder.set_fwd_curve(df_fwd_curve=df_fwd_curve, country=country_code)

        logger.info(f"||| Setting spot prices for {key}")
        builder.set_spot_prices(df_spot_prices=df_spot_prices, country=country_code)



        # Create the model
        logger.info(f"||| Creating Kluge model for {key}")
        kluge_model = KlugeModel(builder)

        # Calibrate the model, simulate and save the results

        logger.info(f"||| Calibrating {key}")
        kluge_model.calibrate()

        logger.info(f"|||  {str(kluge_model)}")

        logger.info(f"||| Simulating {key}")
        kluge_model.simulate()

        logger.info(f"||| Plotting {key}")
        kluge_model.plot(directory_path=out_sim_path)
        kluge_model.check_spot_sim_convergence(directory_path=out_sim_path)

        logger.info(f"||| Saving {key}")
        kluge_model.to_csv(out_sim_path_and_file)

        logger.info(f"||| Country: {key} done")
```

References

[1] O.E. Barndorff-Nielsen. Processes of Normal Inverse Gaussian Type. *Finance and Stochastics*, 2(1):41–68, 1998.

[2] F.E. Benth, J.S. Benth, and S. Koekebakker. *Stochastic Modeling of Electricity and Related Markets*. Number 6811 in World Scientific Books. World Scientific Publishing Co. Pte. Ltd., July 2008. ISBN ARRAY(0x50724998). URL `https://ideas.repec.org/b/wsi/wsbook/6811.html`.

[3] D. Brigo, A. Dalessandro, Neugebauer M., and F. Triki. A Stochastic Process Toolkit for Risk Management. Technical report, 2007.

[4] R. H. Böerger, R. Kiesel, and G. Schindlmayr. A Two-Factor Model for the Electricity Forward Market. *Quantitative Finance*, 9(3):279–287, 2009. doi: https://doi.org/10.1080/14697680802126530.

[5] A. Cartea and M. Figueroa. Pricing in Electricity Markets: a Mean Reverting Jump Diffusion Model with Seasonality. *Applied Mathematical Finance, No. 4, December 2005*, 12(4):313–335, 2005.

[6] R. Cont and E. Voltchkova. Integro-Differential Equations for Option Prices in Exponential lévy Models. *Advances in Futures and Options Research*, 9:265–285, 1997.

[7] N. Cufaro Petroni and P. Sabino. Coupling Poisson Processes by Self-decomposability. *Mediterranean Journal of Mathematics*, 14(2):69, 2017.

[8] N. Cufaro Petroni and P. Sabino. Fast Pricing of Energy Derivatives with Mean-reverting Jump-diffusion Processes. available at: https://arxiv.org/abs/1908.03137, 2020.

[9] M. Gardini, P. Sabino, and E. Sasso. A Bivariate Normal Inverse Gaussian Process with Stochastic Delay: Efficient Simulations and Applications to Energy Markets. *Applied Mathematical Finance*, 28(2):178–199, 2021. doi: 10.1080/1350486X.2021.2010106. URL `https://doi.org/10.1080/1350486X.2021.2010106`.

[10] T. Kluge. Pricing Swing Options and other Electricity Derivatives. Technical report, 2006. PhD Thesis, Available at http://perso-math.univ-mlv.fr/users/bally.vlad/publications.html.

[11] P. Sabino. Exact Simulation of Variance Gamma-Related OU Processes: Application to the Pricing of Energy Derivatives. *Applied Mathematical Finance*, 0(0):1–21, 2020. doi: 10.1080/1350486X.2020.1813040.

[12] P. Sabino and N. Cufaro-Petroni. Gamma-Related Ornstein–Uhlenbeck Processes and Their Simulation. *Journal of Statistical Computation and Simulation*, 0(0):1–26, 2020. doi: 10.1080/00949655.2020.1842408.

[13] K. Sato. *Lévy Processes and Infinitely Divisible Distributions*. Cambridge U.P., Cambridge, 1999.

[14] E. Schwartz. The Stochastic Behavior of Commodity Prices: Implications for Valuation and Hedging. *The Journal of Finance*, 52(3):923–973, 1997. doi: https://doi.org/10.1111/j.1540-6261.1997.tb02721.x. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1540-6261.1997.tb02721.x`.

[15] P. Schwartz and J.E. Smith. Short-term Variations and Long-term Dynamics in Commodity Prices. *Management Science*, 46(7):893–911, 2000.

[16] J. Seifert and M. Uhrig-Homburg. Modelling jumps in electricity prices - theory and empirical evidence. *Review of derivatives research*, 10(1):59 – 85, 2007. ISSN 1380-6645, 1573-7144.

[17] R. Seydel. *Tools for Computational Finance*. Universitext (1979). Springer, 2004. ISBN 9783540406044.

[18] S.E. Shreve. *Stochastic Calculus for Finance Volume 2*. Springer, 2004.