# Computational Finance - Case Study - 4

Sowrya Gali - sg4150

May 2023

## 1 Simulating Heston

We have used the Euler discretization to simulate the Heston's Stochastic Volatility Model. The following equations show the discretization.

$$\begin{aligned} \nu_{i+1} &= \nu_i + \kappa(\theta - \nu_i^+)dt + \xi\sqrt{\nu_i^+}\, dW_{i+1}^{\nu^+} \\ S_{i+1} &= S_i \exp{(\mu - \frac{1}{2}\nu_i^+)dt + \sqrt{\nu_i^+ dt}\, dW_{i+1}^S} \end{aligned} \tag{1}$$

The following code implements the simulation using eq.1.

```python
import numpy as np
def simulate_heston(params, S0, T):
    dt = 1.0/T
    mu, kappa, theta, lbd, rho, v0 = params
    sim_array = np.zeros((T+1,2))
    sim_array[0,:] = S0, v0
    w = np.random.normal(size=(T,2))
    for i in range(1,T+1):
        prev_s, prev_v = sim_array[i-1, :]
        vi = prev_v + kappa*(theta-max(0,prev_v))*dt + lbd*np.sqrt(max(0,prev_v)*dt)
    *w[i-1][0]
        w_corr = rho * w[i-1, 0] + np.sqrt(1-rho**2) * w[i-1, 1]
        si = prev_s + (mu - 0.5*max(0,prev_v))*dt + np.sqrt(max(0,prev_v)*dt)*w_corr
        sim_array[i, :] = si,vi
    return sim_array[:,0]
```

Listing 1: Simulation of Heston using Euler Discretization

## 2 Extended Kalman Filter

The following the code snippet implements the Extended Kalman Filter and uses scipy's *fmin* optimizer to get the parameters.

```python
import numpy as np
from scipy.optimize import fmin
heston_parameters = [0.02, 1.5, 0.05, 0.18, 0.5, 0.04]
def extended_kalman(params, H, P, T, s):
    mu, kappa, theta, lbd, rho, v0 = params
    dt = 1.0/T
    F = np.array([[1., -0.5*dt],
                  [0., 1 - kappa*dt]])
    Q = np.array([[1., rho],
                 [rho, 1.]])
    loglikelihood = 0
    v = v0

    for t in range(1, T+1):
        U = np.diag([np.sqrt(v*dt), lbd * np.sqrt(v*dt)])
        prev_y = s[t-1]
        x1 = np.array([prev_y + (mu - 0.5*v)*dt,
                       v + kappa*(theta-v)*dt])
        P1 = F@P@F.T + U@Q@U.T
        S = H@P1@H.T
        K = P1@H.T/S
        v1 = x1[1]
        e = s[t] - s[t-1] - (mu - 0.5*v1)*dt
        loglikelihood += np.log(S) + e**2/S
        x = x1 + K*e
        v = x[1]
        P = (np.diag([1., 1.]) - np.outer(K, H)) @ P1
    return loglikelihood

```

```
H = np.array([1, 0])
P = np.diag([0.01, 0.01])
initial_param = [0.033, 1.7, 0.08, 0.14, 0.36, 0.023]
T = 10*365
S0 = 3.66
s = simulate_heston(heston_parameters, S0, T)
ekm = fmin(extended_kalman, initial_param, args=(H,P,T,s), full_output=True, disp=
    True, xtol=10, ftol=20)
ekm_param = np.round(ekm[0], 4)
```

Listing 2: Extended Kalman Filter

The line 3 in the snippet contains the original parameters. The obtained parameters are [0.0345 1.6622 0.0658 0.1464 0.363 0.0261]. The absolute error is 1.0144 and RMSE is 1.0460.

## 3 Particle Filter

The importance sampling employed in the particle filter has the problem that the variance of the weights increases over time, so the algorithm will diverge this is known as the degeneracy problem. To curb this, we regenerate particles with higher weight and eliminate those with lower weight. The following snippet contains the code for it.

```
# Avoids degeneracy problem
def resampling(w,x):
    N = len(x)
    c = np.cumsum(w)
    U = (np.arange(N) + np.random.rand(N))/N
    i = 1
    x_new = np.zeros(N)
    for j in range(N):
        while U[j] > c[i]:
            i += 1
        x_new[j] = x[i]
    w_new = np.ones(N)/N
    return x_new, w_new
```

Listing 3: Resampling

The following the code snippet implements the Particle Filter and uses scipy's *fmin* optimizer to get the parameters.

```
def n(x,m,s):
    return np.exp(-((x-m)**2)/(2*s**2))/(np.sqrt(2*np.pi)*s)

def particleFilter(params, N, T, s):
    mu, kappa, theta, lbd, rho, v0 = params
    dt = 1.0/T
    w = np.ones(N)/N
    c = 1/(1 + (rho*lbd*dt)/2)
    c2 = rho*lbd*dt
    v = v0*np.ones(N)
    loglikelihood = 0
    for t in range(1, T+1):
        z1 = np.random.normal(N)
        z2 = np.random.normal(N)
        vt = (v + kappa*(theta-v)*dt + (-c2/2)*v \
              + lbd*np.sqrt(v*(1-rho**2)*dt)*z2 \
              + rho*lbd*np.sqrt(v*dt)*z1)/c
        yt, y_prev = s[t], s[t-1]
        m_i = v + kappa*(theta - v)*dt + lbd*rho*(yt-y_prev - (mu - 0.5*v)*dt)
        s_i = lbd*np.sqrt(v*(1-rho**2)*dt)
        m_T = c*(v + kappa*(theta - v)*dt + 0.5*lbd*rho*v*dt)
        s_T = c*lbd*np.sqrt(v*dt)
        m_L = y_prev + (mu - 0.5*vt) * dt
        s_L = np.sqrt(v * dt)
        r = n(yt, m_L, s_L)*n(vt, m_T, s_T) / n(vt, m_i, s_i)
```

```
26         w *= r
27         loglikelihood += np.log(np.sum(w))
28         w /= np.sum(w)
29         v, w = resampling(w, vt)
30     return -loglikelihood
31 
32 initial_param = [0.033, 1.7, 0.08, 0.14, 0.36, 0.023]
33 T = 10*365
34 S0 = 3.66
35 s = simulate_heston(heston_parameters, S0, T)
36 N = 20
37 pf = fmin(particleFilter, initial_param, args=(N,T,s), full_output=True, disp=True,
       xtol=10, ftol=20)
38 pf_param = np.round(pf[0], 4)
```

Listing 4: Particle Filter

The line 3 in the snippet contains the original parameters. The obtained parameters are [0.0385 1.3094 0.0804 0.0622 0.6373 0.0271]. The absolute error is 0.1373 and RMSE is 0.2655.

## 4 Analysis

- The values obtained from both filters are close the true parametrs.
- However, the particle filter's values have lower error compared to the Kalman filter's output.
- This happens because the Kalman filter is designed to be used with linear systems, whereas Particle Filter generalizes ell to the non-linear systems.
- But the computational time for the Particle Filter is much higher compared to Kalman Filter. This is because Particle filter is a Bayes Filter can the particle filter can suffer from the "curse of dimensionality" when dealing with high-dimensional state spaces, which can lead to performance degradation.
- Kalman filter is computationally efficient but may give accurate results for noisy data like stock price movements.