Comprehensive Explanation of the Kalman Filter in a Stochastic Volatility Model

May 21, 2025

1 Mathematical and Conceptual Overview of the Kalman Filter and Stochastic Volatility Model

The Kalman Filter is a mathematical algorithm designed to estimate the hidden state of a dynamic system from noisy observations. In the context of the Python code (`KF_ProblemSet_Solution_Part1.ipynb`), it is used to estimate the latent log-volatility in a stochastic volatility (SV) model, which describes how the volatility of financial returns evolves over time. Below, we explain the mathematical components, the SV model, and the purpose of the code in detail, ensuring clarity for readers without prior knowledge.

1.1 Conceptual Understanding of the Kalman Filter

The Kalman Filter is like a smart assistant that makes an educated guess about a hidden variable (e.g., volatility) based on past data and then refines that guess using new observations. It operates recursively in two steps:

- **Prediction Step (Time Update)**: Predicts the next state and its uncertainty based on the system's dynamics.
- **Update Step (Measurement Update)**: Incorporates new observations to correct the prediction, accounting for noise.

The filter assumes the system follows a *state-space model*, which consists of:

- **State Equation**: Describes how the hidden state (e.g., log-volatility) evolves over time.
- **Measurement Equation**: Relates the hidden state to observed data (e.g., financial returns).

In this case, the Kalman Filter estimates the hidden log-volatility process in an SV model, which is crucial because volatility in financial markets is not constant but changes randomly over time, affecting asset prices.

1.2 The Stochastic Volatility Model

The SV model is a framework used to model financial returns, recognizing that volatility (the degree of price fluctuation) is not fixed but follows a random process. The model assumes returns are generated as:

SV Model Return Equation

$$r_t = \sigma \cdot \exp(0.5 \cdot h_t) \cdot \xi_t, \quad \xi_t \sim N(0,1)$$

- r_t: The return at time t, representing the percentage change in an asset's price.
- σ: A scaling parameter (e.g., 1.0 in the code).
- h_t: The log-volatility, a hidden variable that drives the volatility.
- ξ_t: A standard normal random variable (mean 0, variance 1), representing random market noise.
- $\exp(0.5 \cdot h_t)$: Converts log-volatility to volatility, making volatility time-varying.

The log-volatility h_t follows an autoregressive (AR(1)) process:

SV Model State Equation

$$h_{t+1} = \Phi \cdot h_t + \eta_t, \quad \eta_t \sim N(0,Q)$$

- Φ: Persistence parameter (e.g., 0.95), indicating how slowly volatility changes.
- η_t: Random innovation, normally distributed with variance Q (e.g., $Q = \sigma_\eta^2 = 0.04$).

The SV model indicates *volatility clustering*, a common feature in financial data where periods of high volatility are followed by more high volatility, and vice versa. The Kalman Filter is used to estimate h_t, which is not directly observable, from the noisy returns r_t.

1.3 Purpose of the Code and Why the Kalman Filter is Used

The Python code implements the Kalman Filter to: 1. **Estimate Log-Volatility (h_t)**: The filter processes observed data to estimate the hidden log-volatility, which determines the volatility of returns. 2. **Compute the Log-Likelihood**: The filter calculates the log-likelihood, a measure of how well the SV model fits the data, used to estimate parameters like Φ, Q, and σ.

The SV model is nonlinear because of the exponential term in $r_t = \sigma \cdot \exp(0.5 \cdot h_t) \cdot \xi_t$. To apply the Kalman Filter, which requires a linear state-space model, the code uses the *Harvey Transform*. This transforms the returns into log-squared returns:

$$y_t = \log(r_t^2) = \log(\sigma^2 \cdot \exp(h_t) \cdot \xi_t^2) = \log(\sigma^2) + h_t + \log(\xi_t^2)$$

Since $\xi_t \sim N(0,1)$, $\xi_t^2 \sim \chi^2(1)$, and $\log(\xi_t^2)$ follows a log-chi-squared distribution with:

- Mean: $\mu_{\log \chi^2} \approx -1.27$.
- Variance: $\sigma^2_{\log \chi^2} \approx \pi^2/2 \approx 4.93$.

This allows the measurement equation to be written as:

Measurement Equation

$$y_t = a + B \cdot h_t + \epsilon_t, \quad \epsilon_t \sim N(0, H)$$

- a: Intercept, typically $\log(\sigma^2) + \mu_{\log \chi^2}$.
- B: Coefficient, usually 1.
- ϵ_t: Noise term, $\log(\xi_t^2) - \mu_{\log \chi^2}$, with variance $H \approx 4.93$.

The Kalman Filter processes y_t to estimate h_t and computes the log-likelihood to optimize the model parameters, ensuring the model accurately describes the data.

1.4 Mathematical Components of the Kalman Filter

The Kalman Filter operates on the state-space model defined above, iterating through:

- **Prediction Step**:
  - State prediction:
    $$a_{t|t-1} = \Phi \cdot a_{t-1|t-1}$$
    where $a_{t|t-1}$ is the prior state mean (predicted log-volatility).
  - Variance prediction:
    $$P_{t|t-1} = \Phi^2 \cdot P_{t-1|t-1} + Q$$
    where $P_{t|t-1}$ is the prior state variance (uncertainty).
- **Update Step**:
  - Prediction error:
    $$v_t = y_t - (a + B \cdot a_{t|t-1})$$
  - Prediction error variance:
    $$F_t = B \cdot P_{t|t-1} \cdot B + H$$
  - Kalman gain:
    $$K_t = \frac{P_{t|t-1} \cdot B}{F_t}$$
  - State update:
    $$a_{t|t} = a_{t|t-1} + K_t \cdot v_t$$
  - Variance update:
    $$P_{t|t} = P_{t|t-1} \cdot (1 - K_t \cdot B)$$
- **Log-Likelihood**:

$$\log L_t = -0.5 \cdot \left(\log(2\pi) + \log(F_t) + \frac{v_t^2}{F_t} \right)$$

$$\log L = \sum_{t=1}^{T} \log L_t$$

Example: Suppose $\Phi = 0.95$, $Q = 0.04$, $a = -1.27$, $B = 1$, $H = 4.93$, initial state $a_{1|0} = 0$, $P_{1|0} = \frac{0.04}{1-0.95^2} \approx 0.421$, and observation $y_1 = -2.0$. The filter computes:

- $F_1 = 1 \cdot 0.421 \cdot 1 + 4.93 = 5.351$
- $K_1 = \frac{0.421 \cdot 1}{5.351} \approx 0.0787$
- $v_1 = -2.0 - (-1.27 + 1 \cdot 0) = -0.73$
- $a_{1|1} = 0 + 0.0787 \cdot (-0.73) \approx -0.0574$
- $P_{1|1} = 0.421 \cdot (1 - 0.0787 \cdot 1) \approx 0.3879$
- $\log L_1 \approx -0.5 \cdot (1.837 + 1.677 + 0.099) \approx -1.92$

2 Kalman Filter Implementation in Python

The Python function `kalman_filter` implements the Kalman Filter for the SV model, processing log-squared returns (y_t) to estimate log-volatility (h_t) and compute the log-likelihood.

2.1 Code Listing

```python
def kalman_filter(y, a, B, Phi, H, Q, a1, P1, return_loglike=False):
    """
    Kalman Filter for an SV model.

    Parameters
    ----------
    y : array-like of shape (T,)
        Observations of the log-squared-return series
    a : float
        Intercept in the measurement equation
    B : float
        Coefficient on the state in the measurement equation
    Phi : float
        Coefficient on the state in the transition equation
    H : float
        Measurement error variance
    Q : float
        State innovation variance
    a1 : float
        Mean of the initial state
    P1 : float
        Variance of the initial state
    return_loglike : boolean, optional
        If True, returns the negative log-likelihood

    Returns
    -------
    If return_loglike:
        negative log-likelihood
    else:
        a_post, P_post, a_prior, P_prior arrays
    """
    Tsize = len(y)
    a_prior = np.zeros(Tsize)
    P_prior = np.zeros(Tsize)
    a_post  = np.zeros(Tsize)
    P_post  = np.zeros(Tsize)

    a_prior[0] = a1
    P_prior[0] = P1

    log_like = 0.0

    for t in range(Tsize):
        # Measurement update
        F_t = B * P_prior[t] * B + H  # Prediction error variance
        K_t = (P_prior[t] * B) / F_t  # Kalman gain
        v_t = y[t] - (a + B*a_prior[t])  # Prediction error

        # Update log-likelihood
        log_like += -0.5 * (np.log(2*np.pi) + np.log(F_t) + (v_t
           **2)/F_t)

        # Update a_post[t] and P_post[t]
        a_post[t] = a_prior[t] + K_t * v_t
        P_post[t] = P_prior[t] * (1 - K_t * B)

        # Time update (except final step)
        if t < Tsize - 1:
            a_prior[t+1] = Phi * a_post[t]
            P_prior[t+1] = (Phi**2) * P_post[t] + Q

    if return_loglike:
        return -log_like
    else:
        return a_post, P_post, a_prior, P_prior
```

2.2 Code Explanation

1. `def kalman_filter(y, a, B, Phi, H, Q, a1, P1, return_loglike=False):`
   - Defines the Kalman Filter function for the SV model.
   - **Inputs**:
     - y: Array of log-squared returns ($y_t = \log(r_t^2)$).

  - `a`: Intercept in $y_t = a + B \cdot h_t + \epsilon_t$.
  - `B`: Coefficient linking h_t to y_t.
  - `Phi`: Persistence in $h_{t+1} = \Phi \cdot h_t + \eta_t$.
  - `H`: Variance of ϵ_t.
  - `Q`: Variance of η_t.
  - `a1`: Initial state mean ($a_{1|0}$).
  - `P1`: Initial state variance ($P_{1|0}$).
  - `return_loglike`: If `True`, returns $-\log L$.
- **Equations**:

$$y_t = a + B \cdot h_t + \epsilon_t, \quad \epsilon_t \sim N(0, H)$$
$$h_{t+1} = \Phi \cdot h_t + \eta_t, \quad \eta_t \sim N(0, Q)$$

- **Example**: If y_t has 2500 returns, $a = -1.27$, $B = 1$, $\Phi = 0.95$, $H = 4.93$, $Q = 0.04$, $a1 = 0$, $P1 = 0.421$.

2. `Tsize = len(y)`
   - Computes the number of observations (T).
   - **Example**: For 2500 returns, `Tsize = 2500`.
3. `a_prior = np.zeros(Tsize)`
   - Initializes an array for prior state means ($a_{t|t-1}$).
   - **Example**: Creates $[0, 0, \ldots, 0]$ for 2500 time steps.
4. `P_prior = np.zeros(Tsize)`
   - Initializes an array for prior state variances ($P_{t|t-1}$).
   - **Example**: Will store values like 0.421.
5. `a_post = np.zeros(Tsize)`
   - Initializes an array for posterior state means ($a_{t|t}$).
   - **Example**: Will store values like -0.0574.
6. `P_post = np.zeros(Tsize)`
   - Initializes an array for posterior state variances ($P_{t|t}$).
   - **Example**: Will store values like 0.3879.
7. `a_prior[0] = a1`
   - Sets the initial prior state mean.
   - **Equation**: $a_{1|0} = a1$.
   - **Example**: If `a1 = 0`, $a_{1|0} = 0$.
8. `P_prior[0] = P1`

- Sets the initial prior state variance.
- **Equation**: $P_{1|0} = P1 = \frac{Q}{1-\Phi^2}$.
- **Example**: If $Q = 0.04$, $\Phi = 0.95$, then $P1 \approx 0.421$.

9. `log_like = 0.0`
    - Initializes the log-likelihood.
    - **Equation**: $\log L = \sum_{t=1}^{T} \log L_t$.
    - **Example**: Starts at 0, accumulates values like -1.92.
10. `for t in range(Tsize):`
    - Loops over time steps $t = 0, 1, \ldots, T-1$.
11. `F_t = B * P_prior[t] * B + H # Prediction error variance`
    - Computes:
    $$F_t = B \cdot P_{t|t-1} \cdot B + H$$
    - **Example**: If $B = 1$, $P_{t|t-1} = 0.421$, $H = 4.93$, then $F_t = 5.351$.
12. `K_t = (P_prior[t] * B) / F_t # Kalman gain`
    - Computes:
    $$K_t = \frac{P_{t|t-1} \cdot B}{F_t}$$
    - **Example**: If $P_{t|t-1} = 0.421$, $B = 1$, $F_t = 5.351$, then $K_t \approx 0.0787$.
13. `v_t = y[t] - (a + B*a_prior[t]) # Prediction error`
    - Computes:
    $$v_t = y_t - (a + B \cdot a_{t|t-1})$$
    - **Example**: If $y_t = -2.0$, $a = -1.27$, $B = 1$, $a_{t|t-1} = 0$, then $v_t = -0.73$.
14. `log_like += -0.5 * (np.log(2*np.pi) + np.log(F_t) + (v_t**2)/F_t)`
    - Updates:
    $$\log L_t = -0.5 \cdot \left(\log(2\pi) + \log(F_t) + \frac{v_t^2}{F_t} \right)$$
    - **Example**: If $F_t = 5.351$, $v_t = -0.73$, then $\log L_t \approx -1.92$.
15. `a_post[t] = a_prior[t] + K_t * v_t`
    - Computes:
    $$a_{t|t} = a_{t|t-1} + K_t \cdot v_t$$
    - **Example**: If $a_{t|t-1} = 0$, $K_t = 0.0787$, $v_t = -0.73$, then $a_{t|t} \approx -0.0574$.
16. `P_post[t] = P_prior[t] * (1 - K_t * B)`
    - Computes:
    $$P_{t|t} = P_{t|t-1} \cdot (1 - K_t \cdot B)$$

- **Example**: If $P_{t|t-1} = 0.421$, $K_t = 0.0787$, $B = 1$, then $P_{t|t} \approx 0.3879$.

17. `if t < Tsize - 1:`
    - Ensures the time update is not performed for the last time step.
18. `a_prior[t+1] = Phi * a_post[t]`
    - Computes:
    $$a_{t+1|t} = \Phi \cdot a_{t|t}$$
    - **Example**: If $\Phi = 0.95$, $a_{t|t} = -0.0574$, then $a_{t+1|t} \approx -0.0545$.
19. `P_prior[t+1] = (Phi**2) * P_post[t] + Q`
    - Computes:
    $$P_{t+1|t} = \Phi^2 \cdot P_{t|t} + Q$$
    - **Example**: If $\Phi = 0.95$, $P_{t|t} = 0.3879$, $Q = 0.04$, then $P_{t+1|t} \approx 0.3898$.
20. `if return_loglike:  return -log_like`
    - Returns the negative log-likelihood.
    - **Example**: If $\log L = -4800$, returns 4800.
21. `else:  return a_post, P_post, a_prior, P_prior`
    - Returns arrays of state estimates.
    - **Example**: Returns arrays with 2500 elements each.

3 Mean and Variance of $\log(\chi^2(1))$

3.1 Code Listing

```python
# Create a chi-squared distribution with 1 degree of freedom
chi2_1 = stats.chi2(df=1)

def integrand_mean(x):
    return np.log(x) * chi2_1.pdf(x)

mean_logchi2, _ = quad(integrand_mean, 0, np.inf)


def integrand_second_moment(x):
    return (np.log(x))**2 * chi2_1.pdf(x)

second_moment_logchi2, _ = quad(integrand_second_moment, 0, np.
    inf)


var_logchi2 = second_moment_logchi2 - mean_logchi2**2
```

3.2 Code Explanation

The code computes parameters for the measurement equation $y_t = a + h_t + \epsilon_t$, where $a = E[\log(\xi_t^2)]$, $H = \text{Var}[\log(\xi_t^2)]$, and $\xi_t^2 \sim \chi^2(1)$.

1. `chi2_1 = stats.chi2(df=1)`
   Initializes a $\chi^2(1)$ distribution with PDF:

   $$f(x) = \frac{1}{\sqrt{2\pi x}} e^{-x/2}, \quad x > 0$$

   Used to compute expectations for $\log(\xi_t^2)$.

2. `def integrand_mean(x):`
   Defines a function for the integrand of the mean.

3. ` return np.log(x) * chi2_1.pdf(x)`
   Implements the integrand $\log(x) f(x)$, where $f(x)$ is the $\chi^2(1)$ PDF.

4. `mean_logchi2, _ = quad(integrand_mean, 0, np.inf)`
   Numerically integrates to compute:

   $$E[\log(\chi^2(1))] \approx -1.2704$$

   Sets a in the measurement equation.

5. `def integrand_second_moment(x):`
   Defines a function for the second moment integrand.

6. ` return (np.log(x))**2 * chi2_1.pdf(x)`
   Implements $\log(x)^2 f(x)$, for the second moment.

7. `second_moment_logchi2, _ = quad(integrand_second_moment, 0, np.inf)`
   Integrates to compute:

   $$E[\log(\chi^2(1))^2] \approx 6.575$$

8. `var_logchi2 = second_moment_logchi2 - mean_logchi2**2`
   Computes:

   $$\text{Var}[\log(\chi^2(1))] \approx 6.575 - (-1.2704)^2 \approx 4.961$$

   Sets H, the noise variance.