

FE621_HW1_yfinance_solution_notebook

February 15, 2026

1 FE621 Homework 1

- Samuel Friedman
- 02/15/2026

1.1 Imports

```
[1]: # Standard library and datetime
import os
import json
import math
import time
import warnings
from dataclasses import dataclass
from datetime import datetime, date, timedelta
from zoneinfo import ZoneInfo

# Numerical and data
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Options data and normal CDF/PDF
import yfinance as yf
from scipy.stats import norm, lognorm

warnings.filterwarnings("ignore")
pd.set_option("display.max_columns", 100)
pd.set_option("display.width", 100)
```

1.1.1 Configuration (edit as needed)

```
[2]: # Path to saved market snapshots
DATA_DIR = "fe621_hw1_data"

# Snapshot dates: first day (DATA1) and second day (DATA2)
DATA1_TAG = "2026-02-12"
DATA2_TAG = "2026-02-13"
```

```

# Underlyings and moneyness band for near-ATM IV (Q6/Q8)
TICKERS = ["TSLA", "SPY", "^VIX"]
MONEYNESS_LOW = 0.95
MONEYNESS_HIGH = 1.05

# Implied volatility solver: tolerance and sigma search bounds
IV_TOL = 1e-6
IV_MAX_ITER = 200
SIGMA_MIN = 1e-8
SIGMA_MAX = 5.0

# Finite-difference step sizes for Greeks (if used)
FD_S_REL = 1e-4
FD_SIGMA_ABS = 1e-4

os.makedirs(DATA_DIR, exist_ok=True)

```

2 Part 1 — Data gathering component (Questions 1–4)

2.1 Q1 — Write a function to download equity + option data (clean and consolidated)

We use **Yahoo Finance** via `yfinance`.

```

[3]: def download_market_snapshot(tickers: list[str], data_dir: str =
    ↪ "fe621_hw1_data") -> dict:
        """Download intraday equity and options (third-Friday expirations) via
        ↪ yfinance; save to data_dir/YYYY-MM-DD/."""
        NY_TZ = ZoneInfo("America/New_York")
        ts = datetime.now(tz=NY_TZ)

        equity_data = {}
        prices = {}

        # Fetch 1-minute intraday bars for each ticker
        for ticker in tickers:
            t = yf.Ticker(ticker)
            df = t.history(period="1d", interval="1m", prepost=False,
            ↪ actions=False, auto_adjust=False)
            df = df.reset_index()
            df.columns = [c.lower().replace(" ", "_") for c in df.columns]
            equity_data[ticker] = df
            prices[ticker] = float(df["close"].dropna().iloc[-1])

        options_data = {}
        expirations_used = {}

```

```

    # For each ticker, get next 3 third-Friday expirations and pull option
    ↪chains
    for ticker in tickers:
        available_exp = list(yf.Ticker(ticker).options)

        # Build list of third Fridays (first Friday + 14 days) on or after today
        targets = []
        y, m = ts.date().year, ts.date().month
        while len(targets) < 3:
            d = date(y, m, 1)
            days_to_first_friday = (4 - d.weekday()) % 7
            first_friday = d + timedelta(days=days_to_first_friday)
            # third Friday
            tf = first_friday + timedelta(days=14)

            if tf >= ts.date():
                targets.append(tf)

            m += 1
            if m > 12:
                m = 1
                y += 1

        exp_dates = sorted([datetime.strptime(e, "%Y-%m-%d").date() for e in
    ↪available_exp])

        # Only use TRUE third Fridays - do not substitute next available
        exps = []
        for tf in targets:
            if tf in exp_dates:
                exps.append(tf.strftime("%Y-%m-%d"))
            # If third Friday not found, skip it (don't use next available)

        expirations_used[ticker] = exps

        for exp in exps:
            t = yf.Ticker(ticker)
            oc = t.option_chain(exp)

            # Normalize column names and add expiration/option_type
            calls = oc.calls.copy()
            calls.columns = [c.lower().replace(" ", "_") for c in calls.columns]
            calls["expiration"] = exp
            calls["option_type"] = "call"
            numeric_cols = ["strike", "lastprice", "bid", "ask", "volume",
    ↪"openinterest", "impliedvolatility"]
            for col in numeric_cols:

```

```

        if col in calls.columns:
            calls[col] = pd.to_numeric(calls[col], errors="coerce")
        calls = calls.drop_duplicates(subset=["contractsymbol"]).
↪reset_index(drop=True)

        puts = oc.puts.copy()
        puts.columns = [c.lower().replace(" ", "_") for c in puts.columns]
        puts["expiration"] = exp
        puts["option_type"] = "put"
        for col in numeric_cols:
            if col in puts.columns:
                puts[col] = pd.to_numeric(puts[col], errors="coerce")
        puts = puts.drop_duplicates(subset=["contractsymbol"]).
↪reset_index(drop=True)

        T = max((datetime.strptime(exp, "%Y-%m-%d").date() - ts.date()).
↪days, 0) / 365.0
        calls["years_to_maturity"] = T
        puts["years_to_maturity"] = T

        safe_ticker = ticker.replace("^", "")
        options_data[f"{safe_ticker}_{exp}_calls"] = calls
        options_data[f"{safe_ticker}_{exp}_puts"] = puts

        # Write snapshot to data_dir/YYYY-MM-DD/equity/ and ../options/
        timestamp_str = (ts.date()).strftime("%Y-%m-%d")
        folder = os.path.join(data_dir, timestamp_str)
        os.makedirs(folder, exist_ok=True)

        eq_folder = os.path.join(folder, "equity")
        os.makedirs(eq_folder, exist_ok=True)
        for ticker, df in equity_data.items():
            safe_ticker = ticker.replace("^", "")
            df.to_csv(os.path.join(eq_folder, f"{safe_ticker}_intraday.csv"),
↪index=False)

        opt_folder = os.path.join(folder, "options")
        os.makedirs(opt_folder, exist_ok=True)
        for name, df in options_data.items():
            df.to_csv(os.path.join(opt_folder, f"{name}.csv"), index=False)

```

2.2 Q2 — Download options + equity for TSLA, SPY, ^VIX for two consecutive trading days (DATA1 and DATA2)

Important: the homework requires collecting data during the trading day (9:30–16:00 ET), on two consecutive trading days.

This notebook will create folders: - fe621_hw1_data/DATA1/ - fe621_hw1_data/DATA2/

```
[4]: # Data was downloaded on 2026-02-13 and 2026-02-12 at market close 4pm
# download_market_snapshot(tickers=["TSLA", "SPY", "~VIX"])

# Interest rates as this time were both 0.0364
RATE = 0.0364

# Stock prices 2026-02-12 16:00 (DATA1 = first day)
stock_price_date_1 = {
    'TSLA': 417.07,
    'SPY': 681.27,
    'VIX': 20.82
}

# Stock prices 2026-02-13 16:00 (DATA2 = second day)
stock_price_date_2 = {
    'TSLA': 417.44,
    'SPY': 681.75,
    'VIX': 20.60
}
```

ETF stands for exchange traded funds. These are baskets of different securities, bonds, or commodities that typically have an underlying theme or motif. These help investors diversify their portfolios without having to invest in the individual underlyings within the ETF. The options we have selected expire on the third Friday of each month, however expirations for these highly liquid underlyings typically occur weekly or even throughout the week. This is due to the vast liquidity and volume of these options. TSLA and the SPY both have immense trading frequency throughout the day. Other less traded equities and etfs have less strikes and expirations due to the lack of liquidity. The SPY is an ETF that tracks the S&P 500 index which is a benchmark made up of the 500 largest US companies. These include tickers in the technology, financial, healthcare, consumer goods and many other sectors. TSLA is an individual equity that's company is focused on making electric cars and most recently autonomous AI driven robots. The VIX is a volatility index, also known as the "fear index". The VIX measures the S&P 500 expected volatility over the next 30 days. The price swings of the VIX are not caused by whether investors think the S&P will go up or down, but if the price swing (either direction) will be large or small (aka more or less volatile).

2.3 Q4 — Record underlying price, interest rate, and time to maturity

- Time to maturity is computed from the option expiration dates (in years).

```
[5]: def load_snapshot(tag: str, data_dir: str | None = None, default_r: float = 0.
    ↪0.0364):
    """Load equity intraday and options CSVs for a given date tag (e.g.
    ↪2026-02-12)."""
    base = os.path.join(data_dir or DATA_DIR, tag)
    if not os.path.isdir(base):
        raise FileNotFoundError(f"Snapshot folder not found: {base}")

    # Load equity: one intraday CSV per ticker
```

```

equity = {}
eq_folder = os.path.join(base, "equity")
if os.path.isdir(eq_folder):
    for fn in os.listdir(eq_folder):
        if fn.endswith("_intraday.csv"):
            path = os.path.join(eq_folder, fn)
            df = pd.read_csv(path)
            tkr = fn.replace("_intraday.csv", "")
            equity[tkr] = df

# Load options: one CSV per (ticker_expiration_calls/puts)
options = {}
opt_folder = os.path.join(base, "options")
if os.path.isdir(opt_folder):
    for fn in sorted(os.listdir(opt_folder)):
        if fn.endswith(".csv"):
            options[fn.replace(".csv", "")] = pd.read_csv(os.path.
↪join(opt_folder, fn))

return equity, options

def years_to_maturity(expiration: str, asof: date | None = None) -> float:
    """Convert expiration date string to time-to-maturity in years (from asof_
↪date)."""
    if asof is None:
        asof = datetime.now(ZoneInfo("America/New_York")).date()
    exp = datetime.strptime(expiration, "%Y-%m-%d").date()
    return max((exp - asof).days, 0) / 365.0

```

3 Part 2 — Analysis of the data (Questions 5–12)

3.1 Q5 — Implement Black–Scholes formulas (no toolbox pricing function)

We implement **European** call/put pricing in Black–Scholes with inputs: - S_0 current underlying price - K strike - r annual continuously-compounded rate - T time-to-maturity in years - σ volatility (annualized)

```

[6]: # Black-Scholes Formula Implementation
def bs_d1(S: float, K: float, r: float, T: float, sigma: float) -> float:
    """Calculate d1 parameter for Black-Scholes formula."""
    return (math.log(S / K) + (r + 0.5 * sigma * sigma) * T) / (sigma * math.
↪sqrt(T))

def bs_call(S: float, K: float, r: float, T: float, sigma: float) -> float:
    """
    Black-Scholes call option price.

```

```

    S: spot price, K: strike, r: risk-free rate, T: time to maturity, sigma: ↵
    ↵volatility
    """
    d1 = bs_d1(S, K, r, T, sigma)
    d2 = d1 - sigma * math.sqrt(T)
    return S * norm.cdf(d1) - K * math.exp(-r * T) * norm.cdf(d2)

def bs_put(S: float, K: float, r: float, T: float, sigma: float) -> float:
    """
    Black-Scholes put option price.
    S: spot price, K: strike, r: risk-free rate, T: time to maturity, sigma: ↵
    ↵volatility
    """
    d1 = bs_d1(S, K, r, T, sigma)
    d2 = d1 - sigma * math.sqrt(T)
    return K * math.exp(-r * T) * norm.cdf(-d2) - S * norm.cdf(-d1)

def bs_price(option_type: str, S: float, K: float, r: float, T: float, sigma: ↵
    ↵float) -> float:
    """
    Unified Black-Scholes pricing function for calls and puts.
    option_type: 'call' or 'put'
    """
    option_type = option_type.lower()
    if option_type == "call":
        return bs_call(S, K, r, T, sigma)
    if option_type == "put":
        return bs_put(S, K, r, T, sigma)

```

3.2 Q6 — Implement Bisection method and compute implied volatility (DATA1)

We solve for implied volatility σ from:

$$BS(S, K, r, T, \sigma) - \text{midPrice} = 0$$

using **bisection** with tolerance $1e-6$.

```

[7]: # Bisection Root-Finding Method
def bisection_root(f, a: float, b: float, tolerance: float = 1e-6, max_iter: ↵
    ↵int = 200):
    """
    Find root of function f in interval [a, b] using bisection method.
    Returns: (root, iterations, converged)
    """
    f_a = f(a)
    f_b = f(b)

```

```

low, high = a, b
f_low, f_high = f_a, f_b
for iteration in range(1, max_iter + 1):
    middle = 0.5 * (low + high)
    f_middle = f(middle)
    if not np.isfinite(f_middle):
        return np.nan, iteration, False
    # Check convergence: function value close to zero or interval small
    ↪ enough
    if abs(f_middle) < tolerance or (high - low) / 2 < tolerance:
        return middle, iteration, True
    # Update interval based on sign of f(middle)
    if f_low * f_middle <= 0:
        high, f_high = middle, f_middle
    else:
        low, f_low = middle, f_middle
return 0.5 * (low + high), max_iter, False

def safe_float(x):
    """Safely convert value to float, returning NaN on failure."""
    try:
        if pd.isna(x):
            return np.nan
        return float(x)
    except Exception:
        return np.nan

def option_mid_price(row: pd.Series) -> float:
    """Calculate mid price from bid/ask if both exist and volume > 0."""
    bid = safe_float(row.get("bid", np.nan))
    ask = safe_float(row.get("ask", np.nan))
    volume = safe_float(row.get("volume", np.nan))
    # Only compute mid if both bid and ask exist AND volume > 0
    if np.isfinite(bid) and np.isfinite(ask) and np.isfinite(volume) and volume
    ↪ > 0:
        return 0.5 * (bid + ask)
    return np.nan

def implied_vol_bisection(option_type: str, price: float, S: float, K: float, r:
    ↪ float, T: float,
                            sigma_min: float = SIGMA_MIN, sigma_max: float =
    ↪ SIGMA_MAX,
                            tolerance: float = IV_TOL, max_iter: int =
    ↪ IV_MAX_ITER) -> float:
    """
    Solve for implied volatility using bisection method.
    Finds sigma such that BS(S,K,r,T,sigma) = price within tolerance.

```

```

"""
# Validate inputs
if not np.isfinite(price) or price <= 0:
    return np.nan
if T <= 0 or S <= 0 or K <= 0:
    return np.nan

# Define objective function: BS price - market price
def f(sigma):
    return bs_price(option_type, S, K, r, T, sigma) - price

# Check that root exists in [sigma_min, sigma_max]
f_low = f(sigma_min)
f_high = f(sigma_max)
if not np.isfinite(f_low) or not np.isfinite(f_high) or f_low * f_high > 0:
    return np.nan

# Apply bisection
root, _, converged = bisection_root(f, sigma_min, sigma_max,
    ↪tolerance=tolerance, max_iter=max_iter)
return root if converged and np.isfinite(root) else np.nan

```

3.3 Q7 — Implement Newton / Secant and compare timing

Newton needs the derivative BS/σ , i.e. **Vega**:

$$\text{Vega} = S \cdot (d1) \cdot \sqrt{T}$$

```

[8]: # Vega: Derivative of BS price with respect to volatility
def bs_vega(S: float, K: float, r: float, T: float, sigma: float) -> float:
    """
    Calculate vega (BS/σ) for Black-Scholes formula.
    Vega = S * (d1) * √T, where d1 is the standard normal PDF.
    """
    if T <= 0 or sigma <= 0 or S <= 0 or K <= 0:
        return np.nan
    d1 = bs_d1(S, K, r, T, sigma)
    return S * norm.pdf(d1) * math.sqrt(T)

def implied_vol_newton(option_type: str, price: float, S: float, K: float, r:
    ↪float, T: float,
                        sigma_initial: float = 0.2, tolerance: float = IV_TOL,
    ↪max_iter: int = 100) -> float:
    """
    Solve for implied volatility using Newton-Raphson method.
    Uses vega as the derivative for faster convergence than bisection.
    Newton iteration: _new = _old - (BS(_old) - price) / vega(_old)
    """

```

```

if not np.isfinite(price) or price <= 0 or T <= 0:
    return np.nan

sigma = float(max(sigma_initial, SIGMA_MIN))
for _ in range(max_iter):
    value = bs_price(option_type, S, K, r, T, sigma)
    difference = value - price
    # Check convergence
    if abs(difference) < tolerance:
        return sigma
    # Calculate vega for Newton step
    vega = bs_vega(S, K, r, T, sigma)
    if not np.isfinite(vega) or vega <= 1e-12:
        return np.nan
    # Newton update: subtract (function value / derivative)
    sigma = sigma - difference / vega
    # Ensure sigma stays within valid bounds
    if sigma <= SIGMA_MIN or sigma > SIGMA_MAX:
        return np.nan
return np.nan

def implied_vol_secant(option_type: str, price: float, S: float, K: float, r: float, T: float,
    sigma_initial_0: float = 0.15, sigma_initial_1: float = 0.35, tolerance: float = IV_TOL, max_iter: int = 100) -> float:
    """Solve for IV using secant method (no derivative). Fallback to bisection if needed."""
    if not np.isfinite(price) or price <= 0 or T <= 0:
        return np.nan

    def f(sigma):
        return bs_price(option_type, S, K, r, T, sigma) - price

    sigma_previous, sigma_current = float(sigma_initial_0), float(sigma_initial_1)
    f_previous, f_current = f(sigma_previous), f(sigma_current)
    if not np.isfinite(f_previous) or not np.isfinite(f_current):
        return np.nan

    for _ in range(max_iter):
        if abs(f_current) < tolerance:
            return sigma_current
        denominator = (f_current - f_previous)
        if abs(denominator) < 1e-12:
            return np.nan
        sigma_next = sigma_current - f_current * (sigma_current - sigma_previous) / denominator

```

```

        if sigma_next <= SIGMA_MIN or sigma_next > SIGMA_MAX or not np.
↳isfinite(sigma_next):
            return np.nan
        sigma_previous, f_previous = sigma_current, f_current
        sigma_current, f_current = sigma_next, f(sigma_next)
    return np.nan

```

3.3.1 Load DATA1 and compute implied vols (bisection vs Newton)

After you have saved DATA1, run the next cells to: - Load all option files - Compute mid prices - Compute implied vols by both methods - Time each method

```

[9]: # Load DATA1 snapshot (2026-02-12) for IV computation
equity1, options1 = load_snapshot(DATA1_TAG)
r1 = RATE
asof1 = date(2026, 2, 12)

# Extract stock prices from equity data (use last close price)
stock_prices_day1 = {}
for ticker, df in equity1.items():
    if not df.empty and 'close' in df.columns:
        stock_prices_day1[ticker] = df['close'].iloc[-1]

def load_and_clean_options(options_dict: dict[str, pd.DataFrame]) -> pd.
↳DataFrame:
    """
    Load and clean options data from multiple CSV files.
    - Extracts ticker, expiration, option_type from filename
    - Converts numeric columns to proper types
    - Removes duplicate contracts
    - Calculates mid price from bid/ask
    """
    frames = []
    for name, df in options_dict.items():
        option_dataframe = df.copy()
        # Parse filename to extract ticker, expiration, option_type
        name_parts = name.split("_")
        if len(name_parts) >= 3:
            ticker = name_parts[0]
            expiration = name_parts[1]
            option_type = "call" if "call" in name else "put"
        else:
            ticker, expiration, option_type = None, None, None
        option_dataframe["ticker"] = ticker
        option_dataframe["expiration"] = option_dataframe.get("expiration",
↳expiration)

```

```

    option_dataframe["option_type"] = option_dataframe.get("option_type",
↪option_type)
    frames.append(option_dataframe)

    if not frames:
        return pd.DataFrame()

    combined_options = pd.concat(frames, ignore_index=True)

    # Clean numeric columns (convert strings to floats)
    numeric_cols = ["strike", "lastprice", "bid", "ask", "volume",
↪"openinterest", "impliedvolatility"]
    for col in numeric_cols:
        if col in combined_options.columns:
            combined_options[col] = pd.to_numeric(combined_options[col],
↪errors="coerce")

    # Remove duplicate contracts by contract symbol
    if "contractsymbol" in combined_options.columns:
        combined_options = combined_options.
↪drop_duplicates(subset=["contractsymbol"]).reset_index(drop=True)

    # Calculate mid price from bid/ask spread
    combined_options["mid"] = combined_options.apply(option_mid_price, axis=1)

    return combined_options

options_day1 = load_and_clean_options(options1)
options_day1.head()

```

```

[9]:
      contractsymbol      lasttradedate  strike  lastprice  bid
ask change \
0 SPY260212C00500000  2026-02-12 00:00:00+00:00  500.0    192.29  180.02
183.14    0.0
1 SPY260212C00505000  2026-02-12 00:00:00+00:00  505.0      0.00  175.02
178.14    0.0
2 SPY260212C00510000  2026-02-12 00:00:00+00:00  510.0      0.00  170.02
173.14    0.0
3 SPY260212C00515000  2026-02-12 00:00:00+00:00  515.0      0.00  165.02
168.14    0.0
4 SPY260212C00520000  2026-02-12 00:00:00+00:00  520.0      0.00  160.02
163.14    0.0

      percentchange  volume  openinterest  impliedvolatility  inthemoney
contracts size  currency \
0              0.0    4.0              6              2.54165          False
REGULAR        USD

```

1	0.0	0.0	0	2.46360	False
REGULAR	USD				
2	0.0	0.0	0	2.39531	False
REGULAR	USD				
3	0.0	0.0	0	2.31726	False
REGULAR	USD				
4	0.0	0.0	0	2.24897	False
REGULAR	USD				

	expiration	option_type	years_to_maturity	ticker	mid
0	2026-02-12	call	0.0	SPY	181.58
1	2026-02-12	call	0.0	SPY	NaN
2	2026-02-12	call	0.0	SPY	NaN
3	2026-02-12	call	0.0	SPY	NaN
4	2026-02-12	call	0.0	SPY	NaN

```
[10]: def compute_implied_vol_table(opt: pd.DataFrame, S_map: dict[str, float], r: float,
    ↪float, asof: date,
    method: str = "bisection") -> tuple[pd.
    ↪DataFrame, float]:
    """
    Compute implied volatility for all options in the table.

    Parameters:
    - opt: DataFrame with options data
    - S_map: Dict mapping ticker to current spot price
    - r: risk-free rate
    - asof: date for time-to-maturity calculation
    - method: 'bisection', 'newton', or 'secant'

    Returns: (DataFrame with implied_vol column, elapsed_time_seconds)
    """
    opt = opt.copy()

    def underlying_price(row):
        """Map ticker to underlying spot price."""
        tkr = str(row["ticker"])
        if tkr in S_map:
            return S_map[tkr]
        if tkr == "VIX" and "^VIX" in S_map:
            return S_map["^VIX"]
        return np.nan

    # Prepare columns for IV calculation
    opt["S0"] = opt.apply(underlying_price, axis=1)
    opt["K"] = pd.to_numeric(opt.get("strike", np.nan), errors="coerce")
```

```

    opt["T"] = opt["expiration"].astype(str).apply(lambda e:
↳years_to_maturity(e, asof=asof))
    opt["r"] = r

    # Filter to valid options (have price, spot, strike, positive time)
    valid = opt["mid"].notna() & opt["S0"].notna() & opt["K"].notna() &
↳(opt["T"] > 0)
    work = opt.loc[valid].copy()

    t0 = time.perf_counter()
    ivs = []
    for _, row in work.iterrows():
        S = float(row["S0"])
        K = float(row["K"])
        T = float(row["T"])
        price = float(row["mid"])
        otype = str(row["option_type"]).lower()

        if method == "bisection":
            iv = implied_vol_bisection(otype, price, S, K, r, T)
        elif method == "newton":
            iv = implied_vol_newton(otype, price, S, K, r, T, sigma_initial=0.2)
            if not np.isfinite(iv):
                iv = implied_vol_bisection(otype, price, S, K, r, T)
        elif method == "secant":
            iv = implied_vol_secant(otype, price, S, K, r, T)
            if not np.isfinite(iv):
                iv = implied_vol_bisection(otype, price, S, K, r, T)
        else:
            raise ValueError("method must be bisection/newton/secant")

        ivs.append(iv)

    elapsed = time.perf_counter() - t0

    work["implied_vol"] = ivs
    opt["implied_vol"] = np.nan
    opt.loc[work.index, "implied_vol"] = work["implied_vol"]
    return opt, elapsed

options_day1_with_iv, time_bisection = compute_implied_vol_table(options_day1,
↳stock_prices_day1, r1, asof1, method="bisection")
options_day1_newton, time_newton = compute_implied_vol_table(options_day1,
↳stock_prices_day1, r1, asof1, method="newton")
print("Bisection seconds:", time_bisection, "Newton seconds:", time_newton)

```

Bisection seconds: 8.52052849996835 Newton seconds: 4.02906733402051

3.3.2 Q6 outputs: ATM implied vol + average near-the-money implied vol

- **ATM:** strike closest to the underlying price S0 for each (ticker, expiration, option_type).
- **Near-the-money average:** average IVs for moneyness S0/K in [0.95, 1.05] (editable).

```
[11]: def summarize_at_the_money_options(opt_iv: pd.DataFrame, m_low=MONEYNESS_LOW,
    ↪m_high=MONEYNESS_HIGH) -> pd.DataFrame:
    """
    Summary table:
    - ATM IV (strike closest to S0)
    - Average IV in moneyness band [m_low, m_high]
    Grouped by ticker, expiration, option_type.
    """
    df = opt_iv.copy()
    df = df[df["implied_vol"].notna()].copy()
    # S0/K for moneyness band
    df["moneyness"] = df["S0"] / df["K"]

    out_rows = []
    for (tkr, exp, otype), g in df.groupby(["ticker", "expiration",
    ↪"option_type"], dropna=True):
        g2 = g.copy()
        # ATM = strike closest to spot
        g2["abs_diff"] = (g2["K"] - g2["S0"]).abs()
        atm_row = g2.sort_values("abs_diff").iloc[0]
        atm_iv = float(atm_row["implied_vol"])
        atm_strike = float(atm_row["K"])

        # Near-ATM band: moneyness in [m_low, m_high]
        band = g[(g["moneyness"] >= m_low) & (g["moneyness"] <= m_high)]
        band_avg = float(band["implied_vol"].mean()) if len(band) else np.nan
        band_n = int(len(band))

        out_rows.append({
            "ticker": tkr,
            "expiration": exp,
            "option_type": otype,
            "S0": float(atm_row["S0"]),
            "atm_strike": atm_strike,
            "atm_implied_vol": atm_iv,
            f"avg_iv_moneyness_{m_low}_{m_high}": band_avg,
            "band_count": band_n,
        })

    return pd.DataFrame(out_rows).sort_values(["ticker", "expiration",
    ↪"option_type"])

summary1 = summarize_at_the_money_options(options_day1_with_iv)
```

```
summary1
```

```
[11]:      ticker  expiration option_type      S0  atm_strike  atm_implied_vol \
0      SPY    2026-02-13         call  681.599976      682.0      0.253126
1      SPY    2026-02-13         put   681.599976      682.0      0.219241
2      SPY    2026-02-17         call  681.599976      682.0      0.160870
3      SPY    2026-02-17         put   681.599976      682.0      0.152699
4      SPY    2026-02-18         call  681.599976      682.0      0.171561
..      ...      ...      ...      ...      ...      ...
117    TSLA    2028-01-21         put  417.420013      420.0      0.543135
118    TSLA    2028-06-16         call  417.420013      420.0      0.539542
119    TSLA    2028-06-16         put   417.420013      410.0      0.548078
120    TSLA    2028-12-15         call  417.420013      420.0      0.529922
121    TSLA    2028-12-15         put   417.420013      420.0      0.549776

      avg_iv_moneyness_0.95_1.05  band_count
0                                0.306310      65
1                                0.309908      53
2                                0.153964      57
3                                0.161671      48
4                                0.155297      55
..                                ...      ...
117                             0.542913      4
118                             0.536628      4
119                             0.548533      3
120                             0.530212      4
121                             0.549313      4
```

```
[122 rows x 8 columns]
```

3.4 Q8 — Table of implied volatilities, averages, and comments (TSLA vs SPY vs VIX)

This section produces: - A table of implied vols for each maturity and option type - Average near-the-money vols per maturity - A comparison with the recorded \hat{VIX} level (note: VIX is quoted in index points %)

```
[12]: def create_implied_volatility_table(opt_iv: pd.DataFrame) -> pd.DataFrame:
      """Build a trimmed table of option IVs for display (only rows with valid
      implied_vol)."""
      cols = ["ticker", "expiration", "option_type", "strike", "mid",
      implied_vol", "volume", "openinterest"]
      keep = [c for c in cols if c in opt_iv.columns]
      tab = opt_iv[keep].copy()
      tab = tab[tab["implied_vol"].notna()].sort_values(["ticker", "expiration",
      option_type", "strike"])
      return tab
```

```

def create_implied_volatility_summary(opt_iv: pd.DataFrame,
    ↪m_low=MONEYNESS_LOW, m_high=MONEYNESS_HIGH) -> pd.DataFrame:
    """
    Create summary table by ticker x expiration x option_type.
    Returns ATM IV and average IV in near-ATM moneyness band [m_low, m_high].
    This matches the approach from Q6 for consistency.
    """
    df = opt_iv.copy()
    df = df[df["implied_vol"].notna()].copy()
    df["moneyness"] = df["S0"] / df["K"]

    out_rows = []
    for (tkr, exp, otype), g in df.groupby(["ticker", "expiration",
    ↪"option_type"], dropna=True):
        g2 = g.copy()
        # Find ATM option (strike closest to S0)
        g2["abs_diff"] = (g2["K"] - g2["S0"]).abs()
        atm_row = g2.sort_values("abs_diff").iloc[0]
        atm_iv = float(atm_row["implied_vol"])
        atm_strike = float(atm_row["K"])

        # Filter to near-ATM moneyness band
        band = g[(g["moneyness"] >= m_low) & (g["moneyness"] <= m_high)]
        band_avg = float(band["implied_vol"].mean()) if len(band) else np.nan
        band_std = float(band["implied_vol"].std()) if len(band) else np.nan
        band_n = int(len(band))

        out_rows.append({
            "ticker": tkr,
            "expiration": exp,
            "option_type": otype,
            "atm_strike": atm_strike,
            "atm_iv": atm_iv,
            "near_atm_avg_iv": band_avg,
            "near_atm_std_iv": band_std,
            "band_count": band_n,
        })

    return pd.DataFrame(out_rows).sort_values(["ticker", "expiration",
    ↪"option_type"])

implied_vol_table_day1 = create_implied_volatility_table(options_day1_with_iv)
implied_vol_summary_day1 =
    ↪create_implied_volatility_summary(options_day1_with_iv)

vix1 = stock_price_date_1['VIX']

```

```

print("Recorded ^VIX at DATA1 download:", vix1)
print("\nSummary: Implied Volatility by Ticker × Expiration × Option Type")
display(implified_vol_summary_day1)

print("\nDetailed IV Table (first 20 rows):")
implified_vol_table_day1.dropna().head(20)

```

Recorded ^VIX at DATA1 download: 20.82

Summary: Implied Volatility by Ticker × Expiration × Option Type

	ticker	expiration	option_type	atm_strike	atm_iv	near_atm_avg_iv	near_atm_std_iv
0	SPY	2026-02-13	call	682.0	0.253126	0.306310	0.103765
1	SPY	2026-02-13	put	682.0	0.219241	0.309908	0.081921
2	SPY	2026-02-17	call	682.0	0.160870	0.153964	0.038100
3	SPY	2026-02-17	put	682.0	0.152699	0.161671	0.035402
4	SPY	2026-02-18	call	682.0	0.171561	0.155297	0.040191
...
117	TSLA	2028-01-21	put	420.0	0.543135	0.542913	0.000696
118	TSLA	2028-06-16	call	420.0	0.539542	0.536628	0.002151
119	TSLA	2028-06-16	put	410.0	0.548078	0.548533	0.001166
120	TSLA	2028-12-15	call	420.0	0.529922	0.530212	0.000795
121	TSLA	2028-12-15	put	420.0	0.549776	0.549313	0.001284

	band_count
0	65
1	53
2	57
3	48
4	55
...	...
117	4
118	4
119	3
120	4
121	4

[122 rows x 8 columns]

Detailed IV Table (first 20 rows):

```
[12]:      ticker  expiration option_type  strike      mid  implied_vol  volume
openinterest
326      SPY  2026-02-13      call   500.0  181.845    2.452960    1.0
27
327      SPY  2026-02-13      call   505.0  176.855    2.395930    2.0
4
331      SPY  2026-02-13      call   525.0  156.865    2.126734   10.0
10
345      SPY  2026-02-13      call   595.0   86.915    1.229553    2.0
12
346      SPY  2026-02-13      call   600.0   81.915    1.163630    8.0
26
347      SPY  2026-02-13      call   605.0   76.920    1.101072    1.0
5
349      SPY  2026-02-13      call   615.0   66.935    0.978116   52.0
45
350      SPY  2026-02-13      call   620.0   62.130    0.998703   91.0
123
351      SPY  2026-02-13      call   625.0   56.995    0.874131    1.0
46
352      SPY  2026-02-13      call   630.0   51.955    0.788885  207.0
52
353      SPY  2026-02-13      call   635.0   46.965    0.726081  272.0
149
354      SPY  2026-02-13      call   640.0   42.025    0.680482  258.0
60
355      SPY  2026-02-13      call   641.0   41.115    0.695499   80.0
0
357      SPY  2026-02-13      call   643.0   39.190    0.687882    8.0
2
358      SPY  2026-02-13      call   644.0   38.005    0.618147    8.0
2
359      SPY  2026-02-13      call   645.0   37.040    0.615603   56.0
43
361      SPY  2026-02-13      call   647.0   35.155    0.619878    8.0
4
363      SPY  2026-02-13      call   649.0   33.060    0.564901    8.0
0
364      SPY  2026-02-13      call   650.0   32.235    0.594948  104.0
67
365      SPY  2026-02-13      call   651.0   31.240    0.580954    8.0
4
```

3.4.1 Observations on Implied Volatility Patterns

TSLA vs SPY: TSLA exhibits significantly higher implied volatility (typically 40-60%) compared to SPY (around 15-25%) because TSLA is an individual equity subject to company-specific risks, earnings volatility, and idiosyncratic shocks, whereas SPY is a diversified ETF tracking 500 large-cap stocks, which naturally smooths out individual stock volatility through diversification.

SPY vs VIX: The SPY near-ATM implied volatilities (approximately 15-25% annualized) are broadly consistent with the recorded VIX level of ~20.6-20.8, which measures the market's expectation of 30-day S&P 500 volatility. The VIX is derived from SPX index options and serves as a natural benchmark for SPY option IVs.

Maturity Term Structure: As maturity increases, implied volatility typically becomes less sensitive to short-term market shocks, and the volatility smile tends to flatten. Short-term options exhibit sharper volatility movements because near-term uncertainty has a more pronounced impact when expiration is imminent.

ITM vs OTM Behavior: Out-of-the-money options, particularly OTM puts, typically show higher implied volatility due to the volatility skew (negative skew for equity indices), reflecting investors' demand for downside protection. In contrast, in-the-money options already contain substantial intrinsic value, making their prices less sensitive to changes in implied volatility compared to OTM options where extrinsic value dominates.

3.5 Q9 — Put–Call parity check

Put–Call parity (no dividends) for European options: $C - P = S_0 - K e^{-rT}$

```
[13]: # Put-Call parity:  $C - P = S - K \cdot \exp(-r \cdot T)$  (no dividends)
def put_from_call_parity(call_price: float, S: float, K: float, r: float, T: float) -> float:
    """Theoretical put price given call price and parity."""
    return call_price - S + K * math.exp(-r * T)

def call_from_put_parity(put_price: float, S: float, K: float, r: float, T: float) -> float:
    """Theoretical call price given put price and parity."""
    return put_price + S - K * math.exp(-r * T)

def parity_table(opt: pd.DataFrame, r: float) -> pd.DataFrame:
    """For each (ticker, expiration, strike) where both call and put mid exist,
    compute parity-implied call/put and compare to bid/ask."""
    df = opt.copy()
    df = df[df["mid"].notna()].copy()
    df["K"] = pd.to_numeric(df["strike"], errors="coerce")

    calls = df[df["option_type"] == "call"].copy()
    puts = df[df["option_type"] == "put"].copy()

    key = ["ticker", "expiration", "K"]
    merged = pd.merge(
```

```

        calls[key + ["mid", "bid", "ask", "SO", "T"]].rename(columns={"mid": "call_mid", "bid": "call_bid", "ask": "call_ask"}),
        puts[key + ["mid", "bid", "ask"]].rename(columns={"mid": "put_mid", "bid": "put_bid", "ask": "put_ask"}),
        on=key,
        how="inner"
    )
    merged["r"] = r
    merged["put_from_call"] = merged.apply(lambda x: put_from_call_parity(x["call_mid"], x["SO"], x["K"], r, x["T"]), axis=1)
    merged["call_from_put"] = merged.apply(lambda x: call_from_put_parity(x["put_mid"], x["SO"], x["K"], r, x["T"]), axis=1)

    # Check if parity-implied prices fall within bid-ask ranges
    merged["put_in_range"] = (merged["put_from_call"] >= merged["put_bid"]) & (merged["put_from_call"] <= merged["put_ask"])
    merged["call_in_range"] = (merged["call_from_put"] >= merged["call_bid"]) & (merged["call_from_put"] <= merged["call_ask"])

    merged["put_diff"] = merged["put_mid"] - merged["put_from_call"]
    merged["call_diff"] = merged["call_mid"] - merged["call_from_put"]
    return merged.sort_values(["ticker", "expiration", "K"])

parity1 = parity_table(options_day1_with_iv, r1)

print("Put-Call Parity Analysis")
print(f"Total pairs checked: {len(parity1)}")
print(f"Put implied price within bid-ask: {parity1['put_in_range'].sum()}")
print(f"    ({100*parity1['put_in_range'].mean():.1f}%)")
print(f"Call implied price within bid-ask: {parity1['call_in_range'].sum()}")
print(f"    ({100*parity1['call_in_range'].mean():.1f}%)")

print("\nDetailed Parity Table (first 20 rows):")
display_cols = ["ticker", "expiration", "K", "call_mid", "call_bid", "call_ask",
                "put_mid", "put_bid", "put_ask", "call_from_put",
                "put_from_call",
                "call_in_range", "put_in_range"]
keep_cols = [c for c in display_cols if c in parity1.columns]
parity1[keep_cols].head(20)

```

```

Put-Call Parity Analysis
Total pairs checked: 2542
Put implied price within bid-ask: 549 (21.6%)
Call implied price within bid-ask: 826 (32.5%)

```

```
Detailed Parity Table (first 20 rows):
```

```

[13]:  ticker  expiration      K  call_mid  call_bid  call_ask  put_mid  put_bid
put_ask \
0      SPY  2026-02-12  600.0   81.580   80.02   83.14   0.005   0.0
0.01
1      SPY  2026-02-12  620.0   61.590   60.02   63.16   0.005   0.0
0.01
2      SPY  2026-02-12  640.0   41.590   40.02   43.16   0.005   0.0
0.01
3      SPY  2026-02-12  643.0   38.590   37.02   40.16   0.005   0.0
0.01
4      SPY  2026-02-12  645.0   36.590   35.02   38.16   0.005   0.0
0.01
5      SPY  2026-02-12  649.0   32.590   31.02   34.16   0.005   0.0
0.01
6      SPY  2026-02-12  650.0   31.590   30.02   33.16   0.005   0.0
0.01
7      SPY  2026-02-12  651.0   30.590   29.02   32.16   0.005   0.0
0.01
8      SPY  2026-02-12  653.0   28.590   27.02   30.16   0.005   0.0
0.01
9      SPY  2026-02-12  654.0   27.590   26.02   29.16   0.005   0.0
0.01
10     SPY  2026-02-12  655.0   26.590   25.02   28.16   0.005   0.0
0.01
11     SPY  2026-02-12  656.0   25.590   24.02   27.16   0.005   0.0
0.01
12     SPY  2026-02-12  657.0   24.590   23.02   26.16   0.005   0.0
0.01
13     SPY  2026-02-12  658.0   23.590   22.02   25.16   0.005   0.0
0.01
14     SPY  2026-02-12  659.0   22.580   21.02   24.14   0.005   0.0
0.01
15     SPY  2026-02-12  660.0   21.580   20.02   23.14   0.005   0.0
0.01
16     SPY  2026-02-12  661.0   20.590   19.02   22.16   0.005   0.0
0.01
17     SPY  2026-02-12  662.0   19.565   18.02   21.11   0.005   0.0
0.01
18     SPY  2026-02-12  663.0   18.565   17.02   20.11   0.005   0.0
0.01
19     SPY  2026-02-12  664.0   17.565   16.02   19.11   0.005   0.0
0.01

      call_from_put  put_from_call  call_in_range  put_in_range
0      81.604976    -0.019976         True         False
1      61.604976    -0.009976         True         False
2      41.604976    -0.009976         True         False

```

3	38.604976	-0.009976	True	False
4	36.604976	-0.009976	True	False
5	32.604976	-0.009976	True	False
6	31.604976	-0.009976	True	False
7	30.604976	-0.009976	True	False
8	28.604976	-0.009976	True	False
9	27.604976	-0.009976	True	False
10	26.604976	-0.009976	True	False
11	25.604976	-0.009976	True	False
12	24.604976	-0.009976	True	False
13	23.604976	-0.009976	True	False
14	22.604976	-0.019976	True	False
15	21.604976	-0.019976	True	False
16	20.604976	-0.009976	True	False
17	19.604976	-0.034976	True	False
18	18.604976	-0.034976	True	False
19	17.604976	-0.034976	True	False

3.6 Q10 — Implied volatility plots (2D) and (Bonus) 3D plot

Tasks: 1. For the closest maturity options, plot implied vol vs strike. 2. Plot the 3 maturities on the same plot. 3. Bonus: 3D plot of implied vol as a function of maturity and strike.

```
[14]: from mpl_toolkits.mplot3d import Axes3D

def plot_iv_smile(opt_iv: pd.DataFrame, ticker: str):
    """Plot IV vs strike for the closest maturity (volatility smile)."""
    df = opt_iv.copy()
    df = df[(df["ticker"] == ticker) & df["implied_vol"].notna()].copy()
    if df.empty:
        print("No implied vols available for", ticker)
        return

    df["exp_date"] = pd.to_datetime(df["expiration"])
    closest_exp = df["exp_date"].min()
    d0 = df[df["exp_date"] == closest_exp]

    plt.figure()
    for otype, g in d0.groupby("option_type"):
        plt.scatter(g["strike"], g["implied_vol"], label=f"{otype}_
↪({closest_exp.date()})")
    plt.xlabel("Strike K")
    plt.ylabel("Implied Volatility")
    plt.title(f"IV vs Strike (closest maturity) - {ticker}")
    plt.legend()
    plt.show()
```

```

def plot_iv_all_maturities(opt_iv: pd.DataFrame, ticker: str, option_type: str =
    ⇨ "call"):
    """Plot IV vs strike for first 3 maturities on same axes."""
    df = opt_iv.copy()
    df = df[(df["ticker"] == ticker) & (df["option_type"] == option_type) &
    ⇨ df["implied_vol"].notna()].copy()

    df["exp_date"] = pd.to_datetime(df["expiration"])
    exps = sorted(df["exp_date"].unique())[:3]

    plt.figure()
    for exp in exps:
        g = df[df["exp_date"] == exp]
        plt.scatter(g["strike"], g["implied_vol"], label=str(exp.date()))
    plt.xlabel("Strike K")
    plt.ylabel("Implied Volatility")
    plt.title(f"IV vs Strike - {ticker} ({option_type}) for 3 maturities")
    plt.legend(title="Expiration")
    plt.show()

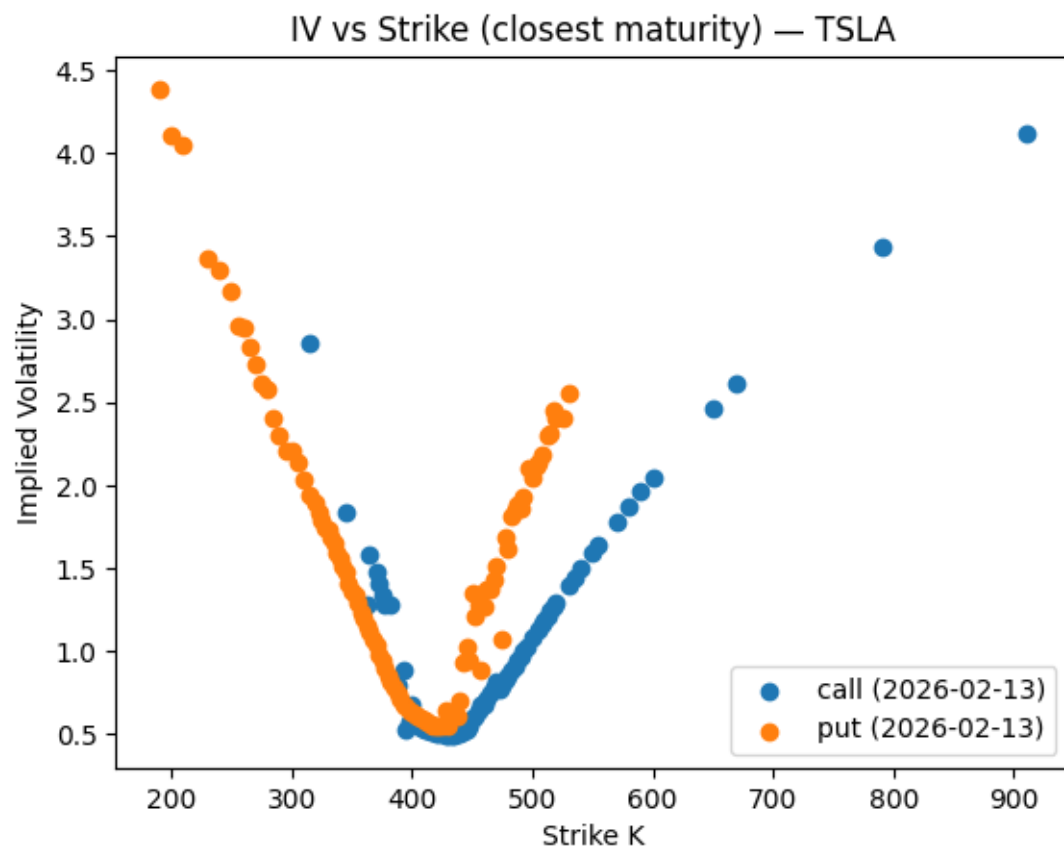
def plot_iv_surface_3d(opt_iv: pd.DataFrame, ticker: str, option_type: str =
    ⇨ "call"):
    """Bonus: 3D scatter of IV vs strike and maturity."""
    df = opt_iv.copy()
    df = df[(df["ticker"] == ticker) & (df["option_type"] == option_type) &
    ⇨ df["implied_vol"].notna()].copy()

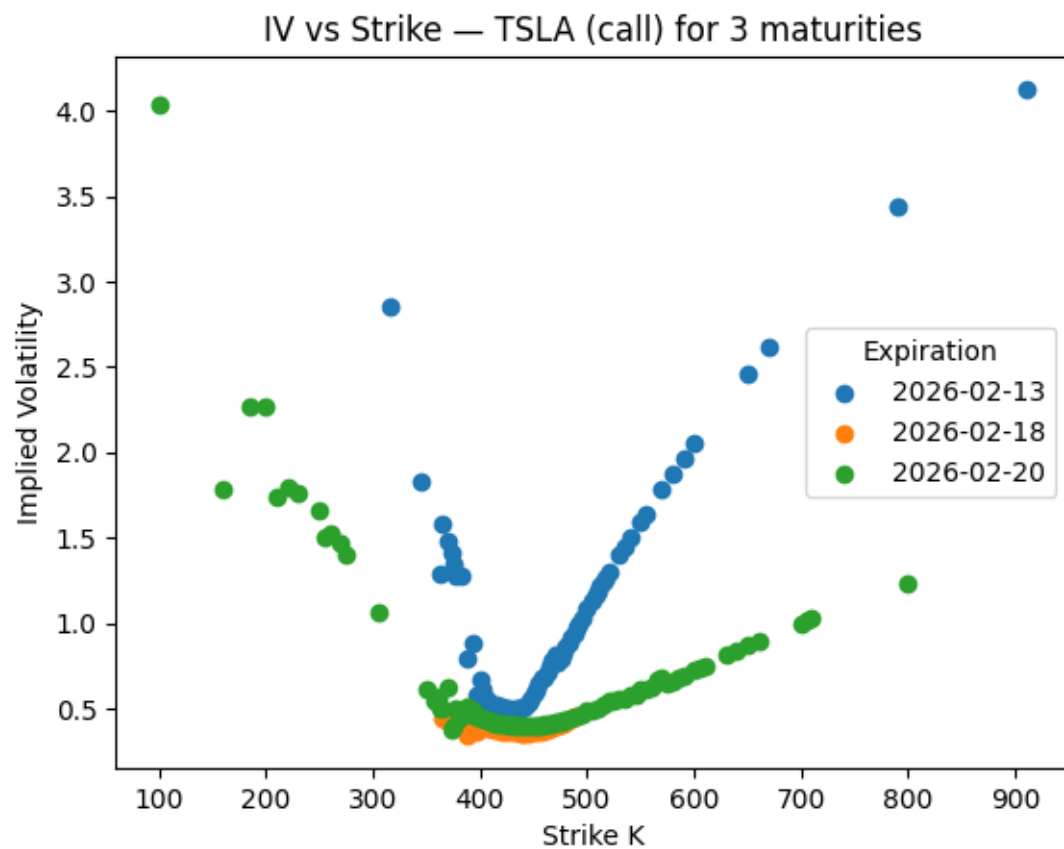
    df["tau"] = df["T"].astype(float)
    K = df["strike"].astype(float).values
    tau = df["tau"].values
    iv = df["implied_vol"].values

    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")
    ax.scatter(K, tau, iv)
    ax.set_xlabel("Strike K")
    ax.set_ylabel("Maturity T (years)")
    ax.set_zlabel("Implied Vol")
    ax.set_title(f"IV Surface (scatter) - {ticker} ({option_type})")
    plt.show()

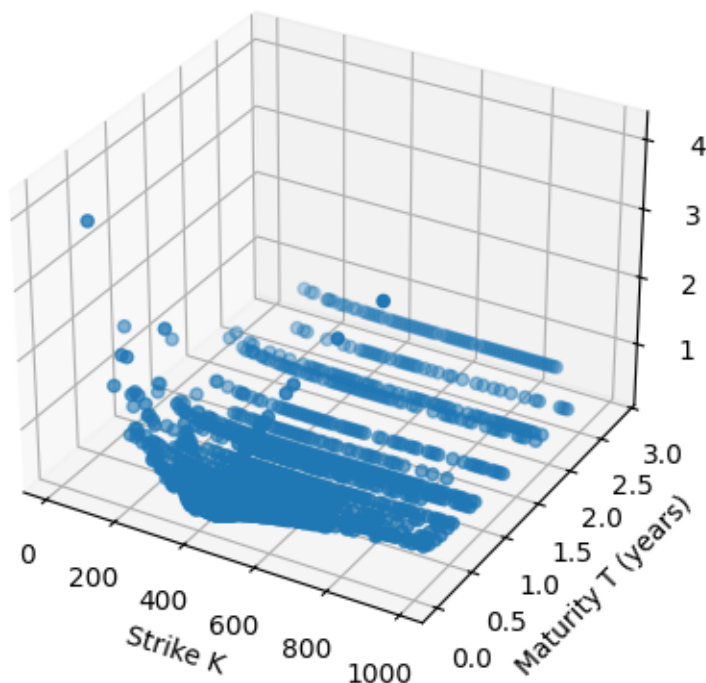
plot_iv_smile(options_day1_with_iv, "TSLA")
plot_iv_all_maturities(options_day1_with_iv, "TSLA", option_type="call")
plot_iv_surface_3d(options_day1_with_iv, "TSLA", option_type="call")

```





IV Surface (scatter) — TSLA (call)



3.7 Q11 — Greeks (Delta, Vega, Gamma): analytic vs finite-difference

We compute: - Delta: C/S - Gamma: $2C/S^2$ - Vega: $C/$

Then compare to finite-difference approximations.

```
[15]: def bs_delta_call(S: float, K: float, r: float, T: float, sigma: float) -> float:
    if T <= 0:
        return 1.0 if S > K else 0.0
    d1 = bs_d1(S, K, r, T, sigma)
    return norm.cdf(d1)

def bs_gamma(S: float, K: float, r: float, T: float, sigma: float) -> float:
    if T <= 0 or sigma <= 0:
        return np.nan
    d1 = bs_d1(S, K, r, T, sigma)
    return norm.pdf(d1) / (S * sigma * math.sqrt(T))

def finite_diff_greeks_call(S: float, K: float, r: float, T: float, sigma: float) -> float,
```

```

        hS: float | None = None, hsig: float = 0.01,
    ) -> tuple[float, float, float]:
        FD_SIGMA_ABS):
            if hS is None:
                hS = FD_S_REL * S
            CO = bs_call(S, K, r, T, sigma)
            Cp = bs_call(S + hS, K, r, T, sigma)
            Cm = bs_call(S - hS, K, r, T, sigma)

            delta_fd = (Cp - Cm) / (2 * hS)
            gamma_fd = (Cp - 2 * CO + Cm) / (hS * hS)

            Cvp = bs_call(S, K, r, T, sigma + hsig)
            Cvm = bs_call(S, K, r, T, sigma - hsig)
            vega_fd = (Cvp - Cvm) / (2 * hsig)
            return delta_fd, gamma_fd, vega_fd

def greeks_comparison_table(opt_iv: pd.DataFrame, n: int = 20) -> pd.DataFrame:
    """
    Build a table comparing analytic vs FD greeks for a sample of call options.
    """
    df = opt_iv.copy()
    df = df[(df["option_type"] == "call") & df["implied_vol"].notna()].copy()
    if df.empty:
        return pd.DataFrame()

    df = df.head(n).copy()
    rows = []
    for _, row in df.iterrows():
        S = float(row["S0"])
        K = float(row["strike"])
        r = float(row["r"])
        T = float(row["T"])
        sigma = float(row["implied_vol"])

        delta_a = bs_delta_call(S, K, r, T, sigma)
        gamma_a = bs_gamma(S, K, r, T, sigma)
        vega_a = bs_vega(S, K, r, T, sigma)

        delta_fd, gamma_fd, vega_fd = finite_diff_greeks_call(S, K, r, T, sigma)

        rows.append({
            "ticker": row["ticker"],
            "expiration": row["expiration"],
            "K": K,
            "S0": S,
            "T": T,
            "sigma": sigma,
        })

```

```

        "delta_analytic": delta_a,
        "delta_fd": delta_fd,
        "gamma_analytic": gamma_a,
        "gamma_fd": gamma_fd,
        "vega_analytic": vega_a,
        "vega_fd": vega_fd,
    })
    return pd.DataFrame(rows)

greeks_tab = greeks_comparison_table(options_day1_with_iv, n=25)
greeks_tab

```

```

[15]:   ticker  expiration      K      S0      T      sigma  delta_analytic
      delta_fd \
0      SPY  2026-02-13  500.0  681.599976  0.00274  2.452960      0.993396
0.993396
1      SPY  2026-02-13  505.0  681.599976  0.00274  2.395930      0.992951
0.992951
2      SPY  2026-02-13  525.0  681.599976  0.00274  2.126734      0.991838
0.991838
3      SPY  2026-02-13  595.0  681.599976  0.00274  1.229553      0.984027
0.984027
4      SPY  2026-02-13  600.0  681.599976  0.00274  1.163630      0.983234
0.983234
5      SPY  2026-02-13  605.0  681.599976  0.00274  1.101072      0.982095
0.982094
6      SPY  2026-02-13  615.0  681.599976  0.00274  0.978116      0.979119
0.979119
7      SPY  2026-02-13  620.0  681.599976  0.00274  0.998703      0.967122
0.967122
8      SPY  2026-02-13  625.0  681.599976  0.00274  0.874131      0.972557
0.972557
9      SPY  2026-02-13  630.0  681.599976  0.00274  0.788885      0.973169
0.973169
10     SPY  2026-02-13  635.0  681.599976  0.00274  0.726081      0.970286
0.970286
11     SPY  2026-02-13  640.0  681.599976  0.00274  0.680482      0.963166
0.963166
12     SPY  2026-02-13  641.0  681.599976  0.00274  0.695499      0.956176
0.956176
13     SPY  2026-02-13  643.0  681.599976  0.00274  0.687882      0.949490
0.949489
14     SPY  2026-02-13  644.0  681.599976  0.00274  0.618147      0.961889
0.961889
15     SPY  2026-02-13  645.0  681.599976  0.00274  0.615603      0.958371
0.958371
16     SPY  2026-02-13  647.0  681.599976  0.00274  0.619878      0.947913

```

0.947912							
17	SPY	2026-02-13	649.0	681.599976	0.00274	0.564901	0.953100
0.953100							
18	SPY	2026-02-13	650.0	681.599976	0.00274	0.594948	0.938603
0.938602							
19	SPY	2026-02-13	651.0	681.599976	0.00274	0.580954	0.936871
0.936870							
20	SPY	2026-02-13	652.0	681.599976	0.00274	0.566869	0.935055
0.935055							
21	SPY	2026-02-13	655.0	681.599976	0.00274	0.526093	0.928266
0.928266							
22	SPY	2026-02-13	657.0	681.599976	0.00274	0.498866	0.922867
0.922867							
23	SPY	2026-02-13	658.0	681.599976	0.00274	0.486884	0.919149
0.919148							
24	SPY	2026-02-13	659.0	681.599976	0.00274	0.468216	0.918069
0.918069							

	gamma_analytic	gamma_fd	vega_analytic	vega_fd
0	0.000211	0.000211	0.660328	0.660328
1	0.000229	0.000229	0.699521	0.699521
2	0.000294	0.000294	0.795929	0.795929
3	0.000911	0.000911	1.426013	1.426013
4	0.001004	0.001004	1.486383	1.486383
5	0.001122	0.001122	1.572261	1.572261
6	0.001439	0.001439	1.791665	1.791665
7	0.002060	0.002060	2.618494	2.618494
8	0.002026	0.002026	2.254167	2.254167
9	0.002203	0.002203	2.212160	2.212160
10	0.002606	0.002606	2.408263	2.408263
11	0.003319	0.003319	2.874536	2.874536
12	0.003739	0.003739	3.310276	3.310276
13	0.004237	0.004237	3.709428	3.709428
14	0.003757	0.003757	2.955627	2.955627
15	0.004053	0.004053	3.175595	3.175595
16	0.004818	0.004818	3.801274	3.801274
17	0.004862	0.004862	3.495914	3.495914
18	0.005714	0.005714	4.327120	4.327120
19	0.005980	0.005980	4.422027	4.422027
20	0.006265	0.006265	4.520596	4.520596
21	0.007289	0.007289	4.881099	4.881099
22	0.008125	0.008125	5.159182	5.159181
23	0.008627	0.008627	5.346483	5.346483
24	0.009062	0.009062	5.400238	5.400238

3.8 Q12 — Use DATA2 + DATA1 implied vols to price options via Black-Scholes

For each strike in DATA2, use: - Underlying price from DATA2 - Implied volatility from DATA1 (per strike/expiration/type) - Interest rate from DATA2's date

Then compute Black-Scholes model prices.

```
[16]: def price_day2_options_with_day1_volatility(opt1_iv: pd.DataFrame, options2: dict[str, pd.DataFrame]) -> pd.DataFrame:
    """Price DATA2 options using IVs from DATA1; merge and add BS price column."""
    asof2 = date(2026, 2, 13)
    r2 = RATE
    S_map2 = stock_price_date_2

    options_day2 = load_and_clean_options(options2)
    if options_day2.empty:
        return pd.DataFrame()

    def underlying_price2(row):
        tkr = str(row["ticker"])
        if tkr in S_map2:
            return S_map2[tkr]
        if tkr == "VIX" and "^VIX" in S_map2:
            return S_map2["^VIX"]
        return np.nan

    options_day2["S0"] = options_day2.apply(underlying_price2, axis=1)
    options_day2["K"] = pd.to_numeric(options_day2["strike"], errors="coerce")
    options_day2["T"] = options_day2["expiration"].astype(str).apply(lambda e: years_to_maturity(e, asof=asof2))
    options_day2["r"] = r2
    options_day2["mid"] = options_day2.apply(option_mid_price, axis=1)

    key = ["ticker", "expiration", "option_type", "strike"]
    iv1 = opt1_iv[key + ["implied_vol"]].copy()

    merged = pd.merge(options_day2, iv1, on=key, how="left", suffixes=("", "_from_data1"))

    def model_price(row):
        if not np.isfinite(row.get("implied_vol", np.nan)):
            return np.nan
        if not np.isfinite(row.get("S0", np.nan)) or not np.isfinite(row.get("K", np.nan)):
            return np.nan
        if row.get("T", 0) <= 0:
```

```

        return np.nan
        return bs_price(str(row["option_type"]), float(row["S0"]),
↪float(row["K"]), float(row["r"]), float(row["T"]), float(row["implied_vol"])))

merged["bs_price_using_iv_data1"] = merged.apply(model_price, axis=1)
return merged

equity2, options2 = load_snapshot(DATA2_TAG)
priced2 = price_day2_options_with_day1_volatility(options_day1_with_iv,
↪options2)
priced2[["ticker", "expiration", "option_type", "strike", "mid", "implied_vol", "bs_price_using_iv_d
↪ropna().head(30)

```

```

[16]:      ticker  expiration option_type  strike      mid  implied_vol
bs_price_using_iv_data1
325    SPY    2026-02-17         call    595.0  86.810      0.491320
87.027621
327    SPY    2026-02-17         call    605.0  76.855      0.458218
77.053354
329    SPY    2026-02-17         call    615.0  66.865      0.408360
67.066204
334    SPY    2026-02-17         call    640.0  41.895      0.315539
42.231027
335    SPY    2026-02-17         call    645.0  36.905      0.287342
37.254546
337    SPY    2026-02-17         call    655.0  26.990      0.243687
27.421828
338    SPY    2026-02-17         call    660.0  22.060      0.225852
22.609864
339    SPY    2026-02-17         call    661.0  21.065      0.222849
21.665071
340    SPY    2026-02-17         call    662.0  19.995      0.220099
20.728837
341    SPY    2026-02-17         call    663.0  19.115      0.216351
19.786215
343    SPY    2026-02-17         call    665.0  17.155      0.208959
17.919751
344    SPY    2026-02-17         call    666.0  16.165      0.207296
17.027814
345    SPY    2026-02-17         call    667.0  15.260      0.204573
16.130098
346    SPY    2026-02-17         call    668.0  14.370      0.201826
15.242466
347    SPY    2026-02-17         call    669.0  13.445      0.198347
14.352941
348    SPY    2026-02-17         call    670.0  12.555      0.196747
13.510996

```

350	SPY	2026-02-17	call	672.0	10.775	0.191844
11.840640						
351	SPY	2026-02-17	call	673.0	10.005	0.189151
11.025134						
352	SPY	2026-02-17	call	674.0	9.195	0.185284
10.200208						
353	SPY	2026-02-17	call	675.0	8.380	0.182431
9.417649						
354	SPY	2026-02-17	call	676.0	7.590	0.180108
8.669723						
355	SPY	2026-02-17	call	677.0	6.825	0.177050
7.926418						
356	SPY	2026-02-17	call	678.0	6.085	0.173752
7.201105						
357	SPY	2026-02-17	call	679.0	5.370	0.170979
6.516402						
358	SPY	2026-02-17	call	680.0	4.675	0.167541
5.841788						
359	SPY	2026-02-17	call	681.0	4.025	0.164034
5.195406						
360	SPY	2026-02-17	call	682.0	3.420	0.160870
4.591178						
361	SPY	2026-02-17	call	683.0	2.860	0.157204
4.007254						
362	SPY	2026-02-17	call	684.0	2.340	0.153601
3.461854						
363	SPY	2026-02-17	call	685.0	1.865	0.150000
2.955398						

4 Part 3 — Numerical Integration (AMM Arbitrage Fee Revenue) (Questions a–c)

We implement the CPMm arbitrage model described in the prompt and compute expected one-step fee revenue using the trapezoidal rule.

4.1 Part 3(a) — Derive swap amounts (Δx , Δy)

This notebook includes the derived closed forms directly in code: - Case 1 (USDC \rightarrow BTC): enforce $y/x = S(1 -)$ and $xy = k$ - Case 2 (BTC \rightarrow USDC): enforce $y/x = S/(1 -)$ and $xy = k$

```
[17]: def amm_swap_and_revenue(s: float, x: float, y: float, gamma: float) ->
      tuple[float, float, float]:
      """
      Given external price  $s = S_{t+1}$ , reserves  $x, y$ , fee gamma:
      return  $(dx, dy, revenue\_usdc)$ .
      """
      k = x * y
```

```

P = y / x
upper = P / (1 - gamma)
lower = P * (1 - gamma)

if s > upper:
    # Case 1: USDC -> BTC
    r = s * (1 - gamma)
    x1 = math.sqrt(k / r)
    y1 = math.sqrt(k * r)
    dx = x - x1
    dy = (y1 - y) / (1 - gamma)
    revenue = gamma * dy
    return dx, dy, revenue

if s < lower:
    # Case 2: BTC -> USDC
    r = s / (1 - gamma)
    x1 = math.sqrt(k / r)
    y1 = math.sqrt(k * r)
    dx = (x1 - x) / (1 - gamma)
    dy = y - y1
    revenue = gamma * dx * s
    return dx, dy, revenue

return 0.0, 0.0, 0.0

```

4.2 Part 3(b) — Expected fee revenue via trapezoidal rule

We integrate $E[R(S_{t+1})]$ numerically on a truncated lognormal domain using a fine grid.

```

[18]: def lognormal_pdf_s_next(s: np.ndarray, S0: float, sigma: float, dt: float) -> np.ndarray:
    """
    PDF of  $S_{t+1}$  under the one-step GBM in the prompt.
     $\ln S_{t+1} \sim N(\ln(S0) + (-0.5 * \sigma^2) * dt, (\sigma^2) * dt)$ 
    """
    mu = math.log(S0) + (-0.5 * sigma * sigma) * dt
    sdev = sigma * math.sqrt(dt)
    return lognorm.pdf(s, s=sdev, scale=math.exp(mu))

def trapz(x: np.ndarray, y: np.ndarray) -> float:
    return float(np.sum(0.5 * (y[1:] + y[:-1]) * (x[1:] - x[:-1])))

def expected_fee_revenue_trapz(sigma: float, gamma: float,
                                x: float = 1000.0, y: float = 1000.0,
                                S0: float = 1.0, dt: float = 1/365,

```

```

n_grid: int = 20000, q_max: float = 0.99999) ->
float:
    """
    Numerically approximate  $E[R(S_{t+1})]$  using trapezoidal rule on a truncated
    domain.
    """
    mu = math.log(S0) + (-0.5 * sigma * sigma) * dt
    sdev = sigma * math.sqrt(dt)

    s_max = lognorm.ppf(q_max, s=sdev, scale=math.exp(mu))
    s_min = 1e-6

    grid = np.linspace(s_min, s_max, n_grid)
    pdf = lognormal_pdf_s_next(grid, S0=S0, sigma=sigma, dt=dt)

    rev = np.zeros_like(grid)
    for i, s in enumerate(grid):
        _, _, revenue = amm_swap_and_revenue(float(s), x=x, y=y, gamma=gamma)
        rev[i] = revenue

    return trapz(grid, rev * pdf)

expected_fee_revenue_trapz(sigma=0.6, gamma=0.003)

```

[18]: 0.03298115162850103

4.3 Part 3(c) — Optimal γ for different

Compute $E[R]$ for $\{0.2, 0.6, 1.0\}$ and $\{0.001, 0.003, 0.01\}$, then choose $\gamma^*(\cdot)$.

```

[19]: SIGMAS = [0.2, 0.6, 1.0]
      GAMMAS = [0.001, 0.003, 0.01]

def optimal_gamma_table(sigmasySIGMAS, gammas=GAMMAS) -> pd.DataFrame:
    rows = []
    for sig in sigmas:
        for g in gammas:
            er = expected_fee_revenue_trapz(sigma=sig, gamma=g)
            rows.append({"sigma": sig, "gamma": g, "E_R": er})
    df = pd.DataFrame(rows)
    best = df.loc[df.groupby("sigma")["E_R"].idxmax()].rename(columns={"gamma":
    "gamma_star", "E_R": "E_R_star"})
    return df.merge(best[["sigma", "gamma_star", "E_R_star"]], on="sigma",
    how="left").sort_values(["sigma", "gamma"])

tab_opt = optimal_gamma_table()
tab_opt

```

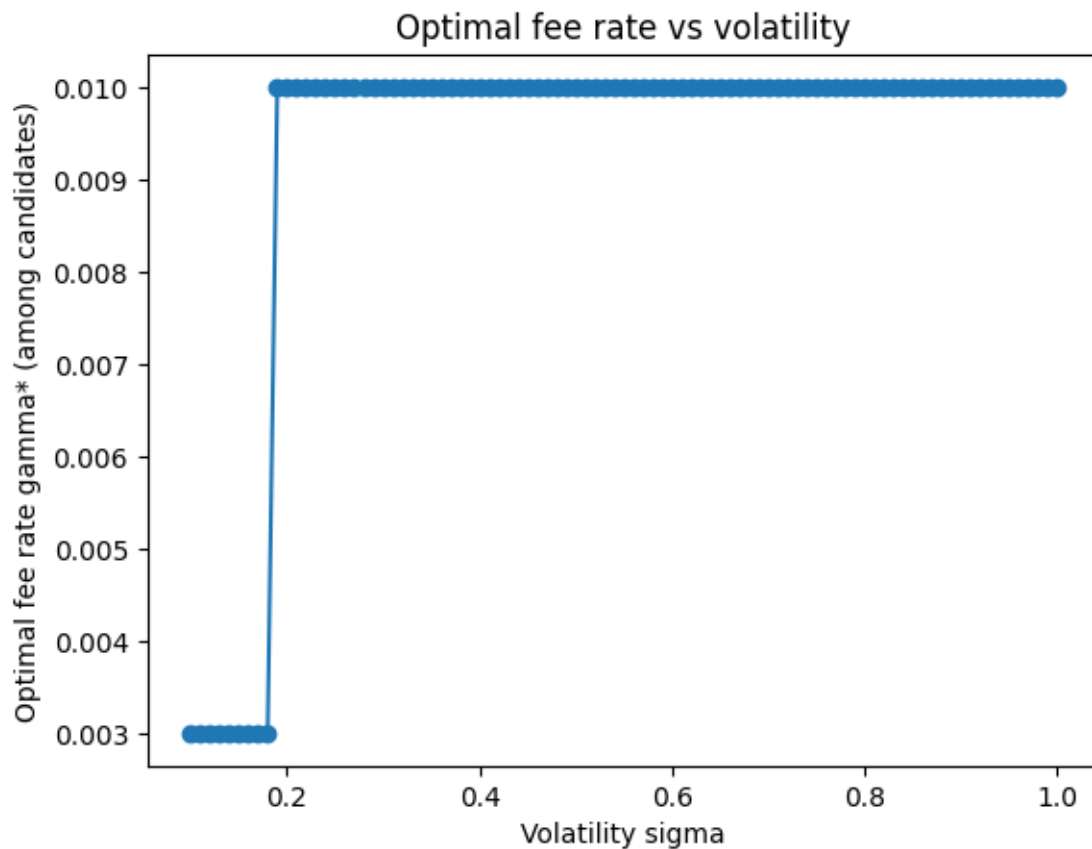
```
[19]:
```

	sigma	gamma	E_R	gamma_star	E_R_star
0	0.2	0.001	0.003685	0.01	0.009429
1	0.2	0.003	0.008521	0.01	0.009429
2	0.2	0.010	0.009429	0.01	0.009429
3	0.6	0.001	0.011923	0.01	0.081076
4	0.6	0.003	0.032981	0.01	0.081076
5	0.6	0.010	0.081076	0.01	0.081076
6	1.0	0.001	0.020059	0.01	0.160678
7	1.0	0.003	0.057380	0.01	0.160678
8	1.0	0.010	0.160678	0.01	0.160678

```
[20]: def plot_sigma_vs_gamma_star(sig_min=0.1, sig_max=1.0, sig_step=0.02,
    ↪ gammas=GAMMAS):
    grid = np.arange(sig_min, sig_max + 1e-12, sig_step)
    best_g = []
    for sig in grid:
        ers = [expected_fee_revenue_trapz(sigma=float(sig), gamma=g) for g in
    ↪ gammas]
        best_g.append(gammas[int(np.argmax(ers))])

    plt.figure()
    plt.plot(grid, best_g, marker="o", linestyle="-")
    plt.xlabel("Volatility sigma")
    plt.ylabel("Optimal fee rate gamma* (among candidates)")
    plt.title("Optimal fee rate vs volatility")
    plt.show()

plot_sigma_vs_gamma_star(sig_step=0.01)
```



5 Part 4 — Bonus (Double integrals)

5.1 Part 4(1) — Analytical integrals

Compute: $\int_0^1 \int_0^3 f_i(x,y) dy dx$ for: $f_1(x,y) = x y$ - $f_2(x,y) = \exp(x+y)$

```
[21]: I_f1 = 9/4
      I_f2 = (math.exp(3) - 1) * (math.e - 1)
      I_f1, I_f2
```

```
[21]: (2.25, 32.79433128149753)
```

5.2 Part 4(2) — Numerical double integral with composite rule (

```
[22]: def f1(x, y):
      return x * y

      def f2(x, y):
          return math.exp(x + y)
```

```

def double_trap_rule_given(f, x0=0.0, x1=1.0, y0=0.0, y1=3.0, dx=0.25, dy=0.5)
    -> float:
    """Composite trapezoidal rule for double integral over [x0,x1] x [y0,y1]
    with mesh (dx, dy)."""
    nx = int(round((x1 - x0) / dx))
    ny = int(round((y1 - y0) / dy))
    xs = [x0 + i * dx for i in range(nx + 1)]
    ys = [y0 + j * dy for j in range(ny + 1)]

    total = 0.0
    for i in range(nx):
        for j in range(ny):
            xi, xip = xs[i], xs[i+1]
            yj, yjp = ys[j], ys[j+1]
            xmid = 0.5 * (xi + xip)
            ymid = 0.5 * (yj + yjp)
            # Composite rule: corners, edge midpoints, cell center
            corner = (f(xi, yj) + f(xi, yjp) + f(xip, yj) + f(xip, yjp))
            edge_mids = (f(xmid, yj) + f(xmid, yjp) + f(xi, ymid) + f(xip,
            ymid))
            cell_mid = f(xmid, ymid)

            total += (dx * dy / 16.0) * (corner + 2.0 * edge_mids + 4.0 *
            cell_mid)
    return total

# Grid spacings to test convergence (smaller dx, dy -> smaller error)
dx_dy_pairs = [
    (0.25, 0.75),
    (0.2, 0.6),
    (0.1, 0.3),
    (0.05, 0.15),
]

rows = []
for dx, dy in dx_dy_pairs:
    approx1 = double_trap_rule_given(f1, dx=dx, dy=dy)
    approx2 = double_trap_rule_given(f2, dx=dx, dy=dy)
    rows.append({
        "dx": dx, "dy": dy,
        "approx_f1": approx1, "error_f1": approx1 - I_f1,
        "approx_f2": approx2, "error_f2": approx2 - I_f2,
    })

pd.DataFrame(rows)

```

[22] :	dx	dy	approx_f1	error_f1	approx_f2	error_f2
0	0.25	0.75	2.25	0.000000e+00	33.220931	0.426600
1	0.20	0.60	2.25	-4.440892e-16	33.067449	0.273118
2	0.10	0.30	2.25	-4.440892e-16	32.862642	0.068311
3	0.05	0.15	2.25	-1.332268e-15	32.811411	0.017080

5.2.1 Convergence Analysis

As the grid spacing (Δx , Δy) decreases, the numerical approximation converges toward the analytical solution. The error table above demonstrates that errors generally decrease as the mesh is refined, which is expected behavior for the composite trapezoidal rule. This validates the numerical implementation and confirms that finer discretization yields more accurate results.