Part 1: Data Gathering

Problem 1

Download TSLA historical prices and current option chain from Yahoo Fiance

In [2]:
```
# %reset -f
import yfinance as yf
import pandas as pd
import numpy as np
from datetime import datetime
```

In [3]:
```
# define the undelying asset and time period
index = "TSLA"
time_start = "2025-01-01"
time_end = datetime.today().strftime('%Y-%m-%d')
tsla = yf.Ticker(index)
und_data = tsla.history(start=time_start,end=time_end)

# transform UTC time zone into Easten time zone
und_data.index = und_data.index.tz_convert('US/Eastern')
# delete time zone information to save data
und_data.index = und_data.index.tz_localize(None)
# Transfrom the index w.r.t. date into column
und_data = und_data.reset_index()
# print part of data
print(und_data.head())
```

```
        Date        Open        High         Low       Close     Volume  \
0 2025-01-02  390.100006  392.730011  373.040009  379.279999  109710700
1 2025-01-03  381.480011  411.880005  379.450012  410.440002   95423300
2 2025-01-06  423.200012  426.429993  401.700012  411.049988   85516500
3 2025-01-07  405.829987  414.329987  390.000000  394.359985   75699500
4 2025-01-08  392.950012  402.500000  387.399994  394.940002   73038800

   Dividends  Stock Splits
0        0.0           0.0
1        0.0           0.0
2        0.0           0.0
3        0.0           0.0
4        0.0           0.0
```

In [4]:

```python
# Extract all maturity. Note: for each loop, we can only get one maturity
maturity = tsla.options
option_data = []
for i in maturity:
    temp = tsla.option_chain(i)
    # Get call option
    Call = temp.calls.copy()
    # Add more label
    Call['maturity'] = i
    Call['type'] = 'call'
    # Get put option
    Put = temp.puts.copy()
    # Add more label
    Put['maturity'] = i
    Put['type'] = 'put'
    # Consolidate date
    option_data.append(Call)
    option_data.append(Put)
# consolidate data
option_data = pd.concat(option_data)

# transform UTC time zone into Easten time zone
option_data['lastTradeDate'] = option_data['lastTradeDate'].dt.tz_convert('US/Eastern')
# delete time zone information to save data
option_data['lastTradeDate'] = option_data['lastTradeDate'].dt.tz_localize(None)
# print part of data
print(option_data.head())
```

```
# save underlying asset and option data in excel
filename = f'C:/Users/18509/Desktop/Course/FE 621 A/Assignment 1/HW1_Problem1.1_{time_end}.xlsx'
with pd.ExcelWriter(filename) as writer:
    und_data.to_excel(writer,sheet_name='underlying',index=False)
    option_data.to_excel(writer,sheet_name='option',index=False)
```

```
       contractSymbol       lastTradeDate  strike  lastPrice    bid    ask  \
0  TSLA260218C00322500 2026-02-13 15:53:38   322.5      94.95  94.25  96.00
1  TSLA260218C00325000 2026-02-12 13:32:36   325.0      91.44  90.80  94.45
2  TSLA260218C00330000 2026-02-12 13:24:30   330.0      87.10  85.85  89.40
3  TSLA260218C00332500 2026-02-12 13:32:20   332.5      84.06  83.30  87.00
4  TSLA260218C00337500 2026-02-12 13:24:01   337.5      79.74  78.30  81.85

     change  percentChange  volume  openInterest  impliedVolatility  \
0  0.449997       0.476187     2.0             6           1.271488
1  0.000000       0.000000    20.0             3           1.238285
2  0.000000       0.000000    10.0             8           1.170903
3  0.000000       0.000000    10.0             1           1.158207
4  0.000000       0.000000    31.0            15           1.027349

   inTheMoney contractSize currency    maturity  type
0        True      REGULAR      USD  2026-02-18  call
1        True      REGULAR      USD  2026-02-18  call
2        True      REGULAR      USD  2026-02-18  call
3        True      REGULAR      USD  2026-02-18  call
4        True      REGULAR      USD  2026-02-18  call
```

Bouns 1

Download and combine multiple assets

For simplicity, I only select the close price of underlying assets

In [5]:
```
# %reset -f
import yfinance as yf
import pandas as pd
import numpy as np
from datetime import datetime
```

In [6]:
```
index_all = ["TSLA","AAPL"]
time_start = "2025-01-01"
time_end = datetime.today().strftime('%Y-%m-%d')
asset_data = pd.DataFrame()
# extract close price for each stock in a loop
for i in index_all:
    asset = yf.Ticker(i)
    temp = asset.history(start=time_start,end=time_end)
    asset_data[i] = temp['Close']

# transform UTC time zone into Easten time zone
asset_data.index = asset_data.index.tz_convert('US/Eastern')
# delete time zone information to save data
asset_data.index = asset_data.index.tz_localize(None)
# Transfrom the index w.r.t. date into column
asset_data = asset_data.reset_index()
# print part of data
print(asset_data.head())

filename = f'C:/Users/18509/Desktop/Course/FE 621 A/Assignment 1/HW1_Bouns1_{time_end}.xlsx'
# save underlying asset in excel
with pd.ExcelWriter(filename) as writer:
    asset_data.to_excel(writer,sheet_name='underlying',index=False)
```

```
        Date        TSLA        AAPL
0 2025-01-02  379.279999  242.525162
1 2025-01-03  410.440002  242.037811
2 2025-01-06  411.049988  243.668915
3 2025-01-07  394.359985  240.894073
4 2025-01-08  394.940002  241.381409
```

Problem 3

- SPY

The State Street SPDR S&P 500 ETF Trust (SPY), managed by State Street Global Advisors, is the oldest and largest American ETFs, which is used for tracking the S&P 500 Index. It is a highly liquid unit investment trust, and include major tech gaints, such as

NVDIA, Amazon, Apple, and Microsoft.

- VIX

The CBOE Volatility Index (VIX) is a real-time measure of the market's expectation of the 30-day volatility of the S&P 500 index (SPX). It is derived from SPX index options and is commonly referred to as the "fear index." Because of the leverage effect, there is typically a significant negative relationship between the VIX and the SPX.

- option symbol

(1) contractSymbol: It is the unique identifier for an option contract.

(2) strike: It is the agreed price at which the option can be exercised.

(3) maturity: It is the expiration date of the option contract.

(4) type: A call gives the holder the right to buy the underlying asset, and a put gives the holder the right to sell the underlying asset.

(5) bid: It is the price at which the market is willing to buy.

(6) ask: It is the price at which the market is willing to sell.

(7) inTheMoney: It defines whether exercising the option would currently be profitable.

(8) openInterest: It is the total number of outstanding option contracts.

(9) Volume: It is the traded quantity on a specific date.

Problem 2 & 4

- Declaration: For this assignment, I plan to show two consecutive days of option data for VIX, SPY, and TSLA (2026-02-12 to 2026-02-13) and base the analysis on Day 1 (DATA1). Unfortunately, I accidentally delete the 2026-02-12 VIX and SPY data during debugging, and I couldn't find a reliable source for historical option data for that date. So, the analysis in this report

uses only the 2026-02-13 data. I understand the assignment requirements and have done my best to present a complete analysis with the data available. If you need, I can provide the full dataset I had downloaded.

In this part, I first retrieve option maturities before the third Friday of the next three months and exclude options maturing today. Later, I download the high-frequency data for underlying assets, because for some illiquid option data, the quotes are not continous. Therefore, I need to align the underlying asset and option data at the identical timestaps.

In addition, I will implement problem 1.4 to collect the risk-free interest rate and compute maturity as the fraction of a year, which is used for the following option pricing. For the risk-free interest rate, I select 13-week Treasury yield from the Yahoo finance website. Note that these data are typically released with a 1–2 day delay. Moreover, I estimate the maturity at the minute level as follows:

$$T = T_d - 1 + \frac{t_{end} - t}{t_{end} - t_{start}}$$

where T_d means the number of trading days from today to the option maturity, t means the last trading time of a chosen option today. t_{start} and t_{end} represent the beginning and ending trading session within a day. For simplicity, I fix $t_{start} = 9:30$, and $t_{end} = 9:30$.

Data filtration criteria include:

- Obtain maturities that occur before the third Friday of the next three months.
- Exclude options whose lastTradeDate falls outside today's regular trading hours (9:30–16:00).
- Exclude options with zero bid and ask price.
- For each strike–maturity pair, if either the call or the put is absent after applying the above filters, both contracts are removed.

In [7]:
```
# %reset -f
import yfinance as yf
import pandas as pd
from pandas_datareader import data as web
import numpy as np
import openpyxl
import os
from datetime import datetime, timedelta
```

In [8]:
```
# ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
# Define the current date
# # today = datetime.today()
# today = pd.to_datetime('2026-02-12')
today = pd.to_datetime('2026-02-13')

# Define the underlying assets
# asset_all = ["TSLA"]
asset_all = ["TSLA","SPY","^VIX"]
# ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !

# Obtain the trading date on the third Friday of the next three months
date_max = today + pd.DateOffset(months=3)
date_max = date_max.replace(day=1) + pd.offsets.WeekOfMonth(week=2,weekday=4)
date_max = date_max.strftime('%Y-%m-%d')
today = today.strftime('%Y-%m-%d')
print(f"Maximum Maturity: {date_max}")
```

```
Maximum Maturity: 2026-05-15
```

In [ ]:
```
filename = f'C:/Users/18509/Desktop/Course/FE 621 A/Assignment 1/HW1_Problem1.2_{today}.xlsx'
if not os.path.exists(filename):
    pd.DataFrame().to_excel(filename, index=False)

# Collect option data for each underlying asset data in a loop:
for k in asset_all:
    temp = yf.Ticker(k)
    # All maturities
    maturity_all = temp.options
    # Transform it into series format for the following data filteration
    maturity_all = pd.Series(maturity_all)

    # Obtain the target maturity data and exclude the option maturing today
    maturity = maturity_all[(maturity_all <= date_max) & (maturity_all > today)]
    print(f"Target maturity of {k} option:{maturity.head()}")

    # Get options from the target maturities
    option_data = pd.DataFrame()
    for i in maturity:
        temp2 = temp.option_chain(i)
```

```
        Call = temp2.calls.copy()
        Call['maturity'] = i
        Call['Type'] = 'call'
        Put = temp2.puts.copy()
        Put['maturity'] = i
        Put['Type'] = 'put'
        option_data = pd.concat([option_data,Call,Put],ignore_index=True)
    # Timestamp Transformation
    option_data['lastTradeDate'] = option_data['lastTradeDate'].dt.tz_convert\
    ('US/Eastern')
    option_data['lastTradeDate'] = option_data['lastTradeDate'].dt.tz_localize\
    (None)

    # Obtain the high-frequency asset data at one-minute intervals
    end_date = (datetime.strptime(today, '%Y-%m-%d') + timedelta(days=1)).strftime('%Y-%m-%d')
    asset = temp.history(start=today, end=end_date, interval="1m")
    # transform UTC time zone into Easten time zone
    asset.index = asset.index.tz_convert('US/Eastern')
    # delete time zone information to save data
    asset.index = asset.index.tz_localize(None)
    # Transfrom the index w.r.t. date into column
    asset = asset.reset_index()

    # save underlying asset and option data in excel in each loop
    # export data into the same excel file but different sheets
    with pd.ExcelWriter(filename, mode='a', engine='openpyxl', if_sheet_exists='replace') as writer:
        asset.to_excel(writer,sheet_name=f'{k}_underlying',index=False)
        option_data.to_excel(writer,sheet_name=f'{k}_option',index=False)
```

```
Target maturity of TSLA option:0    2026-02-18
1    2026-02-20
2    2026-02-23
3    2026-02-25
4    2026-02-27
dtype: object
```

In [ ]:

```
# Define a function to filter option data for missing call or put prices at a certain strike price and maturity and with zero
def filter_option_data_miss(option_data):
    # Filter out data that have zero bid or ask price
    option_data = option_data[(option_data['bid'] > 0) & (option_data['ask'] > 0)]
```

```
    # Group the maturity and strike price (Maintain the original order)
    # grouped function is not used for storing the data, but for specifying the strike price and maturity
    data_group = option_data.groupby(['maturity', 'strike'], sort=False)
    # Filter out groups that do not have both call and put options
    # filter function will preseve the 'True' value and drop the 'False' value
    option_data_filtered = data_group.filter(lambda x: set(x['Type']) == {'call', 'put'})
    return option_data_filtered
```

In [ ]:

```
# Import Data path
filename = f'C:/Users/18509/Desktop/Course/FE 621 A/Assignment 1/HW1_Problem1.2_{today}.xlsx'

# Export Data path
# The data on 2026/2/12 is defined as DATA 1, and the data on 2026/2/13 is defined as DATA 2. The following analysis is for DA
if today == '2026-02-12':
    filename2 = f'C:/Users/18509/Desktop/Course/FE 621 A/Assignment 1/HW1_DATA1.xlsx'
elif today == '2026-02-13':
    filename2 = f'C:/Users/18509/Desktop/Course/FE 621 A/Assignment 1/HW1_DATA2.xlsx'
else:
    filename2 = f'C:/Users/18509/Desktop/Course/FE 621 A/Assignment 1/HW1_DATA.xlsx'

# Filter option data for each underlying asset data in a loop:
for k in asset_all:
    # Import data and filter data
    option_data = pd.read_excel(filename, sheet_name=f'{k}_option')
    asset = pd.read_excel(filename, sheet_name=f'{k}_underlying')

    # Exclude options with lastTradeDate outside today regular trading time (9:30-16:00)
    time_min_today = pd.to_datetime(f"{today} 09:30:00")
    time_max_today = pd.to_datetime(f"{today} 16:00:00")
    option_data = option_data[(option_data['lastTradeDate'] >= time_min_today)&\
        (option_data['lastTradeDate'] <= time_max_today)]
    # drop old index and generate new index
    option_data = option_data.reset_index(drop=True)

    # Compute the fractional maturity
    option_data['fmaturity'] = pd.to_datetime(option_data['maturity'])
    today_2 = pd.to_datetime(today)
    # The number of remaining trading dates
    # busday_count function doesn't consider the final day
    # lamda x is a temporary function, which is similar to @(x) f(x) in Matlab
```

```
option_data['fmaturity'] = option_data['fmaturity'].apply(\
    lambda x: np.busday_count(time_min_today.date(),x.date()))
# The fraction of remainning time today (assuem 250 trading days in a year)
option_data['fmaturity'] = option_data['fmaturity'] + (time_max_today - option_data['lastTradeDate'])/(\
    time_max_today - time_min_today)
option_data['fmaturity'] = option_data['fmaturity']/250

# Import the 13-week Treasury rate as the proxy for risk-free rate
tbill = yf.Ticker("^IRX")
rf = tbill.history(period="1d")
option_data['rf_rate'] = rf['Close'].values[0]/100

# Match the option data and close price underlying asset data
# Preserve the left the option data and then drop the timestamp w.r.t. asset
# It is possible that two types of data cannot exactly match at the second level, thus I use the fuzzy match
# Preserve the original order of option data, and then match the data, finally restore the original ordering
option_data = option_data.reset_index(drop=False).rename(columns = {'index':'original_index'})
option_data = option_data.sort_values('lastTradeDate')
option_data = pd.merge_asof(option_data,\
    asset[['Datetime','Close']],left_on = 'lastTradeDate',\
    right_on = 'Datetime',direction = 'backward').drop(columns='Datetime')
option_data = option_data.sort_values('original_index').drop(columns = 'original_index')
# Drop the unnecessary columns
option_data = option_data.drop(columns={'change','percentChange','inTheMoney','contractSize','currency'})
print(f'{k}_option_data.shape: {option_data.shape}')

# filter option data for missing call or put prices at a certain strike price and maturity and with zero bid or ask price
option_data = filter_option_data_miss(option_data)
option_data = option_data.reset_index(drop=True)
print(f'{k}_option_data_after_first_filtering.shape: {option_data.shape}')

# I define a new column for the type of option, namely, call->1 and put->0, for the following regression analysis
option_data['option_type'] = option_data['Type'].apply(lambda x: 1 if x == 'call' else 0)

# Compute the mid price for the following analysis
option_data['mid'] = (option_data['bid'] + option_data['ask'])/2

# Collect necessary data for the following analysis
option_data_backup = option_data.copy()
new_order = ['Close','strike','rf_rate','fmaturity','bid','ask','mid','option_type','maturity']
option_data_new = option_data[new_order]
```

```
    option_data_new.columns = ['S','K','r','T','bid','ask','mid','type','maturity']

    if not os.path.exists(filename2):
        pd.DataFrame().to_excel(filename2, index=False)

    print(option_data_new.head())
    with pd.ExcelWriter(filename2, mode='a', engine='openpyxl', if_sheet_exists='replace') as writer:
        option_data_new.to_excel(writer,sheet_name=f'{k}_option',index=False)
```

```
TSLA_option_data.shape: (1536, 14)
TSLA_option_data_after_first_filtering.shape: (964, 14)
            S      K        r         T     bid     ask      mid  type  \
0  416.819489  250.0  0.03598  0.005429  163.00  170.20  166.600     1
1  415.834991  270.0  0.03598  0.004692  143.00  150.15  146.575     1
2  416.269989  280.0  0.03598  0.004791  133.00  140.40  136.700     1
3  415.359985  285.0  0.03598  0.004064  128.00  135.40  131.700     1
4  418.433990  290.0  0.03598  0.005998  123.15  129.95  126.550     1

     maturity
0  2026-02-13
1  2026-02-13
2  2026-02-13
3  2026-02-13
4  2026-02-13
```

Part 2 Analysis of the data.

In this part, I use the mid-point between the bid and ask quotes as the proxy for the option price in the subsequent computations. It is straightforward to extend the analysis to use only bid or ask prices. I will implement this latter in Problem 9 in Part 2.

Problem 5

I check the data, and find that the selected underlying assets did not undergo stock splits or pay dividends during the sample period. For clarity, I define the current time as $t = 0$, so the remaining time to maturity is $T - t = T$.

- Call Option Price

$$C = S_0 N(d_1) - K e^{-rT} N(d_2)$$

- Put Option Price

$$P = K e^{-rT} N(-d_2) - S_0 N(-d_1)$$

where:

$$d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

I can also use the Put-Call Parity to transform the option price.

- Put-Call Parity

$$C - P = S_0 - Ke^{-rT}$$

In [ ]:
```
import yfinance as yf
import pandas as pd
from pandas_datareader import data as web
import numpy as np
import openpyxl
import os
from datetime import datetime, timedelta
import scipy as sp
from scipy.stats import norm
import time
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

In [ ]:
```
# Define the Black-Scholes formula for call and put options
def price_BS(S, K, T, r, sigma, option_type):
    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    # # Call option
    # if option_type == 1:
```

```
    #     price = S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
    # # Put option
    # else:
    #     price = K * np.exp(-r * T) * norm.cdf(-d2) - S * norm.cdf(-d1)

    # I use the Put-Call Parity to compute the price of call and put options, which is equivalent to the above formula
    price_temp = S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
    price = option_type * price_temp + (1 - option_type) * (price_temp - S + K * np.exp(-r * T))

    return price
```

Problem 6

In this part, I use the Bisection method to invert the option-implied volatility. Due to limited computational resources, I set the maximum number of iterations (2000) and the tolerance (10^{-6}) as the stopping criteria.

The procedure is formulated as:

$$\text{Initialize: } \sigma_{\text{low}}, \sigma_{\text{up}}$$

$$\text{While } |\sigma_{\text{up}} - \sigma_{\text{low}}| > \delta:$$

$$\sigma_{\text{mid}} = \frac{\sigma_{\text{low}} + \sigma_{\text{up}}}{2}$$

$$C_{\text{mid}} = C_{BS}(S_0, K, T, r, \sigma_{\text{mid}})$$

$$\text{If } C_{\text{mid}} < C_{\text{market}}: \quad \sigma_{\text{low}} = \sigma_{\text{mid}}$$

$$\text{Else}: \quad \sigma_{\text{up}} = \sigma_{\text{mid}}$$

Note that some option prices lie below intrinsic value: for call options, $C < \max(S_0 - Ke^{-rT}, 0)$; for put options, $P < \max(Ke^{-rT} - S_0, 0)$). For such options, even if the implied volatility is set larger than zero, the Black–Scholes price would still be higher than the market price. A typical approach is to directly remove them. However, this would lead to the absence of option pairs for certain strikes and maturities. Given that we have already filtered a large portion in the preceding parts, I set their IV equal to zero to preserve more of the IV surface information.

In [ ]:
```python
# Define the implied volatility function using bisection method
def sigma_bisection(S, K, T, r, price, option_type, sigma_up, sigma_low, max_iter, tol):
    intrinsic_value = max(0, S - K * np.exp(-r * T)) if option_type == 1 else max(0, K * np.exp(-r * T) - S)
    # If the option price is lower than the intrinsic value, it will be set to zero
    if price < intrinsic_value:
        return 0

    diff_low = price_BS(S, K, T, r, sigma_low, option_type) - price
    diff_up = price_BS(S, K, T, r, sigma_up, option_type) - price

    try:
        if diff_low * diff_up > 0:
            raise ValueError("The implied volatility boundary is not wide enough.")

        for _ in range(max_iter):
            sigma_mid = (sigma_low + sigma_up) / 2
            price_mid = price_BS(S, K, T, r, sigma_mid, option_type)
            if abs(price_mid - price) < tol:
                return sigma_mid
            elif price_mid < price:
                sigma_low = sigma_mid
            else:
                sigma_up = sigma_mid

        return sigma_mid

    except ValueError as e:
        print(f"Error: {e} S: {S}, K: {K}, T: {T}, r: {r}, price: {price}, option_type: {option_type}")
        return None
```

In [ ]:
```python
# Define the underlying assets
asset_all = ["TSLA","SPY"]
# asset_all = ["TSLA"]
# Import Data path
filename = f'C:/Users/18509/Desktop/Course/FE 621 A/Assignment 1/HW1_DATA2.xlsx'

# Preset the parameters for the bisection method
# Initial upper and lower bounds
sigma_up = 10
sigma_low = 1e-10
```

```python
# Maximum number of iterations
max_iter = 2000
# Tolerance level
tol = 1e-6

# record the overall running time
time_start = time.time()
for k in asset_all:
    # Import data and filter data
    option_data = pd.read_excel(filename, sheet_name=f'{k}_option')
    # Compute implied volatility computation using bisection method
    option_data['IV_bisec'] = option_data.apply(lambda x: sigma_bisection(x['S'], x['K'], x['T'], x['r'], x['mid'], x['type'],
                                                          sigma_up, sigma_low, max_iter, tol), axis=1)
    globals()[f'{k}_option_data'] = option_data
    print(f"{k} option implied volatility using bisection method:")
    print(globals()[f'{k}_option_data']['IV_bisec'].describe())
time_end = time.time()
print(f"Overall Running time of bisection method: {time_end - time_start:.4f} seconds")
# print(option_data.head())
```

```
TSLA option implied volatility using bisection method:
count    846.000000
mean       0.448431
std        0.237799
min        0.000000
25%        0.381718
50%        0.422142
75%        0.492521
max        2.310909
Name: IV_bisec, dtype: float64
SPY option implied volatility using bisection method:
count    504.000000
mean       0.194886
std        0.113101
min        0.000000
25%        0.148649
50%        0.168793
75%        0.206200
max        0.957718
Name: IV_bisec, dtype: float64
Overall Running time of bisection method: 2.2664 seconds
```

Report the implied volatility at the money and average all the implied volatilities for the options between in-the-money and out-of-the-money.

In [ ]:
```
for k in asset_all:
    maturity = globals()[f'{k}_option_data']['maturity'].unique()
    for i in maturity:
        temp = globals()[f'{k}_option_data'][globals()[f'{k}_option_data']['maturity'] == i].copy()

        # Find the ATM option (where strike is closest to spot price)
        temp['strike_diff'] = abs(temp['K'] - temp['S'])
        atm_index = temp['strike_diff'].idxmin()
        atm_iv = temp.loc[atm_index, 'IV_bisec']
        print(f"{k} option ATM IV for maturity {i}: {atm_iv:.4f}")

        # Average IV for each maturity
        avg_iv = temp['IV_bisec'].mean()
        print(f"{k} option Average IV for maturity {i}: {avg_iv:.4f}")
```

TSLA option ATM IV for maturity 2026-02-18: 0.3509
TSLA option Average IV for maturity 2026-02-18: 0.4079
TSLA option ATM IV for maturity 2026-02-20: 0.3776
TSLA option Average IV for maturity 2026-02-20: 0.5252
TSLA option ATM IV for maturity 2026-02-23: 0.3949
TSLA option Average IV for maturity 2026-02-23: 0.3985
TSLA option ATM IV for maturity 2026-02-25: 0.3947
TSLA option Average IV for maturity 2026-02-25: 0.3769
TSLA option ATM IV for maturity 2026-02-27: 0.3992
TSLA option Average IV for maturity 2026-02-27: 0.4359
TSLA option ATM IV for maturity 2026-03-06: 0.4124
TSLA option Average IV for maturity 2026-03-06: 0.4236
TSLA option ATM IV for maturity 2026-03-13: 0.4161
TSLA option Average IV for maturity 2026-03-13: 0.4490
TSLA option ATM IV for maturity 2026-03-20: 0.4295
TSLA option Average IV for maturity 2026-03-20: 0.4709
TSLA option ATM IV for maturity 2026-03-27: 0.4274
TSLA option Average IV for maturity 2026-03-27: 0.4271
TSLA option ATM IV for maturity 2026-04-17: 0.4403
TSLA option Average IV for maturity 2026-04-17: 0.4355
TSLA option ATM IV for maturity 2026-05-15: 0.4671
TSLA option Average IV for maturity 2026-05-15: 0.4781
SPY option ATM IV for maturity 2026-02-18: 0.1097
SPY option Average IV for maturity 2026-02-18: 0.2295
SPY option ATM IV for maturity 2026-02-19: 0.1966
SPY option Average IV for maturity 2026-02-19: 0.1914
SPY option ATM IV for maturity 2026-02-20: 0.2594
SPY option Average IV for maturity 2026-02-20: 0.2941
SPY option ATM IV for maturity 2026-02-23: 0.2062
SPY option Average IV for maturity 2026-02-23: 0.1545
SPY option ATM IV for maturity 2026-02-24: 0.1947
SPY option Average IV for maturity 2026-02-24: 0.1538
SPY option ATM IV for maturity 2026-02-27: 0.1919
SPY option Average IV for maturity 2026-02-27: 0.2141
SPY option ATM IV for maturity 2026-03-06: 0.1901
SPY option Average IV for maturity 2026-03-06: 0.2005
SPY option ATM IV for maturity 2026-03-13: 0.1817
SPY option Average IV for maturity 2026-03-13: 0.1844
SPY option ATM IV for maturity 2026-03-20: 0.1745
SPY option Average IV for maturity 2026-03-20: 0.2420
SPY option ATM IV for maturity 2026-03-27: 0.1638

```
SPY option Average IV for maturity 2026-03-27: 0.1650
SPY option ATM IV for maturity 2026-03-31: 0.1780
SPY option Average IV for maturity 2026-03-31: 0.1873
SPY option ATM IV for maturity 2026-04-17: 0.1739
SPY option Average IV for maturity 2026-04-17: 0.1808
SPY option ATM IV for maturity 2026-04-30: 0.1716
SPY option Average IV for maturity 2026-04-30: 0.1738
SPY option ATM IV for maturity 2026-05-15: 0.1691
SPY option Average IV for maturity 2026-05-15: 0.1904
```

Problem 7

In this part, I use the Newton method method to invert the option-implied volatility. Due to limited computational resources, I set the maximum number of iterations (2000) and the tolerance (10^{-6}) as the stopping criteria.

The procedure is formulated as:

$$\text{Initialize: } \sigma_0$$

$$\text{For } n = 0, 1, 2, \ldots:$$

$$d_1 = \frac{\ln(S_0/K) + (r + \sigma_n^2/2)T}{\sigma_n\sqrt{T}}$$

$$C_{BS}(\sigma_n) = S_0 N(d_1) - K e^{-rT} N(d_1 - \sigma_n\sqrt{T})$$

$$\text{Vega}_n = S_0\phi(d_1)\sqrt{T}$$

$$\sigma_{n+1} = \sigma_n - \frac{C_{BS}(\sigma_n) - C_{\text{mkt}}}{\text{Vega}_n}$$

$$\text{If } |\sigma_{n+1} - \sigma_n| < \delta: \quad \text{break}$$

The approach for handling option prices that lie below intrinsic value is the same as in the previous problem

In [ ]:
```
# Define the implied volatility function using the Newton-Raphson method
def sigma_newton(S, K, T, r, price, option_type, sigma_init, max_iter, tol):
```

```
    intrinsic_value = max(0, S - K * np.exp(-r * T)) if option_type == 1 else max(0, K * np.exp(-r * T) - S)
    # If the option price is lower than the intrinsic value, it will be set to zero
    if price < intrinsic_value:
        return 0

    sigma = sigma_init
    for _ in range(max_iter):
        price_est = price_BS(S, K, T, r, sigma, option_type)
        vega = (S * norm.pdf((np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))) * np.sqrt(T))
        if vega == 0:
            # print(f"Warning: Vega is zero. Return zero sigma estimation")
            return 0

        sigma_new = sigma - (price_est - price) / vega
        if abs(sigma_new - sigma) < tol:
            return sigma_new

        sigma = sigma_new

    return sigma
```

In [ ]:

```
# Define the underlying assets
asset_all = ["TSLA","SPY"]
# asset_all = ["TSLA"]
# Import Data path
filename = f'C:/Users/18509/Desktop/Course/FE 621 A/Assignment 1/HW1_DATA2.xlsx'

# Preset the parameters for the Newton method
# Initial upper and lower bounds
sigma_ini = 1
# Maximum number of iterations
max_iter = 2000
# Tolerance level
tol = 1e-6

# record the overall running time
time_start = time.time()
for k in asset_all:
    # Import data and filter data
    option_data = pd.read_excel(filename, sheet_name=f'{k}_option')
```

```
    # Compute implied volatility computation using Newton method
    globals()[f'{k}_option_data']['IV_newton'] = option_data.apply(lambda x: sigma_newton(x['S'], x['K'], x['T'], x['r'], x['m
                                                    sigma_ini, max_iter, tol), axis=1)
    print(f"{k} option implied volatility using Newton method:")
    print(globals()[f'{k}_option_data']['IV_newton'].describe())
time_end = time.time()
print(f"Overall Running time of Newton method: {time_end - time_start:.4f} seconds")
# print(option_data.head())
```

```
TSLA option implied volatility using Newton method:
count    846.000000
mean       0.439626
std        0.218056
min        0.000000
25%        0.381085
50%        0.421342
75%        0.490297
max        1.880292
Name: IV_newton, dtype: float64
SPY option implied volatility using Newton method:
count    504.000000
mean       0.194886
std        0.113101
min        0.000000
25%        0.148649
50%        0.168793
75%        0.206200
max        0.957716
Name: IV_newton, dtype: float64
Overall Running time of Newton method: 0.6690 seconds
```

- It can be observed that, under the same stopping criteria, the total running time of the Bisection method is approximately 2.2 seconds, while the Newton method requires about 0.7 seconds. In other words, the Newton method is more than three times faster than the Bisection method.

Problem 8

If I understand the assignment correctly, I need to average the volatility within the at-the-money (ATM), in-the-money (ITM), and out-of-the-money (OTM) options for each maturity, separately.

The option monneyness is given by

$$M = \begin{cases} \ln(S/K) & \text{call} \\ \ln(K/S) & \text{put} \end{cases}$$

Moreover, I define values of moneyness between 0.95 and 1.05 for ATM region

Without loss of generality, I use the estimated implied volatility from the Newton method for subsequent analysis.

In [ ]:
```
# Report the implied volatility values obtained for every maturity, option type and stock
# Call option is labeled as 1, and put option is labeled as 0 in the 'option_type' column
for k in asset_all:
    print(f"{k} option implied volatility by maturity and option type:")
    print(globals()[f'{k}_option_data'][['K','maturity','type','IV_newton']])
```

```
TSLA option implied volatility by maturity and option type:
         K    maturity  type  IV_newton
0    322.5  2026-02-18     1   0.866780
1    342.5  2026-02-18     1   0.000000
2    355.0  2026-02-18     1   0.975347
3    357.5  2026-02-18     1   0.000000
4    360.0  2026-02-18     1   1.052768
..     ...         ...   ...        ...
841  460.0  2026-05-15     0   0.482157
842  470.0  2026-05-15     0   0.504317
843  480.0  2026-05-15     0   0.513550
844  490.0  2026-05-15     0   0.464225
845  500.0  2026-05-15     0   0.441031

[846 rows x 4 columns]
SPY option implied volatility by maturity and option type:
       K    maturity  type  IV_newton
0    709  2026-02-18     1   0.109693
1    710  2026-02-18     1   0.140716
2    709  2026-02-18     0   0.294402
3    710  2026-02-18     0   0.373364
4    672  2026-02-19     1   0.196602
..   ...         ...   ...        ...
499  705  2026-05-15     0   0.167704
500  710  2026-05-15     0   0.176885
501  714  2026-05-15     0   0.167090
502  720  2026-05-15     0   0.170653
503  770  2026-05-15     0   0.231567

[504 rows x 4 columns]
```

Note that for some ITM and OTM regions, none of the available data meet the above filtration criteria, such as zero bid or ask prices. Thus, they are removed. The details are described in Problems 2 and 4. As a compromise, we directly set their average implied volatility equal to that of the ATM options.

To compare the average implied volatility (IV) of TSLA and SPY options with the VIX, I first set the close price of VIX as the benchmark, and then plot the term structures of the average IV for TSLA and SPY options alongside the VIX.

In [ ]:

```
# Average the volatility within the at-the-money (ATM), in-the-money (ITM), and out-of-the-money (OTM) options for each maturi
for k in asset_all:
    maturity = globals()[f'{k}_option_data']['maturity'].unique()
    iv_temp = []

    for i in maturity:
        temp = globals()[f'{k}_option_data'][globals()[f'{k}_option_data']['maturity'] == i].copy()

        # Compute the moneyness
        temp['moneyness'] = temp.apply(lambda x: x['S']/x['K'] if x['type'] == 1 else x['K']/x['S'], axis=1)

        # Find the ATM region
        M1 = 0.95
        M2 = 1.05
        temp_atm = (temp['moneyness'] >= M1) & (temp['moneyness'] <= M2)
        avg_iv_atm = temp[temp_atm]['IV_newton'].mean()
        # print(f"{k} option ATM average IV for maturity {i}: {avg_iv_atm:.4f}")

        # Find the ITM region
        temp_itm = (temp['moneyness'] > M2) & (temp['type'] == 1) | (temp['moneyness'] < M1) & (temp['type'] == 0)
        avg_iv_itm = temp[temp_itm]['IV_newton'].mean()
        # print(f"{k} option ITM average IV for maturity {i}: {avg_iv_itm:.4f}")

        # Find the OTM region
        temp_otm = (temp['moneyness'] < M1) & (temp['type'] == 1) | (temp['moneyness'] > M2) & (temp['type'] == 0)
        avg_iv_otm = temp[temp_otm]['IV_newton'].mean()
        # print(f"{k} option OTM average IV for maturity {i}: {avg_iv_otm:.4f}")

        iv_temp.append({'maturity': i, 'avg_iv_itm': avg_iv_itm, 'avg_iv_atm': avg_iv_atm, 'avg_iv_otm': avg_iv_otm})
        # Set the outlier at the ITM and OTM region to be the average IV of the ATM region, with the assumption that the IV sm
        iv_temp[-1]['avg_iv_itm'] = iv_temp[-1]['avg_iv_atm'] if not (iv_temp[-1]['avg_iv_itm'] > 0) else iv_temp[-1]['avg_iv_
        iv_temp[-1]['avg_iv_otm'] = iv_temp[-1]['avg_iv_atm'] if not (iv_temp[-1]['avg_iv_otm'] > 0) else iv_temp[-1]['avg_iv_

    globals()[f'{k}_IV_table'] = pd.DataFrame(iv_temp)
    print(f"{k} option average IV by maturity and moneyness:")
    print(globals()[f'{k}_IV_table'])

today = "2026-02-13"
VIX_index = yf.Ticker("^VIX")
VIX_data = VIX_index.history(start=today,period='1d')
```

```
VIX_data = VIX_data['Close'].iloc[-1]
VIX_data = VIX_data/100
print(f'VIX value:{VIX_data}')
```

```
TSLA option average IV by maturity and moneyness:
      maturity  avg_iv_itm  avg_iv_atm  avg_iv_otm
0   2026-02-18    0.451948    0.365618    0.397128
1   2026-02-20    0.536167    0.380860    0.503653
2   2026-02-23    0.493988    0.396458    0.356201
3   2026-02-25    0.397696    0.397696    0.354258
4   2026-02-27    0.423772    0.396431    0.416495
5   2026-03-06    0.380561    0.409069    0.476713
6   2026-03-13    0.477064    0.420344    0.435429
7   2026-03-20    0.486268    0.425528    0.467621
8   2026-03-27    0.430073    0.424086    0.426758
9   2026-04-17    0.408144    0.435500    0.459406
10  2026-05-15    0.490433    0.469265    0.468983
SPY option average IV by maturity and moneyness:
      maturity  avg_iv_itm  avg_iv_atm  avg_iv_otm
0   2026-02-18    0.229544    0.229544    0.229544
1   2026-02-19    0.191357    0.191357    0.191357
2   2026-02-20    0.320832    0.237621    0.257186
3   2026-02-23    0.154540    0.154540    0.154540
4   2026-02-24    0.153759    0.153759    0.153759
5   2026-02-27    0.292847    0.178578    0.167030
6   2026-03-06    0.306453    0.173765    0.158074
7   2026-03-13    0.293014    0.161417    0.166370
8   2026-03-20    0.324503    0.165461    0.165461
9   2026-03-27    0.181640    0.161525    0.161525
10  2026-03-31    0.250980    0.156581    0.185175
11  2026-04-17    0.223321    0.162876    0.195730
12  2026-04-30    0.227522    0.166226    0.153501
13  2026-05-15    0.256745    0.165311    0.162281
VIX value:0.20600000381469727
```

- The above results show that the overall average implied volatility of TSLA options is larger than that of SPY options. In other words, the market anticipates that a single stock is more volatile than a market index.

- In addition, the average implied volatility of SPY options is closer to the VIX than that of TSLA options. This aligns with the intuition that the VIX is computed based on SPX options
- Finally, as moneyness increases (i.e., moving from OTM to ITM options) or as maturity increases, the average implied volatility typically rise.

Problem 9

In this part, I compute the price difference as measured by Put-Call parity,

$$D = C - P - S_0 + Ke^{-rT}$$

Note that in the data collection step, I use minute-level option data and underlying data. Consequently, for a specific maturity and strike, the last quote times for the call and put options may differ, causing their maturities and underlying prices to diverge slightly. As a compromise, I use the average value of the maturities and the underlying prices as the proxy.

After that, I compare the price differences at the mid point and bid/ask value.

In [ ]:
```python
# compute the price difference as measured by Put-Call parity
for k in asset_all:
    option_temp = globals()[f'{k}_option_data'].copy()
    diff_temp = []
    data_group = option_temp.groupby(['K', 'maturity'], sort=False)

    for (K,maturity), group in data_group:
        S = np.mean(group['S'])
        K = np.mean(group['K'])
        r = np.mean(group['r'])
        T = np.mean(group['T'])
        # Use the mid point price for the put-call parity calculation
        C = group[group['type'] == 1]['mid'].iloc[0]
        P = group[group['type'] == 0]['mid'].iloc[0]
        diff_mid = C - P - (S - K * np.exp(-r * T))

        # Use the ask price for the put-call parity calculation
        C_ask = group[group['type'] == 1]['ask'].iloc[0]
```

```
        P_ask = group[group['type'] == 0]['ask'].iloc[0]
        diff_ask = C_ask - P_ask - (S - K * np.exp(-r * T))

        # Use the bid price for the put-call parity calculation
        C_bid = group[group['type'] == 1]['bid'].iloc[0]
        P_bid = group[group['type'] == 0]['bid'].iloc[0]
        diff_bid = C_bid - P_bid - (S - K * np.exp(-r * T))

        diff_temp.append({'maturity': maturity, 'strike': K, 'mid_diff': diff_mid, 'ask_diff': diff_ask, 'bid_diff': diff_bid}
    globals()[f'{k}_diff'] = pd.DataFrame(diff_temp)
    print(f"{k} option price difference by maturity and strike:")
    print(globals()[f'{k}_diff'])
```

```
TSLA option price difference by maturity and strike:
       maturity  strike  mid_diff  ask_diff  bid_diff
0    2026-02-18   322.5  0.275183  1.140183 -0.589817
1    2026-02-18   342.5 -1.492548 -0.627548 -2.357548
2    2026-02-18   355.0  1.287044  3.032044 -0.457956
3    2026-02-18   357.5 -0.338207  0.881793 -1.558207
4    2026-02-18   360.0  1.766217  3.406217  0.126217
..          ...     ...       ...       ...       ...
418  2026-05-15   460.0 -1.505776 -1.555776 -1.455776
419  2026-05-15   470.0 -2.874627 -3.849627 -1.899627
420  2026-05-15   480.0 -3.476442 -4.276442 -2.676442
421  2026-05-15   490.0 -0.504924 -1.329924  0.320076
422  2026-05-15   500.0  0.494741 -0.155259  1.144741

[423 rows x 5 columns]
SPY option price difference by maturity and strike:
       maturity  strike  mid_diff  ask_diff  bid_diff
0    2026-02-18   709.0 -2.810290 -4.165290 -1.455290
1    2026-02-18   710.0 -1.826226 -3.181226 -0.471226
2    2026-02-19   672.0  0.248391  0.383391  0.113391
3    2026-02-20   495.0 -0.355093  0.964907 -1.675093
4    2026-02-20   565.0 -3.286481 -3.081481 -3.491481
..          ...     ...       ...       ...       ...
247  2026-05-15   705.0 -3.129303 -4.859303 -1.399303
248  2026-05-15   710.0 -4.751070 -6.481070 -3.021070
249  2026-05-15   714.0 -5.261721 -6.996721 -3.526721
250  2026-05-15   720.0 -4.133861 -5.753861 -2.513861
251  2026-05-15   770.0 -8.127757 -9.647757 -6.607757

[252 rows x 5 columns]
```

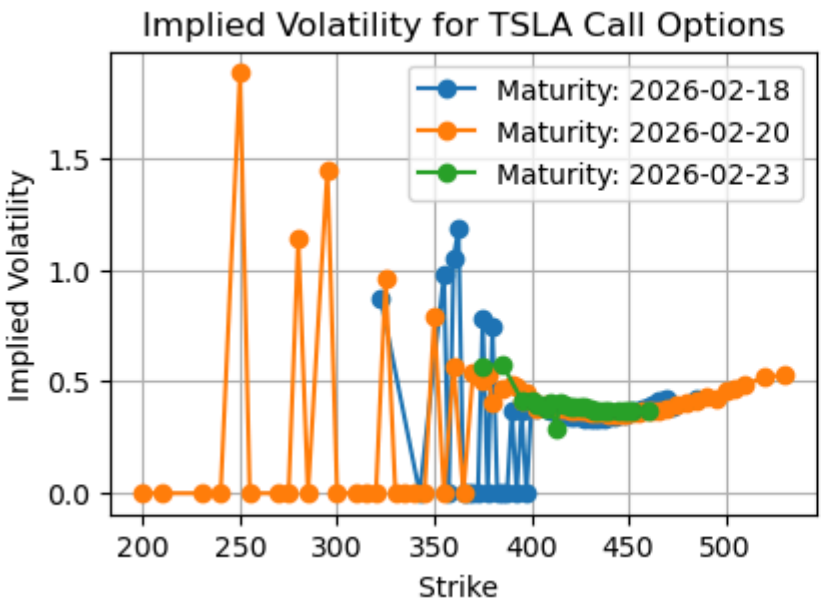
The above results indicate that most of the price differences are negative. This suggests that, for a specific maturity and strike, the put option is relatively more expensive than the corresponding call option. Some empirical studies have drawn similar conclusions and have explained that this may stem from the insurance provided by put options during market crash periods.

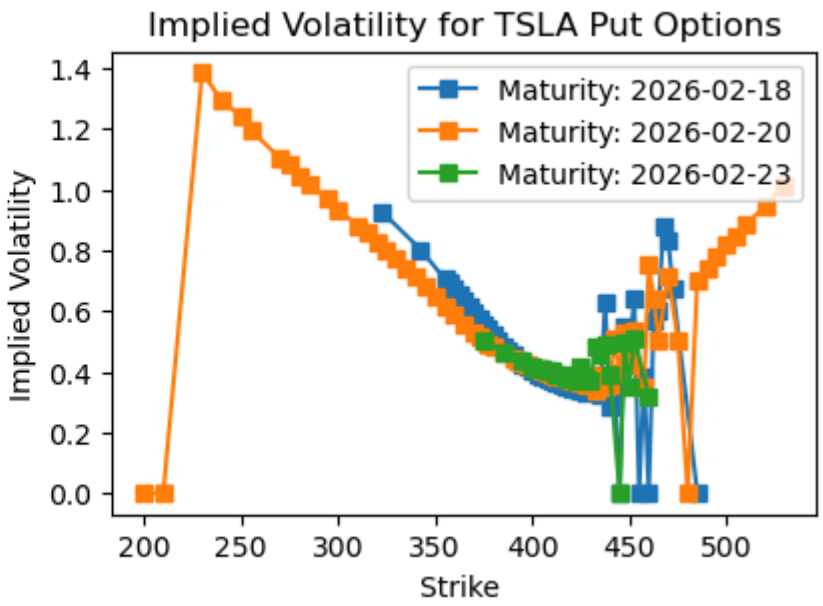
Additionally, the price difference implied by the midpoint is typically larger than that of the ask value, but smaller than that of the bid value. I think this may stem from market makers or other large financial institutions acting as option sellers, which can provide more reasonable quotes.

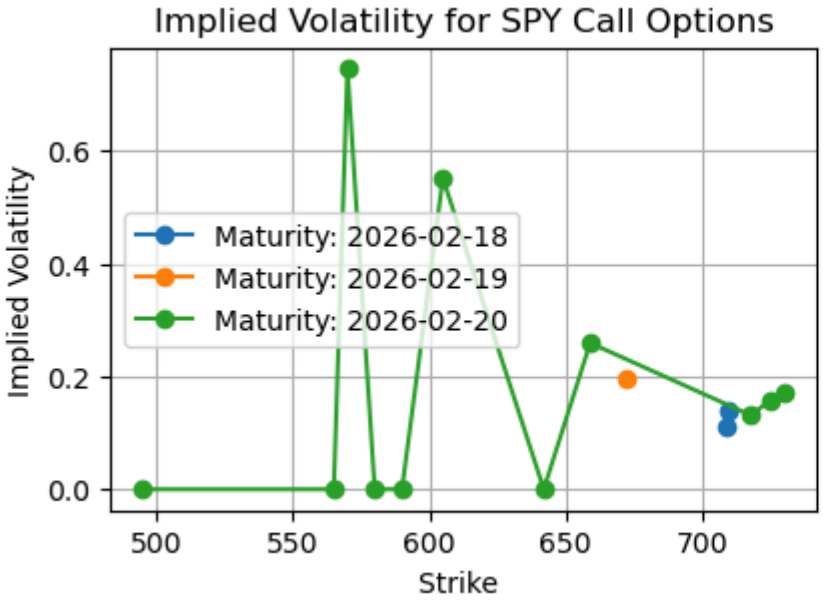
Problem 10

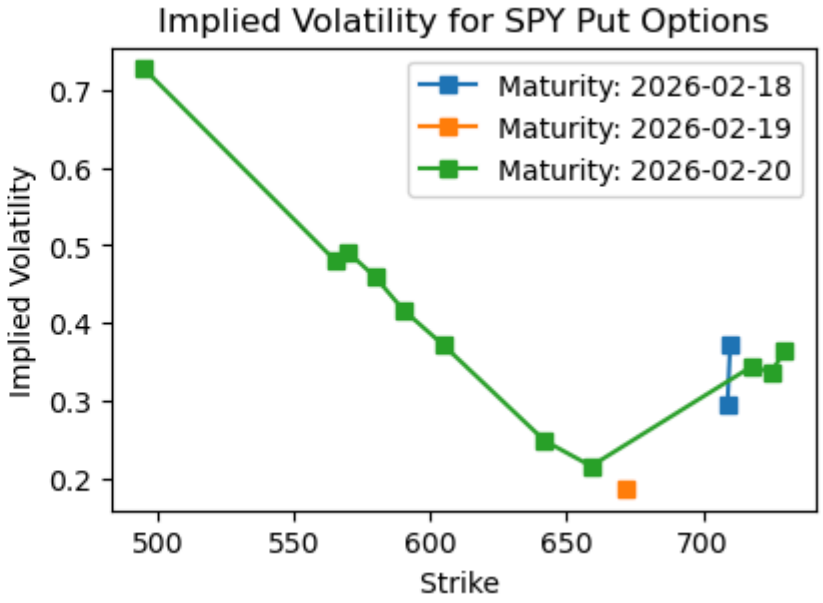
Create a two-dimensional plot of implied volatilities versus strike price K for the three nearest-to-maturity options.

In [ ]:
```
# Create a two-dimensional plot of implied volatilities versus strike price K for the three nearest-to-maturity options for ea
for k in asset_all:
    option_temp = globals()[f'{k}_option_data'].copy()
    maturity = option_temp['maturity'].unique()
    # Obtain the three nearest-to-maturity
    maturity = sorted(maturity)[:3]
    option_temp = option_temp[option_temp['maturity'].isin(maturity)]
    fig, (ax_call,ax_put) = plt.subplots(1,2,figsize=(10,3))
    for m in maturity:
        # Call option
        temp = option_temp[(option_temp['maturity'] == m) & ((option_temp['type'] == 1))].copy()
        ax_call.plot(temp['K'], temp['IV_newton'], marker='o', label=f'Maturity: {m}')
        ax_call.set_title(f'Implied Volatility for {k} Call Options')
        ax_call.set_xlabel('Strike ')
        ax_call.set_ylabel('Implied Volatility')
        ax_call.legend()
        ax_call.grid()


        # Put option
        temp = option_temp[(option_temp['maturity'] == m) & ((option_temp['type'] == 0))].copy()
        ax_put.plot(temp['K'], temp['IV_newton'], marker='s', label=f'Maturity: {m}')
        ax_put.set_title(f'Implied Volatility for {k} Put Options')
        ax_put.set_xlabel('Strike ')
        ax_put.set_ylabel('Implied Volatility')
        ax_put.legend()
        ax_put.grid()

    # plt.tight_layout()
    plt.grid()
    plt.show()
```









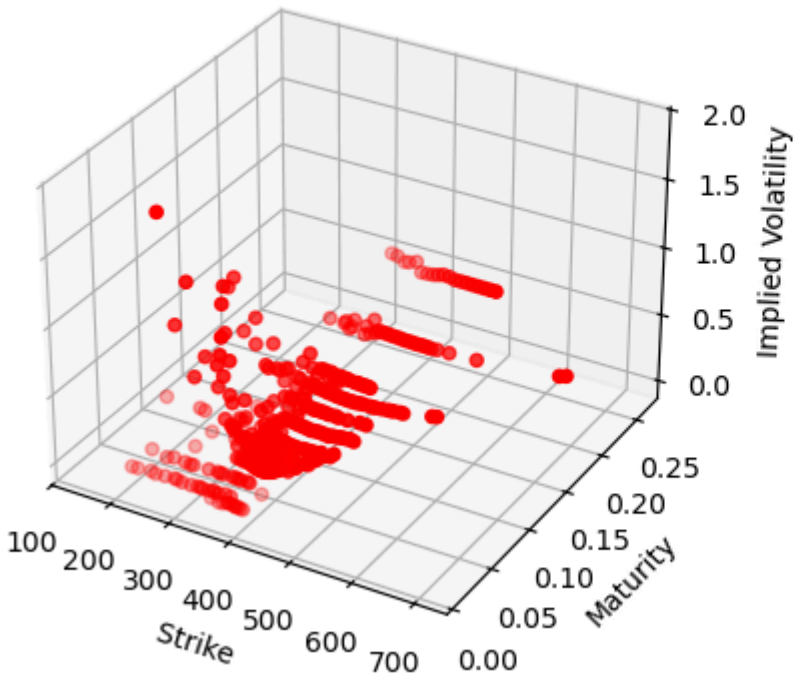
Note that for SPY options, only a few implied volatilities are available because these data do not meet the above filtration criteria, such as zero bid or ask prices. Consequently, they are removed. The details are described in Problems 2 and 4.

- The figures show that the curvature and slope at the ATM option are typically more pronounced at shorter maturities.

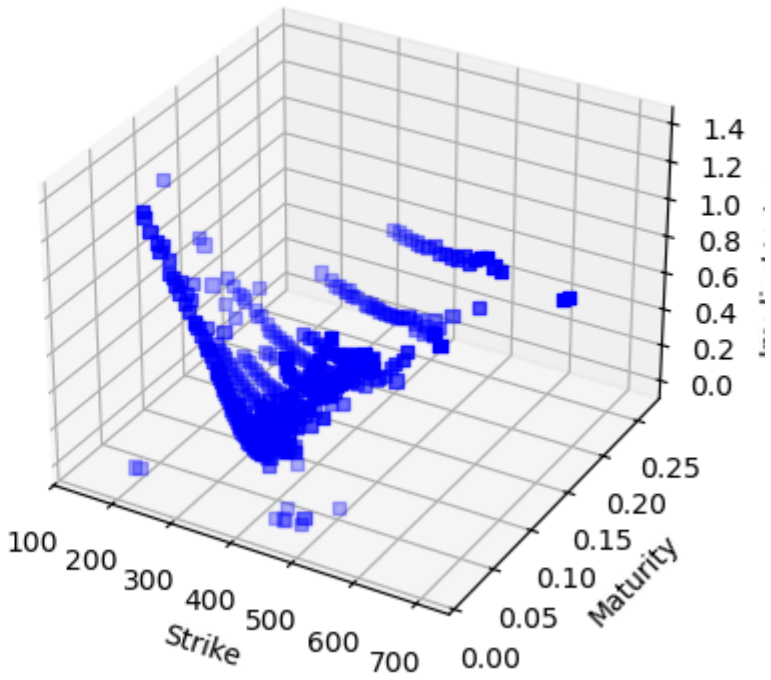
Bonus 2

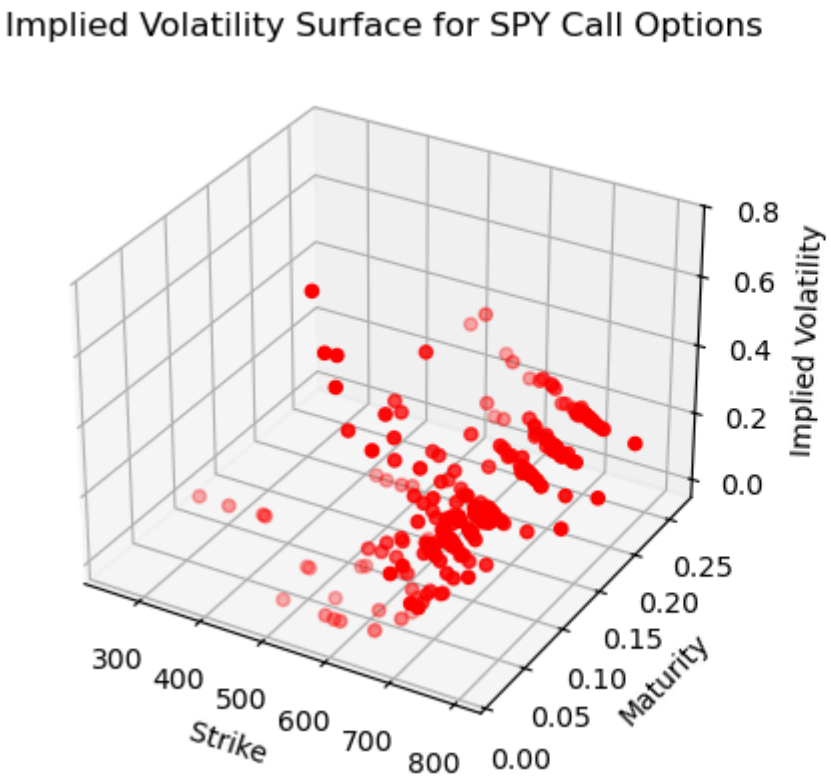
In [ ]:
```
# Create a 3D plot of implied volatilities versus strike price K for the all options for each underlying asset
for k in asset_all:
    option_temp = globals()[f'{k}_option_data'].copy()
    maturity = option_temp['maturity'].unique()
    fig = plt.figure(figsize=(10,6))
    ax_call = fig.add_subplot(121, projection='3d') # 121 means 1 row and 2 columns, and this is the first subplot
    ax_put = fig.add_subplot(122, projection='3d')
    for m in maturity:
        # Call option
        temp = option_temp[(option_temp['maturity'] == m) & ((option_temp['type'] == 1))].copy()
        ax_call.scatter(temp['K'], temp['T'], temp['IV_newton'], c ='red', marker='o')
        ax_call.set_title(f'Implied Volatility Surface for {k} Call Options')
        ax_call.set_xlabel('Strike ')
        ax_call.set_ylabel('Maturity')
        ax_call.set_zlabel('Implied Volatility')
        ax_call.grid()


        # Put option
        temp = option_temp[(option_temp['maturity'] == m) & ((option_temp['type'] == 0))].copy()
        ax_put.scatter(temp['K'], temp['T'], temp['IV_newton'], c ='blue', marker='s')
        ax_put.set_title(f'Implied Volatility surface for {k} Put Options')
        ax_put.set_xlabel('Strike')
        ax_put.set_ylabel('Maturity')
        ax_put.set_zlabel('Implied Volatility')
        ax_put.grid()

    # plt.tight_layout()
    plt.grid()
    plt.show()
```

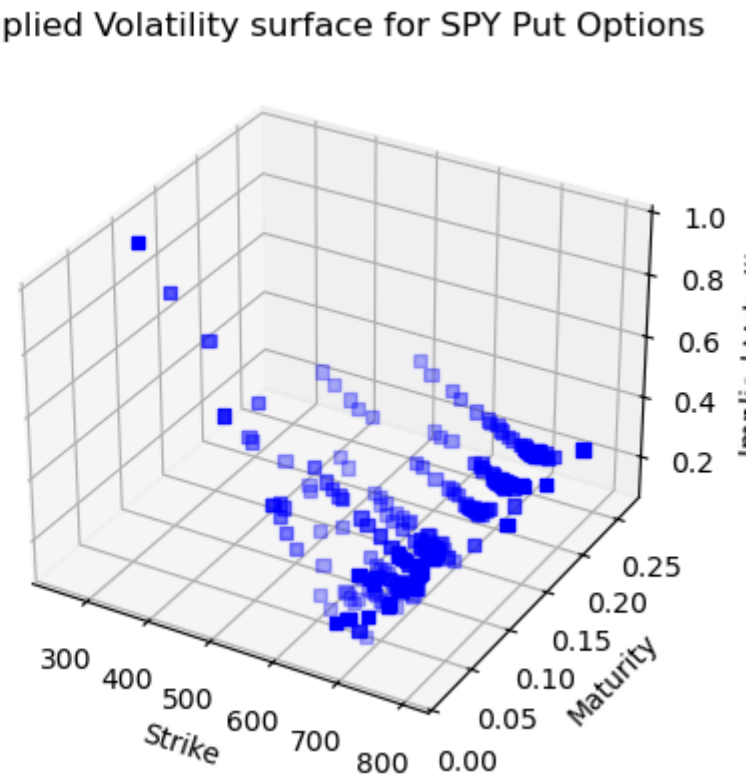
Implied Volatility Surface for TSLA Call Options



Implied Volatility surface for TSLA Put Options







Problem 11

In this part, I calculate the derivatives of the call option price with respect to S (Δ and Γ), and vega (ν).

I use the closed-form formula from the Black–Scholes model and a finite-difference approximation method.

- Analytical Formulas

$$\Delta_{\text{call}} = N(d_1)$$

$$\Delta_{\text{put}} = N(d_1) - 1$$

$$\Gamma = \frac{\phi(d_1)}{S_0 \sigma \sqrt{T}}$$

$$\nu = S_0 \phi(d_1) \sqrt{T}$$

where:

$$d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

$$\phi(d_1) = \frac{1}{\sqrt{2\pi}} e^{-d_1^2/2}$$

- Finite Difference

$$\Delta \approx \frac{C_{BS}(S_0 + h) - C_{BS}(S_0 - h)}{2h}$$

$$\Gamma \approx \frac{C_{BS}(S_0 + h) - 2C_{BS}(S_0) + C_{BS}(S_0 - h)}{h^2}$$

$$\nu \approx \frac{C_{BS}(\sigma + h) - C_{BS}(\sigma - h)}{2h}$$

where h is a small number, which I set as 0.001.

In [ ]:
```python
# Define a function to compute the delta, gamma, and vega of the option using the Black-Scholes formula
def greeks_BS_closed(S, K, T, r, sigma, option_type):
    # For zero volatility, the greeks will be set to zero
    if sigma == 0:
        return 0, 0, 0
    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    delta = norm.cdf(d1) * option_type + (1-option_type) * (norm.cdf(d1) - 1)
    gamma = norm.pdf(d1) / (S * sigma * np.sqrt(T))
    vega = S * norm.pdf(d1) * np.sqrt(T)

    return delta, gamma, vega
```

In [ ]:
```python
# Define a function to compute the delta, gamma, and vega of the option using the finite difference method
def greeks_finite_diff(S, K, T, r, sigma, option_type, h):
    # For zero volatility, the greeks will be set to zero
    if sigma == 0:
        return 0, 0, 0

    price = price_BS(S, K, T, r, sigma, option_type)
    price_up = price_BS(S + h, K, T, r, sigma, option_type)
    price_down = price_BS(S - h, K, T, r, sigma, option_type)
    delta = (price_up - price_down) / (2 * h)
    gamma = (price_up - 2 * price + price_down) / (h ** 2)

    price_sigma_up = price_BS(S, K, T, r, sigma + h, option_type)
    price_sigma_down = price_BS(S, K, T, r, sigma - h, option_type)
    vega = (price_sigma_up - price_sigma_down) / (2 * h)

    return delta, gamma, vega
```

In [ ]:
```python
# Define the small number used for the finite difference method
h = 1e-3
for k in asset_all:
    option_temp = globals()[f'{k}_option_data'].copy()
    # Compute the greeks using the closed-form formula
    # Output multiple columns by setting result_type='expand'
    globals()[f'{k}_option_data'][['delta_closed', 'gamma_closed', 'vega_closed']] = \
        option_temp.apply(lambda x: greeks_BS_closed(x['S'], x['K'], x['T'], x['r'], x['IV_newton'], x['type']), axis=1, resul
    # Compute the greeks using the finite difference method
```

```
    globals()[f'{k}_option_data'][['delta_finite', 'gamma_finite', 'vega_finite']] = \
        option_temp.apply(lambda x: greeks_finite_diff(x['S'], x['K'], x['T'], x['r'], x['IV_newton'], x['type'], h), axis=1,
    print(f"{k} option greeks:")
    print(globals()[f'{k}_option_data'][['K','maturity','type','delta_closed','delta_finite','gamma_closed','gamma_finite','ve
```

TSLA option greeks:

	K	maturity	type	delta_closed	delta_finite	gamma_closed	\
0	322.5	2026-02-18	1	0.997133	0.997133	0.000221	
1	342.5	2026-02-18	1	0.000000	0.000000	0.000000	
2	355.0	2026-02-18	1	0.910673	0.910673	0.003168	
3	357.5	2026-02-18	1	0.000000	0.000000	0.000000	
4	360.0	2026-02-18	1	0.870719	0.870719	0.003866	
..	...	...	...	...	...	...	
841	460.0	2026-05-15	0	-0.580873	-0.580873	0.003767	
842	470.0	2026-05-15	0	-0.595320	-0.595320	0.003551	
843	480.0	2026-05-15	0	-0.622686	-0.622686	0.003419	
844	490.0	2026-05-15	0	-0.697881	-0.697881	0.003513	
845	500.0	2026-05-15	0	-0.748638	-0.748638	0.003391	

	gamma_finite	vega_closed	vega_finite
0	0.000221	0.402703	0.402706
1	0.000000	0.000000	0.000000
2	0.003168	8.439320	8.439317
3	0.000000	0.000000	0.000000
4	0.003866	10.910508	10.910505
..	...	...	...
841	0.003767	84.050722	84.050703
842	0.003551	83.848361	83.848338
843	0.003419	82.203122	82.203092
844	0.003514	74.591251	74.591189
845	0.003391	67.903032	67.902945

[846 rows x 9 columns]

SPY option greeks:

	K	maturity	type	delta_closed	delta_finite	gamma_closed	\
0	709	2026-02-18	1	0.005234	0.005234	0.001662	
1	710	2026-02-18	1	0.004518	0.004518	0.001240	
2	709	2026-02-18	0	-0.838511	-0.838511	0.009958	
3	710	2026-02-18	0	-0.775363	-0.775363	0.009755	
4	672	2026-02-19	1	0.721403	0.721403	0.019807	
..	...	...	...	...	...	...	
499	705	2026-05-15	0	-0.584511	-0.584511	0.006666	
500	710	2026-05-15	0	-0.593176	-0.593176	0.006247	
501	714	2026-05-15	0	-0.640619	-0.640619	0.006412	
502	720	2026-05-15	0	-0.668704	-0.668704	0.006056	
503	770	2026-05-15	0	-0.811971	-0.811971	0.003344	

```
     gamma_finite  vega_closed  vega_finite
0        0.001662     1.244208     1.244615
1        0.001240     0.999803     1.000026
2        0.009959    20.438469    20.438390
3        0.009755    24.651444    24.651402
4        0.019807    28.957169    28.957059
..            ...          ...          ...
499      0.006666   136.036575   136.036418
500      0.006247   136.256153   136.255984
501      0.006413   130.476090   130.475729
502      0.006056   127.296898   127.296431
503      0.003344    93.874308    93.873757

[504 rows x 9 columns]
```

The above results show that the Greeks obtained from the closed-form formula and from the finite-difference approximation are almost identical. Their numerical differences are in three decimal places.

Problem 12

In this part, I match option data from two consecutive days, and then compute the option price for the maturity, the same strike price, and implied volatility, except for the underlying asset price and the risk-free interest rate.

In [ ]:
```
# Import Data
filename = f'C:/Users/18509/Desktop/Course/FE 621 A/Assignment 1/HW1_DATA1.xlsx'
TSLA_option_data2 = pd.read_excel(filename, sheet_name='TSLA_option')

# Match the option data in terms of strike price and maturity between the two datasets
data_temp = TSLA_option_data2[['K', 'maturity', 'S', 'r', 'T','mid']].copy()
data_temp = data_temp.rename(columns={'S': 'S2', 'r': 'r2', 'T': 'T2', 'mid': 'mid2'})
TSLA_option_data = TSLA_option_data.merge(data_temp, on=['K', 'maturity'], how='left')
```

In [ ]:
```
# Compute the predicted mid price using the implied volatility and strike from DATA 2
TSLA_option_data['mid2_pred'] = TSLA_option_data.apply(lambda x: price_BS(x['S2'], x['K'], x['T2'], x['r2'], x['IV_newton'], x
print(TSLA_option_data[['K', 'maturity', 'type', 'mid2', 'mid2_pred']])
```

```
C:\Users\18509\AppData\Local\Temp\ipykernel_15148\3363726036.py:3: RuntimeWarning: divide by zero encountered in scalar divide
  d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
```

	K	maturity	type	mid2	mid2_pred
0	322.5	2026-02-18	1	94.650	94.531054
1	322.5	2026-02-18	1	0.060	94.065125
2	322.5	2026-02-18	1	94.650	94.531054
3	322.5	2026-02-18	1	0.060	94.065125
4	322.5	2026-02-18	1	94.650	94.531054
...	...	...	...	...	...
5811	500.0	2026-05-15	0	93.925	90.152219
5812	500.0	2026-05-15	0	15.275	92.019394
5813	500.0	2026-05-15	0	93.925	90.152219
5814	500.0	2026-05-15	0	15.275	92.019394
5815	500.0	2026-05-15	0	93.925	90.152219

[5816 rows x 5 columns]

The above results show that for short maturities, even when using the implied volatility from consecutive days, the estimated price is typically close to the market price. However, for longer maturities, pricing discrepancies become more pronounced.

Part 3 Numerical Integration

Problem a

For the case one, due to the product of quantities must stay constant,

$$(y_t + (1-\gamma)\Delta_y)(x_t - \Delta_x) = k$$

The corresponding boundary condition is given by

$$\frac{y_t + (1-\gamma)\Delta_y}{x_t - \Delta_x} \cdot \frac{1}{1-\gamma} = S_{t+1}$$

For the case two, due to the product of quantities must stay constant,

$$(x_t + (1-\gamma)\Delta_x)(y_t - \Delta_y) = k$$

The corresponding boundary condition is given by

$$\frac{y_t - \Delta y}{x_t + (1-\gamma)\Delta x} \cdot (1-\gamma) = S_{t+1}$$

The swap size for the case one is,

$$\Delta x_{S_{t+1} > \frac{x_t}{1-\gamma}} = x_t - \sqrt{\frac{k}{(1-\gamma)S_{t+1}}}$$

$$\Delta y_{S_{t+1} > \frac{x_t}{1-\gamma}} = \frac{\sqrt{(1-\gamma)S_{t+1}k} - y_t}{1-\gamma}$$

The swap size for the case two is,

$$\Delta x_{S_{t+1} < \frac{x_t}{1-\gamma}} = \frac{\sqrt{\frac{k(1-\gamma)}{S_{t+1}}} - x_t}{1-\gamma}$$

$$\Delta y_{S_{t+1} < \frac{x_t}{1-\gamma}} = y_t - \sqrt{\frac{S_{t+1}k}{1-\gamma}}$$

Note that the square root in the operation should ensure that $\Delta x > 0, \Delta y > 0$.

Problem b

In this part, I sample the terminal price from the GBM model and then substitute it into the corresponding integrand over the price range.

There are two approaches for this problem:

- (1) Derive the density of the terminal price, and then substitute it into the integral, which is subsequently computed by the trapezoidal rule.

$$\mathbb{E}[R(S_{t+1})] = \int_{F_t/(1-\gamma)}^{\infty} \gamma \Delta_y(S_{t+1}; x_t, y_t, \gamma, k)\, f_{S_{t+1}}(s)\, ds + \int_0^{F_t(1-\gamma)} \gamma \Delta_x(S_{t+1}; x_t, y_t, \gamma, k)\, S_{t+1}\, f_{S_{t+1}}(s)\, ds$$

where

$$f_{S_{t+1}}(s) = \frac{1}{s\,\sigma\sqrt{2\pi\Delta t}} \exp\left(-\frac{\left(\ln\frac{s}{S_t} + \frac{1}{2}\sigma^2\Delta t\right)^2}{2\sigma^2\Delta t} \right), \qquad s > 0$$

The general form of the integrand under the trapezoidal rule can be reformulated as

$$\int_\alpha^\beta g(s)\,ds \;\approx\; h\left[\frac{1}{2}g(s_0) + \sum_{i=1}^{n-1} g(s_i) + \frac{1}{2}g(s_n)\right]$$

where $s_i = \alpha + ih$, $\quad h = \frac{\beta-\alpha}{n}$

- (2) Note that the truncated integration form can be rewritten as the numerical mean value, which can be approximated by the Monte Carlo method.

$$\mathbb{E}[R(S_{t+1})] = \int_{F_t/(1-\gamma)}^{\infty} \gamma \Delta_y(S_{t+1}; x_t, y_t, \gamma, k)\, f_{S_{t+1}}(s)\, ds + \int_0^{F_t(1-\gamma)} \gamma \Delta_x(S_{t+1}; x_t, y_t, \gamma, k)\, S_{t+1}\, f_{S_{t+1}}(s)\, ds$$

$$= \mathbb{E}\left[\gamma \Delta_y(S_{t+1})\, \mathbf{1}_{\{S_{t+1} \geq \frac{F_t}{1-\gamma}\}}\right] + \mathbb{E}\left[\gamma \Delta_x(S_{t+1})\, S_{t+1}\, \mathbf{1}_{\{S_{t+1} \leq F_t(1-\gamma)\}}\right]$$

where:

$$S_{t+1} = S_t \exp\left(-\frac{1}{2}\sigma^2\Delta t + \sigma\sqrt{\Delta t}\, Z\right)$$

$$Z \sim N(0,1)$$

$$\Delta y_{S_{t+1}>\frac{P_t}{1-\gamma}} = \frac{\sqrt{(1-\gamma)S_{t+1}k} - x_t}{1-\gamma}$$

$$\Delta x_{S_{t+1}<\frac{P_t}{1-\gamma}} = \frac{\sqrt{\frac{k(1-\gamma)}{S_{t+1}}} - x_t}{1-\gamma}$$

In [1]:
```
import pandas as pd
from pandas_datareader import data as web
import numpy as np
import openpyxl
import os
from datetime import datetime, timedelta
import scipy as sp
from scipy.stats import norm
import time
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

In [40]:
```
# Define the initial parameters
sigma = 0.2
gamma = 0.003
dt = 1/365
St = 1
xt = 1000
yt = 1000
Pt = yt/xt
k = xt*yt

# First approach
N = 10000 # The number of intervals for numerical integration
# second integral
a = 0.001
b = Pt*(1 - gamma)
ST = np.linspace(a, b, N+1)
pdf = (1 / (ST * sigma * np.sqrt(2 * np.pi * dt))) * np.exp(-((np.log(ST / St) + 0.5 * sigma**2 * dt)**2) / (2 * sigma**2 * dt
dx = (np.sqrt((1 - gamma)*k/ST) - xt)/(1 - gamma)
temp = gamma*dx*pdf*ST
```

```
E2 = (b-a)/N * (0.5 * temp[0] + 0.5 * temp[-1] + np.sum(temp[1:-1]))

# first integral
a = Pt/(1 - gamma)
b = a*20
ST = np.linspace(a, b, N+1)
pdf = (1 / (ST * sigma * np.sqrt(2 * np.pi * dt))) * np.exp(-((np.log(ST / St) + 0.5 * sigma**2 * dt)**2) / (2 * sigma**2 * dt
dy = (np.sqrt((1 - gamma)*ST*k) - yt)/(1 - gamma)
temp = gamma*dy*pdf
E1 = (b-a)/N * (0.5 * temp[0] + 0.5 * temp[-1] + np.sum(temp[1:-1]))

ER = E1 + E2
print(f"Expected one-step fee revenue under the trapezoidal rule: {ER:.4f}")

# Second approach
# Sample the price path of the underlying asset using Geometric Brownian Motion
M = 100000 # The number of simulations
ST = St*np.exp(( - 0.5 * sigma ** 2) * dt + sigma * np.sqrt(dt) * np.random.randn(M))

upper = Pt/(1-gamma)
lower = Pt*(1-gamma)

dy_all = (np.sqrt((1-gamma)*ST*k) - yt)/(1-gamma)
dx_all = (np.sqrt((1-gamma)*k/ST) - xt)/(1-gamma)

payoff = np.zeros(M)

mask1 = ST > upper
payoff[mask1] = gamma * dy_all[mask1]

mask2 = ST < lower
payoff[mask2] = gamma * dx_all[mask2] * ST[mask2]

E_R = payoff.mean()
print(f"Expected one-step fee revenue under the Monte Carlo method: {E_R:.4f}")
```

```
Expected one-step fee revenue under the trapezoidal rule: 0.0085
Expected one-step fee revenue under the Monte Carlo method: 0.0085
```

The values of the two approaches are identical.

Problem c

In [2]:
```
# Define a one-step fee revenue funtion using the trapezoidal rule for optimal Fee Rate under different volatilities
def fee_revenue(gamma, sigma, dt, St, xt, yt, N):
    k = xt*yt
    # second integral
    a = 0.001
    b = yt/xt*(1 - gamma)
    ST = np.linspace(a, b, N+1)
    pdf = (1 / (ST * sigma * np.sqrt(2 * np.pi * dt))) * np.exp(-((np.log(ST / St) + 0.5 * sigma**2 * dt)**2) / (2 * sigma**2
    dx = (np.sqrt((1 - gamma)*k/ST) - xt)/(1 - gamma)
    temp = gamma*dx*pdf*ST
    E2 = (b-a)/N * (0.5 * temp[0] + 0.5 * temp[-1] + np.sum(temp[1:-1]))

    # first integral
    a = yt/xt/(1 - gamma)
    b = a*20
    ST = np.linspace(a, b, N+1)
    pdf = (1 / (ST * sigma * np.sqrt(2 * np.pi * dt))) * np.exp(-((np.log(ST / St) + 0.5 * sigma**2 * dt)**2) / (2 * sigma**2
    dy = (np.sqrt((1 - gamma)*ST*k) - yt)/(1 - gamma)
    temp = gamma*dy*pdf
    E1 = (b-a)/N * (0.5 * temp[0] + 0.5 * temp[-1] + np.sum(temp[1:-1]))

    ER = E1 + E2
    return ER
```

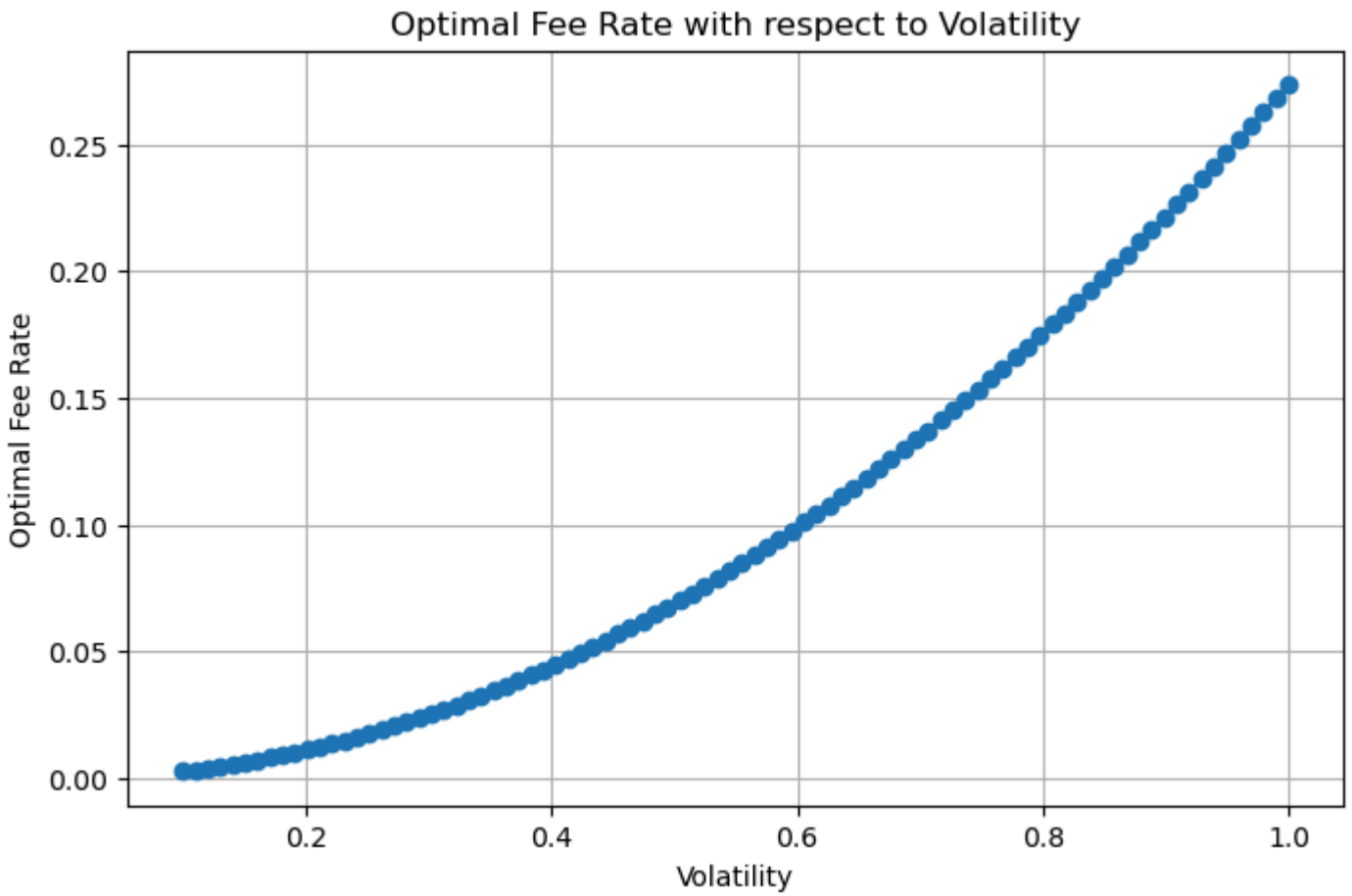
In [4]:
```
# Define the initial parameters
dt = 1/365
St = 1
xt = 1000
yt = 1000
Pt = yt/xt
k = xt*yt
N = 10000 # The number of intervals for numerical integration

sigma_all = [0.2,0.6,1.0]
gamma_all = [0.001,0.003,0.01]
ER = np.zeros((len(gamma_all), len(sigma_all)))
```

```
for i in range(len(sigma_all)):
    for j in range(len(gamma_all)):
        ER[j,i] = fee_revenue(gamma_all[j], sigma_all[i], dt, St, xt, yt, N)
print("Expected one-step fee revenue under different volatilities and fee rates:")
print(f'sigma: {sigma_all}')
print(ER)
```

```
Expected one-step fee revenue under different volatilities and fee rates:
sigma: [0.2, 0.6, 1.0]
[[0.00367949 0.01192146 0.02005957]
 [0.00850544 0.03297756 0.05738031]
 [0.00939396 0.08106406 0.16067856]]
```

In [5]:
```
# Define a denser grid of sigma and gamma for the optimal fee rate analysis
sigma_denser = np.linspace(0.1, 1.0, round((1-0.1)/0.01))
gamma_denser = np.linspace(0.001, 0.999, 1000)
ER_denser = np.zeros((len(gamma_denser), len(sigma_denser)))
for i in range(len(sigma_denser)):
    for j in range(len(gamma_denser)):
        ER_denser[j,i] = fee_revenue(gamma_denser[j], sigma_denser[i], dt, St, xt, yt, N)
ER_optimal = np.max(ER_denser, axis=0)
```

In [6]:
```
# Plot the curve of optimal fee rate with respect to volatility
plt.figure(figsize=(8,5))
plt.plot(sigma_denser, ER_optimal, marker='o')
plt.title('Optimal Fee Rate with respect to Volatility')
plt.xlabel('Volatility')
plt.ylabel('Optimal Fee Rate')
plt.grid()
```



The figure shows that as volatility increases, the optimal fee rate rises.

In [ ]: