JPM Practicum Project

Topic: An Arbitrage-free SVI Volatility Surface Method to Calibrate Local Volatility Based on SP500

Name：Honglin Qian

University of Illinois Urbana-Champaign

Date: 2024 – 5 – 1

Introduction: SVI

SVI (Stochastic Volatility Inspired) and SSVI (Surface Stochastic Volatility Inspired) are methods used in the financial industry for modeling volatility surfaces. They play a crucial role in options pricing and risk management.

1. SVI (Stochastic Volatility Inspired)

SVI is a model introduced by Jim Gatheral for fitting the implied volatility surface observed in the options market. Implied volatility is the market's forecast of the volatility of the underlying asset, and it varies with different strike prices and maturities. SVI uses a simple parametric formula to fit the implied volatility surface based on market data. The formula of **Raw SVI Parameterization** is typically expressed as:

$$\omega(\mathrm{k}, \aleph_R) = \mathrm{a} + \mathrm{b}\left(\rho(\mathrm{k}-\mathrm{m}) + \sqrt{(\mathrm{k}-\mathrm{m})^2 + \sigma^2}\right)$$

Where:

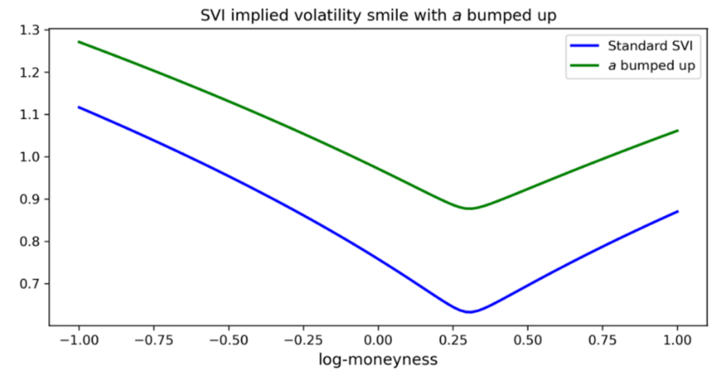
- k is the log-moneyness (logarithm of the strike price relative to the spot price).
- $\aleph_R = \{a, b, \rho, m, \sigma\}$ are parameters that need to be calibrated.
- $\omega(k, \aleph_R)$ represents total implied variance.

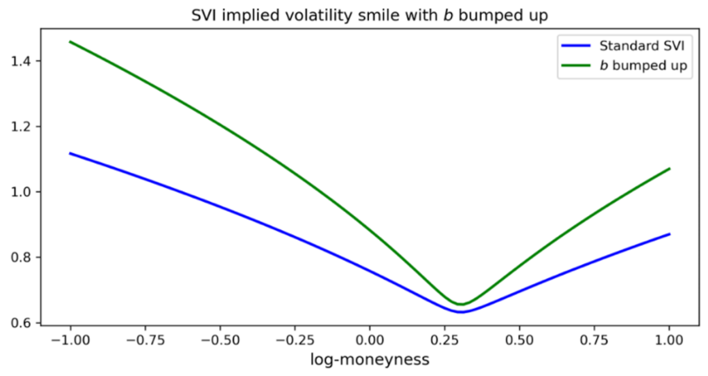
SVI is mainly used for:

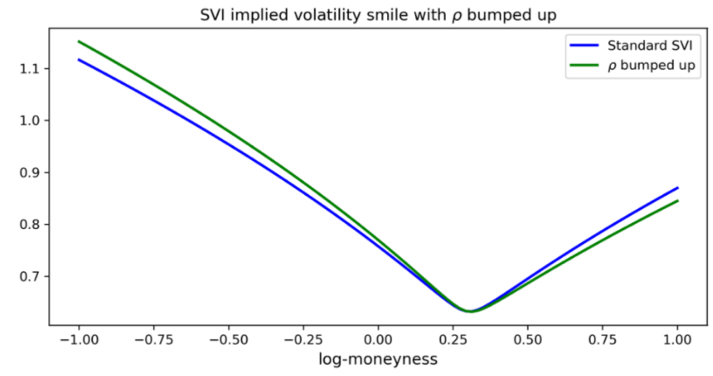
- Fitting the implied volatility surface in the options market.
- Smoothing market data through a parametric model.
- Assisting in options pricing and designing hedging strategies.

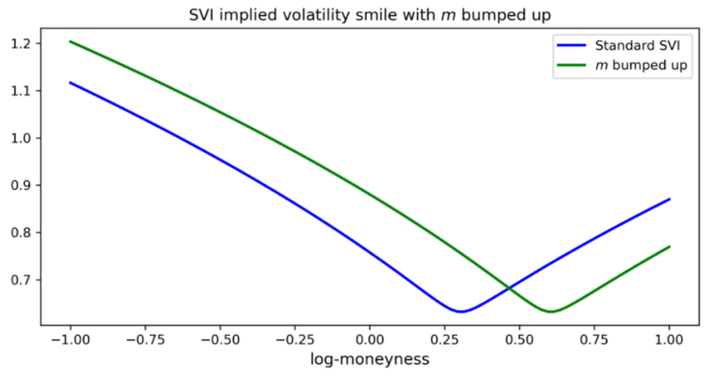
It follows immediately that changes in the parameters have the following effects:

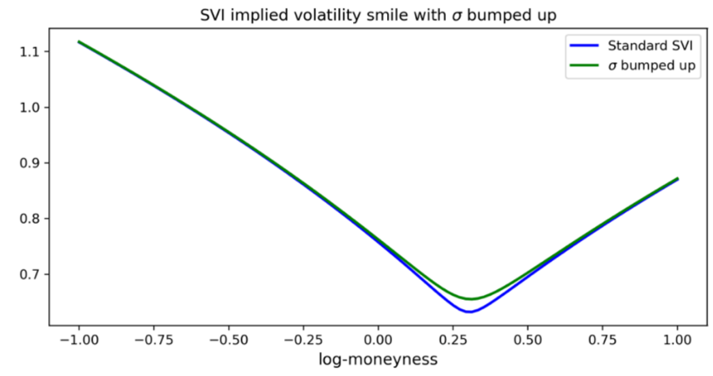
- Increasing a increases the general level of variance, a vertical translation of the smile.
- Increasing b increases the slopes of both the put and call wings, tightening the smile.
- Increasing ρ decreases the slope of the left wing, a counter clockwise rotation of the smile.
- Increasing m translates the smile to the right.
- Increasing σ reduces the at-the-money (ATM) curvature of the smile.











2. SSVI (Surface Stochastic Volatility Inspired)

SSVI is an extension of SVI that captures the dynamic evolution of the entire volatility surface. It introduces a time dimension into the SVI model to describe the time evolution of the volatility surface. SSVI is better suited to describe the behavior of implied volatility across different maturities and strike prices. The SSVI formula is an extension of **Natural SVI Parameterization** to include the time dimension, typically expressed as:

$$w\left(k,\theta_t\right)=\frac{\theta_t}{2}\left\{1+\rho\varphi\left(\theta_t\right)k+\sqrt{\left(\varphi\left(\theta_t\right)k+\rho\right)^2+\left(1-\rho^2\right)}\right\}$$

Where:

- θ_t is at-the-money implied total variance.
- $w(k,\theta_t)$ is total implied variance.

- φ is a smooth function.

SSVI is mainly used for:

- Modeling the entire volatility surface, applicable to different maturities.
- Providing a more comprehensive framework for options pricing and risk management.
- Capturing the dynamic changes and trends in implied volatility in the market.

Overall, SVI and SSVI are essential tools in financial mathematics for capturing the volatility structure in the options market, particularly in high-frequency trading and complex financial product pricing.

3. Free of Arbitrage

A volatility surface w is free of static arbitrage if and only if the following conditions are satisfied:

- It is free of calendar spread arbitrage.
- Each time slice is free of butterfly arbitrage.

Calendar Spread Arbitrage:

The volatility surface w is **free of calendar spread arbitrage** if and only if:

$$\partial_t w(k,t) \geq 0, \quad \text{for all } k \in R \text{ and } t > 0.$$

Butterfly Arbitrage:

A slice **(fixed T)** is said to be **free of butterfly arbitrage** if the corresponding density is non-negative:

$$g(k) := \left(1 - \frac{kw'(k)}{2w(k)}\right)^2 - \frac{w'(k)^2}{4}\left(\frac{1}{w(k)} + \frac{1}{4}\right) + \frac{w''(k)}{2} \geq 0$$

Since here we just calibration 2-dimensional SVI Parameterization (fixed T), we just check free of butterfly arbitrage.

Introduction : Surface SVI Parameterization

1 Natural SVI Parameterization:

For a given parameter set $\chi_N = \{\Delta, \mu, \rho, \omega, \zeta\}$, the **natural SVI parameterization** of total implied variance reads:

$$w\left(k; \chi_N\right) = \Delta + \frac{\omega}{2}\left\{1 + \zeta\rho\left(k - \mu\right) + \sqrt{\left(\zeta\left(k - \mu\right) + \rho\right)^2 + \left(1 - \rho^2\right)}\right\}$$

Where: $\omega \geq 0$, $\Delta \in R$, $\mu \in R$, $|\rho| < 1$, and $\zeta > 0$.

2 SSVI:

We now introduce a class of SVI volatility surfaces—which we shall call SSVI (for 'Surface SVI')—**as an extension of the natural SVI parameterization.**

We refer to as SSVI the surface defined by:

$$\text{total variance: } w(k, \theta_t) = \frac{\theta_t}{2}\left\{1 + \rho\varphi(\theta_t)k + \sqrt{(\varphi(\theta_t)k + \rho)^2 + (1 - \rho^2)}\right\}.$$

Where:

- $\theta_t := \sigma_{\text{BS}}^2(0, t)t$: A scalar parameter at time t, typically related to the total variance.
- $k = log(K/F)$: **The standardized strike price,** where K is the strike price and F is the forward price. This variable reflects how far the strike price K is from the forward price F.
- ρ: The correlation coefficient, which controls the **skewness** of the implied volatility curve.
- $\varphi(\theta_t)$: A function dependent on θ_t, used to adjust the **influence of volatility across different time horizons.**

SSVI corresponds to the natural SVI volatility surface parameterization with

$$\chi_N = \{0,0, \rho, \theta_t, \varphi(\theta_t)\}.$$

3 Example of $\boldsymbol{\varphi(\theta_t)}$

3.1 Heston-like parameterization:

$$\varphi(\theta) \equiv \frac{1}{\gamma\theta}\left\{1 - \frac{1 - e^{-\gamma\theta}}{\gamma\theta}\right\}$$

3.2 Power-law parameterization:

$$\varphi(\theta) = \eta\theta^{-\gamma}$$

4 Free of Arbitrage

The SSVI surface is **free of static arbitrage** if the following conditions are satisfied:

1. $\partial_t\theta_t \geq 0$, for all $t > 0$
2. $0 \leq \partial_\theta\left(\theta\varphi(\theta)\right) \leq \frac{1}{\rho^2}\left(1 + \sqrt{1 - \rho^2}\right)\varphi(\theta)$, for all $\theta > 0$
3. $\theta\varphi(\theta)(1 + |\rho|) < 4$, for all $\theta > 0$
4. $\theta\varphi(\theta)^2(1 + |\rho|) \leq 4$, for all $\theta > 0$

Where, 1 and 2 promise SSVI surface is free of calendar arbitrage, while 3 and 4 promise SSVI surface is free of butterfly arbitrage.

Local Volatility Calibration Based on SSVI

1 Math Formula:

1.1 We consider the following SSVI parameterization of the total implied variance surface:

$$total\ variance: w(k, \theta_T) = \frac{\theta_T}{2}\left\{1 + \rho\varphi(\theta_T)k + \sqrt{(\varphi(\theta_T)k + \rho)^2 + (1 - \rho^2)}\right\}$$

Where:

- θ_T: denotes the total at-the-money variance.
- $k = log\left(\frac{S}{F_T}\right)$: denotes the log-moneyness.

1.2 We consider the following smooth function:

$$Power\ Law: \varphi(\theta_T) = \frac{\eta}{\theta_T^{\gamma}}$$

1.3 Then, the local volatility function is:

$$\sigma_{loc}^2(k, T) \coloneqq \frac{\partial_T w(k, \theta_T)}{g\big(k, w(k, \theta_T)\big)} = \frac{\frac{\partial w}{\partial T}}{\left(\frac{k}{2w}\frac{\partial w}{\partial k} - 1\right)^2 + \frac{1}{2}\frac{\partial^2 w}{\partial k^2} - \frac{1}{4}\left(\frac{1}{4} + \frac{1}{w}\right)\left(\frac{\partial w}{\partial k}\right)^2}\Big|_{k=log\left(\frac{S}{F_T}\right)}$$

And

$$\theta_T = \theta \cdot T + (V - \theta)\frac{1 - e^{-\kappa_1 T}}{\kappa_1} + (V' - \theta)\frac{\kappa_1}{\kappa_1 - \kappa_2}\left(\frac{1 - e^{-\kappa_2 T}}{\kappa_2} - \frac{1 - e^{-\kappa_1 T}}{\kappa_1}\right)$$

Hence, we need to calibrate 6 parameters:

$$\boldsymbol{\rho} \quad \boldsymbol{\eta} \quad \boldsymbol{\gamma} \quad \boldsymbol{V} \quad \boldsymbol{V'} \quad \boldsymbol{\theta}$$

The parameter θ_T in this context is primarily used to describe the evolution of the at-the-money (ATM) volatility over time. It is defined using a double mean-reverting (DMR) model, which accounts for two different mean-reverting processes to describe both short-term and long-term volatility behavior. To better understand the significance of θ_T, let's break down the formula:

$$\theta_T = \theta \cdot T + (V - \theta)\frac{1 - e^{-\kappa_1 T}}{\kappa_1} + (V' - \theta)\frac{\kappa_1}{\kappa_1 - \kappa_2}\left(\frac{1 - e^{-\kappa_2 T}}{\kappa_2} - \frac{1 - e^{-\kappa_1 T}}{\kappa_1}\right)$$

Meaning of Each Term in the Formula:

1. $\theta \cdot T$:

- This is a linear term representing the ATM volatility increasing proportionally with time T. It usually captures the long-term linear trend in the volatility.

2. $(V - \theta)\frac{1-e^{-\kappa_1 T}}{\kappa_1}$:

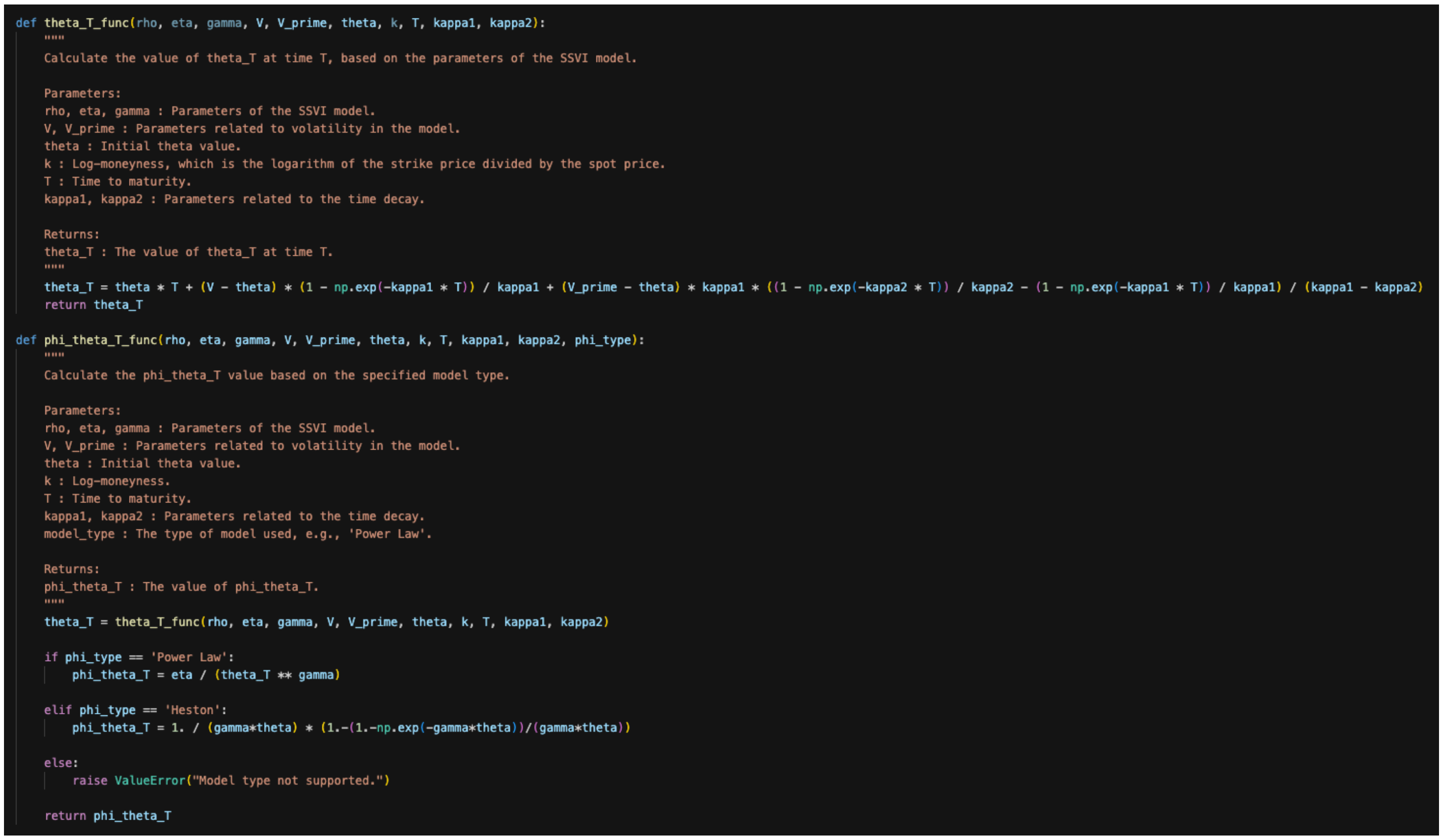
- This is the first mean-reverting term, where V represents the short-term volatility level, and θ is the long-term mean of the ATM volatility.
- κ_1 is the **mean-reversion speed for short-term** volatility, dictating how quickly the short-term volatility reverts to the long-term mean θ.
- The factor $\frac{1-e^{-\kappa_1 T}}{\kappa_1}$ is an adjustment that stabilizes over time, indicating the process of volatility reverting from its initial level towards its long-term average.

3. $(V' - \theta)\frac{\kappa_1}{\kappa_1-\kappa_2}\left(\frac{1-e^{-\kappa_2 T}}{\kappa_2} - \frac{1-e^{-\kappa_1 T}}{\kappa_1}\right)$:

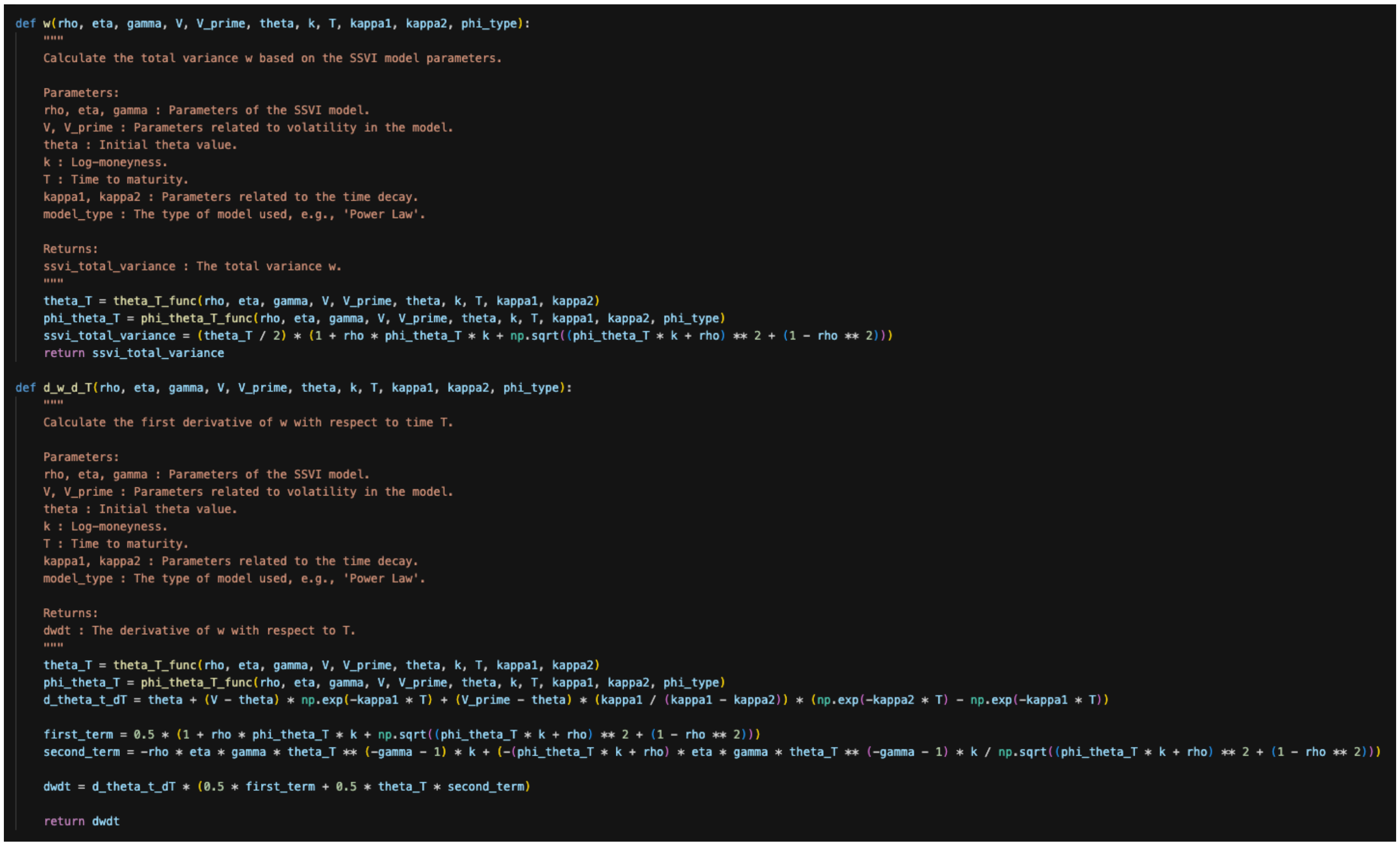
- This is the second mean-reverting term, where V' is the long-term volatility level.
- κ_2 is the **mean-reversion speed for long-term** volatility, like κ_1 but acting over a longer time horizon.
- This term is more complex and accounts for the transition of volatility from the short-term level V to the long-term level V', considering both mean-reversion speeds κ_1 and κ_2.

Overall Significance of θ_T:

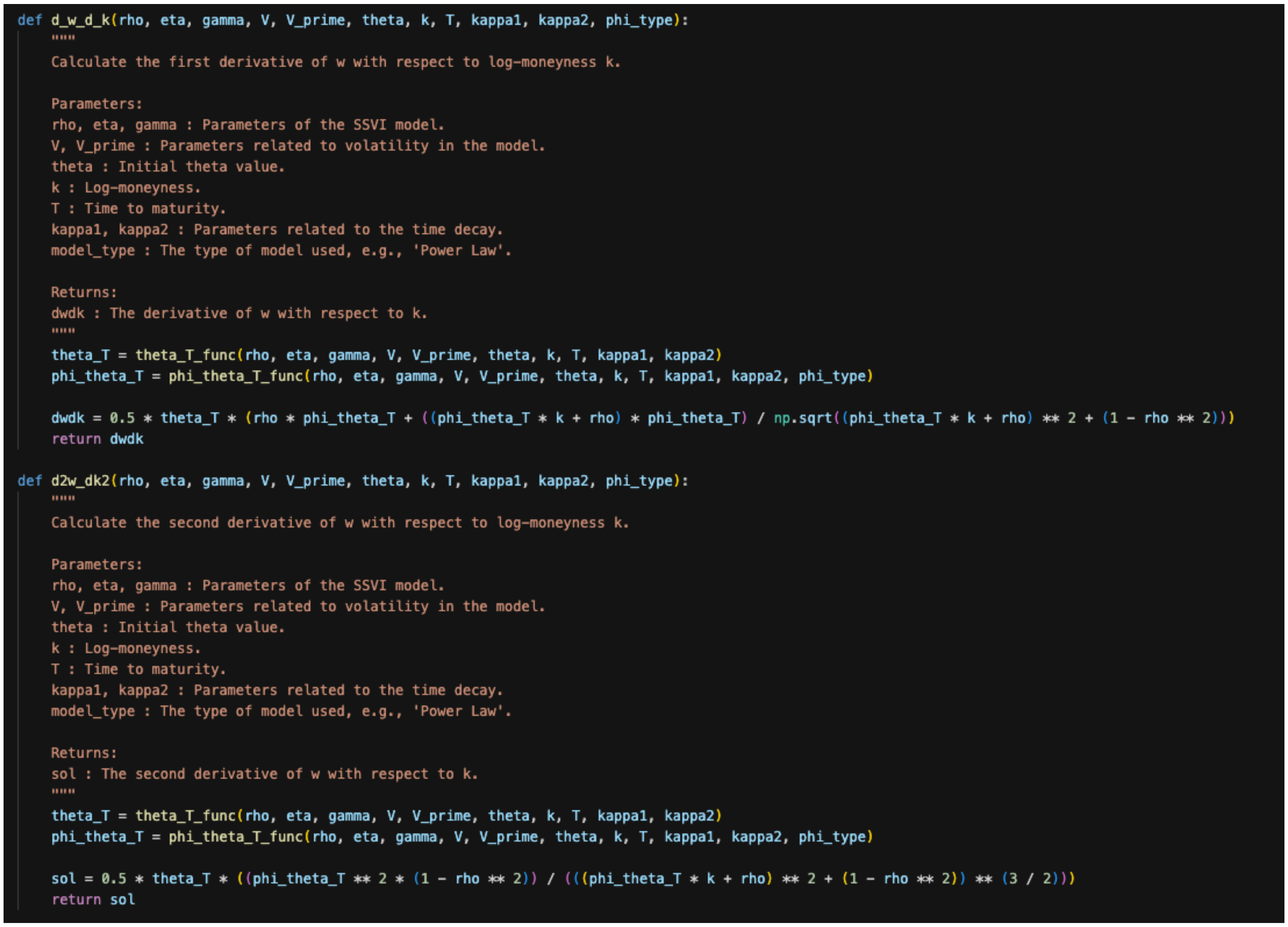
θ_T represents the evolution of ATM volatility over time, incorporating both short-term and long-term characteristics of volatility. Through this formula, θ_T captures how volatility gradually reverts from its initial level to its long-term mean over different time scales. The mean-reversion speeds κ_1 and κ_2 determine how quickly this process occurs.



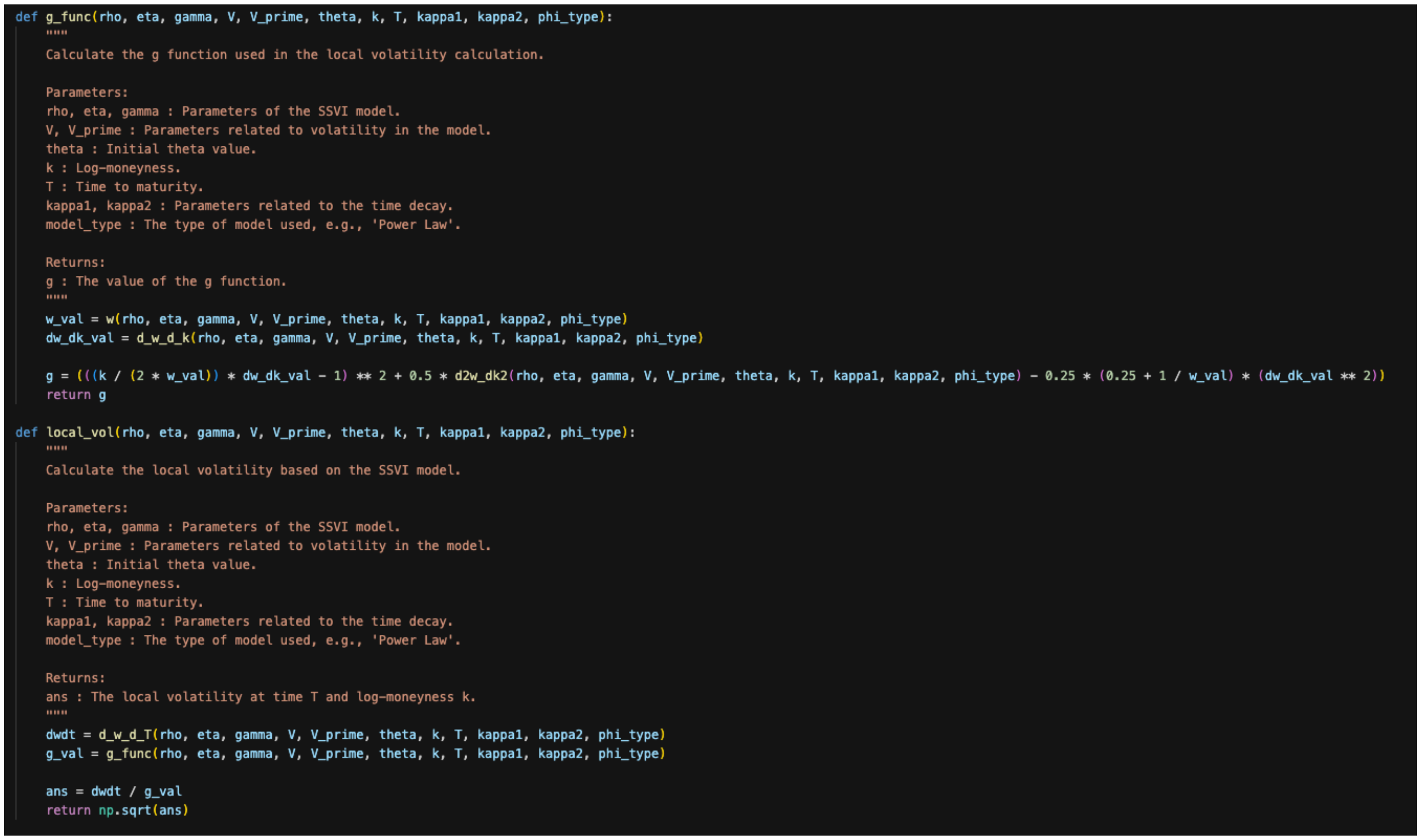
```python
def theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2):
    """
    Calculate the value of theta_T at time T, based on the parameters of the SSVI model.

    Parameters:
    rho, eta, gamma : Parameters of the SSVI model.
    V, V_prime : Parameters related to volatility in the model.
    theta : Initial theta value.
    k : Log-moneyness, which is the logarithm of the strike price divided by the spot price.
    T : Time to maturity.
    kappa1, kappa2 : Parameters related to the time decay.

    Returns:
    theta_T : The value of theta_T at time T.
    """
    theta_T = theta * T + (V - theta) * (1 - np.exp(-kappa1 * T)) / kappa1 + (V_prime - theta) * kappa1 * ((1 - np.exp(-kappa2 * T)) / kappa2 - (1 - np.exp(-kappa1 * T)) / kappa1) / (kappa1 - kappa2)
    return theta_T

def phi_theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type):
    """
    Calculate the phi_theta_T value based on the specified model type.

    Parameters:
    rho, eta, gamma : Parameters of the SSVI model.
    V, V_prime : Parameters related to volatility in the model.
    theta : Initial theta value.
    k : Log-moneyness.
    T : Time to maturity.
    kappa1, kappa2 : Parameters related to the time decay.
    model_type : The type of model used, e.g., 'Power Law'.

    Returns:
    phi_theta_T : The value of phi_theta_T.
    """
    theta_T = theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2)

    if phi_type == 'Power Law':
        phi_theta_T = eta / (theta_T ** gamma)

    elif phi_type == 'Heston':
        phi_theta_T = 1. / (gamma*theta) * (1.-(1.-np.exp(-gamma*theta))/(gamma*theta))

    else:
        raise ValueError("Model type not supported.")

    return phi_theta_T
```



```python
def w(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type):
    """
    Calculate the total variance w based on the SSVI model parameters.

    Parameters:
    rho, eta, gamma : Parameters of the SSVI model.
    V, V_prime : Parameters related to volatility in the model.
    theta : Initial theta value.
    k : Log-moneyness.
    T : Time to maturity.
    kappa1, kappa2 : Parameters related to the time decay.
    model_type : The type of model used, e.g., 'Power Law'.

    Returns:
    ssvi_total_variance : The total variance w.
    """
    theta_T = theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2)
    phi_theta_T = phi_theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type)
    ssvi_total_variance = (theta_T / 2) * (1 + rho * phi_theta_T * k + np.sqrt((phi_theta_T * k + rho) ** 2 + (1 - rho ** 2)))
    return ssvi_total_variance

def d_w_d_T(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type):
    """
    Calculate the first derivative of w with respect to time T.

    Parameters:
    rho, eta, gamma : Parameters of the SSVI model.
    V, V_prime : Parameters related to volatility in the model.
    theta : Initial theta value.
    k : Log-moneyness.
    T : Time to maturity.
    kappa1, kappa2 : Parameters related to the time decay.
    model_type : The type of model used, e.g., 'Power Law'.

    Returns:
    dwdt : The derivative of w with respect to T.
    """
    theta_T = theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2)
    phi_theta_T = phi_theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type)
    d_theta_t_dT = theta + (V - theta) * np.exp(-kappa1 * T) + (V_prime - theta) * (kappa1 / (kappa1 - kappa2)) * (np.exp(-kappa2 * T) - np.exp(-kappa1 * T))

    first_term = 0.5 * (1 + rho * phi_theta_T * k + np.sqrt((phi_theta_T * k + rho) ** 2 + (1 - rho ** 2)))
    second_term = -rho * eta * gamma * theta_T ** (-gamma - 1) * k + (-(phi_theta_T * k + rho) * eta * gamma * theta_T ** (-gamma - 1) * k / np.sqrt((phi_theta_T * k + rho) ** 2 + (1 - rho ** 2)))

    dwdt = d_theta_t_dT * (0.5 * first_term + 0.5 * theta_T * second_term)

    return dwdt
```



```python
def d_w_d_k(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type):
    """
    Calculate the first derivative of w with respect to log-moneyness k.

    Parameters:
    rho, eta, gamma : Parameters of the SSVI model.
    V, V_prime : Parameters related to volatility in the model.
    theta : Initial theta value.
    k : Log-moneyness.
    T : Time to maturity.
    kappa1, kappa2 : Parameters related to the time decay.
    model_type : The type of model used, e.g., 'Power Law'.

    Returns:
    dwdk : The derivative of w with respect to k.
    """
    theta_T = theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2)
    phi_theta_T = phi_theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type)

    dwdk = 0.5 * theta_T * (rho * phi_theta_T + ((phi_theta_T * k + rho) * phi_theta_T) / np.sqrt((phi_theta_T * k + rho) ** 2 + (1 - rho ** 2)))
    return dwdk

def d2w_dk2(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type):
    """
    Calculate the second derivative of w with respect to log-moneyness k.

    Parameters:
    rho, eta, gamma : Parameters of the SSVI model.
    V, V_prime : Parameters related to volatility in the model.
    theta : Initial theta value.
    k : Log-moneyness.
    T : Time to maturity.
    kappa1, kappa2 : Parameters related to the time decay.
    model_type : The type of model used, e.g., 'Power Law'.

    Returns:
    sol : The second derivative of w with respect to k.
    """
    theta_T = theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2)
    phi_theta_T = phi_theta_T_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type)

    sol = 0.5 * theta_T * ((phi_theta_T ** 2 * (1 - rho ** 2)) / (((phi_theta_T * k + rho) ** 2 + (1 - rho ** 2)) ** (3 / 2)))
    return sol
```



```python
def g_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type):
    """
    Calculate the g function used in the local volatility calculation.

    Parameters:
    rho, eta, gamma : Parameters of the SSVI model.
    V, V_prime : Parameters related to volatility in the model.
    theta : Initial theta value.
    k : Log-moneyness.
    T : Time to maturity.
    kappa1, kappa2 : Parameters related to the time decay.
    model_type : The type of model used, e.g., 'Power Law'.

    Returns:
    g : The value of the g function.
    """
    w_val = w(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type)
    dw_dk_val = d_w_d_k(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type)

    g = (((k / (2 * w_val)) * dw_dk_val - 1) ** 2 + 0.5 * d2w_dk2(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type) - 0.25 * (0.25 + 1 / w_val) * (dw_dk_val ** 2))
    return g

def local_vol(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type):
    """
    Calculate the local volatility based on the SSVI model.

    Parameters:
    rho, eta, gamma : Parameters of the SSVI model.
    V, V_prime : Parameters related to volatility in the model.
    theta : Initial theta value.
    k : Log-moneyness.
    T : Time to maturity.
    kappa1, kappa2 : Parameters related to the time decay.
    model_type : The type of model used, e.g., 'Power Law'.

    Returns:
    ans : The local volatility at time T and log-moneyness k.
    """
    dwdt = d_w_d_T(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type)
    g_val = g_func(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type)

    ans = dwdt / g_val
    return np.sqrt(ans)
```

2 Parameter Calibration:

Let's make two direct calibrations (no quasi-explicit reparameterization). One with **Brute Force** global optimization and a refinement with a simplex algorithm, and the second through a **Differential Evolution - DE**, optimization refined by a L-BFGS-B method.

Objective Function:

$$\text{WRMSE} = \sqrt{\frac{\sum_{i=1}^{n} w_i \cdot (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} w_i}}$$

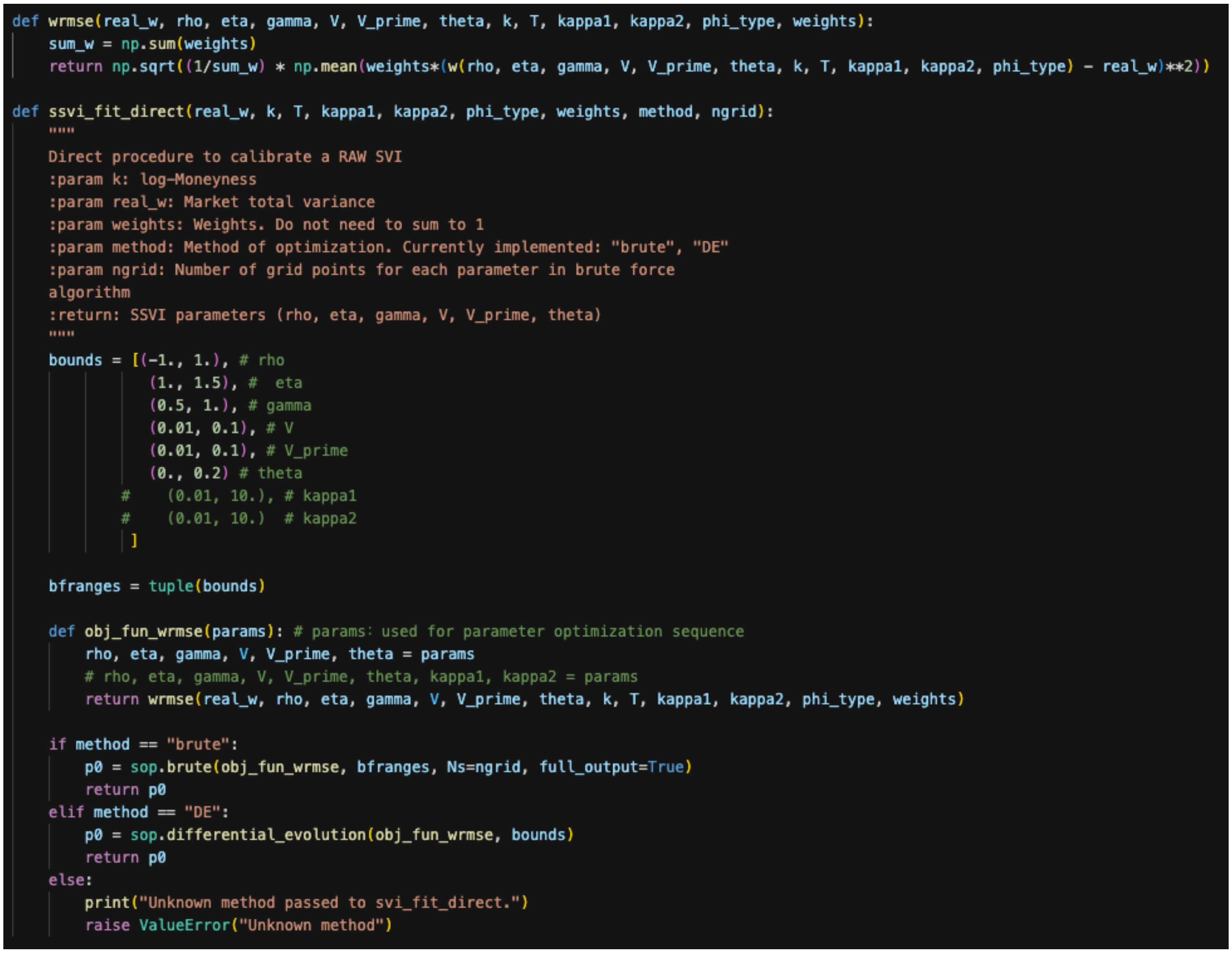
1. Brute-Force Search (Method = "brute")

- **Principle:** Brute-force search is an exhaustive search method that evaluates the objective function at a grid of points within a defined range for each parameter. It is a simple, yet computationally expensive method because it checks every possible solution within the grid.
- **Mathematical Operation:**
    - The “brute( )” function systematically varies each parameter within its specified range (“bfranges”), with the number of grid points controlled by “ngrid”.
    - It evaluates the objective function “obj_fun_wrmse” at each point in the grid.
    - The method returns the point where the objective function reaches its minimum value, which is the optimal solution.
- **Usage:** This method is best when the search space is small and well-defined, and when the function has no local minima that could trap other optimization methods.

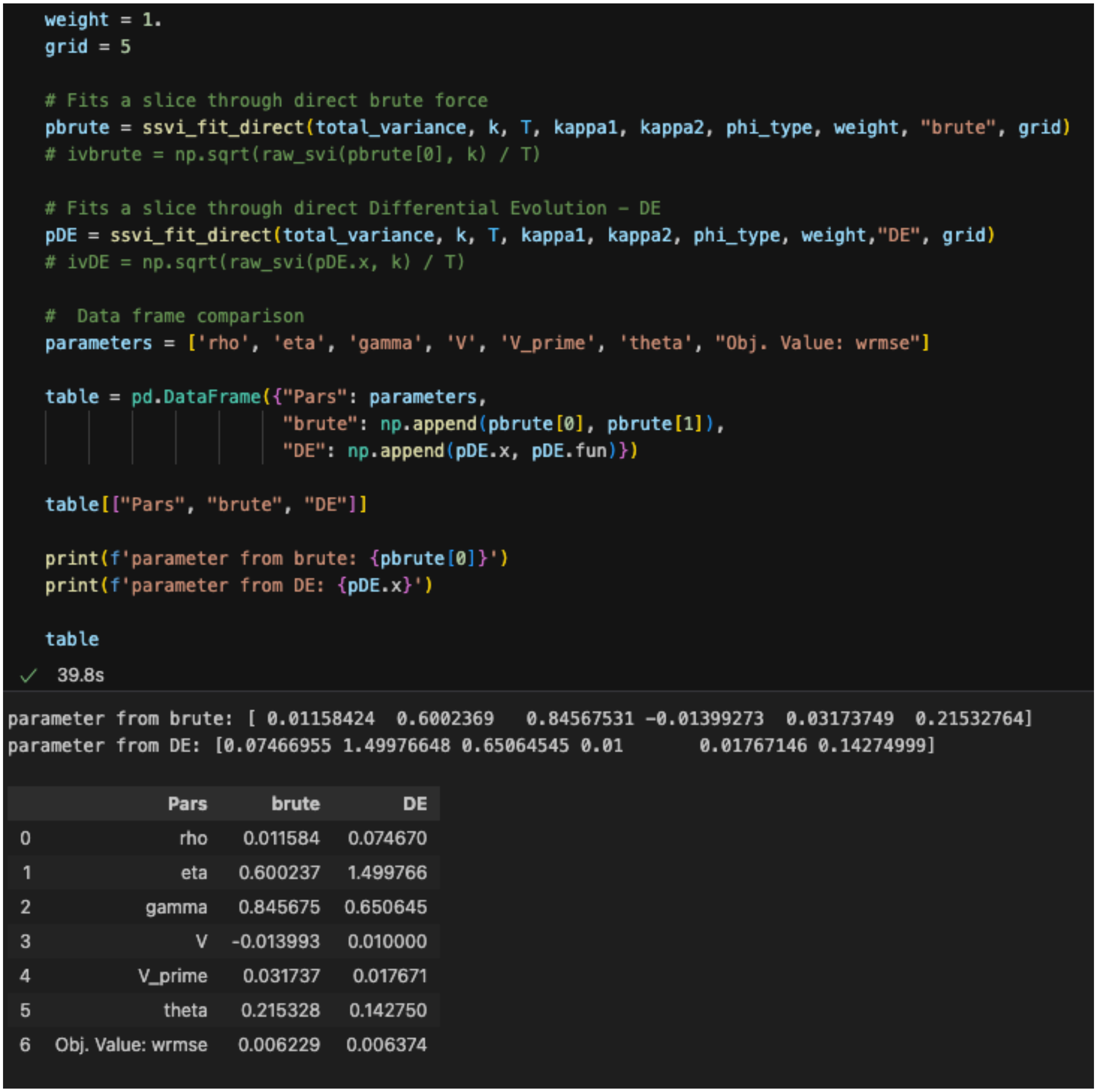
2. Differential Evolution (Method = "DE")

- **Principle:** Differential evolution (DE) is a stochastic population-based optimization method. It is a type of genetic algorithm that uses concepts of mutation, crossover, and selection to iteratively improve a population of candidate solutions.
- **Mathematical Operation:**
    - **Population Initialization:** A population of candidate solutions is randomly initialized within the bounds (bounds).
    - **Mutation:** For each candidate, a new candidate (mutant vector) is generated by adding the weighted difference between two randomly selected population members to a third one.
    - **Crossover:** The mutant vector is mixed with the original candidate vector to create a trial vector.

  - **Selection:** If the trial vector yields a lower objective function value than the original candidate, it replaces the original candidate.
  - This process is repeated for a specified number of generations or until a convergence criterion is met.
- **Usage:** Differential evolution is particularly useful for problems with complex landscapes, such as those with many local minima, because it is less likely to get trapped in a local minimum compared to traditional gradient-based methods.



```python
def wrmse(real_w, rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type, weights):
    sum_w = np.sum(weights)
    return np.sqrt((1/sum_w) * np.mean(weights*(w(rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type) - real_w)**2))

def ssvi_fit_direct(real_w, k, T, kappa1, kappa2, phi_type, weights, method, ngrid):
    """
    Direct procedure to calibrate a RAW SVI
    :param k: log-Moneyness
    :param real_w: Market total variance
    :param weights: Weights. Do not need to sum to 1
    :param method: Method of optimization. Currently implemented: "brute", "DE"
    :param ngrid: Number of grid points for each parameter in brute force
    algorithm
    :return: SSVI parameters (rho, eta, gamma, V, V_prime, theta)
    """
    bounds = [(-1., 1.), # rho
                 (1., 1.5), #  eta
                 (0.5, 1.), # gamma
                 (0.01, 0.1), # V
                 (0.01, 0.1), # V_prime
                 (0., 0.2) # theta
            #    (0.01, 10.), # kappa1
            #    (0.01, 10.)  # kappa2
             ]

    bfranges = tuple(bounds)

    def obj_fun_wrmse(params): # params: used for parameter optimization sequence
        rho, eta, gamma, V, V_prime, theta = params
        # rho, eta, gamma, V, V_prime, theta, kappa1, kappa2 = params
        return wrmse(real_w, rho, eta, gamma, V, V_prime, theta, k, T, kappa1, kappa2, phi_type, weights)

    if method == "brute":
        p0 = sop.brute(obj_fun_wrmse, bfranges, Ns=ngrid, full_output=True)
        return p0
    elif method == "DE":
        p0 = sop.differential_evolution(obj_fun_wrmse, bounds)
        return p0
    else:
        print("Unknown method passed to svi_fit_direct.")
        raise ValueError("Unknown method")
```



```python
weight = 1.
grid = 5

# Fits a slice through direct brute force
pbrute = ssvi_fit_direct(total_variance, k, T, kappa1, kappa2, phi_type, weight, "brute", grid)
# ivbrute = np.sqrt(raw_svi(pbrute[0], k) / T)

# Fits a slice through direct Differential Evolution - DE
pDE = ssvi_fit_direct(total_variance, k, T, kappa1, kappa2, phi_type, weight,"DE", grid)
# ivDE = np.sqrt(raw_svi(pDE.x, k) / T)

#  Data frame comparison
parameters = ['rho', 'eta', 'gamma', 'V', 'V_prime', 'theta', "Obj. Value: wrmse"]

table = pd.DataFrame({"Pars": parameters,
                      "brute": np.append(pbrute[0], pbrute[1]),
                      "DE": np.append(pDE.x, pDE.fun)})

table[["Pars", "brute", "DE"]]

print(f'parameter from brute: {pbrute[0]}')
print(f'parameter from DE: {pDE.x}')

table
```

✓ 39.8s

```
parameter from brute: [ 0.01158424  0.6002369   0.84567531 -0.01399273  0.03173749  0.21532764]
parameter from DE: [0.07466955 1.49976648 0.65064545 0.01       0.01767146 0.14274999]
```

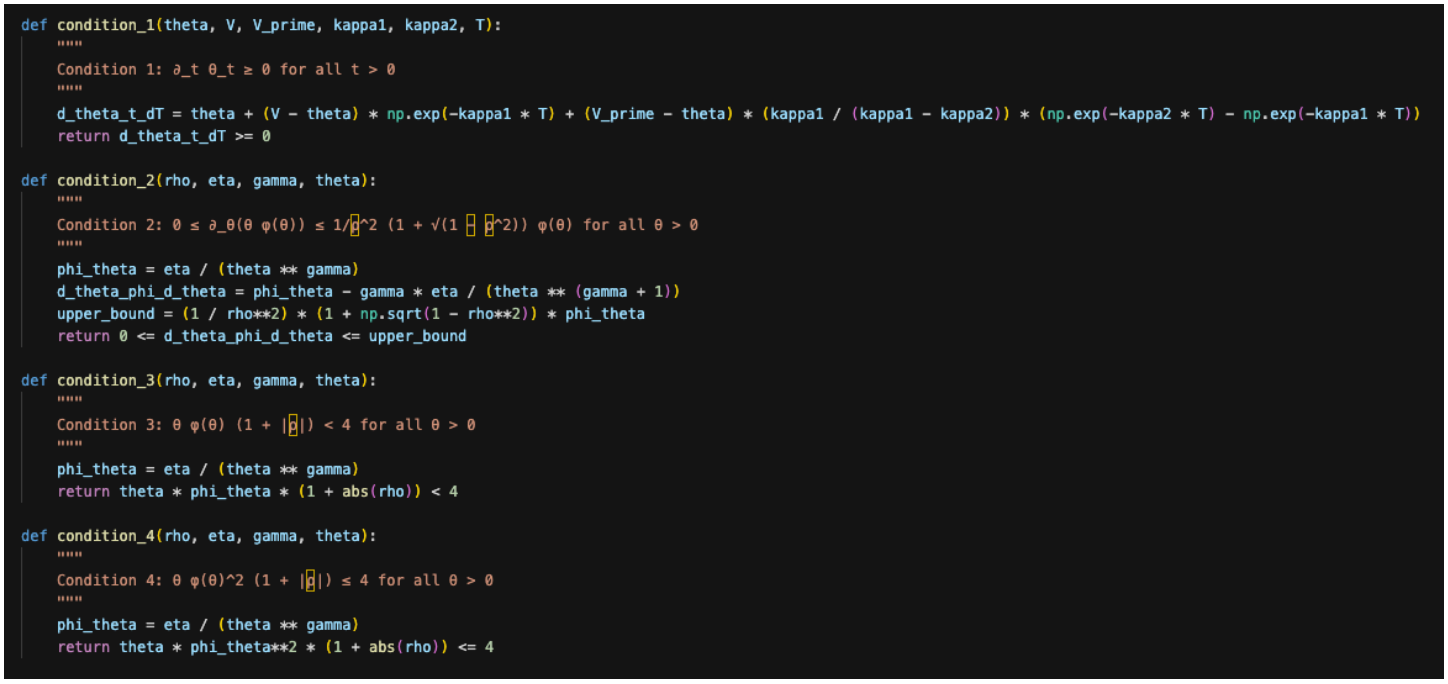
	Pars	brute	DE
0	rho	0.011584	0.074670
1	eta	0.600237	1.499766
2	gamma	0.845675	0.650645
3	V	-0.013993	0.010000
4	V_prime	0.031737	0.017671
5	theta	0.215328	0.142750
6	Obj. Value: wrmse	0.006229	0.006374

From table, we can see that there is no big difference between Brute method and DE method. Since DE method provide a faster calibration, we can utilize **Differential Evolution** method as our parameter calibration method.

2 Free of Arbitrage Checking:

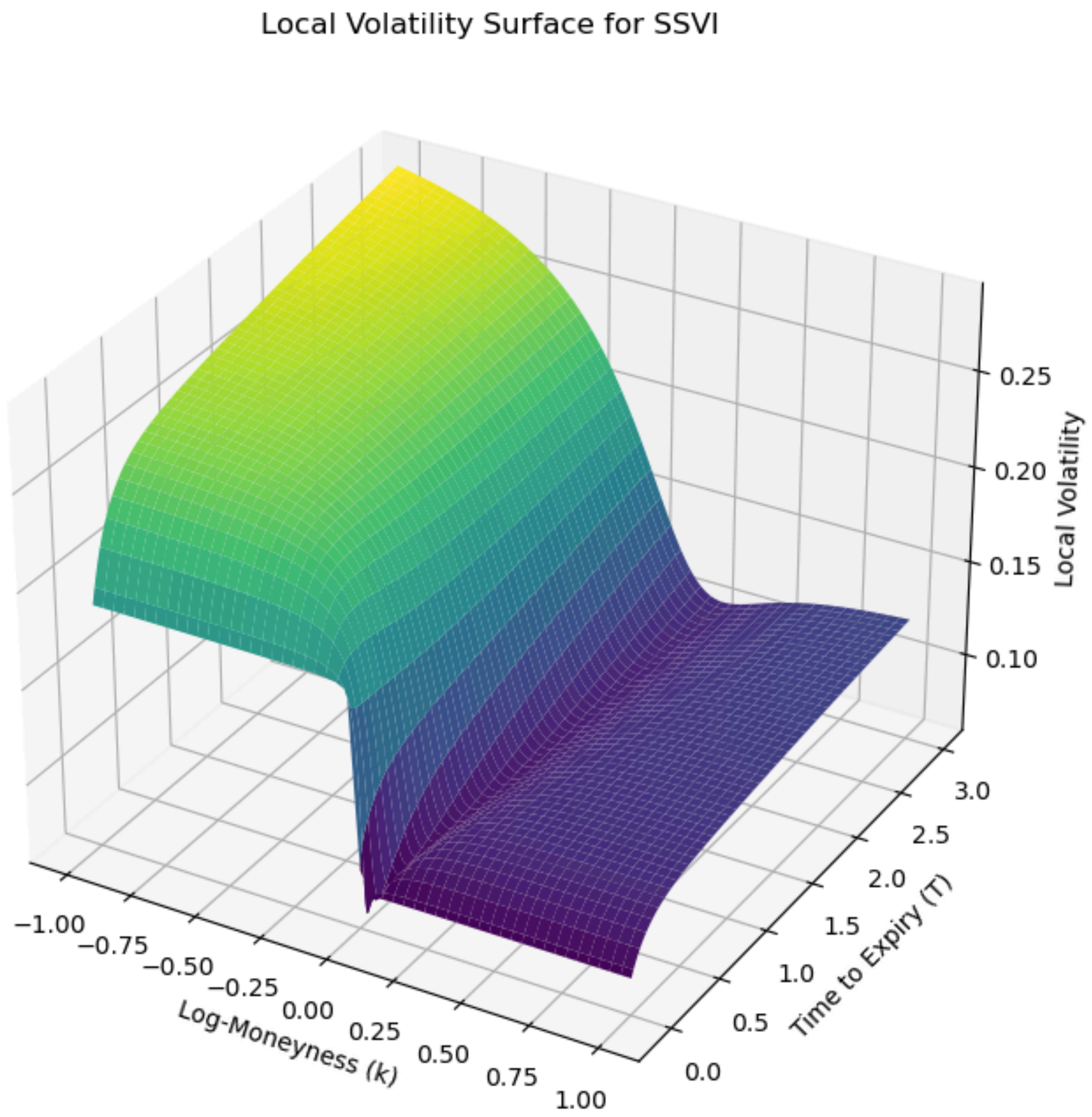
The SSVI surface is **free of static arbitrage** if the following conditions are satisfied:

1. $\partial_t \theta_t \geq 0$, for all $t > 0$
2. $0 \leq \partial_\theta\big(\theta\varphi(\theta)\big) \leq \frac{1}{\rho^2}\left(1 + \sqrt{1 - \rho^2}\right)\varphi(\theta)$, for all $\theta > 0$
3. $\theta\varphi(\theta)(1 + |\rho|) < 4$, for all $\theta > 0$
4. $\theta\varphi(\theta)^2(1 + |\rho|) \leq 4$, for all $\theta > 0$



```python
def condition_1(theta, V, V_prime, kappa1, kappa2, T):
    """
    Condition 1: ∂_t θ_t ≥ 0 for all t > 0
    """
    d_theta_t_dT = theta + (V - theta) * np.exp(-kappa1 * T) + (V_prime - theta) * (kappa1 / (kappa1 - kappa2)) * (np.exp(-kappa2 * T) - np.exp(-kappa1 * T))
    return d_theta_t_dT >= 0

def condition_2(rho, eta, gamma, theta):
    """
    Condition 2: 0 ≤ ∂_θ(θ φ(θ)) ≤ 1/ρ^2 (1 + √(1 − ρ^2)) φ(θ) for all θ > 0
    """
    phi_theta = eta / (theta ** gamma)
    d_theta_phi_d_theta = phi_theta - gamma * eta / (theta ** (gamma + 1))
    upper_bound = (1 / rho**2) * (1 + np.sqrt(1 - rho**2)) * phi_theta
    return 0 <= d_theta_phi_d_theta <= upper_bound

def condition_3(rho, eta, gamma, theta):
    """
    Condition 3: θ φ(θ) (1 + |ρ|) < 4 for all θ > 0
    """
    phi_theta = eta / (theta ** gamma)
    return theta * phi_theta * (1 + abs(rho)) < 4

def condition_4(rho, eta, gamma, theta):
    """
    Condition 4: θ φ(θ)^2 (1 + |ρ|) ≤ 4 for all θ > 0
    """
    phi_theta = eta / (theta ** gamma)
    return theta * phi_theta**2 * (1 + abs(rho)) <= 4
```

3 Local Volatility Surface for SSVI:


Local Volatility Surface for SSVI
Local Volatility
Log-Moneyness (k)
Time to Expiry (T)