ON THE CLASS OF MODULATED VOLTERRA STOCHASTIC VOLATILITY PROCESSES

by

Kees Groeneweg (CID: 01287450)

Department of Mathematics
Imperial College London
London SW7 2AZ
United Kingdom

Thesis submitted as part of the requirements for the award of the MSc in Mathematics and Finance, Imperial College London, 2017-2018

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Signature and date:

Acknowledgements

I would like to thank Dr. Antoine Jacquier who enabled me to based my work in this topic, as well as for his constant help and guidance throughout the project.

Contents

1 Introduction

In 2014, J. Gatheral, T. Jaisson and M. Rosenbaum present a thorough analysis of financial time series data from equity markets showing that log-volatility display properties of fractional Brownian Motion.

In the same work, [1], the authors use estimates of the log-volatility process to conclude that, across numerous assets, the same stylized facts are observed. In particular, J. Gatheral, T. Jaisson and M. Rosenbaum corroborate the well-known stylized fact that distribution of log-volatility is very close to be Gaussian and more importantly, that it presents a scaling property with constant smoothness.

These properties lead the authors to suggest a model of log-volatility driven by fractional Brownian Motion $(B_t^H)_{t\in\mathbb{R}}$ with Hurst parameter H, which is a centered self-similar Gaussian process with stationary increments and satisfying the following property:

$$\mathbb{E}[|B_{t+\Delta}^H - B_t^H|^q] = K_q \Delta^{qH} \quad \text{for any } t \in \mathbb{R}, \Delta \geq 0, q \geq 0$$

where $K_q = \int_{-\infty}^{\infty} |x|^q \frac{\exp(-\frac{x^2}{2})}{\sqrt{2\pi}} \mathrm{d}x$.

In addition, they consider a Hurst parameter $H \leq \frac{1}{2}$ to obtain a model consistent with the shape of the volatility surface from equity markets, claiming that "Volatility is Rough".

In a later work, [2], from 2005, C. Bayer, P. Friz and J. Gatheral continue with the assumptions introduced to model log-volatility and introduce the *Rough Bergomi* able to capture the shape of the SPX volatility smile and give impressive fits with only four parameters and a quite small Hurst parameter.

However, when applying this model of forward variance and stock price dynamics to price VIX options, it produces implied volatility smiles approximately flat, a well-known problem of the model. This is due to the fact that the forward variance process, $\mathbb{E}[v_u|\ \mathcal{F}_T]$ is approximately log-normal in the *Rough Bergomi* model, which ultimately passes this distribution to the VIX monitoring formula

$$\mathrm{VIX}_T := \sqrt{\zeta(T)}, \qquad \zeta(T) = \frac{1}{\Theta} \int_T^{T+\Theta} \mathbb{E}[v_u|\ \mathcal{F}_T] \mathrm{d}u$$

Very recently, in 2018, B. Horvath, A. Jacquier and P. Tankov introduced in [3] a set of Modulated Stochastic Volatility models, integrating the *rBergomi* as part of it, and with the capacity of producing non-flat VIX smiles.

This is achieved by introducing a new process Γ that modulates the instantaneous volatility process. More precisely they define the instantaneous volatility process as

$$\sigma_t = \theta(t)\ \exp(\int_0^t \sqrt{\Gamma_s} g(t,s) \mathrm{d}B_s).$$

By doing so, they achieve impressive fits to VIX smiles using a Lévy-driven Ornstein-Uhlenbek process to modulate the above integral.

In Section 2 of this thesis, we will review the *Rough Bergomi* model and show the properties of the model. We will generate both SPX and VIX smiles corroborating the statements above about their shape and present an implementation of the exact simulation proposed in [2].

In Section 3, the set of Modulated Stochastic Volatility models are presented with different implementations of the instantaneous volatility process. One of them is the Hybrid Scheme proposed by M. Bennedsen, A. Lunde and M. S. Pakkanen. We will compare the different simulation methods and propose a standard Stock Price dynamics using the instantaneous volatility dynamics introduced in [3].

This model will be used to price European Call option prices and calibrate SPX smiles on May 14, 2014. The aim here is to show that by introducing the process Γ not only the shape of the VIX smiles are captured but also the fit to SPX smiles can be as good as in the *rBergomi* model.

We will use the Lévy-driven Ornstein-Uhlenbek proposed by B. Horvath, A. Jacquier and P. Tankov and explore the potential of the Cox-Ingersoll-Ross process as the process modulating the integral $\int_0^t \sqrt{\Gamma_s} g(t,s) \mathrm{d}B_s$.

Finally, we will use the Cox-Ingersoll-Ross process to calibrate VIX smiles with the model proposed in [3] and compare the fit to the one obtained there with the Lévy-driven Ornstein-Uhlenbek process.

2 The Rough Bergomi Model

This model introduced in [2] has been studied as a Rough Fractional Volatility Model (RFSV) in both volatility and forward variance dynamics. The Rough Bergomi model (also known as *rBergomi*) will be presented here as a starting model before introducing the class of Modulated Volterra stochastic volatility processes presented in [3] which can be seen as a generalization of the first.

The Rough Bergomi model is capable of replicating some of the stylized facts present in the volatility surfaces in equity markets. A thorough study of these properties is present in [1] where, in particular, the authors show that the increments of the log-volatility of several equity assets display the following properties

- The scaling property: $\mathbb{E}[|log(\sigma_\Delta) - log(\sigma_0)|] = K_q \Delta^{\zeta_q}$.
- Distribution very close to be Gaussian.

Which motivates the authors to suggest the following simple RFSV model ([1], page 14):

$$\sigma_u = \sigma \ \exp(\nu B_u^H) \tag{2.1}$$

Where $(B_u^H)_{u\in\mathbb{R}}$ is a fractional Brownian Motion process with Hurst parameter $H \in (0,1)$ defined on a probability space $(\Omega, \mathcal{F}, (\mathcal{F}_u)_{u\geq 0}), \mathbb{P})$ and $\nu > 0$ is the volatility of volatility.

This process certainly captures the properties mentioned as can be immediately checked by taking increments of the log process:

$$log(\sigma_{u+\Delta}) - log(\sigma_u) = \nu \ (B_{u+\Delta}^H - B_u^H) \tag{2.2}$$

Proceeding in a similar way as done in [2], by taking the Mandelbrot-Van Ness integral representation of fractional Brownian motion and substituting into (2.1) we obtain

$$\sigma_u = \sigma \ \exp(\nu \ C_H \{ \int_{-\infty}^{u} (u-s)^{H-\frac{1}{2}} \mathrm{d}B_s - \int_{-\infty}^{0} (-s)^{H-\frac{1}{2}} \mathrm{d}B_s \})$$

Where $C_H = \sqrt{\frac{2H\Gamma(3/2-H)}{\Gamma(H+1/2)\Gamma(2-2H)}}$ and Γ is the Gamma function.

To avoid working in the entire real line we can drop the integral parts from $-\infty$ to 0 so

$$\sigma_u = \sigma \ \exp(\nu C_H \int_0^u (u-s)^{H-\frac{1}{2}} \mathrm{d}B_s)$$

which give us the instantaneous volatility process of the *rBergomi* model.

The instantaneous variance process is given for free by taking the square of σ_u, $v_u := \sigma_u^2$. The forward variance process can be defined as $\xi_t(u) := \mathbb{E}[v_u|\mathcal{F}_t]$ which is a martingale by the properties of the conditional expectation.

This martingale process has the following explicit dynamics

$$d\xi_t(u) = 2\xi_t(u)\nu C_H(u-t)^{H-\frac{1}{2}}\mathrm{d}B_s \tag{2.3}$$

which are the dynamics of the stochastic Dolans-Dade exponential $\mathcal{E}(.)$. (Again, by the Novikov's condition, it is easy to see that the process is a martingale.)

To see how this dynamics are obtained we can start by expanding the expectation $\mathbb{E}[v_u|\mathcal{F}_t]$

$$\begin{aligned}\mathbb{E}[v_u|\mathcal{F}_t] &= \mathbb{E}[\sigma^2\ \exp(2\nu C_H\int_0^u (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s)|\mathcal{F}_t] \\ &= \mathbb{E}[\sigma^2\ \exp(2\nu C_H\int_0^t (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s)\ \exp(2\nu C_H\int_t^u (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s)|\mathcal{F}_t]\end{aligned}$$

the first term is $\mathcal{F}_t$-measurable so it can be taken out of the conditional expectation and the second term is independent of $(\mathcal{F}_s)_{s\leq t}$ so we have

$$\begin{aligned}\mathbb{E}[v_u|\mathcal{F}_t] &= \sigma^2\ \exp(2\nu C_H\int_0^t (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s)\ \mathbb{E}[\exp(2\nu C_H\int_t^u (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s)] \\ &= \sigma^2\ \exp(2\nu C_H\int_0^t (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s)\ \exp(2(\nu C_H)^2\int_t^u (u-s)^{2H-1}\mathrm{d}s)\end{aligned}$$

where the last equality can be obtained by using the moment generating function of the normal distribution ($\mathbb{E}[\exp(tN)] = \exp(\mu t + \frac{1}{2}\sigma^2 t^2)$) and the fact that

$$2\nu C_H\int_0^t (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s$$

is normally distributed with mean 0 and variance $4(\nu C_H)^2\int_t^u (u-s)^{2H-1}\mathrm{d}s$ (given by the Itô isometry).

Consider $t \leq T \leq u$, then $\xi_T(u)$ can be expressed as follows

$$\begin{aligned}\xi_T(u) = \mathbb{E}[v_u|\mathcal{F}_T] = \\ &= \sigma^2\ \exp(2\nu C_H\int_0^T (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s)\ \exp(2(\nu C_H)^2\int_T^u (u-s)^{2H-1}\mathrm{d}s) \\ &= \sigma^2\ \exp(2\nu C_H\int_0^t (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s + 2\nu C_H\int_t^T (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s) \\ &\quad \exp(2(\nu C_H)^2\int_t^u (u-s)^{2H-1}\mathrm{d}s - 2(\nu C_H)^2\int_t^T (u-s)^{2H-1}\mathrm{d}s)\end{aligned}$$

$$=\xi_t(u) \quad \exp(2\nu C_H \int_t^T (u-s)^{H-\frac{1}{2}} \mathrm{d}B_s - 2(\nu C_H)^2 \int_t^T (u-s)^{2H-1} \mathrm{d}s)$$
$$=\xi_t(u)\ \mathcal{E}(2\nu C_H \int_t^T (u-s)^{H-\frac{1}{2}} \mathrm{d}B_s).$$

this proves that the forward variance process in the *rBergomi* model have the dynamics (2.3).

It is worth noting that by the above we can recover the instantaneous variance process but in a different expression

$$v_u = \mathbb{E}[v_u|\mathcal{F}_u] = \xi_u(u) = \xi_t(u)\ \mathcal{E}(2\nu C_H \int_t^u (u-s)^{H-\frac{1}{2}} \mathrm{d}B_s)$$
$$= \mathbb{E}[v_u|\mathcal{F}_t]\ \mathcal{E}(2\nu C_H \int_t^u (u-s)^{H-\frac{1}{2}} \mathrm{d}B_s)$$

This, with the dynamics of the stock process, give us the rBergomi model proposed by Bayer, Friz and Gatheral in [2] under the risk neutral probability measure $\mathbb{Q}$ on a filtered probability space $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\geq 0}), \mathbb{Q})$

$$\mathrm{d}S_t = S_t\ \sqrt{v_t}\mathrm{d}W_t \tag{2.4}$$
$$v_t = \xi_0(t)\ \mathcal{E}(2\nu C_H \int_0^t (t-s)^{H-\frac{1}{2}} \mathrm{d}B_s) \tag{2.5}$$

In the same work, the authors propose a simulation method that will be presented here in addition to the Python implementation shown at the end of this section. This has been done here to be able to replicate their results and comment the advantages of this model with respect to stochastic volatility models in which the volatility process is driven by standard Brownian Motion.

The method that the authors present is an exact simulation method for which the dependence structure of the two Brownian motions, W_t and B_t, driving the stock and variance dynamics respectively needs to be known.

In addition, and for completeness, the derivation of these results are presented here in the following proposition.

Proposition 2.1. *Let B_t and W_t be two standard Brownian motion defined on the same probability space $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\geq 0}), \mathbb{Q})$ such that $\mathrm{d}\langle B, W\rangle_t = \rho\ \mathrm{d}t$ and define*

$$\tilde{B}_t := \sqrt{2H}\ \int_0^t (t-s)^{H-\frac{1}{2}} dB_s.$$

Then for $v \geq u$

$$\mathbb{E}[\tilde{B}_v\ \tilde{B}_u] = \frac{2H\Gamma(1)\Gamma(H+\frac{1}{2})}{\Gamma(H+\frac{3}{2})}\ v^{2H}\ (\frac{u}{v})^{H+\frac{1}{2}}\ {}_2F_1(\frac{1}{2}-H, 1, H+\frac{3}{2}, \frac{u}{v}) \tag{2.6}$$
$$\mathbb{E}[\tilde{B}_v\ W_u] = \rho\ \frac{\sqrt{2H}}{H+\frac{1}{2}}\ (v^{H+1/2} - (v-u)^{H+1/2})$$
$$\mathbb{E}[\tilde{B}_u\ W_v] = \rho\ \frac{\sqrt{2H}}{H+\frac{1}{2}}\ u^{H+1/2}$$

Proof. In the first equality, notice that $\tilde{B}_v$ and $\tilde{B}_u$ are stochastic integrals with respect to the same Brownian motion B_t so by the Itô isometry and the change of variables $x = s/u$ we obtain

$$\begin{aligned}
\mathbb{E}[\tilde{B}_v\ \tilde{B}_u] &= \mathbb{E}[2H\ \int_0^v (v-s)^{H-\frac{1}{2}}\mathrm{d}B_s\ \int_0^v \mathbb{1}_{\{s\le u\}}(u-s)^{H-\frac{1}{2}}\mathrm{d}B_s] \\
&= 2H\int_0^v (v-s)^{H-\frac{1}{2}}\mathbb{1}_{\{s\le u\}}(u-s)^{H-\frac{1}{2}}\mathrm{d}s = 2H\int_0^u (v-s)^{H-\frac{1}{2}}(u-s)^{H-\frac{1}{2}}\mathrm{d}s \\
&= 2H\int_0^1 (v-xu)^{H-\frac{1}{2}}(u-xu)^{H-\frac{1}{2}}u\ \mathrm{d}x \\
&= 2Hu^{H+\frac{1}{2}}v^{H-\frac{1}{2}}\int_0^1 (1-x)^{H-\frac{1}{2}}(1-u/vx)^{H-\frac{1}{2}}\mathrm{d}x
\end{aligned}$$

Using Euler's type formula

$$\int_0^1 x^{b-1}(1-x)^{c-b-1}(1-zx)^{-a}\mathrm{d}x = B(b,c-b)\ {}_2F_1(a,b;c,z)$$

where B is the Beta function $B(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$ and ${}_2F_1$ the hypergeometric function,

we obtain

$$\mathbb{E}[\tilde{B}_v\ \tilde{B}_u] = \frac{2H\ \Gamma(1)\Gamma(H+\frac{1}{2})}{\Gamma(H+\frac{3}{2})}\ v^{2H}\ (\frac{u}{v})^{H+\frac{1}{2}}\ {}_2F_1(\frac{1}{2}-H,1,H+\frac{3}{2},\frac{u}{v})$$

Notice here that the absolute value of u/v is smaller than 1, an assumption needed to use the Euler's type formula.

It is interesting to see that the Gauss's Hypergeometric Theorem can be applied to the above expression (when $v = u$) to obtain the variance of the process.

$${}_2F_1(\frac{1}{2}-H,1,H+\frac{3}{2},1) = \frac{\Gamma(H+3/2)\Gamma(2H)}{\Gamma(2H+1)\Gamma(H+1/2)} = \frac{\Gamma(H+3/2)}{2H\Gamma(H+1/2)}$$

with the last equality is given by the identity $\Gamma(z+1) = \Gamma(z)z$. Then,

$$\begin{aligned}
\mathrm{Var}[\tilde{B}_v] &= \frac{2H\ \Gamma(1)\Gamma(H+\frac{1}{2})}{\Gamma(H+\frac{3}{2})}\ v^{2H}\ {}_2F_1(\frac{1}{2}-H,1,H+\frac{3}{2},1) \\
&= \frac{2H\ \Gamma(1)\Gamma(H+\frac{1}{2})}{\Gamma(H+\frac{3}{2})}\ v^{2H}\ \frac{\Gamma(H+3/2)}{2H\Gamma(H+1/2)} = \ v^{2H}
\end{aligned} \tag{2.7}$$

For the remains identities consider $W_t = \rho B_t + \sqrt{1-\rho^2}B_t^{\perp}$ where $B_t^{\perp}$ is a standard Brownian motion such that $\mathrm{d}\langle B, B^{\perp}\rangle_t = 0\mathrm{d}t$.

Then by using this expression we have

$$\begin{aligned}
\mathbb{E}[\tilde{B}_v\ W_u] &= \mathbb{E}[\sqrt{2H}\ \int_0^v (v-s)^{H-\frac{1}{2}}\mathrm{d}B_s\ \int_0^v \mathbb{1}_{\{s\le u\}}\mathrm{d}W_s] \\
&= \mathbb{E}[\sqrt{2H}\ \int_0^v (v-s)^{H-\frac{1}{2}}\mathrm{d}B_s\ \int_0^v \mathbb{1}_{\{s\le u\}}\mathrm{d}(\rho B_s + \sqrt{1-\rho^2}B_s^{\perp})] \\
&= \mathbb{E}[\sqrt{2H}\ \int_0^v (v-s)^{H-\frac{1}{2}}\mathbb{1}_{\{s\le u\}}\rho\ \mathrm{d}\langle B,B\rangle_s] \\
&+ \mathbb{E}[\sqrt{2H}\ \int_0^v (v-s)^{H-\frac{1}{2}}\mathbb{1}_{\{s\le u\}}\sqrt{1-\rho^2}\ \mathrm{d}\langle B,B^{\perp}\rangle_s]
\end{aligned}$$

$$= \sqrt{2H} \int_0^u (v-s)^{H-\frac{1}{2}} \rho \, \mathrm{d}s = \rho \, \frac{\sqrt{2H}}{H+\frac{1}{2}} \, (v^{H+1/2} - (v-u)^{H+1/2})$$

Similarly

$$\begin{aligned}
\mathbb{E}[\tilde{B}_u \, W_v] &= \mathbb{E}[\sqrt{2H} \int_0^v \mathbb{1}_{\{s \le u\}} (u-s)^{H-\frac{1}{2}} \mathrm{d}B_s \int_0^v \mathrm{d}W_s] \\
&= \mathbb{E}[\sqrt{2H} \int_0^v (u-s)^{H-\frac{1}{2}} \mathbb{1}_{\{s \le u\}} \mathrm{d}B_s \int_0^v \mathrm{d}(\rho B_s + \sqrt{1-\rho^2} B_s^{\perp})] \\
&= \mathbb{E}[\sqrt{2H} \int_0^v (u-s)^{H-\frac{1}{2}} \mathbb{1}_{\{s \le u\}} \rho \, \mathrm{d}\langle B, B \rangle_s] \\
&+ \mathbb{E}[\sqrt{2H} \int_0^v (u-s)^{H-\frac{1}{2}} \mathbb{1}_{\{s \le u\}} \sqrt{1-\rho^2} \, \mathrm{d}\langle B, B^{\perp} \rangle_s] \\
&= \sqrt{2H} \int_0^u (u-s)^{H-\frac{1}{2}} \rho \, \mathrm{d}s = \rho \, \frac{\sqrt{2H}}{H+\frac{1}{2}} \, u^{H+1/2}
\end{aligned}$$

□

Remark 2.2. The expression (2.6) is slightly different from the one given in [2] but consistent with the results. An interesting way in which the reader can be check the consistency of the expression presented here is by computing $\mathrm{Var}[\tilde{B}_v]$ directly using the Itô isometry instead of the Hypergeometric Theorem from Gauss.

$$\begin{aligned}
\mathbb{E}[\tilde{B}_v \, \tilde{B}_v] &= \mathbb{E}[2H \int_0^v (v-s)^{H-\frac{1}{2}} \mathrm{d}B_s \int_0^v (v-s)^{H-\frac{1}{2}} \mathrm{d}B_s] \\
&= 2H \int_0^v (v-s)^{2H-1} \mathrm{d}s = v^{2H}
\end{aligned}$$

2.1 Simulation of the rBergomi Model

The simulation method proposed by Bayer, Friz and Gatheral can be resumed in the three steps

- Compute the following $2N \times 2N$ joint covariance matrix

$$C = \begin{pmatrix}
\mathbb{E}[\tilde{B}_{t_1}\tilde{B}_{t_1}] & \dots & \mathbb{E}[\tilde{B}_{t_1}\tilde{B}_{t_N}] & \mathbb{E}[\tilde{B}_{t_1}W_{t_1}] & \dots & \mathbb{E}[\tilde{B}_{t_1}W_{t_N}] \\
\mathbb{E}[\tilde{B}_{t_2}\tilde{B}_{t_1}] & \dots & \mathbb{E}[\tilde{B}_{t_2}\tilde{B}_{t_N}] & \mathbb{E}[\tilde{B}_{t_2}W_{t_1}] & \dots & \mathbb{E}[\tilde{B}_{t_2}W_{t_N}] \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\mathbb{E}[\tilde{B}_{t_N}\tilde{B}_{t_1}] & \dots & \mathbb{E}[\tilde{B}_{t_N}\tilde{B}_{t_N}] & \mathbb{E}[\tilde{B}_{t_N}W_{t_1}] & \dots & \mathbb{E}[\tilde{B}_{t_n}W_{t_N}] \\
\mathbb{E}[W_{t_1}\tilde{B}_{t_1}] & \dots & \mathbb{E}[W_{t_1}\tilde{B}_{t_N}] & \mathbb{E}[W_{t_1}W_{t_1}] & \dots & \mathbb{E}[W_{t_1}W_{t_N}] \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\mathbb{E}[W_{t_N}\tilde{B}_{t_1}] & \dots & \mathbb{E}[W_{t_N}\tilde{B}_{t_N}] & \mathbb{E}[W_{t_N}W_{t_1}] & \dots & \mathbb{E}[W_{t_N}W_{t_N}]
\end{pmatrix}$$

where $T_i = i * h, \quad h = \frac{T}{N}$ and $i = 0, \dots, N$. (Notice that $\mathbb{E}[\tilde{B}_t] = \mathbb{E}[W_t] = 0$)

- For each path, generate a discretization of the paths $(\tilde{B}_t)_{0\leq t\leq T}$ and $(W_t)_{0\leq t\leq T}$ as follows
  - Simulate a multivariate standard normal vector $\mathbf{Z}$ of size $2N \times 1$
  - Use the Cholesky decomposition to get the lower triangular matrix $\mathbf{L}$ such that $\mathbf{LL^T} = C$
  - Multiply to obtain the paths,

$$[\tilde{B}_{t_1} \dots \tilde{B}_{t_N}, W_{t_i} \dots Wt_N]^T = \mathbf{LZ}$$

- Finally, use Euler discretization to obtain $(v_{t_i})_{i=0,\dots,N}$ and $(S_{t_i})_{i=0,\dots,N}$

$$S_{t_i} = S_{t_{i-1}} \exp(\sqrt{v_{t_{i-1}}}\,(W_{t_i} - W_{t_{i-1}}) - \frac{1}{2}v_{i-1}h) \tag{2.8}$$

$$v_{t_i} = \xi_0 \exp(\mu\tilde{B}_{t_i} - \frac{1}{2}\mu^2 t_i^{2H}) \tag{2.9}$$

with $\mu = \frac{2\nu C_H}{\sqrt{2H}}$, $\quad h = \frac{T}{N}$

This simulation method is precise but, as pointed out by the authors, extremely slow. For this reason and because we are interested in simulating a more general set of Rough Stochastic Volatility Models, the faster Hybrid Scheme introduced by M. Bennedsen, A. Lunde and M. S. Pakkanen in [4] (and presented in 3.1.3) will be used here to generate some of the results below.

Before presenting these results, let us consider some justification of equations (2.8) and (2.9)

By discretizing the integrals, the stock process can be rewritten as follows

$$S_{t_i} = S_{t_0} \exp(\int_0^{t_i} \sqrt{v_s}\mathrm{d}W_s - \frac{1}{2}\int_0^{t_i} v_s\mathrm{d}s) = S_{t_0} \exp(\sum_{k=0}^{i-1}\int_{t_k}^{t_{k+1}} \sqrt{v_s}\mathrm{d}W_s - \frac{1}{2}\sum_{k=0}^{i-1}\int_{t_k}^{t_{k+1}} v_s\mathrm{d}s)$$
$$= S_{t_0} \exp(\sum_{k=0}^{i-2}\int_{t_k}^{t_{k+1}} \sqrt{v_s}\mathrm{d}W_s - \frac{1}{2}\sum_{k=0}^{i-2}\int_{t_k}^{t_{k+1}} v_s\mathrm{d}s) \exp(\int_{t_{i-1}}^{t_i} \sqrt{v_s}\mathrm{d}W_s - \frac{1}{2}\int_{t_{i-1}}^{t_i} v_s\mathrm{d}s)$$

and by taking left hand approximations we obtain (2.8)

$$= S_{t_{i-1}} \exp(\int_{t_{i-1}}^{t_i} \sqrt{v_s}\mathrm{d}W_s - \frac{1}{2}\int_{t_{i-1}}^{t_i} v_s\mathrm{d}s) \approx S_{t_{i-1}} \exp(\sqrt{v_{t_{i-1}}}\int_{t_{i-1}}^{t_i} \mathrm{d}W_s - \frac{1}{2}v_{t_{i-1}}\int_{t_{i-1}}^{t_i} \mathrm{d}s)$$
$$= S_{t_{i-1}} \exp(\sqrt{v_{t_{i-1}}}(W_{t_i} - W_{t_{i-1}}) - \frac{1}{2}v_{t_{i-1}}h)$$

Similarly for the variance process we have

$$v_{t_i} = \xi_0\, \mathcal{E}(2\nu C_H \int_0^{t_i} (t_i - s)^{H-\frac{1}{2}}\mathrm{d}B_s) = \xi_0\, \mathcal{E}(\mu \int_0^{t_i} \sqrt{2H}(t_i - s)^{H-\frac{1}{2}}\mathrm{d}B_s)$$
$$= \xi_0 \exp(\mu\tilde{B}_{t_i} - \frac{1}{2}2H\mu^2 \int_0^{t_i} (t_i - s)^{2H-1}\mathrm{d}s) = \xi_0 \exp(\mu\tilde{B}_{t_i} - \frac{1}{2}\mu^2 t_i^{2H})$$

Notice that the function $g(t-s) = (t-s)^{H-\frac{1}{2}}$ has a singularity on $s = t$, however, we have the following remark

Remark 2.3. The integral $\int_0^t (t-s)^{2H-1}\mathrm{d}s$ is convergent for all $H \geq 0$.

This can be seen by taking limits and a suitable change of variables: $(u = t - s)$

$$\int_0^t (t-s)^{2H-1}\mathrm{d}s = -\int_t^0 (u)^{2H-1}\mathrm{d}u = \lim_{c\to 0_+}\int_c^t (u)^{2H-1}\mathrm{d}u$$
$$= \lim_{c\to 0_+}\frac{1}{2H}[t^{2H} - c^{2H}] = \frac{1}{2H}t^{2H}$$

Since $2H \geq 0$

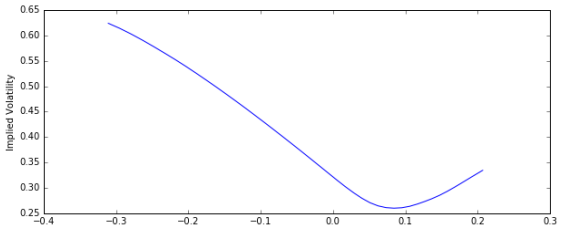
Then the random variable $2\nu C_H \int_0^t (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s$, as already said before, has a finite second moment that can be obtained by integrating the deterministic integral above.

2.2 Consistency with observed SPX smiles

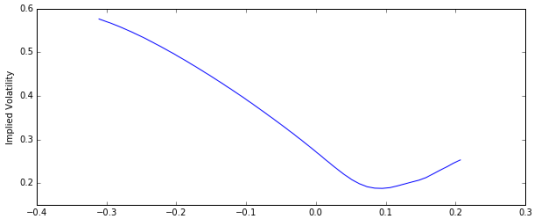
As shown by C. Bayer, P. Friz, J. Gatheral in ([2], page 18), the fits of the Rough Bergomi model to observed implied volatility smiles can be very impressive. This feature and the small number of parameters of the model make this, and (RFSV) models in general, superior to conventional stochastic volatility in terms of capturing the shape of the volatility surface.

Here, instead of presenting the fits to real data (which can be seen in [2]), we present different graphs of the implied volatility as a variable of the Hurst parameter H. This may help the reader to understand the role of this parameter.

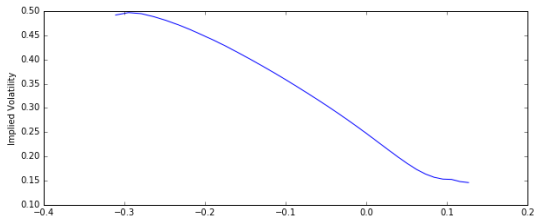
The volatility of volatility has been fixed and chosen to match as much as possible the shortest dated SPX smile as of February 4, 2010. ([2], page 19).



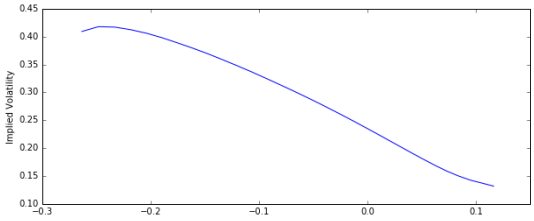
$H = 0.07$



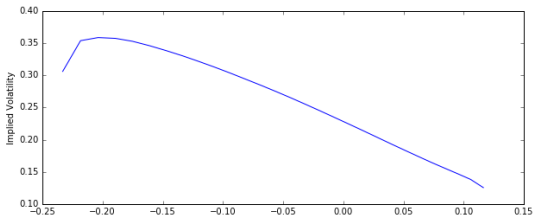
$H = 0.14$



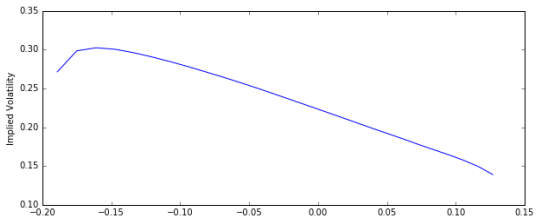
$H = 0.21$



$H = 0.28$



$H = 0.35$



$H = 0.45$

Figure 3: Generated implied volatilities as a function of the log-strike $k := \log(K/S_0)$ from the *rBergomi* model. Prices have been simulated with the Hybrid Scheme and using the parameters of the table below.

T	S_0	ν	ρ	M	n	k
0.041	1.0	1.2287	-0.9	100000	500	1

Notice that ν has been chosen to obtain $\mu = 1.9$.([2], page 18). The parameters M, n and k follow the notation used in 3.1.3.

2.3 Failure to fit VIX smiles

The VIX index measures the expected volatility of U.S. stocks over a period of 30 days. It is computed from Put and Call option prices on the S&P 500 Index. A thorough explanation of how the Index is calculated can be found in [11].

Despite the consistency of the *rBergomi* model with SPX smiles, it is a well known problem that this models fails to capture the VIX smiles. It produces implied volatility smiles that are approximately flat. The reason of this is that the forward variance process is approximately log-normal. Let us see this.

The following monotoring formula is used to replicate the Index

$$\text{VIX}_T := \sqrt{\zeta(T)}, \qquad \zeta(T) = \frac{1}{\Theta}\int_T^{T+\Theta} \mathbb{E}[\, v_u \mid \mathcal{F}_T]\mathrm{d}u$$

where Θ is one month and, as seen before, $\mathbb{E}[\, v_u \mid \mathcal{F}_T] = \xi_T(u)$ is the forward variance process which can be rewritten as

$$\mathbb{E}[\, v_u \mid \mathcal{F}_T] = \xi_t(u)\ \mathcal{E}(2\nu C_H \int_t^T (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s)$$

then, taking the natural logarithm, the log-forward variance is

$$\log(\xi_T(u)) = \eta(u) + 2\nu C_H \int_t^T (u-s)^{H-\frac{1}{2}}\mathrm{d}B_s$$

which is log-normal for any fixed $u \geq T$.

This eventually makes the VIX approximately log-normal producing flat volatility smiles which differs from the observed volatility smiles form the markets that are upward-sloping to the right.

Here we checked this feature of the model numerically. This have been done using the rectangle scheme presented by B.Horvath, A.Jacquier and P.Tankov in [3] meant to simulate VIX options prices with a more general set of forward variance processes that include the *rBergomi*.(See 3.4)

To recover the *rBergomi* model from the more general set of Modulated Volterra Stochastic Volatility models we only need to take the Γ process to be constant an equal to 1. Where Γ is the process that modulates the instantaneous volatility process σ_t as we will see in 3.

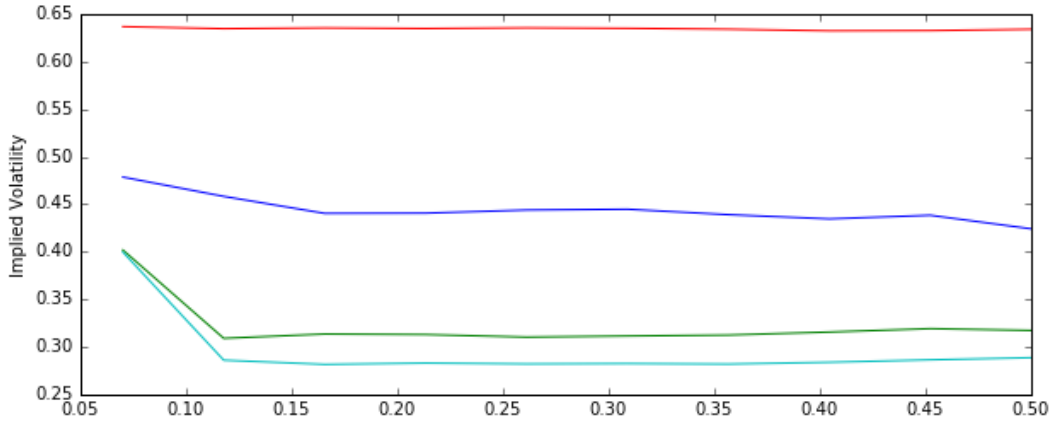


Figure 4: Implied Volatility smiles as a function of the log-strike, $k = \mathrm{K}/\mathrm{S}_0$, produced by the *rBergomi* model using the rectangle scheme presented in [3]. All computations are based in 5000 Monte Carlo replications, $n = 100$, NYear $= 500$. The rest of parameters producing the flat smiles from top to bottom are given in the table below.
A similar figure can be seen in ([3], pag 13)

T	x	α	H
1.0	0.053	0.4	0.1
0.3	0.053	0.2	0.1
1.0	0.053	0.2	0.1
1.0	0.053	0.2	0.2

2.4 Implementation

```
%%cython --cplus --force
from libcpp.vector cimport vector
from scipy.special import gamma as Gamma
from scipy.special import hyp2f1 as F1
import numpy as np
cimport numpy as np

cdef extern from "math.h":
    double sqrt(double m)
    double exp(double m)
    double pow(double base, double exponent)

cdef class rBergomi(object):

    cdef public double H, mu, rho, gamma, T;
    cdef public double mean, var, covij;
    cdef public vector[ vector[double] ] BW, V, S;
    cdef public int N, M;
    cpdef public cov;

    def __cinit__(self, H, mu, rho):
        self.H = H
        self.mu = mu
        self.rho = rho
        self.gamma = H - 0.5

    cdef double G(self, double x):
        return (2.*self.H)*(Gamma(1.0)*Gamma(self.H+0.5)/Gamma(self.H+1.5))
               *pow(x, self.H + 0.5)*F1(-self.gamma, 1., self.H + 1.5, x);

    cpdef covariance(self, int N, double T):
        self.T = T;
        self.N = N;
        C = np.zeros((2*N, 2*N));
        cdef double h = T/float(N);
        cdef double Dh = sqrt(2.*self.H)/(self.H + 0.5)*self.rho;
        cdef double ti;
        cdef double tj;

        for i in range (1, N + 1):
            ti = i*h;
            C[i - 1][i - 1] = pow(ti, 2.0*self.H); #B
```

```
43              C[i - 1 + N][i - 1 + N] = ti;           #W
44              for j in range(1, i):
45                  tj = j*h; #ti>tj
46                  C[i - 1][j - 1] = pow(ti, 2.0*self.H)*self.G(tj/ti);
47                  C[j - 1][i - 1] = C[i - 1][j - 1];
48                  C[i - 1 + N][j - 1 + N] = tj;
49                  C[j - 1 + N][i - 1 + N] = C[i - 1 + N][j - 1 + N];
50                  C[i - 1 + N][j - 1] = Dh*pow(ti, self.H + 0.5);
51                  C[j - 1][i - 1 + N] = C[i - 1 + N][j - 1];
52              for j in range(i, N + 1):
53                  tj = j*h; #ti<=tj
54                  C[i - 1 + N][j - 1] = Dh*(pow(tj, self.H + 0.5)
55                                          - pow(tj - ti, self.H + 0.5));
56                  C[j - 1][i - 1 + N] = C[i - 1 + N][j - 1];
57
58          self.cov = C;
59
60      cpdef B_W(self, int M):
61          self.M = M;
62          cdef vector[ vector[double] ] BW;
63          BW.resize(M);
64          for m in range(M):
65              BW[m].resize(2*self.N);
66              BW[m] = np.random.multivariate_normal(np.zeros(2*self.N), self.cov );
67
68          self.BW = BW;
69
70      cpdef meanVarB(self):
71          cdef double mean = 0.;
72          cdef double var = 0.;
73          for m in range (self.M):
74              mean += self.BW[m][self.N - 1];
75              var += self.BW[m][self.N - 1]*self.BW[m][self.N - 1];
76
77          mean /= float(self.M);
78          var /= float(self.M);
79          var -= mean;
80          self.mean = mean;
81          self.var = var;
82
83
84
85
86
87
```

```
88      cpdef covB(self, int i, int j):
89          cdef double meani = 0.;
90          cdef double meanj = 0.;
91          cdef double covij = 0.;
92          for m in range (self.M):
93              meani += self.BW[m][i];
94              meanj += self.BW[m][j];
95              covij += self.BW[m][i]*self.BW[m][j];
96
97          meani /= float(self.M);
98          meanj /= float(self.M);
99          covij /= float(self.M);
100         covij -= meani*meanj;
101         self.covij = covij;
102
103     cpdef volPath(self, double varOrSigma, c):
104         cdef double h = self.T/float(self.N);
105         cdef double ti;
106         cdef vector[ vector[double] ] Vol;
107         Vol.resize(self.M);
108
109         if (c == "variance"):
110             for m in range(self.M):
111                 Vol[m].resize(self.N + 1);
112                 Vol[m][0] = sqrt(varOrSigma);
113                 for i in range(0, self.N):
114                     ti = (i+1)*h;
115                     Vol[m][i + 1] = Vol[m][0]*sqrt( exp(self.mu*self.BW[m][i]
116                                     - 0.5*self.mu*self.mu*pow(ti, 2.*self.H)) );
117
118         elif (c == "instantaneous"):
119             for m in range(self.M):
120                 Vol[m].resize(self.N + 1);
121                 Vol[m][0] = varOrSigma;
122                 for i in range(0, self.N):
123                     ti = (i + 1)*h;
124                     Vol[m][i + 1] = Vol[m][0]*exp( self.mu*self.BW[m][i] );
125
126         self.V = Vol;
127
128
129
130
131
132
```

```
133     cpdef stockPath(self, double S0):
134         cdef double h = self.T/float(self.N);
135         cdef double volti;
136         cdef vector[ vector[double] ] S;
137         S.resize(self.M);
138 
139         for m in range(self.M):
140             S[m].resize(self.N + 1);
141             S[m][0] = S0;
142             volti = self.V[m][0];
143             S[m][1] = S[m][0]*exp(volti*self.BW[m][self.N] - 0.5*h*volti*volti);
144             for i in range(1, self.N):
145                 volti = self.V[m][i];
146                 S[m][i + 1] = S[m][i]*exp(volti*(self.BW[m][i + self.N]
147                             - self.BW[m][i - 1 + self.N]) - 0.5*h*volti*volti);
148 
149         self.S = S;
```

Notice that instead of using the Cholesky decomposition, we are directly sampling from the multivariate $\mathcal{N}(\mathbf{0}, C)$ using he method

```
np.random.multivariate_normal(np.zeros(2*self.N), self.cov )
```

3 Modulated Volterra Stochastic Volatility models

In the previous section, we saw that the rough Bergomi model while being able to capture the SPX smile better than standard stochastic volatility models failed to calibrate VIX smiles. As an attempt to obtain better fits to VIX smiles B.Horvath, A.Jacquier and P.Tankov introduced the set of Modulated Volterra Stochastic Volatility processes in [3]. In addition, they present results on simulation schemes for this set of models and a impressive fit to VIX smiles on May 14, 2014. These results were obtained using a Lévy-driven Ornstein-Uhlenbek process to modulate the instantaneous volatility process as will be explained below.

In this section we will corroborate that this set of models is still able to capture the shape of SPX smiles obtaining similar shapes and fits as the ones observed in section 2 with the *rBergomi* model. For this numerical illustration we will use the Lévy-driven Ornstein-Uhlenbek process proposed by B.Horvath, A.Jacquier and P.Tankov as well as another regular affine process, the Cox-Ingersoll-Ross process.

Then we will explore the capacity of this model together with the Cox-Ingersoll-Ross process to capture the shape of the VIX smiles on May 14, 2014.

3.1 Instantaneous Volatility process

Let us first consider the instantaneous volatility process as introduced in [3]. For the remaining of the work only the 1-dimensional case is considered.

B.Horvath, A.Jacquier and P.Tankov introduced the following dynamics for the instantaneous volatility process

$$\sigma_t = \theta(t)\ \exp(\int_0^t \sqrt{\Gamma_s} g(t,s)\mathrm{d}B_s) \tag{3.1}$$

where B standard Brownian motion defined on a probability space $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\geq 0}), \mathbb{Q})$, Γ is a time-homogeneous positive conservative affine process independent of B and g(t,s) is a deterministic kernel function such that

$$\int_0^t g(t,s)^2 \mathrm{d}s < \infty$$

for all $t \geq 0$. $\theta(t)$ is a deterministic function able to capture the initial volatility curve.

The process Γ has the capability to modulate the integral $\int_0^t \sqrt{\Gamma_s} g(t,s)\mathrm{d}B_s$ so that forward variance has a distribution different from the log-normal seen in the *rBergomi* model which, ultimately, can modulate the VIX implied volatility smiles providing non-flat curves.

However, notice that if the information (the filtration) of the Γ process is known, then we can recover a log-normal distribution. The process,

$$X_t := \int_0^t \sqrt{\Gamma_s} g(t,s) \mathrm{d}B_s$$

conditioned to the process Γ is normally distributed with mean zero and variance $\int_0^t \Gamma_s g(t,s)^2 \mathrm{d}s$.

In addition, by using the tower property of the expectation we can see that the process X_t has always mean zero (having or not information on the Γ process)

$$\mathbb{E}[X_t \mid \Gamma] = \mathbb{E}[\int_0^t \sqrt{\Gamma_s} g(t,s) \mathrm{d}B_s \mid \Gamma] = \mathbb{E}[\, \mathbb{E}[\int_0^t \sqrt{\Gamma_s} g(t,s) \mathrm{d}B_s \mid \Gamma \,]\,] = 0$$

The covariance structure of this process can be obtained similarly. Assume $v \geq u$ then

$$\begin{aligned}
\mathbb{E}[X_v X_u] &= \mathbb{E}[\int_0^v \sqrt{\Gamma_s} g(v,s) \mathrm{d}B_s \int_0^u \sqrt{\Gamma_s} g(u,s) \mathrm{d}B_s \,] \\
&= \mathbb{E}[\, \mathbb{E}[\int_0^v \sqrt{\Gamma_s} g(v,s) \mathrm{d}B_s \int_0^v \mathbb{1}_{\{s \leq u\}} \sqrt{\Gamma_s} g(u,s) \mathrm{d}B_s \mid \Gamma] \,] \\
&= \mathbb{E}[\, \mathbb{E}[\int_0^v \Gamma_s g(v,s) g(u,s) \mathbb{1}_{\{s \leq u\}} \mathrm{d}s \mid \Gamma] \,] = \mathbb{E}[\int_0^u \Gamma_s g(v,s) g(u,s) \mathrm{d}s \,]
\end{aligned}$$

where the last equality is given by the Itô isometry which holds since the filtration of the Γ process is independent from the filtration of the Brownian motion B.

Finally, by using Tonellis theorem to change order of integration, we obtain

$$\mathbb{E}[X_v X_u] = \int_0^u \mathbb{E}[\, \Gamma_s \,] g(v,s) g(u,s) \mathrm{d}s \tag{3.2}$$

Possible due to the fact that the process Γ has being conveniently defined as being positive.

Notice here that the dynamics (3.1) integrates the *rBergomi* model in the set of Modulated Volterra Stochastic Volatility models. This can be seen by setting

- $\Gamma_t(\omega) = 1$ for all $\omega \in \Omega$ and $t \geq 0$
- $g(t,s) = \alpha (t-s)^{H-\frac{1}{2}}$ where $\alpha = \nu \sqrt{\frac{2H\Gamma(3/2-H)}{\Gamma(H+1/2)\Gamma(2-2H)}}$ and ν is the volatility of volatility

The above is very similar to how rough Bergomi model is presented in [3] but the coefficient α is slightly different. Here α has been chosen to be consistent with the definition of the rBergomi model as proposed by Bayer, Friz and Gatheral in [2].

In either case this might result in small difference between the volatility of volatility as defined in both cases.

3.1.1 Simulation of the instantaneus volatility process

Here is presented an easy and straightforward method to simulate the instantaneous volatility process

$$\sigma_t = \theta(t)\ \exp(\int_0^t \sqrt{\Gamma_s} g(t,s)\mathrm{d}B_s) \tag{3.3}$$

To simulate this process it will be assumed that the path $(\Gamma_s)_{0\leq s\leq t}$ is given. As a first approach we can start by discretizing the integral $X_t := \int_0^t \sqrt{\Gamma_s} g(t,s)\mathrm{d}B_s$ by taking left hand approximations, which is a standard Riemann sum approximation.

To do so, we can start by discretizing the time interval $[0,T]$ where the process will be simulated. Here a a equidistant grid $\{t_0, t_1, t_2, ..., t_N\}$ is chosen where $t_i = ih$ and $h = \frac{1}{N}$. Then the process X_t can be rewritten as a sum and approximated as follows

$$\begin{aligned} X_{t_i} &= \int_0^{t_i} \sqrt{\Gamma_s} g(t_i,s)\mathrm{d}B_s = \sum_{j=0}^{i-1} \int_{t_j}^{t_{j+1}} \sqrt{\Gamma_s} g(t_i,s)\mathrm{d}B_s \approx \\ &\approx \sum_{j=0}^{i-1} \sqrt{\Gamma_{t_j}} g(t_i,t_j) \triangle B_{t_j} \end{aligned} \tag{3.4}$$

where $\triangle B_{t_j} = B_{t_{j+1}} - B_{t_j}$ can be obtained by simulating a standard Brownian motion path on the grid $\{t_0, t_1, t_2, ..., t_N\}$ or directly by the identity $B_{t_{j+1}} - B_{t_j} = \sqrt{h} Z_j$ where $(Z_i)_{i\in\{0,1,...,N-1\}}$ are independent samples from the standard normal distribution $\mathcal{N}(0,1)$.

The simulation of the instantaneous volatility process on the grid $\{t_0, t_1, t_2, ..., t_N\}$ would simply consists on simulating the process X_t on the same grid and set σ_{t_i} as

$$\sigma_{t_i} = \theta(t_i)\ \exp(\sum_{j=0}^{i-1} \sqrt{\Gamma_{t_j}} g(t_i,t_j) \triangle B_{t_j}) \tag{3.5}$$

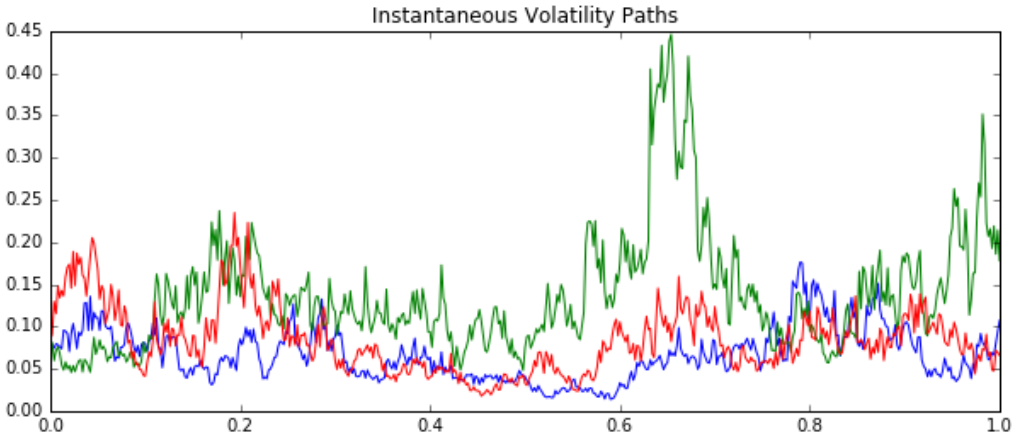


Figure 5: 3 simulated paths of the instantaneous volatility process generated using the method explained above. The Γ process has been set to 1 for all $(\omega, t) \in \Omega \times \mathbb{R}_{t\geq 0}$. Paths computed with 500 time steps, T = 1.0, $\sigma_0 = 0.085$ and H= 0.21.

The method presented above is fast and straightforward. It is capable to produce meaningful results when used for example to calibrate SPX smiles, but is not as precise as methods where a covariance structure and a kernel approximation is used such as in the exact simulation method presented in [2].

Since the approximation has been apply to the process $(X_s)_{s\leq t}$, one way of testing the accuracy of these methods is by simulating paths $(X_s)_{s\leq T}$ and numerically computing the variance of the random variable X_T for a given T.

The next figure shows how the previous method fails to approximate this value for small Hurst parameter H. To be able to compare this method with the exact simulation method discussed in the previous section the following is assumed

- $\Gamma_t(\omega) = 1$ for all $\omega \in \Omega$ and $t \geq 0$
- $g(t,s) = \alpha(t-s)^{H-\frac{1}{2}}$ where $\alpha = \nu\sqrt{\frac{2H\Gamma(3/2-H)}{\Gamma(H+1/2)\Gamma(2-2H)}}$ and ν is the volatility of volatility
- $\nu = \sqrt{\frac{\Gamma(H+1/2)\Gamma(2-2H)}{\Gamma(3/2-H)}}$

with this choice we have $X_T = \tilde{B}_T$ as defined in section 2. Then, from proposition 2.1 we have the equality $Var[X_T] = T^{2H}$ which is used to compare the accuracy of the methods.

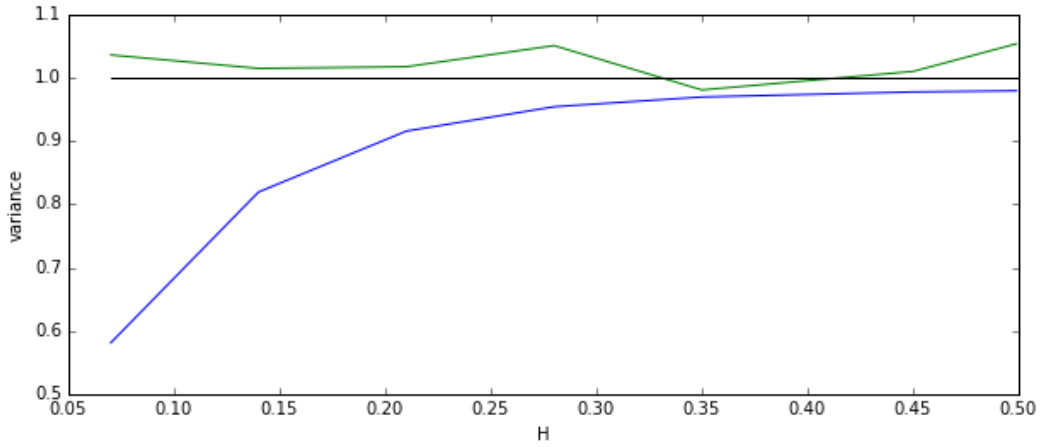


Figure 6: Var$[X_T]$ with $T = 1$. computed using Monte Carlo simulations using the Riemann sum method (blue line) with 300 time steps and 10.000 trajectories. In green: exact simulation method presented by Bayer, Friz and Gatheral in [2] with 100 time steps and 2000 trajectories.

Comment 3.1. *The randomness observed in the computations of Var$[X_T]$, using the exact method, is due to the low number of replications. However, this simulation method gives values very close to the theoretical ones even for number of replications used.*

Notice here that the the exact simulation method gives a variance of X_T very close to 1 for any Hurst parameter H while the Riemann sum method presents a low accuracy for small H values.

There are two main reasons that explain these results. The first is the lack of a covariance structured used in (3.5) and the second, the explosion of the kernel function around T which occurs faster as the parameter H decreases.

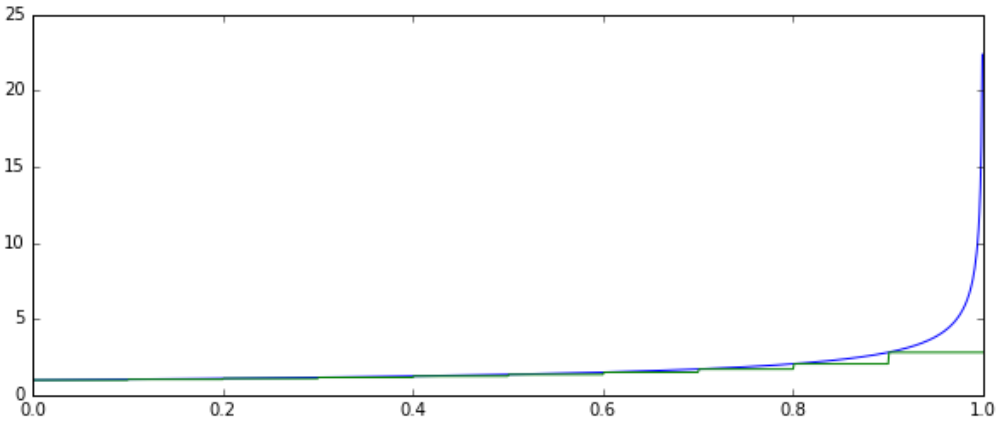


Figure 7: Step approximation of the kernel function $g(x) = (1-x)^{H-\frac{1}{2}}$ with 10 time steps and Hurst parameter $H = 0.05$

3.1.2 Implementaton of the Riemann sum method

```
%%cython --cplus --force
from libcpp.vector cimport vector
import numpy as np
cimport numpy as np
from scipy.special import gamma as Gamma

cdef extern from "math.h":
    double sqrt(double m)
    double exp(double m)
    double pow(double base, double exponent);

cdef double g(double x, double gamma):
    return pow(x, gamma)

cdef class naiveVol(object):

    cdef public double H, nu, gamma, T, alpha;
    cdef public double mean, var, cov;
```

```
19      cdef public N, M;
20      cpdef public vector[ vector[double] ] Y, V;
21
22      def __cinit__(self, H, nu, N, M):
23          self.N, self.M, self.H, self.nu = N, M, H, nu;
24          self.gamma = H - 0.5;
25          self.alpha = nu*sqrt(2.*H*Gamma(1.5 - H)/(Gamma(H + 0.5)*Gamma(2. - 2.*H)));
26
27      cpdef I(self, double T, vector[ vector[double] ] normalM,
28              vector[ vector[double] ] sigmaPaths):
29          self.T = T;
30          cdef double h = T/float(self.N);
31          cdef double ti;
32          cdef double tj;
33          cdef vector[ vector[double] ] YPaths;
34          YPaths.resize(self.M);
35
36          for m in range (0, self.M):
37              YPaths[m].resize(self.N + 1);
38              for i in range(1, self.N + 1):
39                  ti = i*h;
40                  for j in range (0, i):
41                      tj = j*h;
42                      YPaths[m][i] += sqrt(sigmaPaths[m][j])*g(ti - tj, self.gamma)
43                                      *sqrt(h)*normalM[m][j];
44
45                  YPaths[m][i]*= self.alpha;
46
47          self.Y = YPaths;
48
49      cpdef Vol(self, double theta):
50          cdef vector[ vector[double] ] VolPaths;
51          VolPaths.resize(self.M);
52
53          for m in range (0, self.M):
54              VolPaths[m].resize(self.N + 1);
55              for i in range(0, self.N + 1):
56                  VolPaths[m][i] = theta*exp(self.Y[m][i]);
57
58          self.V = VolPaths;
59
60      cpdef meanVarI(self):
61          cdef double mean = 0.;
62          cdef double var = 0.;
63          for m in range (0, self.M):
```

```
64              mean += self.Y[m][self.N];
65              var += self.Y[m][self.N]*self.Y[m][self.N];
66
67          mean /= float(self.M);
68          var /= float(self.M);
69          var -= mean;
70          self.mean = mean;
71          self.var = var;
```

3.1.3 Hybrid Scheme

As seen in section 3.1.1, the Riemann sum approximation of the integral

$$X_t = \int_0^t \sqrt{\Gamma_s} g(t,s) \mathrm{d}B_s$$

produces trajectories with a considerable error when estimating $\mathrm{Var}[X_T]$ for $g(t,s) = \alpha(t-s)^{H-\frac{1}{2}}$ and small Hurst parameter.

As it can be seen from the last figure, a step approximation of the function is not enough accurate around t, where the function $g(t,s)$ has the singularity.

To deal with this problem and obtain a faster and more general method than the exact simulation method of the *rBergomi* model given in [2], M. Bennedsen, A. Lunde and M. S. Pakkanen introduced in [4] a simulation scheme for Brownian semistationary processes by approximating the kernel function by a power function near the singularity. This approximation, which results in the so-called Hybrid Scheme, outperforms the Riemann sum approximation presented above. For this reason, the Hybrid Scheme will be presented here as it will be used to simulate the instantaneous volatility process (3.3) in some of the numerical experiments.

The Brownian semistationary process was first introduced by O. E. Barndorff-Nielsen and J. Schimegel in [5] as a stochastic process Y_t such that

$$Y_t = \mu + \int_{-\infty}^t \Sigma_s g(t,s) \mathrm{d}B_s + \int_{-\infty}^t a_s q(t,s) \mathrm{d}s$$

Where μ is constant, B is Brownian motion, $(\Sigma_t)_{t\in\mathbb{R}}$ and $(a_t)_{t\in\mathbb{R}}$ are $\{\mathcal{F}_t\}$-predictable processes with locally bounded trajectories and $g(t,s) = g(t-s)$, $q(t,s) = q(t-s) : (0,\infty) \to [0,\infty)$ are Borel measurable functions.

M. Bennedsen, A. Lunde and M. S. Pakkanen also present an extension of the scheme to truncated Brownian semistationary processes ($\mathcal{TBSS}$) which are the processes in which we will be interested.

It will be useful to recall the definition of this process.

Definition 3.2. A Truncated Brownian semistationary process Y_t is a stochastic process defined on a probability space $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\geq 0}, \mathbb{P})$ such that

$$Y_t = \int_0^t \Sigma_s g(t,s) \mathrm{d}B_s$$

Where B is Brownian motion , $(\Sigma_t)_{t\geq 0}$ is an $\{\mathcal{F}_t\}$-predictable process with locally bounded trajectories and $g(t,s) = g(t-s) : (0,\infty) \to [0,\infty)$ is a Borel measurable function.

This is basically a reduced form of a Brownian semistationary process where in additon the integral has been truncated.

In [4] is also assumed that $\int_o^\infty g(x)^2 \mathrm{d}x$ is finite so the stochastic integral is well defined and that the process $(\Sigma_t)_{t\geq 0}$ is covariance stationary with finite second moments, i.e. $\mathbb{E}[\Sigma_t^2] < \infty$ for all $t \geq 0$.

More assumptions are introduced in their work considering the kernel function $g(x)$ to ensure the roughness of the process Y_t and that the approximation of this function by a power function near zero is reasonable. With all this, the Hybrid Scheme is composed of two parts: a Riemann sum approximation where the kernel function has been approximated using a step function and a sum of Wiener integrals of the power function near t.

$$Y_n(t) = \sum_{j=1}^{min\{\lfloor nt \rfloor, k\}} \mathcal{L}_g\left(\frac{j}{n}\right) \Sigma_{t-\frac{j}{n}} \int_{t-\frac{j}{n}}^{t-\frac{j}{n}+\frac{1}{n}} (t-s)^\beta \mathrm{d}B_s + \sum_{j=k+1}^{\lfloor nt \rfloor} g\left(\frac{b_j}{n}\right) \Sigma_{t-\frac{j}{n}} \int_{t-\frac{j}{n}}^{t-\frac{j}{n}+\frac{1}{n}} \mathrm{d}B_s \tag{3.6}$$

The function $\mathcal{L}_g : (0,1] \to [0,\infty)$ is related to one of the assumptions that requires $g(x) = x^\gamma \mathcal{L}_g(x)$ with $\gamma \in (-1/2, 1/2) - \{0\}$ and $x \in (0,1]$. In addition, $\mathcal{L}_g$ is assumed to be continuously differentiable, slowly varying at 0 and bounded away from 0.

n, k are natural numbers determining the time-step discretization and the number of cells in which the kernel function is approximated by the power function near zero. $(b_j)_{j=k+1}^{\lfloor nt \rfloor}$ with $b_j \in [j-1, j]$ a sequence of numbers that defines the step function approximation.

Remark 3.3. By choosing $k = 0$, $b_j = t - \frac{j}{n}$ and $\Sigma_{t-\frac{j}{n}} = \sqrt{\Gamma_{t-\frac{j}{n}}}$ we recover the Riemann sum method (3.4) presented above.

3.1.4 Implementation of the Hybrid Scheme

Given n, k natural numbers, $T \geq 0$ and $\{\Sigma_i\}_{i=0}^{\lfloor nT \rfloor - 1}$ a discretized path of Σ, on a equidistant grid $\{0, \frac{1}{n}, \frac{2}{n}, ..., \frac{\lfloor nT \rfloor}{n}\}$ the method to simulate a path of the ($\mathcal{TBSS}$) proposed by M. Bennedsen, A. Lunde and M. S. Pakkanen is as follows:

- Generate independent samples $\mathbf{B}_i$ from the multivariate Gaussian distribution $\mathcal{N}(\mathbf{0}, C)$ for $i = 0, 1, ..., \lfloor nT \rfloor - 1$ where C is a $(k+1) \times (k+1)$ covariance matrix defined as

$$\begin{aligned} C_{1,1} &= \frac{1}{n} \\ C_{1,j} &= \frac{(j-1)^{\gamma+1} - (j-2)^{\gamma+1}}{(\gamma+1)n^{\gamma+1}}, \ j = 2, ..., k+1 \\ C_{j,j} &= \frac{(j-1)^{2\gamma+1} - (j-2)^{2\gamma+1}}{(2\gamma+1)n^{2\gamma+1}}, \ j = 2, ..., k+1 \end{aligned}$$

$$C_{j,l} = \frac{(j-1)^{\gamma+1} - (l-1)^{\gamma}}{(\gamma+1)n^{2\gamma+1}} \; {}_2F_1(-\gamma, 1, \gamma+2, \frac{j-1}{l-1}) \\ - \frac{(j-2)^{\gamma+1} - (l-2)^{\gamma}}{(\gamma+1)n^{2\gamma+1}} \; {}_2F_1(-\gamma, 1, \gamma+2, \frac{j-2}{l-2}) \\ j, l = 2, ..., k+1, \; j < l$$

- Compute $Y\left(\frac{i}{n}\right)$ using the equation

$$Y\left(\frac{i}{n}\right) = \sum_{j=1}^{min\{i,k\}} \mathcal{L}_g\left(\frac{j}{n}\right) \Sigma_{i-j} B_{i-j,j} + \sum_{j=k+1}^{i} g\left(\frac{b_j^*}{n}\right) \Sigma_{i-j} B_{i-j} \tag{3.7}$$

Finally, as in the previous case, simulating the instantaneous volatility process would simply consist in simulating first the ($\mathcal{TBSS}$) and on the same grid compute σ_{t_i} as

$$\sigma_{t_i} = \theta(t_i) \; \exp(\; Y_{t_i}) \quad t_i = \frac{i}{n}$$

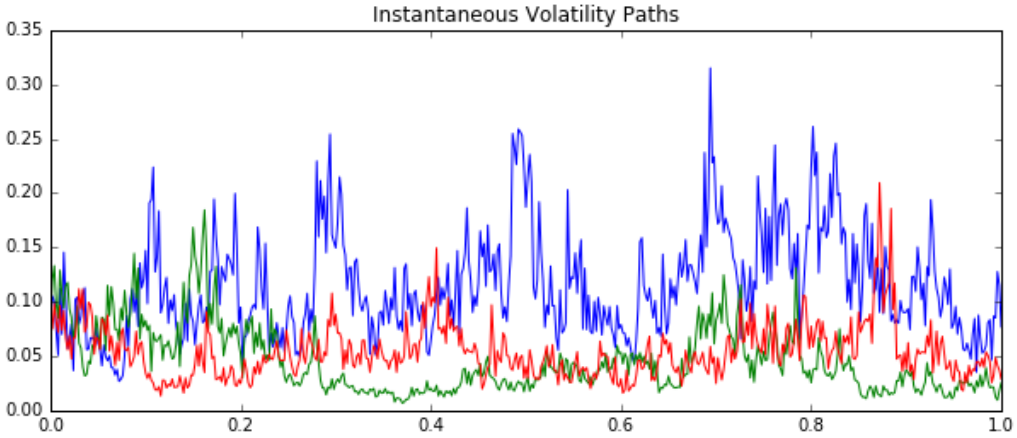


Figure 8: 3 simulated paths of the instantaneous volatility process generated using the Hybrid Scheme with $n = 500$ and $k = 1$. The Γ process has been set to 1 for all $(\omega, t) \in \Omega \times \mathbb{R}_{t \geq 0}$. Paths computed with 500 time steps, T = 1.0, $\sigma_0 = 0.085$ and H= 0.21.

From now on, when using the Hybrid Scheme in this work, the sequence $(b_j)_{j=k+1}^{\lfloor nt \rfloor}$ will be the optimal sequence from ([4], Proposition 2.8, pag 10).

$$b*_j = \left(\frac{j^{H+\frac{1}{2}} - (j-1)^{H+\frac{1}{2}}}{H+\frac{1}{2}} \right)^{\frac{1}{H-\frac{1}{2}}}$$

.

In addition, notice that the assumptions presented in [4] are meant for a more general framework that the one we are interested in here so from now on we will consider

- $\Sigma_t = \sqrt{\Gamma_t}$
- $g(t,s) = \alpha(t-s)^{H-\frac{1}{2}}$ so we have $\gamma = H - \frac{1}{2}$ and $\mathcal{L}_g(x) \equiv \alpha$.
- $\alpha = \nu\sqrt{\frac{2H\Gamma(3/2-H)}{\Gamma(H+1/2)\Gamma(2-2H)}}$

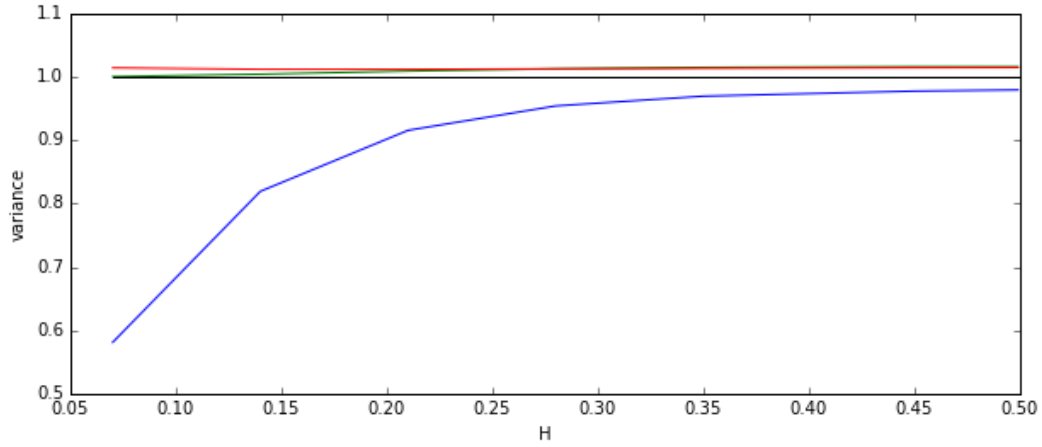


Figure 9: $Var[X_T]$ with $T = 1$. computed using Monte Carlo simulations with 300 time steps and 10.000 trajectories with two methods: Riemann sum method in blue and Hybrid Scheme with $k = 1$ in green and $k = 2$ in red.

The figure above shows the advantage of using the Hybrid Scheme over the Riemann sum method or the exact simulation proposed in [2] in the *rBergomi* model.

Implementation

```
%%cython --cplus --force
from libcpp.vector cimport vector
from math import floor
import numpy as np
cimport numpy as np
from scipy.special import gamma as Gamma
from scipy.special import hyp2f1 as F1

cdef extern from "math.h":
    double sqrt(double m)
    double exp(double m)
    double pow(double base, double exponent);
```

```
13
14 cdef double mx(double a, double b):
15     if (a > b):
16         return a
17     else:
18         return b
19
20 cdef double g(double x, double H):
21     return pow(x, H - 0.5)
22
23 cdef class HybridScheme(object):
24
25     cdef public double H, nu, gamma, T, alpha, theta, S0;
26     cdef public double mean, var, covij;
27     cdef public int k, n, N, M;
28     cpdef public vector[ vector[double] ] Y, V, S;
29     cdef public vector[ double ] b, call;
30     cpdef public cov, L;
31
32     def __cinit__(self, H, nu, k, n, M, T):
33         self.k, self.n, self.M, self.H, self.nu, self.T = k, n, M, H, nu, T;
34         self.gamma = H - 0.5;
35         self.N = floor(T*n);
36         self.alpha = nu*sqrt(2.*H*Gamma(1.5 - H)/(Gamma(H + 0.5)*Gamma(2. - 2.*H)));
37
38     cpdef covariance(self):
39         cdef double g1 = self.gamma + 1.;
40         cdef double g2 = self.gamma + 2.;
41         cdef double g3 = 2.*self.gamma + 1.;
42         cdef vector[double] b;
43         b.resize(self.N - self.k)
44         E = np.ndarray(shape = (self.k + 1, self.k + 1));
45
46         E[0][0] = 1./float(self.n);
47         for j in range (2, self.k + 2):
48             E[0][j-1] = (pow(j - 1., g1) - pow(j - 2., g1))/(g1*pow(self.n, g1));
49             E[j-1][0] = E[0][j-1];
50             E[j-1][j-1] = (pow(j - 1., g3) - pow(j - 2., g3))/(g3*pow(self.n, g3));
51             for t in range(j + 1, self.k + 2):
52                 E[j-1][t-1] = pow(j - 1., g1)*pow(t - 1., self.gamma)
53                 *F1(-self.gamma, 1., g2, (j - 1.)/(t - 1.)) - pow(j - 2., g1)
54                 *pow(t - 2., self.gamma)*F1(-self.gamma, 1., g2, (j - 2.)/(t - 2.));
55                 E[j-1][t-1] /= (g1*pow(self.n, g3));
56                 E[t-1][j-1] = E[j-1][t-1];
57
```

```
58          self.cov = E;
59          self.L = np.linalg.cholesky(E);
60
61          for l in range (self.k + 1, self.N + 1):
62              b[l - (self.k + 1)] = pow((pow(l, g1) - pow(l - 1., g1))/g1,
63                                        1./self.gamma);
64
65          self.b = b;
66
67      cpdef TBSS(self, normalM, vector[ vector[double] ] sigmaPaths):
68          cdef vector[ vector[double] ] YPaths;
69          YPaths.resize(self.M);
70          cdef vector[ vector[double] ] W;
71          W.resize(self.N);
72          WAux = np.zeros(self.k + 1);
73
74          for m in range(self.M):
75              YPaths[m].resize(self.N + 1);
76              for i in range(self.N):
77                  for j in range(self.k + 1):
78                      WAux[j] = normalM[i][m + j*self.M];
79                  W[i].resize(self.k + 1);
80                  W[i] = self.L.dot(WAux);
81
82              for i in range(1, self.k + 1): #i<=k
83                  for j in range(1, i + 1):
84                      YPaths[m][i] += sqrt(sigmaPaths[m][i - j])*W[i - j][j]
85                  YPaths[m][i] *= self.alpha;
86
87              for i in range(self.k + 1, self.N + 1): #i>k
88                  for j in range(1, self.k + 1):
89                      YPaths[m][i] += sqrt(sigmaPaths[m][i - j])*W[i - j][j]
90                  for j in range(self.k + 1, i + 1):
91                      YPaths[m][i] += g(self.b[j - (self.k + 1)]/float(self.n), self.H)
92                                      *sqrt(sigmaPaths[m][i - j])*W[i - j][0];
93                  YPaths[m][i] *= self.alpha;
94
95          self.Y = YPaths;
96
97      cpdef Vol(self, double theta):
98          self.theta = theta;
99          cdef vector[ vector[double] ] VolPaths;
100         VolPaths.resize(self.M);
101
102         for m in range (0, self.M):
```

```
103             VolPaths[m].resize(self.N + 1);
104             for i in range(0, self.N + 1):
105                 VolPaths[m][i] = theta*exp(self.Y[m][i]);
106 
107         self.V = VolPaths;
108 
109     cpdef Stock(self, double S0, vector[ vector[double] ] normalM):
110         self.S0 = S0;
111         h = self.T/float(self.N);
112         cdef vector[ vector[double] ] SPaths;
113         SPaths.resize(self.M);
114 
115         for m in range (0, self.M):
116             SPaths[m].resize(self.N + 1);
117             SPaths[m][0] = S0;
118             for i in range(1, self.N + 1):
119                 SPaths[m][i] = SPaths[m][i - 1]*exp( self.V[m][i - 1]*sqrt(h)
120                                 *normalM[i - 1][m]
121                                 - 0.5*h*self.V[m][i - 1]*self.V[m][i - 1] );
122 
123             self.S = SPaths;
124 
125     cpdef meanVarTBSS(self):
126         cdef double mean = 0.;
127         cdef double var = 0.;
128         for m in range (0, self.M):
129             mean += self.Y[m][self.N];
130             var += self.Y[m][self.N]*self.Y[m][self.N];
131 
132         mean /= float(self.M);
133         var /= float(self.M);
134         var -= mean;
135         self.mean = mean;
136         self.var = var;
```

The `HybridScheme` class first needs to be initialized by providing the parameters H, ν, k, n, M, T where M is the number of paths wanted and the rest of parameters are as presented above.

```
Hs = HybridScheme(H, nu, k, n, M)
```

Then the covariance matrix C need to be computed before generating any ($\mathcal{TBSS}$) path. This is done by calling the member function: `covariance()`

```
Hs.covariance()
```

Finally M ($\mathcal{TBSS}$) paths are generated using the equation (3.7) by calling

```
Hs.TBSS(normalM, sigmaPaths)
```

Here `sigmaPaths` should be a container with M distretized paths $\{\Sigma_i\}_{i=0}^{N}$ with `N = Hs.N`. `normalM` should be a container of `size = (Hs.N, Hs.M*(Hs.k + 1))` with independent samples drawn from a standard normal distribution.

The main advantage of having this as an input is that the randomness can be fixed. This is desirable when calibrating where we want a change in the value only produced by changes in the parameters and not in the randomness. Otherwise, the random seed should be prefixed but having `normalM` as an input also allows to generate it only once.

3.2 Stock Price model under Modulated Stochastic Volatility processes

Let $(S_t)_{t\geq 0}$ and $(\sigma_t)_{t\geq 0}$ be stochastic processes defined on a probability space $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\geq 0}, \mathbb{Q})$ with dynamics

$$\begin{aligned} \mathrm{d}S_t &= S_t \sigma_t \mathrm{d}W_t \\ \sigma_t &= \theta(t)\ \exp(\int_0^t \sqrt{\Gamma_s} g(t,s)\mathrm{d}B_s) \end{aligned} \tag{3.8}$$

where B and W are standard Brownian motions with correlation ρ, i.e. $\mathrm{d}W_t \mathrm{d}B_t = \rho \mathrm{d}t$.

The process $(\Gamma_s)_{s\leq t}$ and the deterministic functions $\theta(t)$ and $g(s,t)$ are as presented previously in section 3.

Remark 3.4. Being the model specified under the risk neutral probability measure we could find differences between the parameters obtained from calibration (pricing) under this probability space and the ones obtained from statistical estimations (physical measure).

The dynamics of the stock price process are proposed here without a drift term to avoid dealing with the change of measure to obtain a truly martingale price process.

However, it does not seem obvious how to prove that $(S_t)_{t\geq 0}$ is actually a martingale.

First, notice that under this dynamics the price process becomes the stochastic exponential of $\int_0^t \sigma_s \mathrm{d}W_s$. This is easy to check by defining the process $Y_t := log(S_t)$ and applying Itô's lemma

$$\mathrm{d}Y_t = \frac{1}{S_t}\mathrm{d}S_t - \frac{1}{2}\frac{1}{S_t^2}(\mathrm{d}S_t)^2 = \sigma_t \mathrm{d}W_t - \frac{1}{2}\sigma_t^2 \mathrm{d}t$$

then integrating from 0 to t

$$log(S_t) - log(S_0) = \int_0^t \sigma_s \mathrm{d}W_s - \frac{1}{2}\int_0^t \sigma_s^2 \mathrm{d}s$$

and finally by exponentiating both sides of the equality

$$S_t = S_0 \exp(\int_0^t \sigma_s \mathrm{d}W_s - \frac{1}{2}\int_0^t \sigma_s^2 \mathrm{d}s) \tag{3.9}$$

$$\tag{3.10}$$

Being the volatility process $(\sigma_t)_{t\geq 0}$ a Wiener integral of a deterministic function, the instantaneous volatility process becomes a left continuous $\{\mathcal{F}_t\}$-adapted process. This allow us to conclude that $(S_t)_{t\geq 0}$ is a local martingale.

There are several ways of checking that a local martingale is a martingale. In this case, having $S_t = \mathcal{E}(\sigma_t)$ where $\mathcal{E}(\cdot)$ is the stochastic exponential we may be tempted to use Noviko's condition which is a sufficient condition for $\mathcal{E}(\sigma_t)$ to be a martingale. The condition is

$$\mathbb{E}[\exp(\frac{1}{2}\int_0^t \sigma_s^2 \mathrm{d}s)] < \infty$$

However, the process $(\Gamma_t)_{0\leq t}$ that modulates the volatility process does not help with this computations and actually this does not seem a straightforward condition to check.

3.2.1 Simulation of the Stock Price process

The simulation of the Stock price process is straightforward having already a path $\{\sigma_{t_i}\}_{i=0}^N$ of the instantaneous volatility process.

It can be derived similarly as in section 2 by splitting the integrals into sums and taking left hand side approximations

$$\begin{aligned}
S_{t_i} &= S_{t_0}\ \exp(\int_0^{t_i}\sigma_s \mathrm{d}W_s - \frac{1}{2}\int_0^{t_i}\sigma_s^2\mathrm{d}s) = S_{t_0}\ \exp(\sum_{k=0}^{i-1}\int_{t_k}^{t_{k+1}}\sigma_s \mathrm{d}W_s - \frac{1}{2}\sum_{k=0}^{i-1}\int_{t_k}^{t_{k+1}}\sigma_s^2\mathrm{d}s) \\
&= S_{t_0}\ \exp(\sum_{k=0}^{i-2}\int_{t_k}^{t_{k+1}}\sigma_s \mathrm{d}W_s - \frac{1}{2}\sum_{k=0}^{i-2}\int_{t_k}^{t_{k+1}}\sigma_s^2\mathrm{d}s)\ \exp(\int_{t_{i-1}}^{t_i}\sigma_s \mathrm{d}W_s - \frac{1}{2}\int_{t_{i-1}}^{t_i}\sigma_s^2\mathrm{d}s) \\
&= S_{t_{i-1}}\exp(\int_{t_{i-1}}^{t_i}\sigma_s \mathrm{d}W_s - \frac{1}{2}\int_{t_{i-1}}^{t_i}\sigma_s^2\mathrm{d}s) \approx S_{t_{i-1}}\exp(\sigma_{t_{i-1}}\int_{t_{i-1}}^{t_i}\mathrm{d}W_s - \frac{1}{2}\sigma_{t_{i-1}}^2\int_{t_{i-1}}^{t_i}\mathrm{d}s) \\
&= S_{t_{i-1}}\exp(\sigma_{t_{i-1}}(W_{t_i} - W_{t_{i-1}}) - \frac{1}{2}\sigma_{t_{i-1}}^2 h)
\end{aligned}$$

Recall that the Brownian motions B and W driving the instantaneous volatility and the stock price processes are correlated with parameter ρ. Notice that W can be defined as $W_t := \rho B_t + \sqrt{1-\rho^2}B^\perp$ where B and $B^\perp$ are independent Brownian motions. This is a consequence of the Lévy Characterization of Brownian motion, since W is a continuous martingale with $W_0 = 0$ and $(\mathrm{d}W_t)^2 = \mathrm{d}t$.

In practice we will sample from a standard normal random variable and use the property that Brownian motion has normally distributed increments with mean 0 and variance given by the time difference.

$$W_{t_i} - W_{t_{i-1}} = \sqrt{t_i - t_{i-1}}Z \quad Z \sim \mathcal{N}(0,1)$$

This means that we only need independent samples from a standard normal variable.

To generate a stock price path we need to

- Choose a discretization of the interval $[0,T]$. Here a equidistant grid $\{t_0, t_1, t_2, ..., t_N\}$ is chosen with $t_i = ih$, $i = 0, 1, 2, ..., N$
- Generate a path $\{\Gamma_{t_i}\}_{i=0}^N$ of the process Γ that modulates the stochastic integral of the instantaneous volatility process.
- Generate paths $\{B_{t_i}\}_{i=1}^N$ (or $\{\mathbf{B}_{t_i}\}_{i=1}^N$ if the Hybrid Scheme is chosen) and $\{W_{t_i}\}_{i=1}^N$.

  In case of using the Riemann sum method to generate the instantaneous volatility process we could proceed by simulating two independent multivariate standard normal vectors $\mathbf{B}$, $\mathbf{B}^\perp$ and set $Z = \rho\mathbf{B} + \sqrt{1-\rho^2}\mathbf{B}^\perp$.
- Generate a path $\{\sigma_{t_i}\}_{i=0}^N$ of the instantaneous volatility process by using either the Riemann sum method or the Hybrid scheme presented above.

- Finally use the following equation to generate a path $\{S_{t_i}\}_{i=0}^{N}$ of the stock price

$$S_{t_i} = S_{t_{i-1}} \exp(\sigma_{t_{i-1}}(W_{t_i} - W_{t_{i-1}}) - \frac{1}{2}\sigma_{t_{i-1}}^2 h)$$

3.2.2 Implementation

To complete the implementations presented above only an extra member function is needed for the stock price.

```
cpdef Stock(self, double S0, vector[ vector[double] ] normalM):
        self.S0 = S0;
        h = self.T/float(self.N);
        cdef vector[ vector[double] ] SPaths;
        SPaths.resize(self.M);

        for m in range (0, self.M):
            SPaths[m].resize(self.N + 1);
            SPaths[m][0] = S0;
            for i in range(1, self.N + 1):
                SPaths[m][i] = SPaths[m][i - 1]*exp( self.V[m][i - 1]*sqrt(h)
                               *normalM[i - 1][m]
                               - 0.5*h*self.V[m][i - 1]*self.V[m][i - 1] );

            self.S = SPaths;
```

The following class will be helpful to generate the multivariate standard normal samples.

```
1   class normalGen(object):
2
3   def __init__(self, k, Nrows, Ncolumns):
4       self.k, self.N, self.M = k, Nrows, Ncolumns
5
6   def vol(self):
7       self.B = np.random.normal(size=(self.N, self.M*(self.k + 1)))
8
9   def stockAndVol(self, rho):
10      self.rho = rho
11      self.B = np.random.normal(size=(self.N, self.M*(self.k + 1)))
12      Bbar = np.random.normal(size=(self.N, self.M))
13      self.W = rho*self.B[:, :self.M] + np.sqrt(1. - rho*rho)*Bbar
```

The `normalGen` class first needs to be initialized by providing the parameters k, `Nrows`, `Ncolumns`.

```
normalMatrix = normalGen(k, Nrows, Ncolumns)
```

Then, if we only want to generate a vol path the following member function will give us the normal samples wanted.

```
normalMatrix.vol()
```

On the other hand, if we want correlated samples to generate instantaneous volatility and stock price paths, we can call

```
normalMatrix.stockAndVol(rho)
```

The results are stored as member data, `normalMatrix.B` and `normalMatrix.W` for the volatility and the stock paths respectively.

This class supports both the Riemann sum method ($k = 0$) and the Hybrid Scheme.

3.3 Calibration to SPX smiles

As seen in section 2, fractional stochastic volatility models such as the rBergomi model show a stronger consistency with the SPX volatility surface than stochastic volatility models driven by a standard Brownian motion process.

In the case of the set of Modulated Volterra stochastic processeses it is expected that the consistency with the SPX volatility surface should be as good as the one presented by the rough Bergomi model.

In this section we will calibrate the model to SPX smiles using the dynamics (3.8) presented above. This is done by calibrating the parameters of the model to match the prices of liquid derivatives on the SPX Index. Usually European Call and Put options are used for calibration.

Here European Call option prices are chosen to calibrate the model. As the reader probably knows already, European Call options are financial contracts that gives the holder of the option the right but not the obligation to buy an underlying at maturity T for a certain price K previously agreed

Assuming that there is no discounting factor and that we are on a probability space $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\geq 0}, \mathbb{Q})$ where the stock price process is a martingale, by the martingale pricing theory, the price C_T of a European Call option with maturity T and strike K is

$$C_T = \mathbb{E}_0[(S_T - K)_+]$$

Recall that the dynamics (3.8) were presented with the intention to start under this assumptions.

Therefore, the pricing of this financial contract can be easily obtain by simulating multiple paths of the stock process as already explained and using the estimation

$$\mathbb{E}_0[(S_T - K)_+] \approx \frac{1}{M}\sum_{m=1}^{M}(S_T^m - K)_+$$

where S_T^m is the value at T of the m path simulated of the stock process.

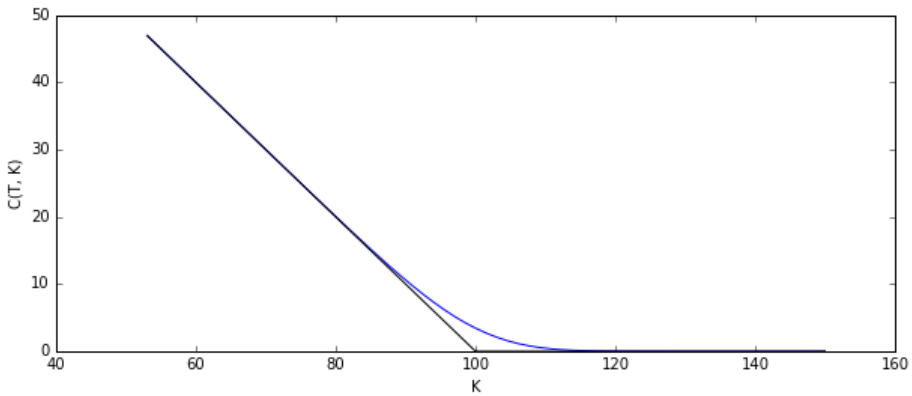


Figure 10: Consistency test based in the property that the European Call option price is decreasing and convex as a function of the strike. Simulation based in 10000 replications, 500 time steps and parameters: $T = 1.0$, $S_0 = 100.0$, $\sigma_0 = 0.085$, $H = 0.1$, $\nu = 1.0$, $\rho = -0.8$. Γ is the Lévy-driven Ornstein-Uhlenbek process as in 3.3.3 with parameters: $\Gamma_0 = 0.012$, $\lambda = 0.01$, $A = 5.82$, $a = 19.82$.

3.3.1 Monte Carlo Implementation

Once the paths have been generated and stored it is very easy to obtain the Call option price as explained above. Then, it will be enough to add the following member function to the classes introduced before

```
cpdef Call(self, vector[double] K):
        N_K = K.size();
        cdef vector[double] call;
        call.resize(N_K);

        for m in range(self.M):
            for l in range(N_K):
                call[l] += mx(self.S[m][self.N] - K[l], 0.);

        for l in range(N_K):
            call[l] /= float(self.M);

        self.call = call
```

Nevertheless, when using this implementation we are performing unnecessary computations as well as storaging values of the paths that are no longer needed after their pay-off evaluation. This slows down the pricing significantly which is something someone would not want when calibrating a model.

For this reason either the following implementation based in the Hybrid Scheme or the one based in the Riemann sums (A.3) will be used.

```
cdef class CallHS(object):

    cdef public double H, nu, T, vol0, S0;
    cdef public double alpha, gamma, h;
    cdef public int k, n, N, M;
    cdef public vector[ double ] b, call;
    cpdef public cov, L;

    def __cinit__(self, H, nu, k, n, M, T, vol0, S0):
        self.k, self.n, self.M = k, n, M;
        self.H, self.nu, self.T, self.vol0, self.S0 = H, nu, T, vol0, S0;
        self.gamma = H - 0.5;
        self.alpha = nu*sqrt(2.*H*Gamma(1.5 - H)/(Gamma(H + 0.5)*Gamma(2. - 2.*H)));
        self.N = floor(T*n);
        self.h = self.T/float(self.N);
```

```
16
17      cpdef covariance(self):
18          cdef double g1 = self.gamma + 1.;
19          cdef double g2 = self.gamma + 2.;
20          cdef double g3 = 2.*self.gamma + 1.;
21          cdef vector[double] b;
22          b.resize(self.N - self.k)
23          E = np.ndarray(shape = (self.k + 1, self.k + 1));
24
25          E[0][0] = 1./float(self.n);
26          for j in range (2, self.k + 2):
27              E[0][j-1] = (pow(j - 1., g1) - pow(j - 2., g1))/(g1*pow(self.n, g1));
28              E[j-1][0] = E[0][j-1];
29              E[j-1][j-1] = (pow(j - 1., g3) - pow(j - 2., g3))/(g3*pow(self.n, g3));
30              for t in range(j + 1, self.k + 2):
31                  E[j-1][t-1] = pow(j - 1., g1)*pow(t - 1., self.gamma)
32                  *F1(-self.gamma, 1., g2, (j - 1.)/(t - 1.)) - pow(j - 2., g1)
33                  *pow(t - 2., self.gamma)*F1(-self.gamma, 1., g2, (j - 2.)/(t - 2.));
34                  E[j-1][t-1] /= (g1*pow(self.n, g3));
35                  E[t-1][j-1] = E[j-1][t-1];
36
37          self.cov = E;
38          self.L = np.linalg.cholesky(E);
39
40          for l in range (self.k + 1, self.N + 1):
41              b[l - (self.k + 1)] = pow((pow(l, g1) - pow(l - 1., g1))/g1, 1./self.gamma);
42
43          self.b = b;
44
45      cpdef Call(self, vector[double] K, vector[ vector[double] ] normalVol,
46                 vector[ vector[double] ] normalStock,
47                 vector[ vector[double] ] sigmaPaths):
48          cdef int N_K = K.size();
49          cdef vector[double] call;
50          call.resize(N_K);
51          cdef vector[ vector[double] ] W;
52          W.resize(self.N);
53          WAux = np.zeros(self.k + 1);
54          cdef double Y = 0.;
55          cdef double volti;
56          cdef double StockExpSum;
57          cdef double ST;
58
59
60          for m in range(self.M):
```

```
61              for i in range(self.N):
62                  for j in range(self.k + 1):
63                      WAux[j] = normalVol[i][m + j*self.M];
64                  W[i].resize(self.k + 1);
65                  W[i] = self.L.dot(WAux);
66
67              StockExpSum = self.vol0*sqrt(self.h)*normalStock[0][m]
68                          - 0.5*self.h*self.vol0*self.vol0;
69              for i in range(1, self.k + 1): #i<=k
70                  for j in range(1, i + 1):
71                      Y += sqrt(sigmaPaths[m][i - j])*W[i - j][j]
72                  volti = self.vol0*exp(self.alpha*Y);
73                  StockExpSum += volti*sqrt(self.h)*normalStock[i][m]
74                              - 0.5*self.h*volti*volti;
75                  Y = 0.;
76
77              for i in range(self.k + 1, self.N): #i>k
78                  for j in range(1, self.k + 1):
79                      Y += sqrt(sigmaPaths[m][i - j])*W[i - j][j]
80                  for j in range(self.k + 1, i + 1):
81                      Y += g(self.b[j - (self.k + 1)]/float(self.n), self.H)
82                           *sqrt(sigmaPaths[m][i - j])*W[i - j][0];
83                  volti = self.vol0*exp(self.alpha*Y);
84                  StockExpSum += volti*sqrt(self.h)*normalStock[i][m]
85                              - 0.5*self.h*volti*volti;
86                  Y = 0.;
87
88              ST = self.S0*exp(StockExpSum);
89
90              for l in range(N_K):
91                  call[l] += mx(ST - K[l], 0.);
92
93          for l in range(N_K):
94              call[l] /= float(self.M);
95
96          self.call = call;
```

3.3.2 Control Variate

As seen in the previous section, pricing Call options via Monte Carlo replications under this model is straightforward. However, notice that the simulation of three processes are involved which turns out in a computationally expensive model.

For this reason, two control variate methods are presented and tested here with the intention to decrease significantly the number of replications needed to price.

Control variate methods have been proved to be one of the most effective variance reduction techniques. This is obviously the goal, by decreasing variance we will achieve a more accurate estimation with the same number of replications.

In this particular case, $V = (S_T - K)_+$ is the random variable that we want to estimate. The estimator used here is the sample mean $V \approx \frac{1}{M}\sum_{m=1}^{M} V_m$ as shown above where $V_1, V_2, ..., V_m$ are M samples drawn from V.

The idea behind the control variate method is to define a new random variable with same mean as the one that we are interested but lower variance. To illustrate the idea, below is presented a standard derivation of how this can be done. There are multiples references about this method, a complete an extended discussion of the problem can be found in [6].

Lets take $b \in \mathbb{R}$ and any random variable X. We can define the following

$$C := V - b(X - \mathbb{E}[X])$$

This new random variable has the same mean as the original V

$$\mathbb{E}[C] = \mathbb{E}[V] - \mathbb{E}[b(X - \mathbb{E}[X])] = \mathbb{E}[V] - b(\mathbb{E}[X] - \mathbb{E}[X]) = \mathbb{E}[V]$$

and its variance can be computed as

$$\begin{aligned} Var_b[C] = \; & Var[V - b(X - \mathbb{E}[X])] = Var[V] + b^2 Var[X - \mathbb{E}[X]] - 2bCov[V, X - \mathbb{E}[X]] \\ = \; & Var[V] + b^2 Var[X] - 2bCov[V, X] = Var[V] + b^2 Var[X] - 2b\rho_{V,X}\sqrt{Var[V]Var[X]} \end{aligned}$$

Now notice that the equation $Var_b[C]$ defines a parabola as a function of the parameter b. Moreover, this parabola is opening to the top since the term multiplying b^2 is $Var[X]$ which is always positive. Therefore, the function has a minimum which can be obtained by using standard calculus derivatives. The minimum is attained at

$$b^* = \frac{\rho_{V,X}\sqrt{Var[V]Var[X]}}{Var[X]} = \frac{Cov[V, X]}{Var[X]}$$

As mentioned in [6] b^* is not usually known and in practice it can be estimated with the following estimator

$$\hat{b}_M = \frac{\sum_{i=1}^{M}(X_i - \overline{X})(V_i - \overline{V})}{\sum_{i=1}^{M}(X_i - \overline{X})^2}$$

Be aware that b^* is the value for which the function $Var_b[C]$ attained its minimum but it does not ensure $Var_{b^*}[C]$ to be smaller than $Var[V]$. By equating $Var_{b^*}[C] < Var[V]$ we obtain the condition

$$b^2\sqrt{Var[X]} < 2b\rho_{V,X}\sqrt{Var[V]} \tag{3.11}$$

Then, to find and effective control variate, the random variable X has to be carefully chosen so it satisfies condition(3.11)

To be able to asses and compare the performance of different random variables X as control variate we will use the variance ratio

$$R_{V,X} = \frac{Var[V - b^*(X - \mathbb{E}[X])}{Var[V]}$$

Substituting b^* in this equation we obtain

$$R_{V,X} = 1 - \rho^2_{V,X}.$$

This equality tell us that the larger the correlation between V and X the more variance reduction we will obtain.

Numerical experiment

We now present the performance of two control variates for multiple strikes. In both cases Γ is a Lévy-driven Ornstein-Uhlenbek as in process 3.3.3 and the instantaneous volatility process is simulated using the method of Riemann sums.

To obtain meaningful results the parameters have been chosen based in real data after calibrating the model to European Call option on the SPX Index prices on the fourteenth of May 2014 without control variates.

The first control variate considered is $X = S_T$ which is widely used in derivative pricing. As expected the correlation between X and V turns out to be stronger for lower strikes

As explained before, the method of control variates is based in sampling from a new random variable C to be defined knowing the expectation of the control variate X. Notice that under the dynamics (3.8) the stock price process is a martingale so $\mathbb{E}[S_T] = S_0$.

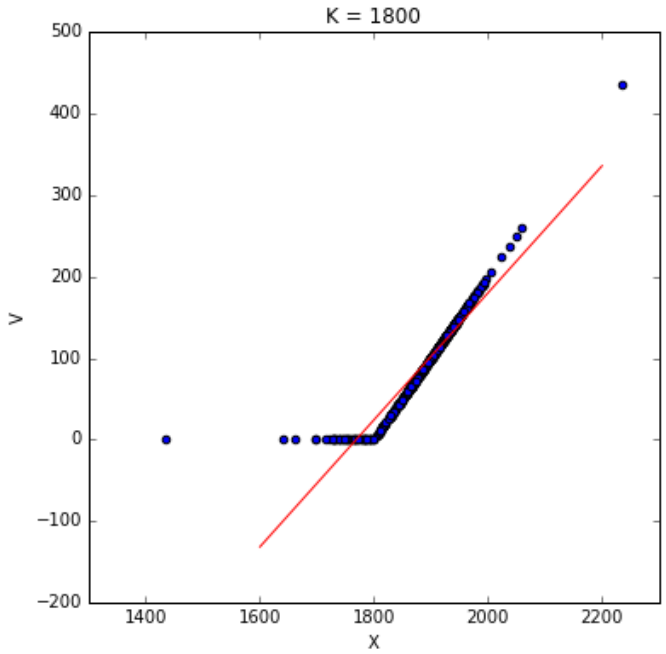


Figure 11: Scatter plot of the sample pairs (X_i, V_i) for strike $K = 1800$. Regression line in red. Sample correlation = 0.94.

Variance ratio obtained across strikes from K = 1700 to 2000.

$X = S_T$								
K	1700	1750	1800	1850	1900	1950	2000	2050
$R_{V,X}$	0.08289	0.11593	0.17382	0.28837	0.51096	0.81483	0.93369	0.94880

The second control variate considered in this experiment is $X = (\tilde{S}_T - K)_+$ where $(\tilde{S}_t)_{t\geq 0}$ is given by the dynamics

$$S_t = S_0 \exp(\int_0^t \sigma_0 \mathrm{d}W_s - \frac{1}{2}\int_0^t \sigma_0^2 \mathrm{d}s) = S_0 \exp(\sigma_0 W_t - \frac{1}{2}\sigma_0^2 t)$$

The use of this control variate is motivated by the fact that the volatility of volatility might be small. In that case, considering that the instantaneous volatility will not move too much from its starting value is a sensible assumption. However, we can see that a significant variance reduction can be achieved even for a volatility of volatility not as small as expected.

In this case we are in the framework of the Black-Scholes model without drift so, assuming no interest rates,

$$\mathbb{E}[(\tilde{S}_T - K)_+] = \Phi(d_+)S_0 - \Phi(d_-)K$$

where, as usual,

$$d_+ = \frac{log(S_0/K) + \frac{1}{2}\sigma^2 T}{\sigma^2\sqrt{T}} \quad d_- = d_+ - \sigma\sqrt{T}$$

with Φ the cumulative distribution of the standard normal distribution.

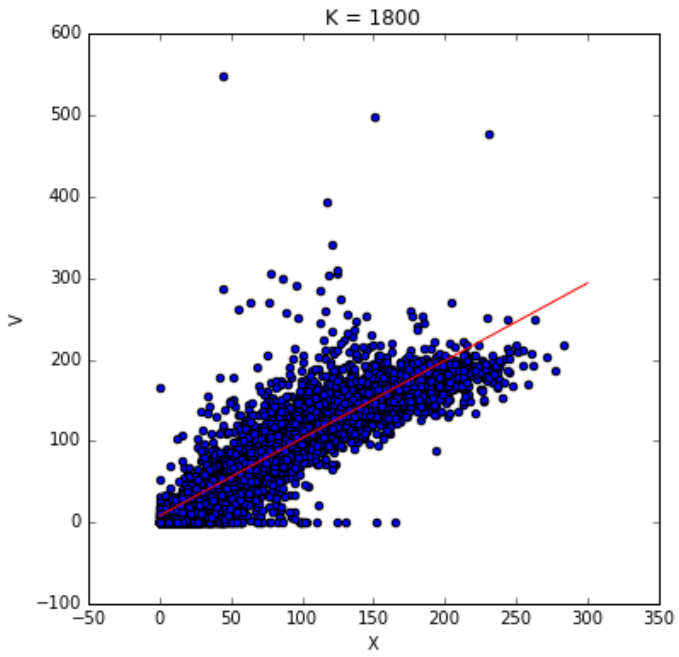


Figure 12: Scatter plot of the sample pairs (X_i, V_i) for strike $K = 1800$ and regression line. Sample correlation = 0.90.

$X = (\tilde{S}_T - K)_+$								
K	1700	1750	1800	1850	1900	1950	2000	2050
$R_{V,X}$	0.22991	0.22105	0.22601	0.28160	0.48424	0.87586	0.99804	1.00000

The next table shows the parameters used.

M	N	T	S_0	σ_0	ρ
10000	100	38./365.25	1888.53	0.085	-0.736
H	ν	x	λ	A	a
0.0437	0.9984	0.012	0.01	5.82	19.82

where x, λ, A, a defined the process Γ as explained in 3.3.3.

Notice that the first control variate generally outperforms the other. For this reason and for being computationally less expensive, we will used the control variate $S = S_T$ to calibrate SPX smiles.

Implementation of the Control Variate method

The following is a member function of the class `CallHS`.

```
cpdef CallCV(self, vector[double] K, vector[ vector[double] ] normalVol,
             vector[ vector[double] ] normalStock, vector[ vector[double] ] sigmaPaths):
        cdef vector[ vector[double] ] W;
        W.resize(self.N);
        WAux = np.zeros(self.k + 1);
        cdef double I = 0.;
        cdef double volti;
        cdef double StockExpSum;
        cdef double ST;
        cdef int N_K = K.size();
        cdef vector[vector[double]] X;
        X.resize(self.M);
        cdef vector[double] XMean_M;
        XMean_M.resize(N_K);
        cdef vector[vector[double]] Y;
        Y.resize(self.M);
        cdef vector[double] YMean;
        YMean.resize(N_K);
        cdef vector[double] b;
        cdef vector[double] bAux;
        b.resize(N_K);
        bAux.resize(N_K);
        cdef vector[double] call;
        call.resize(N_K);

        for m in range(self.M):
            for i in range(self.N):
                for j in range(self.k + 1):
                    WAux[j] = normalVol[i][m + j*self.M];
                W[i].resize(self.k + 1);
                W[i] = self.L.dot(WAux);

            StockExpSum = self.vol0*sqrt(self.h)*normalStock[0][m]
                        - 0.5*self.h*self.vol0*self.vol0;
            for i in range(1, self.k + 1): #i<=k
                for j in range(1, i + 1):
                    I += sqrt(sigmaPaths[m][i - j])*W[i - j][j]
                volti = self.vol0*exp(self.alpha*I);
                StockExpSum += volti*sqrt(self.h)*normalStock[i][m]
                             - 0.5*self.h*volti*volti;
                I = 0.;
```

```
42
43              for i in range(self.k + 1, self.N): #i>k
44                  for j in range(1, self.k + 1):
45                      I += sqrt(sigmaPaths[m][i - j])*W[i - j][j]
46                  for j in range(self.k + 1, i + 1):
47                      I += g(self.b[j - (self.k + 1)]/float(self.n), self.H)
48                          *sqrt(sigmaPaths[m][i - j])*W[i - j][0];
49                  volti = self.vol0*exp(self.alpha*I);
50                  StockExpSum += volti*sqrt(self.h)*normalStock[i][m]
51                              - 0.5*self.h*volti*volti;
52                  I = 0.;
53
54              ST = self.S0*exp(StockExpSum);
55
56              X[m].resize(N_K);
57              Y[m].resize(N_K);
58              for l in range(N_K):
59                  Y[m][l] = mx(ST - K[l], 0.);
60                  X[m][l] = ST;
61
62          for l in range(N_K):
63              for m in range(self.M):
64                  YMean[l] += Y[m][l];
65                  XMean_M[l] += X[m][l];
66              YMean[l] /= float(self.M)
67
68          for l in range(N_K):
69              for m in range(self.M):
70                  b[l] += Y[m][l]*X[m][l];
71                  bAux[l] += X[m][l]*X[m][l];
72
73              b[l] = b[l] - YMean[l]*XMean_M[l];
74              bAux[l] = bAux[l] - XMean_M[l]*XMean_M[l]/float(self.M);
75              b[l] /= bAux[l];
76
77          for l in range(N_K):
78              for m in range(self.M):
79                  call[l] += Y[m][l] - b[l]*X[m][l];
80
81              call[l] /= float(self.M)
82              call[l] = call[l] + b[l]*self.S0;
83
84          self.call = call
```

3.3.3 Lévy-driven Ornstein-Uhlenbek process

Here we briefly discuss the simulation method used to generate paths from the affine regular process suggested by B.Horvath, A.Jacquier and P.Tankov in [3] to calibrate VIX smiles.

The dynamics of the process are

$$\mathrm{d}\Gamma_t = -\lambda\Gamma_t \mathrm{d}t + \mathrm{d}L_t \tag{3.12}$$

where L_t is a Lévy driving Compound Poisson process with jump intensity A and exponential law with parameter a.

The dynamics (3.12) can be rewritten as

$$\mathrm{d}(\exp(\lambda t))\Gamma_t) = \exp(\lambda t)\mathrm{d}L_t$$

now, integrating from 0 to t we obtain

$$\Gamma_t \exp(\lambda t) = \Gamma_0 + \int_0^t \exp(\lambda s)\mathrm{d}L_s = \Gamma_0 + \sum_{0\leq s\leq t} \exp(\lambda s)\Delta L_s$$

Finally, the process can expressed in the following form

$$\Gamma_t = \Gamma_0 \exp(-\lambda t) + \sum_{0\leq s\leq t} \exp(-\lambda(t-s))\Delta L_s$$

See ([7], Chapter 11) for an introduction to stochastic calculus for jump processes.

To generate a path on the interval $[0, T]$ we need to

- Generate the Compound Poisson process on the same interval. This is straightforward but see [6] for reference.
- Discretize the interval $[0, T]$ on a grid $\{t_i\}_{i=0}^N$. Here we choose a equidistant grid.
- In each interval find the number of jumps and sum up the jumps $\sum_{j=1}^{k_i} \Delta L_j$ where k_i denotes the number of jump on the interval $[t_{i-1}, t_i]$.
- Compute Γ_{t_i} as

  $$\Gamma_{t_i} = \exp(-\lambda h)(\Gamma_{t_{i-1}} + \sum_{j=1}^{k_i} \Delta L_j)$$

  with $h = \frac{T}{N}$.

The implementation can be found in the next page under the class name `LevyOU`. It has two member functions, `CPoissonExp` and `Paths` which computes the Compound Poisson and the Lévy-driven Ornstein-Uhlenbek processes respectively.

```
%%cython --cplus --force
from libcpp.vector cimport vector
import numpy as np
cimport numpy as np
from numpy cimport ndarray

cdef extern from "math.h":
    double exp(double m)

cdef double Exp(a):
    return np.random.exponential(1/a);

cdef class LevyOU(object):

    cdef public double T, levy0, lamb, A, a;
    cdef public vector[double] jumpTimes, jumpSizes;
    cdef public int N, M;
    cdef public vector[vector[double]] levyPaths;

    def __cinit__(self, levy0, lamb, A, a, T):
        self.levy0, self.lamb, self.A, self.a, self.T = levy0, lamb, A, a, T;

    cpdef CPoissonExp(self):
        cdef double TT = Exp(self.A);
        cdef double auxSize = Exp(self.a);

        while TT <= self.T:
            self.jumpTimes.push_back(TT);
            self.jumpSizes.push_back(auxSize);
            TT += Exp(self.A)
            auxSize += Exp(self.a)

        if TT > self.T:
            self.jumpTimes.push_back(0.);
            self.jumpSizes.push_back(0.); #This means no jump before T;

    cpdef Paths(self, int N, int M):
        cdef double h = self.T/float(N);
        cdef vector[double] levy;
        levy.resize(N + 1);
        levy[0] = self.levy0;

        cdef double jumps = 0.0;
        cdef int kCurrent = 0;
```

```
45          cdef int kPrev = 0;
46          cdef vector[vector[double]] CP;
47
48          cdef vector[vector[double]] Paths;
49          Paths.resize(M);
50
51          for m in range(M):
52              self.CPoissonExp();
53              self.jumpSizes.insert(self.jumpSizes.begin(), 0.);
54              jumps = 0.0;
55              kCurrent = 0;
56              kPrev = 0;
57              for i in range(1, N + 1):
58                  while (self.jumpTimes[kCurrent] <= (i*h)) and
59                         (kCurrent < self.jumpTimes.size()- 1):
60                      kCurrent +=1;
61                  jumps = 0.0;
62                  for j in range(kPrev , kCurrent):
63                      jumps += exp(-self.lamb*((i-1)*h - self.jumpTimes[j]))
64                               *(self.jumpSizes[j+1] - self.jumpSizes[j]);
65
66                  kPrev = kCurrent;
67                  levy[i] = exp(-self.lamb*h)*(levy[i - 1] + jumps)
68
69              Paths[m] = levy;
70
71          self.levyPaths = Paths
```

3.3.4 Cox-Ingersoll-Ross process

The CIR process has dynamics

$$\mathrm{d}\Gamma_t = k(\theta - \Gamma_t)\mathrm{d}t + \sigma\sqrt{\Gamma_t}\mathrm{d}W_t$$

where θ is the long term mean, k to the speed adjustment and σ to the volatility.

To simulate this process we will use the Ninomiya-Victoir scheme, [9], assuming that the Feller condition is satisfied

$$\sigma^2 \leq 2k\theta$$

Remark 3.5. The assumption presented above is not needed for our study but the Ninomiya-Victoir scheme is very fast and accurate in this case, being a desirable feature for calibration. However, the assumption will restrict our calibration results. Calibration without restriction on the parameters is left for future research. This can be done by using either a exact simulation method or even the faster second order scheme proposed by A. Alfonsi in [10] or similars.

Implementation

```
%%cython --cplus --force
from libcpp.vector cimport vector
cimport numpy as np
from numpy cimport ndarray

cdef extern from "math.h":
    double sqrt(double m)
    double exp(double m)

cpdef cir2(double x0, double k, double theta, double sigma, double T, int N, int M,
           vector[vector[double]] NormalMatrix):
    #Assuming the Feller condition is satisfied
    cdef double h = T/float(N);
    cdef double e = exp(-k*h/2.);
    cdef double p_phi_k = (k*theta - sigma*sigma/4.)*(1. - e)/k;
    cdef double r, temp;
    cdef vector[double] cir_path;
    cir_path.resize(N + 1);
    cir_path[0] = x0;

    cdef vector[vector[double]] Paths;
    Paths.resize(M);
    for j in range (0, M):
        for i in range(0, N ):
            #Now t = h, x = previous cir, Dist = Z[i]
            r = sqrt(p_phi_k + e*cir_path[i]);
            temp = r + sigma*sqrt(h)*NormalMatrix[i][j]/2.;
            cir_path[i + 1] = e*temp*temp + p_phi_k;

        Paths[j] = cir_path;

    return Paths
```

3.3.5 Calibration Results

Here we present the calibration results obtained on May 14, 2014 using the Lévy-driven Ornstein-Uhlenbek proposed by B. Horvath, A. Jacquier and P.Tankov and the Cox-Ingersoll-Ross process to modulate the instananeous volatility process.

We have calibrated only four parameters, H, ν, ρ and σ_0 by minimising the sum of squared differences between model prices and mid market prices of European Call option prices on the SPX Index. Data has been obtained from WRDS (Wharton Research Data Services).

Prices have been simulated using the class `CallRS` (A.3) based in the Riemann sum method with 20000 Monte Carlo replications and 500 time steps per year.

The reason behind choosing only those parameters resides in the desirable goal of achieving joint calibration to SPX and VIX smiles. Here, we will take the parameters that define the Γ process as given from calibration to VIX smiles.

Lévy-driven Ornstein-Uhlenbek case

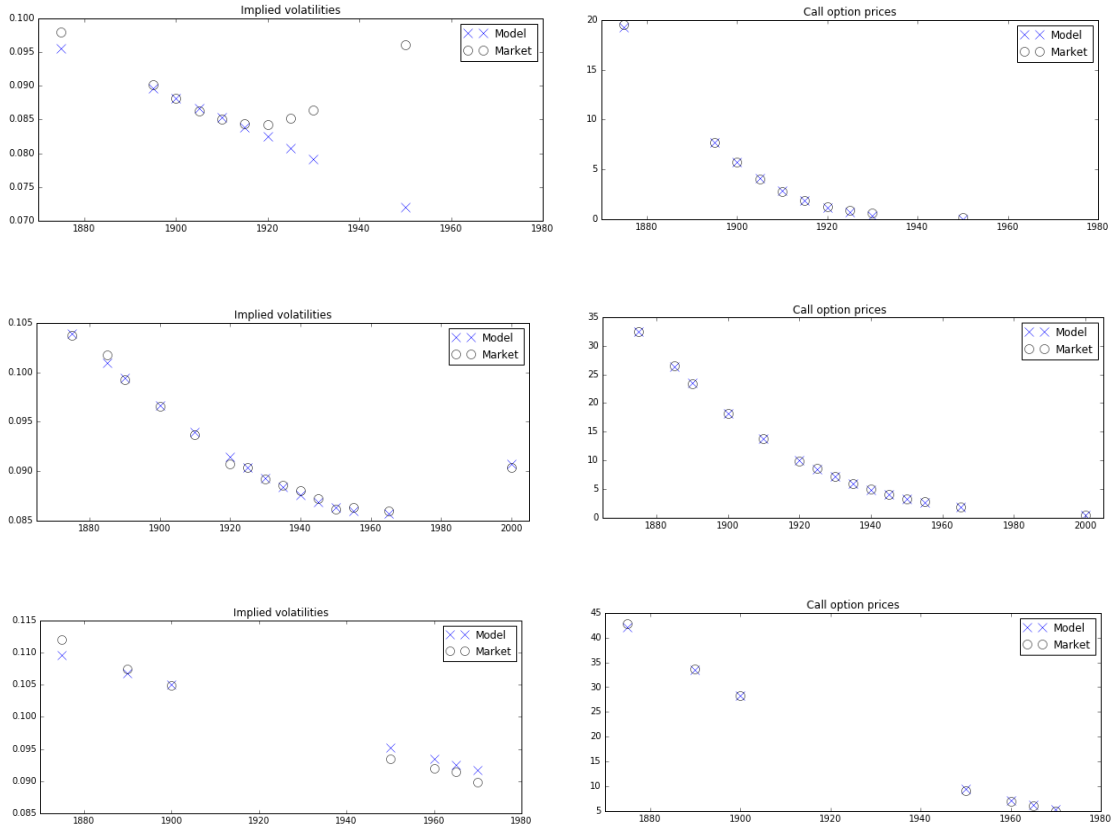


Figure 15: Implied volatilities and option prices as observed in the market and those simulated by the model after calibration. From top to bottom the maturities are: 9, 38, 66 days. The Γ process has been defined using the calibrated parameters from ([3], Table 1, pag 22) for the following maturities: 7, 35 and 63 days.

Lévy-driven Ornstein-Uhlenbek					
T	H	ν	ρ	σ_0	Error
9	0.08703	5.99945	-1.0	0.09116	0.0005
38	0.15234	7.03255	-0.71281	0.08734	1.90e-06
66	0.2293	5.49893	-0.99347	0.10142	1.57e-05

Cox-Ingersoll-Ross case

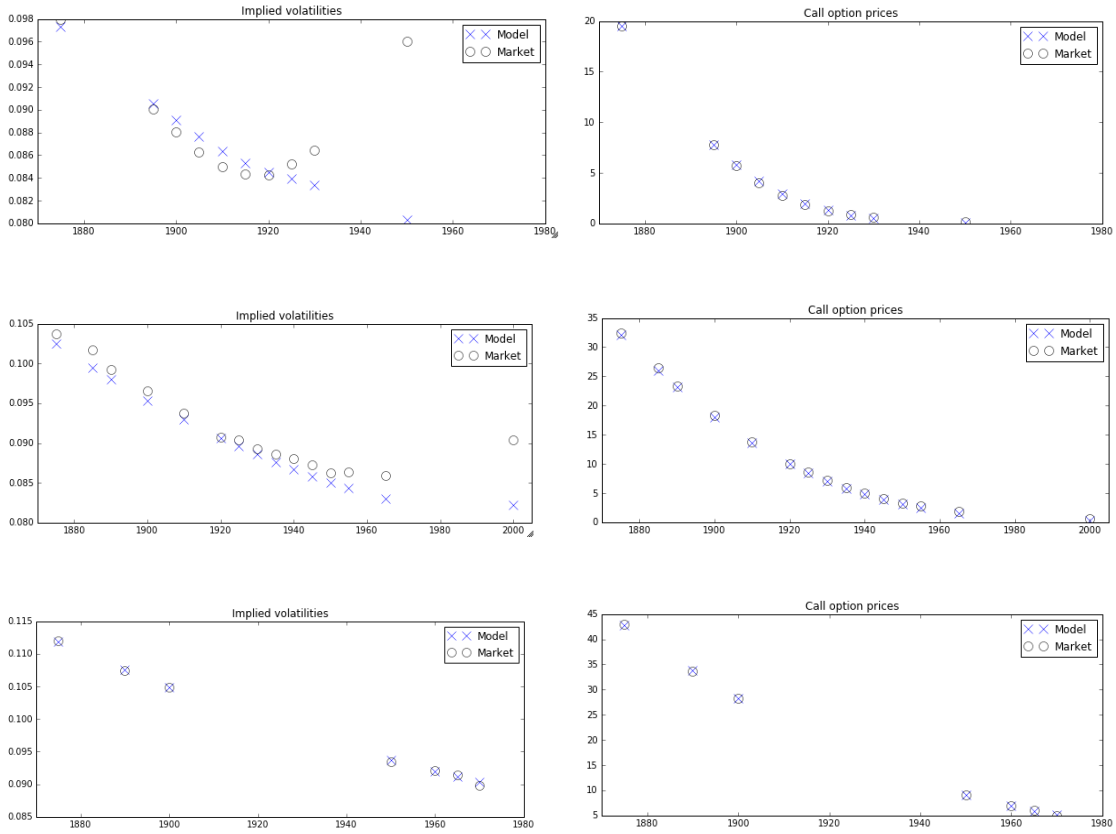


Figure 18: Implied volatilities and option prices as observed in the market and those simulated by the model after calibration. From top to bottom the maturities are: 9, 38, 66 days. The Γ process has been defined using the calibrated parameters from 3.4.3

Cox-Ingersoll-Ross					
T	H	ν	ρ	σ_0	Error
9	0.09426	6.09117	-0.98583	0.0927	0.0005
38	0.13658	5.84783	-0.65588	0.08879	7.11e-05
66	0.10872	5.54235	-0.73286	0.09462	4.81e-07

The minimization has been done using the `TNC` algorithm from Python `optimize` toolbox and the error used in both cases is the sum of the squared differences between observed implied volatilities and the ones obtained from the model prices.

Notice that volatility of volatility obtained turned out to be very large. This is due to how the parameter α of the kernel function has been chosen in 3.1.3. The parameter α proposed in [3] seems a more sensible option, giving a volatility of volatility much lower.

Remark 3.6. During calibration, the maximum number of iteration was achieved in multiple occasions which means that better results can be obtained. From empirical tests, it has been seen that calibrating to all the parameters also reduces the error.

The implementation using the Hybrid Scheme as presented 3.1.3 (`CallHS`) is ready to use for calibration purposes. Tests on the same data are left for future research.

However, notice that the Γ processes considered here are not covariance stationary so some of the result derived in [4] might not apply.

3.4 Forward Variance

As mentioned at the beginning of section 3, B.Horvath, A.Jacquier and P.Tankov introduced the set of Modulated Volterra stochastic volatility process meant to capture the shape of the VIX smiles. Similarly as we did when obtaining the SPX smiles, to obtain the VIX smiles we need to be able to price VIX options. However, as mentioned in [3], working directly with the instantaneous volatility process is not as easy as working with the forward variance dynamics. Again in [3] these dynamics together with methods to price VIX options via Monte Carlo are presented.

In this section some of these results will be presented to be able to explore the behaviour of the model when the the process Γ that modulates the integral $\int_0^t \sqrt{\Gamma_s} g(t,s) \mathrm{d}B_s$ is given by the Cox-Ingersoll-Ross dynamics.

It will be useful to recall the definition of a regular affine process.

Definition 3.7. A stochastic process Γ_t defined on a probability space $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\geq 0}, \mathbb{P}_x)$ and with state space $\mathcal{D} = \mathbb{R}_+^m \times \mathbb{R}_n$ is a regular affine process if it satisfies the following properties

- $\mathbb{P}_x$ is defined as a probability measure on $(\Omega, \mathcal{F})$ such that $\mathbb{P}(X_0 = x) = 1$.
- Γ_t is a time-homogeneous Markov process.
- There exist functions $\phi \in \mathbb{C}$ and $\psi \in \mathbb{C}^m \times \mathbb{C}^n$ such that for all $(t,u) \in \mathbb{R}_+ \times \mathcal{U}$

$$\mathbb{E}[\exp(\langle \Gamma_t, u\rangle)|X_0 = x] = \mathbb{E}_x[\exp(\langle \Gamma_t, u\rangle)] = \exp(\phi(t,u) + \langle x, \psi(t,u)\rangle)$$

where $\mathcal{U} = \{u \in \mathbb{C} \mid \mathcal{R}e(\langle x, u\rangle) \leq 0\}$

In the 1-dimensional real valued case the last condition simply becomes

$$\mathbb{E}_x[\exp(\Gamma_t u)] = \exp(\phi(t,u) + x\psi(t,u))$$

so the log-characteristic function of the process Γ is affine on u.

Example 3.1. *The CIR process with dynamics*

$$\mathrm{d}\Gamma_t = k(\theta - \Gamma_t)\mathrm{d}t + \sigma\sqrt{\Gamma_t}\mathrm{d}W_t$$

is a regular affine process.

A standard procedure to check this property is to assume that such ψ and ϕ functions exist such that

- *ψ and ϕ are at least in $\mathbb{R}^1$.*
- *$\psi(0,u) = u$ and $\phi(0,u) = 0$ for all $u \in \mathbb{C}$*

Then we can define $f(t,\Gamma_t) := \exp(\phi(T-t,u) + \Gamma_t\psi(T-t,u))$ so if f_t were a martingale

$$\begin{aligned}\mathbb{E}[f_T] &= \mathbb{E}[\exp(\phi(0,u) + \Gamma_t\psi(0,u))] = \mathbb{E}[\exp(\Gamma_T u)] \\ &= f_0 = \exp(\phi(T,u) + \Gamma_t\psi(T,u))\end{aligned}$$

applying now Itô formula we have

$$\begin{aligned}\mathrm{d}f(t,\Gamma_t) &= f_t(t,\Gamma_t)\mathrm{d}t + f_x(t,\Gamma_t)\mathrm{d}\Gamma_t + \frac{1}{2}f_{xx}(t,\Gamma_t)(\mathrm{d}\Gamma_t)^2 \\ &= f(t,\Gamma_t)(-\phi_t(T-t,u) - \Gamma_t\psi_t(T-t,u))\mathrm{d}t + f(t,\Gamma_t)\psi(T-t,u)k(\theta-\Gamma_t)\mathrm{d}t \\ &+ f(t,\Gamma_t)\psi(T-t,u)\sigma\sqrt{\Gamma_t}\mathrm{d}W_t + \frac{1}{2}f(t,\Gamma_t)\psi^2(T-t,u)\sigma^2\Gamma_t\mathrm{d}t\end{aligned}$$

So $f(t,\Gamma_t)$ is a local martingale if it has zero drift i.e.

$$\phi_t(T-t,u) + \Gamma_t\psi_t(T-t,u) = \frac{1}{2}\psi^2(T-t,u)\sigma^2\Gamma_t + \psi(T-t,u)k(\theta-\Gamma_t)$$

Since this equation needs to hold for all $(t,\Gamma_t) \in \mathbb{R}_{\geq 0} \times \mathbb{R}$ we obtain

$$\begin{cases}\phi_t(t,u) = k\theta\psi(t,u) \\ \psi_t(t,u) = \frac{1}{2}\psi^2(t,u)\sigma^2 - k\psi(t,u)\end{cases}$$

with $\psi(0,u) = u$ and $\phi(0,u) = 0$.

The second equation is known as the Bernoulli differential equation with $n = 2$ so it can be solved by the change of variable $y(t) = \psi(t,u)^{-1}$ as follows

$$y_t(t) = \ -\psi^{-2}(t,u)\psi_t(t,u) = -\psi^{-2}(t,u)(\frac{1}{2}\psi^2(t,u)\sigma^2 - k\psi(t,u))$$

Then

$$(\exp(-kt)y(t))_t = \exp(-kt)y_t(t) - k\ \exp(-kt)y(t) = -\exp(-kt)\frac{1}{2}\sigma^2$$

and integrating from 0 to t we obtain

$$y(t) = \exp(kt)y(0) - \frac{\sigma^2}{2k}(\exp(kt) - 1)$$

with $y(0) = \frac{1}{u}$ we finally have

$$\psi(t,u) = \frac{u\ \exp(-kt)}{1 - \frac{\sigma^2 u}{2k}(1 - \exp(-kt))}$$

$\phi(t,u)$ is directly obtained by integrating $k\theta\psi(t,u)$ from 0 to t

$$\phi(t,u) = -\frac{2k\theta}{\sigma^2}log(1 - \frac{\sigma^2 u}{2k}(1 - \exp(-kt)))$$

Now having the characteristic function of the process, which totally defines its probability distribution, we can extend the example and obtain the second moment of X_t which can be computed using 3.2

Let us define $char_{\Gamma_T}(u) := \mathbb{E}_t[\exp(\Gamma_T iu)]$, then by the properties of the characteristic function

$$\mathbb{E}[\ \Gamma_T\ ] = -i\ char'_{\Gamma_T}(0) = -i\ \exp(\phi(T,0) + \Gamma_t\psi(T,0))(\phi_u(T,0) + \Gamma_t\psi_u(T,0))$$

Where

- $\phi(T,0) = 0$
- $\psi(T,0) = 0$
- $\phi_u(T,0) = -\frac{2k\theta}{\sigma^2}(-\frac{\sigma^2 i}{2k}(1 - \exp(-kT))) = i\ \theta((1 - \exp(-kT)))$
- $\psi_u(T,0) = i\ \exp(-kT)$

So finally by substituting this identities into the last equation we obtain

$$\mathbb{E}[\ \Gamma_T\ ] = \theta((1 - \exp(-kT)) + \Gamma_0 \exp(-kT)$$

And the second moment of X_T is

$$\mathbb{E}[X_T X_T] = \int_0^T \mathbb{E}[\ \Gamma_s\ ]\ g(T,s)^2 \mathrm{d}s = \int_0^T (\theta((1 - \exp(-ks)) + \Gamma_0 \exp(-ks))\ g(T,s)^2 \mathrm{d}s$$

As cited in [3] from the work of D. Duffie, D. Filipović and W. Schachermayer in [8], the infinitesimal generator of the process Γ can presented in the following form

$$\mathcal{L}f(x) = k(\theta - x)\frac{\partial f}{\partial x}(x) + \frac{\sigma^2 x}{2}\frac{\partial^2 f}{\partial x^2}(x) + \\ + \int_0^\infty (f(x+z) - f(x))m(\mathrm{d}z) + \int_0^\infty (f(x+z) - f(x))x\mu(\mathrm{d}z)$$

where k, θ and σ are non negative constants and m and μ are positve measures in $(0,\infty)$ such that $\int_0^\infty (x \wedge 1)(m(\mathrm{d}z) + \mu(\mathrm{d}z))$ is finite.

The following functions and assumption are introduced in [3]

$$R(u) := -ku + \frac{\sigma^2}{2}u^2 + \int_0^\infty (\exp(zu) - 1)\mu(\mathrm{d}z)$$
$$F(u) := k\theta u + \int_0^\infty (\exp(zu) - 1)m(\mathrm{d}z)$$
$$G(u) := \int_0^u g^2(s)\mathrm{d}s$$

Assumption 3.8. For fixed $T \geq 0$, there exists $A > 0$ such that $\int_0^\infty z\ \exp(zA)(m(\mathrm{d}z) + \mu(\mathrm{d}z))$ and $2G(T) + T(0 \vee R(A)) \leq A$.

Under this assumption it is showed in [[3], Proposition 4] that the forward variance has the following dynamics

$$\xi_t(u) = \xi_0(u)\exp(2\int_0^t \sqrt{\Gamma_s}g(u-s)\mathrm{d}B_s + \psi(u-t)\Gamma_t + \phi(u-t) - \psi(u)\Gamma_0 - \phi(u)) \qquad (3.13)$$

where $t \leq u \leq T$ and ψ and ϕ are functions that satisfy the following ordinary differential equations

$$\partial_t\psi(t) = 2g^2(t) + R(\psi(t))$$
$$\partial_t\phi(t) = F(\psi(t))$$

with initial conditions $\psi(0) = \phi(0) = 0$.

Consider now the particular example of the CIR case. The infinitesimal generator of this process takes the following form

$$\mathcal{L}f(x) = k(\theta - x)\frac{\partial f}{\partial x}(x) + \frac{\sigma^2 x}{2}\frac{\partial^2 f}{\partial x^2}(x)$$

This can be checked by applying the Itô formula to $f(X_t)$ where $f : \mathbb{R} \to \mathbb{R}$ and

$$\mathrm{d}X_t = k(\theta - X_t)\mathrm{d}t + \sigma\sqrt{X_t}\mathrm{d}W_t$$

This give us

$$\begin{aligned}\mathrm{d}f(X_t) =&\ f'(X_t)\mathrm{d}X_t + \frac{1}{2}f''(X_t)(\mathrm{d}X_t)^2 \\ =&\ f'(X_t)[k(\theta - X_t) + \frac{\sigma^2}{s}X_t f''(X_t)]\mathrm{d}t + f'(X_t)\sigma\sqrt{X_t}\mathrm{d}W_t\end{aligned}$$

Coming back to the CIR example, and taking the kernel function as $g(t-s) = \alpha(t-s)^{H-\frac{1}{2}}$, we obtain the following ordinary differential equations

$$\begin{aligned}\partial_t\psi(t) &= 2\alpha^2 t^{2H-1} - k\psi(t) + \frac{\sigma^2}{2}\psi^2(t)\\ \partial_t\phi(t) &= k\theta\psi(t)\end{aligned} \tag{3.14}$$

In practice these equations will need to be solved numerically. However, we can simplify the equations by making the following change of variables $\tilde{\psi}(t) = \exp(kt)\psi(t)$ so

$$\begin{aligned}\partial_t\tilde{\psi}(t) &= k\exp(kt)\psi(t) + \exp(kt)\psi_t(t) = k\exp(kt)\psi(t) + \exp(kt)(2\alpha^2 t^{2H-1} - k\psi(t) + \frac{\sigma^2}{2}\psi^2(t))\\ &= k\exp(kt)\psi(t) + \exp(kt)2\alpha^2 t^{2H-1} - \exp(kt)k\psi(t) + \exp(kt)\frac{\sigma^2}{2}\psi^2(t)\\ &= \exp(kt)2\alpha^2 t^{2H-1} + \exp(-kt)\frac{\sigma^2}{2}\tilde{\psi}^2(t)\end{aligned}$$

with $\tilde{\psi}(0) = \exp(k0)\psi(0) = 0$.

$\phi(t)$ is easily obtained integrating from 0 to t

$$\phi(t) = \phi(0) + k\theta\int_0^t \psi(t) = k\theta\int_0^t \psi(t)$$

Possible implementation using the original ordinary differential equations.

```
cpdef double dv_dt_or(double v, double t, double sigma, double k, double alpha,
                      double gamma):
    return 2.0*alpha*alpha*pow(t,2.0*gamma) + sigma*sigma*0.5*v*v - k*v

cpdef psi_or(double x, double x0, double psi_0, double epsilon, double sigma,
             double k, double alpha, double beta):
    cdef double psi;
    if (x == x0):
        return psi_0;
    tt = [epsilon + x0, x];
    sol_v = odeint(dv_dt_or, psi_0, tt, args=(sigma, k, alpha, beta),
                   tcrit = [epsilon]);
    psi = sol_v[-1];
    return psi

cpdef phi_or(double x, double x0, double phi_0, double psi_0, double epsilon,
             double sigma, double theta, double k, double alpha, double gamma):
    res = quad(psi_or, x0, x, args=(x0, psi_0, epsilon, sigma, k, alpha, gamma))[0];
    return k*theta*res + phi_0
```

3.4.1 Pricing VIX options via Monte Carlo

As mentioned in [3], princing of VIX options can be done using the formula

$$\mathbb{E}\left[f\left(\frac{1}{\Theta} \int_T^{T+\Theta} x(u)\xi_T(\gamma, u) \right) \right]$$

with $\xi_T(u)$ following the dynamics (3.13) and $\Gamma_0 = \gamma$.

In the case of Call options the function f takes the form $f(x) = (\sqrt{x} - K)_+$.

Here we will use the kernel $g(t-s) = \alpha(t-s)^{H-\frac{1}{2}}$ and rectangle scheme introduced in [3] which is based in the following simple discretization

$$\frac{1}{\Theta} \int_T^{T+\Theta} x\ \xi_T(\gamma, u) = \frac{x\Delta}{\Theta} \sum_{i=0}^{n-1} \xi_T(\gamma, t_i) = \frac{x}{n} \sum_{i=0}^{n-1} \xi_T(\gamma, t_i)$$

Where, for simplicity, $x(u) \equiv x$ constant and $t_i = T + i\Delta$, $\Delta = \frac{\Theta}{n}$.

Then it is easy to see that for a given path $(\Gamma_t)_{t\geq 0}$,

$$\log(\xi_T(\gamma, t_i)) = 2\int_0^T \sqrt{\Gamma_s} g(t_i - s)\mathrm{d}B_s + \psi(t_i - T)\Gamma_T + \phi(t_i - T) - \psi(t_i)\Gamma_0 - \phi(t_i)$$

is a Gaussian random variable with mean

$$m_i = \Gamma_T \psi(t_i - T) + \phi(t_i - T) - \gamma\psi(t_i) - \phi(t_i)$$

and that the sequence $(\log(\xi_T(\gamma, t_i)))_{i=0}^{n-1}$ has the following covariance structure

$$C_{ij} = 4\alpha^2 \int_0^T \Gamma_s (t_i - s)^{H-\frac{1}{2}} (t_j - s)^{H-\frac{1}{2}} \mathrm{d}s$$

This integral can be discretized on a grid $\{T_k\}_{k=0}^{N-1}$ with $T_k = kh$ and $h = \frac{T}{N}$

$$\begin{aligned} C_{ij} &= 4\alpha^2 \int_0^T \Gamma_s (t_i - s)^{H-\frac{1}{2}} (t_j - s)^{H-\frac{1}{2}} \mathrm{d}s \\ &\approx 4\alpha^2 \sum_{k=0}^{K-1} \Gamma_{T_k} \int_{T_k}^{T_{k+1}} (t_i - s)^{H-\frac{1}{2}} (t_j - s)^{H-\frac{1}{2}} ds := 4\alpha^2 \sum_{k=0}^{K-1} \Gamma_{T_k} C_{ij}^k \end{aligned}$$

C_{ij}^k can be computed as shown in [[3], page 16] ,

where for $t_j > t_i$

$$\begin{aligned} C_{ij}^k &= \int_{T_k}^{T_{k+1}} (t_i - s)^{H-\frac{1}{2}} (t_j - s)^{H-\frac{1}{2}} \mathrm{d}s \\ &= \frac{1}{H + \frac{1}{2}} (t_j - t_i)^{H-\frac{1}{2}} (t_i - T_k)^{H+\frac{1}{2}} {}_2F_1\left(\frac{1}{2} - H, \frac{1}{2} + H, \frac{3}{2} + H, -\frac{t_i - T_k}{t_j - t_i} \right) \\ &- \frac{1}{H + \frac{1}{2}} (t_j - t_i)^{H-\frac{1}{2}} (t_i - T_{k+1})^{H+\frac{1}{2}} {}_2F_1\left(\frac{1}{2} - H, \frac{1}{2} + H, \frac{3}{2} + H, -\frac{t_i - T_{k+1}}{t_j - t_i} \right). \end{aligned}$$

and $t_j = t_i$

$$C_{ii}^k = \int_{T_k}^{T_{k+1}} (t_i - s)^{2H-1} \mathrm{d}s = \frac{1}{2H}((t_i - T_k)^{2H} - (ti - T_{k+1})^{2H})$$

The implementations is as follows

Choose n, N and M natural numbers, where n controls the discretization of the integral $\int_T^{T+\Theta} \xi_T(\gamma, u)$, N the discretization of the covariance coefficients Cij and M the number of Monte Carlo replications.

- For each replication m, generate independent samples $\mathbf{Z}_i$ from the multivariate Gaussian distribution $\mathcal{N}(\mathbf{m}, C)$ for $i = 0, 1, ..., n$ where $C = (C_{ij})_{i \leq j}^n$ and $\mathbf{m} = (m_i)_{i=1}^n$.
- Compute the average $X_m := \frac{x}{n} \sum_{i=0}^{n-1} \exp(Z_i)$
- After M replications estimate the price using the sample mean

$$\mathbb{E}\left[\left(\sqrt{\frac{1}{\Theta} \int_T^{T+\Theta} x(u) \xi_T(\gamma, u)} - K \right)_+ \right] \approx \frac{1}{M} \sum_{m=1}^{M} (\sqrt{X_m} - K)_+$$

3.4.2 Implementation

```
%%cython --cplus --force
from libcpp.vector cimport vector
from math import floor
import time
import numpy as np
cimport numpy as np
from scipy.special import gamma as Gamma
from scipy.special import hyp2f1 as F1
from __main__ import cir2, rBer, psi_or, phi_or,
                     psi_levy, phi_levy, LevyOU, volimp

cdef extern from "math.h":
    double sqrt(double m)
    double exp(double m)
    double pow(double base, double exponent);

cdef double mx(double a, double b):
    if (a > b):
        return a
    else:
        return b

cdef class CallRec(object):
```

```
24
25      cdef public double H, nu, T, Theta, x;
26      cdef public double alpha, gamma, delta, forwardPrice;
27      cdef public double c1, c2, c3, c4;
28      cdef public int n, N, NYear, M;
29      cdef public vector[ double ] call, strikes, impliedVol;
30      cpdef public cov;
31      cpdef public modProcress;
32      cdef public vector[vector[double]] sigmaPaths, meanPaths, normalMx;
33      cpdef public vector[vector[vector[double]]] auxCov;
34
35      def __cinit__(self, H, nu, T, Theta, x, n, NYear, M):
36          self.H, self.nu, self.T, self.Theta, self.x = H, nu, T, Theta, x;
37          self.n, self.NYear, self.M = n, NYear, M;
38          self.gamma = H - 0.5;
39          #self.alpha = nu*sqrt(2.*H*Gamma(1.5 - H)/(Gamma(H + 0.5)*Gamma(2. - 2.*H)));
40          self.alpha = 2.*nu*sqrt(Gamma(1.5 - H)/(Gamma(H + 1.)*Gamma(2. - 2.*H)));
41          self.N = int(floor(T*NYear));
42          self.delta = self.Theta/float(self.n);
43          #self.N = NYear
44
45      cpdef normalCIR(self, vector[vector[double]] normalMx):
46          self.normalMx = normalMx;
47
48      cpdef sPaths(self, double c1, double c2, double c3, double c4, modProcress):
49          self.c1, self.c2, self.c3, self.c4 = c1, c2, c3, c4;
50          self.modProcress = modProcress;
51
52          if ( modProcress == "Levy"):
53              sigma = LevyOU(c1, c2, c3, c4, self.T) #levy0, lamb, A, a
54              sigma.CPoissonExp();
55              sigma.Paths(self.N, self.M);
56              self.sigmaPaths = sigma.levyPaths;
57
58          elif ( modProcress == "CIR"):
59              self.sigmaPaths = cir2(c1, c2, c3, c4, self.T, self.N, self.M,
60                                     self.normalMx);
61                                   #x0, k, theta, sigma
62
63          elif ( modProcress == "Bergomi"):
64              self.sigmaPaths = rBer(self.N, self.M);
65
66      cpdef auxCovariance(self):
67          start = time.time();
68          cdef double g1 = self.H + 0.5;
```

```
69          cdef double g2 = self.H + 1.5;
70          cdef double h = self.T/float(self.N);
71          cdef double aux1;
72          cdef double aux2;
73
74          cdef vector[vector[vector[double]]] auxCov;
75          auxCov.resize(self.n);
76
77          for i in range(self.n):
78              auxCov[i].resize(self.n);
79              for j in range(i , self.n):
80                  auxCov[i][j].resize(self.N);
81                  if (j == i):
82                      for k in range(self.N):
83                          auxCov[i][j][k] = (pow(self.T + i*self.delta - k*h, 2.*self.H)
84                                          - pow(self.T + i*self.delta - (k+1)*h, 2.*self.H));
85                          auxCov[i][j][k] *= (self.alpha*self.alpha*2./self.H);
86                  else:
87                      for k in range(self.N):
88                          aux1 = pow(self.T + i*self.delta - k*h, g1)
89                                 *F1(-self.gamma, g1, g2,
90                                     -(self.T + i*self.delta - k*h)/((j -i)*self.delta));
91                          aux2 = pow(self.T + i*self.delta - (k+1)*h, g1)
92                                 *F1(-self.gamma, g1, g2, -(self.T + i*self.delta
93                                     - (k+1)*h)/((j - i)*self.delta));
94                          auxCov[i][j][k] = 4.*self.alpha*self.alpha
95                                            *pow((j - i)*self.delta, self.gamma)
96                                            *(aux1 - aux2)/g1;
97
98          self.auxCov = auxCov;
99
100     cpdef covariance(self, vector[double] sigmaPath):
101         cdef double Cij = 0.;
102         cdef double Cii = 0.;
103         C = np.zeros((self.n, self.n));
104
105         for i in range(self.n):
106             for j in range(i , self.n):
107                 if (j == i):
108                     for k in range(self.N):
109                         Cii += sigmaPath[k]*self.auxCov[i][j][k];
110                     C[i][i] = Cii;
111                     Cii = 0.;
112                 else:
113                     for k in range(self.N):
```

```
114                         Cij += sigmaPath[k]*self.auxCov[i][j][k];
115
116                     C[i][j] = Cij;
117                     C[j][i] = C[i][j];
118                     Cij = 0.;
119
120         self.cov = C;
121
122     cpdef meanCIR(self):
123         start = time.time();
124         cdef double epsilon = 0.0000000001
125         cdef vector[double] psi1;
126         psi1.resize(self.n - 1);
127         cdef double phi1;
128         cdef double phi2;
129         cdef double psi2;
130         cdef double mean0;
131         cdef vector[double] auxMean;
132         auxMean.resize(self.n - 1);
133
134         cdef vector[ vector[double] ] meanPaths;
135         meanPaths.resize(self.M);
136
137         for i in range(1, self.n):
138             psi1[i - 1] = psi_or(self.delta*i, 0., 0., epsilon, self.c4,
139                                  self.c2, self.alpha, self.gamma);
140             phi1 = phi_or(self.delta*i, 0., 0., 0., epsilon, self.c4, self.c3,
141                           self.c2, self.alpha, self.gamma);
142             psi2 = psi_or(self.T + self.delta*i, 0., 0., epsilon, self.c4,
143                           self.c2, self.alpha, self.gamma);
144             phi2 = phi_or(self.T + self.delta*i, 0., 0., 0., epsilon, self.c4,
145                           self.c3, self.c2, self.alpha, self.gamma);
146             auxMean[i - 1] = phi1 - psi2*self.c1 - phi2;
147
148         mean0 = -psi_or(self.T, 0., 0., epsilon, self.c4, self.c2, self.alpha,
149                         self.gamma)
150                  *self.c1
151                 - phi_or(self.T, 0., 0., 0., epsilon, self.c4, self.c3, self.c2,
152                          self.alpha, self.gamma);
153         for m in range(self.M):
154             meanPaths[m].resize(self.n);
155             meanPaths[m][0] = mean0;
156             for i in range(1, self.n):
157                 meanPaths[m][i] = psi1[i - 1]*self.sigmaPaths[m][self.N]
158                                   + auxMean[i - 1];
```

```
159
160         self.meanPaths = meanPaths;
161         print( "Mean computed in: ", time.time() - start, " seconds" );
162
163     cpdef meanLevy(self):
164         start = time.time();
165         cdef vector[double] psi1;
166         psi1.resize(self.n - 1);
167         cdef double phi1;
168         cdef double phi2;
169         cdef double psi2;
170         cdef double mean0;
171         cdef vector[double] auxMean;
172         auxMean.resize(self.n - 1);
173
174         cdef vector[ vector[double] ] meanPaths;
175         meanPaths.resize(self.M);
176
177         for i in range(1, self.n):#levy0, lamb, A, a
178             psi1[i - 1] = psi_levy(self.delta*i, 0., self.c2, self.alpha,
179                                    self.gamma);
180             phi1 = phi_levy(self.delta*i, 0., self.c3, self.c4, self.c2,
181                             self.alpha, self.gamma);
182             psi2 = psi_levy(self.T + self.delta*i, 0., self.c2, self.alpha,
183                             self.gamma);
184             phi2 = phi_levy(self.T + self.delta*i, 0.,self.c3, self.c4, self.c2,
185                             self.alpha, self.gamma);
186             auxMean[i - 1] = phi1 - psi2*self.c1 - phi2;
187
188         mean0 = -psi_levy(self.T, 0., self.c2, self.alpha, self.gamma)
189                  *self.c1 - phi_levy(self.T, 0., self.c3, self.c4, self.c2,
190                                      self.alpha, self.gamma);
191         for m in range(self.M):
192             meanPaths[m].resize(self.n);
193             meanPaths[m][0] = mean0;
194             for i in range(1, self.n):
195                 meanPaths[m][i] = psi1[i - 1]*self.sigmaPaths[m][self.N]
196                                   + auxMean[i - 1];
197
198         self.meanPaths = meanPaths;
199         print( "Mean computed in: ", time.time() - start, " seconds" );
200
201     cpdef meanBerg(self):
202         start = time.time();
203         cdef vector[ vector[double] ] meanPaths;
```

```
204         meanPaths.resize(self.M);
205         for m in range(self.M):
206             meanPaths[m].resize(self.n);
207 
208         self.meanPaths = meanPaths;
209         print( "Mean computed in: ", time.time() - start, " seconds" );
210 
211     cpdef mean(self):
212 
213         if ( self.modProcress == "CIR" ):
214             self.meanCIR();
215 
216         elif ( self.modProcress == "Bergomi" ):
217             self.meanBerg();
218 
219         elif ( self.modProcress == "Levy" ):
220             self.meanLevy();
221 
222     cpdef Call(self, vector[double] K, double c1, double c2, double c3, double c4,
223                modProcress):
224         self.strikes = K;
225         cdef vector[double] Z;
226         cdef double X = 0.;
227         cdef double forward = 0.;
228         cdef int N_K = K.size();
229         cdef vector[double] call;
230         call.resize(N_K);
231 
232         self.sPaths(c1, c2, c3, c4, modProcress);
233         self.mean();
234         self.auxCovariance();
235 
236         for m in range(self.M):
237             self.covariance(self.sigmaPaths[m])
238             np.random.seed(0)
239             Z = np.random.multivariate_normal(self.meanPaths[m], self.cov);
240             for j in range(self.n):
241                 X += exp(Z[j]);
242 
243             X *= (self.x/float(self.n));
244             forward += sqrt(X);
245             for l in range(N_K):
246                 call[l] += mx(sqrt(X) - K[l], 0.);
247 
248             X = 0.;
```

```
249
250         for l in range(N_K):
251             call[l] /= float(self.M);
252
253         self.call = call
254         self.forwardPrice = forward/float(self.M);
255
256     cpdef impVol(self):
257         cdef vector[double] IVol;
258         cdef int N_K = self.strikes.size();
259         IVol.resize( N_K );
260         for i in range( N_K ):
261             IVol[i] = volimp(self.forwardPrice, self.strikes[i], self.T, 0.,
262                              self.call[i]);
263
264         self.impliedVol = IVol;
```

Notice that this class can be used for the *rBergomi* model and the class of Modulated Stochastic Volatility processes for both CIR and Lévy-driven Ornstein-Uhlenbek cases.

The expected forward price is stored as member data since is needed for the implied volatilities computations.

3.4.3 Calibration Results

Here we present the calibration results obtained on May 14, 2014 using the the Cox-Ingersoll-Ross process to modulate the instantaneous volatility process.

We have calibrated the parameters x, H, ν, Γ_0, k, θ, σ by minimizing the sum of squared differences between model prices and market prices of European Call option prices on the VIX Index.

Prices have been simulated using the class `CallRec` presented above using 5000 Monte Carlo replications, $n = 100$ and α as presented in [3].

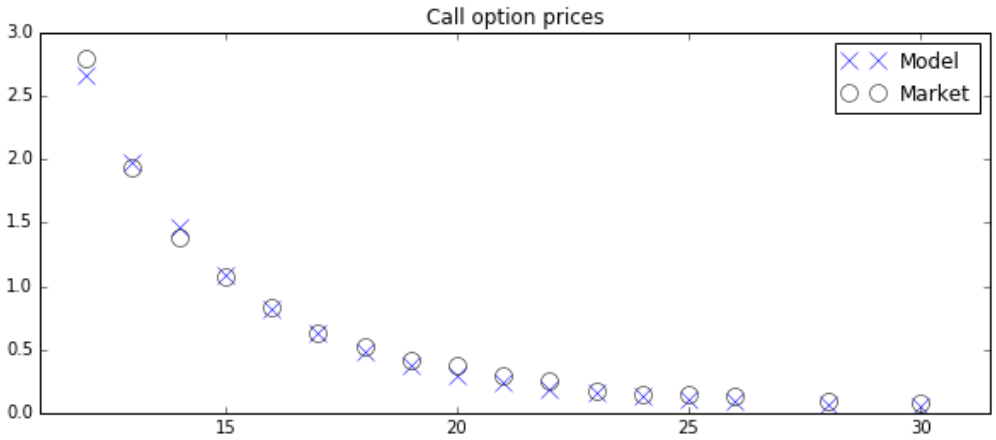


Figure 19: Option prices as observed in the market and those simulated by the model after calibration. Maturity: 35 days

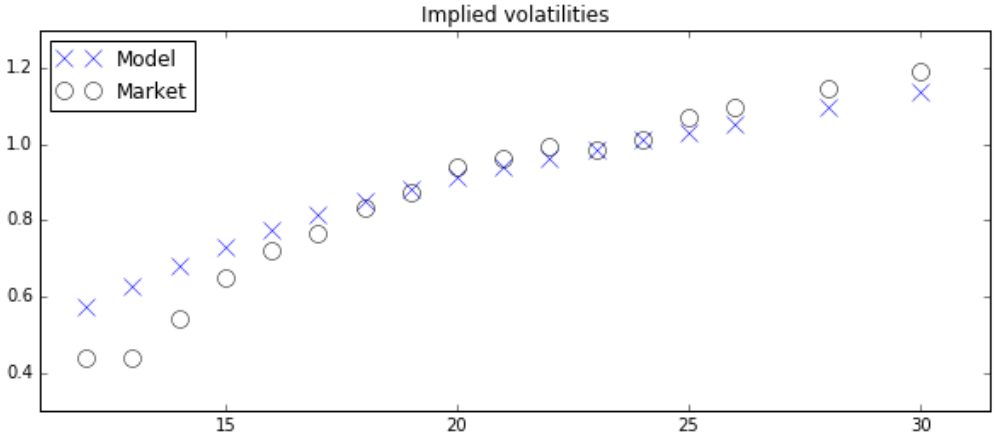


Figure 20: Implied volatilities as observed in the market and those obtained from the simulated prices by the model after calibration. Maturity: 35 days

The calibrated parameters are shown in the table below

Cox-Ingersoll-Ross								
T	x	H	ν	Γ_0	k	θ	σ	Error
35	0.08511	0.04446	0.8199	0.00331	24.867	0.04513	0.09820	0.09714

The minimization used the `TNC` algorithm from Python `optimize` toolbox and the error computed as the sum of the squared differences between observed implied volatilities and the ones obtained from the model prices.

The error here is higher compared to the fits obtained in [3] using the Lévy-driven Ornstein-Uhlenbek process and the approximate pricing method proposed in ([3], pag 19).

However, the tests performed do not seem enough to discard the Cox-Ingersoll-Ross as a suitable Γ process that modulates the instantaneous volatility for the following reasons:

- Low number of replications and discretization steps.
- Restriction on the parameters to satisfy the Feller condition.
- Lack of accuracy when solving the system of ordinary differential equations (3.14) due to the explosion of the kernel function $g(t) = \alpha\, t^{H-\frac{1}{2}}$ around zero, where the initial conditions are defined.

4 Conclusion

To conclude, we have seen that the class of Modulated Stochastic Volatility processes introduced in [3] is able to capture the shape of the VIX smiles and has the potential to provide good fits to SPX smiles.

However, joint calibration of SPX and VIX smiles needs further tests under these models. From empirical experiments, it seems that either the initial volatility is too high when pricing VIX options or too low for the pricing of options on the SPX.

Nevertheless, the class of Modulated Stochastic Volatility processes offers immense possibilities due to the presence of the process Γ that modulates the instantaneous volatility and seems a model to explore when looking for accurate fits to VIX smiles.

A Appendix

A.1 Comments on the Implementation

All the code presented in this thesis has been coded from scratch an based in the cited works. The implementation has been coded using `Cython` a C-extension for `Python` which shows an outstanding performance compared to `Python`.

The user will need to load `Cython`

```
%load_ext cython
```

Then every class/function presented needs to be compiled before calling it.

Example A.1. *The following commands will generate and plot M paths of the instantaneous volatility process using the Hybrid Scheme.*

```
normalM = normalGen(k_tilde, N, M)
normalM.vol()
Hs = HybridScheme(H, nu, k_tilde, N, M, T)
Hs.covariance()
Hs.TBSS(normalM.B, rBerPaths)
Hs.Vol(vol0)

tt = np.linspace(0.0, T, N + 1)
for j in range(0, M):
    plt.plot(tt, Hs.V[j])

plt.title("Instantaneous Volatility Paths")
plt.show()
```

A.2 `rBer` function

Produces constant paths equal to 1.

```
%%cython --cplus --force
from libcpp.vector cimport vector
import numpy as np
cimport numpy as np

cpdef rBer(int N, int M):
    cdef vector[vector[double]] Paths;
    Paths.resize(M);
```

```
    for m in range (0, M):
        Paths[m].resize(N + 1);
        for i in range(0, N + 1):
            Paths[m][i] = 1.;
    return Paths
```

A.3 Implementation of CallRS

```
%%cython --cplus --force
from libcpp.vector cimport vector
from math import floor
import numpy as np
cimport numpy as np
from scipy.special import gamma as Gamma
from scipy.special import hyp2f1 as F1

cdef extern from "math.h":
    double sqrt(double m)
    double exp(double m)
    double pow(double base, double exponent);

cdef double mx(double a, double b):
    if (a > b):
        return a
    else:
        return b

cdef class CallRS(object):

    cdef public double H, nu, T, vol0, S0;
    cdef public double alpha, gamma, h;
    cdef public int n, N, M;
    cdef public vector[ double ] b, call;
    cdef public vector[vector[double]] kernel;
    cpdef public cov, L;

    def __cinit__(self, H, nu, n, M, T, vol0, S0, kernel):
        self.n, self.M, self.H, self.nu = n, M, H, nu;
        self.T, self.vol0, self.S0 = T, vol0, S0;
        self.kernel = kernel;
        self.gamma = H - 0.5;
        self.alpha = nu*sqrt(2.*H*Gamma(1.5 - H)/(Gamma(H + 0.5)*Gamma(2. - 2.*H)));
        self.N = floor(T*n);
        self.h = self.T/float(self.N);
```

```
37
38      cpdef naive(self, vector[double] K, vector[ vector[double] ] normalMx,
39                  vector[ vector[double] ] sigmaPaths):
40          cdef int N_K = K.size();
41          cdef vector[double] call;
42          call.resize(N_K);
43          cdef double h_H = pow(self.h, self.H);
44          cdef double volti;
45          cdef double auxVolSum = 0.;
46          cdef double stockExpSum;
47          cdef double ST;
48
49          for m in range(self.M):
50              for i in range(1, self.N + 1):
51                  for j in range (i - 1):
52                      auxVolSum += sqrt(sigmaPaths[m][j])*self.kernel[i - 1][j]
53                                   *normalMx[j][m];
54
55                  vol_ti = self.vol0*exp(self.alpha*h_H*auxVolSum);
56                  auxVolSum = 0.;
57                  stockExpoSum += vol_ti*sqrt(self.h)*normalMx[i - 1][m + self.M]
58                                  - self.h*vol_ti*vol_ti*0.5;
59
60              ST = self.S0*exp(stockExpoSum);
61              stockExpoSum = 0.;
62
63              for l in range(N_K):
64                  call[l] += mx(ST - K[l], 0.);
65
66          for l in range(N_K):
67              call[l] /= float(self.M);
68
69          self.call = call;
70
71      cpdef controlV(self, vector[double] K, vector[ vector[double] ] normalMx,
72                     vector[ vector[double] ] sigmaPaths):
73          cdef double h_H = pow(self.h, self.H);
74          cdef double auxVolSum = 0.;
75          cdef double volti;
76          cdef double stockExpSum = 0.;
77          cdef double ST;
78          cdef int N_K = K.size();
79          cdef vector[vector[double]] X;
80          X.resize(self.M);
81          cdef vector[double] XMean_M;
```

```
82          XMean_M.resize(N_K);
83          cdef vector[vector[double]] Y;
84          Y.resize(self.M);
85          cdef vector[double] YMean;
86          YMean.resize(N_K);
87          cdef vector[double] b;
88          cdef vector[double] bAux;
89          b.resize(N_K);
90          bAux.resize(N_K);
91          cdef vector[double] call;
92          call.resize(N_K);
93
94          for m in range(self.M):
95              for i in range(1, self.N + 1):
96                  for j in range (i - 1):
97                      auxVolSum += sqrt(sigmaPaths[m][j])*self.kernel[i - 1][j]
98                                   *normalMx[j][m];
99
100                 vol_ti = self.vol0*exp(self.alpha*h_H*auxVolSum);
101                 auxVolSum = 0.;
102                 stockExpoSum += vol_ti*sqrt(self.h)*normalMx[i - 1][m + self.M]
103                                 - self.h*vol_ti*vol_ti*0.5;
104
105             ST = self.S0*exp(stockExpoSum);
106             stockExpoSum = 0.;
107
108             X[m].resize(N_K);
109             Y[m].resize(N_K);
110
111             for l in range(N_K):
112                 Y[m][l] = mx(ST - K[l], 0.);
113                 X[m][l] = ST;
114
115         for l in range(N_K):
116             for m in range(self.M):
117                 YMean[l] += Y[m][l];
118                 XMean_M[l] += X[m][l];
119             YMean[l] /= float(self.M)
120
121         for l in range(N_K):
122             for m in range(self.M):
123                 b[l] += Y[m][l]*X[m][l];
124                 bAux[l] += X[m][l]*X[m][l];
125
126             b[l] = b[l] - YMean[l]*XMean_M[l];
```

```
            bAux[l] = bAux[l] - XMean_M[l]*XMean_M[l]/float(self.M);
            b[l] /= bAux[l];

        for l in range(N_K):
            for m in range(self.M):
                call[l] += Y[m][l] - b[l]*X[m][l];

            call[l] /= float(self.M)
            call[l] = call[l] + b[l]*self.S0;

        self.call = call
```

A.4 Implementation of the Levy functions

These functions here are presented in ([3], pag 17).

```
%%cython --cplus --force
import numpy as np
cimport numpy as np
from scipy.integrate import odeint, quad

cdef extern from "math.h":
    double sqrt(double m)
    double exp(double m)
    double pow(double base, double exponent)

cpdef double fun_levy(double s, double t, double lamb, double alpha, double gamma):
    return 2.*exp(- lamb*(t - s))*pow(s, 2.*gamma)*alpha*alpha

cpdef psi_levy(double t, double t0, double lamb, double alpha, double gamma):
    res = quad(fun_levy, t0, t, args=(t, lamb, alpha, gamma))[0]
    return res

cpdef auxLevy(double u, double u0, double A, double a, double lamb, double alpha,
              double gamma):
    psi = psi_levy(u, u0, lamb, alpha, gamma)
    return A*psi/(a - psi);

cpdef phi_levy(double t, double t0, double A, double a, double lamb, double alpha,
               double gamma):
    res = quad(auxLevy, t0, t, args=(t0, A, a, lamb, alpha, gamma))[0]
    return res
```

References

[1] J. Gatheral, T. Jaisson and M. Rosenbaum. *Volatility is rough*, 2014. Available at arXiv:1410.3394.

[2] C. Bayer, P. Friz and J. Gatheral. *Pricing under rough volatility.* Quantitative Finance, 16: 887-904, 2016.

[3] B. Horvath, A. Jacquier and P. Tankov. *Volatility options in rough volatility models*, 2018. Preprint available at arXiv:1802.01641.

[4] M. Bennedsen, A. Lunde and M. S. Pakkanen. *Hybrid scheme for Brownian semistationary processes*, 2017 Available at arXiv:1507.03004.

[5] O. E. Barndorff-Nielsen and J. Schimegel. *Brownian semistationary processes and volatility/intermittency*, 2009 H. Albrecher, W.J. Runggaldier and W. Schachermayer (Eds). *Advanced financial modelling*, volume 8 of *Radon Series. Comput. Appl. Math.,* pp. 1-25, Walter de Gruyter, Berlin.

[6] P. Glasserman. *Monte Carlo Methods in Financial Engineering.* Springer-Verlag New York, 2004.

[7] S. E. Shreve. *Stochastic calculus for finance. II, Continuous-time models.* Springer New York, 2004.

[8] D. Duffie, D. Filipović and W. Schachermayer. Affine processes and applications in finance. *Annals of Applied Probability, 13(3): 984-1053*, 2003.

[9] S. Ninomiya and N. Victoir. Weak approximation of stochastic differential equations and application to derivative pricing. *Applied Mathematical Finance, Vol. 15, No. 2, pp. 107-121*, 2008.

[10] A. Alfonsi. High order discretization schemes for the CIR process: application to Affine Term Structure and Heston models. *Mathematics of Computation, American Mathematical Society 79 (269), pp.209-237.*, 2010.

[11] Cboe. VIX White Paper. `http://www.cboe.com/framed/pdfframed?content=/micro/vix/vixwhite.pdf&section=SECT_MINI_SITE&title=VIX+White+Paper`