n07_mcs_heston

May 15, 2017

```
In [3]: %matplotlib inline

        from pylab import *
        from matplotlib import pyplot as pl
        from BCC_option_valuation import H93_call_value

        import numpy as np
```

0.1 The Heston Model and Its Simulation

from Yves Hilpisch "Derivatives Analytics with Python" Wiley Finance (2015) (in my humble personal opinion, one of the most useful books in quantitative finance of recent years) You cannot miss to visit his web site!!!

0.1.1 General Description

In this section we are going to consider the stochastic volatility model MH93 with constant short rate. This section values European call and put options in this model by MCS. As for the ZCB values, we also have available a semi-analytical pricing formula which generates natural benchmark values against which to compare the MCS estimates. For $0 \leq t \leq T$, the risk-neutral dynamics of the index in the H93 stochastic volatility model are given by

$$dS_t = rS_t dt + \sqrt{v_t} S_t dZ_t^1$$

with the variance following the square-root diffusion

$$dv_t = \kappa_v(\theta_v - v_t)dt + \sigma_v \sqrt{v_t} dZ_t^2$$

The two Brownian motions are instantaneously correlated with $dZ_t^1 dZ_t^2 = \rho$ This correlation introduces a new problem dimension into the discretization for simulation purposes. To avoid problems arising from correlating normally distributed increments (of S) with chi-squared distributed increments (of v), we will in the following only consider Euler schemes for both the S and v process. This has the advantage that the increments of v become normally distributed as well and can therefore be easily correlated with the increments of S.

Following the presentation of Hilpisch we consider two discretization schemes for S and seven discretization schemes for v. For S we have the **simple** Euler discretization scheme (with $s = t - \Delta t$)

$$S_t = S_s \left(e^{r\Delta t} + \sqrt{v_t}\sqrt{\Delta t} z_t^1 \right)$$

As an alternative we consider the **exact log** Euler scheme

$$S_t = S_s e^{(r - v_t/2)\Delta t + \sqrt{v_t}\sqrt{\Delta t} z_t^1}$$

This one is obtained by considering the dynamics of $\log S_t$ and applying Ito's lemma to it. These schemes can be combined with any of the following Euler schemes for the square-root diffusion ($x^+ = \max[0, x]$): 1. **Full Truncation**

$$\tilde{x}_t = \tilde{x}_s + \kappa(\theta - \tilde{x}_s^+)\Delta t + \sigma\sqrt{\tilde{x}_s^+}\sqrt{\Delta t}z_t$$
$$x_t = \tilde{x}_t^+$$

2. **Partial Truncation**

$$\tilde{x}_t = \tilde{x}_s + \kappa(\theta - \tilde{x}_s)\Delta t + \sigma\sqrt{\tilde{x}_s^+}\sqrt{\Delta t}z_t$$
$$x_t = \tilde{x}_t^+$$

3. **Truncation**

$$x_t = \max\left[0, \tilde{x}_s + \kappa(\theta - \tilde{x}_s)\Delta t + \sigma\sqrt{\tilde{x}_s}\sqrt{\Delta t}z_t\right]$$

4. **Reflection**

$$\tilde{x}_t = |\tilde{x}_s| + \kappa(\theta - |\tilde{x}_s|)\Delta t + \sigma\sqrt{|\tilde{x}_s|}\sqrt{\Delta t}z_t$$
$$x_t = |\tilde{x}_t|$$

5. **Hingham-Mao**

$$\tilde{x}_t = \tilde{x}_s + \kappa(\theta - \tilde{x}_s)\Delta t + \sigma\sqrt{|\tilde{x}_s|}\sqrt{\Delta t}z_t$$
$$x_t = |\tilde{x}_t|$$

6. **Simple Reflection**

$$\tilde{x}_t = \left|\, \tilde{x}_s + \kappa(\theta - \tilde{x}_s)\Delta t + \sigma\sqrt{\tilde{x}_s}\sqrt{\Delta t}z_t \,\right|$$

7. **Absorption**

$$\tilde{x}_t = \tilde{x}_s^+ + \kappa(\theta - \tilde{x}_s^+)\Delta t + \sigma\sqrt{\tilde{x}_s^+}\sqrt{\Delta t}z_t$$
$$x_t = \tilde{x}_t^+$$

In the literature there are a lot of tests and numerical studies available that compare efficiency and precision of different discretization schemes.

0.1.2 Square-Root Diffusion Simulation Function

In the following cell the Python code which implement the previous schema for the variance evolution equation. This is the list of **input parameters**:

```
x0: float
    initial value
kappa: float
    mean-reversion factor
theta: float
    long-run mean
sigma: float
    volatility factor
T: float
    final date/time horizon
M: int
    number of time steps
I: int
    number of paths
row: int
    row number for random numbers
cho_matrix: NumPy array
    cholesky matrix
```

In **return** we get a NumPy array containing the simulated variance paths

```
x: NumPy array
    simulated variance paths
```

```
In [4]: def SRD_generate_paths(x_disc, x0, kappa, theta, sigma,
                               T, M, I, rand, row, cho_matrix):

            dt = T / M
            x = np.zeros((M + 1, I), dtype=np.float)
            x[0] = x0
            xh = np.zeros_like(x)
            xh[0] = x0
            sdt = math.sqrt(dt)
            for t in xrange(1, M + 1):
                ran = np.dot(cho_matrix, rand[:, t])
                if x_disc == 'Full Truncation':
                    xh[t] = (xh[t - 1] + kappa * (theta -
                         np.maximum(0, xh[t - 1])) * dt +
                         np.sqrt(np.maximum(0, xh[t - 1])) * sigma * ran[row] * sdt)
                    x[t] = np.maximum(0, xh[t])
                elif x_disc == 'Partial Truncation':
                    xh[t] = (xh[t - 1] + kappa * (theta - xh[t - 1]) * dt +
                         np.sqrt(np.maximum(0, xh[t - 1])) * sigma * ran[row] * sdt)
                    x[t] = np.maximum(0, xh[t])
                elif x_disc == 'Truncation':
                    x[t] = np.maximum(0, x[t - 1]
                         + kappa * (theta - x[t - 1]) * dt +
                         np.sqrt(x[t - 1]) * sigma * ran[row] * sdt)
                elif x_disc == 'Reflection':
                    xh[t] = (xh[t - 1]
                         + kappa * (theta - abs(xh[t - 1])) * dt +
                         np.sqrt(abs(xh[t - 1])) * sigma * ran[row] * sdt)
                    x[t] = abs(xh[t])
                elif x_disc == 'Higham-Mao':
                    xh[t] = (xh[t - 1] + kappa * (theta - xh[t - 1]) * dt +
                         np.sqrt(abs(xh[t - 1])) * sigma * ran[row] * sdt)
                    x[t] = abs(xh[t])
                elif x_disc == 'Simple Reflection':
                    x[t] = abs(x[t - 1] + kappa * (theta - x[t - 1]) * dt +
                         np.sqrt(x[t - 1]) * sigma * ran[row] * sdt)
                elif x_disc == 'Absorption':
                    xh[t] = (np.maximum(0, xh[t - 1])
                         + kappa * (theta - np.maximum(0, xh[t - 1])) * dt +
                         np.sqrt(np.maximum(0, xh[t - 1])) * sigma * ran[row] * sdt)
                    x[t] = np.maximum(0, xh[t])
                else:
                    print x_disc
                    print "No valid Euler scheme."
                    sys.exit(0)
            return x
```

0.1.3 Variance Reduction Generator

To improve upon valuation accuracy we use both moment matching and antithetic paths for our Python implementation. To generate antithetic paths we use both the pseudo-random number $z_{t,i}$ and its negative value $-z_{t,i}$ where we generate only $I/2$ random numbers. With regard to moment matching, we correct the first two moments of the pseudo-random numbers delivered by Python. The respective code for both antithetic paths and moment matching looks like:

```
In [5]: def random_number_generator(M, I):
            ''' Function to generate pseudo-random numbers.

            Parameters
            ==========
            M: int
                time steps
            I: int
                number of simulation paths

            Returns
            =======
            rand: NumPy array
                random number array
            '''
            if antipath:
                rand = np.random.standard_normal((2, M + 1, I / 2))
                rand = np.concatenate((rand, -rand), 2)
            else:
                rand = np.random.standard_normal((2, M + 1, I))
            if momatch:
                rand = rand / np.std(rand)
                rand = rand - np.mean(rand)
            return rand
```

0.1.4 Function for Heston Asset Process

Input Parameters

```
S0: float
    initial value
r: float
    constant short rate
v: NumPy array
    simulated variance paths
row: int
    row/matrix of random number array to use
cho_matrix: NumPy array
    Cholesky matrix
```

Returns

```
S: NumPy array
    simulated index level paths
```

Note that depending on the time interval Δt used, the drift of the index level process may also show a non-negligible bias (even after correction of the random numbers). We can correct the first moment of the

index level process in a fashion similar to the pseudo-random numbers (the correction is activated by the boolean variable *momatch*).

```
In [6]: def H93_generate_paths(S0, r, v, row, cho_matrix):
            S = np.zeros((M + 1, I), dtype=np.float)
            S[0] = S0
            bias = 0.0
            sdt = math.sqrt(dt)
            for t in xrange(1, M + 1, 1):
                ran = np.dot(cho_matrix, rand[:, t])
                if momatch:
                    bias = np.mean(np.sqrt(v[t]) * ran[row] * sdt)
                if s_disc == 'Log':
                    S[t] = S[t - 1] * np.exp((r - 0.5 * v[t]) * dt +
                            np.sqrt(v[t]) * ran[row] * sdt - bias)
                elif s_disc == 'Naive':
                    S[t] = S[t - 1] * (math.exp(r * dt) +
                            np.sqrt(v[t]) * ran[row] * sdt - bias)
                else:
                    print "No valid Euler scheme."
                    exit(0)
            return S
```

```
In [14]: r       = 0.05   # Fixed Short Rate
         theta_v = 0.20   # long-term variance level
         S0      = 10.0   # initial index level
         K       = 10.0

         v0      = 0.01
         kappa_v = 1.5
         sigma_v = 0.15
         rho     = 0.1

         covariance_matrix    = np.zeros((2, 2), dtype=np.float)
         covariance_matrix[0] = [1.0, rho]
         covariance_matrix[1] = [rho, 1.0]
         cho_matrix           = np.linalg.cholesky(covariance_matrix)

         # time step (per year)
         M0      = 50
         # expiry (y)
         T       = 1.0
         # number of paths per valuation
         I       = 100000

         antipath = False # antithetic paths for variance reduction
         momatch  = True # random number correction (std + mean + drift)
         x_disc   = 'Full Truncation'
         s_disc   = 'Log'

         # memory clean-up
         v, S, rand, h = 0.0, 0.0, 0.0, 0.0
         M = int(M0 * T)  # number of total time steps
         dt = T / M  # time interval in years
         # random numbers
```

```
        rand = random_number_generator(M, I)
        # volatility process paths
        v = SRD_generate_paths(x_disc, v0, kappa_v, theta_v,sigma_v, T, M, I, rand, 1, cho_matrix)
        # index level process paths
        S = H93_generate_paths(S0, r, v, 0, cho_matrix)
        # discount factor
        BOT = math.exp(-r * T)

        # European call option value (semi-analytical)
        C0  = H93_call_value(S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0)

        # inner value matrix call
        h = np.maximum(S - K, 0)

        pv     = BOT * h[-1]                # present value vector
        V0_MCS = np.sum(pv) / I             # MCS estimator
        SE     = np.std(pv) / math.sqrt(I)  # standard error

        rel_error = (V0_MCS - C0) / C0
```

In [15]:
```
print "Call Price with Monte Carlo Simulation = ", V0_MCS, "+/-", SE
print "Call Price with Semi-Analytical Model  = ", C0
print "Relative Error (%)                     = ", 100*rel_error
```

```
Call Price with Monte Carlo Simulation =  1.50003232119 +/- 0.00774714371796
Call Price with Semi-Analytical Model  =  1.49125362287
Relative Error (%)                     =  0.588679093948
```

In [ ]: