Technical documentation of `mc_bayer_roughbergomi`

Background

`mc_bayer_roughbergomi` computes the price of a European call option in the rough Bergomi model (Bayer, Friz and Gatheral) using Monte Carlo simulation based on the hybrid scheme of Bennedsen, Lunde and Pakkanen.

The rough Bergomi model is described by the dynamics

$$dS_t = \sqrt{v_t} S_t dZ_t,$$
$$v_t = \xi_0(t) \exp\left(\eta \widetilde{W}_t - \frac{1}{2}\eta^2 t^{2H} \right),$$

where W, Z denote two *correlated* standard Brownian motions–with correlation ρ–, ξ_0 denotes the forward variance curve[1] at time t, $\eta > 0$ is interpreted as a volatility of volatility parameter. More interestingly, for $0 < H < 1$ we have

$$\widetilde{W}_t = \int_0^t K(t,s) dW_s, \quad K(t,s) = \sqrt{2H}(t-s)^{H-1/2}.$$

This defines a variant of the fractional Brownian motion, sometimes called "Riemann-Liouville fractional Brownian motion", which is not the standard fBm. It can be simulated either by Cholesky factorization of the covariance matrix, or by the above mentioned *hybrid scheme*. Without loss of generality, we shall assume $S_0 = 1$ in what follows.

Conditioning on the σ-algebra generated by W, we have

$$C_{RB}(T,K) = E\left[(S_T - K)^+ \right] =$$
$$E\left[C_{BS}\left(S_0 = e^{\rho \int_0^T \sqrt{v_t} dW_t - \frac{1}{2}\rho^2 \int_0^T v_t dt},\ K = K,\ T = 1,\ \sigma^2 = (1-\rho^2)\int_0^T v_t dt \right) \right],$$

where C_{BS} denotes the Black-Scholes price. Hence, we need to simulate the random variables

$$\int_0^T \sqrt{v_t} dW_t, \quad \int_0^T v_t dt.$$

Implementation

The key challenge of the algorithm is the (joint) simulation of W and $\widetilde{W}$. Indeed, the above integrals will be discretized by simple left-point rules based on a uniform time-grid $0 = t_0 < t_1 < \cdots < t_N = T$. As v is directly defined in terms of $\widetilde{W}$, we hence only need to simulate the Gaussian random vectors

$$\left(W_{t_1}, \dots, W_{t_N}\right) \text{ and } \left(\widetilde{W}_{t_1}, \dots, \widetilde{W}_{t_N}\right).$$

We now explain the function `compute_Wtilde`, which is responsible to simulate $\widetilde{W}$. In the context of this function, time is re-scaled to $T = 1$, and the hybrid

[1] In the current implementation, we assume ξ_0 to be constant.

scheme (for $\kappa = 1$) requires two (independent) standard Gaussian random vectors of size N as input, `W1` and `W1perp`. The variable `W1` corresponds to the random vector $(W_{t_1}, \ldots, W_{t_N})$ by

$$W1[i] = \sqrt{N} \left(W_{t_{i+1}} - W_{t_i} \right).^{2}$$

`compute_Wtilde` computes a variable `Wtilde` which directly corresponds to the random vector $\left(\widetilde{W}_{t_1}, \ldots, \widetilde{W}_{t_N}\right)$—again noting the re-scaling.

Computationally, the most demanding work in `compute_Wtilde` is the computation of a discrete convolution of the array `W1` with an array `Gamma`, which is done by FFT (lines 104–114 in rBergomi.cpp). The steps are rather self-explaining, apart from the rather confusing steps required by the library FFTW used for performing the FFT. In short, the command `fftw_execute(fPlanX)` computes the (forward) DFT of the complex array `xC` and saves the result in the complex array `xHat`–and similarly for `fPlanY`. On the other hand, `fftw_execute(fPlanZ)` computes the inverse DFT of the complex array `zHat` and saves the result in the complex array `zC`.

As far as my understanding of FFTW is concerned, the following happens underneath. The scenario is the following: assume we want to compute the DFT $\hat{x}$ of many different complex vectors x, each of size n—this is true in our case. We first allocate two arrays of type `fftw_complex` of length n each, let us call them `xC` and `xHat`, respectively. Now we generate an *FFT-plan* using the function `fftw_plan_dft_1d`. In this FFT-plan both the type of the DFT (forward or inverse) and the length of the vectors involved and even the memory location of the arrays representing those vectors are hard-coded. The idea of this step is (apparently) that it allows FFTW to do various optimizations based on the specific architecture, dimension and memory layout involved. After an FFT-plan was generated, `fftw_execute` simply computes the DFT of whatever is saved in `xC` and stores the result in `xHat`.

Note that constructing an FFT-plan can be rather costly, which is why all the complex arrays and the FFT-plans are only generated once and then passed on to the auxiliary functions for computations.

[2] Recall the re-scaling $T = 1$ here!