PART 1.

Market data for TSLA, SPY and VIX was downloaded using yfinance at one-minute interval. The download time was recorded in Eastern Time to make sure the pricing snapshot is clearly defined. This time stamp was later used for time-to-maturity calculation.

The overnight federal funds rate (DFF series) was retrieved from FRED API. The latest available value was converted into decimal form for pricing purpose. This rate was used as the risk-free rate in all Black–Scholes calculations. For each ticker, option chains were downloaded and filtered to keep the next three monthly expirations. Both calls and puts were stored in a single dataframe with expiration and type labels added. Basic cleaning was applied by removing zero volume contracts and invalid bid–ask quotes.

Equity data, option chains, and metadata were saved as separate CSV files. The metadata file contains download time, interest rate, and spot prices for reference. These files serves as the input for the subsequent implied volatility and pricing analysis. Process can be seen from python script part 1.

PART 2.

**(5) Black–Scholes implementation**

The Black–Scholes pricing formula was coded using the standard expressions for $d_1$ and $d_2$. For options close to maturity the function returns intrinsic value, which avoids numerical issue. This function were then used for the rest of the calculations. Function can be seen in part 2 python script.

**(6) Implied volatility (Bisection)**

```
TSLA ATM Avg IV: 0.43560493115482235
SPY ATM Avg IV: 0.47840976454718365
```

Implied volatility was obtained by solving the pricing equation numerically with bisection method. Mid prices was used as market price and time to maturity was computed from the download date. ATM average implied volatility were calculated using strikes within 5% of the spot.

**(7) Newton vs Bisection**

```
Bisection IV: 0.43171452454480536
Newton IV: 0.4317144923031262
Bisection time: 0.0022115707397460938
Newton time: 0.0002205371856689453
```

Newton method was implemented using Vega as the derivative term. Method converged to similar implied volatility values for ATM options. In practice Newton was faster while bisection are more stable.

**(8) Term structure and comparison**

```
TSLA Maturity IV:
   expiration  implied_vol
0  2026-02-20     1.625995
1  2026-03-20     1.228434
2  2026-04-17     1.033763

SPY Maturity IV:
   expiration  implied_vol
0  2026-02-20     1.095968
1  2026-03-20     0.513791
2  2026-04-17     0.262036
```

```
TSLA Call vs Put:
   expiration  type  implied_vol
0  2026-02-20  call     1.095791
1  2026-02-20   put     2.025570
2  2026-03-20  call     1.106902
3  2026-03-20   put     1.360171
4  2026-04-17  call     0.800139
5  2026-04-17   put     1.283839

SPY Call vs Put:
   expiration  type  implied_vol
0  2026-02-20  call     0.626915
1  2026-02-20   put     1.526394
2  2026-03-20  call     0.540976
3  2026-03-20   put     0.488860
4  2026-04-17  call     0.231669
5  2026-04-17   put     0.294832
```

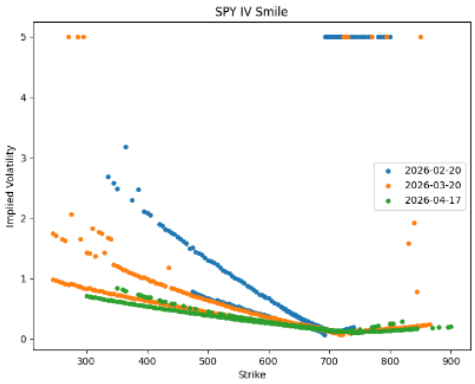
Implied volatilities was grouped by expiration to see how they change across maturities. The results shows that volatility is not flat over time. Comparing with VIX level gives an idea how different the index volatility can be.

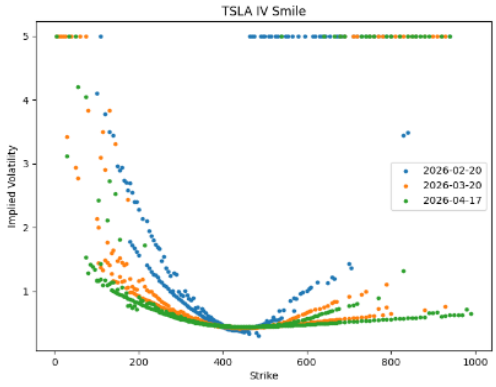
**(9) Put–call parity check**

```
TSLA Parity Mean Diff: 1.633133587121295
SPY Parity Mean Diff: 1.0666944448616866
```

Put–call parity was checked by comparing $C - P$to $S - Ke^{-rT}$for matching strikes and maturities. The average difference are small but not exactly zero. This likely comes from bid–ask spread and small market frictions.

**(10) Volatility smile**




Implied volatility was plotted against strike for each expiration. The graph show a curved shape instead of flat line. This is consistent with what is usually observed in equity option market.

**(11) Greeks comparison**

| | Greek | Analytic | Finite_Diff |
|---|---|---|---|
| 0 | Delta | 0.526577 | 0.526577 |
| 1 | Gamma | 0.014986 | 0.014990 |
| 2 | Vega | 24.491117 | 24.491117 |

Delta, Gamma, and Vega was computed analytically from the Black–Scholes formula. Finite difference approximations were then calculated to compare with the analytic results. The two approaches gives very similar values which confirms the implementation.

**(12) Comparison using DATA2**

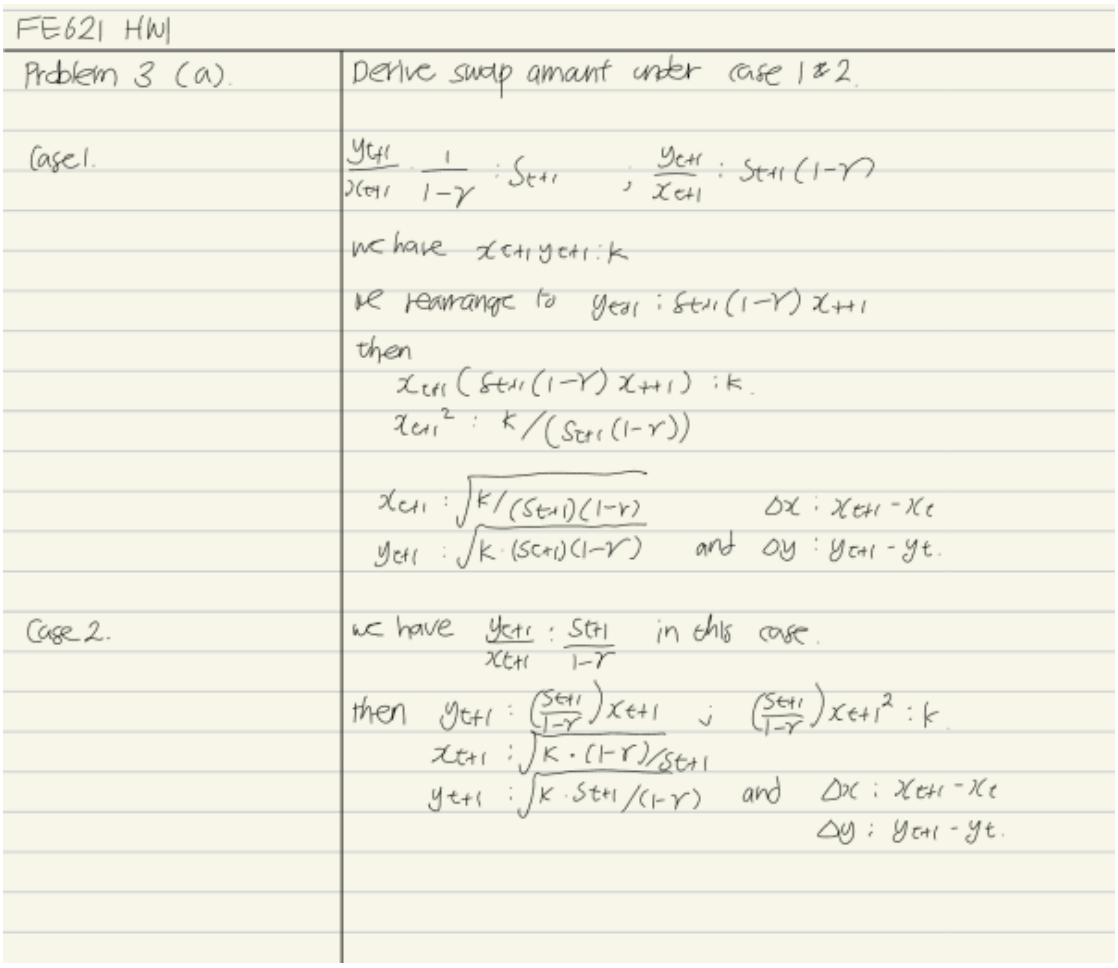
| | strike | expiration | type | theoretical_price | market_price | pricing_error |
|---|---|---|---|---|---|---|
| 0 | 100.0 | 2026-02-20 | call | 321.743582 | 323.425 | -1.681418 |
| 1 | 110.0 | 2026-02-20 | call | 312.959563 | 327.000 | -14.040437 |
| 2 | 120.0 | 2026-02-20 | call | 301.868656 | 303.375 | -1.506344 |
| 3 | 130.0 | 2026-02-20 | call | 291.838681 | 293.375 | -1.536319 |
| 4 | 140.0 | 2026-02-20 | call | 281.965190 | 283.225 | -1.259810 |

The second dataset was merged with DATA1 by matching strike, expiration and option type. Using the implied volatility from DATA1, theoretical Black–Scholes prices was computed for DATA2 market conditions. The comparison shows that the theoretical prices does not perfectly match the new market prices, which reflects market movement and model limitation.

PART 3.

a) Derivation process included in appendix (a)
b) The expectation was evaluated numerically under the assumption that $S_{t+1}$follows one-step GBM. The lognormal distribution was expressed using a standard normal transformation, and the integral was approximated using the trapezoidal rule over a truncated grid, -4 to 4. Result of trapezoidal rule shows the expected fee of 0.01307. this can be seen in result appendix (b)
c) For each volatility level $0.2$,$0.6$,$1.0$, the expected fee was computed for fee candidates $0.001$, $0.003$, $0.01$. The optimal fee was selected by comparing the numerical expectation values and choosing the maximum. The procedure was then extended over a grid of volatility values in 0.1 to 1, and the result was plotted to analyze how optimal fee varies with volatility. Although theory may suggest an interior optimal fee, the numerical results select the highest candidate γ in the tested set for all volatility levels, which may reflect the simplified integration approach.

Appendix.




Appendix (a). Part3, part a derivation process for case 1 and case 2.

```
C:\Users\sungw\PycharmProjects\FE621HW1\.venv\Scripts\python.exe C:\Users\sungw\PycharmProjects\FE621HW1\part3.py
Expected fee: 0.01307555586884058
Sigma 0.2: best gamma = 0.01
Sigma 0.6: best gamma = 0.01
Sigma 1.0: best gamma = 0.01
   sigma  gamma       E_R
0    0.2  0.001  0.004198
1    0.2  0.003  0.013076
2    0.2  0.010  0.060069
3    0.6  0.001  0.012549
4    0.6  0.003  0.037876
5    0.6  0.010  0.132927
6    1.0  0.001  0.020926
7    1.0  0.003  0.062994
8    1.0  0.010  0.214949
```

Appendix (b). Part 3 calculation result form python

Python Code Part 1.

```
import yfinance as yf
import pandas as pd
from datetime import datetime
from fredapi import Fred
import pytz

SYMBOLS = ["TSLA", "SPY", "^VIX"]
OUTPUT_PREFIX = "DATA2"
TIMEZONE = pytz.timezone("US/Eastern")

# Verifying time of data import.
now = datetime.now(TIMEZONE)
print("Download Time (ET):", now)

# Using FRED to import interest rate data from H15
FRED_API_KEY = "0fa4f50d7dcb9b1075c13c9e3a24c732 "
fred = Fred(api_key=FRED_API_KEY)
rate_series = fred.get_series_latest_release("DFF")
interest_rate = rate_series.iloc[-1] / 100
print("Interest Rate (decimal):", interest_rate)

# Equity and option data generating functions.
def download_equity(symbol):
    ticker = yf.Ticker(symbol)
    hist = ticker.history(period="1d", interval="1m")
    current_price = hist.iloc[-1]["Close"]
    return current_price, hist

def get_monthly_expirations(ticker):
     expirations = ticker.options
     monthly = []

     for exp in expirations:
         date_obj = datetime.strptime(exp, "%Y-%m-%d")
         if date_obj.weekday() == 4 and 15 <= date_obj.day <= 21:
             monthly.append(exp)
     return monthly[:3]

def download_option_chain(symbol):
    ticker = yf.Ticker(symbol)
    expirations = get_monthly_expirations(ticker)
    all_options = []

    for exp in expirations:
        chain = ticker.option_chain(exp)

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

```
        all_options.append(calls)
        all_options.append(puts)

    df = pd.concat(all_options, ignore_index=True)

    # cleaning data
    df = df.drop_duplicates()
    df = df[(df["volume"] > 0)]
    df = df[(df["bid"] > 0) & (df["ask"] > 0)]
    df = df.reset_index(drop=True)

    return df

# generating prices for given symbols above.
equity_prices = {}

for sym in SYMBOLS:
    price, hist = download_equity(sym)
    equity_prices[sym] = price
    hist.to_csv(f"{sym}_{OUTPUT_PREFIX}_equity.csv")
    print(f"{sym} price:", price)

# generating option chains for next 3 expiry
for sym in ["TSLA", "SPY"]:
    options_df = download_option_chain(sym)
    options_df.to_csv(f"{sym}_{OUTPUT_PREFIX}_options.csv", index=False)
    print(f"{sym} options saved.")

# Saving data in csv file format
meta = pd.DataFrame({
    "Download_Time_ET": [now],
    "Interest_Rate": [interest_rate],
    "TSLA_price": [equity_prices["TSLA"]],
    "SPY_price": [equity_prices["SPY"]],
    "VIX_price": [equity_prices["^VIX"]]
})

meta.to_csv(f"metadata_{OUTPUT_PREFIX}.csv", index=False)
```

Python code Part 2.

```
import pandas as pd
import numpy as np
from scipy.stats import norm
from datetime import datetime
import time
import matplotlib.pyplot as plt

#========================= Key Functions ========================
# black_scholes function / ##Problem 5 requirement##
def black_scholes(S, K, T, r, sigma, option_type):

    if T <= 0:
        return max(S-K,0) if option_type=="call" else max(K-S,0)

    d1 = (np.log(S/K) + (r + 0.5*sigma**2)*T) / (sigma*np.sqrt(T))
    d2 = d1 - sigma*np.sqrt(T)

    if option_type == "call":
        return S*norm.cdf(d1) - K*np.exp(-r*T)*norm.cdf(d2)
    else:
        return K*np.exp(-r*T)*norm.cdf(-d2) - S*norm.cdf(-d1)

# time to maturity calculation
def time_to_maturity(expiration, current_time):

    exp = datetime.strptime(expiration, "%Y-%m-%d")
    delta = exp - current_time
    return max(delta.days/365, 1e-8)

# Mid-price function
def mid_price(row):
    return (row["bid"] + row["ask"]) / 2

# implied vol calculation function
def implied_vol_bisection(S, K, T, r, market_price, option_type):
    tol = 1e-6
    max_iter = 100
    lower = 1e-6
    upper = 5.0
    for i in range(max_iter):
        mid = (lower + upper) / 2
        price = black_scholes(S, K, T, r, mid, option_type)
        diff = price - market_price
        if abs(diff) < tol:
            return mid
        price_low = black_scholes(S, K, T, r, lower, option_type)
        if (price_low - market_price) * diff < 0:
            upper = mid
        else:
            lower = mid
    return mid

def compute_iv_for_dataframe(df, S, r, current_time):
```

```
    iv_list = []
    for _, row in df.iterrows():
        K = row["strike"]
        option_type = row["type"]
        market_price = mid_price(row)
        T = time_to_maturity(row["expiration"], current_time)

        iv = implied_vol_bisection(S, K, T, r, market_price, option_type)
        iv_list.append(iv)
    df["implied_vol"] = iv_list
    return df

# Vega calculation
def vega(S, K, T, r, sigma):
    if T <= 0:
        return 0
    d1 = (np.log(S/K) + (r + 0.5*sigma**2)*T) / (sigma*np.sqrt(T))
    return S * np.sqrt(T) * norm.pdf(d1)

# Newton Methodi
def implied_vol_newton(S, K, T, r, market_price, option_type):
    tol = 1e-6
    max_iter = 100
    sigma = 0.3  # initial guess
    for i in range(max_iter):
        price = black_scholes(S, K, T, r, sigma, option_type)
        diff = price - market_price
        if abs(diff) < tol:
            return sigma
        v = vega(S, K, T, r, sigma)
        if abs(v) < 1e-8:
            break
        sigma = sigma - diff / v
        if sigma <= 0:
            sigma = 1e-6
    return sigma

# Put call parity function for data frame
def check_put_call_parity(df, S, r, current_time):
    results = []
    # separating put/calls and remerging to have both data in one row
    calls = df[df["type"] == "call"]
    puts  = df[df["type"] == "put"]
    merged = pd.merge(calls, puts, on=["strike", "expiration"],
suffixes=("_call", "_put"))

    for _, row in merged.iterrows():
        K = row["strike"]
        T = time_to_maturity(row["expiration"], current_time)
        C = (row["bid_call"] + row["ask_call"]) / 2
        P = (row["bid_put"] + row["ask_put"]) / 2

        # Parity equation
        lhs = C - P
        rhs = S - K * np.exp(-r*T)
        diff = lhs - rhs
```

```
        results.append({"strike": K,"expiration":
row["expiration"],"parity_diff": diff})
    return pd.DataFrame(results)

# gamma calculation function
def greeks_call(S, K, T, r, sigma):
    d1 = (np.log(S/K) + (r + 0.5*sigma**2)*T) / (sigma*np.sqrt(T))
    delta = norm.cdf(d1)
    gamma = norm.pdf(d1) / (S*sigma*np.sqrt(T))
    vega  = S*np.sqrt(T)*norm.pdf(d1)
    return delta, gamma, vega
def delta_fd_call(S, K, T, r, sigma, h=1e-4):
    return (black_scholes(S+h, K, T, r, sigma, "call") - black_scholes(S-h,
K, T, r, sigma, "call")) / (2*h)
def gamma_fd_call(S, K, T, r, sigma, h=1e-4):
    return (black_scholes(S+h, K, T, r, sigma, "call") - 2*black_scholes(S,
K, T, r, sigma, "call") + black_scholes(S-h, K, T, r, sigma, "call")) /
(h**2)
def vega_fd_call(S, K, T, r, sigma, h=1e-4):
    return (black_scholes(S, K, T, r, sigma+h, "call") - black_scholes(S, K,
T, r, sigma-h, "call")) / (2*h)
def compare_greeks_call(S, K, T, r, sigma):
    delta_a, gamma_a, vega_a = greeks_call(S, K, T, r, sigma)

    delta_n = delta_fd_call(S, K, T, r, sigma)
    gamma_n = gamma_fd_call(S, K, T, r, sigma)
    vega_n  = vega_fd_call(S, K, T, r, sigma)

    return pd.DataFrame({"Greek": ["Delta","Gamma","Vega"],"Analytic":
[delta_a, gamma_a, vega_a],"Finite_Diff": [delta_n, gamma_n, vega_n]})


#========================= Calculations ==========================
# Importing data 1
tsla = pd.read_csv("TSLA_DATA1_options.csv")
spy = pd.read_csv("SPY_DATA1_options.csv")
meta = pd.read_csv("metadata_DATA1.csv")

r = meta["Interest_Rate"].iloc[0]
S_tsla = meta["TSLA_price"].iloc[0]
S_spy = meta["SPY_price"].iloc[0]

# Calculating IV for data1 on the data frame. ##Problem 6 requirement##
current_time =
datetime.strptime(meta["Download_Time_ET"].iloc[0].split()[0],"%Y-%m-%d")

tsla_iv = compute_iv_for_dataframe(tsla, S_tsla, r, current_time)
spy_iv = compute_iv_for_dataframe(spy, S_spy, r, current_time)
tsla_iv["moneyness"] = S_tsla / tsla_iv["strike"]
spy_iv["moneyness"] = S_spy / spy_iv["strike"]
tsla_atm_avg = tsla_iv[(tsla_iv["moneyness"] >= 0.95) & (tsla_iv["moneyness"]
<= 1.05)]["implied_vol"].mean()
spy_atm_avg = spy_iv[(spy_iv["moneyness"] >= 0.95) & (spy_iv["moneyness"] <=
1.05)]["implied_vol"].mean()
```

```
print("TSLA ATM Avg IV:", tsla_atm_avg)
print("SPY ATM Avg IV:", spy_atm_avg)

# comparing processing time for bisection and newton methods. ##Problem 7
requirement##
atm_row = tsla_iv.iloc[(tsla_iv["strike"] - S_tsla).abs().argsort()[:1]]
row = atm_row.iloc[0]
K = row["strike"]
option_type = row["type"]
market_price = mid_price(row)
T = time_to_maturity(row["expiration"], current_time)

# Bisection
start = time.time()
iv_bis = implied_vol_bisection(S_tsla, K, T, r, market_price, option_type)
end = time.time()
bis_time = end - start
# Newton
start = time.time()
iv_new = implied_vol_newton(S_tsla, K, T, r, market_price, option_type)
end = time.time()
new_time = end - start

print("Bisection IV:", iv_bis)
print("Newton IV:", iv_new)
print("Bisection time:", bis_time)
print("Newton time:", new_time)

# Problem 8 requirement, finding average IV grouped by expiration and option
types.
tsla_summary =
tsla_iv.groupby("expiration")["implied_vol"].mean().reset_index()
spy_summary  =
spy_iv.groupby("expiration")["implied_vol"].mean().reset_index()

print("TSLA Maturity IV:")
print(tsla_summary)
print("\nSPY Maturity IV:")
print(spy_summary)

tsla_cp =
tsla_iv.groupby(["expiration","type"])["implied_vol"].mean().reset_index()
spy_cp  =
spy_iv.groupby(["expiration","type"])["implied_vol"].mean().reset_index()

print("\nTSLA Call vs Put:")
print(tsla_cp)
print("\nSPY Call vs Put:")
print(spy_cp)

VIX = meta["VIX_price"].iloc[0]
print("\nVIX level:", VIX)

#Problem 9 requirement, put call parity
tsla_parity = check_put_call_parity(tsla, S_tsla, r, current_time)
spy_parity  = check_put_call_parity(spy, S_spy, r, current_time)
```

```
print("\nTSLA Parity Mean Diff:", tsla_parity["parity_diff"].mean())
print("SPY Parity Mean Diff:", spy_parity["parity_diff"].mean())

#Problem 10 requirement, IV smile plot
plt.figure(figsize=(8,6))
for exp in tsla_iv["expiration"].unique():
    subset = tsla_iv[tsla_iv["expiration"] == exp]
    plt.scatter(subset["strike"],subset["implied_vol"],label=exp,s=10)

plt.xlabel("Strike")
plt.ylabel("Implied Volatility")
plt.title("TSLA IV Smile")
plt.legend()
plt.show()

plt.figure(figsize=(8,6))
for exp in spy_iv["expiration"].unique():
    subset = spy_iv[spy_iv["expiration"] == exp]
    plt.scatter(subset["strike"],subset["implied_vol"],label=exp, s=15)

plt.xlabel("Strike")
plt.ylabel("Implied Volatility")
plt.title("SPY IV Smile")
plt.legend()
plt.show()

#Problem 11 requirement, Greeks
atm_call = tsla_iv[(tsla_iv["type"]=="call")].iloc[(tsla_iv["strike"] -
S_tsla).abs().argsort()[:1]].iloc[0]
K = atm_call["strike"]
sigma = atm_call["implied_vol"]
T = time_to_maturity(atm_call["expiration"], current_time)
comparison_table = compare_greeks_call(S_tsla, K, T, r, sigma)
print("\n", comparison_table)

#Problem 12, calculation with data2
tsla2 = pd.read_csv("TSLA_DATA2_options.csv")
spy2  = pd.read_csv("SPY_DATA2_options.csv")
meta2 = pd.read_csv("metadata_DATA2.csv")
S_tsla_2 = meta2["TSLA_price"].iloc[0]
S_spy_2  = meta2["SPY_price"].iloc[0]
current_time_2 =
datetime.strptime(meta2["Download_Time_ET"].iloc[0].split()[0],"%Y-%m-%d")

r2 = meta2["Interest_Rate"].iloc[0]
merged = pd.merge(tsla_iv[["strike", "expiration", "type",
"implied_vol"]],tsla2,on=["strike", "expiration", "type"],suffixes=("_data1",
"_data2"))
theoretical_prices = []
market_prices = []

for _, row in merged.iterrows():
    K = row["strike"]
    sigma = row["implied_vol"]
    option_type = row["type"]
    T = time_to_maturity(row["expiration"], current_time_2)
```

```
    price = black_scholes(S_tsla_2, K, T, r2, sigma, option_type)
    theoretical_prices.append(price)
merged["theoretical_price"] = theoretical_prices
merged["market_price"] = (merged["bid"] + merged["ask"]) / 2
merged["pricing_error"] = merged["theoretical_price"] -
merged["market_price"]

print("\n",merged[["strike","expiration","type","theoretical_price","market_p
rice","pricing_error"]].head())
```

Python Code Part 3.

```
import numpy as np
from scipy import integrate
import pandas as pd
import matplotlib.pyplot as plt

# expected fee calculation using trapezoidal rule
def expected_fee_trap(S0, x0, y0, gamma, sigma=0.2, dt=1/365, n=2000):
    k = x0 * y0
    m = -0.5*sigma**2*dt
    sd = sigma*np.sqrt(dt)

    z = np.linspace(-4, 4, n)

    phi = (1/np.sqrt(2*np.pi))*np.exp(-0.5*z**2)
    S = S0 * np.exp(m + sd*z)
    R_vals = []

    for s in S:
        x1 = np.sqrt(k / ((1-gamma)*s))
        dx = x1 - x0
        R = gamma * abs(dx)
        R_vals.append(R)

    integrand = np.array(R_vals) * phi
    expected = integrate.trapezoid(integrand, z)

    return expected

# running example with expected fee
S0 = 1
x0 = 1000
y0 = 1000
gamma = 0.003

val = expected_fee_trap(S0, x0, y0, gamma)
print("Expected fee:", val)

# (c) requirement
```

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

for s in sigmas:
    best_gamma = None
    best_val = -1
    for g in gammas:
        val = expected_fee_trap(S0, x0, y0, g, sigma=s)
        results.append((s, g, val))
        if val > best_val:
            best_val = val
            best_gamma = g
    print(f"Sigma {s}: best gamma = {best_gamma}")

df_results = pd.DataFrame(results, columns=["sigma", "gamma", "E_R"])
print(df_results)

sigma_grid = np.linspace(0.1, 1.0, 20)
gamma_star = []

for s in sigma_grid:
    vals = [expected_fee_trap(S0, x0, y0, g, sigma=s) for g in gammas]
    gamma_star.append(gammas[np.argmax(vals)])

plt.plot(sigma_grid, gamma_star)
plt.xlabel("sigma")
plt.ylabel("gamma")
plt.show()
```