USDHKD volatility surface build

Given market inputs:

- Spot: S_0
- Forward: F
- ATM Call price at strike K = F : C_{mkt}
- ATM Put price at strike K = F : P_{mkt}

We want to solve for jump model parameters:

- p: probability of no jump
- q: probability of jump down -d
- d: log-jump size (assume symmetric, so +d and -d only

Model Assumptions

- $U \sim \text{Unif}[a, b]$, e.g. $[7.75, 7.85]$
- Spot return:

$$S_T = e^{R_{\text{jump}}} \cdot U$$

- Three-point jump:

$$R_{\text{jump}} = \begin{cases} -d & \text{with prob } q \\ 0 & \text{with prob } p \\ +d & \text{with prob } 1 - p - q \end{cases}$$

Step 1: Forward Constraint

From risk-neutral pricing:

$$F = \mathbb{E}[S_T] = \mu \cdot (qe^{-d} + p + (1 - p - q)e^d) \quad \text{where } \mu = \frac{a + b}{2} \tag{1}$$

Step 2: ATM Call Pricing

$$C_{\text{mkt}} = q \cdot \Phi_{\text{call}}(F; -d) + p \cdot \Phi_{\text{call}}(F; 0) + (1 - p - q) \cdot \Phi_{\text{call}}(F; d) \tag{2}$$

Step 3: ATM Put Pricing

$$P_{\text{mkt}} = q \cdot \Phi_{\text{put}}(F; -d) + p \cdot \Phi_{\text{put}}(F; 0) + (1 - p - q) \cdot \Phi_{\text{put}}(F; d) \tag{3}$$

Definitions of $\Phi_{\text{call}}(K; x)$ and $\Phi_{\text{put}}(K; x)$

Let $a_x = ae^x$, $b_x = be^x$, $\Delta_x = b_x - a_x = \Delta e^x$:

$$\Phi_{\text{call}}(K;x) = \begin{cases} \frac{a_x+b_x}{2} - K, & K \le a_x \\ \frac{(b_x-K)^2}{2\Delta_x}, & a_x < K < b_x \\ 0, & K \ge b_x \end{cases}$$

$$\Phi_{\text{put}}(K;x) = \begin{cases} 0, & K \le a_x \\ \frac{(K-a_x)^2}{2\Delta_x}, & a_x < K < b_x \\ K - \frac{a_x+b_x}{2}, & K \ge b_x \end{cases}$$

🚀 Calibration Strategy

Option A: Fix $p = 0$

Then:

- From (1):

$$q = \frac{e^d - \frac{F}{\mu}}{e^d - e^{-d}} \tag{4}$$

- Plug into (2), (3) → only unknown is d

Use root solver (Brent / Newton) to solve for d

Option B: General case (solve p, q, d)

1. Use (1) to eliminate p:

$$p = \frac{F}{\mu} - qe^{-d} - (1-p-q)e^d$$

2. Plug into (2), (3) → Two equations, two unknowns: q, d

Use root finder / system solver (e.g. `scipy.optimize.root`)

Final Output

Once you solve for (p, q, d), you can:

- Build full risk-neutral PDF
- Price any vanilla option with closed-form
- Generate implied vol surface
- Derive 25Δ RR, Fly, and Greeks

4. Case-by-Case Pricing Formula (7 Cases)

Let $a_{-d} = ae^{-d}$, $b_{-d} = be^{-d}$, $a_0 = a$, $b_0 = b$, $a_d = ae^d$, $b_d = be^d$

Case 1: $K < ae^{-d}$

$$C(K) = q\left(\frac{a_{-d}+b_{-d}}{2} - K\right) + p\left(\frac{a_0+b_0}{2} - K\right) + (1-p-q)\left(\frac{a_d+b_d}{2} - K\right)$$

Case 2: $ae^{-d} \leq K < be^{-d}$

$$C(K) = q \cdot \frac{(b_{-d}-K)^2}{2\Delta_{-d}} + p\left(\frac{a_0+b_0}{2} - K\right) + (1-p-q)\left(\frac{a_d+b_d}{2} - K\right)$$

Case 3: $be^{-d} \leq K < a$

$$C(K) = p\left(\frac{a_0+b_0}{2} - K\right) + (1-p-q)\left(\frac{a_d+b_d}{2} - K\right)$$

Case 4: $a \leq K < b$

$$C(K) = p \cdot \frac{(b_0-K)^2}{2\Delta_0} + (1-p-q)\left(\frac{a_d+b_d}{2} - K\right)$$

Case 5: $b \leq K < ae^d$

$$C(K) = (1-p-q)\left(\frac{a_d+b_d}{2} - K\right)$$

Case 6: $ae^d \leq K < be^d$

$$C(K) = (1-p-q) \cdot \frac{(b_d-K)^2}{2\Delta_d}$$

Case 7: $K \geq be^d$

$$C(K) = 0$$

Put cases follow symmetric logic using Φ_{put}.

5. Third Equation: Forward Constraint

The forward price (risk-neutral expectation of S_T):

$$F = \mathbb{E}[S_T] = q \cdot \mu e^{-d} + p \cdot \mu + (1-p-q) \cdot \mu e^d, \quad \text{where } \mu = \frac{a+b}{2}$$

6. Calibration: Solve for (p, q, d)

Use three equations:

- Call price: $C_{\text{mkt}} = C(K = F)$

- Put price: $P_{\text{mkt}} = P(K = F)$
- Forward equation above

General (p, q, d)

Minimize objective:

$$\text{loss} = (F_{\text{model}} - F)^2 + (C_{\text{model}} - C_{\text{mkt}})^2 + (P_{\text{model}} - P_{\text{mkt}})^2$$

Use `scipy.optimize.minimize` with constraints $p \geq 0$, $q \geq 0$, $p + q \leq 1$, $d > 0$

Output

- Exact analytical expressions for call/put across all strikes
- Can be used to compute:
  - Implied volatility surface
  - 25Δ Risk Reversal and Butterfly
  - Greeks: Delta, Vega, etc.

In [2]:

```
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# Define Φ_call and Φ_put for vectorized strike K and fixed x
def phi_call_vector(K, x):
    ax = a * np.exp(x)
    bx = b * np.exp(x)
    Dx = bx - ax
    result = np.piecewise(K,
                          [K <= ax, (K > ax) & (K < bx), K >= bx],
                          [lambda K: (ax + bx) / 2 - K,
                           lambda K: (bx - K) ** 2 / (2 * Dx),
                           0.0])
    return result

def phi_put_vector(K, x):
    ax = a * np.exp(x)
    bx = b * np.exp(x)
    Dx = bx - ax
    result = np.piecewise(K,
                          [K <= ax, (K > ax) & (K < bx), K >= bx],
                          [0.0,
                           lambda K: (K - ax) ** 2 / (2 * Dx),
                           lambda K: K - (ax + bx) / 2])
    return result

# Full 3-variable calibration: objective function
def calibration_objective(x):
    p, q, d = x
    if p < 0 or q < 0 or (p + q > 1) or d <= 0:
        return 1e6  # infeasible
    w0 = p
    w1 = q
    w2 = 1 - p - q
    f_model = mu * (w1 * np.exp(-d) + w0 + w2 * np.exp(d))
    c_model = w1 * phi_call(F, -d) + w0 * phi_call(F, 0) + w2 * phi_call(F, d
    p_model = w1 * phi_put(F, -d) + w0 * phi_put(F, 0) + w2 * phi_put(F, d)
    return (f_model - F) ** 2 + (c_model - C_mkt) ** 2 + (p_model - P_mkt) **
```

```
# Initial guess: p, q, d
x0 = [0.5, 0.25, 0.01]
bounds = [(0, 1), (0, 1), (0.001, 1.0)]

res_full = minimize(calibration_objective, x0=x0, bounds=bounds)
p_fit, q_fit, d_fit = res_full.x

# Evaluate PDF over K for plotting
K_vals = np.linspace(7.70, 7.90, 200)
pdf_vals = (
    q_fit / (b * np.exp(-d_fit) - a * np.exp(-d_fit)) * ((K_vals >= a * np.ex
    p_fit / (b - a) * ((K_vals >= a) & (K_vals <= b)) +
    (1 - p_fit - q_fit) / (b * np.exp(d_fit) - a * np.exp(d_fit)) * ((K_vals
)

# Plot the calibrated PDF
plt.figure(figsize=(8, 4))
plt.plot(K_vals, pdf_vals, label="Calibrated Risk-Neutral PDF")
plt.axvline(F, color='gray', linestyle='--', label='Forward (ATM strike)')
plt.title("Risk-Neutral Density for USD/HKD with Jump Model")
plt.xlabel("Strike K")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

(p_fit, q_fit, d_fit)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 44
     41 x0 = [0.5, 0.25, 0.01]
     42 bounds = [(0, 1), (0, 1), (0.001, 1.0)]
---> 44 res_full = minimize(calibration_objective, x0=x0, bounds=bounds)
     45 p_fit, q_fit, d_fit = res_full.x
     47 # Evaluate PDF over K for plotting

File ~/opt/anaconda3/envs/myenv/lib/python3.10/site-packages/scipy/optimize/_
minimize.py:713, in minimize(fun, x0, args, method, jac, hess, hessp, bounds,
constraints, tol, callback, options)
    710     res = _minimize_newtoncg(fun, x0, args, jac, hess, hessp, callbac
k,
    711                              **options)
    712 elif meth == 'l-bfgs-b':
--> 713     res = _minimize_lbfgsb(fun, x0, args, jac, bounds,
    714                            callback=callback, **options)
    715 elif meth == 'tnc':
    716     res = _minimize_tnc(fun, x0, args, jac, bounds, callback=callbac
k,
    717                         **options)

File ~/opt/anaconda3/envs/myenv/lib/python3.10/site-packages/scipy/optimize/_
lbfgsb_py.py:309, in _minimize_lbfgsb(fun, x0, args, jac, bounds, disp, maxco
r, ftol, gtol, eps, maxfun, maxiter, iprint, callback, maxls, finite_diff_rel
_step, **unknown_options)
    306     iprint = disp
    308 # _prepare_scalar_function can use bounds=None to represent no bounds
--> 309 sf = _prepare_scalar_function(fun, x0, jac=jac, args=args, epsilon=ep
s,
    310                               bounds=bounds,
    311                               finite_diff_rel_step=finite_diff_rel_st
ep)
    313 func_and_grad = sf.fun_and_grad
    315 fortran_int = _lbfgsb.types.intvar.dtype

File ~/opt/anaconda3/envs/myenv/lib/python3.10/site-packages/scipy/optimize/_
```

```
optimize.py:402, in _prepare_scalar_function(fun, x0, jac, args, bounds, epsi
lon, finite_diff_rel_step, hess)
    398     bounds = (-np.inf, np.inf)
    400 # ScalarFunction caches. Reuse of fun(x) during grad
    401 # calculation reduces overall function evaluations.
--> 402 sf = ScalarFunction(fun, x0, args, grad, hess,
    403                     finite_diff_rel_step, bounds, epsilon=epsilon)
    405 return sf

File ~/opt/anaconda3/envs/myenv/lib/python3.10/site-packages/scipy/optimize/_
differentiable_functions.py:166, in ScalarFunction.__init__(self, fun, x0, ar
gs, grad, hess, finite_diff_rel_step, finite_diff_bounds, epsilon)
    163     self.f = fun_wrapped(self.x)
    165 self._update_fun_impl = update_fun
--> 166 self._update_fun()
    168 # Gradient evaluation
    169 if callable(grad):

File ~/opt/anaconda3/envs/myenv/lib/python3.10/site-packages/scipy/optimize/_
differentiable_functions.py:262, in ScalarFunction._update_fun(self)
    260 def _update_fun(self):
    261     if not self.f_updated:
--> 262         self._update_fun_impl()
    263         self.f_updated = True

File ~/opt/anaconda3/envs/myenv/lib/python3.10/site-packages/scipy/optimize/_
differentiable_functions.py:163, in ScalarFunction.__init__.<locals>.update_f
un()
    162 def update_fun():
--> 163     self.f = fun_wrapped(self.x)

File ~/opt/anaconda3/envs/myenv/lib/python3.10/site-packages/scipy/optimize/_
differentiable_functions.py:145, in ScalarFunction.__init__.<locals>.fun_wrap
ped(x)
    141 self.nfev += 1
    142 # Send a copy because the user may overwrite it.
    143 # Overwriting results in undefined behaviour because
    144 # fun(self.x) will change self.x, with the two no longer linked.
--> 145 fx = fun(np.copy(x), *args)
    146 # Make sure the function returns a true scalar
    147 if not np.isscalar(fx):

Cell In[2], line 35, in calibration_objective(x)
     33 w1 = q
     34 w2 = 1 - p - q
---> 35 f_model = mu * (w1 * np.exp(-d) + w0 + w2 * np.exp(d))
     36 c_model = w1 * phi_call(F, -d) + w0 * phi_call(F, 0) + w2 * phi_call
(F, d)
     37 p_model = w1 * phi_put(F, -d) + w0 * phi_put(F, 0) + w2 * phi_put(F,
d)

NameError: name 'mu' is not defined
```

In [3]:

```python
# Consolidate all required functions and run global calibration again

import numpy as np
from scipy.optimize import minimize

# Constants (based on prior session)
a, b = 7.75, 7.85
mu = 0.5 * (a + b)
F = mu  # Approximate ATM forward
C_mkt = 0.025  # Example market ATM call price
P_mkt = 0.020  # Example market ATM put price

# Define elementary phi functions
def phi_call(K, x):
    ax, bx = a * np.exp(x), b * np.exp(x)
```

```python
    Dx = bx - ax
    if K <= ax:
        return (ax + bx) / 2 - K
    elif ax < K < bx:
        return (bx - K)**2 / (2 * Dx)
    else:
        return 0.0

def phi_put(K, x):
    ax, bx = a * np.exp(x), b * np.exp(x)
    Dx = bx - ax
    if K <= ax:
        return 0.0
    elif ax < K < bx:
        return (K - ax)**2 / (2 * Dx)
    else:
        return K - (ax + bx) / 2

# Call price by region
def call_price_7case_with_label(K, p, q, d):
    components = []
    for x, label in zip([-d, 0.0, d], ['-d', '0', '+d']):
        ax, bx = a * np.exp(x), b * np.exp(x)
        Dx = bx - ax
        if K <= ax:
            value, region = (ax + bx) / 2 - K, "linear"
        elif ax < K < bx:
            value, region = (bx - K)**2 / (2 * Dx), "quadratic"
        else:
            value, region = 0.0, "zero"
        weight = q if label == '-d' else p if label == '0' else 1 - p - q
        components.append({
            "jump": label,
            "support": (ax, bx),
            "type": region,
            "weight": weight,
            "contrib": weight * value
        })
    total_price = sum(c["contrib"] for c in components)
    return total_price, components

# Generate 7 sample Ks across regions
def generate_strike_examples(d):
    return [
        a * np.exp(-d) - 0.01,
        (a * np.exp(-d) + b * np.exp(-d)) / 2,
        b * np.exp(-d) + 0.001,
        (a + b) / 2,
        b + 0.001,
        (a * np.exp(d) + b * np.exp(d)) / 2,
        b * np.exp(d) + 0.01
    ]

# Global calibration objective across all Ks
def global_calibration_loss(x, Ks, market_prices):
    p, q, d = x
    if not (0 <= p <= 1 and 0 <= q <= 1 and 0 <= p + q <= 1 and d > 0):
        return 1e6
    total_loss = 0
    for K, market_price in zip(Ks, market_prices):
        model_price, _ = call_price_7case_with_label(K, p, q, d)
        total_loss += (model_price - market_price)**2
    return total_loss
```

```
# Step 1: use single point to get initial d estimate
initial_d = 0.0028
K_examples = generate_strike_examples(initial_d)

# Step 2: simulate model prices at these K (using previously calibrated d=0.0
p0, q0, d0 = 0.0, 0.0, 0.00273
market_prices = [call_price_7case_with_label(K, p0, q0, d0)[0] for K in K_exa

# Step 3: perform global calibration
x0_global = [0.2, 0.2, 0.01]
bounds_global = [(0, 1), (0, 1), (1e-5, 0.5)]
global_result = minimize(global_calibration_loss, x0=x0_global,
                         bounds=bounds_global, args=(K_examples, market_price

# Final calibrated values
p_global, q_global, d_global = global_result.x
p_global, q_global, 1 - p_global - q_global, d_global
```

Out[3]: (0.05794801727223921, 0.0, 0.9420519827277608, 0.0028792238937540887)

In [ ]: