In [ ]:
```
# all imports here

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

Part 1. (20 points) Data gathering component

1. Write a function (program) to connect to sources and download data from one of the following sources:

   (a) GOOGLE finance http://www.google.com/finance

   (b) Yahoo Finance http://finance.yahoo.com

   (c) Bloomberg

   Notes. For extra credit you can turn in code to download data from the other two sources. Please note that the program needs to download both option data and equity data. For this problem (and only for this problem) you may use any built in function or toolbox that will facilitate your work. The data will have to be clean (no duplicated values, only one exchange, every column labeled properly, in other words, consolidated).

Bonus (5 points) Create a program that is capable of downloading multiple assets, combine them with the associated time column, and save the data into a csv or excel file.

In [32]:
```
def download_data(tickers,start,end,target_name):
  all_data = pd.DataFrame()
  for ticker in tickers:
    data = yf.download(ticker,start,end)
    df = data["Close"].copy()
    all_data = pd.concat([all_data,df],axis=1)

  all_data.to_csv(f"{target_name}.csv")
  return all_data


def download_option_data(tickers,dataset_name):
  all_data_options = pd.DataFrame()
  current_prices = []

  for ticker in tickers:
    data = yf.Ticker(ticker)
```

```
    spot_price = data.fast_info['lastPrice']
    current_prices.append({'Ticker': ticker, 'Price': spot_price})
    available_dates = data.options
    target_dates = []

    for t in available_dates:
      date_obj = pd.to_datetime(t)

      if date_obj.dayofweek == 4 and 15 <= date_obj.day <= 21:
        target_dates.append(t)

      if len(target_dates) == 3:
        break

    print("target dates :",target_dates)

    for date in target_dates:
      calls = data.option_chain(date).calls
      calls['Type'] = 'Call'
      puts = data.option_chain(date).puts
      puts['Type'] = 'Put'

      daily_chain = pd.concat([calls,puts])
      daily_chain['Ticker'] = ticker
      daily_chain['Expiration'] = date
      daily_chain['Spot_Price'] = spot_price
      all_data_options = pd.concat([all_data_options,daily_chain])

  all_data_options.to_csv(f"{dataset_name}_option_1.csv",index=False)
  pd.DataFrame(current_prices).to_csv(f"{dataset_name}_equity_1.csv", index=False)
  return all_data_options
```

In [33]:
```
target_symbols = ['TSLA', 'SPY', '^VIX']
download_option_data(target_symbols,'DATA1')
```

```
target dates : ['2026-02-20', '2026-03-20', '2026-04-17']
target dates : ['2026-02-20', '2026-03-20', '2026-04-17']
target dates : []
```

Out[33]:

	contractSymbol	lastTradeDate	strike	lastPrice	bid	ask	change	perc
0	TSLA260220C00100000	2026-02-12 20:10:06+00:00	100.0	315.58	316.50	318.65	0.000000	
1	TSLA260220C00110000	2025-11-07 14:45:01+00:00	110.0	321.40	325.70	328.30	0.000000	
2	TSLA260220C00120000	2026-01-16 17:08:41+00:00	120.0	319.63	295.40	299.70	0.000000	
3	TSLA260220C00130000	2026-01-16 16:40:36+00:00	130.0	309.87	285.40	289.70	0.000000	
4	TSLA260220C00140000	2026-01-16 20:53:02+00:00	140.0	299.78	275.45	279.70	0.000000	
...	...	...	...	...	...	...	...	
154	SPY260417P00805000	2026-02-06 20:50:47+00:00	805.0	111.94	121.89	124.60	0.000000	
155	SPY260417P00810000	2026-01-29 16:20:40+00:00	810.0	121.76	126.89	129.61	0.000000	
156	SPY260417P00815000	2026-01-29 16:20:16+00:00	815.0	127.17	131.89	134.61	0.000000	
157	SPY260417P00820000	2026-02-13 21:14:44+00:00	820.0	138.20	136.89	139.61	-2.400009	
158	SPY260417P00825000	2026-01-29 16:20:33+00:00	825.0	136.91	141.89	144.61	0.000000	

2093 rows × 18 columns

In [ ]:
```
target_symbols = ['TSLA', 'SPY', '^VIX']
DATA2 =download_option_data(target_symbols,'DATA2')
```

Stored this data in data folder

In [23]:
```
DATA1 = pd.read_csv('data/DATA1_option.csv')
DATA2 = pd.read_csv('data/DATA2_option.csv')
```

In [24]:
```
DATA1.head()
```

Out[24]:

	contractSymbol	lastTradeDate	strike	lastPrice	bid	ask	change	percen
0	TSLA260220C00100000	2026-02-12 20:10:06+00:00	100.0	315.58	315.70	319.95	-4.910004	
1	TSLA260220C00110000	2025-11-07 14:45:01+00:00	110.0	321.40	325.70	328.30	0.000000	
2	TSLA260220C00120000	2026-01-16 17:08:41+00:00	120.0	319.63	295.55	300.00	0.000000	
3	TSLA260220C00130000	2026-01-16 16:40:36+00:00	130.0	309.87	284.70	290.00	0.000000	
4	TSLA260220C00140000	2026-01-16 20:53:02+00:00	140.0	299.78	274.65	280.00	0.000000	

3

Write a paragraph describing the symbols you are downloading data for. Explain what is SPY and its purpose. (Hint: look up the definition of an ETF). Explain what is ˆVIX and its purpose. Understand the options symbols. Understand when each option expires. Write this information and turn it in.

We are downloading data from yahoo finance library for 3 assets, Tesla, SPY and ^VIX.

- SPY (SPDR S&P 500 ETF Trust) is an Exchange Traded Fund (ETF) that tracks the performance of the S&P 500 Index.
- ^VIX is the CBOE Volatility Index, which measures the market's expectation of future volatility for the next 30 days. It is calculated real time using the prices of S&P 500 out of the money call and put options. When market is calm VIX usually trends downwards. When there is panic or uncertainity in the market VIX usually trends upwards. Because investors are scared and rush to buy protecitive put options.
- TSLA (Tesla, Inc.) is an electric vehicle manufacturer and energy company and is a leading innovator in electric vehicle technology.

Options Symbols (TSLA260220C00100000)

1. Root symbol : this is usually 1-6 letters long and is the ticker symbol of the underlying asset. In this case it is TSLA
2. Expiry Date : this is usually 2-4 letters long and is the expiry date of the option. In this case it is 260220
3. Option Type : this is usually 1 letter long and is the option type. In this case it is C
4. Strike Price : this is usually 1-6 digits long and is the strike price of the option. (Multiplied by 1000) In this case it is 100.000

Standard Equity and Index Options expire on the third Friday of the month. These are historically most liquid options. Modern needs of hedging and protection has introduced (0TDE) i.e zero days to expiry options to accommodate the needs of more sophisicated users.

4

4. The following items will also need to be recorded:
   - The underlying equity, ETF, or index price at the exact moment when the rest of the data is downloaded.
   - The short-term interest rate which may be obtained here: http://www.federalreserve.gov/releases/H15/Current/. There are a lot of rates posted on the site - they are all yearly, *they are in percents and need to be converted to numbers*. There is no theoretical recommendation on which to use, I used to use 3-months Treasury bills which are not available anymore. Since then I have been using the "Federal funds (effective)" rate but you can go ahead and try others. You should remember to be consistent in your choice. Also, make sure that the interest rate that you use is for the same day when the data you use for the implied volatility was gathered and note that the data is typically quoted in percents (you will need numbers). The same site has a link to past (historical) interest rates.
   - Time to Maturity.

In [28]:
```
DATA1_equity = pd.read_csv('data/DATA1_equity.csv')
DATA2_equity = pd.read_csv('data/DATA2_equity.csv')
print(DATA1_equity.head())
print(DATA2_equity.head())
```

```
  Symbol        Date   Price         Source
0   TSLA  2026-02-12  417.07  Yahoo Finance
1    SPY  2026-02-12  681.27  Yahoo Finance
2   ^VIX  2026-02-12   20.82  Yahoo Finance
  Symbol        Date   Price         Source
0   TSLA  2026-02-13  417.44  Yahoo Finance
1    SPY  2026-02-13  681.75  Yahoo Finance
2   ^VIX  2026-02-13   20.60  Yahoo Finance
```

I am planning to use 3.64% as my short term interest rate (Federal Funds Effective Rate) based on the website https://www.federalreserve.gov/releases/h15/#fn3 which was last refreshed on 13th Feb 2026.

Time to Maturity

In [38]:
```
target_dates = ['2026-02-20', '2026-03-20', '2026-04-17']

# time difference from 12th feb 2026
data1_ttm = (pd.to_datetime(target_dates) - pd.to_datetime('2026-02-12')).days
```

```
#time difference from 13th feb 2026
data2_ttm = (pd.to_datetime(target_dates) - pd.to_datetime('2026-02-13')).days

#annualized ttm
DATA1_ttm = data1_ttm / 365
DATA2_ttm = data2_ttm / 365

print(DATA1_ttm)
print(DATA2_ttm)
```

```
Index([0.021917808219178082, 0.09863013698630137, 0.17534246575342466], dtype='float
64')
Index([0.019178082191780823, 0.0958904109589041, 0.1726027397260274], dtype='float6
4')
```

Part 2

5. Using your choice of computer programming language implement the Black-Scholes formulas as a function of current stock price S_0, volatility σ, time to expiration $T - t$ (in years), strike price K and short-term interest rate r (annual). Please note that no toolbox function is allowed but you may call the normal CDF function (e.g., **pnorm** in **R** or **scipy.stats.norm.cdf** in **Python**).

Call Option Price (C):

$$C = S_0 N(d_1) - K e^{-r(T-t)} N(d_2)$$

Put Option Price (P):

$$P = K e^{-r(T-t)} N(-d_2) - S_0 N(-d_1)$$

Where d_1 and d_2 are calculated as:

$$d_1 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}}$$

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

In [74]:
```
def black_scholes(S0, sigma, T , K , r , option_type="call"):

# we shall use the black_scholes  closed formula to calculate the price of the opti

    d1 = (np.log(S0/K) + (r + 0.5 * sigma **2 )* T ) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    if option_type.lower() == "call":
        price = S0 * norm.cdf(d1) - K*np.exp(-r * T) * norm.cdf(d2)
    elif option_type.lower() == 'put':
        price = K * np.exp(-r * T) * norm.cdf(-d2) - S0 * norm.cdf(-d1)
```

```
    else:
        raise ValueError("Invalid option_type. Please use 'call' or 'put'.")
    return price
```

6. Implement the Bisection method to find the root of arbitrary functions. Apply this method to calculate the implied volatility on the first day you downloaded (DATA1). For this purpose use as the option value the average of bid and ask price if they both exist (and if their corresponding volume is nonzero). Also use a tolerance level of 10^{-6}. Report the implied volatility at the money (for the option with strike price closest to the traded stock price). You need to do it for both the stock and the

ETF data you have (you do not need to do this for ^VIX). Then average all the implied volatilities for the options between in-the-money and out-of-the-money.
Note. There is no clearly defined boundary between options at-the-money and out-of-the-money or in-the-money options. If we define moneyness as the ratio between S_0 the stock price today and K the strike price of the option some people use values of moneyness between 0.95 and 1.05 to define the options at the money. Yet, other authors use between 0.9 and 1.1. Use these guidelines if you wish to determine which options' implied volatilities should be averaged.

used to find root of a continuous function

In [92]:
```
def bisection_method(func , a ,  b , threshold=1e-6):
    c = a
    while(b-a >= threshold):
        c = (a + b )/2
        if( func(c)==0):
            break
        elif( func(a) * func(c) < 0):
            b = c
        else:
            a = c
    return c


def test_function(x):
    return x**2 - 17

print(bisection_method(test_function , a = 3 , b = 5))
```

```
4.123105049133301
```

Volatility Calculation

we will make a function :

$$f(\sigma) = BS(\sigma) - \text{ actual market price}$$

and we will try to find the root of this function, the σ corresponding to this would be our implied volatility

In [59]:
```
#data1 contains options data for both tesla and spy , we will seggregate it
data_options_1 = DATA1.copy()
print(data_options_1.isnull().sum())
data_options_1 = data_options_1[data_options_1.notna().all(axis=1)]
```

```
contractSymbol               1
lastTradeDate                1
strike                       1
lastPrice                    1
bid                          1
ask                          1
change                       1
percentChange                1
volume                      69
openInterest                 1
impliedVolatility            1
inTheMoney                   1
contractSize                 1
currency                     1
Underlying_Symbol            0
Underlying_Price_Snapshot    0
Expiration                   1
Download_Date                0
Type                         0
dtype: int64
```

In [60]:
```
print(data_options_1.isnull().sum())
```

```
contractSymbol               0
lastTradeDate                0
strike                       0
lastPrice                    0
bid                          0
ask                          0
change                       0
percentChange                0
volume                       0
openInterest                 0
impliedVolatility            0
inTheMoney                   0
contractSize                 0
currency                     0
Underlying_Symbol            0
Underlying_Price_Snapshot    0
Expiration                   0
Download_Date                0
Type                         0
dtype: int64
```

In [61]:
```
data_options_2 = DATA2.copy()
print(data_options_2.isnull().sum())
data_options_2 = data_options_2[data_options_2.notna().all(axis=1)]
```

```
contractSymbol                1
lastTradeDate                 1
strike                        1
lastPrice                     1
bid                           1
ask                           1
change                        1
percentChange                 1
volume                       64
openInterest                  1
impliedVolatility             1
inTheMoney                    1
contractSize                  1
currency                      1
Underlying_Symbol             0
Underlying_Price_Snapshot     0
Expiration                    1
Download_Date                 0
Type                          0
dtype: int64
```

In [62]:
```
print(data_options_2.isnull().sum())
```

```
contractSymbol                0
lastTradeDate                 0
strike                        0
lastPrice                     0
bid                           0
ask                           0
change                        0
percentChange                 0
volume                        0
openInterest                  0
impliedVolatility             0
inTheMoney                    0
contractSize                  0
currency                      0
Underlying_Symbol             0
Underlying_Price_Snapshot     0
Expiration                    0
Download_Date                 0
Type                          0
dtype: int64
```

In [80]:
```
data_options_1["target"] = (data_options_1.bid + data_options_1.ask) / 2
data_options_2["target"] = (data_options_2.bid + data_options_2.ask) / 2

data_options_1.columns
```

Out[80]:
```
Index(['contractSymbol', 'lastTradeDate', 'strike', 'lastPrice', 'bid', 'ask',
       'change', 'percentChange', 'volume', 'openInterest',
       'impliedVolatility', 'inTheMoney', 'contractSize', 'currency',
       'Underlying_Symbol', 'Underlying_Price_Snapshot', 'Expiration',
       'Download_Date', 'Type', 'target', 'ttm', 'implied_volatility_calc'],
      dtype='object')
```

In [77]:
```
data_equity_1 = pd.read_csv("data/DATA1_equity.csv")
data_equity_1.head()
```

Out[77]:

	Symbol	Date	Price	Source
0	TSLA	2026-02-12	417.07	Yahoo Finance
1	SPY	2026-02-12	681.27	Yahoo Finance
2	^VIX	2026-02-12	20.82	Yahoo Finance

Day 1

In [82]:
```
r = 0.0364
target_date = pd.to_datetime('2026-02-12')

# Create TTM Column (Time to Maturity in Years)
data_options_1['ttm'] = (pd.to_datetime(data_options_1['Expiration']) - target_date

# Applying the Bisection Method
# findinf sigma where (BlackScholes(sigma) - MarketPrice) == 0
data_options_1['implied_volatility_calc'] = data_options_1.apply(
    lambda row: bisection_method(
        func=lambda sigma: black_scholes(
            S0=row['Underlying_Price_Snapshot'],
            sigma=sigma,
            T=row['ttm'],
            K=row['strike'],
            r=r,
            option_type=row['Type']
        ) - row['target'],        # subtracting  market price of option
        a=0.01,                  # Lower bound (1% vol)
        b=5.0                    # Upper bound (500% vol)
    ),
    axis=1
)

# Display
print(data_options_1[['contractSymbol', 'Type', 'strike', 'target', 'implied_volati
```

```
        contractSymbol  Type  strike   target  implied_volatility_calc  \
0  TSLA260220C00100000  Call   100.0  317.825                 4.438942
1  TSLA260220C00110000  Call   110.0  327.000                 4.999999
2  TSLA260220C00120000  Call   120.0  297.775                 3.835905
3  TSLA260220C00130000  Call   130.0  287.350                 3.085490
4  TSLA260220C00140000  Call   140.0  277.325                 2.832727

   impliedVolatility
0           4.507817
1           8.879399
2           3.913086
3           3.253908
4           3.021487
```

Moneyness and IV ATM

In [83]:
```
data_options_1['Moneyness'] = data_options_1['Underlying_Price_Snapshot'] / data_op
tsla_data = data_options_1[data_options_1['Underlying_Symbol'] == 'TSLA'].copy()
tsla_data['distance_ATM'] = abs(tsla_data['Moneyness'] - 1.0)

tesla_atm_row = tsla_data.loc[tsla_data['distance_ATM'].idxmin()]
print(tesla_atm_row)
```

```
contractSymbol                    TSLA260220C00417500
lastTradeDate               2026-02-12 20:59:58+00:00
strike                                        417.5
lastPrice                                      10.4
bid                                            10.3
ask                                           10.35
change                                         -6.9
percentChange                             -39.88439
volume                                       3347.0
openInterest                                 1299.0
impliedVolatility                          0.428656
inTheMoney                                    False
contractSize                                REGULAR
currency                                        USD
Underlying_Symbol                              TSLA
Underlying_Price_Snapshot                417.070007
Expiration                                2026-02-20
Download_Date               2026-02-12 20:52:55
Type                                          Call
target                                       10.325
ttm                                        0.021918
implied_volatility_calc                    0.421139
Moneyness                                   0.99897
distance_ATM                                0.00103
Name: 65, dtype: object
```

In [85]:
```
data_options_1['Moneyness'] = data_options_1['Underlying_Price_Snapshot'] / data_op
spy_data = data_options_1[data_options_1['Underlying_Symbol'] == 'SPY'].copy()
spy_data['distance_ATM'] = abs(spy_data['Moneyness'] - 1.0)

spy_atm_row = spy_data.loc[spy_data['distance_ATM'].idxmin()]
print(spy_atm_row)
```

```
contractSymbol                SPY260220C00681000
lastTradeDate          2026-02-12 21:14:24+00:00
strike                                     681.0
lastPrice                                    8.2
bid                                          8.2
ask                                         8.23
change                                     -6.37
percentChange                         -43.719975
volume                                     1389.0
openInterest                               1421.0
impliedVolatility                         0.20118
inTheMoney                                   True
contractSize                              REGULAR
currency                                      USD
Underlying_Symbol                             SPY
Underlying_Price_Snapshot                681.27002
Expiration                              2026-02-20
Download_Date                2026-02-12 20:52:55
Type                                        Call
target                                     8.215
ttm                                     0.021918
implied_volatility_calc                 0.194015
Moneyness                               1.000397
distance_ATM                            0.000397
Name: 1069, dtype: object
```

Average Implied Volatility ITM and OTM

In [87]:
```
data_near_money = data_options_1[
    (data_options_1["Moneyness"] >= 0.95) &
    (data_options_1["Moneyness"] < 1.05) &
    (data_options_1["implied_volatility_calc"].notna()) &
    (data_options_1["implied_volatility_calc"] < 5)
]

spy_avg_iv = data_near_money[data_near_money['Underlying_Symbol']=="SPY"]["implied_
tsla_avg_iv = data_near_money[data_near_money['Underlying_Symbol']=="TSLA"]["implie

print(f"SPY Average IV:  {spy_avg_iv:.4f}")
print(f"TSLA Average IV: {tsla_avg_iv:.4f}")
```

```
SPY Average IV:  0.1876
TSLA Average IV: 0.4379
```

7. Implement the Newton method/Secant method or Muller method to find the root of arbitrary functions. You will need to discover the formula for the option's derivative with respect to the volatility σ. Apply these methods to the same options as in the previous problem. Compare the time it takes to get the root with the same level of accuracy.

Newton Raphson's Method

$$\sigma_{new} = \sigma_{old} - \frac{f(\sigma_{old})}{f'(\sigma_{old})}$$

- f will calculate the current BS(sigma) price minus the market price
- f' will calculate the vega from initial guess of sigma
- we will stop the condition when difference between the sigmas are below the threshold

In [ ]:
```
# we can get closed form vega from any reference textbook or we can find it ourself
def calc_vega(S0, K, T, r, sigma):
    d1 = (np.log(S0 / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    vega = S0 * np.sqrt(T) * norm.pdf(d1)
    return vega


def newton_raphson(row,threshold = 1e-6):

    S0 = row['Underlying_Price_Snapshot']
    K = row['strike']
    T = row['ttm']
    r = 0.0364
    target_price = row['target']
    option_type = row['Type']

    max_iter = 100
    sigma = 0.8

    for i in range(max_iter):

        current_price = black_scholes(S0=S0,sigma=sigma, T=T , K=K , r=r, option_ty
        deltaPrice = current_price - target_price
        if abs(deltaPrice )< threshold:
            return sigma
        vega = calc_vega(S0,K,T,r,sigma)
        if vega < 1e-10:
            return np.nan
        sigma = sigma - deltaPrice/vega
        if sigma <= 0:
            sigma = 0.00001

    return np.nan


print("bisection method time")
start_bisection = time.time()
data_options_1["iv_bisection"] = data_options_1.apply(
    lambda row: bisection_method(
        func=lambda sigma: black_scholes(
            S0=row['Underlying_Price_Snapshot'],
            sigma=sigma,
            T=row['ttm'],
            K=row['strike'],
            r=r,
            option_type=row['Type']
        ) - row['target'],
        a=0.01,
        b=5.0
    ),
    axis=1
```

```
)
end_bisection = time.time()


print("newton method time")
start_newton = time.time()
data_options_1["iv_newton"] = data_options_1.apply(lambda row : newton_raphson(row)
end_newton = time.time()

print(f"Bisection Method took: {end_bisection - start_bisection:.4f} seconds")
print(f"Newton's Method took:  {end_newton - start_newton:.4f} seconds")

print(data_options_1[['strike', 'Type', 'Underlying_Price_Snapshot' ,'target', 'iv_
```

```
bisection method time
newton method time
Bisection Method took: 7.7617 seconds
Newton's Method took:  0.8498 seconds
   strike  Type  Underlying_Price_Snapshot   target  iv_newton  iv_bisection
0   100.0  Call                417.070007  317.825        NaN      4.438942
1   110.0  Call                417.070007  327.000        NaN      4.999999
2   120.0  Call                417.070007  297.775        NaN      3.835905
3   130.0  Call                417.070007  287.350        NaN      3.085490
4   140.0  Call                417.070007  277.325        NaN      2.832727
```

Having a nan value for the iv_newton is not surprising result, because the options are deep ITM therefore vega is technically very very small, if we check moneyness values, they should give us some insight. But there us one very crucial observation, the time it took for newton's method is 10 times faster. than the bisection method but has issues calculating the iv's at deeper ITM and OTM. THerefore a hybrid approach is used in real.

In [105…

```
data_options_1['Moneyness'] = data_options_1['Underlying_Price_Snapshot'] / data_op
tsla_data = data_options_1[data_options_1['Underlying_Symbol'] == 'TSLA'].copy()
tsla_data['distance_ATM'] = abs(tsla_data['Moneyness'] - 1.0)

tesla_atm_row = tsla_data.loc[tsla_data['distance_ATM'].idxmin()]
print(tesla_atm_row[['iv_newton' , 'iv_bisection' , 'impliedVolatility' , 'strike'


data_options_1['Moneyness'] = data_options_1['Underlying_Price_Snapshot'] / data_op
spy_data = data_options_1[data_options_1['Underlying_Symbol'] == 'SPY'].copy()
spy_data['distance_ATM'] = abs(spy_data['Moneyness'] - 1.0)

spy_atm_row = spy_data.loc[spy_data['distance_ATM'].idxmin()]
print(spy_atm_row[['iv_newton' , 'iv_bisection' , 'impliedVolatility' , 'strike' ,
```

```
iv_newton                    0.421139
iv_bisection                 0.421139
impliedVolatility            0.428656
strike                          417.5
Underlying_Price_Snapshot    417.070007
Name: 65, dtype: object
iv_newton                    0.194015
iv_bisection                 0.194015
impliedVolatility             0.20118
strike                          681.0
Underlying_Price_Snapshot    681.27002
Name: 1069, dtype: object
```

8. Present a table reporting the implied volatility values obtained for every maturity, option type and stock. Also compile the average volatilities as described in the previous point. Comment on the observed difference in values obtained for TSLA and SPY. Compare with the current value of the ^VIX. Comment on what happens when the maturity increases. Comment on what happen when the options become in the money respectively out of the money.

In [111…

```
data_near_money_volatility = data_options_1.copy()


data_near_money_volatility = data_near_money_volatility[
    (data_near_money_volatility["Moneyness"] >= 0.95) &
    (data_near_money_volatility["Moneyness"] < 1.05) &
    (data_near_money_volatility["impliedVolatility"].notna())
]

cols = ['iv_bisection' , 'iv_newton' , 'impliedVolatility' ]

iv_means_all = data_near_money_volatility.groupby('Underlying_Symbol')[cols].mean()
iv_means_all["VIX"] = data_equity_1[data_equity_1.Symbol == '^VIX']['Price'].iloc[0
print(iv_means_all)
```

```
                  iv_bisection  iv_newton  impliedVolatility     VIX
Underlying_Symbol
SPY                   0.187613   0.175642           0.180209  0.2082
TSLA                  0.437937   0.437937           0.439715  0.2082
```

SPY here is considered a safer less volatile asset because it constitutes the component of the market, the VIX implied volatility is forward looking measure of S&P 500 and matches nearly with the value we calculated from both bisection method, newton method and the given Implied volatility from Yahoo finance.

Comparison of Tesla and SPY

- Tesla is nearly double the vol of both vix and spy , reason being tesla is a concentrated stock and has high shocks subjected to sentiments and news. Whereas spy's high volatility is supressed by low correlation between constituent. This also proves that diversification reduces risk, also VIX is a little higher than current SPY IV becuase it also

considers broad OTM and ITM options, in our case we ignored those to calculate the avg of near the money IV's.

Impact of Maturity

- In calm markets, IV increases with maturity, this is because there is more time for higher / lower price movements than expected.
- If there is a major news coming then short term IV spikes than the long term IV
- Generally, for a stable stock, as maturity increases, the IV values across different strikes tend to "flatten" as the immediate noise is averaged out over time.

Impact of Moneyness

- OTM Puts : show the highest IV, this accounts for skew, investors pay high risk premium for crash protection
- ATM : Usually show the lowest IV. These are the most liquid and represent the market's current consensus.
- ITM : For calls, as they become deep ITM, the IV can sometimes rise or become erratic due to low liquidity and high capital requirements, often mirroring the high IV of the OTM puts via Put-Call Parity.

> 9. For each option in your table calculate the price of the different type (Call/Put) using the Put-Call parity (please see Section 4 from [2]). Compare the resulting values with the BID/ASK values for the corresponding option if they exist.

The formula for put call parity :

$$C - P = S_0 - K \cdot e^{-rT}$$

In [ ]:
```
calls = data_near_money_volatility[data_near_money_volatility['Type'] == 'Call'].co
puts = data_near_money_volatility[data_near_money_volatility['Type'] == 'Put'].copy

#rename columns
calls_subset = calls[['Underlying_Symbol', 'strike', 'Expiration', 'ttm', 'Underlyi
    columns={'target': 'call_market_price', 'bid': 'call_bid', 'ask': 'call_ask'}
)

puts_subset = puts[['Underlying_Symbol', 'strike', 'Expiration', 'target', 'bid', '
    columns={'target': 'put_market_price', 'bid': 'put_bid', 'ask': 'put_ask'}
)

# merge data
parity_df = pd.merge(
    calls_subset,
    puts_subset,
    on=['Underlying_Symbol', 'strike', 'Expiration']
)
```

```
def calculate_synthetic_prices(row):
    S0 = row['Underlying_Price_Snapshot']
    K = row['strike']
    T = row['ttm']
    r = 0.0364

    # PV of Strike: K * e^(-rT)
    pv_k = K * np.exp(-r * T)
    # 1. Synthetic Call = P + S0 - PV(K)
    synthetic_call = row['put_market_price'] + S0 - pv_k
    # 2. Synthetic Put = C - S0 + PV(K)
    synthetic_put = row['call_market_price'] - S0 + pv_k
    return pd.Series([synthetic_call, synthetic_put])

parity_df[['synthetic_call', 'synthetic_put']] = parity_df.apply(calculate_syntheti
parity_df.head()
```

Out[ ]:

	Underlying_Symbol	strike	Expiration	ttm	Underlying_Price_Snapshot	call_market_
0	TSLA	397.5	2026-02-20	0.021918	417.070007	2
1	TSLA	400.0	2026-02-20	0.021918	417.070007	2
2	TSLA	402.5	2026-02-20	0.021918	417.070007	1
3	TSLA	405.0	2026-02-20	0.021918	417.070007	1
4	TSLA	407.5	2026-02-20	0.021918	417.070007	1

In [121…

```
parity_df['parity_holds'] = (parity_df['synthetic_call'] >= parity_df['call_bid'])

# counting how many rows hold
print(parity_df[['Underlying_Symbol', 'strike', 'call_bid', 'synthetic_call', 'call
```

```
  Underlying_Symbol  strike  call_bid  synthetic_call  call_ask  parity_holds
0              TSLA   397.5     23.75       23.812010     23.90          True
1              TSLA   400.0     21.75       21.839003     21.95          True
2              TSLA   402.5     19.85       19.940997     20.05          True
3              TSLA   405.0     17.90       18.117991     18.20          True
4              TSLA   407.5     16.30       16.369985     16.45          True
```

In [ ]:

```
# Calculate overall percentage
parity_percent = parity_df['parity_holds'].mean() * 100

# Calculate percentage by Stock (TSLA vs SPY)
percentage_by_stock = parity_df.groupby('Underlying_Symbol')['parity_holds'].mean()

print(f"Overall Put-Call Parity holds for {parity_percent:.2f}% of options.")
print("\nPercentage by Stock:")
print(percentage_by_stock)
```

```
Overall Put-Call Parity holds for 32.61% of options.

Percentage by Stock:
Underlying_Symbol
SPY     27.918782
TSLA    60.606061
Name: parity_holds, dtype: float64
```

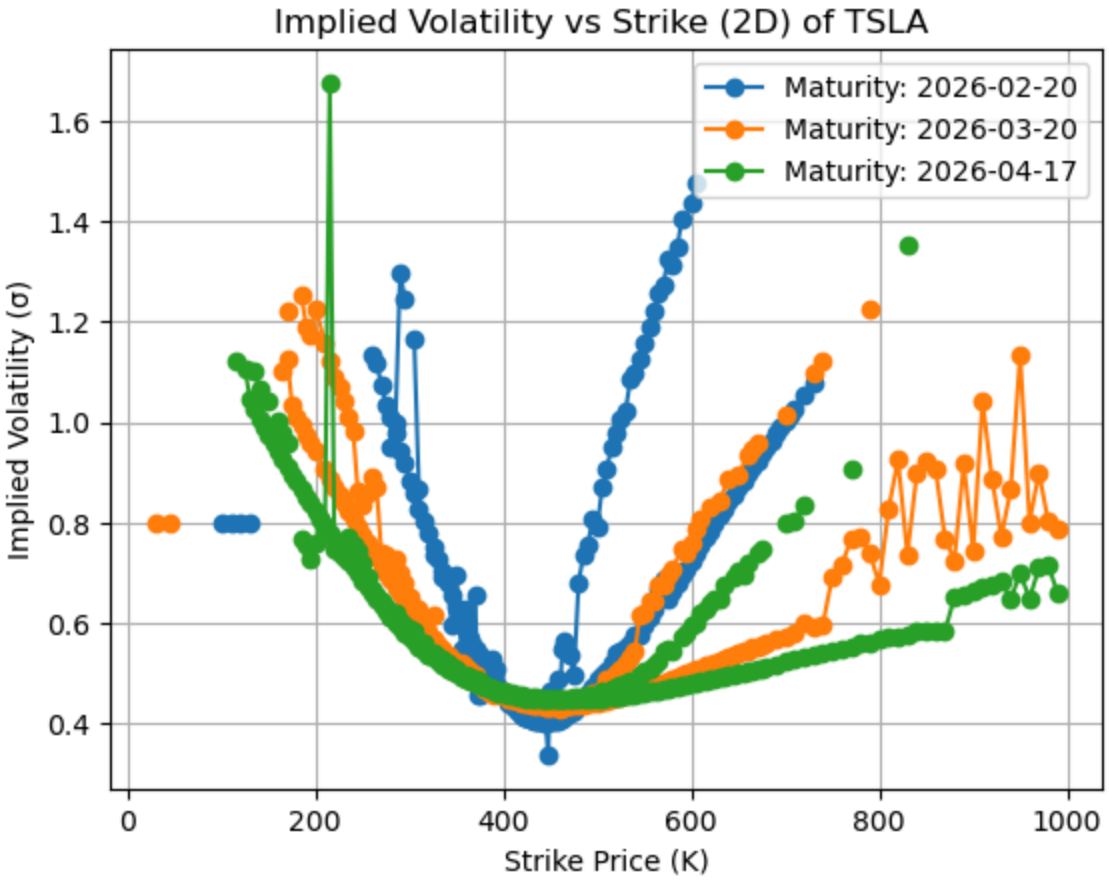
We can see from our obsevation that parity does not hold true even for the near the money option values. it does hold true for 60% of tesla data though, A higher percentage usually indicates that market is efficient. Also the variability in the percentage of S&P options because SPY pays dividends too , so we would need to change the formula to account for dividends for an accurate answer.

Due to high volatiltiy in tesla stock, Market Makers usually have higher bid ask spread, therefore the likelihood of option lying in between is also high, adding to the high parity hold percentage.

10. Consider the implied volatility values obtained in the previous parts. Create a 2 dimensional plot of implied volatilities versus strike K for the closest to maturity options. What do you observe? Plot all implied volatilities for the three different maturities on the same plot, where you use a different color for each maturity. In total there should be 3 sets of points plotted with different color. (**Bonus** 5 Points) Create a 3D plot of the same implied vols as a function of both maturity and strike, i.e.: $\sigma(\tau_i, K_j)$ where $i = 1, 2, 3$, and $j = 1, 2, \ldots, 20$.

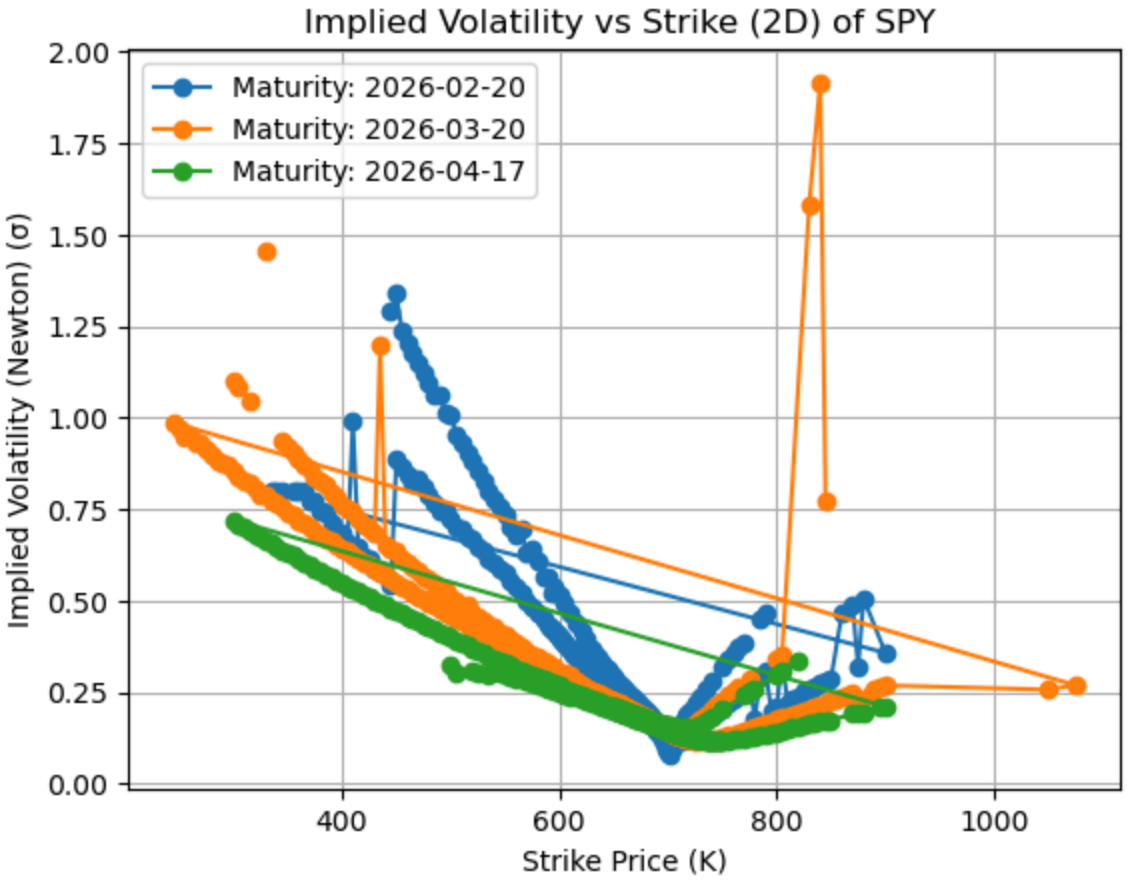
In [130…

```python
#plots

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

maturities = sorted(data_options_1['Expiration'].unique())

for maturity in maturities:
    subset = data_options_1[(data_options_1['Expiration'] == maturity) & (data_opti
    plt.plot(subset['strike'], subset['iv_newton'], marker='o', label=f'Maturity: {

plt.title('Implied Volatility vs Strike (2D) of TSLA ')
plt.xlabel('Strike Price (K)')
plt.ylabel('Implied Volatility (σ)')
plt.legend()
plt.grid(True)
```

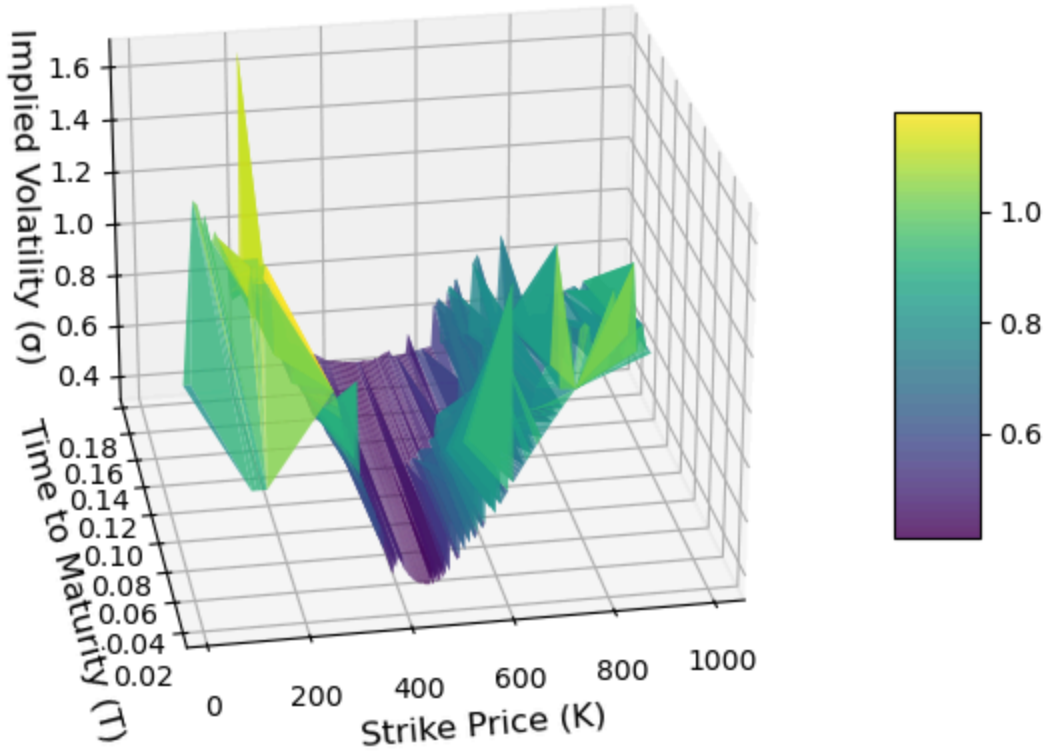


In [132…

```
#plots

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

maturities = sorted(data_options_1['Expiration'].unique())

for maturity in maturities:
    subset = data_options_1[(data_options_1['Expiration'] == maturity) & (data_opti
    plt.plot(subset['strike'], subset['iv_newton'], marker='o', label=f'Maturity: {

plt.title('Implied Volatility vs Strike (2D) of SPY ')
plt.xlabel('Strike Price (K)')
plt.ylabel('Implied Volatility (Newton) (σ)')
plt.legend()
plt.grid(True)
```

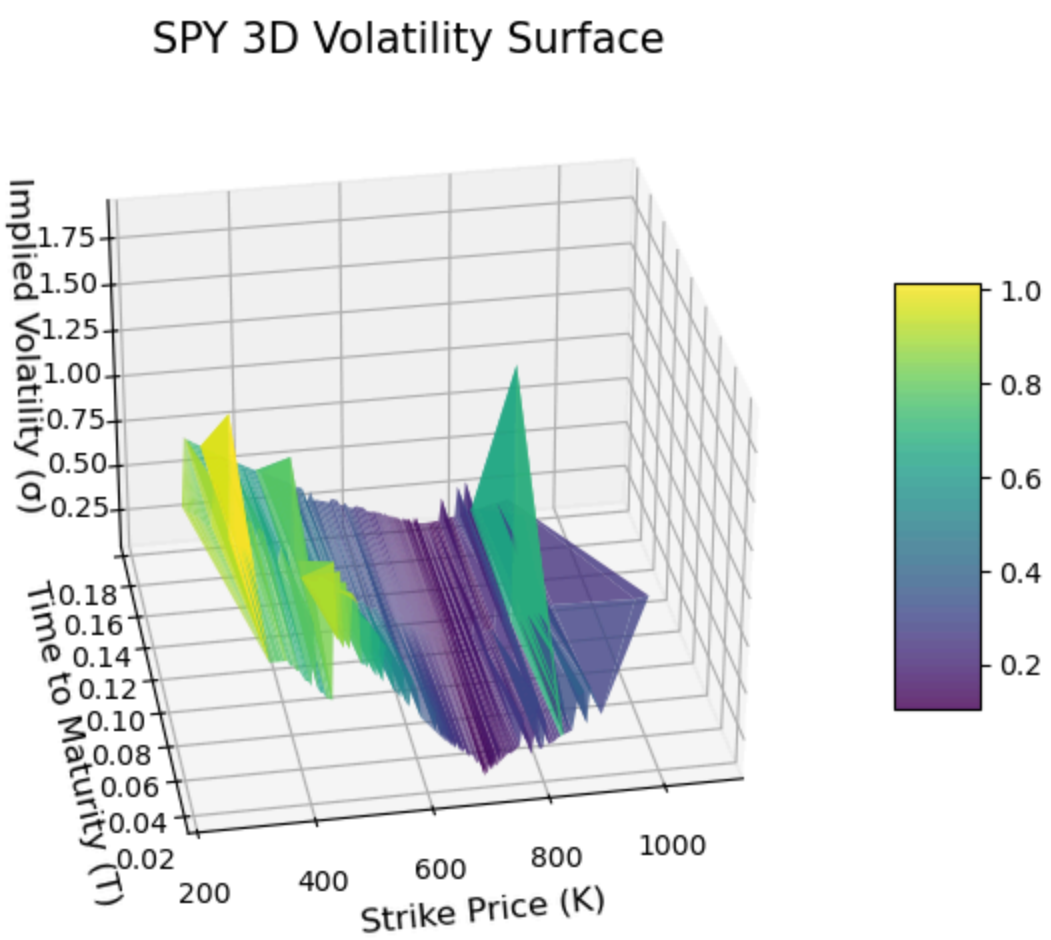


In [ ]:
```
#3d plot

tsla_data = data_options_1[data_options_1['Underlying_Symbol'] == 'TSLA'].copy()
plot_subset = tsla_data[tsla_data['Expiration'].isin(maturities)].copy()
plot_subset = plot_subset.dropna(subset=['iv_newton'])
figure = plt.figure()
ax = figure.add_subplot(111, projection='3d')

surf = ax.plot_trisurf(
    plot_subset['strike'],    # X-axis
    plot_subset['ttm'],       # Y-axis (Time to Maturity)
    plot_subset['iv_newton'], # Z-axis (Implied Volatility)
    cmap='viridis',           # Color theme
    edgecolor='none',
    alpha=0.8
)

ax.set_title('TSLA 3D Volatility Surface', fontsize=15)
ax.set_xlabel('Strike Price (K)', fontsize=12)
ax.set_ylabel('Time to Maturity (T)', fontsize=12)
ax.set_zlabel('Implied Volatility (σ)', fontsize=12)

figure.colorbar(surf, shrink=0.5, aspect=5)
ax.view_init(elev=25, azim=-100)
plt.tight_layout()
plt.show()
```



In [149…

```
#3d plot

tsla_data = data_options_1[data_options_1['Underlying_Symbol'] == 'SPY'].copy()
plot_subset = tsla_data[tsla_data['Expiration'].isin(maturities)].copy()
plot_subset = plot_subset.dropna(subset=['iv_newton'])
figure = plt.figure()
ax = figure.add_subplot(111, projection='3d')

surf = ax.plot_trisurf(
    plot_subset['strike'],    # X-axis
    plot_subset['ttm'],       # Y-axis (Time to Maturity)
    plot_subset['iv_newton'], # Z-axis (Implied Volatility)
    cmap='viridis',           # Color theme
    edgecolor='none',
    alpha=0.8
)

ax.set_title('SPY 3D Volatility Surface', fontsize=15)
ax.set_xlabel('Strike Price (K)', fontsize=12)
ax.set_ylabel('Time to Maturity (T)', fontsize=12)
ax.set_zlabel('Implied Volatility (σ)', fontsize=12)

figure.colorbar(surf, shrink=0.5, aspect=5)
ax.view_init(elev=30, azim=-100)
plt.tight_layout()
plt.show()
```



This is called a volatility surface. The lowest vol is ATM,

Observations:

The most obvious difference is the baseline height of the Z-axis (Implied Volatility). Looking at the deep purple "valleys" in both charts, SPY bottoms out around 0.15 to 0.20 (15-20% IV). TSLA, on the other hand, bottoms out much higher, around 0.40 to 0.60 (40-60% IV). This visually proves that TSLA is inherently far riskier and more volatile than the diversified S&P 500 index

Smile and Skew

TSLA displays a Volatility Smile. As we move away from the center (At-The-Money) toward both lower and higher strikes, the surface curves upwards. The market is pricing in the possibility of explosive moves in either direction.

SPY displays a Volatility Skew (or Smirk). The left side (low strikes / downside risk) shoots up aggressively to almost 1.75, while the right side (high strikes / upside risk) remains relatively flat.

11. (Greeks) Calculate the derivatives of the call option price with respect to S (Delta), and σ (Vega) and the second derivative with respect to S (Gamma). First use the Black Scholes formula then approximate these derivatives using an approximation of the partial derivatives. Compare the numbers obtained by the two methods. Output a table containing all derivatives thus calculated.

Delta (Δ): The change in option price for a $1 change in the stock. $\Delta = N(d_1)$ $Gamma$ (\Gamma): $The change in Delta for a 1$ change in the stock (the second derivative).

$$\Gamma = \frac{N'(d_1)}{S_0 \sigma \sqrt{T}}$$

Vega ($\mathcal{V}$): The change in option price for a 1% change in volatility.

$$\mathcal{V} = S_0 \sqrt{T} N'(d_1)$$

Instead of using calculus , we can use the finite difference method to calculate the greeks and check how the values compare to the calculated greeks

Approx Delta:

$$\frac{C(S+\epsilon) - C(S-\epsilon)}{2\epsilon}$$

Approx Gamma:

$$\frac{C(S+\epsilon) - 2C(S) + C(S-\epsilon)}{\epsilon^2}$$

Approx Vega:

$$\frac{C(\sigma+\epsilon) - C(\sigma-\epsilon)}{2\epsilon}$$

In [154…

```
def greeks_bs(row):
    S = row['Underlying_Price_Snapshot']
    K = row['strike']
    T = row['ttm']
    r = 0.0364
    sigma = row['iv_newton']

    if pd.isna(sigma):
        return pd.Series([np.nan, np.nan, np.nan, np.nan])

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

    delta = norm.cdf(d1)
```

```
    gamma = norm.pdf(d1) / (S * sigma * np.sqrt(T))
    vega = S * norm.pdf(d1) * np.sqrt(T)

    return pd.Series([delta, gamma, vega])


def greeks_finite_diff(row):
    S = row['Underlying_Price_Snapshot']
    K = row['strike']
    T = row['ttm']
    r = 0.0364
    sigma = row['iv_newton']

    if pd.isna(sigma) or T <= 0:
        return pd.Series([np.nan, np.nan, np.nan])

    ds = 0.001 * S
    dsigma = 0.001 * sigma

    call_base = black_scholes(S , sigma,  T , K , r , 'call')
    call_up = black_scholes(S + ds , sigma,  T , K , r , 'call')
    call_down = black_scholes(S - ds , sigma,  T , K , r , 'call')

    delta_approx = (call_up - call_down) / (2 * ds)
    gamma_approx = (call_up - 2 * call_base + call_down) / (ds ** 2)

    vega_up = black_scholes(S , sigma + dsigma,  T , K , r , 'call')
    vega_down = black_scholes(S , sigma - dsigma,  T , K , r , 'call')
    vega_approx = (vega_up - vega_down) / (2 * dsigma)

    return pd.Series([delta_approx, gamma_approx, vega_approx])



print("Calculating Greeks BS ")
calls[['Delta_BS', 'Gamma_BS', 'Vega_BS']] = calls.apply(greeks_bs, axis=1)

print("Calculating Greeks FD ")
calls[['Delta_FD', 'Gamma_FD', 'Vega_FD']] = calls.apply(greeks_finite_diff, axis=1


comparison_table = calls[[
    'Underlying_Symbol', 'strike',
    'Delta_BS', 'Delta_FD',
    'Gamma_BS', 'Gamma_FD',
    'Vega_BS', 'Vega_FD'
]]

comparison_table.head(20)
```

```
Calculating Greeks BS
Calculating Greeks FD
```

Out[154...

	Underlying_Symbol	strike	Delta_BS	Delta_FD	Gamma_BS	Gamma_FD	Vega_BS	V
57	TSLA	397.5	0.771567	0.771558	0.010534	0.010534	18.677268	18
58	TSLA	400.0	0.745957	0.745949	0.011329	0.011329	19.788159	19
59	TSLA	402.5	0.718010	0.718002	0.012107	0.012107	20.856336	20
60	TSLA	405.0	0.689080	0.689072	0.012930	0.012930	21.811628	21
61	TSLA	407.5	0.655848	0.655842	0.013548	0.013548	22.728590	22
62	TSLA	410.0	0.621737	0.621732	0.014155	0.014155	23.477043	23
63	TSLA	412.5	0.586115	0.586110	0.014686	0.014686	24.056880	24
64	TSLA	415.0	0.549105	0.549102	0.015133	0.015133	24.446173	24
65	TSLA	417.5	0.510947	0.510945	0.015336	0.015336	24.623742	24
66	TSLA	420.0	0.472219	0.472219	0.015467	0.015466	24.573267	24
67	TSLA	422.5	0.433420	0.433422	0.015408	0.015408	24.289171	24
68	TSLA	425.0	0.395570	0.395573	0.015124	0.015123	23.784157	23
69	TSLA	427.5	0.358012	0.358017	0.014762	0.014761	23.055877	23
70	TSLA	430.0	0.323203	0.323209	0.014165	0.014164	22.172580	22
71	TSLA	432.5	0.287608	0.287616	0.013587	0.013586	21.053586	21
72	TSLA	435.0	0.255984	0.255993	0.012814	0.012814	19.867126	19
73	TSLA	437.5	0.224913	0.224923	0.012011	0.012011	18.514296	18
399	TSLA	400.0	0.650917	0.650916	0.006178	0.006178	48.469388	48
400	TSLA	405.0	0.618973	0.618972	0.006422	0.006422	49.913278	49
401	TSLA	410.0	0.585941	0.585940	0.006617	0.006617	51.037295	51

In [155...

```
comparison_table_spy = comparison_table[comparison_table['Underlying_Symbol'] == 'S
comparison_table_spy.head(20)
```

Out[155…

	Underlying_Symbol	strike	Delta_BS	Delta_FD	Gamma_BS	Gamma_FD	Vega_BS
1037	SPY	649.0	0.877101	0.877080	0.006895	0.006895	20.517438
1038	SPY	650.0	0.871809	0.871787	0.007166	0.007166	21.130067
1039	SPY	651.0	0.867156	0.867133	0.007442	0.007442	21.657556
1040	SPY	652.0	0.861644	0.861620	0.007732	0.007733	22.269323
1041	SPY	653.0	0.857650	0.857626	0.008029	0.008029	22.703800
1042	SPY	654.0	0.851505	0.851480	0.008346	0.008346	23.358528
1043	SPY	655.0	0.845050	0.845024	0.008675	0.008675	24.028534
1044	SPY	656.0	0.838455	0.838429	0.009016	0.009016	24.694788
1045	SPY	657.0	0.832051	0.832024	0.009371	0.009371	25.324651
1046	SPY	658.0	0.824960	0.824932	0.009740	0.009740	26.002932
1047	SPY	659.0	0.817357	0.817329	0.010121	0.010121	26.708250
1048	SPY	660.0	0.808762	0.808733	0.010507	0.010507	27.479161
1049	SPY	661.0	0.801421	0.801391	0.010928	0.010928	28.115934
1050	SPY	662.0	0.792280	0.792250	0.011343	0.011342	28.881550
1051	SPY	663.0	0.783537	0.783507	0.011786	0.011786	29.586272
1052	SPY	664.0	0.773660	0.773630	0.012228	0.012228	30.350905
1053	SPY	665.0	0.762569	0.762539	0.012657	0.012656	31.170536
1054	SPY	666.0	0.752088	0.752058	0.013127	0.013127	31.908109
1055	SPY	667.0	0.740547	0.740517	0.013587	0.013586	32.679838
1056	SPY	668.0	0.728868	0.728838	0.014069	0.014069	33.418585

12. Next we will use the second dataset DATA2. For each strike price in the data use the Stock price for the same day, the implied volatility you calculated from DATA1 and the current short-term interest rate (corresponding to the day on which DATA2 was gathered). Use the Black-Scholes formula, to calculate the option price.

In [167…

```
#since we got few nan for newton iv method, we can use the bisection method to fill

data_options_1['hybrid_iv'] = data_options_1['iv_newton'].fillna(data_options_1['iv
iv_table = data_options_1[['Underlying_Symbol', 'strike', 'Expiration', 'Type', 'hy
target_date = pd.to_datetime('2026-02-13')
data_options_2['ttm'] = (pd.to_datetime(data_options_2['Expiration']) - target_date
#merge the iv data with data_options_2
data_options_2 = data_options_2.merge(iv_table, on=['Underlying_Symbol', 'strike',
```

```
r = 0.0364

#calculate the price of the option using the hybrid iv
def calculate_option_price(row):
    S0 = row['Underlying_Price_Snapshot']
    K = row['strike']
    T = row['ttm']
    opt_type = row['Type']
    sigma = row['hybrid_iv']

    if pd.isna(sigma):
        return np.nan

    return black_scholes(S0, sigma, T, K, r, opt_type)


data_options_2['option_price_theoretical'] = data_options_2.apply(calculate_option_
print(data_options_2[['strike', 'Type', 'target', 'option_price_theoretical']].head
```

```
   strike  Type   target  option_price_theoretical
0   100.0  Call  319.950                320.359709
1   110.0  Call  327.000                311.397137
2   120.0  Call  300.225                300.329834
3   130.0  Call  290.175                290.070056
4   140.0  Call  280.275                280.058511
5   150.0  Call  269.775                270.351161
6   155.0  Call  265.175                265.076155
7   160.0  Call  259.775                260.195002
8   165.0  Call  254.900                255.106900
9   170.0  Call  249.650                250.151886
```

Part 3

Part 3. (30 points) Numerical Integration of real-valued functions. AMM Arbitrage Fee Revenue

AMMs are decentralized exchanges that quote prices using pool reserves rather than an order book. Compared to Traditional Finance order books, a key advantage is **continuous liquidity from the pool without needing a matching counterparty**. Liquidity Providers (LPs) earn revenue mainly from **swap fees**, so selecting a good fee rate γ under different volatility levels matters [5].

For simplicity we consider the following Constant Product Market Maker (CPMM) for a **BTC/USDC** pool. These are the pool elements:

- x_t: BTC reserves at time t
- y_t: USDC reserves at time t
- pool mid price is calculated as $P_t = \frac{y_t}{x_t}$ (USDC per BTC)
- external market price S_t (USDC per BTC)
- fee rate $\gamma \in (0, 1)$

The pool satisfies the constant-product rule, that is at every moment in time the product of quantities must stay constant.

$$x_{t+1}y_{t+1} = x_t y_t = k.$$

Part 3

(a)

given

$x \cdot y = k$ (constant)

γ = our rate.

$$\begin{cases} x_t + (1-\gamma)\Delta x)(y_t - \Delta y) = k \\ x_{t+1} \cdot y_{t+1} = k \end{cases}$$

→ no arbitrage bound

$$S_{t+1} = \left[P_t(1-\gamma), \frac{P_t}{(1-\gamma)}\right]$$

Case 1: BTC shoots up in real world.

short BTC in real, buy BTC from AMM by giving USDC.

Boundary condition

$$P_{t+1} \times \frac{1}{1-\gamma} = \frac{y_{t+1}}{x_{t+1}} \cdot \frac{1}{1-\gamma} = S_{t+1}.$$

Basically the outside price should come in the boundary.

corresponding fee $= \gamma \cdot \Delta y$

(We are dumping USDC)

↓

Δy

New pool after the trade

$$x_{t+1} = x_t - \Delta x$$

$$y_{t+1} = y_t + (1-\gamma)\Delta y$$

Boundary.

$$S_{t+1} = \frac{y_{t+1}}{x_{t+1}} \cdot \frac{1}{1-\gamma}$$

$$y_{t+1} = (x_{t+1})(1-\gamma) \cdot S_{t+1}$$

we know that $x_{t+1} \cdot y_{t+1} = k$

$$x_{t+1} \cdot (x_{t+1}) \cdot (1-\gamma) \cdot S_{t+1} = k$$

$$x^2_{t+1} = \frac{k}{(1-\gamma) \cdot S_{t+1}} \Rightarrow x_{t+1} = \sqrt{\frac{k}{(1-\gamma) \cdot S_{t+1}}}$$

$$\therefore y_{t+1} = \frac{k}{x_{t+1}} = \sqrt{k \cdot (1-\gamma) \cdot s_{t+1}}$$

Since we know that

$$y_{t+1} = y_t + (1-\gamma) \cdot \Delta y$$

$$\Delta y = \frac{y_{t+1} - y_t}{1-\gamma}$$

$$\boxed{\Delta y = \frac{\sqrt{k(1-\gamma) \cdot s_{t+1}} - y_t}{(1-\gamma)}}$$

Case: 2.

when $s_{t+1} < p_t(1-\gamma)$

Bitcoin is cheaper in external market, ∴ Buy in real and dump/sell in pool. (increasing x),

$$x_{t+1} = x_t + (1-\gamma) \cdot x_{t+1}$$ we should take some fee from Δx shares dumped.

$$y_{t+1} = y_t - \Delta y$$

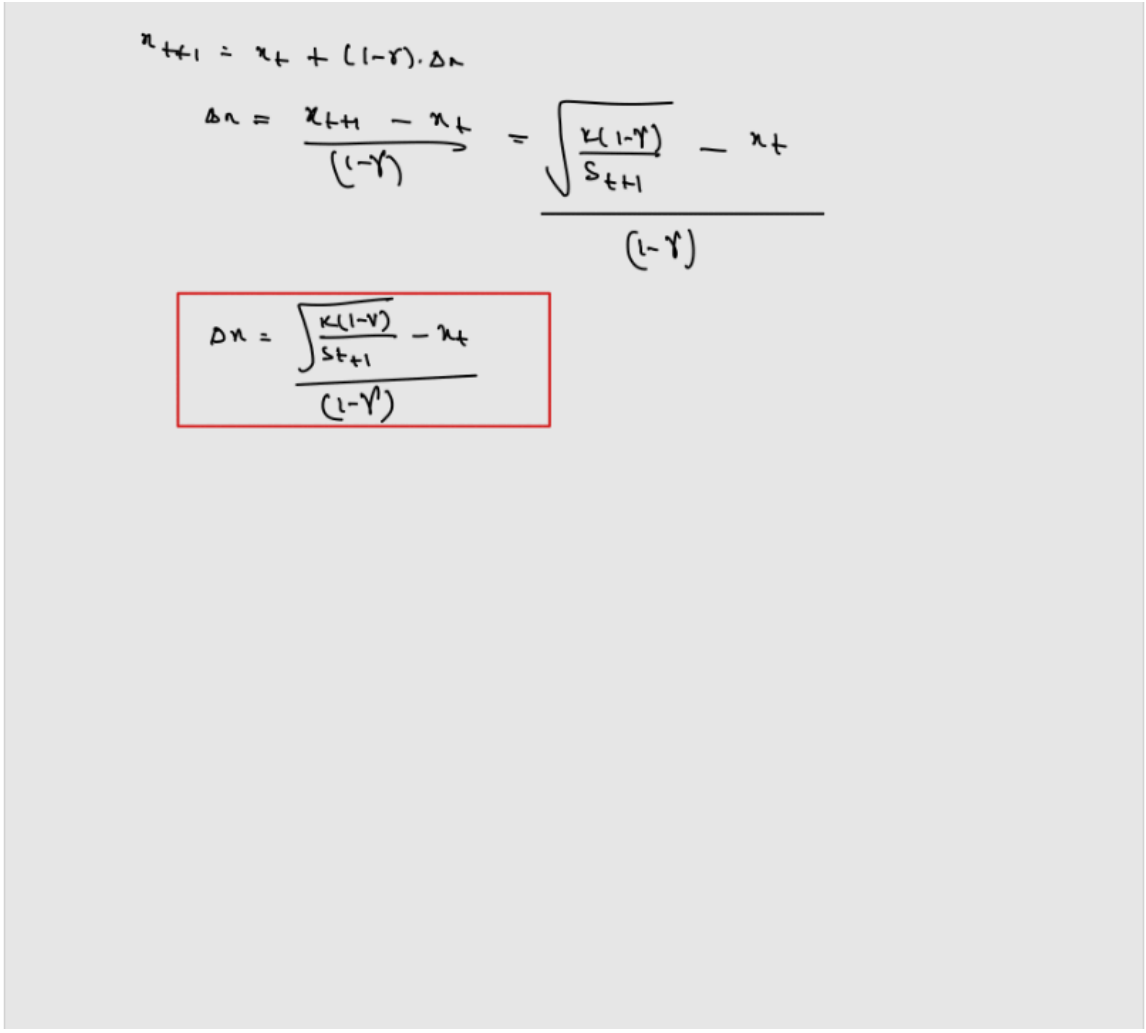
$$s_{t+1} = \frac{y_{t+1}}{x_{t+1}} \cdot (1-\gamma)$$

$$y_{t+1} = x_{t+1} \cdot \frac{s_{t+1}}{(1-\gamma)}, \quad x_{t+1} * y_{t+1} = k$$

$$x_{t+1} \cdot (x_{t+1}) \cdot \frac{s_{t+1}}{(1-\gamma)} = k$$

$$x_{t+1} = \sqrt{\frac{k(1-\gamma)}{s_{t+1}}}$$

Swap size Δx for pool



In [ ]:
```
#since we got few nan for newton iv method, we can use the bisection method to fill

data_options_1['hybrid_iv'] = data_options_1['iv_newton'].fillna(data_options_1['iv
iv_table = data_options_1[['Underlying_Symbol', 'strike', 'Expiration', 'Type', 'hy
target_date = pd.to_datetime('2026-02-13')
data_options_2['ttm'] = (pd.to_datetime(data_options_2['Expiration']) - target_date
#merge the iv data with data_options_2
data_options_2 = data_options_2.merge(iv_table, on=['Underlying_Symbol', 'strike',

r = 0.0364

#calculate the price of the option using the hybrid iv
def calculate_option_price(row):
    S0 = row['Underlying_Price_Snapshot']
    K = row['strike']
    T = row['ttm']
    opt_type = row['Type']
    sigma = row['hybrid_iv']

    if pd.isna(sigma):
        return np.nan

    return black_scholes(S0, sigma, T, K, r, opt_type)
```

```
data_options_2['option_price_theoretical'] = data_options_2.apply(calculate_option_
print(data_options_2[['strike', 'Type', 'target', 'option_price_theoretical']].head
```

```
   strike  Type   target  option_price_theoretical
0   100.0  Call  319.950                320.359709
1   110.0  Call  327.000                311.397137
2   120.0  Call  300.225                300.329834
3   130.0  Call  290.175                290.070056
4   140.0  Call  280.275                280.058511
5   150.0  Call  269.775                270.351161
6   155.0  Call  265.175                265.076155
7   160.0  Call  259.775                260.195002
8   165.0  Call  254.900                255.106900
9   170.0  Call  249.650                250.151886
```

(b) (10 pts) Expected fee revenue

Assume the initial BTC/USDC pool reserves are $x_t = y_t = 1000$, so $P_t = \frac{y_t}{x_t} = 1$. Let $S_t = 1$ and $\Delta t = \frac{1}{365}$. Assume the external price follows one-step GBM:

$$S_{t+1} = S_t \exp\left(-\tfrac{1}{2}\sigma^2 \Delta t + \sigma\sqrt{\Delta t}\, Z\right), \qquad Z \sim N(0,1),$$

so S_{t+1} is lognormal with density $f_{S_{t+1}}(s)$.

Using the setup above and the solution from (a), the expected one-step fee revenue is:

$$\mathbb{E}[R(S_{t+1})] = \int_{P_t/(1-\gamma)}^{\infty} \gamma \Delta y(S_{t+1}; x_t, y_t, \gamma, k)\, f_{S_{t+1}}(s)\, ds + \int_{0}^{P_t(1-\gamma)} \gamma \Delta x(S_{t+1}; x_t, y_t, \gamma, k) S_{t+1}\, f_{S_{t+1}}(s)\, ds.$$

Task: Numerically approximate $\mathbb{E}[R(S_{t+1})]$ using trapezoidal rule learned in class.

In [173…

```
import numpy as np
from scipy.stats import lognorm

x_t = 1000.0
y_t = 1000.0
k = x_t * y_t
P_t = y_t / x_t
S_t = 1.0
dt = 1.0 / 365.0

def delta_y(S_next, gamma):
    """USDC swapped into the pool when price spikes"""
    return (np.sqrt(k * (1 - gamma) * S_next) - y_t) / (1 - gamma)

def delta_x(S_next, gamma):
    """BTC swapped into the pool when price crashes"""
    return (np.sqrt(k * (1 - gamma) / S_next) - x_t) / (1 - gamma)

def calculate_expected_revenue(sigma , gamma):

    mu = -0.5 * (sigma**2) * dt
    std_dev = sigma * np.sqrt(dt)
```

```
    # no arbitrage band
    lower_band = P_t * (1 - gamma)
    upper_band = P_t / (1 - gamma)

    #CASE 1

    # Market Price Spikes (Right Tail)
    # Create 1,000 possible prices from the upper band up to 1.5
    prices_up = np.linspace(upper_band, 1.5, 1000)
    # Calculate the Probability of each price
    probs_up = lognorm.pdf(prices_up, s=std_dev, scale=np.exp(mu))
    # Calculate the Payout (Revenue) for each price
    payout_up = gamma * delta_y(prices_up, gamma)
    # Area under the curve (Payout * Probability)
    expected_case1 = np.trapz(payout_up * probs_up, prices_up)

    #CASE 2

    # Market Price Crashes (Left Tail)
    # Create 1,000 possible prices from near zero up to the lower band
    prices_down = np.linspace(0.0001, lower_band, 1000)
    probs_down = lognorm.pdf(prices_down, s=std_dev, scale=np.exp(mu))
    # We multiply by 'prices_down' to convert the BTC fee back to USDC
    payout_down = gamma * delta_x(prices_down, gamma) * prices_down

    expected_case2 = np.trapz(payout_down * probs_down, prices_down)

    # Total Expected Revenue
    return expected_case1 + expected_case2
```

(c) (10 pts) Optimal Fee Rate under different volatilities

Use $\sigma \in \{0.2, 0.6, 1.0\}$ and $\gamma \in \{0.001, 0.003, 0.01\}$. For each σ, compute $\mathbb{E}[R]$ for the three fee rates and select:

$$\gamma^*(\sigma) = \arg\max_{\gamma} \mathbb{E}[R(S_{t+1})].$$

Construct a table reporting the $\mathbb{E}[R]$ values and the best $\gamma^*(\sigma)$ among the 3 options.

Then for each $\sigma \in [0.1, 1.0]$ on a grid (e.g. 0.01 step), compute the optimal $\gamma^*(\sigma)$. Finally, produce a scatter plot or line plot for σ vs. $\gamma^*(\sigma)$, and comment briefly on the pattern you observe.

In [174…

```
sigmas_to_test = [0.2, 0.6, 1.0]
gammas_to_test = [0.001, 0.003, 0.01]

table_data = []
```

```
for sigma in sigmas_to_test:
    row = {'Sigma (σ)': sigma}
    best_gamma = None
    best_revenue = -np.inf
    for gamma in gammas_to_test:
        rev = calculate_expected_revenue(sigma, gamma)
        if rev > best_revenue:
            best_revenue = rev
            best_gamma = gamma
    row['Optimal gamma (g)'] = best_gamma
    table_data.append(row)


df_table = pd.DataFrame(table_data)
print("Expected Revenue Table")
print(df_table.to_string(index=False))
```

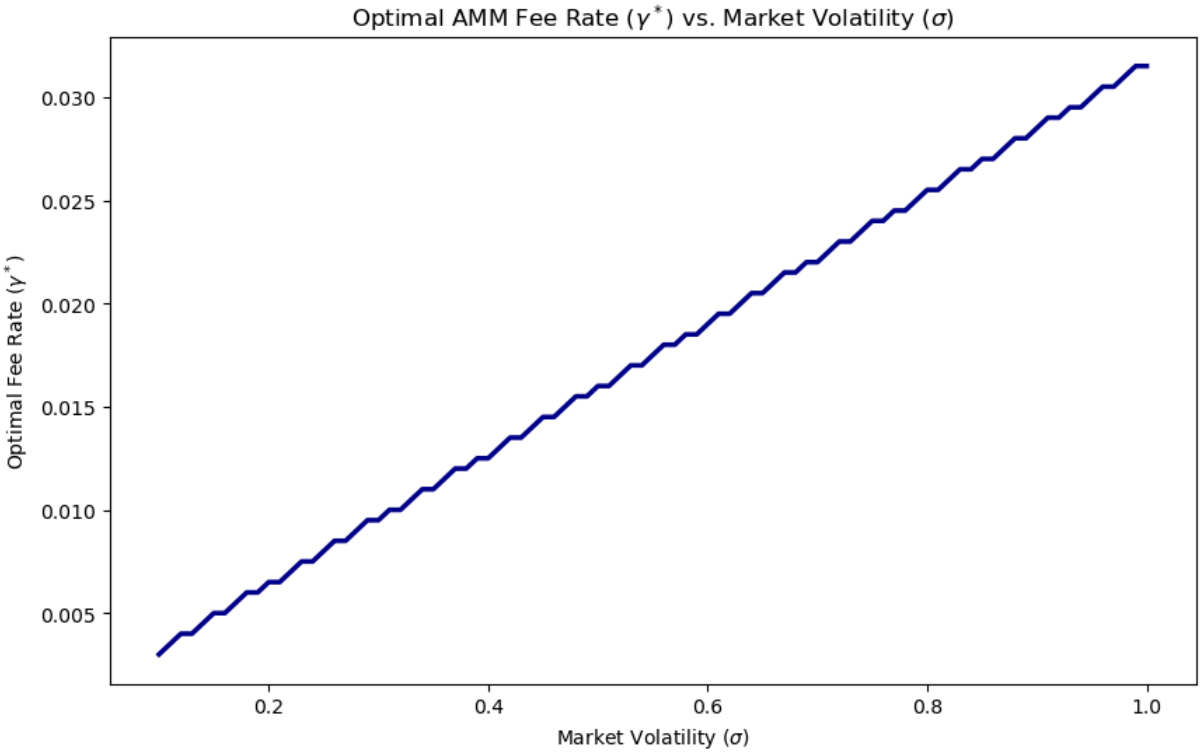
```
Expected Revenue Table
 Sigma (σ)  Optimal gamma (g)
       0.2              0.01
       0.6              0.01
       1.0              0.01
```

In [177…

```
#plot

print("Calculating optimal fees for the plot")

# Create the grid for Volatility
sigma_grid = np.arange(0.1, 1.01, 0.01)

# Create a fine grid of possible fees to search for the absolute best one
# We will test fees from 0.05% up to 5%
gamma_search_grid = np.arange(0.0005, 0.5, 0.0005)

optimal_gammas = []

for sig in sigma_grid:
    best_rev = -1.0
    best_gamma = None

    # Test every fee in our search grid
    for g in gamma_search_grid:
        rev = calculate_expected_revenue(sig, g)
        if rev > best_rev:
            best_rev = rev
            best_gamma = g

    optimal_gammas.append(best_gamma)

# Draw the plot
plt.figure(figsize=(10, 6))
plt.plot(sigma_grid, optimal_gammas, color='darkblue', linewidth=2.5)
plt.title('Optimal AMM Fee Rate ($\gamma^*$) vs. Market Volatility ($\sigma$)')
plt.xlabel('Market Volatility ($\sigma$)')
```

```
plt.ylabel('Optimal Fee Rate ($\gamma^*$)')
plt.show()
```

```
Calculating optimal fees for the plot
```



There is no optimal fee rate here , we can have a dynamic fee rate based on the sigma of the market, becuase the higher the sigma the more arbitrage opportunities will arise, so increasing the fee rate will reduce the arbitrage opportunities and reduce their profit margin while maxiimizing ours. Similarly for calm markets we can have a lower fee rate.

The steps we see in the graph are basically our grid search in action.

In [ ]: