Volatility Surface Estimation Report

Task: Volatility Surface Estimation with Interpolation

* Scrape or download option chain data (for example using yfinance, you could probably use Perplexity now also (I saw they have financial data integrations now!))
* Calculate implied volatilities using the Black-Scholes formula
* Interpolate the data to construct a smooth surface (e.g., using cubic spline, RBF or similar)
* Visualize implied volatility across strike and expiry.
* (Optional) Fit a parametric model such as SABR or SVI for deeper analysis

1. **Download the option chain data**

An option chain is the table that lists all listed call and put contracts for a single underlying (e.g., a stock or ETF), organized by expiration dates and strike prices. It shows the going market prices and analytics you use to compare contracts at a glance. Most platforms group by expiration; within each expiry, rows are strikes, with calls on one side and puts on the other. This table view is the standard way traders browse and pick contracts.

Implied Volatility (IV) is the market-implied annualized volatility embedded in the option’s price. A higher IV generally means higher option premiums. In this report, we will build a smooth volatility surface, which requires a sufficiently dense grid across both strikes and expirations. Expiration includes multiple expiration dates, ideally spanning from short-term (days/weeks) to longer-term (months, maybe years). Strikes are mainly around at-the-money, but we will also include out-of-the-money (OTM) and in-the-money (ITM) strikes to capture the full “volatility smile” or skew. We need enough amount of data, otherwise too sparse data—few strikes or expirations—can lead to poor surface estimates or arbitrage‑violating shapes. In the following analysis, we will select one option chain SPY for the target underlying, ideally exporting all the expirations and strikes available. Here are the codes,

import pandas as pd

import yfinance as yf

def download\_option\_chain(ticker="SPY"):

tk = yf.Ticker(ticker)

# 1) list all expiration dates available for this underlying

expirations = tk.options # e.g., ['2025-08-22', '2025-08-29', ...]

if not expirations:

raise RuntimeError(f"No option expirations found for {ticker}.")

all\_rows = []

# 2) loop through expirations and pull calls/puts for each

for exp in expirations:

chain = tk.option\_chain(exp) # returns namedtuple(calls=..., puts=...)

calls = chain.calls.copy()

puts = chain.puts.copy()

# tag with metadata

calls["expiration"] = exp

calls["type"] = "call"

puts["expiration"] = exp

puts["type"] = "put"

all\_rows.append(calls)

all\_rows.append(puts)

# 3) combine into one DataFrame

df = pd.concat(all\_rows, ignore\_index=True)

# 4) (optional) keep a compact set of useful columns if present

keep = [

"contractSymbol", "type", "expiration", "strike", "lastPrice",

"bid", "ask", "volume", "openInterest", "inTheMoney", "impliedVolatility"

]

df = df[[c for c in keep if c in df.columns]]

return df

if \_\_name\_\_ == "\_\_main\_\_":

df = download\_option\_chain("SPY")

print(df.head())

# Optional: save to CSV for later IV/surface work

df.to\_csv("option\_chain\_SPY.csv", index=False)

print("Saved: option\_chain\_SPY.csv")

This code downloads and saves the code as a csv file.

1. **Implied volatilities calculation by Black-Scholes formula**

The Black–Scholes model is a mathematical framework used to price European-style options, assuming the underlying asset follows a geometric Brownian motion with constant volatility and drift. The European call option price C is given by

In this formula,

S: current stock (underlying asset) price

K: strike price

r: risk-free interest rate

T−t: time to expiration

N(x): cumulative standard normal distribution

d1:

d2:

σ: volatility of the underlying asset’s returns, which is the parameter we would like to fit.

Implied volatility is the volatility input that, when plugged into the Black–Scholes formula, gives a theoretical option price matching the current market price. Because the formula is monotonic in 𝜎, there’s a unique solution for implied volatility given the market price. This solution can be numerically solved (e.g., using root-finding, bisection, or Newton–Raphson) for 𝜎, such that the model price equals the market price. In the Python code next, we will use the Brant method to find the implied volatility. Brent’s method is a root-finding algorithm that combines the reliability of bisection with the speed of secant and inverse quadratic interpolation approaches. It’s designed to converge efficiently and robustly, even when derivative information isn’t available. In Python code, we may use the line scipy.optimize.brentq to introduce the Brant method. The full code to find the implied volatility is

import numpy as np

import pandas as pd

import yfinance as yf

from scipy.optimize import brentq

from math import log, sqrt, exp

from datetime import datetime, timezone

# --- Black–Scholes helpers (European) with continuous dividend yield q ---

from math import erf

def \_ncdf(x):

# standard normal CDF without scipy.stats

return 0.5 \* (1.0 + erf(x / np.sqrt(2.0)))

def bs\_price(S, K, T, r, q, sigma, option\_type):

if T <= 0 or sigma <= 0 or S <= 0 or K <= 0:

return np.nan

d1 = (np.log(S/K) + (r - q + 0.5\*sigma\*sigma)\*T) / (sigma\*sqrt(T))

d2 = d1 - sigma\*sqrt(T)

if option\_type == "call":

return S\*exp(-q\*T)\*\_ncdf(d1) - K\*exp(-r\*T)\*\_ncdf(d2)

else: # put

return K\*exp(-r\*T)\*\_ncdf(-d2) - S\*exp(-q\*T)\*\_ncdf(-d1)

def bs\_vega(S, K, T, r, q, sigma):

if T <= 0 or sigma <= 0 or S <= 0 or K <= 0:

return 0.0

d1 = (np.log(S/K) + (r - q + 0.5\*sigma\*sigma)\*T) / (sigma\*sqrt(T))

return S\*exp(-q\*T)\*sqrt(T) \* (1.0/np.sqrt(2\*np.pi)) \* np.exp(-0.5\*d1\*d1)

def implied\_vol(price, S, K, T, r, q, option\_type, sigma\_lo=1e-6, sigma\_hi=5.0):

# Guardrails: no-arbitrage-ish bounds

disc = exp(-r\*T); dq = exp(-q\*T)

intrinsic = max(0.0, (S - K) if option\_type=="call" else (K - S))

upper = (S if option\_type=="call" else K) # very loose

# If outside rough bounds, skip

if not (intrinsic - 1e-8 <= price <= upper + 1e-8):

return np.nan

def f(sig):

return bs\_price(S,K,T,r,q,sig,option\_type) - price

try:

return brentq(f, sigma\_lo, sigma\_hi, maxiter=100, xtol=1e-8)

except ValueError:

# Try nudging brackets if target is slightly out-of-range due to micro noise

try:

return brentq(f, 1e-8, 10.0, maxiter=100, xtol=1e-8)

except Exception:

return np.nan

# --- Main routine: attach IV to your option chain DataFrame ---

def compute\_iv\_surface\_inputs(df, ticker="SPY", r=0.0, q=None):

"""

df: option chain with at least columns:

['type','expiration','strike','bid','ask','lastPrice'] (case-insensitive ok)

r: risk-free rate (annualized, cont. comp assumed). Use e.g. 3M T-bill as proxy.

q: dividend yield (annualized, cont. comp). If None, try to infer a rough value.

"""

# Normalize column names

cols = {c.lower(): c for c in df.columns}

def col(name):

return cols.get(name, name)

# Spot (S)

S = float(yf.Ticker(ticker).history(period="1d")["Close"].iloc[-1])

# Dividend yield q (continuous). If not given, you can insert a recent SPY yield ~1.1%/yr.

# For rigorous use, pass q explicitly from your data source.

if q is None:

q = 0.011 # rough recent SPY dividend yield as a default (adjust as desired)

# Current time

now = datetime.now(timezone.utc)

# Mid price

df = df.copy()

if "mid" not in df.columns:

bid = df[col("bid")] if col("bid") in df.columns else np.nan

ask = df[col("ask")] if col("ask") in df.columns else np.nan

last = df[col("lastPrice")] if col("lastPrice") in df.columns else np.nan

df["mid"] = np.where(np.isfinite(bid) & np.isfinite(ask) & (ask>0),

(bid + ask)/2.0,

last)

# Time to expiry (ACT/365)

exp\_series = pd.to\_datetime(df[col("expiration")], utc=True, errors="coerce")

T = (exp\_series - now).dt.total\_seconds() / (365.0\*24\*3600)

df["T"] = T.clip(lower=0.0)

# Option type normalization: 'call'/'put'

typ = df[col("type")].str.lower().map({"c":"call","call":"call","p":"put","put":"put"})

df["otype"] = typ

# Compute IV per row

strikes = df[col("strike")].astype(float).values

prices = df["mid"].astype(float).values

Ts = df["T"].astype(float).values

types = df["otype"].values

ivs = []

for price\_i, K\_i, T\_i, ty in zip(prices, strikes, Ts, types):

if not np.isfinite(price\_i) or not np.isfinite(K\_i) or not np.isfinite(T\_i) or ty not in ("call","put"):

ivs.append(np.nan); continue

if T\_i <= 0:

ivs.append(np.nan); continue

iv = implied\_vol(price\_i, S, K\_i, T\_i, r, q, ty)

ivs.append(iv)

df["iv\_bs"] = ivs

# (optional) Drop rows with missing IV, illiquid/noisy quotes

# df = df[(df["iv\_bs"].notna()) & (df[col("openInterest")] > 0)]

df.attrs["spot"] = S

df.attrs["r"] = r

df.attrs["q"] = q

return df

One note here: SPY options are American-style (early exercise possible), so Black–Scholes (European) is an approximation.

1. **Visualize implied volatility by constructing a smooth surface**

We continue the code in section 2 and visualize the implied volatility. We will use scipy.interpolate.griddata (scattered → grid) with a nearest-fill fallback; optionally switch to RBF Interpolator for an extra-smooth surface. We avoid extrapolation where possible.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from scipy.interpolate import griddata, RBFInterpolator

from math import exp

def build\_iv\_surface(df,

iv\_col="iv\_bs",

k\_steps=121,

T\_steps=50,

method="griddata", # or "rbf"

rbf\_kernel="thin\_plate", rbf\_smoothing=0.0):

S = float(df.attrs.get("spot", np.nan))

r = float(df.attrs.get("r", 0.0))

q = float(df.attrs.get("q", 0.0))

now = pd.Timestamp.utcnow()

df2 = df.copy()

df2 = df2[df2[iv\_col].notna() & df2["T"].notna() & df2["strike"].notna()]

df2 = df2[(df2["T"] > 0) & (df2[iv\_col] > 0)]

if df2.empty:

raise ValueError("No valid IV data to interpolate.")

T = df2["T"].to\_numpy(float)

K = df2["strike"].to\_numpy(float)

F = S \* np.exp((r - q) \* T)

k = np.log(K / F)

w = (df2[iv\_col].to\_numpy(float)\*\*2) \* T

k\_lo, k\_hi = np.percentile(k, [2.5, 97.5])

T\_lo, T\_hi = T.min(), T.max()

k\_grid = np.linspace(k\_lo, k\_hi, k\_steps)

T\_grid = np.linspace(T\_lo, T\_hi, T\_steps)

KK, TT = np.meshgrid(k\_grid, T\_grid)

pts = np.column\_stack([k, T])

if method == "griddata":

w\_lin = griddata(pts, w, (KK, TT), method="linear")

w\_near = griddata(pts, w, (KK, TT), method="nearest")

W = np.where(np.isfinite(w\_lin), w\_lin, w\_near)

else:

rbf = RBFInterpolator(pts, w, kernel=rbf\_kernel, smoothing=rbf\_smoothing)

W = rbf(np.column\_stack([KK.ravel(), TT.ravel()])).reshape(KK.shape)

TT\_safe = np.clip(TT, 1e-8, None)

IV\_grid = np.sqrt(np.clip(W, 0, None) / TT\_safe)

F\_grid = S \* np.exp((r - q) \* TT)

K\_grid = F\_grid \* np.exp(KK)

return K\_grid, TT \* 365, IV\_grid # T in days

def plot\_iv\_surface(K\_grid, T\_days, IV\_grid, title="Implied Vol Surface"):

from mpl\_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(10, 6))

ax = fig.add\_subplot(111, projection="3d")

surf = ax.plot\_surface(K\_grid, T\_days, IV\_grid, cmap="viridis", edgecolor='none')

ax.set\_xlabel("Strike (K)")

ax.set\_ylabel("Days to Expiry")

ax.set\_zlabel("Implied Volatility")

ax.set\_title(title)

fig.colorbar(surf, shrink=0.6, label="IV")

plt.show()

plt.figure(figsize=(9, 5))

cp = plt.contourf(K\_grid, T\_days, IV\_grid, levels=20, cmap="viridis")

plt.xlabel("Strike (K)")

plt.ylabel("Days to Expiry")

plt.title(f"{title} (Contour)")

plt.colorbar(cp, label="IV")

plt.show()

df\_iv = pd.read\_csv("option\_chain\_SPY\_with\_IV.csv")

# If attrs lost, reinstate:

df\_iv.attrs["spot"] = 645.31

df\_iv.attrs["r"] = 0.043

df\_iv.attrs["q"] = 0.011

Kg, Td, IVg = build\_iv\_surface(df\_iv, method="griddata")

plot\_iv\_surface(Kg, Td, IVg, title="SPY IV Surface")

# For a smoother view:

Kg2, Td2, IVg2 = build\_iv\_surface(df\_iv, method="rbf", rbf\_smoothing=1e-6)

plot\_iv\_surface(Kg2, Td2, IVg2, title="SPY IV Surface (smooth RBF)")

Here is the graph plotted,

图表, 表面图

AI 生成的内容可能不正确。

图表, 直方图

AI 生成的内容可能不正确。

Here K means the strike price, not the unit kilo.

From the plot, we can see a volatility smile. When you plot implied volatility (IV) against strike price for options with the same expiration, a volatility smile appears as a U‑shaped curve—higher IV for deep in-the-money (ITM) and out-of-the-money (OTM) options, with the lowest at at-the-money (ATM) strikes. This pattern reflects the market pricing in "fat tails"—the perceived risk of extreme price moves not captured by the Black–Scholes model.

1. **SVI parameter fitting**

SVI (raw form) models total implied variance as a function of log-forward moneyness .

We will fit the parameters a, b, m, and with constraints b>0, , -1<<1. The parameter fitting is done via nonlinear least squares optimization, specifically using the L-BFGS-B algorithm from scipy.optimize.minimize.

import numpy as np

import pandas as pd

from scipy.optimize import minimize

# --- SVI model (raw) on total variance w(k) ---

def svi\_w\_raw(k, a, b, rho, m, sigma):

return a + b \* (rho \* (k - m) + np.sqrt((k - m)\*\*2 + sigma\*\*2))

def fit\_svi\_slice(k, w, weights=None):

"""

Fit raw SVI parameters (a,b,rho,m,sigma) for one expiry slice.

k: array of log-forward moneyness

w: array of total implied variance (IV^2 \* T)

weights: optional weights (same length as k)

"""

k = np.asarray(k, float)

w = np.asarray(w, float)

if weights is None:

weights = np.ones\_like(w)

else:

weights = np.asarray(weights, float)

# Initial guesses (robust, data-driven)

a0 = np.maximum(1e-6, np.percentile(w, 10))

m0 = np.median(k)

# rough slope from linear fit as seed for b\*rho; start b>0 and rho<0 (equity skew)

slope = np.polyfit(k, w, 1)[0] if len(k) >= 2 else 0.0

b0 = np.maximum(1e-4, 0.5 \* np.std(w)) # magnitude

rho0 = np.clip(np.sign(slope) \* (-0.5), -0.99, 0.99) # equity: negative skew

sigma0 = np.maximum(1e-3, 0.1 \* (np.percentile(k, 84) - np.percentile(k, 16)))

x0 = np.array([a0, b0, rho0, m0, sigma0], dtype=float)

# Bounds: (keep broad but safe)

kmin, kmax = np.min(k), np.max(k)

bounds = [

(1e-10, 5.0), # a >= 0

(1e-8, 10.0), # b > 0

(-0.999, 0.999), # rho in (-1,1)

(kmin-1.0, kmax+1.0), # m near data range

(1e-6, 5.0), # sigma > 0

]

# Weighted least squares objective on total variance

def obj(x):

a,b,rho,m,sigma = x

w\_hat = svi\_w\_raw(k, a,b,rho,m,sigma)

res = (w\_hat - w)

return np.mean(weights \* res \* res)

# A little regularization to avoid degenerate fits (optional)

def obj\_reg(x, lam=1e-8):

a,b,rho,m,sigma = x

return obj(x) + lam\*(b\*\*2 + sigma\*\*2)

res = minimize(obj\_reg, x0, method="L-BFGS-B", bounds=bounds, options={"maxiter": 2000})

return res.x, res.fun, res.success

def prepare\_slice\_inputs(df\_slice, S, r, q):

"""

For a single expiry slice: compute k and total variance w.

Requires columns: 'strike', 'T', 'iv\_bs'

"""

T = df\_slice["T"].to\_numpy(float)

K = df\_slice["strike"].to\_numpy(float)

iv = df\_slice["iv\_bs"].to\_numpy(float)

# Forward F = S \* exp((r - q) \* T)

F = S \* np.exp((r - q) \* T)

k = np.log(K / F)

w = np.maximum(0.0, (iv\*\*2) \* T)

return k, w

def calibrate\_svi\_per\_expiry(df\_iv, use\_weights=True):

"""

Calibrate raw-SVI per expiry in df\_iv (as built earlier).

Returns a DataFrame with columns: expiration, a,b,rho,m,sigma, n\_pts, loss

"""

# Pull S,r,q from attrs; fallback to zeros if missing

S = float(df\_iv.attrs.get("spot", np.nan))

r = float(df\_iv.attrs.get("r", 0.0))

q = float(df\_iv.attrs.get("q", 0.0))

if not np.isfinite(S) or S <= 0:

raise ValueError("Spot missing. Set df\_iv.attrs['spot'] or pass S.")

# Clean: keep finite IV/T/strike

data = df\_iv.copy()

data = data[np.isfinite(data["iv\_bs"]) & np.isfinite(data["T"]) & np.isfinite(data["strike"])]

data = data[(data["iv\_bs"] > 0) & (data["T"] > 0)]

out = []

for exp, grp in data.groupby("expiration"):

if len(grp) < 5:

continue # need enough points

k, w = prepare\_slice\_inputs(grp, S, r, q)

# Optional weights: prefer liquid quotes

weights = None

if use\_weights:

if "openInterest" in grp.columns:

# higher OI -> more weight

weights = 1.0 + np.asarray(grp["openInterest"].fillna(0.0), float)

elif {"bid","ask"}.issubset(grp.columns):

# tighter spreads -> more weight

spr = (grp["ask"] - grp["bid"]).astype(float).clip(lower=1e-4).to\_numpy()

weights = 1.0 / spr

else:

weights = None

(a,b,rho,m,sigma), loss, ok = fit\_svi\_slice(k, w, weights)

out.append({"expiration": exp, "a": a, "b": b, "rho": rho, "m": m, "sigma": sigma,

"n\_pts": len(grp), "loss": loss, "success": bool(ok)})

res\_df = pd.DataFrame(out).sort\_values("expiration")

return res\_df

# ---- Example usage (after you've computed iv\_df with compute\_iv\_surface\_inputs) ----

iv\_df = pd.read\_csv("option\_chain\_SPY\_with\_IV.csv")

# If attrs were lost when saving CSV, restore them:

iv\_df.attrs["spot"] = 645.31

iv\_df.attrs["r"] = 0.043

iv\_df.attrs["q"] = 0.011

params = calibrate\_svi\_per\_expiry(iv\_df, use\_weights=True)

print(params[["expiration","a","b","rho","m","sigma","n\_pts","loss","success"]].to\_string(index=False))

Here are the outputs for the first few expiration dates (rounded),

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Expiration | a | b |  | m |  |
| 2025/08/25 | 1e-10 | 0.0514 | 0.4903 | 0.0270 | 1e-6 |
| 2025/08/26 | 1e-10 | 0.0457 | 0.4957 | 0.0218 | 0.00454 |
| 2025/08/27 | 1e-10 | 0.0429 | 0.4958 | 0.0277 | 0.00445 |
| 2025/08/28 | 1e-10 | 0.0506 | 0.4957 | 0.0262 | 0.00296 |

1. **Conclusion**

In the report, we calculated implied volatilities by the option chain data SPY. We interpolated the data to construct a smooth surface by RBF method and visualized implied volatility across strike and expiry. We further analyzed the data by the parametric model SVI. The SVI parameters were fitted by nonlinear least squares optimization under specific constraints.