FE 621 Assignment

February 15, 2026

0.1 Part 1 - Data Collection

[1]:
```
import yfinance as yf
import pandas as pd
from datetime import datetime
import csv
```

[20]:
```
symbols = ["TSLA", "SPY", "^VIX"]
option_months = 3
```

[21]:
```
def get_timestamp():
    return datetime.now()
```

[22]:
```
def clean_dataframe(df):
    df = df.drop_duplicates()
    df.columns = [c.lower().strip() for c in df.columns]
    df = df.drop_duplicates()

    date_cols = ["expiration", "download_time", "lasttradedate"]
    for col in date_cols:
        if col in df.columns:
            df[col] = pd.to_datetime(df[col])

    percent_cols = ["percentchange", "impliedvolatility"]
    for col in percent_cols:
        if col in df.columns:
            df[col] = df[col] / 100
    return df
```

[23]:
```
def is_third_friday(date_str):
    date = pd.to_datetime(date_str)
    return date.weekday() == 4 and 15 <= date.day <= 21
```

[24]:
```
def get_underlying_price(symbol):
    ticker = yf.Ticker(symbol)
    timestamp = get_timestamp()
    price = ticker.fast_info["last_price"]
    df = pd.DataFrame({
        "symbol": [symbol],
        "underlying_price": [price],
        "timestamp": [timestamp]
    })
    return clean_dataframe(df)
```

[25]:
```
def equity_data_function(symbol):
    ticker = yf.Ticker(symbol)
    price = ticker.history(period="1d", interval="1m")
    price["symbol"] = symbol
    price["download_time"] = datetime.now()

    return clean_dataframe(price.reset_index())
```

[26]:
```
def option_data_function(symbol, num_months):
    ticker = yf.Ticker(symbol)
    try:
        all_expirations = ticker.options
    except:
        return pd.DataFrame()

    if symbol == "^VIX":
        expirations = all_expirations[:num_months]
    else:
        third_fridays = [d for d in all_expirations if is_third_friday(d)]
        expirations = third_fridays[:num_months]

    option_data = []

    for exp in expirations:
        try:
            chain = ticker.option_chain(exp)
        except:
            print(f"Error downloading {symbol} {exp}")
            continue
        calls = chain.calls.copy()
        puts = chain.puts.copy()

        if calls.empty and puts.empty:
            continue
        calls["type"] = "call"
        puts["type"] = "put"
        df = pd.concat([calls, puts])
        df["expiration"] = pd.to_datetime(exp)
        df["symbol"] = symbol
        df["download_time"] = datetime.now()
        df["ttm"] = (
            df["expiration"] - df["download_time"]
        ).dt.total_seconds() / (365 * 24 * 60 * 60)
        option_data.append(df)

    return clean_dataframe(pd.concat(option_data).reset_index(drop=True))
```

[ ]:
```
#Ran on 2/11/2026 3:30 PM
def data_collection(tag="DATA1"):
    equity_list = []
    option_list = []
    snapshot_list = []
    for sym in symbols:
        snap = get_underlying_price(sym)
        eq = equity_data_function(sym)
        opt = option_data_function(sym, option_months)
        snapshot_list.append(snap)
        equity_list.append(eq)
        option_list.append(opt)

    equity_df = pd.concat(equity_list)
    DATA1 = pd.concat(option_list)
    snapshot_df = pd.concat(snapshot_list)
    equity_df.to_csv(f"{tag}_equity.csv", index=False)
    DATA1.to_csv(f"{tag}_options.csv", index=False)
    snapshot_df.to_csv(f"{tag}_underlying.csv", index=False)

data_collection("DATA1")
```

[33]:
```
#Ran on 2/12/2026 @ 2:25 PM
def data_collection(tag="DATA2"):
    equity_list = []
    option_list = []
    snapshot_list = []
    for sym in symbols:
        snap = get_underlying_price(sym)
        eq = equity_data_function(sym)
        opt = option_data_function(sym, option_months)

        snapshot_list.append(snap)
        equity_list.append(eq)
        option_list.append(opt)

    equity_df = pd.concat(equity_list)
    DATA1 = pd.concat(option_list)
    snapshot_df = pd.concat(snapshot_list)
    equity_df.to_csv(f"{tag}_equity.csv", index=False)
    DATA1.to_csv(f"{tag}_options.csv", index=False)
    snapshot_df.to_csv(f"{tag}_underlying.csv", index=False)
```

[34]:
```
data_collection("DATA2")
```

[6]:
```
DATA1 = pd.read_csv("DATA1_options.csv")
DATA1_underlying_df = pd.read_csv("DATA1_underlying.csv")
DATA1["volume"] = DATA1["volume"].fillna(0)
DATA2 = pd.read_csv("DATA2_options.csv")
underlying_data2_df = pd.read_csv("DATA2_underlying.csv")
DATA2["volume"] = DATA2["volume"].fillna(0)
r = 0.0364
r2 = 0.00364
```

There are multiple option expirations because they provide flexibility in the market. Since the market is unpredictable and constantly changing, having more maturities creates additional hedging opportunities for investors.

SPY is an ETF composed of 500 large-cap U.S. stocks, therefore its a diversified portfolio across multiple sectors. It is commonly used as an indicator of the overall health of the U.S. economy. For investors, SPY provides exposure to many different companies without needing to buy each stock individually, which helps reduce risk. TSLA is the stock for Tesla, a company that designs and manufactures electric vehicles, energy-generation products, and energy-storage systems. It is part of the consumer cyclical sector and the auto manufacturers industry. VIX is the CBOE Volatility Index, which measures the expected volatility of the S&P 500 over the next 30 days. It is viewed as a meter of investor sentiment, so readings below 20 typically indicate market stability, while readings above 30 suggest uncertainty or fear. Option symbols are formatted as so "TSLA260218C00322500". The structure includes the ticker symbol, the expiration date (year-month-day), the option type (C for call, P for put), and the strike price * 1000. For VIX, options expire 30 days before the S&P 500 option expiration and typically expire on Wednesdays. For SPY and TSLA, their monthly options expire on the third Friday of each month.

0.2 Part 2 Analysis

[7]:
```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt
import time
from math import log, sqrt, exp
from tabulate import tabulate
```

[8]:
```
def black_scholes_function(S0,K,r,sigma,T, option_type):
    d1 = (log(S0/K) + (r + 0.5 * sigma**2) *T)/(sigma * sqrt(T))
    d2 = d1 - sigma * sqrt(T)
    if (option_type == "call"):
        price = S0 * norm.cdf(d1) - K * exp(-r*T) * norm.cdf(d2)
    elif (option_type == "put"):
        price = K * norm.cdf(-d2) * exp(-r*T) - S0 * norm.cdf(-d1)
    return price
```

[9]:
```
def bisection(f,a,b, tolerance = 1e-6, max_iter = 1000):
    fa = f(a)
    fb = f(b)
    if fa * fb >= 0:
        return None
    for _ in range(max_iter):
        mid = (a + b) / 2
        f_mid = f(mid)
        if (b - a) < tolerance:
            return mid
        if fa * f_mid < 0:
            b = mid
            fb = f_mid
        else:
            a = mid
            fa = f_mid
    return (a + b)/2

def implied_vol_metrics(DATA1, DATA1_underlying_df, r):
    results = {}
    for symbol in ["TSLA", "SPY"]:
        S0 = DATA1_underlying_df.loc[
            DATA1_underlying_df["symbol"] == symbol,
            "underlying_price"
        ].values[0]
        vol_df = DATA1[DATA1["symbol"] == symbol].copy()
        vol_df = vol_df[
            (vol_df["volume"] > 0) &
            (vol_df["bid"] > 0) &
            (vol_df["ask"] > 0)
            ]
        vol_df["market_price"] = (vol_df["bid"] + vol_df["ask"]) / 2
        iv_list = []
        for _, row in vol_df.iterrows():
            K = row["strike"]
            T = row["ttm"]
            market_price = row["market_price"]
            option_type = row["type"]
            def f(sigma):
                return black_scholes_function(S0,K,r,sigma,T, option_type) -␣
 ↪market_price
            iv = bisection(f, 0.0001, 1)
            if iv is not None:
                iv_list.append(iv)
            else:
                iv_list.append(np.nan)
        vol_df["iv"] = iv_list
        vol_df = vol_df.dropna(subset=["iv"])
        atm = (vol_df["strike"] - S0).abs().idxmin()
        atm_vol = vol_df.loc[atm, "iv"]
        calls = vol_df["type"] == "call"
        puts = vol_df["type"] == "put"
        itm = (
            (calls & (vol_df["strike"] < S0)) |
            (puts & (vol_df["strike"] > S0))
        )
        otm = (
            (calls & (vol_df["strike"] > S0)) |
            (puts & (vol_df["strike"] < S0))
        )
        boundary = vol_df[
            (vol_df["strike"] >= 0.9 * S0) &
            (vol_df["strike"] <= 1.1 * S0)
        ]
        avg_boundary_vol = boundary["iv"].mean()
        avg_itm_vol =vol_df.loc[itm, "iv"].mean()
        avg_otm_vol = vol_df.loc[otm, "iv"].mean()
        results[symbol] = {
            "ATM Implied Vol": atm_vol,
            "ITM Average Implied Vol": avg_itm_vol,
            "OTM Average Implied Vol": avg_otm_vol,
        }
    return results

implied_vol_results = implied_vol_metrics(DATA1, DATA1_underlying_df, r)
bisectional_metric_results = pd.DataFrame(implied_vol_results).T
print(tabulate(bisectional_metric_results, headers="keys",␣
 ↪tablefmt="fancy_grid"))
```

```
          ATM Implied Vol     ITM Average Implied Vol     OTM Average Implied
Vol


 TSLA            0.421985                    0.567677
0.551506
```

```
 SPY            0.148551                     0.293023
0.277975
```

[10]:
```
def vega(S0,K,T,r,sigma):
    d1 = (log(S0/K) + (r + 0.5 * sigma**2) *T)/(sigma * sqrt(T))
    return S0 * norm.pdf(d1) * sqrt(T)

def newton_method(func, deriv, x0=0.1, tol=1e-6, max_iter=1000):
    x_current = x0
    for _ in range(max_iter):
        f_val = func(x_current)
        d_val = deriv(x_current)
        if abs(d_val) < 1e-6:
            return None
        x_next = x_current - f_val / d_val
        if abs(x_next - x_current) < tol:
            return x_next
        if x_next <= 0:
            return None
        x_current = x_next
    return x_current


def implied_vol_metrics_newton(DATA1, DATA1_underlying_df, r):
    results = {}
    for symbol in ["TSLA", "SPY"]:
        S0 = DATA1_underlying_df.loc[
            DATA1_underlying_df["symbol"] == symbol,
            "underlying_price"
        ].values[0]
        vol_df = DATA1[DATA1["symbol"] == symbol].copy()
        vol_df = vol_df[
            (vol_df["volume"] > 0) &
            (vol_df["bid"] > 0) &
            (vol_df["ask"] > 0)
        ]
        vol_df["market_price"] = (vol_df["bid"] + vol_df["ask"]) / 2
        iv_list = []
        for _, row in vol_df.iterrows():
            K = row["strike"]
            T = row["ttm"]
            market_price = row["market_price"]
            option_type = row["type"]
            def f(sigma):
                return black_scholes_function(S0, K, r, sigma, T, option_type)␣
↪- market_price
            def f_newton(sigma):
                return vega(S0, K, r, sigma, T)
            iv = newton_method(f, f_newton, x0=0.1)
            if iv is not None:
                iv_list.append(iv)
            else:
                iv_list.append(np.nan)
        vol_df["iv"] = iv_list
        vol_df = vol_df.dropna(subset=["iv"])
        atm_newton = (vol_df["strike"] - S0).abs().idxmin()
        atm_vol_newton = vol_df.loc[atm_newton, "iv"]
        calls = vol_df["type"] == "call"
        puts = vol_df["type"] == "put"
        itm_newton = (
            (calls & (vol_df["strike"] < S0)) |
            (puts & (vol_df["strike"] > S0))
        )
        otm_newton = (
            (calls & (vol_df["strike"] > S0)) |
            (puts & (vol_df["strike"] < S0))
        )
        boundary = vol_df[
            (vol_df["strike"] >= 0.9 * S0) &
            (vol_df["strike"] <= 1.1 * S0)
        ]
        avg_boundary_vol_newton = boundary["iv"].mean()
        avg_itm_vol_newton = vol_df.loc[itm_newton,"iv"].mean()
        avg_otm_vol_newton = vol_df.loc[otm_newton,"iv"].mean()
        results[symbol] = {
            "Newton ATM Implied Vol": atm_vol_newton,
            "Newton ITM Average Implied Vol": avg_itm_vol_newton,
            "Newton OTM Average Implied Vol": avg_otm_vol_newton,
        }
    return results

implied_vol_newton_results = implied_vol_metrics_newton(DATA1,␣
↪DATA1_underlying_df, r)
newton_metric_results = pd.DataFrame(implied_vol_newton_results).T
print(tabulate(newton_metric_results, headers="keys", tablefmt="fancy_grid"))
```

```
          Newton ATM Implied Vol     Newton ITM Average Implied Vol     Newton
OTM Average Implied Vol
```

```
 TSLA                    0.436779                          0.418713
0.436779


 SPY                     0.133398                          0.115684
0.12112
```

[11]:
```python
start = time.perf_counter()
bisect_results = implied_vol_metrics(DATA1, DATA1_underlying_df, r)
bisect_time = time.perf_counter() - start

start = time.perf_counter()
newton_results = implied_vol_metrics_newton(DATA1, DATA1_underlying_df, r)
newton_time = time.perf_counter() - start

print("Bisection Time:", bisect_time)
print("Newton Time:", newton_time)
```

```
Bisection Time: 8.539054700173438
Newton Time: 7.700994300656021
```

When comparing the two models, the Newton computes the values in a shorter time than the bisectional model in this example the newton model is only a tad bit faster than the bisectional model. The model is faster because it uses fewer interations.

[13]:
```python
#8 - Implied Volatility Table
pd.set_option('display.max_columns', None)
all_results = []
for symbol in ["TSLA", "SPY"]:
    S0 = DATA1_underlying_df.loc[
        DATA1_underlying_df["symbol"] == symbol,
        "underlying_price"
    ].values[0]
    df = DATA1[DATA1["symbol"] == symbol].copy()
    df = df[
        (df["volume"] > 0) &
        (df["bid"] > 0) &
        (df["ask"] > 0)
    ]
    df["market_price"] = (df["bid"] + df["ask"]) / 2
    iv_list = []
    for _, row in df.iterrows():
        K = row["strike"]
        T = row["ttm"]
        option_type = row["type"]
        market_price = row["market_price"]
        def f(sigma):
            return black_scholes_function(
                S0, K, r, sigma, T, option_type
            ) - market_price

        def f_newton(sigma):
            return vega(S0, K, T, r, sigma)
        iv = newton_method(f, f_newton, x0=0.2)
        iv_list.append(iv)
    df["iv"] = iv_list
    df["symbol"] = symbol
    df = df.dropna(subset=["iv"])
    atm_table = (df["strike"] - S0).abs().idxmin()
    atm_vol_table = df.loc[atm_table, "iv"]
    calls = df["type"] == "call"
    puts = df["type"] == "put"
    itm_table = (
        (calls & (df["strike"] < S0)) |
        (puts & (df["strike"] > S0))
    )
    otm_table = (
        (calls & (df["strike"] > S0)) |
        (puts & (df["strike"] < S0))
    )
    boundary = df[
        (df["strike"] >= 0.9 * S0) &
        (df["strike"] <= 1.1 * S0)
    ]
    avg_boundary_vol_table = boundary["iv"].mean()
    avg_itm_vol_table = df.loc[itm_table,"iv"].mean()
    avg_otm_vol_table = df.loc[otm_table,"iv"].mean()
    df["ATM Vol"] = atm_vol_table
    df["Average Implied Vol"] = avg_boundary_vol_table
    df["ITM AVG Vol"] = avg_itm_vol_table
    df["OTM AVG Vol"] = avg_otm_vol_table
    all_results.append(
        df[["symbol", "ttm", "type", "strike", "iv", "bid", "ask", "volume",
            "ATM Vol","Average Implied Vol", "ITM AVG Vol", "OTM AVG Vol" ]]
    )
implied_volatility_table = pd.concat(all_results, ignore_index=True)

print(implied_volatility_table)
```

```
   symbol       ttm  type  strike        iv    bid    ask  volume  \
0    TSLA  0.022890  call   395.0  0.505059  33.55  34.50    37.0
1    TSLA  0.022890  call   400.0  0.500816  29.65  30.15   681.0
2    TSLA  0.022890  call   405.0  0.477889  25.40  25.85   425.0
```

```
3     TSLA  0.022890  call   410.0  0.462279   21.60   21.75   546.0
4     TSLA  0.022890  call   412.5  0.451757   19.65   19.80   906.0
..     …        …    …     …        …       …      …      …
912    SPY  0.176315   put   770.0  0.212567   75.66   78.23     1.0
913    SPY  0.176315   put   775.0  0.222628   80.66   83.27     1.0
914    SPY  0.176315   put   780.0  0.232155   85.66   88.26     1.0
915    SPY  0.176315   put   800.0  0.268169  105.55  108.24     6.0
916    SPY  0.176315   put   805.0  0.277660  110.58  113.27     2.0

      ATM Vol  Average Implied Vol  ITM AVG Vol  OTM AVG Vol
0    0.421985             0.424679     0.432197     0.437857
1    0.421985             0.424679     0.432197     0.437857
2    0.421985             0.424679     0.432197     0.437857
3    0.421985             0.424679     0.432197     0.437857
4    0.421985             0.424679     0.432197     0.437857
..        …                    …          …            …
912  0.148551             0.175980     0.202455     0.182663
913  0.148551             0.175980     0.202455     0.182663
914  0.148551             0.175980     0.202455     0.182663
915  0.148551             0.175980     0.202455     0.182663
916  0.148551             0.175980     0.202455     0.182663

[917 rows x 12 columns]
```

For TSLA, the at-the-money implied volatility is 0.42198, and the average implied volatility is 0.424679, which is relatively close to the ATM value. For SPY, the ATM implied volatility is 0.148551, while the average implied volatility is 0.175980, which is higher than the ATM value. As for the ^VIX, the value at the time the data was collected was 17.47. This compares well to the average SPY implied volatility, which makes sense because the VIX represents the expected volatility of the S&P 500 over the next 30 days, so it should be close to SPY's expected volatility. TSLA's implied volatilities are significantly higher than both VIX and SPY because TSLA is a single company with greater volatility caused by public opinion and its financial performance. In contrast, SPY is a diversified portfolio of many stocks, which reduces overall volatility. As for the maturities, the average volatility for both TSLA and SPY remained the same. This may be due to a calculation issue within the function used. For the options are in the money, the volatility for TSLA is 0.432197 and SPY is 0.202455. This shows the the options in the money has greater volatilities which is the same for the out of money volatilities which were 0.437857 and 0.182663.

[14]:
```python
call_put_parity = []
for _, row in implied_volatility_table.iterrows():
    symbol = row["symbol"]
    K = row["strike"]
    T = row["ttm"]
    option_type = row["type"]
    iv = row["iv"]
    S0 = DATA1_underlying_df.loc[
        DATA1_underlying_df["symbol"] == symbol,
        "underlying_price"
    ].values[0]
    price = black_scholes_function(S0, K, r, iv, T, option_type)
    discount_strike = K * np.exp(-r*T)
    if option_type == "call":
        parity_price = price - S0 + discount_strike
        opposite_type = "put"
    else:
        parity_price = price + S0 - discount_strike
        opposite_type = "call"
    opposite = DATA1[
        (DATA1["symbol"] == symbol) &
        (DATA1["strike"] == K) &
        (DATA1["ttm"] == T) &
        (DATA1["type"] == opposite_type)
    ]
    if not opposite.empty:
        bid = opposite["bid"].values[0]
        ask = opposite["ask"].values[0]
        within_spread = (parity_price >= bid) and (parity_price <= ask)
    else:
        bid = np.nan
        ask = np.nan
        within_spread = False
    call_put_parity.append({"Symbol": symbol,"TTM": T,"strike": K,"Type":␣
↪option_type, "Parity_Price": parity_price, "opposite_bid": bid,␣
↪"opposite_ask": ask, "within_bid_ask": within_spread})
call_put_parity_table = pd.DataFrame(call_put_parity)
print(call_put_parity_table)
```

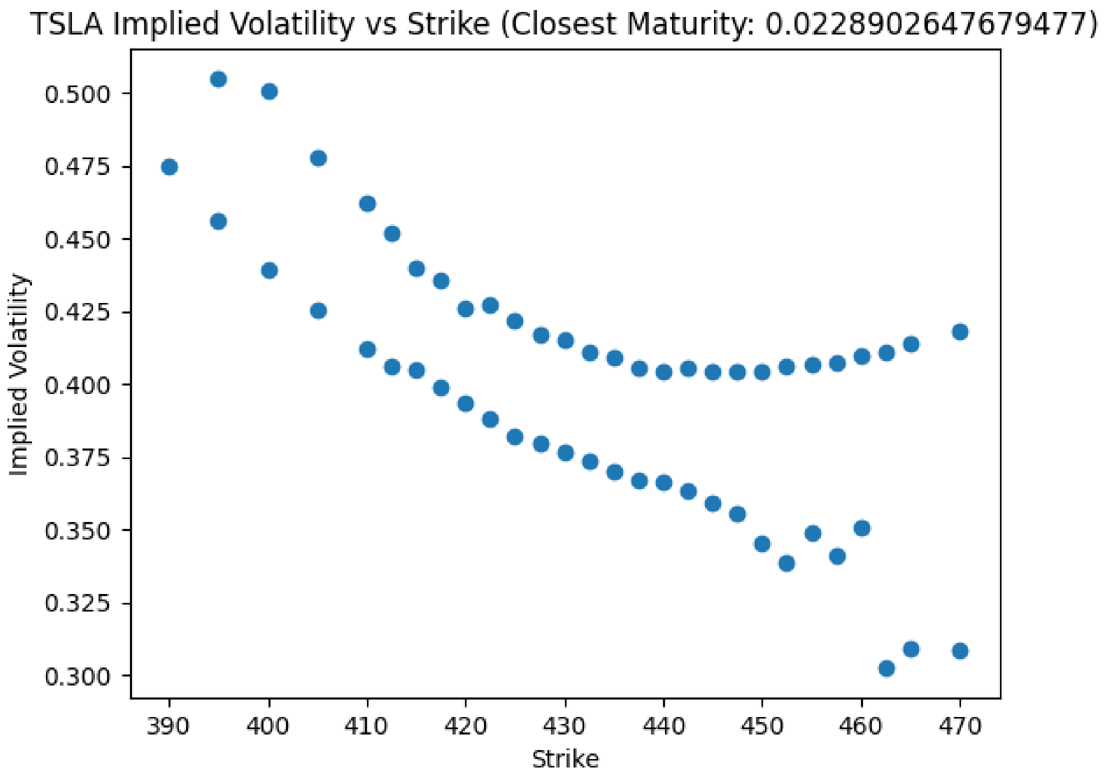
```
    Symbol       TTM  strike  Type  Parity_Price  opposite_bid  opposite_ask  \
0     TSLA  0.022890   395.0  call      2.596015          1.89          1.92
1     TSLA  0.022890   400.0  call      3.466850          2.43          2.46
2     TSLA  0.022890   405.0  call      4.187686          3.15          3.25
3     TSLA  0.022890   410.0  call      5.233522          4.15          4.20
4     TSLA  0.022890   412.5  call      5.781440          4.75          4.80
..     ...       ...     ...   ...           ...           ...           ...
912    SPY  0.176315   770.0   put      4.300923          0.23          0.24
913    SPY  0.176315   775.0   put      4.352909          0.18          0.19
914    SPY  0.176315   780.0   put      4.379896          0.14          0.15
915    SPY  0.176315   800.0   put      4.442842          0.07          0.08
916    SPY  0.176315   805.0   put      4.504829          0.06          0.07

     within_bid_ask
0             False
1             False
2             False
```

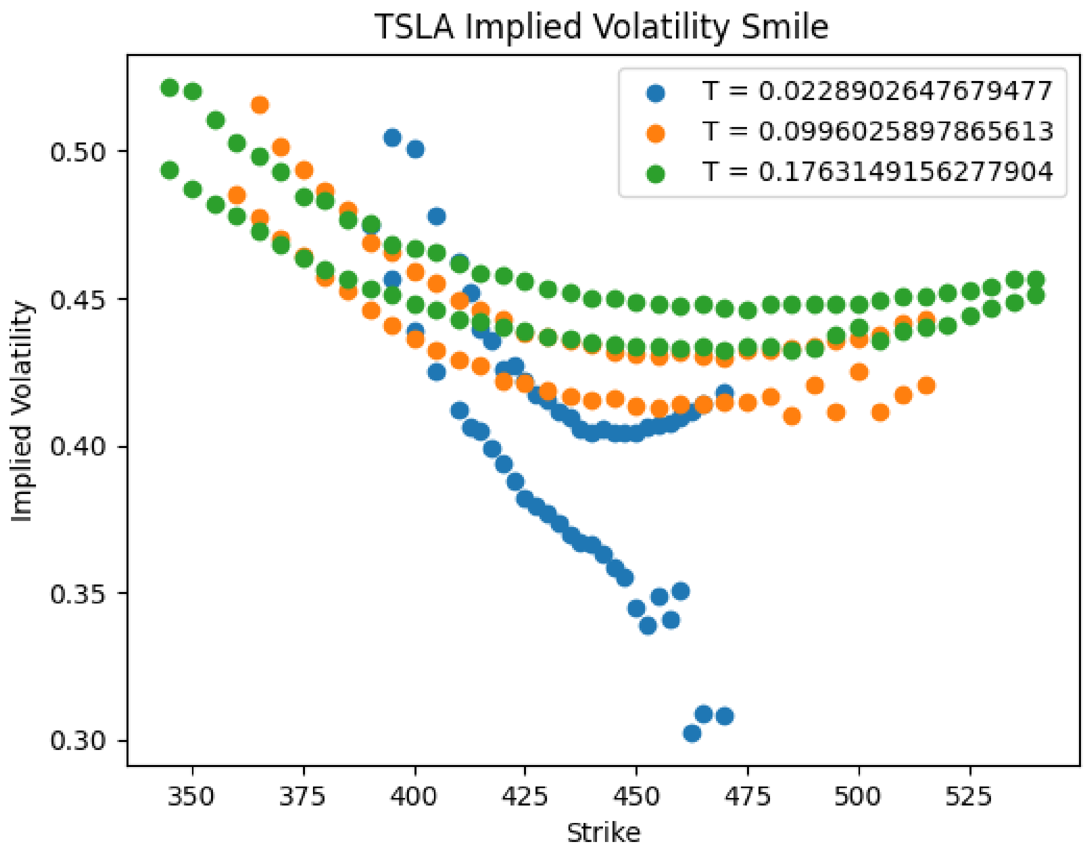
```
3              False
4              False
..               …
912            False
913            False
914            False
915            False
916            False

[917 rows x 8 columns]
```

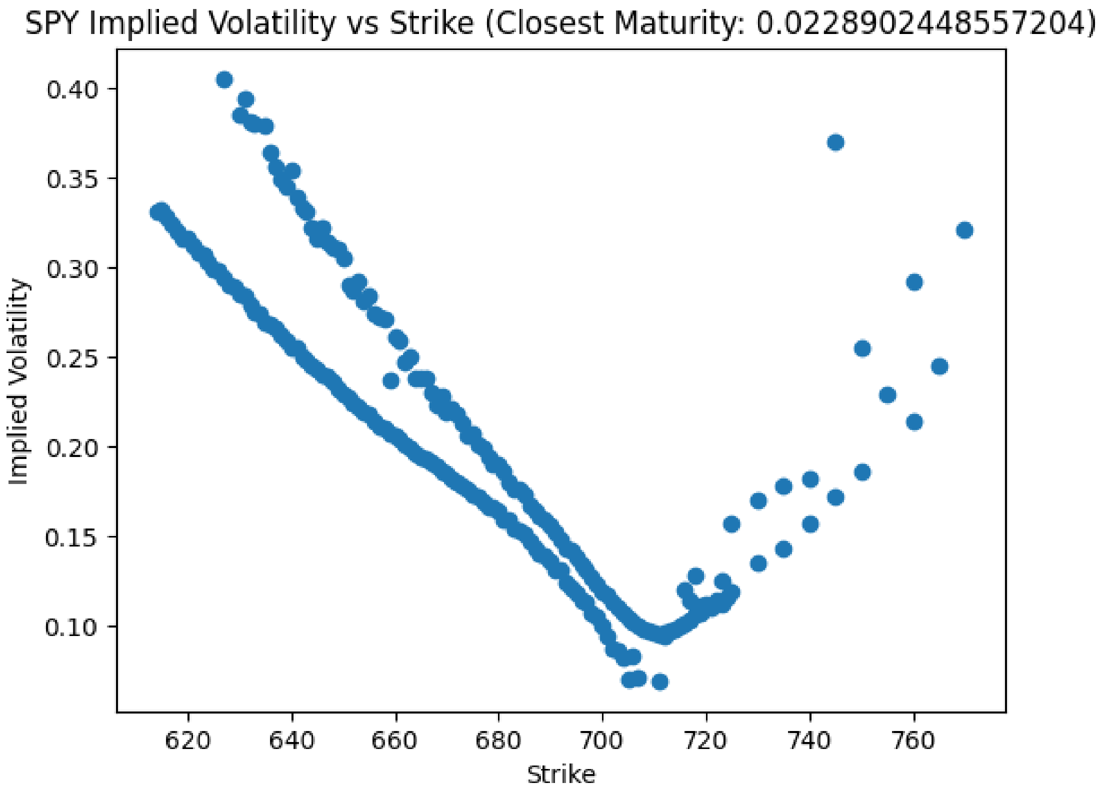
As for the call-put parity analysis, the calculated parity price is relatively close to the ask and bid values with the values being slightly greater than the bid and ask prices. As an example, TSLA put at the maturity of 0.099603 with the strike of 360.0 had a parity price of 71.302835 and the bid was 72.25 and the ask was 72.75. In this case, the parity price only had a difference of around 0.90. This pattern can be seen throughout the table.

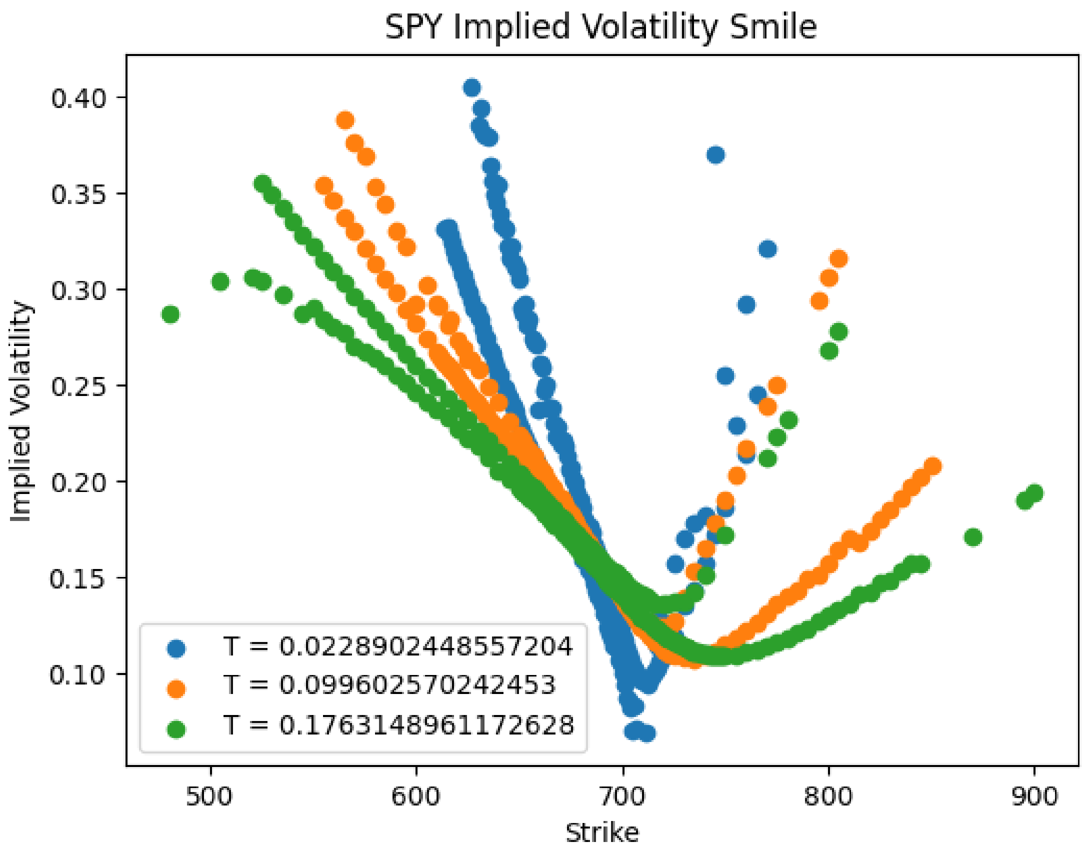
[15]:
```python
#10 - Graph of Implied Volatilities for TSLA
symbol = "TSLA"
df_symbol = implied_volatility_table[implied_volatility_table["symbol"] ==␣
 ↪symbol]
closest_ttm = df_symbol["ttm"].min()
df_closest = df_symbol[df_symbol["ttm"] == closest_ttm]
plt.figure()
plt.scatter(df_closest["strike"], df_closest["iv"])
plt.xlabel("Strike")
plt.ylabel("Implied Volatility")
plt.title(f"{symbol} Implied Volatility vs Strike (Closest Maturity:␣
 ↪{closest_ttm})")
plt.show()

plt.figure()
for maturity in sorted(df_symbol["ttm"].unique()):
    df_maturity = df_symbol[df_symbol["ttm"] == maturity]
    plt.scatter(
        df_maturity["strike"],
        df_maturity["iv"],
        label=f"T = {maturity}"
    )

plt.xlabel("Strike")
plt.ylabel("Implied Volatility")
plt.title(f"{symbol} Implied Volatility Smile")
plt.legend()
plt.show()

#10 - Graph of Implied Volatilities for SPY
```

```
symbol = "SPY"
df_symbol = implied_volatility_table[implied_volatility_table["symbol"] ==␣
↪symbol]
closest_ttm = df_symbol["ttm"].min()
df_closest = df_symbol[df_symbol["ttm"] == closest_ttm]
plt.figure()
plt.scatter(df_closest["strike"], df_closest["iv"])
plt.xlabel("Strike")
plt.ylabel("Implied Volatility")
plt.title(f"{symbol} Implied Volatility vs Strike (Closest Maturity:␣
↪{closest_ttm})")
plt.show()

plt.figure()
for maturity in sorted(df_symbol["ttm"].unique()):
    df_maturity = df_symbol[df_symbol["ttm"] == maturity]
    plt.scatter(
        df_maturity["strike"],
        df_maturity["iv"],
        label=f"T = {maturity}"
    )

plt.xlabel("Strike")
plt.ylabel("Implied Volatility")
plt.title(f"{symbol} Implied Volatility Smile")
plt.legend()
plt.show()
```


TSLA Implied Volatility vs Strike (Closest Maturity: 0.0228902647679477)
Implied Volatility
Strike
0.300
0.325
0.350
0.375
0.400
0.425
0.450
0.475
0.500
390
400
410
420
430
440
450
460
470


TSLA Implied Volatility Smile
T = 0.0228902647679477
T = 0.0996025897865613
T = 0.1763149156277904
Implied Volatility
0.30
0.35
0.40
0.45
0.50
Strike
350
375
400
425
450
475
500
525


SPY Implied Volatility vs Strike (Closest Maturity: 0.0228902448557204)
Implied Volatility
Strike
0.40
0.35
0.30
0.25
0.20
0.15
0.10
620
640
660
680
700
720
740
760



With the graphs, a similar trend is observed which is the volatility smile. The volatility smile differs for the shortest time to maturity and that may be due to closest in time. Even though the short maturity has abnormalies in the graphs, the general pattern is the smile.

[16]:
```
#11 - calculate the derivatives using BS and approx
#Black scholes method
def black_scholes_greeks(S, K, r, sigma, T):
    d1 = (np.log(S/K) + (r + 0.5*sigma**2)*T) / (sigma*np.sqrt(T))
    delta = norm.cdf(d1)
    vega = S * norm.pdf(d1) * np.sqrt(T)
    gamma = norm.pdf(d1) / (S * sigma * np.sqrt(T))
    return delta, vega, gamma

#Approximation method
def numerical_greeks_call(S, K, r, sigma, T, h=100):
    C = black_scholes_function(S, K, r, sigma, T, "call")
    C_up = black_scholes_function(S+h, K, r, sigma, T, "call")
    C_down = black_scholes_function(S-h, K, r, sigma, T, "call")
    delta = (C_up - C_down) / (2*h)
    gamma = (C_up - 2*C + C_down) / (h**2)
    C_vol_up = black_scholes_function(S, K, r, sigma+h, T, "call")
    C_vol_down = black_scholes_function(S, K, r, sigma-h, T, "call")
    vega = (C_vol_up - C_vol_down) / (2*h)
    return delta, vega, gamma

greek_results = []
for _, row in implied_volatility_table.iterrows():
    if row["type"] != "call":
        continue
    S = DATA1_underlying_df.loc[
        DATA1_underlying_df["symbol"] == row["symbol"],
        "underlying_price"
    ].values[0]
    K = row["strike"]
    T = row["ttm"]
    sigma = row["iv"]
    delta, vega, gamma = black_scholes_greeks(S, K, r, sigma, T)
    delta_app, vega_app, gamma_app = numerical_greeks_call(S, K, r, sigma, T)
    greek_results.append({
        "symbol": row["symbol"],"Strike": K,"TTM": T,"Delta": delta,"Delta␣
↪Approx": delta_app,"Vega": vega,"Vega Approx": vega_app,"Gamma":␣
↪gamma,"Gamma Approx": gamma_app
    })
greeks_table = pd.DataFrame(greek_results)
print(greeks_table)
```

```
    symbol  Strike       TTM     Delta  Delta Approx       Vega  Vega Approx  \
0     TSLA   395.0  0.022890  0.851047      0.656872  14.960909     4.103855
1     TSLA   400.0  0.022890  0.811409      0.632020  17.414129     4.128834
2     TSLA   405.0  0.022890  0.773403      0.607140  19.411934     4.153813
3     TSLA   410.0  0.022890  0.724944      0.582196  21.512969     4.178793
4     TSLA   412.5  0.022890  0.698802      0.569715  22.455036     4.191282
..     …       …       …        …         …          …          …
460    SPY   840.0  0.176315  0.002610      0.033156   2.346352     7.635281
461    SPY   845.0  0.176315  0.001917      0.028121   1.773847     7.660121
462    SPY   870.0  0.176315  0.001104      0.016164   1.072416     7.784322
463    SPY   895.0  0.176315  0.000999      0.011185   0.978480     7.908522
464    SPY   900.0  0.176315  0.000981      0.010465   0.962171     7.933362

        Gamma  Gamma Approx
0    0.007128      0.006343
1    0.008367      0.006666
2    0.009774      0.007020
3    0.011198      0.007310
4    0.011960      0.007450
..        …           …
460  0.000176      0.000656
```

```
461  0.000134      0.000557
462  0.000074      0.000320
463  0.000061      0.000221
464  0.000059      0.000206

[465 rows x 9 columns]
```

For the analysis of the greeks, the black scholes method and the numerical method result is different values. The vega had the greatest difference in values with the numerical values either being significantly higher or lower than the black scholes method. As for the delta and gamma, the values were relatively close with the numerical values being a little bit less than the black scholes method.

[17]:
```python
#12
merged_df = implied_volatility_table.merge(
    implied_volatility_table[["symbol", "strike", "ttm", "type", "iv"]],
    on=["symbol", "strike", "ttm", "type"], how="inner" )
print(merged_df.columns)

black_scholes_prices = []
for _, row in merged_df.iterrows():
    S2 = underlying_data2_df.loc[
        underlying_data2_df["symbol"] == row["symbol"],
        "underlying_price" ].values[0]
    K = row["strike"]
    T = row["ttm"]
    sigma = row["iv_x"]
    option_type = row["type"]
    price_data2 = black_scholes_function(S2, K, r2, sigma, T, option_type)
    black_scholes_prices.append(price_data2)
merged_df["Black_Scholes_DATA2_Price"] = black_scholes_prices

print(merged_df[["symbol","strike","ttm","type","iv_x","Black_Scholes_DATA2_Price"]])
```

```
Index(['symbol', 'ttm', 'type', 'strike', 'iv_x', 'bid', 'ask', 'volume',
       'ATM Vol', 'Average Implied Vol', 'ITM AVG Vol', 'OTM AVG Vol', 'iv_y'],
      dtype='object')
    symbol  strike       ttm  type      iv_x  Black_Scholes_DATA2_Price
0     TSLA   395.0  0.022890  call  0.505059                  25.677336
1     TSLA   400.0  0.022890  call  0.500816                  22.024794
2     TSLA   405.0  0.022890  call  0.477889                  18.215596
3     TSLA   410.0  0.022890  call  0.462279                  14.836039
4     TSLA   412.5  0.022890  call  0.451757                  13.192890
..     …       …         …    …      …                          …
912    SPY   770.0  0.176315   put  0.212567                  87.066465
913    SPY   775.0  0.176315   put  0.222628                  92.143840
914    SPY   780.0  0.176315   put  0.232155                  97.198324
915    SPY   800.0  0.176315   put  0.268169                 117.357151
916    SPY   805.0  0.176315   put  0.277660                 122.430944
```

```
[917 rows x 6 columns]
```

0.3 Part 3

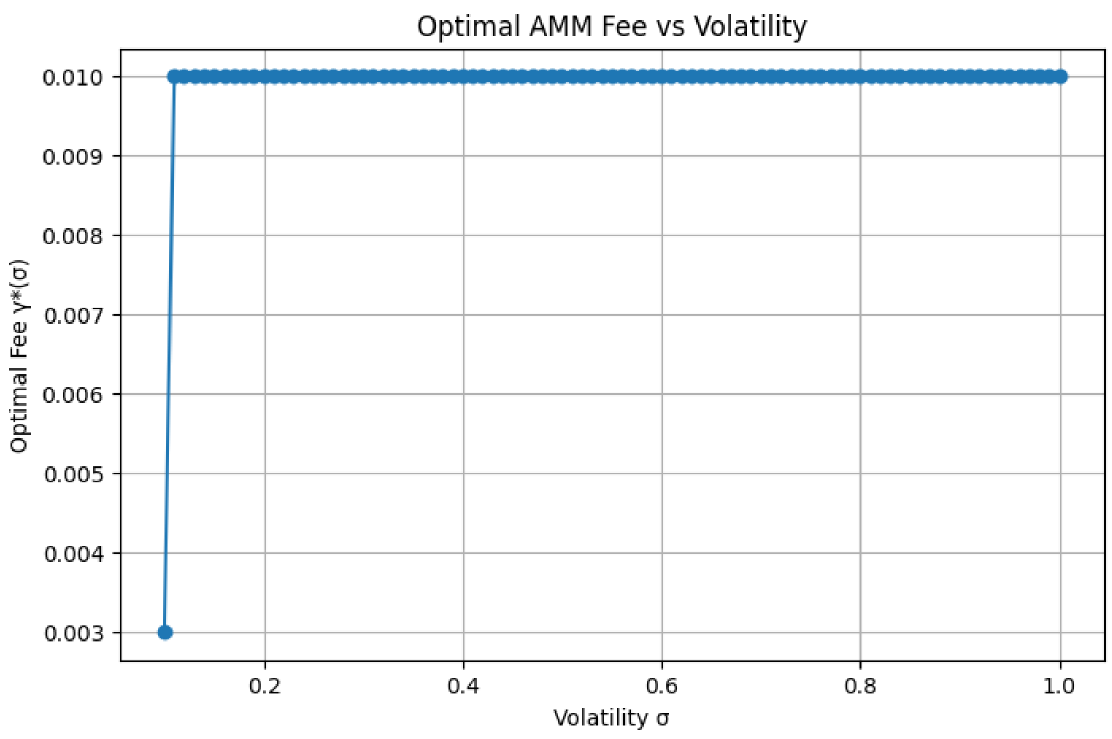
[15]:
```
#A - The goal is to create a formula that one step revenue will be R = if St >␣
↪Pt/1-y = gamma * change in y plus if St1 <Pt(1-gamma) = gamma * change in x␣
↪* St1
# case 1: change in x = xt - xt1, change in y = yt1/(yt + (1-gamma) ), xt1 =␣
↪sqrt(k/(St1(1-gamma)) , yt1 = St1 * (1+ gamma) * xt1
# case 2: k = change in x = xt1/ (xt + (1 - gamma)), change in y = yt - yt1,␣
↪xt1 = sqrt((k(1-y))/St1) , yt1 = (St1/(1-gamma)) * (xt + (1 - gamma) * dx)
def swap_amounts_function(St1, xt, yt, gamma):
    k = xt * yt
    Pt = yt / xt
    upper_band = Pt / (1 - gamma)
    lower_band = Pt * (1 - gamma)
    if lower_band <= St1 <= upper_band :
        return 0,0,0
    if St1 > upper_band:
        xt1 = np.sqrt(k / (St1 * (1 - gamma)))
        dx = xt - xt1
        yt1 = St1 * (1+ gamma) * xt1
        dy = (yt1 - yt)/(1 - gamma)
        if dx > 0 and dy > 0 :
            revenue = gamma * dy
            return dx, dy, revenue

    if St1 < lower_band:
        xt1 = np.sqrt((k * (1 - gamma)) / St1)
        dx = (xt1 - xt)/(1 - gamma)
        yt1 = (St1/(1-gamma)) * xt1
        dy = yt - yt1
        if dx > 0 and dy > 0 :
            revenue = gamma * dx * St1
            return dx, dy, revenue
```

[16]:
```
#B Vol = 0.2 & Fee rate (lambda) = 0.003 (30 basis points) & xt = y
vol_amm = 0.2
fee_rate = 0.003
xt = 1000
yt = 1000
St = 1
delta_t = 1/365
k = xt * yt
Pt = yt / xt
def trapezoidal_rule(f, a, b, n):
    h = (b - a) / (n - 1)
    x = np.linspace(a, b, n)
    fx = f(x)
    return (h / 2) * (fx[0] + 2 * np.sum(fx[1:n-1]) + fx[n-1])
def expected_revenue_function(sigma, gamma, xt, yt, St, delta_t, s_min=0.01,␣
 ↪s_max=2.0, n=1000):
    mu = np.log(St) - 0.5 * sigma**2 * delta_t
    var = sigma**2 * delta_t

    def lognormal_density(s):
        return (1/(s * np.sqrt(2*np.pi*var))) * \
               np.exp(-(np.log(s) - mu)**2 / (2*var))
    def revenue_function(s_array):
        revenues = np.zeros_like(s_array)
        for i, s in enumerate(s_array):
            _, _, revenue = swap_amounts_function(s, xt, yt, gamma)
            revenues[i] = revenue
        return revenues * lognormal_density(s_array)
    return trapezoidal_rule(revenue_function, s_min, s_max, n)

vol_amm = 0.2
fee_rate = 0.003
xt = 1000
yt = 1000
St = 1
delta_t = 1/365
k = xt * yt
Pt = yt / xt

expected_revenue = expected_revenue_function(vol_amm,fee_rate,xt,yt,St,delta_t)
print(expected_revenue)
```

```
0.015536166262306789
```

[53]:
```
vols = [0.2, 0.6, 1.0]
gammas = [0.001, 0.003, 0.01]

results = {}
for sigma in vols:
    results[sigma] = {}
    for gamma in gammas:
        ER = expected_revenue_function(sigma, gamma, xt, yt, St, delta_t,␣
 ↪s_min=0.01, s_max=1.0, n=10)
        results[sigma][gamma] = ER

optimal_gamma = {}
for sigma in vols:
    gamma_star = max(results[sigma], key=lambda g: results[sigma][g])
    optimal_gamma[sigma] = gamma_star

print("      =0.001         =0.003         =0.01          *( )")
for sigma in vols:
    row = f"{sigma:<5} "
    for gamma in gammas:
        row += f"{results[sigma][gamma]:<14.6e} "
    row += f"   {optimal_gamma[sigma]}"
    print(row)

vol_table = np.arange(0.1, 1.01, 0.01)
gamma_star_table = []

for sigma in vol_table:
    ER_values = {}
    for gamma in gammas:
        ER_values[gamma] = expected_revenue_function(sigma, gamma, xt, yt, St,␣
 ↪delta_t, s_min=0.01, s_max=2.0, n=10000)
    gamma_star = max(ER_values, key=lambda g: ER_values[g])
    gamma_star_table.append(gamma_star)

plt.figure(figsize=(8, 5))
plt.plot(vol_table, gamma_star_table, marker='o', linestyle='-')
plt.xlabel("Volatility  ")
plt.ylabel("Optimal Fee  *( )")
plt.title("Optimal AMM Fee vs Volatility")
plt.grid(True)
plt.show()
```

```
      =0.001         =0.003         =0.01          *( )
0.2   3.264621e-28   9.638411e-28   3.029329e-27      0.01
0.6   9.024451e-05   2.664363e-04   8.374028e-04      0.01
1.0   4.437457e-03   1.310112e-02   4.117701e-02      0.01
```



Analysis: In this line plot, the first point starts at 0.003 and then immediately jumps to 0.01 and stays at 0.01 for all the volatility values. This is because the optimal fees are the greatest at the y = 0.01. This may be due to the delta_t value being so small (1 day). Therefore, the pattern is a straight line.

0.4 Part 4 - Bonus

Based on the analytically solved integrals, the f1 integral equals to 9/4 or 2.25 and the f2 integral equals to 32.79 or -exp(3) + 1 + E*(-1 + exp(3)). When doing the integral with the trapezoid rule approximation, I used [(5,5), (10,10), (50,50), (100,100)] for my x and y values. When calculating the integral values based on these delta values, the f1 intergral was the same value everytime (2.25) which the exact value as the analytically solved intergral. However, the f2 intergral changed in value with each increase in delta value - as the delta values increased, the difference between the analytical value and the numerical value decreased. Therefore, the increased number of points will increase the accuracy of the integral to its accutal value.

[27]:
```python
import sympy as sp
x,y = sp.symbols('x y')
f1 = x * y
f2 = sp.exp(x + y)

#1
f1_integral = sp.integrate(sp.integrate(f1, (y, 0, 3)), (x, 0, 1))
```

```
f2_integral = sp.integrate(sp.integrate(f2, (y, 0, 3)), (x, 0, 1))
f2_integral = float(f2_integral)
print(f1_integral)
print(f2_integral)
```

```
9/4
32.79433128149753
```

[25]:
```
#2
def discretization(n, m):
    dx = 1 / (n + 1)
    dy = 3 / (m + 1)
    X = np.array([0 + i*dx for i in range(n+2)])
    Y = np.array([0 + j*dy for j in range(m+2)])
    return X, Y

# Ultizied the Pseudo code from "The Heston Model and Its Extensions in Matlab␣
 ↪and C"
def trap_method (f, X, Y):
    Nx = len(X)
    Ny = len(Y)
    Int = np.zeros((Nx, Ny))
    for y in range(1, Ny):
        a = Y[y-1]
        b = Y[y]
        for x in range(1, Nx):
            c = X[x-1]
            d = X[x]
            term1 = f(a, c) + f(a, d) + f(b, c) + f(b, d)
            term2 = (f((d+c)/2, b) + f((d+c)/2, a) +f(d, (b+a)/2) + f(c, (a+b)/
 ↪2))
            term3 = f((a+b)/2, (c+d)/2)
            Int[x, y] = (b - a) * (d - c) / 16 * (term1 + 2*term2 + 4*term3)
    return np.sum(Int)

delta_values = [(5,5), (10,10), (50,50), (100,100)]
for (n,m) in delta_values:
    X, Y = discretization(n,m)
    f1_trap = lambda x, y: x*y
    f2_trap = lambda x, y: np.exp(x + y)
    trap_intergral_f1 = trap_method(f1_trap, X, Y)
    trap_intergral_f2 = trap_method(f2_trap, X, Y)
    error_f1 = abs(trap_intergral_f1 - f1_integral)
    error_f2 = abs(trap_intergral_f2 - f2_integral)

    print(f"(n={n}, m={m}):")
    print(trap_intergral_f1, error_f1)
```

```python
    print(trap_intergral_f2, error_f2)
    print()
```

```
(n=5, m=5):
2.25 0
32.98403216931599 0.18970088781846073

(n=10, m=10):
2.25 0
32.8507881603624 0.056456878864871385

(n=50, m=50):
2.25 0
32.796958006547804 0.0026267250502769457

(n=100, m=100):
2.25 0
32.79500103368114 0.0006697521836116493
```