Documentation: Pricing, Filtering, Weighting and Calibration for Deribit Inverse Options

(Module documentation for `calibration.py` and `inverse_fft_pricer.py`)

Abstract

This document describes, in mathematical detail, the pricing and calibration pipeline implemented in the modules `inverse_fft_pricer.py` and `calibration.py`. The pipeline ingests Deribit option snapshot data, filters for liquid quotes, builds weighted price residuals, and calibrates one of three models (Black, Heston, SVCJ) by solving a nonlinear least-squares problem where model prices are produced via the Carr–Madan FFT. Function names are referenced exactly as implemented.

Contents

1 Market objects and notation

1.1 Deribit snapshot observations

Fix a snapshot time t. Each row $i \in \{1, \dots, n\}$ corresponds to one listed European option instrument with:

- strike $K_i > 0$ in USD (column `strike`);
- time to maturity $T_i > 0$ in years (column `time_to_maturity`);
- option type $\tau_i \in \{\text{call}, \text{put}\}$ (column `option_type`);
- term futures mark F_i in USD per coin (column `futures_price`);
- bid/ask premium in coin units (columns `bid_price`, `ask_price`);
- liquidity proxies: vega V_i (column `vega`) and open interest OI_i (column `open_interest`).

The module `filter_liquid_options` constructs additional columns:

$$m_i := \tfrac{1}{2}(\text{bid}_i + \text{ask}_i) \quad (\texttt{mid\_price\_clean}), \tag{1}$$

$$s_i := \text{ask}_i - \text{bid}_i \quad (\texttt{spread}), \tag{2}$$

$$\text{rs}_i := \frac{s_i}{\max(m_i, \varepsilon)} \quad (\texttt{rel\_spread}), \tag{3}$$

with a small numerical $\varepsilon > 0$.

1.2 Per-expiry forward proxy and moneyness

Options are grouped by the expiry key `expiry_datetime`. Within each expiry group g, `filter_liquid_options` defines a single forward proxy ("per-expiry median future"):

$$F_{0,g} \;:=\; \text{median}\{F_i : i \in g\}, \tag{4}$$

stored as column `F0`. For each row $i \in g$ the module computes

$$\text{moneyness:} \quad M_i := \frac{K_i}{F_{0,g}} \quad (\texttt{moneyness}), \qquad \text{log-moneyness:} \quad \kappa_i := \log\Big(\frac{K_i}{F_{0,g}}\Big) \quad (\texttt{log\_moneyness}). \tag{5}$$

2 Inverse option pricing via Carr–Madan FFT

2.1 Characteristic functions

Let F_t denote the term futures price (USD per coin) and define

$$X_T := \log F_T. \tag{6}$$

For each model, the pricer assumes that the characteristic function

$$\phi_T(u) := \mathbb{E}\big[e^{\mathrm{i}uX_T}\big], \qquad u \in \mathbb{C}, \tag{7}$$

is available in closed (or semi-closed) form and is evaluable for complex arguments. In `inverse_fft_pricer.py` these are implemented as:

`cf_black(u,T,F0,sigma)`, `cf_heston(u,T,F0,kappa,theta,sigma_v,rho,v0)`, `cf_svcj(u,T,F0,kappa,theta,sigma_v,rho,v0,lam,ell_y,sigma_y,ell_v,rho_j)`.

All three functions return $\phi_T(u)$ for vectorized complex u.

2.2 Carr–Madan transform and damped call prices

Let $C^{\mathrm{USD}}(K)$ denote the (undiscounted) European call price in USD on the futures. Following Carr–Madan, define log-strike $k := \log K$ and the damped call transform

$$c_\alpha(k) := e^{\alpha k} C^{\mathrm{USD}}(e^k), \qquad \alpha > 0. \tag{8}$$

Under mild integrability conditions (existence of exponential moments), c_α admits the Fourier inversion

$$C^{\mathrm{USD}}(K) = \frac{e^{-\alpha k}}{\pi} \int_0^\infty \Re\Big(e^{-\mathrm{i}vk}\,\Psi(v)\Big)\,\mathrm{d}v, \qquad \Psi(v) := \frac{\phi_T\big(v - \mathrm{i}(\alpha+1)\big)}{\alpha^2 + \alpha - v^2 + \mathrm{i}(2\alpha+1)v}. \tag{9}$$

This is implemented in `carr_madan_call_fft` by constructing $\Psi(v)$ on a frequency grid and recovering $C^{\mathrm{USD}}(K)$ on a log-strike grid using an FFT.

2.3 FFT discretization (`carr_madan_call_fft`)

Choose an FFT configuration `FFTParams` with:

$$N \in 2^{\mathbb{N}}, \qquad \eta > 0, \qquad \alpha > 0, \qquad b \in \mathbb{R}, \tag{10}$$

where N is the number of grid points, η the frequency spacing, α the damping factor and b the minimum log-strike on the output grid.

Frequency grid. Define

$$v_j := j\eta, \qquad j = 0, 1, \ldots, N-1. \tag{11}$$

Log-strike grid. Define the dual spacing

$$\lambda := \frac{2\pi}{N\eta}, \tag{12}$$

and the log-strikes

$$k_u := b + u\lambda, \qquad u = 0, 1, \ldots, N-1, \qquad K_u := e^{k_u}. \tag{13}$$

Quadrature weights. Let w_j denote quadrature weights, either trapezoidal or Simpson, produced by `_quadrature_weights(N,use_simpson)`. The discretized integrand sequence is

$$x_j := e^{-\mathrm{i}v_j b}\,\Psi(v_j)\,w_j\,\eta. \tag{14}$$

FFT and call grid. The FFT computes

$$y_u := \sum_{j=0}^{N-1} e^{-2\pi\mathrm{i}\,ju/N}\,x_j, \tag{15}$$

and the damped call values are recovered via

$$c_\alpha(k_u) \approx \frac{1}{\pi}\Re(y_u). \tag{16}$$

Finally,

$$C^{\mathrm{USD}}(K_u) \approx e^{-\alpha k_u}\,c_\alpha(k_u). \tag{17}$$

The function `carr_madan_call_fft` returns the arrays $(K_u)_{u=0}^{N-1}$ and $(C^{\mathrm{USD}}(K_u))_{u=0}^{N-1}$.

2.4 Inverse (coin) prices (`price_inverse_option`)

Deribit quotes option premia in coin units. The pricer converts USD call prices to coin prices using:

$$C^{\text{coin}}(K,T;F_0,\theta) := \frac{C^{\text{USD}}(K,T;F_0,\theta)}{F_0}, \tag{18}$$

where F_0 is the current term futures price used as an exchange rate (USD per coin). In code this is the line `call_coin = call_prices_usd / F0`.

Puts are computed using inverse put–call parity as implemented:

$$C^{\text{coin}}(K,T) - P^{\text{coin}}(K,T) = 1 - \frac{K}{F_0} \quad \Longrightarrow \quad P^{\text{coin}}(K,T) = C^{\text{coin}}(K,T) - \Big(1 - \frac{K}{F_0}\Big). \tag{19}$$

The public API entry point is

`price_inverse_option(model, K, T, F0, params, option_type, fft_params)`.

2.5 Caching of FFT grids

To speed up calibration, the pricing grid $(K_u, C^{\text{USD}}(K_u))$ is cached by an LRU cache:

$$\texttt{\_cached\_pricing\_grid}: \ (m,T,F_0,\theta,\texttt{FFTParams}) \mapsto (K_u, C^{\text{USD}}(K_u)). \tag{20}$$

Thus repeated evaluations at the same maturity T, forward F_0, model name m, parameters θ and FFT grid reuse the FFT output rather than recomputing it.

3 Dataset pricing for calibration (`price_dataframe`)

Given a filtered DataFrame $\mathcal{D}$, `price_dataframe` produces the vector of model prices $\boldsymbol{P}^{\text{model}} \in \mathbb{R}^n$ in coin units by:

1. grouping rows by `expiry_datetime` and `option_type`;
2. for each group g, extracting a representative maturity and forward:

$$T_g := \text{median}\{T_i : i \in g\}, \qquad F_{0,g} := \text{median}\{F_{0,i} : i \in g\}; \tag{21}$$

3. calling `price_inverse_option` once per group with the vector of strikes $\{K_i\}_{i\in g}$.

3.1 Dynamic log-strike centering (`_choose_fft_params_for_group`)

FFT accuracy requires that the strike grid $\{K_u\}$ covers the strikes of interest. The helper `_choose_fft_params_for_group` keeps (N,η,α) fixed, computes

$$\lambda = \frac{2\pi}{N\eta}, \qquad k_{\text{center}} := \log\big(\text{median}\{K_i : i \in g\}\big), \tag{22}$$

and selects the left endpoint

$$b_g := k_{\text{center}} - \frac{N\lambda}{2}. \tag{23}$$

Then the grid $k_u = b_g + u\lambda$ is centered on the group's median strike. This reduces interpolation error and helps ensure that the grid spans the relevant strikes.

4 Filtering and cleaning (`filter_liquid_options`)

4.1 Required columns and numeric coercion

The function `filter_liquid_options` requires the columns

$$\{\texttt{currency}, \texttt{option\_type}, \texttt{strike}, \texttt{time\_to\_maturity}, \texttt{bid\_price}, \texttt{ask\_price}, \texttt{futures\_price}, \texttt{vega}, \texttt{open\_interest}, \texttt{expiry\_datetime}\}.$$

Columns used in computations are coerced to numeric (invalid parses become NaN) before filtering.

4.2 Liquidity and sanity constraints

Let i index a row. The filtration applies the following constraints:

$$\textbf{(TTM domain)} \qquad T_i > 0, \qquad T_i \geq T_{\min}, \qquad (T_i \leq T_{\max} \text{ if provided}). \tag{24}$$

$$\textbf{(Bid/ask present)} \qquad \mathrm{bid}_i > 0, \quad \mathrm{ask}_i > 0, \quad \mathrm{ask}_i \geq \mathrm{bid}_i. \tag{25}$$

$$\textbf{(Vega / OI)} \qquad V_i \geq V_{\min}, \qquad \mathrm{OI}_i \geq \mathrm{OI}_{\min}. \tag{26}$$

$$\textbf{(Relative spread)} \qquad \mathrm{rs}_i = \frac{\mathrm{ask}_i - \mathrm{bid}_i}{\max(m_i, \varepsilon)} \leq \mathrm{rs}_{\max} \quad \text{if provided.} \tag{27}$$

$$\textbf{(Moneyness window)} \qquad M_i = \frac{K_i}{F_{0,g}} \in [M_{\min}, M_{\max}] \quad \text{if provided.} \tag{28}$$

$$\textbf{(Option type)} \qquad \tau_i \in \{\mathrm{call}, \mathrm{put}\} \text{ after lowercasing.} \tag{29}$$

Optionally, synthetic underlyings with names beginning `SYN.` may be dropped via `drop_synthetic_underlyings=True`.

4.3 Output

The returned DataFrame is sorted by `expiry_datetime`, `strike` and `option_type` and includes the derived columns $(m_i, s_i, \mathrm{rs}_i, F_{0,i}, M_i, \kappa_i)$ described above.

5 Weighting scheme (`WeightConfig` and `_weights`)

5.1 Weighted residual vector

Calibration is performed by nonlinear least squares on a residual vector whose components are

$$r_i(\theta) := w_i\Big(P_i^{\mathrm{model}}(\theta) - P_i^{\mathrm{mkt}}\Big), \tag{30}$$

where:

- $P_i^{\mathrm{mkt}} := m_i$ is the cleaned mid price (`mid_price_clean`);
- $P_i^{\mathrm{model}}(\theta)$ is produced by `price_dataframe` using `price_inverse_option`.

5.2 Weight factorization

The weight w_i is constructed multiplicatively from three components (spread, vega, open interest) as implemented in `_weights`:

$$w_i = \Big(s_i + \varepsilon_s\Big)^{-p_s \cdot \mathbf{1}_{\mathrm{spread}}} \Big(V_i + \varepsilon_o\Big)^{p_v \cdot \mathbf{1}_{\mathrm{vega}}} \Big(\mathrm{OI}_i + \varepsilon_o\Big)^{p_o \cdot \mathbf{1}_{\mathrm{oi}}}, \tag{31}$$

where:

- $s_i = \text{ask}_i - \text{bid}_i$ is the bid–ask spread;
- V_i is the snapshot vega;
- OI_i is the snapshot open interest;
- $\varepsilon_s =$ `eps_spread` and $\varepsilon_o =$ `eps_other` are small stabilizers;
- exponents are configured by `spread_power` (p_s), `vega_power` (p_v), and `oi_power` (p_o);
- the indicator flags $\mathbf{1}_{\text{spread}}, \mathbf{1}_{\text{vega}}, \mathbf{1}_{\text{oi}} \in \{0, 1\}$ switch components on/off via `use_spread`, `use_vega`, `use_open_interest`.

Finally, weights may be capped to prevent dominance:

$$w_i \leftarrow \min\{w_i, w_{\max}\} \qquad \text{with } w_{\max} = \texttt{cap}. \tag{32}$$

If w_i is non-finite it is set to 0 (effectively dropping the observation).

Interpretation. With $p_s = 1$, r_i behaves like a (scaled) error normalized by the spread. With $p_v = p_o = \frac{1}{2}$, weights scale as $\sqrt{\text{vega} \times \text{OI}}$.

6 Calibration problem (`calibrate_model`)

6.1 Nonlinear least squares objective

Given a filtered dataset $\mathcal{D}$, define the residual map

$$\boldsymbol{r}(\theta) := \big(r_1(\theta), \ldots, r_n(\theta)\big)^\top \in \mathbb{R}^n, \tag{33}$$

with r_i as in eq. (30). The calibration estimates parameters by

$$\hat{\theta} \in \arg\min_{\theta \in \Theta} \ \frac{1}{2} \|\boldsymbol{r}(\theta)\|_2^2. \tag{34}$$

In code, `scipy.optimize.least_squares` (method `"trf"`) solves eq. (34).

6.2 Parameter constraints and unconstrained reparameterization

The optimizer operates on an unconstrained vector $x \in \mathbb{R}^d$. Each model uses a bijection (or a near-bijection) $\theta = T(x)$ to enforce constraints.

6.2.1 Black model

Parameters: $\theta = (\sigma)$ with $\sigma > 0$.

$$x = \log \sigma \quad \Longleftrightarrow \quad \sigma = e^x.$$

Implemented by `_pack_black` and `_unpack_black`.

6.2.2 Heston model

Parameters: $\theta = (\kappa, \theta, \sigma_v, \rho, v_0)$ with

$$\kappa > 0, \quad \theta > 0, \quad \sigma_v > 0, \quad v_0 > 0, \quad \rho \in (-1, 1).$$

Transform:

$$x_1 = \log \kappa, \qquad \kappa = e^{x_1}, \tag{35}$$
$$x_2 = \log \theta, \qquad \theta = e^{x_2}, \tag{36}$$
$$x_3 = \log \sigma_v, \qquad \sigma_v = e^{x_3}, \tag{37}$$
$$x_4 = \operatorname{arctanh}(\rho), \qquad \rho = \tanh(x_4), \tag{38}$$
$$x_5 = \log v_0, \qquad v_0 = e^{x_5}. \tag{39}$$

Implemented by `_pack_heston` and `_unpack_heston`.

6.2.3 SVCJ model

Parameters extend Heston by jump parameters $\lambda, \ell_y, \sigma_y, \ell_v, \rho_j$:

$$\lambda > 0, \quad \sigma_y > 0, \quad \ell_v > 0,$$

while $\ell_y \in \mathbb{R}$ is unconstrained and ρ_j is constrained indirectly to preserve finiteness of the jump MGF.

Key stability constraint. The SVCJ characteristic function implementation requires

$$1 - \ell_v \rho_j > 0, \tag{40}$$

to keep the denominator of the jump MGF positive (and thus finite) at the relevant complex arguments.

Reparameterization used in code. The module introduces an auxiliary bounded scalar

$$\psi := \ell_v \rho_j \in (-0.99, 0.99), \tag{41}$$

and optimizes over x_{10} such that

$$\psi = 0.99 \tanh(x_{10}), \qquad \rho_j = \frac{\psi}{\ell_v}. \tag{42}$$

This ensures $\psi < 1$ and, in practice, supports eq. (40) for most parameter values. Other parameters are transformed as:

$$\lambda = e^{x_6}, \qquad \ell_y = x_7, \qquad \sigma_y = e^{x_8}, \qquad \ell_v = e^{x_9}. \tag{43}$$

Together with the Heston transforms for $(\kappa, \theta, \sigma_v, \rho, v_0)$, this defines the SVCJ mapping, implemented by `_pack_svcj` and `_unpack_svcj`.

6.3 Initial guesses

If the dataset contains `implied_volatility`, the initializer sets an "ATM" proxy

$$\sigma_{\text{atm}} := \text{median}\{\texttt{implied\_volatility}_i\}, \tag{44}$$

and uses $\theta_{\text{init}} = \sigma_{\text{atm}}^2$ for variance-level parameters. Otherwise a fallback $\sigma_{\text{atm}} = 0.6$ is used. Implemented in `_default_initial_params`.

6.4 Cache management

Before running the optimizer, `calibrate_model` may clear the FFT grid cache by calling `clear_fft_cache()`, which in turn invokes `_cached_pricing_grid.cache_clear()`.

6.5 Fit diagnostics

After obtaining $\hat{\theta}$, the code computes model prices on the fitting set and reports:

$$\text{RMSE} := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(P_i^{\text{model}}(\hat{\theta}) - P_i^{\text{mkt}})^2}, \qquad \text{MAE} := \frac{1}{n}\sum_{i=1}^{n}\left|P_i^{\text{model}}(\hat{\theta}) - P_i^{\text{mkt}}\right|. \tag{45}$$

7 End-to-end algorithm summary

Algorithm 1 Calibration pipeline (implemented by `filter_liquid_options` and `calibrate_model`)

Require: Raw snapshot DataFrame $\mathcal{D}_{\text{raw}}$, model name $m \in \{\text{black,heston,svcj}\}$

1: $\mathcal{D} \leftarrow$ `filter_liquid_options`$(\mathcal{D}_{\text{raw}}, \text{liquidity thresholds})$
2: Construct market mid prices $\boldsymbol{P}^{\text{mkt}}$ and weights $\boldsymbol{w}$ via `_weights` and `WeightConfig`
3: Choose initial parameters $\theta^{(0)}$ via `_default_initial_params`
4: Map to unconstrained $x^{(0)} =$ `_pack_*`$(\theta^{(0)})$
5: Define residual function $x \mapsto \boldsymbol{r}(T(x))$ where model prices use `price_dataframe` $\circ$ `price_inverse_option`
6: Solve $\hat{x} \in \arg\min_x \frac{1}{2}\|\boldsymbol{r}(T(x))\|_2^2$ using `least_squares(method="trf")`
7: Return $\hat{\theta} = T(\hat{x})$ and diagnostics (`CalibrationResult`)

8 Implementation cross-reference

- **Filtering:** `filter_liquid_options` (adds `mid_price_clean`, `spread`, `rel_spread`, `F0`, `moneyness`, `log_moneyness`).
- **Weights:** `WeightConfig` and `_weights` implement eq. (31).
- **Pricing:** `price_dataframe` groups by expiry and calls `price_inverse_option`.
- **FFT grid selection:** `_choose_fft_params_for_group` selects b_g for centered strike coverage.
- **Carr–Madan engine:** `carr_madan_call_fft` implements eq. (9).
- **Inverse conversion:** `price_inverse_option` implements eq. (18) and eq. (19).
- **Calibration:** `calibrate_model` solves eq. (34) with parameter transforms `_pack_*` / `_unpack_*`.