FE621 HW1

0) Setup

```
import os
import json
import time
import math
from dataclasses import dataclass
from pathlib import Path
from datetime import datetime, date, timedelta

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import yfinance as yf
from scipy.stats import norm

from pandas_datareader import data as pdr

try:
    from zoneinfo import ZoneInfo
except Exception:
    ZoneInfo = None

pd.set_option("display.max_columns", 200)
pd.set_option("display.width", 140)

print("Imports loaded.")
```

```
Imports loaded.
```

0.1) Configuration

All important knobs live in one place to keep the notebook clean and beginner-friendly.

```
UNDERLYINGS = ["TSLA", "SPY", "^VIX"]
OPTION_UNDERLYINGS = ["TSLA", "SPY"]

MONEYNESS_BAND = (0.90, 1.10)
TOL = 1e-6

DATA_LABEL = "DATA1"

RUN_DATA_DOWNLOAD = False
```

```
IV_SCOPE = "subset"

SUBSET_STRIKES_PER_TYPE = 20

OUTPUT_ROOT = Path("./output")

print("Config loaded:", DATA_LABEL, "\n"
      "RUN_DATA_DOWNLOAD =", RUN_DATA_DOWNLOAD, "\n"
      "IV_SCOPE =", IV_SCOPE)
```

```
Config loaded: DATA1
RUN_DATA_DOWNLOAD = False
IV_SCOPE = subset
```

1) directories, timestamps, JSON

```
def make_output_dir(data_label: str) -> Path:
    out_dir = OUTPUT_ROOT / data_label
    out_dir.mkdir(parents=True, exist_ok=True)
    return out_dir


def now_et() -> datetime:

    if ZoneInfo is not None:
        try:
            return datetime.now(ZoneInfo("America/New_York"))
        except Exception:
            pass
    return datetime.now()


def now_local() -> datetime:
    return datetime.now()


def save_json(obj: dict, path: Path) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)
    with open(path, "w", encoding="utf-8") as f:
        json.dump(obj, f, indent=2, default=str)


def safe_float(x):
    try:
        return float(x)
    except Exception:
        return float("nan")
```

1.1) Interest rate via FRED

```
def get_fed_funds_effective(target_date: date) -> float:
    start = target_date - timedelta(days=10)
    end = target_date + timedelta(days=1)

    try:
        df = pdr.DataReader("DFF", "fred", start, end)
    except Exception as e:
        raise RuntimeError("failed download DFF from FRED, check
internet/installs.") from e

    df = df.dropna()
    if df.empty:
        raise RuntimeError("No DFF data returned from FRED for the
requested window")

    df = df[df.index.date <= target_date]
    if df.empty:
        last_val = float(df.iloc[0, 0])
    else:
        last_val = float(df.iloc[-1, 0])

    return last_val / 100.0
```

2) Download spot snapshots

```
# Download a 1-minute history for the current day and return the
latest close as snapshot.
def download_spot_snapshot(ticker: str) -> dict:

    t = yf.Ticker(ticker)
    hist = t.history(period="1d", interval="1m")

    if hist is None or hist.empty:
        raise RuntimeError(f"No 1m history returned for {ticker}. Try
again during market hours.")

    last_row = hist.iloc[-1]
    bar_ts = hist.index[-1]

    return {
        "ticker": ticker,
        "spot": safe_float(last_row.get("Close", np.nan)),
        "bar_timestamp": str(bar_ts),
        "download_time_et": str(now_et()),
        "download_time_local": str(now_local()),
    }
```

3) Select next 3 third-Friday monthly expirations

```
def get_next_three_third_fridays(yf_options_list: list) -> list:
    third_fridays = []
    for s in yf_options_list:
        try:
            d = datetime.strptime(s, "%Y-%m-%d").date()
        except Exception:
            continue
        if d.weekday() == 4 and 15 <= d.day <= 21:
            third_fridays.append(d)

    third_fridays = sorted(third_fridays)
    today = date.today()
    third_fridays = [d for d in third_fridays if d >= today]

    return [d.strftime("%Y-%m-%d") for d in third_fridays[:3]]
```

4) Download option chains, clean, compute mid, and time-to-maturity

```
def time_to_maturity_years(expiry: str, download_dt_et: datetime) ->
float:
    exp_date = datetime.strptime(expiry, "%Y-%m-%d").date()

    if ZoneInfo is not None:
        try:
            exp_dt = datetime(exp_date.year, exp_date.month,
exp_date.day, 16, 0, 0, tzinfo=ZoneInfo("America/New_York"))
        except Exception:
            exp_dt = datetime(exp_date.year, exp_date.month,
exp_date.day, 16, 0, 0)
    else:
        exp_dt = datetime(exp_date.year, exp_date.month, exp_date.day,
16, 0, 0)

    dl = download_dt_et
    try:
        if exp_dt.tzinfo is not None and dl.tzinfo is None and
ZoneInfo is not None:
            dl = dl.replace(tzinfo=ZoneInfo("America/New_York"))
    except Exception:
        pass

    dt_seconds = (exp_dt - dl).total_seconds()
    return max(dt_seconds, 0.0) / (365.0 * 24.0 * 3600.0)


def download_option_chain(ticker: str, expiry: str, spot: float, r:
```

```
float, download_ts_et: str) -> pd.DataFrame:
    t = yf.Ticker(ticker)
    chain = t.option_chain(expiry)

    calls = chain.calls.copy()
    puts = chain.puts.copy()
    calls["type"] = "call"
    puts["type"] = "put"

    df = pd.concat([calls, puts], ignore_index=True)

    needed = ["contractSymbol", "lastTradeDate", "strike", "bid",
"ask", "volume", "openInterest", "impliedVolatility", "type"]
    for col in needed:
        if col not in df.columns:
            df[col] = np.nan

    df["mid"] = 0.5 * (df["bid"].astype(float) +
df["ask"].astype(float))

    df = df[(df["bid"].astype(float) > 0) & (df["ask"].astype(float) >
0) & (df["volume"].fillna(0).astype(float) > 0)].copy()

    df["underlying"] = ticker
    df["expiry"] = expiry
    df["spot"] = float(spot)
    df["r"] = float(r)
    df["download_ts_et"] = download_ts_et

    dl_dt = datetime.fromisoformat(download_ts_et) if "T" in
download_ts_et else now_et()
    df["ttm"] = df["expiry"].apply(lambda e: time_to_maturity_years(e,
dl_dt))

    df = df.drop_duplicates(subset=["underlying", "expiry", "type",
"strike", "bid", "ask", "volume", "openInterest"])
    df = df.sort_values(["underlying", "expiry", "type",
"strike"]).reset_index(drop=True)

    return df
```

5) Part 1 runner: download / freeze DATA1 or DATA2

```
def run_part1_download(data_label: str) -> None:
    out_dir = make_output_dir(data_label)

    dl_et = now_et()
    dl_et_str = dl_et.isoformat()

    r = get_fed_funds_effective(dl_et.date())
```

```
    spot_rows = []
    spots = {}
    for ticker in UNDERLYINGS:
        snap = download_spot_snapshot(ticker)
        spot_rows.append(snap)
        spots[ticker] = snap["spot"]

    spots_df = pd.DataFrame(spot_rows)
    spots_path = out_dir / "spots.csv"
    spots_df.to_csv(spots_path, index=False)

    tsla_opts = yf.Ticker("TSLA").options
    expiries = get_next_three_third_fridays(tsla_opts)

    all_opts = []
    for ticker in OPTION_UNDERLYINGS:
        for exp in expiries:
            df_chain = download_option_chain(
                ticker=ticker,
                expiry=exp,
                spot=spots[ticker],
                r=r,
                download_ts_et=dl_et_str,
            )
            all_opts.append(df_chain)

    opts_df = pd.concat(all_opts, ignore_index=True) if all_opts else
pd.DataFrame()

    for ticker in OPTION_UNDERLYINGS:
        sub = opts_df[opts_df["underlying"] == ticker].copy()
        sub.to_csv(out_dir / f"options_{ticker}.csv", index=False)

    spot_panel = pd.DataFrame([{
        "download_ts_et": dl_et_str,
        "TSLA_spot": spots.get("TSLA", np.nan),
        "SPY_spot": spots.get("SPY", np.nan),
        "VIX_spot": spots.get("^VIX", np.nan),
        "r": r,
    }])
    spot_panel.to_csv(out_dir / "spot_panel.csv", index=False)

    meta = {
        "data_label": data_label,
        "download_ts_et": dl_et_str,
        "download_time_local": str(now_local()),
        "rate_source": "FRED:DFF (Effective Federal Funds Rate)",
        "r_decimal": r,
        "underlyings": UNDERLYINGS,
```

```
        "option_underlyings": OPTION_UNDERLYINGS,
        "expiries": expiries,
        "spots": spots,
        "notes": "Options filtered: bid>0, ask>0, volume>0;
mid=(bid+ask)/2.",
    }
    save_json(meta, out_dir / "meta.json")

    print(f"[DONE] Part 1 download for {data_label}")
    print("Saved:", spots_path)
    print("Saved options:", [str(out_dir / f"options_{t}.csv") for t
in OPTION_UNDERLYINGS])
    print("Saved meta:", out_dir / "meta.json")
    print("Saved spot panel:", out_dir / "spot_panel.csv")


if RUN_DATA_DOWNLOAD:
    run_part1_download(DATA_LABEL)
else:
    print("RUN_DATA_DOWNLOAD=False (no download performed).")
```

```
RUN_DATA_DOWNLOAD=False (no download performed).
```

Part 2 Black–Scholes, IV, Tables/Plots

6) Black–Scholes functions (no toolbox pricers)

```
def _d1(S, K, r, T, sigma):
    return (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma *
np.sqrt(T))

def _d2(S, K, r, T, sigma):
    return _d1(S, K, r, T, sigma) - sigma * np.sqrt(T)

# black scholes
def bs_call_price(S, K, r, T, sigma) -> float:
    if T <= 0:
        return max(S - K, 0.0)
    if sigma <= 0:
        return max(S - K*np.exp(-r*T), 0.0)

    d1 = _d1(S, K, r, T, sigma)
    d2 = d1 - sigma*np.sqrt(T)
    return float(S * norm.cdf(d1) - K * np.exp(-r*T) * norm.cdf(d2))

def bs_put_price(S, K, r, T, sigma) -> float:
    if T <= 0:
        return max(K - S, 0.0)
```

```
    if sigma <= 0:
        return max(K*np.exp(-r*T) - S, 0.0)

    d1 = _d1(S, K, r, T, sigma)
    d2 = d1 - sigma*np.sqrt(T)
    return float(K * np.exp(-r*T) * norm.cdf(-d2) - S * norm.cdf(-d1))

def bs_vega(S, K, r, T, sigma) -> float:
    if T <= 0 or sigma <= 0:
        return 0.0
    d1 = _d1(S, K, r, T, sigma)
    return float(S * norm.pdf(d1) * np.sqrt(T))
```

7) Root finding

```
def bisection(f, a, b, tol=1e-6, max_iter=200):

    t0 = time.perf_counter()
    fa = f(a)
    fb = f(b)
    if not np.isfinite(fa) or not np.isfinite(fb):
        return (np.nan, 0, (time.perf_counter()-t0)*1000.0, False)

    if fa == 0:
        return (a, 0, (time.perf_counter()-t0)*1000.0, True)
    if fb == 0:
        return (b, 0, (time.perf_counter()-t0)*1000.0, True)

    if fa * fb > 0:
        return (np.nan, 0, (time.perf_counter()-t0)*1000.0, False)

    lo, hi = a, b
    flo, fhi = fa, fb
    for i in range(1, max_iter+1):
        mid = 0.5 * (lo + hi)
        fmid = f(mid)

        if not np.isfinite(fmid):
            return (np.nan, i, (time.perf_counter()-t0)*1000.0, False)

        if abs(fmid) < tol or (hi - lo) / 2 < tol:
            return (mid, i, (time.perf_counter()-t0)*1000.0, True)

        if flo * fmid <= 0:
            hi, fhi = mid, fmid
        else:
            lo, flo = mid, fmid

    return (0.5*(lo+hi), max_iter, (time.perf_counter()-t0)*1000.0,
False)
```

```
def newton(f, fprime, x0, tol=1e-6, max_iter=50):
    t0 = time.perf_counter()
    x = float(x0)

    for i in range(1, max_iter+1):
        fx = f(x)
        dfx = fprime(x)

        if not (np.isfinite(fx) and np.isfinite(dfx)):
            return (np.nan, i, (time.perf_counter()-t0)*1000.0, False)

        if abs(dfx) < 1e-12:
            return (np.nan, i, (time.perf_counter()-t0)*1000.0, False)

        step = fx / dfx
        x_new = x - step

        if x_new <= 0:
            x_new = 1e-6
        if x_new > 5.0:
            x_new = 5.0

        if abs(fx) < tol:
            return (x, i, (time.perf_counter()-t0)*1000.0, True)

        if abs(x_new - x) < tol:
            return (x_new, i, (time.perf_counter()-t0)*1000.0, True)

        x = x_new

    return (x, max_iter, (time.perf_counter()-t0)*1000.0, False)
```

8) IV wrapper

```
def _bs_price_by_type(opt_type: str, S, K, r, T, sigma) -> float:

    if opt_type == "call":
        return bs_call_price(S, K, r, T, sigma)
    if opt_type == "put":
        return bs_put_price(S, K, r, T, sigma)
    raise ValueError(f"Unknown option type: {opt_type}")


def implied_vol(row: pd.Series):
    S = float(row["spot"])
    K = float(row["strike"])
    r = float(row["r"])
    T = float(row["ttm"])
```

```
    opt_type = str(row["type"]).lower()
    market = float(row["mid"])

    if not (np.isfinite(S) and np.isfinite(K) and np.isfinite(r) and
np.isfinite(T) and np.isfinite(market)):
        return {
            "iv_bisection": np.nan, "iters_bisection": 0,
"ms_bisection": np.nan, "ok_bisection": False,
            "iv_newton": np.nan, "iters_newton": 0, "ms_newton":
np.nan, "ok_newton": False,
        }
    if T <= 0 or market <= 0:
        return {
            "iv_bisection": np.nan, "iters_bisection": 0,
"ms_bisection": np.nan, "ok_bisection": False,
            "iv_newton": np.nan, "iters_newton": 0, "ms_newton":
np.nan, "ok_newton": False,
        }

    def f(sig):
        return _bs_price_by_type(opt_type, S, K, r, T, sig) - market

    def fprime(sig):
        return bs_vega(S, K, r, T, sig)


    a, b = 1e-6, 5.0
    fa, fb = f(a), f(b)
    if np.isfinite(fa) and np.isfinite(fb) and fa * fb > 0:
        b_try = b
        for _ in range(5):
            b_try *= 2.0
            if b_try > 10.0:
                break
            fb2 = f(b_try)
            if np.isfinite(fb2) and fa * fb2 <= 0:
                b = b_try
                break

    iv_b, it_b, ms_b, ok_b = bisection(f, a, b, tol=TOL, max_iter=200)

    guess = 0.6 if str(row["underlying"]).upper() == "TSLA" else 0.3
    iv_n, it_n, ms_n, ok_n = newton(f, fprime, guess, tol=TOL,
max_iter=50)

    return {
        "iv_bisection": iv_b, "iters_bisection": it_b, "ms_bisection":
ms_b, "ok_bisection": ok_b,
```

```
        "iv_newton": iv_n, "iters_newton": it_n, "ms_newton": ms_n,
"ok_newton": ok_n,
    }
```

9) Load DATA1 options and compute IV tables and plots

```
def load_options(data_label: str) -> pd.DataFrame:
    out_dir = make_output_dir(data_label)
    frames = []
    for ticker in OPTION_UNDERLYINGS:
        p = out_dir / f"options_{ticker}.csv"
        if p.exists():
            frames.append(pd.read_csv(p))
        else:
            print("Missing file:", p)
    if not frames:
        raise RuntimeError(f"No options files found under {out_dir}.
Run Part 1 downloads first.")
    return pd.concat(frames, ignore_index=True)


def restrict_to_subset_near_spot(df: pd.DataFrame, strikes_per_type:
int = 20) -> pd.DataFrame:
    df = df.copy()
    df["abs_moneyness_gap"] = (df["strike"].astype(float) -
df["spot"].astype(float)).abs()

    keep = []
    for (u, e, t), g in df.groupby(["underlying", "expiry", "type"],
sort=False):

keep.append(g.sort_values("abs_moneyness_gap").head(strikes_per_type))

    out = pd.concat(keep,
ignore_index=True).drop(columns=["abs_moneyness_gap"])
    return out


def compute_iv_table(data_label: str) -> pd.DataFrame:
    out_dir = make_output_dir(data_label)
    df = load_options(data_label)

    df["mid"] = 0.5 * (df["bid"].astype(float) +
df["ask"].astype(float))

    df["moneyness"] = df["spot"].astype(float) /
df["strike"].astype(float)

    if IV_SCOPE.lower() == "subset":
        df = restrict_to_subset_near_spot(df,
```

```
strikes_per_type=SUBSET_STRIKES_PER_TYPE)

    results = []
    for i, (_, row) in enumerate(df.iterrows(), start=1):
        results.append(implied_vol(row))
        if i % 200 == 0:
            print(f"IV progress: {i}/{len(df)}")

    res_df = pd.DataFrame(results)
    out = pd.concat([df.reset_index(drop=True), res_df], axis=1)

    iv_path = out_dir / "iv_table.csv"
    out.to_csv(iv_path, index=False)
    print("Saved:", iv_path)
    return out


def compute_iv_summary(iv_df: pd.DataFrame, data_label: str) ->
pd.DataFrame:
    out_dir = make_output_dir(data_label)
    lo, hi = MONEYNESS_BAND

    rows = []
    for (u, e, t), g in iv_df.groupby(["underlying", "expiry",
"type"], sort=False):
        S0 = float(g["spot"].iloc[0])
        g = g.copy()
        g["abs_strike_gap"] = (g["strike"].astype(float) - S0).abs()

        atm = g.sort_values("abs_strike_gap").iloc[0]
        atm_iv = float(atm["iv_bisection"]) if
np.isfinite(atm["iv_bisection"]) else np.nan

        g_band = g[(g["moneyness"] >= lo) & (g["moneyness"] <= hi)]
        avg_iv = float(g_band["iv_bisection"].mean()) if not
g_band.empty else np.nan

        rows.append({
            "underlying": u,
            "expiry": e,
            "type": t,
            "spot": S0,
            "atm_strike": float(atm["strike"]),
            "atm_iv_bisection": atm_iv,
            "avg_iv_bisection_band": avg_iv,
            "moneyness_band": f"[{lo}, {hi}]",
            "n_in_band": int(len(g_band)),
        })

    summary = pd.DataFrame(rows)
```

```
    path = out_dir / "iv_summary.csv"
    summary.to_csv(path, index=False)
    print("Saved:", path)
    return summary


def plot_smile_nearest_maturity(iv_df: pd.DataFrame, data_label: str)
-> None:
    out_dir = make_output_dir(data_label)

    for u in OPTION_UNDERLYINGS:
        g_u = iv_df[iv_df["underlying"] == u].copy()
        if g_u.empty:
            continue

        expiries = sorted(g_u["expiry"].unique())
        nearest = expiries[0]
        g = g_u[g_u["expiry"] == nearest]

        plt.figure()
        for t in ["call", "put"]:
            gt = g[g["type"] == t]
            plt.scatter(gt["strike"], gt["iv_bisection"], label=t)

        plt.xlabel("Strike K")
        plt.ylabel("Implied Vol (bisection)")
        plt.title(f"{u} — IV Smile (Nearest Expiry {nearest})")
        plt.legend()
        plt.tight_layout()

        out_path = out_dir / f"iv_smile_nearest_{u}.png"
        plt.savefig(out_path, dpi=300, bbox_inches="tight")
        plt.close()
        print("Saved:", out_path)


def plot_smiles_all_maturities(iv_df: pd.DataFrame, data_label: str) -
> None:
    out_dir = make_output_dir(data_label)

    for u in OPTION_UNDERLYINGS:
        g_u = iv_df[iv_df["underlying"] == u].copy()
        if g_u.empty:
            continue

        expiries = sorted(g_u["expiry"].unique())

        plt.figure()
        for e in expiries:
            ge = g_u[(g_u["expiry"] == e) & (g_u["type"] == "call")]
```

```
            plt.scatter(ge["strike"], ge["iv_bisection"], label=e)

        plt.xlabel("Strike K")
        plt.ylabel("Implied Vol (bisection)")
        plt.title(f"{u} — IV vs Strike (Calls) for 3 maturities")
        plt.legend(title="Expiry")
        plt.tight_layout()

        out_path = out_dir / f"iv_smile_3maturities_{u}.png"
        plt.savefig(out_path, dpi=300, bbox_inches="tight")
        plt.close()
        print("Saved:", out_path)


iv_df_data1 = compute_iv_table("DATA1")
summary_data1 = compute_iv_summary(iv_df_data1, "DATA1")
plot_smile_nearest_maturity(iv_df_data1, "DATA1")
plot_smiles_all_maturities(iv_df_data1, "DATA1")
```

```
IV progress: 200/240
Saved: output\DATA1\iv_table.csv
Saved: output\DATA1\iv_summary.csv
Saved: output\DATA1\iv_smile_nearest_TSLA.png
Saved: output\DATA1\iv_smile_nearest_SPY.png
Saved: output\DATA1\iv_smile_3maturities_TSLA.png
Saved: output\DATA1\iv_smile_3maturities_SPY.png
```

10) Put–Call parity check

```
def put_call_parity_check(iv_df: pd.DataFrame, data_label: str) ->
pd.DataFrame:
    out_dir = make_output_dir(data_label)

    cols = ["underlying", "expiry", "strike", "type", "mid", "spot",
"r", "ttm"]
    df = iv_df[cols].copy()

    calls = df[df["type"] == "call"].rename(columns={"mid":
"call_mid"})
    puts = df[df["type"] == "put"].rename(columns={"mid": "put_mid"})

    merged = pd.merge(calls, puts, on=["underlying", "expiry",
"strike"], how="inner", suffixes=("_call", "_put"))

    S = merged["spot_call"].astype(float)
    K = merged["strike"].astype(float)
    r = merged["r_call"].astype(float)
    T = merged["ttm_call"].astype(float)

    discK = K * np.exp(-r * T)
```

```
    merged["put_from_call"] = merged["call_mid"] - (S - discK)
    merged["call_from_put"] = merged["put_mid"] + (S - discK)

    merged["put_error"] = merged["put_from_call"] - merged["put_mid"]
    merged["call_error"] = merged["call_from_put"] -
merged["call_mid"]

    out_path = out_dir / "parity_table.csv"
    merged.to_csv(out_path, index=False)
    print("Saved:", out_path)
    return merged


parity_df = put_call_parity_check(iv_df_data1, "DATA1")
```

```
Saved: output\DATA1\parity_table.csv
```

11) Greeks

```
def bs_call_delta(S, K, r, T, sigma) -> float:
    if T <= 0 or sigma <= 0:
        return 1.0 if S > K else 0.0
    d1 = _d1(S, K, r, T, sigma)
    return float(norm.cdf(d1))

def bs_call_gamma(S, K, r, T, sigma) -> float:
    if T <= 0 or sigma <= 0:
        return 0.0
    d1 = _d1(S, K, r, T, sigma)
    return float(norm.pdf(d1) / (S * sigma * np.sqrt(T)))

def finite_diff_greeks(S, K, r, T, sigma, hS=0.1, hV=1e-4):
    C = bs_call_price(S, K, r, T, sigma)
    C_up = bs_call_price(S + hS, K, r, T, sigma)
    C_dn = bs_call_price(S - hS, K, r, T, sigma)

    delta_fd = (C_up - C_dn) / (2*hS)
    gamma_fd = (C_up - 2*C + C_dn) / (hS**2)

    C_vup = bs_call_price(S, K, r, T, sigma + hV)
    C_vdn = bs_call_price(S, K, r, T, sigma - hV)
    vega_fd = (C_vup - C_vdn) / (2*hV)

    return float(delta_fd), float(gamma_fd), float(vega_fd)

def greeks_table(iv_df: pd.DataFrame, data_label: str) ->
pd.DataFrame:
    out_dir = make_output_dir(data_label)
```

```
    df = iv_df[(iv_df["type"] == "call") &
np.isfinite(iv_df["iv_bisection"])].copy()

    rows = []
    for _, row in df.iterrows():
        S = float(row["spot"])
        K = float(row["strike"])
        r = float(row["r"])
        T = float(row["ttm"])
        sig = float(row["iv_bisection"])

        delta_a = bs_call_delta(S, K, r, T, sig)
        gamma_a = bs_call_gamma(S, K, r, T, sig)
        vega_a = bs_vega(S, K, r, T, sig)

        hS = 0.5 if str(row["underlying"]).upper() == "TSLA" else 0.1
        delta_fd, gamma_fd, vega_fd = finite_diff_greeks(S, K, r, T,
sig, hS=hS, hV=1e-4)

        rows.append({
            "underlying": row["underlying"],
            "expiry": row["expiry"],
            "strike": K,
            "spot": S,
            "r": r,
            "ttm": T,
            "sigma_iv": sig,
            "delta_analytic": delta_a,
            "delta_fd": delta_fd,
            "gamma_analytic": gamma_a,
            "gamma_fd": gamma_fd,
            "vega_analytic": vega_a,
            "vega_fd": vega_fd,
        })

    out = pd.DataFrame(rows)
    out_path = out_dir / "greeks_table.csv"
    out.to_csv(out_path, index=False)
    print("Saved:", out_path)
    return out

greeks_df = greeks_table(iv_df_data1, "DATA1")
```

```
Saved: output\DATA1\greeks_table.csv
```

12) DATA2 pricing using DATA1 IVs

```
def price_data2_using_data1_iv() -> pd.DataFrame:
    d1_dir = make_output_dir("DATA1")
    d2_dir = make_output_dir("DATA2")
```

```
    iv_path = d1_dir / "iv_table.csv"
    if not iv_path.exists():
        raise RuntimeError("Missing DATA1 iv_table.csv. Run DATA1 IV
computations first.")

    iv_df = pd.read_csv(iv_path)
    iv_df = iv_df[np.isfinite(iv_df["iv_bisection"])].copy()



    d2_opts = load_options("DATA2")
    d2_opts["mid"] = 0.5 * (d2_opts["bid"].astype(float) +
d2_opts["ask"].astype(float))
    d2_opts = d2_opts[(d2_opts["bid"].astype(float) > 0) &
(d2_opts["ask"].astype(float) > 0) &
(d2_opts["volume"].fillna(0).astype(float) > 0)].copy()

    key_cols = ["underlying", "expiry", "type", "strike"]

    merged = pd.merge(d2_opts, iv_df[key_cols + ["iv_bisection"]],
on=key_cols, how="left")

    prices = []
    for _, row in merged.iterrows():
        sig = row.get("iv_bisection", np.nan)
        if not np.isfinite(sig):
            prices.append(np.nan)
            continue

        S = float(row["spot"])
        K = float(row["strike"])
        r = float(row["r"])
        T = float(row["ttm"])
        opt_type = str(row["type"]).lower()

        prices.append(_bs_price_by_type(opt_type, S, K, r, T,
float(sig)))

    merged["bs_price_using_data1_iv"] = prices
    merged["pricing_error_vs_mid"] = merged["bs_price_using_data1_iv"]
- merged["mid"].astype(float)

    out_path = d2_dir / "data2_pricing_comparison.csv"
    merged.to_csv(out_path, index=False)
    print("Saved:", out_path)
    return merged

data2_pricing_df = price_data2_using_data1_iv()
```

```
Saved: output\DATA2\data2_pricing_comparison.csv
```

Part 3 AMM Numerical Integration

```
def lognormal_pdf(s: np.ndarray, m: float, v: float) -> np.ndarray:
    s = np.asarray(s)
    out = np.zeros_like(s, dtype=float)
    mask = s > 0
    out[mask] = (1.0 / (s[mask] * np.sqrt(2*np.pi*v))) * np.exp(-
(np.log(s[mask]) - m)**2 / (2*v))
    return out


def trapz_integral(x: np.ndarray, y: np.ndarray) -> float:
    x = np.asarray(x, dtype=float)
    y = np.asarray(y, dtype=float)
    if len(x) < 2:
        return 0.0
    dx = np.diff(x)
    return float(np.sum(0.5 * (y[:-1] + y[1:]) * dx))


def amm_post_trade_reserves_case1(s: float, x: float, y: float, gamma:
float):
    k = x*y
    x_new = math.sqrt(k / (s * (1 - gamma)))
    y_new = math.sqrt(k * s * (1 - gamma))
    return x_new, y_new


def amm_post_trade_reserves_case2(s: float, x: float, y: float, gamma:
float):

    k = x*y
    x_new = math.sqrt(k * (1 - gamma) / s)
    y_new = math.sqrt(k * s / (1 - gamma))
    return x_new, y_new


def swap_amounts_case1(s: float, x: float, y: float, gamma: float):

    x_new, y_new = amm_post_trade_reserves_case1(s, x, y, gamma)
    dx = x - x_new
    dy = (y_new - y) / (1 - gamma)
    return dx, dy


def swap_amounts_case2(s: float, x: float, y: float, gamma: float):

    x_new, y_new = amm_post_trade_reserves_case2(s, x, y, gamma)
    dx = (x_new - x) / (1 - gamma)
    dy = y - y_new
```

```
    return dx, dy


def one_step_fee_revenue(s: float, x: float, y: float, gamma: float) -
> float:

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

    if s > upper:
        _, dy = swap_amounts_case1(s, x, y, gamma)
        return gamma * dy
    if s < lower:
        dx, _ = swap_amounts_case2(s, x, y, gamma)
        return gamma * dx * s
    return 0.0


def expected_fee_revenue(sigma: float, gamma: float, n_grid: int =
20000) -> float:

    x = 1000.0
    y = 1000.0
    P = y/x
    dt = 1/365.0
    St = 1.0

    v = (sigma**2) * dt
    m = math.log(St) + (-0.5 * sigma**2 * dt)

    upper_band = P / (1 - gamma)
    lower_band = P * (1 - gamma)

    z = 8.0
    s_min = math.exp(m - z*math.sqrt(v))
    s_max = math.exp(m + z*math.sqrt(v))

    # Region 1
    if upper_band < s_max:
        s1 = np.linspace(upper_band, s_max, n_grid)
        f1 = lognormal_pdf(s1, m, v)
        R1 = np.array([one_step_fee_revenue(float(si), x, y, gamma)
for si in s1])
        I1 = trapz_integral(s1, R1 * f1)
    else:
        I1 = 0.0

    # Region 2
    if s_min < lower_band:
```

```
        s2 = np.linspace(s_min, lower_band, n_grid)
        f2 = lognormal_pdf(s2, m, v)
        R2 = np.array([one_step_fee_revenue(float(si), x, y, gamma)
for si in s2])
        I2 = trapz_integral(s2, R2 * f2)
    else:
        I2 = 0.0

    return float(I1 + I2)


def amm_gamma_sweep():

    out_dir = make_output_dir("AMM")

    gammas = [0.001, 0.003, 0.01]
    sigmas = [0.2, 0.6, 1.0]

    rows = []
    for s in sigmas:
        ers = []
        for g in gammas:
            er = expected_fee_revenue(float(s), float(g), n_grid=5000)

            ers.append(er)
            rows.append({"sigma": s, "gamma": g, "E_R": er})
        g_star = gammas[int(np.argmax(ers))]
        rows.append({"sigma": s, "gamma": "BEST", "E_R":
float(np.max(ers)), "gamma_star": g_star})

    table = pd.DataFrame(rows)
    table_path = out_dir / "amm_ER_table.csv"
    table.to_csv(table_path, index=False)
    print("Saved:", table_path)

    sigma_grid = np.round(np.arange(0.10, 1.001, 0.01), 2)
    stars = []
    for s in sigma_grid:
        ers = [expected_fee_revenue(float(s), float(g), n_grid=2000)
for g in gammas]
        g_star = gammas[int(np.argmax(ers))]
        stars.append({"sigma": float(s), "gamma_star": g_star})

    stars_df = pd.DataFrame(stars)
    stars_path = out_dir / "sigma_vs_gamma_star.csv"
    stars_df.to_csv(stars_path, index=False)
    print("Saved:", stars_path)

    plt.figure()
    plt.plot(stars_df["sigma"], stars_df["gamma_star"])
```

```
    plt.xlabel("Volatility sigma")
    plt.ylabel("Optimal fee gamma*")
    plt.title("Optimal fee rate vs volatility")
    plt.tight_layout()
    plot_path = out_dir / "sigma_vs_gamma_star.png"
    plt.savefig(plot_path, dpi=300, bbox_inches="tight")
    plt.close()
    print("Saved:", plot_path)

    return table, stars_df

amm_table, amm_star = amm_gamma_sweep()
```

```
Saved: output\AMM\amm_ER_table.csv
Saved: output\AMM\sigma_vs_gamma_star.csv
Saved: output\AMM\sigma_vs_gamma_star.png
```

Bonus — Part 4 Double Integral

```
def f1(x, y):
    return x*y

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

def analytic_integrals():
    I1 = 9/4
    I2 = (math.e**3 - 1) * (math.e - 1)
    return float(I1), float(I2)


def composite_trapezoid_double(f, dx, dy, x0=0.0, x1=1.0, y0=0.0,
y1=3.0):
    n = int(round((x1 - x0) / dx))
    m = int(round((y1 - y0) / dy))
    xs = np.linspace(x0, x1, n+1)
    ys = np.linspace(y0, y1, m+1)

    total = 0.0
    for i in range(n):
        for j in range(m):
            xi, xi1 = xs[i], xs[i+1]
            yj, yj1 = ys[j], ys[j+1]
            xm = 0.5*(xi + xi1)
            ym = 0.5*(yj + yj1)

            c00 = f(xi,  yj)
            c01 = f(xi,  yj1)
```

```
            c10 = f(xi1, yj)
            c11 = f(xi1, yj1)

            e1 = f(xm, yj)
            e2 = f(xm, yj1)
            e3 = f(xi, ym)
            e4 = f(xi1, ym)

            cc = f(xm, ym)

            cell = (dx*dy/16.0) * (c00 + c01 + c10 + c11 +
2.0*(e1+e2+e3+e4) + 4.0*cc)
            total += cell

    return float(total)


def run_bonus_double_integral():
    out_dir = make_output_dir("BONUS")
    I1_true, I2_true = analytic_integrals()

    pairs = [(0.2, 0.5), (0.1, 0.3), (0.05, 0.2), (0.025, 0.1)]

    rows = []
    for dx, dy in pairs:
        I1_hat = composite_trapezoid_double(f1, dx, dy)
        I2_hat = composite_trapezoid_double(f2, dx, dy)
        rows.append({
            "dx": dx, "dy": dy,
            "I1_hat": I1_hat, "I1_true": I1_true, "I1_error": I1_hat -
I1_true,
            "I2_hat": I2_hat, "I2_true": I2_true, "I2_error": I2_hat -
I2_true,
        })

    df = pd.DataFrame(rows)
    path = out_dir / "double_integral_trapz_results.csv"
    df.to_csv(path, index=False)
    print("Saved:", path)
    return df


bonus_df = run_bonus_double_integral()
```

```
Saved: output\BONUS\double_integral_trapz_results.csv
```