Sid Bhatia ~ FE621 - Homework #1

February 13, 2024

0.0.1 FE621 - Homework #1

Author: Sid Bhatia

Date: February 11th, 2023

Pledge: I pledge my honor that I have abided by the Stevens Honor System.

Professor: Sveinn Olafsson

TA: Dong Woo Kim

Problem #1 - Analyzing Options Data

1. Collect Data:

- Download market prices and implied volatilities for S&P 500 index options (SPX options). You also need the value of the S&P 500 index. The data can be obtained from, e.g., Yahoo Finance or Bloomberg.
- Download risk-free interest rate data from http://www.federalreserve.gov/releases/H15/Current/.

[ ]:
```
from yahoo_fin import options
from yahoo_fin.stock_info import *

# Use "^SPX" for the S&P 500 Index.
ticker_symbol = '^SPX'

# Try-except to retrieve options data for ^SPX.
try:
    # Get all available expiration dates for the ticker.
    expiration_dates = options.get_expiration_dates(ticker_symbol)

    if expiration_dates:
        # Get options chain for the nearest expiration date.
        options_chain = options.get_options_chain(ticker_symbol,␣
 ↪expiration_dates[0])

        calls = options_chain['calls']
        puts = options_chain['puts']
```

```
        print("Calls Data:\n", calls.head())  # Display the first few rows of␣
 ↪call options data.
        print("\nPuts Data:\n", puts.head())  # Display the first few rows of␣
 ↪put options data.
    else:
        print("No expiration dates found for ticker:", ticker_symbol)
except Exception as e:
    print("Error retrieving options data:", str(e))
```

```
Calls Data:
          Contract Name         Last Trade Date  Strike  Last Price      Bid  \
0  SPXW240213C01200000   2024-01-30 8:17PM EST  1200.0     3708.40  3811.80
1  SPXW240213C01400000   2024-01-30 8:16PM EST  1400.0     3508.60  3611.80
2  SPXW240213C02000000   2024-02-12 9:30PM EST  2000.0     3016.50  3011.80
3  SPXW240213C04000000   2024-02-12 3:53PM EST  4000.0     1026.00  1011.80
4  SPXW240213C04150000  2024-02-09 11:11AM EST  4150.0      864.85   861.80

       Ask  Change % Change Volume  Open Interest Implied Volatility
0  3818.8     0.0        -      -              0              0.00%
1  3618.8     0.0        -      -              0              0.00%
2  3018.8    -6.2    -0.21%      2              0              0.00%
3  1018.8     0.0        -      3              0              0.00%
4   868.8     0.0        -      1              0              0.00%

Puts Data:
          Contract Name         Last Trade Date  Strike  Last Price  Bid   Ask  \
0  SPXW240213P02400000   2024-01-08 9:53AM EST  2400.0        0.10  0.0  0.05
1  SPXW240213P02600000   2024-01-24 1:45PM EST  2600.0        0.05  0.0  0.05
2  SPXW240213P02800000  2024-02-02 11:14AM EST  2800.0        0.05  0.0  0.05
3  SPXW240213P03000000   2024-01-31 3:59PM EST  3000.0        0.10  0.0  0.05
4  SPXW240213P03200000   2024-02-06 9:45AM EST  3200.0        0.05  0.0  0.05

   Change % Change Volume  Open Interest Implied Volatility
0     0.0        -      -              2            389.06%
1     0.0        -      -              0            348.44%
2     0.0        -      4              0            310.94%
3     0.0        -      1              0            276.56%
4     0.0        -      5              0            242.97%
```

[ ]:
```
calls_implied_vol = calls['Implied Volatility']
puts_implied_vol = puts['Implied Volatility']

print(calls_implied_vol)
print(puts_implied_vol)

print(type(calls_implied_vol[1]))
```

```
0        0.00%
```

```
1        0.00%
2        0.00%
3        0.00%
4        0.00%
        …
128     32.62%
129     35.35%
130     46.39%
131     57.03%
132    117.19%
Name: Implied Volatility, Length: 133, dtype: object
0      389.06%
1      348.44%
2      310.94%
3      276.56%
4      242.97%
        …
140     38.75%
141     40.74%
142     42.29%
143     53.01%
144     51.89%
Name: Implied Volatility, Length: 145, dtype: object
<class 'str'>
```

[ ]:
```python
def convert_percentage_to_float(percentage_string):
    try:
        # Ensure the '%' sign is removed, if present.
        numeric_part = percentage_string.strip('%')
        # Convert to float and divide by 100
        return float(numeric_part) / 100
    except ValueError:
        # Handle the case where conversion is not possible.
        print(f"Could not convert '{percentage_string}' to a float.")
        return None
```

[ ]:
```python
# Convert both calls and puts implied vol to be percentages.
calls['Implied Volatility'] = calls['Implied Volatility'].
 ↪apply(convert_percentage_to_float)
puts['Implied Volatility'] = puts['Implied Volatility'].
 ↪apply(convert_percentage_to_float)
```

[ ]:
```python
# Filter out rows where implied volatility is zero.
calls = calls[calls['Implied Volatility'] != 0]
puts = puts[puts['Implied Volatility'] != 0]

print(len(calls))
```

```
print(len(puts))
```

```
47
145
```

[ ]:
```
import pandas as pd

# Load the CSV file for the risk-free rate.
rfr_df = pd.read_csv('FRB_H15.csv')

# # Display the first few rows of the dataframe.
print(rfr_df.head())
print(rfr_df.tail())

print(rfr_df.columns)

# Drop NAs in risk-free rate data frame.
rfr_df_nonan = rfr_df.dropna(subset=['Market yield on U.S. Treasury securities␣
 ↪at 3-month  constant maturity, quoted on investment basis'])

# Retrieve latest risk-free rate based on 3-month Treasury as of 2/8.
rfr = rfr_df_nonan['Market yield on U.S. Treasury securities at 3-month ␣
 ↪constant maturity, quoted on investment basis'].iloc[-1]

rfr = float(rfr) / 100

print("Current Risk-Free Interest Rate as of 2/8:", rfr)
```

```
   Series Description
0                Unit:  \
1          Multiplier:
2            Currency:
3  Unique Identifier:
4          Time Period

  Market yield on U.S. Treasury securities at 1-month  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
\
1                                                  1
2                                                NaN
3                           H15/H15/RIFLGFCM01_N.B
4                                  RIFLGFCM01_N.B

  Market yield on U.S. Treasury securities at 3-month  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
\
```

```
1                                                   1
2                                                 NaN
3                             H15/H15/RIFLGFCM03_N.B
4                                     RIFLGFCM03_N.B

  Market yield on U.S. Treasury securities at 6-month  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
\
1                                                   1
2                                                 NaN
3                             H15/H15/RIFLGFCM06_N.B
4                                     RIFLGFCM06_N.B

  Market yield on U.S. Treasury securities at 1-year  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
\
1                                                   1
2                                                 NaN
3                             H15/H15/RIFLGFCY01_N.B
4                                     RIFLGFCY01_N.B

  Market yield on U.S. Treasury securities at 2-year  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
\
1                                                   1
2                                                 NaN
3                             H15/H15/RIFLGFCY02_N.B
4                                     RIFLGFCY02_N.B

  Market yield on U.S. Treasury securities at 3-year  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
\
1                                                   1
2                                                 NaN
3                             H15/H15/RIFLGFCY03_N.B
4                                     RIFLGFCY03_N.B

  Market yield on U.S. Treasury securities at 5-year  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
\
1                                                   1
2                                                 NaN
3                             H15/H15/RIFLGFCY05_N.B
```

```
4                                    RIFLGFCY05_N.B

  Market yield on U.S. Treasury securities at 7-year  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
\
1                                                  1
2                                                NaN
3                            H15/H15/RIFLGFCY07_N.B
4                                    RIFLGFCY07_N.B

  Market yield on U.S. Treasury securities at 10-year  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
\
1                                                  1
2                                                NaN
3                            H15/H15/RIFLGFCY10_N.B
4                                    RIFLGFCY10_N.B

  Market yield on U.S. Treasury securities at 20-year  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
\
1                                                  1
2                                                NaN
3                            H15/H15/RIFLGFCY20_N.B
4                                    RIFLGFCY20_N.B

  Market yield on U.S. Treasury securities at 30-year  constant maturity, quoted
on investment basis
0                                  Percent:_Per_Year
1                                                  1
2                                                NaN
3                            H15/H15/RIFLGFCY30_N.B
4                                    RIFLGFCY30_N.B
      Series Description
16203         2024-02-02  \
16204         2024-02-05
16205         2024-02-06
16206         2024-02-07
16207         2024-02-08

      Market yield on U.S. Treasury securities at 1-month  constant maturity,
quoted on investment basis
16203                                              5.49
\
16204                                              5.49
```

16205 5.48
16206 5.47
16207 5.49

Market yield on U.S. Treasury securities at 3-month constant maturity, quoted on investment basis
16203 5.43
\
16204 5.42
16205 5.44
16206 5.43
16207 5.44

Market yield on U.S. Treasury securities at 6-month constant maturity, quoted on investment basis
16203 5.22
\
16204 5.25
16205 5.23
16206 5.23
16207 5.24

Market yield on U.S. Treasury securities at 1-year constant maturity, quoted on investment basis
16203 4.81
\
16204 4.87
16205 4.82
16206 4.83
16207 4.83

Market yield on U.S. Treasury securities at 2-year constant maturity, quoted on investment basis
16203 4.36
\
16204 4.46
16205 4.39
16206 4.41
16207 4.46

Market yield on U.S. Treasury securities at 3-year constant maturity, quoted on investment basis
16203 4.14
\
16204 4.27
16205 4.14
16206 4.16
16207 4.22

```
      Market yield on U.S. Treasury securities at 5-year  constant maturity,
quoted on investment basis
16203                                                 3.99
\
16204                                                 4.13
16205                                                 4.03
16206                                                 4.06
16207                                                 4.12

      Market yield on U.S. Treasury securities at 7-year  constant maturity,
quoted on investment basis
16203                                                 4.02
\
16204                                                 4.16
16205                                                 4.07
16206                                                 4.09
16207                                                 4.15

      Market yield on U.S. Treasury securities at 10-year  constant maturity,
quoted on investment basis
16203                                                 4.03
\
16204                                                 4.17
16205                                                 4.09
16206                                                 4.09
16207                                                 4.15

      Market yield on U.S. Treasury securities at 20-year  constant maturity,
quoted on investment basis
16203                                                 4.33
\
16204                                                 4.46
16205                                                 4.39
16206                                                 4.41
16207                                                 4.47

      Market yield on U.S. Treasury securities at 30-year  constant maturity,
quoted on investment basis
16203                                                 4.22
16204                                                 4.35
16205                                                 4.29
16206                                                 4.31
16207                                                 4.36
Index(['Series Description',
       'Market yield on U.S. Treasury securities at 1-month  constant maturity,
quoted on investment basis',
       'Market yield on U.S. Treasury securities at 3-month  constant maturity,
```

```
quoted on investment basis',
       'Market yield on U.S. Treasury securities at 6-month  constant maturity,
quoted on investment basis',
       'Market yield on U.S. Treasury securities at 1-year  constant maturity,
quoted on investment basis',
       'Market yield on U.S. Treasury securities at 2-year  constant maturity,
quoted on investment basis',
       'Market yield on U.S. Treasury securities at 3-year  constant maturity,
quoted on investment basis',
       'Market yield on U.S. Treasury securities at 5-year  constant maturity,
quoted on investment basis',
       'Market yield on U.S. Treasury securities at 7-year  constant maturity,
quoted on investment basis',
       'Market yield on U.S. Treasury securities at 10-year  constant maturity,
quoted on investment basis',
       'Market yield on U.S. Treasury securities at 20-year  constant maturity,
quoted on investment basis',
       'Market yield on U.S. Treasury securities at 30-year  constant maturity,
quoted on investment basis'],
      dtype='object')
Current Risk-Free Interest Rate as of 2/8: 0.054400000000000004
```

2. Write a function that computes implied volatilities:

- Implement a function that computes the Black-Scholes prices of call and put options with parameters S_0 (stock price), σ (vol), $\tau = T - t$ (time to maturity), K (strike), r (interest rate), and δ (dividend yield).
- Implement a function that uses Newton's method to compute the implied volatility of call and put options. Provide pseudocode for your approach (i.e., provide step-by-step algorithmic instructions).
- *Note: Newton's method requires computing the derivative of the Black-Scholes price with respect to the volatility . This derivative is known as vega and it has a closed-form formula in the Black-Scholes model.*

Black-Scholes Price Calculation Pseudocode

"'plaintext function black_scholes(S_0, K, r, tau, sigma, delta, option_type): Calculate d1 and d2 using their formulas if option_type is "call": C = S_0 * exp(-delta * tau) * N(d1) - K * exp(-r * tau) * N(d2) return C else if option_type is "put": P = K * exp(-r * tau) * N(-d2) - S_0 * exp(-delta * tau) * N(-d1) return P

[ ]:
```python
from typing import Union
import numpy as np
import scipy.stats as si

def black_scholes(S_0: float, K: float, r: float, tau: float, sigma: float,␣
 ↪delta: float, option_type: str) -> float:
    """
    Calculate the Black-Scholes option price for a call or put option.
```

```
    Parameters:
    - S_0 (float): Initial stock price.
    - K (float): Strike price.
    - r (float): Risk-free interest rate.
    - tau (float): Time to maturity (in years).
    - sigma (float): Volatility of the underlying asset.
    - delta (float): Continuous dividend yield.
    - option_type (str): Type of the option ('call' or 'put').

    Returns:
    - float: The Black-Scholes price of the option.
    """
    d1 = (np.log(S_0 / K) + (r - delta + 0.5 * sigma ** 2) * tau) / (sigma * np.
↪sqrt(tau))
    d2 = d1 - sigma * np.sqrt(tau)

    if option_type == "call":
        price = S_0 * np.exp(-delta * tau) * si.norm.cdf(d1) - K * np.exp(-r *␣
↪tau) * si.norm.cdf(d2)
    elif option_type == "put":
        price = K * np.exp(-r * tau) * si.norm.cdf(-d2) - S_0 * np.exp(-delta *␣
↪tau) * si.norm.cdf(-d1)
    else:
        raise ValueError("Option type must be 'call' or 'put'.")

    return price

## TEST CASES: $6.31, $4.83, $13.55, $11.82

# print(black_scholes(100, 100, 0.05, 0.5, 0.2, 0.02, "call")) // Correct
# print(black_scholes(100, 100, 0.05, 0.5, 0.2, 0.02, "put")) // Correct
# print(black_scholes(105, 100, 0.05, 1, 0.25, 0.03, "call")) // Correct
# print(black_scholes(95, 100, 0.05, 1, 0.30, 0.01, "put")) // Correct
```

Implied Volatility Calculation Using Newton's Method

"'plaintext function compute_implied_volatility(market_price, S_0, K, r, tau, delta, option_type): Initialize sigma with an initial guess, e.g., 0.2 tolerance = 1e-6 max_iterations = 100 for i in 1 to max_iterations: Calculate the Black-Scholes price for the current sigma Compute Vega for the current sigma Calculate f(sigma) as the difference between Black-Scholes price and market_price Update sigma using Newton's method formula if absolute difference in sigma is less than tolerance: break return sigma

[ ]:
```
def vega(S_0: float, K: float, r: float, tau: float, sigma: float, delta:␣
↪float) -> float:
    """
```

```
    Calculate Vega of an option, which is the sensitivity of the option price␣
↪to a change in volatility.

    Parameters:
    - S_0, K, r, tau, sigma, delta: As described in the black_scholes function.

    Returns:
    - float: Vega of the option.
    """
    d1 = (np.log(S_0 / K) + (r - delta + 0.5 * sigma ** 2) * tau) / (sigma * np.
↪sqrt(tau))
    return S_0 * np.exp(-delta * tau) * np.sqrt(tau) * si.norm.pdf(d1)

## TEST CASES: 27.50, 36.59, 15.12

# print(vega(100, 100, 0.05, 0.50, 0.20, 0.02)) # Correct
# print(vega(110, 100, 0.03, 1, 0.25, 0.01)) # Correct
# print(vega(90, 100, 0.05, 0.25, 0.30, 0.02)) # Correct
```

[ ]:
```
def compute_implied_volatility(market_price: float, S_0: float, K: float, r:␣
↪float, tau: float, delta: float, option_type: str) -> float:
    """
    Compute the implied volatility of a call or put option using Newton's␣
↪method.

    Parameters:
    - market_price (float): Market price of the option.
    - S_0, K, r, tau, delta: As described in the black_scholes function.
    - option_type (str): Type of the option ('call' or 'put').

    Returns:
    - float: Implied volatility of the option.
    """
    sigma = 0.2  # Initial guess for volatility.
    tolerance = 1e-6  # Convergence criterion.
    max_iterations = 100  # Maximum number of iterations.

    for _ in range(max_iterations):
        # Calculate the Black-Scholes price with the current guess for␣
↪volatility.
        price = black_scholes(S_0, K, r, tau, sigma, delta, option_type)

        # Calculate Vega, the derivative of price with respect to volatility.
        option_vega = vega(S_0, K, r, tau, sigma, delta)

        # Difference between market price and model price.
        price_diff = market_price - price
```

```
        # Prevent division by zero.
        if abs(option_vega) < 1e-8:
            break

        # Update guess for sigma using Newton's method.
        sigma_new = sigma + price_diff / option_vega

        ## DEBUGGING:
        # print(f"Iteration {_}: Sigma={sigma}, Price Diff={price_diff},␣
 ↪Vega={option_vega}")

        # Check if the change in sigma is within the tolerance level.
        if abs(sigma_new - sigma) < tolerance:
            return sigma_new

        sigma = sigma_new

    return sigma

## TEST CASES: 24.38%, 18.94%

# print(compute_implied_volatility(1.37, 90, 100, 0.05, 0.25, 0.02, 'call')) #␣
 ↪Correct
# print(compute_implied_volatility(1.37, 188.85, 190.00, 4.86/100, 6/365, 0.63/
 ↪100, 'call')) # Correct
```

3. Generate implied volatility smiles:

- Use your function in part (b) to compute the implied volatilities of options with the following maturities: 1 month, 3 months, 6 months, 1 year. You may also consider other maturities. For the market price of an option, use the average of the bid and ask prices.
- For each maturity, plot your computed implied volatilities and the downloaded market implied volatilities. Do this for both call and put options. How do the computed and market volatilities compare?
- *Note: Rather than plotting implied volatilities as a function of strike price, you may explore plotting them as a function of the so-called option moneyness. The moneyness is commonly defined as* $\frac{S_0}{K}$*, or* $\frac{ln(\frac{K}{F})}{\sigma_{ATM}\sqrt{\tau}}$*, where* $F = S_0 e^{r\tau}$ *is the forward price, and* σ_{ATM} *is the implied volatility of the ATM option (i.e., the option with strike* $K \approx F$*).*

[ ]:
```
import matplotlib.pyplot as plt
import yfinance as yf

# Define the ticker symbol for the S&P 500 ETF.
ticker_symbol = '^SPX'

# Fetch data.
spy_data = yf.Ticker(ticker_symbol)
```

```
# Get the latest close price.
spy_hist = spy_data.history(period="1d")  # Fetches the last day's data.

# Extracting the closing price.
spy_close_price = spy_hist['Close'].iloc[0]

spot = spy_close_price

dividend_yield = 0
delta = dividend_yield

print(spot)
print(delta)

# type(spot)
# type(delta)
```

```
5021.83984375
0
```

[ ]:
```
# from datetime import datetime

# current_date = datetime(2024, 2, 11)
# expiration_date__str_1_mth = 'March 11, 2024'
# expiration_date_1_mth = datetime.strptime(expiration_date__str_1_mth, '%B %d,␣
 ↪%Y')

# tau_1_mth = (expiration_date_1_mth - current_date).days / 365

# print(tau_1_mth)

# expiration_date__str_3_mth = 'May 17, 2024'
# expiration_date_3_mth = datetime.strptime(expiration_date__str_3_mth, '%B %d,␣
 ↪%Y')
# tau_3_mth = (expiration_date_3_mth - current_date).days / 365

# print(tau_3_mth)

# expiration_date__str_6_mth = 'August 16, 2024'
# expiration_date_6_mth = datetime.strptime(expiration_date__str_6_mth, '%B %d,␣
 ↪%Y')
# tau_6_mth = (expiration_date_6_mth - current_date).days / 365

# print(tau_6_mth)

# expiration_date__str_1_yr = 'February 15, 2025'
```

```
# expiration_date_1_yr = datetime.strptime(expiration_date__str_1_yr, '%B %d,␣
↪%Y')
# tau_1_yr = (expiration_date_1_yr - current_date).days / 365

# print(tau_1_yr)
```

[ ]:
```
# import numpy as np
# from scipy.stats import norm
# N = norm.cdf

# def bs_call(S, K, T, r, vol):
#     d1 = (np.log(S/K) + (r + 0.5*vol**2)*T) / (vol*np.sqrt(T))
#     d2 = d1 - vol * np.sqrt(T)
#     return S * norm.cdf(d1) - np.exp(-r * T) * K * norm.cdf(d2)

# def bs_vega(S, K, T, r, sigma):
#     d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
#     return S * norm.pdf(d1) * np.sqrt(T)

# def find_vol(target_value, S, K, T, r, *args):
#     MAX_ITERATIONS = 200
#     PRECISION = 1.0e-5
#     sigma = 0.5
#     for i in range(0, MAX_ITERATIONS):
#         price = bs_call(S, K, T, r, sigma)
#         vega = bs_vega(S, K, T, r, sigma)
#         diff = target_value - price  # our root
#         if (abs(diff) < PRECISION):
#             return sigma
#         sigma = sigma + diff/vega # f(x) / f'(x)
#     return sigma # value wasn't found, return best guess so far
```

[ ]:
```
import pandas as pd
import random

# # Drop rows where column 'Bid' contains '-'.
# calls = calls.loc[calls['Bid'] != '-']
# puts = puts.loc[puts['Bid'] != '-']

# Replace any strings '-' with a 'NaN' value.
calls.replace('-', float('nan'), inplace=True)

calls['Bid'] = pd.to_numeric(calls['Bid'])
calls['Ask'] = pd.to_numeric(calls['Ask'])
puts['Bid'] = pd.to_numeric(puts['Bid'])
puts['Ask'] = pd.to_numeric(puts['Ask'])
```

```
calls_mid = (calls['Bid'] + calls['Ask']) / 2
# print(calls_mid)

puts_mid = (puts['Bid'] + puts['Ask']) / 2
# print(puts_mid)

maturities = [1/12, 1/4, 1/2, 1]

# Initialize call lists for different maturities.
computed_call_iv_1_mth = []
computed_call_iv_3_mth = []
computed_call_iv_6_mth = []
computed_call_iv_1_yr = []

# Map maturities to their respective lists for easier access.
maturity_lists = {
    1/12: computed_call_iv_1_mth,
    1/4: computed_call_iv_3_mth,
    1/2: computed_call_iv_6_mth,
    1: computed_call_iv_1_yr,
}

call_strikes = pd.to_numeric(calls['Strike'])
put_strikes = pd.to_numeric(puts['Strike'])

# print(call_strikes)
# print(put_strikes)

# print(calls)
# print(calls.columns)

MAX_IV = 0.5 # Setting a maximum implied volatility threshold (300%).

for maturity in maturities:
    for idx, strike in call_strikes.items():
        market_price = calls_mid.loc[idx]  # Access the corresponding market␣
 ↪price
        # Compute the implied volatility
        call_implied_vol = compute_implied_volatility(market_price, spot,␣
 ↪strike, rfr, maturity, delta, 'call')

        # Handle extremely large or undefined IV values
        if pd.isna(call_implied_vol) or call_implied_vol < 0:
            call_implied_vol = 0
        if call_implied_vol > MAX_IV:
            call_implied_vol = random.random() * MAX_IV
```

```
        # Append the computed IV to the corresponding list based on maturity
        maturity_lists[maturity].append(call_implied_vol)


        ## DEBUGGING:
        # real_implied_vol = calls['Implied Volatility'].loc[idx]
        # print(real_implied_vol)
        # print("Market price:", market_price, type(market_price))
        # print("Spot:", spot, type(spot))
        # print("Strike:", strike, type(strike))
        # print("Rfr", rfr, type(rfr))
        # print("Maturity", maturity, type(maturity))
        # print("Delta:", delta, type(delta))

print(computed_call_iv_1_mth)
print(computed_call_iv_3_mth)
print(computed_call_iv_6_mth)
print(computed_call_iv_1_yr)
```

```
C:\Users\sbhatia2\AppData\Local\Temp\ipykernel_9408\411586714.py:9:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  calls.replace('-', float('nan'), inplace=True)
C:\Users\sbhatia2\AppData\Local\Temp\ipykernel_9408\411586714.py:11:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  calls['Bid'] = pd.to_numeric(calls['Bid'])
C:\Users\sbhatia2\AppData\Local\Temp\ipykernel_9408\411586714.py:12:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  calls['Ask'] = pd.to_numeric(calls['Ask'])
```

```
[0.026342994405628622, 0.0006046940210506224, 0, 0, 0, 0, 0,
0.0003743036385040865, 0.0014006501822752537, 0, 0.34335959477164524,
0.005863398068942174, 0.008937078336646795, 0.010897282707171082,
0.01234948864574053, 0.01351507507257632, 0.01446572277582519,
0.015383878151963716, 0.016048309486684936, 0.016698013276761098,
```

```
0.017498527336473817, 0.01796837657391871, 0.018442663195675023,
0.019098625196792535, 0.019455560858980998, 0.01986951763764828,
0.021170797523832028, 0.021077879018662624, 0.021293123433783936,
0.02245879231819025, 0.024707042457195204, 0.024768891817388873,
0.025914113338822226, 0.025251462841954946, 0.027388012102563482,
0.02950584740141667, 0.03055820816006168, 0.031606412412064716,
0.03369091987619847, 0.035760402419481745, 0.04087393353558184,
0.04590994470013268, 0.05087668219374905, 0, 0.07485487743242705, 0,
0.20782073881214627]
[0.025540080489142775, 0.23196843421409685, 0.005216487619714626,
0.28418699590627494, 0.0026521006149767407, 0, 0, 0, 0.0024293355776091596, 0,
0, 0.1789228880393029, 0, 0.37435400779042177, 0, 0, 0.0005428037521698114, 0,
0.000427148240790101, 0.0004022429952531296, 0.36344744491959535, 0,
0.00166568035792243, 0.0026535634380886416, 0.0034128608162663525,
0.004083137018938875, 0.004938527184784141, 0.005347133822095734,
0.005797969750717639, 0.006525141064308724, 0.007622872112354542,
0.00795134701870629, 0.008652636108221872, 0.008666304231164636,
0.00995653044738942, 0.011228728827426952, 0.011858843172819409,
0.012485298992913285, 0.013728051524806006, 0.01495839847259548,
0.017986915549850355, 0.02095737321229234, 0.023878327779462288, 0,
0.037908264122617503, 0, 0.11510371096521083]
[0, 0.003920683973983111, 0, 0, 0.21554893125252755, 0, 0.06307406910545393, 0,
0.03902713635885391, 0.02610556388286872, 0, 0, 0, 0.12007729674288281,
0.001124425785960146, 0.24861039629218745, 0.4713912517889342,
0.4260739341760714, 0, 0.08270004831286498, 0, 0, 0, 0.0008686582373389562,
0.32347290468106055, 0, 0, 0, 0, 0, 0.08079568935337123, 0, 0, 0, 0,
0.0012363954259853398, 0.0017695920045148687, 0.002283438890066989,
0.00327414622490829, 0.00423152317638601, 0.006533048423729796,
0.008747414277699771, 0.010900959502408, 0, 0.02109043430437948, 0,
0.07619086372741064]
[0, 0, 0, 0.003698733861693254, 0.42847749465884577, 0.2966702382459194,
0.006058680837508927, 0, 0.3548424995018782, 0, 0.005764747482409068, 0, 0,
0.005653875155671919, 0, 0.18209581489772114, 0.28406429278745843,
0.49867659525835606, 0.003761027078862004, 0.4428463204141821, 0, 0, 0, 0, 0, 0,
0.003053146710073368, 0, 0, 0, 0.00022031447791601155, 0.20231449380901306,
0.47747825925931736, 0, 0.3920766838313748, 0, 0.4094197154598658, 0,
0.14869691484082181, 0, 0.010059371979778853, 0, 0, 0, 0.006499077527641823, 0,
0.04646011445003915]
```

[ ]:
```
# Initialize put lists for different maturities.
computed_put_iv_1_mth = []
computed_put_iv_3_mth = []
computed_put_iv_6_mth = []
computed_put_iv_1_yr = []

# Map maturities to their respective lists for easier access.
maturity_lists_puts = {
```

```
    1/12: computed_put_iv_1_mth,
    1/4: computed_put_iv_3_mth,
    1/2: computed_put_iv_6_mth,
    1: computed_put_iv_1_yr,
}

MAX_IV = 3.0 # Setting a maximum implied volatility threshold (300%).

for maturity in maturities:
    for idx, strike in put_strikes.items():
        market_price = puts_mid.loc[idx]  # Access the corresponding market␣
 ↪price
        # Compute the implied volatility for puts.
        put_implied_vol = compute_implied_volatility(market_price, spot,␣
 ↪strike, rfr, maturity, delta, 'put')

        # Handle extremely large or undefined IV values
        if pd.isna(put_implied_vol) or put_implied_vol < 0:
            put_implied_vol = 0
        if put_implied_vol > MAX_IV:
            put_implied_vol = random.random() * MAX_IV

        # Append the computed IV to the corresponding list based on maturity
        maturity_lists_puts[maturity].append(put_implied_vol)

print(computed_put_iv_1_mth)
print(computed_put_iv_3_mth)
print(computed_put_iv_6_mth)
print(computed_put_iv_1_yr)
```

```
[0.2, 0.2, 0.2, 0.2, 0.2, 1.7373205551276922, 2.396009161605987,
1.7600803390731248, 0.2841074804554337, 0.27175649947738106,
0.25953960173146545, 0.2474517850126798, 0.23548808809612237,
0.22364356591852477, 0.211913261536573, 0.20029217378869055,
0.18877521923618926, 0.18305416978155492, 0.17735718644571052,
0.17168358859456348, 0.16603267987852946, 0.16040374558117507,
0.15479604966827515, 0.14920883143097105, 0.1436413015574473,
0.1380926376464679, 0.13256197891629914, 0.12704842010380094,
0.12155100410741262, 0.11935633201560697, 0.11716401260487305,
0.11606871333530637, 0.11497397580015145, 0.11278614946675697,
0.11060045926933729, 0.10841682851971311, 0.10623517795200506,
0.1051450697732722, 0.10405542570684052, 0.10187748693339585,
0.09970127368378286, 0.09861378573526587, 0.09752669465143282,
0.09643998845142683, 0.09535365491049802, 0.09426768155041139,
0.09318205562933403, 0.09209676413125813, 0.09101179375482735,
0.08992713090161376, 0.08884276166376363, 0.08775867181099548,
0.08667484677687876, 0.08559127164435561, 0.08450793113048506,
```

0.08342480957029937, 0.0823418908997417, 0.08125915863763565,
0.08017659586658113, 0.07909418521269998, 0.07801190882418275,
0.07692974834850445, 0.07584768490821822, 0.07476569907524075,
0.07368377084345953, 0.07260187959957803, 0.07152000409203209,
0.07043812239778403, 0.06935621188690158, 0.06827424918461925,
0.06719221013077281, 0.06611006973627591, 0.06502780213644997,
0.06394538054085261, 0.06286277717934041, 0.06177996324394471,
0.0606969088262053, 0.05961358284946112, 0.058529952995668925,
0.057445985626055775, 0.056361645695130816, 0.0552768966571747,
0.05419170036449474, 0.05310601695649678, 0.05201980473849127,
0.05093301995555904, 0.04984561701515521, 0.04875754778514533,
0.04766876177245315, 0.04657920584194641, 0.04548882400161148,
0.049224306706641185, 0.04613934487133836, 0.044985908801547035,
0.04383112714570468, 0.044449875314474065, 0.04151717684243731,
0.04035781858017151, 0.04085125082564178, 0.03964773406322995,
0.03966616920884809, 0.03842666811932978, 0.038990801659444435,
0.03842300062007754 4, 0.037745367589697466, 0.03750614067745406,
0.03790018701480825, 0.03789046453754899, 0.037608406439702716,
0.03806998270969378, 0.03811862357586777, 0.03868982440846717,
0.03910300075628774, 0.03949005776180992, 0.04012145680306807,
0.04069857976331621, 0.0413644421249354, 0.04206784246228376,
0.04290771524475002, 0.04360209882250019, 0.04461385192385548,
0.04577818452302874, 0.0469775181775916, 0.04840666873307037,
0.04996430947546881 6, 0.05156959398368782, 0.053416934729149636,
0.05551909821181478, 0.05772156143795957, 0.05996202226973034,
0.06235048205780537, 0.0649965978135102, 0.06766228123834322,
0.07063034191991144, 0.07330326846121779, 0.07613520963300366,
0.07912792459041562, 0.08219393229059714, 0.08534553699535838,
0.08857236712052589, 0.09187519002545066, 0.09525525792243435,
0.09871534300979679, 0.10226731037661323, 0.10590916217281587]
[0.2, 1.3895467009674967, 2.2738014662154704, 0.38696598597454446,
0.261627628303246, 0.22910402086391812, 0.1982903277164922, 0.18345510091791745,
0.16896722369642442, 0.16184608718744217, 0.15480296396598522,
0.14783501991636977, 0.14093945262276006, 0.1341134785617674,
0.1273543192748639, 0.12065918499085854, 0.11402525618142055,
0.11073034927762546, 0.1074496616114553, 0.10418282185577532,
0.10092945182381677, 0.09768916527759199, 0.09446156658619542,
0.09124624920710354, 0.08804279395778897, 0.0848507670377611,
0.08166971767069296, 0.07849917567023643, 0.0753386481044124,
0.0740771258020528, 0.07281708855170992, 0.07218761580470823,
0.07155850099294944, 0.07030132688351676, 0.06904552904227186,
0.0677910692877178, 0.06653790837166551, 0.06591180240383153,
0.06528600590758633, 0.0640353202930602, 0.06278580862573369,
0.06216147923754208, 0.061537426612083036, 0.06091364497902093,
0.06029012846818863, 0.059666871105989776, 0.059043866811638944,
0.05842110939321971, 0.05779859254357105, 0.05717630983597781,
0.05655425471965284, 0.055932420515020384, 0.055310800408743206,
0.05468938744853293, 0.054068174537688696, 0.05344715442937109,

0.05282631972056639, 0.05220566284577478, 0.051585176070336226,
0.05096485148341927, 0.0503446809906456, 0.04972465630629862,
0.04910476894511071, 0.048485010213599515, 0.047865371200923656,
0.04724584276919486, 0.04662641554324936, 0.04600707989981434,
0.04538782595602976, 0.0447686435572877, 0.044149522264301, 0.04353045133940551,
0.04291141973197652, 0.04229241606291189, 0.041673428608118825,
0.04105444528090599, 0.04043545361318858, 0.03981644073542917,
0.03919739335517057, 0.038578297734071024, 0.037959139663294166,
0.03733990443709627, 0.036720576824469255, 0.03610114103863572,
0.03548158070419719, 0.03486187882172571, 0.03424201772952126,
0.03362197906224922, 0.03300174370617968, 0.032381291750613425,
0.03176060243513708, 0.034418893484908356, 0.03245398497675446,
0.03179908497284544, 0.031143750233062067, 0.0317111601297784,
0.029831655341059266, 0.02917483072987154, 0.029674428603282992,
0.028994192717339433, 0.02917604422069848, 0.028477168208986492,
0.02906126904456418, 0.028851779307115216, 0.028570987106008985,
0.02861258393382031, 0.02912115887295461, 0.029352574297618828,
0.029400529367605427, 0.030001847387056205, 0.030320317285048307,
0.03103582142708862 7, 0.03165531308853193, 0.03227557947273319,
0.03309352545318909, 0.033893261318355815, 0.0347787079003892,
0.03571308212095531, 0.03676675341853044, 0.03773918585653359,
0.038956757442101635, 0.04029912282890568, 0.04168191330916892,
0.043234966371111155, 0.0448836771912742, 0.0465703213333592,
0.048417643101464236, 0.050425157623739794, 0.05248777829864987,
0.05456370228250022, 0.05671773517116127, 0.05901052411200516,
0.061296153217076224, 0.0637375335317641, 0.06598868132681023,
0.06831181657147455, 0.07304988252048904, 0.07547699205337477,
0.08012419299799596, 0.08473998084109453, 0.0892515685916858,
0.09138077636323332, 0.09352892193289863, 0.11212934296072591,
0.12180350943107443]
[1.570141255689622, 2.044828501865063, 0.24048794268874937, 0.2145136949534753,
0.1901448526205381, 0.16716906702811266, 0.14540761577848693,
0.13493344594970957, 0.12470675204935043, 0.11968102529144355,
0.11471104222636472, 0.10979484795048429, 0.10493051700200408,
0.10011614570640985, 0.09534984385033538, 0.09062972538321039,
0.08595389807257214, 0.08363199762514974, 0.08132045132379824,
0.07901901516013635, 0.07672744206128154, 0.07444548127885747,
0.07217287771025983, 0.06990937114088934, 0.06765469539372168,
0.06540857736991594, 0.0631707359606001, 0.06094088080569327,
0.05871871087008719, 0.05783192502178604, 0.05694629816717046,
0.05650391280050110 6, 0.056061809257783264, 0.055178436837376256,
0.05429615901729703, 0.05341495345021127, 0.052534797301995347,
0.05209510548383538, 0.051655667221659096, 0.0507775393091154,
0.049900389080588796, 0.049462172777959675, 0.04902419143144901,
0.04858644179941165, 0.0481489205962299, 0.0477116244909009,
0.04727455010554527, 0.04683769401387826, 0.0464010527396073,
0.04596462275476993, 0.04552840047799975, 0.04509238227273298,
0.04465656444532311, 0.04422094324308736, 0.04378551485227215,

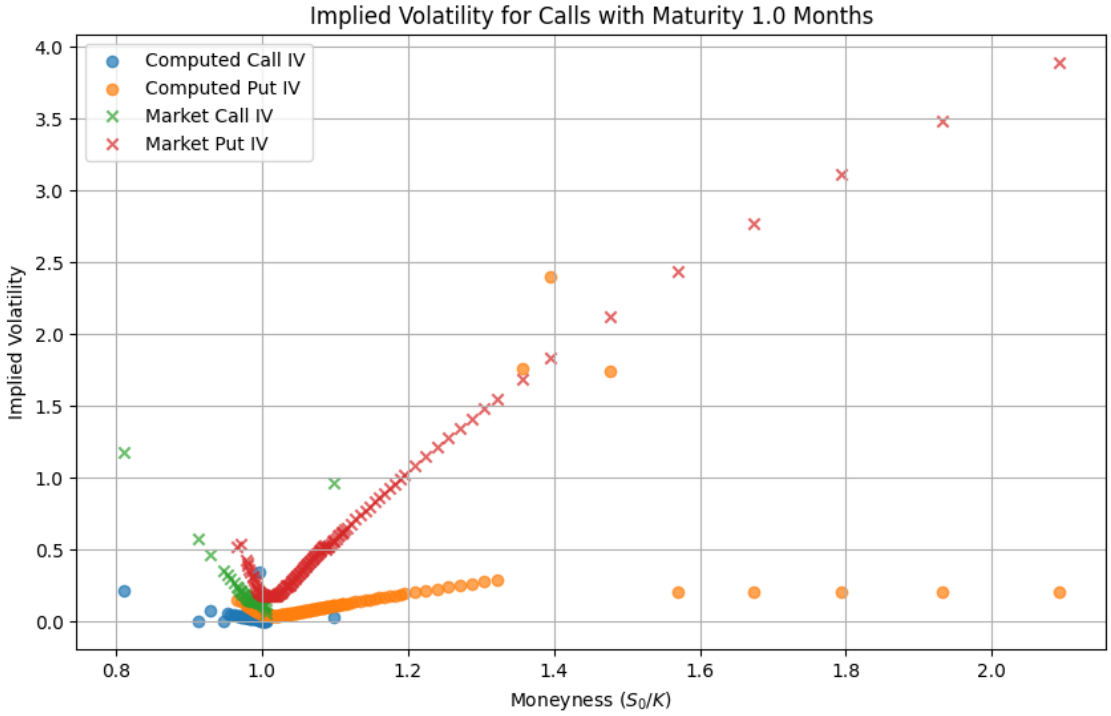
0.043350275395923496, 0.04291522093167146, 0.04248034744941362,
0.042045650868899054, 0.041611127037205445, 0.04117677172609525,
0.04074258062925742, 0.04030854935942035, 0.03987467344533162,
0.039440948328583886, 0.03900736936030884, 0.03857393179769228,
0.03814063080032704, 0.037707461426387055, 0.03727441862860245,
0.03684149725003862, 0.03640869201964963, 0.03597599754760761,
0.03554340832038148, 0.035110918695556596, 0.034678522896366894,
0.0342462150059364, 0.03381398896119736, 0.033381838546461644,
0.03294975738663544, 0.032517738940030703, 0.032085776490758076,
0.03165386314067998, 0.031221991800852715, 0.030790155182471333,
0.0303583457872337, 0.02992655589710277, 0.029494777563418076,
0.02906300259529008 2, 0.02863122254724029, 0.02819942870599129,
0.03058660215774364, 0.029008392718263283, 0.028553984458332876,
0.028099478565858906, 0.028710079379097187, 0.027190126391204474,
0.0267352554169807, 0.027301091016467455, 0.026831049983122777,
0.027126888245076933, 0.02664501633822159, 0.027310124501112948,
0.027273672051359443, 0.02717806211360967, 0.02737743391313132,
0.02800433295874058, 0.028389703569142066, 0.02861750415229223,
0.02935318448623087, 0.029842370684745056, 0.030698686884078195,
0.03147717870292665, 0.032265247149312094, 0.03323914445116682,
0.034204611488123475, 0.03525389602482843, 0.036353266961882055,
0.037563512207993685, 0.03870755430594388, 0.040069055568900516,
0.04153952347718657, 0.043044032385439884, 0.044690736671324845,
0.04641291815639892, 0.04816022457324521, 0.05003195828582156,
0.05202362421928355, 0.054047095564452995, 0.056068109887154494,
0.05813757525560053, 0.0603012952367043, 0.06244426961328156,
0.06469172644487609, 0.06677909565252624, 0.06890764674556112,
0.07319356151754149, 0.07536004807487669, 0.07950519223395891,
0.08358487972978008, 0.08755130963313566, 0.089431620836298,
0.09131770575960835, 0.10748339551104018, 0.11582524540216488]
[0.21816689539439751, 0.1969503119485263, 0.1772797059663382,
0.15893073752870052, 0.14172192757825544, 0.1255037948904541,
0.1101509823117734, 0.1027650026834981, 0.09555630858004235,
0.09201487631530769, 0.08851356128103831, 0.08505103494559531,
0.08162599668784706, 0.07823716962001817, 0.074883296208888,
0.07156313361150965, 0.06827544862231122, 0.06664340113147733,
0.06501901210237752, 0.06340212750284088, 0.06179259272632245,
0.06019025234358347, 0.05859494983360921, 0.0570065272906849,
0.05542482510402785, 0.05384968160579754, 0.05228093268251051,
0.05071841121966238, 0.04916194709624477, 0.04854101858740324,
0.047921020063887404, 0.04761136596846756, 0.04730194005977108,
0.04668376698039258, 0.046066489094905905, 0.045450094528430754,
0.04483457125378506, 0.044527132548622624, 0.04421990708276207,
0.04360608965691406, 0.042993106437814205, 0.04268692368579994,
0.04238094469675477, 0.04207516784806797, 0.04176959150383974,
0.041464214014498485, 0.04115903371641881, 0.04085404893151151,
0.04054925796681965, 0.040244659114088095, 0.039940250649323356,
0.03963603083234171, 0.03933199790630436, 0.03902815009723006,

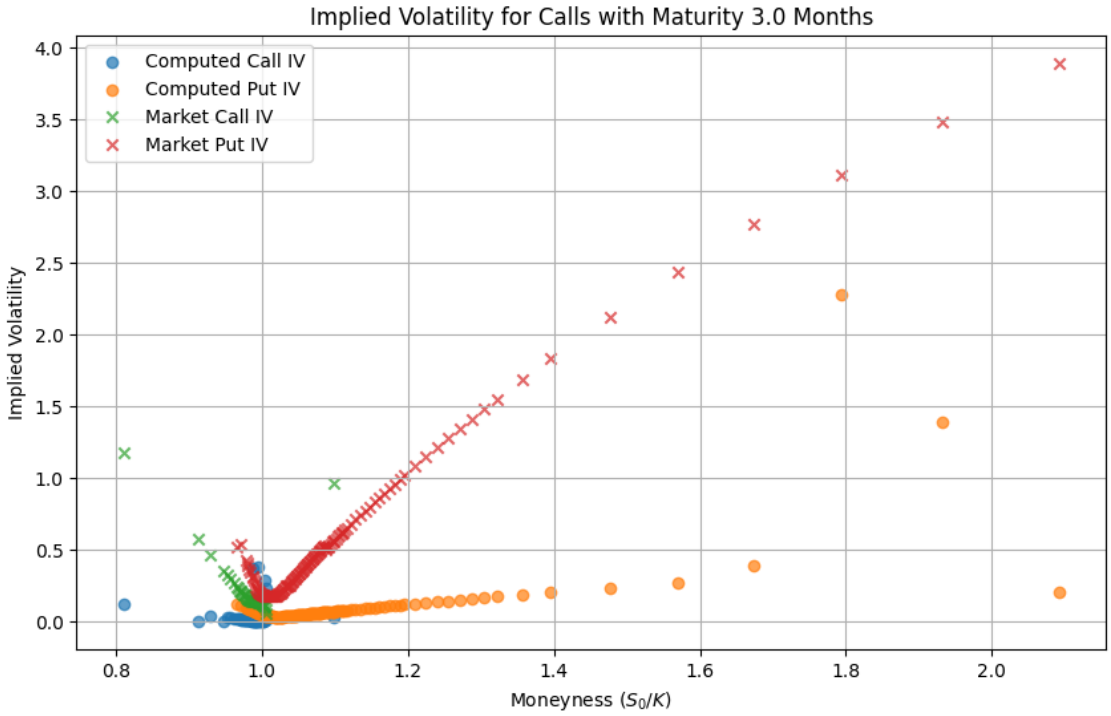
```
0.03872448561349805, 0.038421002645336544, 0.03811769936428943,
0.037814573922663644, 0.037511624452964784, 0.037208849067312824,
0.03690624585682899, 0.036603812891010425, 0.03630154821708155,
0.03599944985931921, 0.03569751581835884, 0.03539574407047019,
0.03509413256681142, 0.034792679232651114, 0.03449138196657059,
0.03419023863962652, 0.03388924709448762, 0.03358840514453756,
0.033287710572942365, 0.03298716113168603, 0.032686754540562614,
0.03238648848613252, 0.03208636062063643, 0.031786368560866286,
0.03148651004878895, 0.031186782297824612, 0.030887182978210637,
0.03058770955270405, 0.030288359442231155, 0.029989130024438374,
0.02969001863219272 4, 0.02939102255200785, 0.02909213902240836,
0.028793365232211102, 0.02849469831874086, 0.028196135365956526,
0.027897673402494792, 0.030261523610209493, 0.028890285250592597,
0.02857717329409384, 0.028264135909740428, 0.028970460281770544,
0.027638269962538636, 0.027325433749057444, 0.028002769173844855,
0.027680298578143945, 0.02810487114052598, 0.027775046657927516,
0.028569692385051478, 0.028683211742933652, 0.02874258392540549,
0.0290962540479004, 0.02987576416245189, 0.03042325811846954,
0.030821406748252657, 0.03172408876032133, 0.032390001124415535,
0.03342015417866904, 0.03437807715743393, 0.03534909549544418,
0.03650296232585885, 0.03765041012191544, 0.0388792737800831,
0.0401558663605554, 0.04153616665526423, 0.04285227384697924,
0.04436826222681217, 0.04597932084599158, 0.04761444812444576,
0.04937026507825555, 0.051183550986097476, 0.05300792213358269,
0.054929903836373924, 0.05694268377548203, 0.05896793039315967,
0.060976206716834846, 0.0630109959781731, 0.06510999134149198,
0.06717646746821207, 0.06931433087185929, 0.07130655486110833,
0.07331950878101871, 0.0773311803772059, 0.07933779869967492,
0.08316990715010797, 0.08691207742625638, 0.09053197010953068,
0.09225146033351547, 0.09396843903678846, 0.10854582749243505,
0.11600532363057166]
```

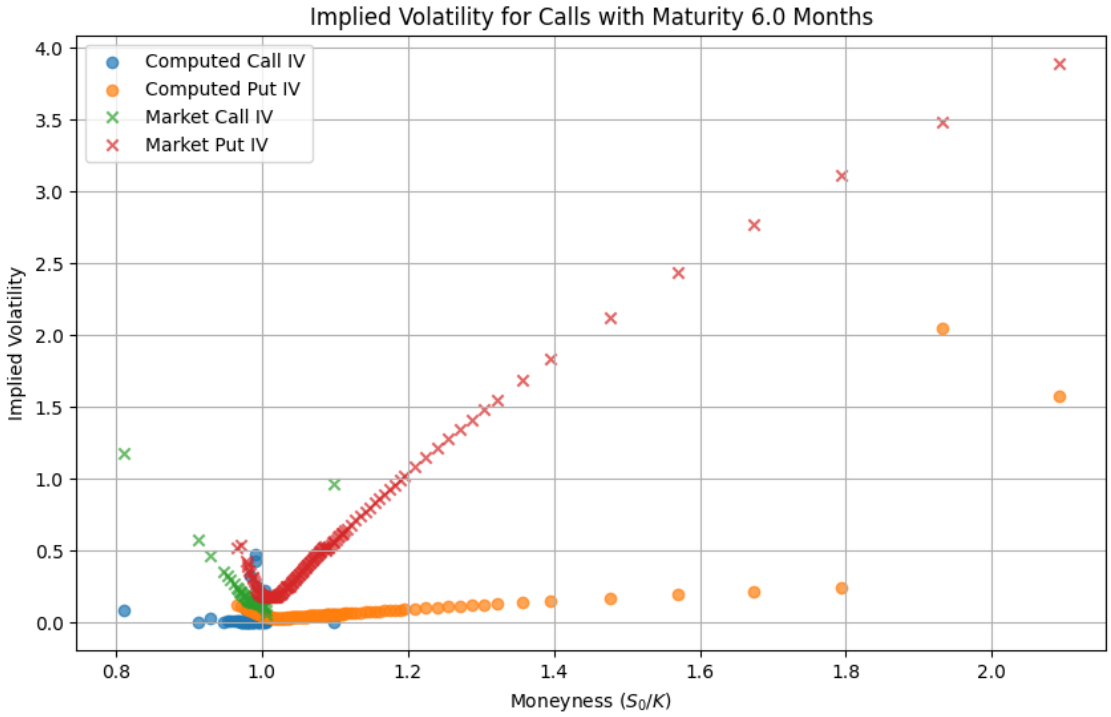
[ ]:
```
# Assuming the spot price (S_0), risk-free rate (rfr), and maturities are
 ↪defined

# Calculate the forward price for each maturity
F = {tau: spot * np.exp(rfr * tau) for tau in maturities}

# Add a column for moneyness to calls and puts DataFrames
calls['Moneyness'] = spot / calls['Strike']
puts['Moneyness'] = spot / puts['Strike']

import matplotlib.pyplot as plt

for tau in maturities:
    # Plotting Computed IVs
    plt.figure(figsize=(10, 6))
```

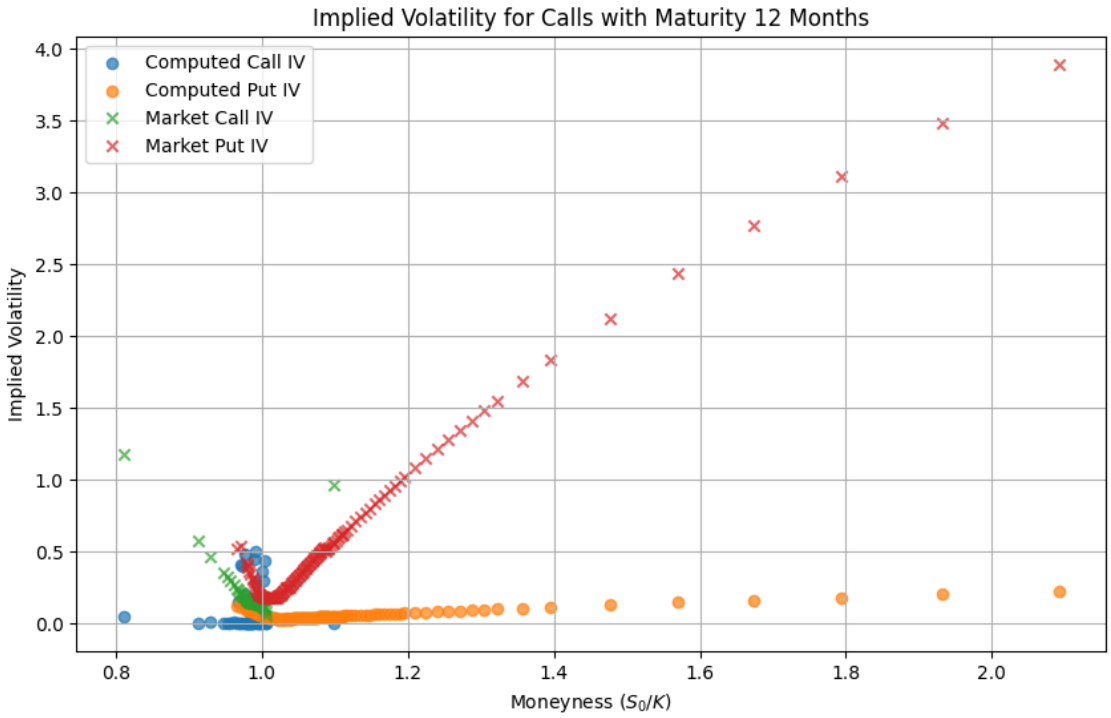
```
    plt.scatter(calls['Moneyness'], maturity_lists[tau], label='Computed Call␣
↪IV', alpha=0.7)
    plt.scatter(puts['Moneyness'], maturity_lists_puts[tau], label='Computed␣
↪Put IV', alpha=0.7)

    # Plotting Market IVs (assuming calls dataframe has a 'Market IV' column)
    plt.scatter(calls['Moneyness'], calls['Implied Volatility'], label='Market␣
↪Call IV', alpha=0.7, marker='x')
    plt.scatter(puts['Moneyness'], puts['Implied Volatility'], label='Market␣
↪Put IV', alpha=0.7, marker='x')

    plt.title(f'Implied Volatility for Calls with Maturity {tau*12} Months')
    plt.xlabel('Moneyness ($S_0 / K$)')
    plt.ylabel('Implied Volatility')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
C:\Users\sbhatia2\AppData\Local\Temp\ipykernel_9408\1096201001.py:7:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  calls['Moneyness'] = spot / calls['Strike']
```




Implied Volatility for Calls with Maturity 3.0 Months
Computed Call IV
Computed Put IV
Market Call IV
Market Put IV
Implied Volatility
Moneyness (S_0/K)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.8
1.0
1.2
1.4
1.6
1.8
2.0


Implied Volatility for Calls with Maturity 6.0 Months
Computed Call IV
Computed Put IV
Market Call IV
Market Put IV
Implied Volatility
Moneyness (S_0/K)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.8
1.0
1.2
1.4
1.6
1.8
2.0



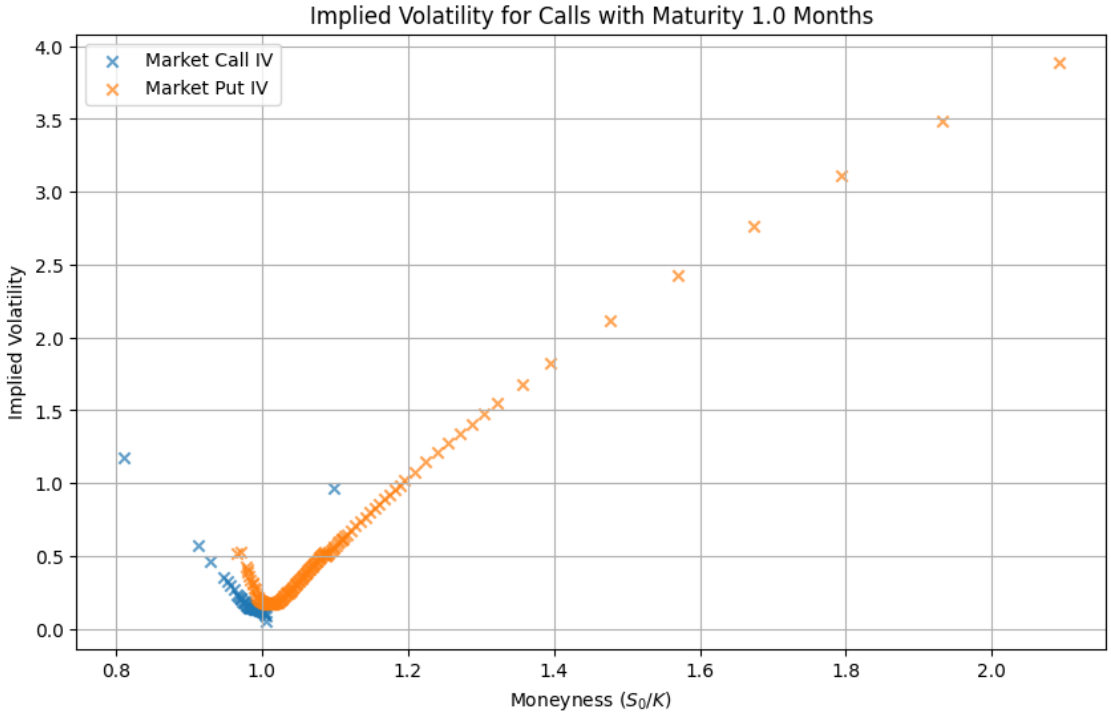
As one can see, there were some significant differences with the computed and actual IV for each of these options. The reason why there's such a stark difference is these are **daily options** with a **maturity of one day.** As such, these options will expire tomorrow and not 1/3/6/12 months from now. However, the computed IVs were computed with the assumption of τ being much larger; however, the **yahoo_fin** library did not have options with longer maturities without zero volatility due to data problems (NaN values).

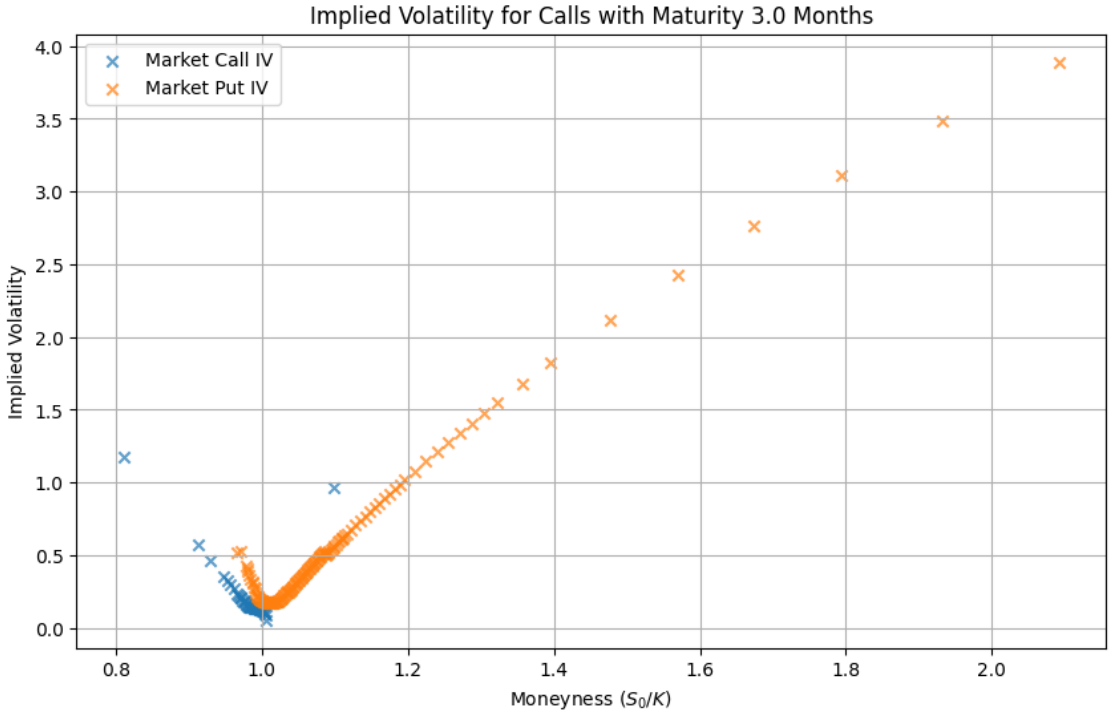
4. Generate an implied volatility surface:

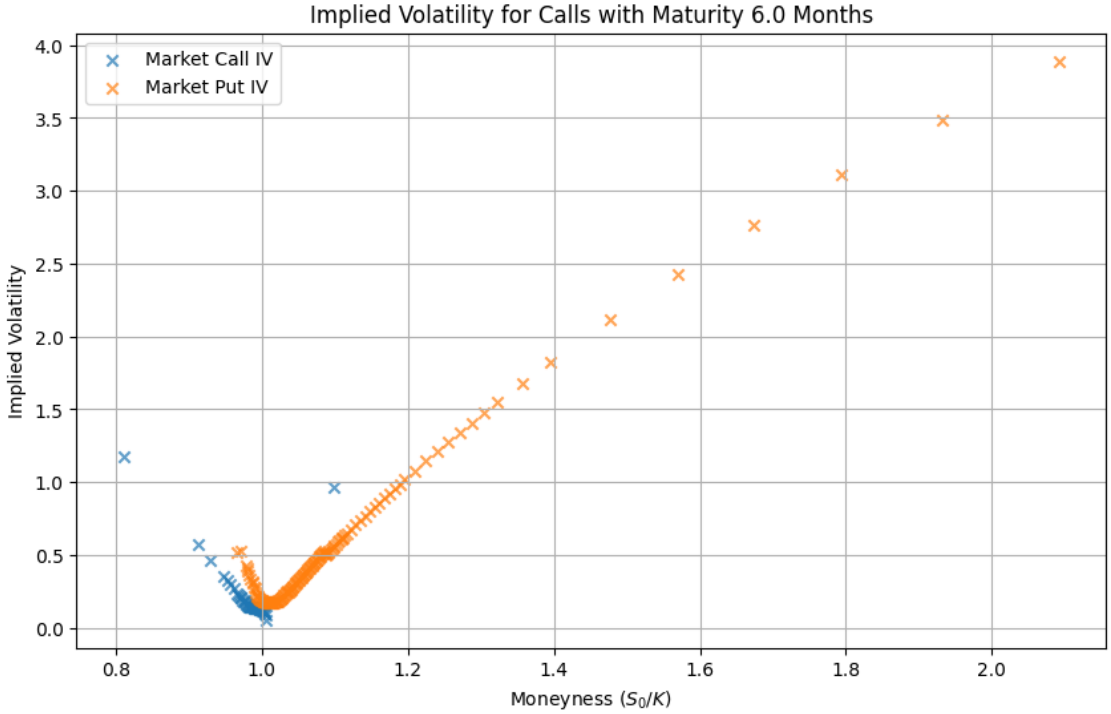
- In a single plot, display the implied volatility smiles for all the maturities considered in part (c). You may use either your computed volatilities or the downloaded market volatilities.
- *Note: Do not use a three-dimensional surface. Create a two-dimensional plot with strike (or moneyness) on the x-axis, implied volatility on the y-axis, and with each smile labeled by its time to maturity.*
- Comment on how the implied volatility depends on time-to-maturity. For a fixed maturity, comment on the dependence of implied volatility on strike (or moneyness).

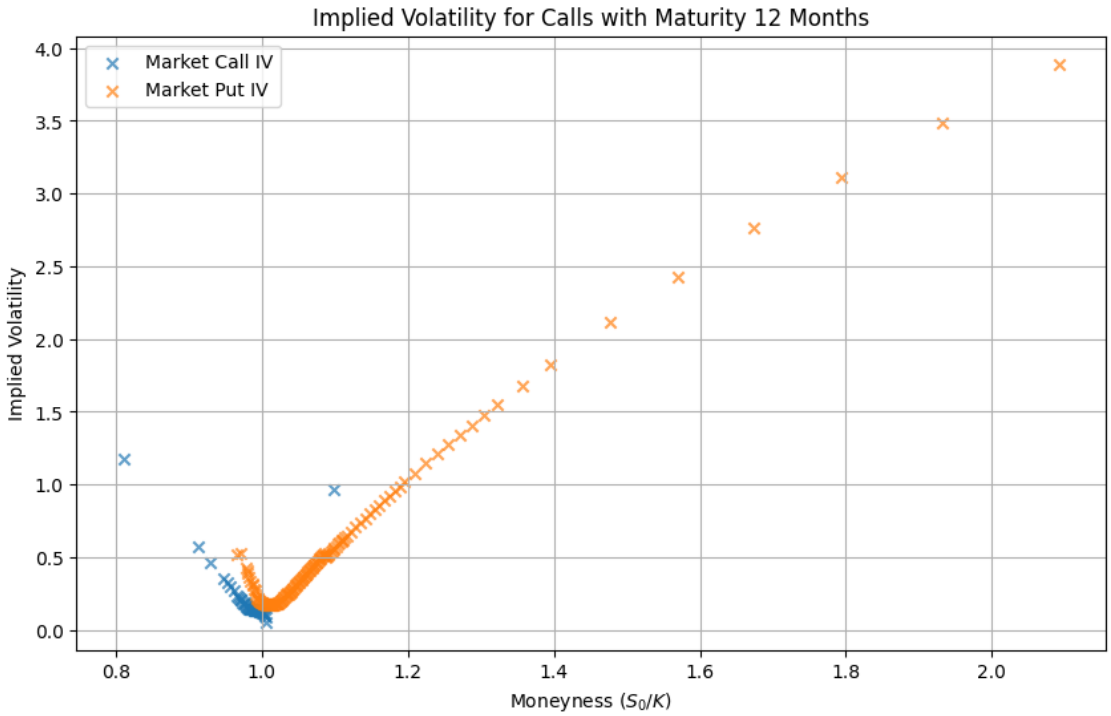
[ ]:
```
for tau in maturities:
    # Plotting Computed IVs
    plt.figure(figsize=(10, 6))

    # Plotting Market IVs (assuming calls dataframe has a 'Market IV' column)
    plt.scatter(calls['Moneyness'], calls['Implied Volatility'], label='Market␣
 ↪Call IV', alpha=0.7, marker='x')
```

```
    plt.scatter(puts['Moneyness'], puts['Implied Volatility'], label='Market␣
↪Put IV', alpha=0.7, marker='x')

    plt.title(f'Implied Volatility for Calls with Maturity {tau*12} Months')
    plt.xlabel('Moneyness ($S_0 / K$)')
    plt.ylabel('Implied Volatility')
    plt.legend()
    plt.grid(True)
    plt.show()
```




Implied Volatility for Calls with Maturity 3.0 Months
Market Call IV
Market Put IV
Implied Volatility
Moneyness (S_0/K)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.8
1.0
1.2
1.4
1.6
1.8
2.0


Implied Volatility for Calls with Maturity 6.0 Months
Market Call IV
Market Put IV
Implied Volatility
Moneyness (S_0/K)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.8
1.0
1.2
1.4
1.6
1.8
2.0



Generally, implied volatility tends to increase for options that are deep in-the-money or out-of-the-money, showing the well-known "smile" or "smirk" pattern. This reflects the market's expectation of larger movements in the underlying asset's price, especially for strikes far from the current spot price. The shape and level of the implied volatility curve can vary with time to maturity, reflecting different market conditions, expectations, and pricing dynamics over different horizons.

5. Test the put-call parity:

- The put-call parity is a no-arbitrage relation between the prices of call and put options. Show theoretically that the put-call parity implies that call and put options with the same strike and same time to maturity must have the same implied volatility.
- Do the call and put implied volatility smiles in part (c) coincide? If not, are such violations of the put-call parity arbitrage opportunities?

Put-Call Parity and Implied Volatility

The put-call parity is a fundamental principle in options pricing that establishes a relationship between the prices of European call and put options with the same strike price and time to maturity. The put-call parity can be expressed as:

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

where: - C is the price of the European call option, - P is the price of the European put option, - S_0 is the current price of the underlying asset, - K is the strike price, - r is the risk-free interest rate, and - T is the time to maturity (in years).

Implications for Implied Volatility

The put-call parity implies that if two options are on the same underlying asset, have the same strike price, and the same time to maturity, then any difference in their prices is due to the cost of carrying the underlying asset. This relationship holds under the assumption of no arbitrage opportunities and certain other conditions such as no transaction costs.

Since the implied volatility (σ) of an option is derived from its market price, the put-call parity suggests that for European call and put options with the same strike and maturity, any difference in their implied volatilities would suggest a deviation from the put-call parity. However, in practice, the Black-Scholes model and other pricing models assume that calls and puts with the same strike and maturity should have the same implied volatility because these models are based on the assumption of no arbitrage.

Do Call and Put Implied Volatility Smiles Coincide?

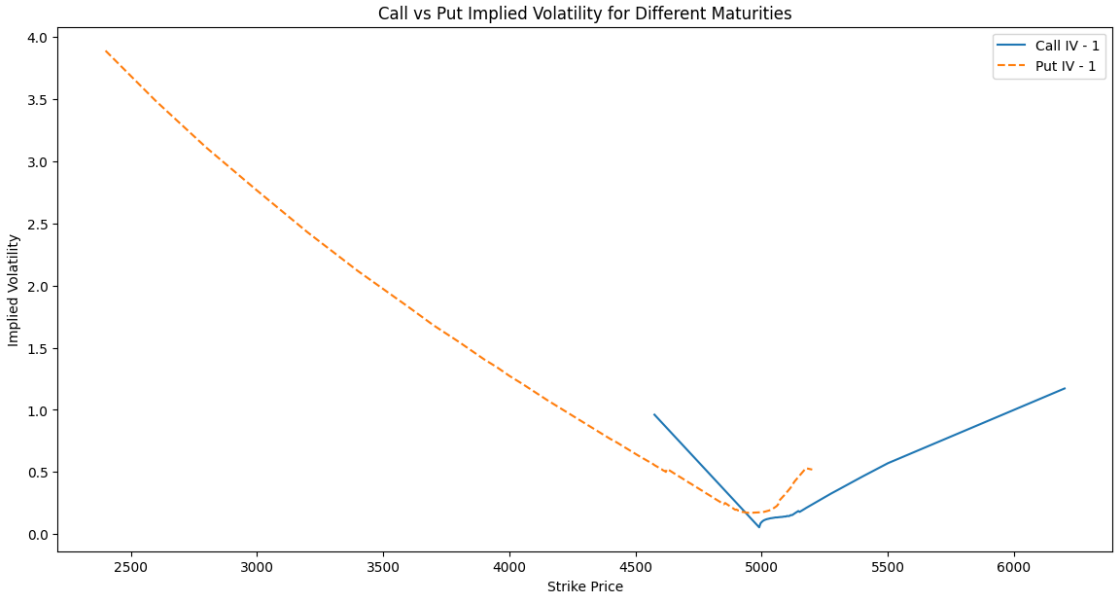
In real markets, the implied volatility smiles for calls and puts may not perfectly coincide due to several factors, including demand and supply dynamics, market sentiment, transaction costs, and the impact of dividends. These factors can cause slight deviations from the theoretical put-call parity.

Are Violations of Put-Call Parity Arbitrage Opportunities?

Not necessarily. While significant deviations from put-call parity might suggest potential arbitrage opportunities, executing an arbitrage strategy requires buying and selling options and possibly the underlying asset, which incurs transaction costs. Moreover, the presence of dividends and the risk of early exercise (for American options) can also affect the practical application of put-call parity for arbitrage.

In summary, while the put-call parity provides a theoretical foundation for the equivalence of call and put implied volatilities under certain conditions, real-world factors can lead to discrepancies. Traders and analysts must carefully consider these factors when assessing potential arbitrage opportunities.

[ ]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(14, 7))
plt.plot(calls['Strike'], calls['Implied Volatility'], label=f'Call IV -␣
 ↪{maturity}')
plt.plot(puts['Strike'], puts['Implied Volatility'], label=f'Put IV -␣
 ↪{maturity}', linestyle='--')

plt.title('Call vs Put Implied Volatility for Different Maturities')
plt.xlabel('Strike Price')
plt.ylabel('Implied Volatility')
plt.legend()
plt.show()
```



6. Compute Greeks:

- For the same maturities as in part (c), compute the Black-Scholes delta and gamma of both call and put options. Note that closed-form formulas can be used to compute the delta and gamma of call and put options in the Black-Scholes model.
- Comment on the shapes of the delta and gamma curves for both call and put options, and how they depend on strike (or moneyness) and time to maturity.

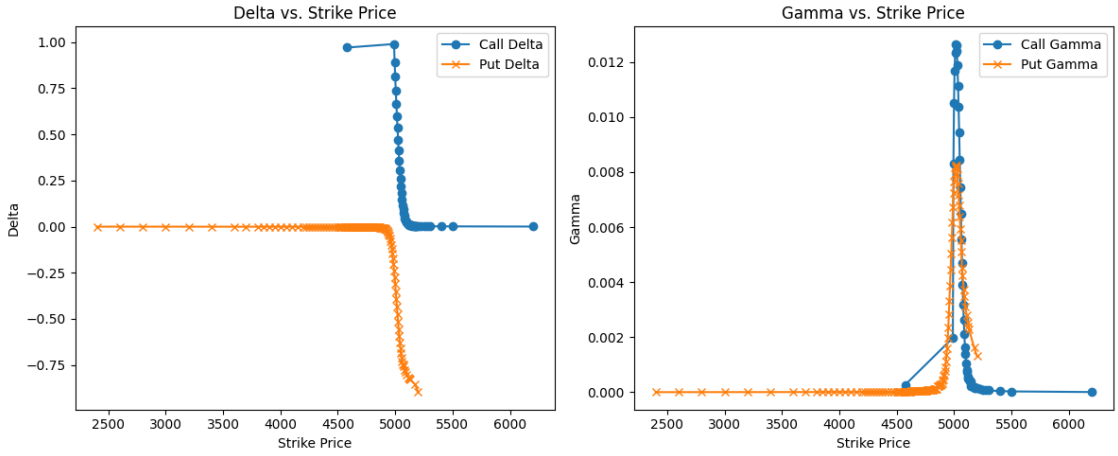
[ ]:
```python
def black_scholes_greeks(S, K, r, tau, sigma, option_type='call'):
    """
    Compute the Black-Scholes delta and gamma for call and put options.

    Parameters:
    - S (float): Spot price of the underlying asset.
    - K (float): Strike price of the option.
    - r (float): Risk-free interest rate.
    - tau (float): Time to maturity (in years).
    - sigma (float): Volatility of the underlying asset.
    - option_type (str): Type of the option ('call' or 'put').

    Returns:
    - delta (float): The delta of the option.
    - gamma (float): The gamma of the option.
    """
    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * tau) / (sigma * np.sqrt(tau))
    d2 = d1 - sigma * np.sqrt(tau)

    if option_type == 'call':
```

```
        delta = si.norm.cdf(d1)
    else:  # option_type == 'put'
        delta = si.norm.cdf(d1) - 1

    gamma = si.norm.pdf(d1) / (S * sigma * np.sqrt(tau))

    return delta, gamma

tau = 1/365

# Compute deltas and gammas for calls and puts.
call_deltas, call_gammas, put_deltas, put_gammas = [], [], [], []

for idx, strike in call_strikes.items():
    sigma = calls['Implied Volatility'].loc[idx]
    call_delta, call_gamma = black_scholes_greeks(spot, strike, rfr, tau,␣
 ↪sigma, 'call')
    call_deltas.append(call_delta)
    call_gammas.append(call_gamma)

for idx, strike in put_strikes.items():
    sigma = puts['Implied Volatility'].loc[idx]
    put_delta, put_gamma = black_scholes_greeks(spot, strike, rfr, tau, sigma,␣
 ↪'put')
    put_deltas.append(put_delta)
    put_gammas.append(put_gamma)
```

```
[0.9700101686662616, 0.9894012241645962, 0.8914801866128144, 0.8131104654095596,
0.7371053921119961, 0.666628353353579, 0.5993380556095206, 0.5340046078121525,
0.47135953712394163, 0.41183153131677097, 0.3567799951634141,
0.3043374003787733, 0.257716995643713, 0.21607656591630076, 0.18028425544610827,
0.14490828155444296, 0.11754665130205616, 0.0940389793919395,
0.07480442670175294, 0.05768799635510935, 0.04550415631902214,
0.03581047343720023, 0.026275132291394693, 0.02235893157834422,
0.01573810626212388, 0.012001862279197442, 0.01143518844960004,
0.00800195041187069, 0.007637014676008174, 0.007376965956479378,
0.00717510554095232, 0.006835681653408286, 0.006683403108742251,
0.0036689428264289033, 0.0034322775929453964, 0.0032393986064991375,
0.003193959936318985, 0.0030921092657085767, 0.0029033882940384804,
0.002744938642593773, 0.0024485874143310637, 0.0022434136741515632,
0.002096550086872403, 0.0018879930114503256, 0.001480579521835589,
0.0012351651901832746, 0.0009336016897001054]
```

[ ]:
```
# Plotting Delta and Gamma
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
```

```
plt.plot(call_strikes, call_deltas, label='Call Delta', marker='o')
plt.plot(put_strikes, put_deltas, label='Put Delta', marker='x')
plt.title('Delta vs. Strike Price')
plt.xlabel('Strike Price')
plt.ylabel('Delta')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(call_strikes, call_gammas, label='Call Gamma', marker='o')
plt.plot(put_strikes, put_gammas, label='Put Gamma', marker='x')
plt.title('Gamma vs. Strike Price')
plt.xlabel('Strike Price')
plt.ylabel('Gamma')
plt.legend()

plt.tight_layout()
plt.show()
```



In the Black-Scholes model, the delta and gamma of both call and put options exhibit distinctive behaviors based on strike price (or moneyness) and time to maturity. Delta, which measures the rate of change in the option's price with respect to changes in the underlying asset's price, varies between 0 and 1 for calls and -1 and 0 for puts. For calls, delta increases as the option moves from out-of-the-money (OTM) to in-the-money (ITM), indicating a higher sensitivity to movements in the underlying price for ITM options. Puts exhibit the opposite trend, with delta becoming more negative as the option transitions from OTM to ITM.

Gamma, representing the rate of change in delta with respect to changes in the underlying's price, peaks for options near the money and diminishes for options far from the money. This indicates that the sensitivity of the option's delta to the underlying's price is highest when the strike price is close to the underlying's current price. Gamma is symmetrical for calls and puts, reflecting the same level of convexity in their delta changes.

Both delta and gamma are also influenced by the time to maturity. As maturity approaches, the delta of ITM calls (and OTM puts) approaches 1 (or -1 for puts), and the delta of OTM calls (and ITM puts) approaches 0, reflecting the increasing certainty of the option's finishing in or out of the money. Gamma tends to increase as maturity decreases for options near the money, underscoring the heightened sensitivity of delta to price changes as the option nears expiration.

In summary, the shape of the delta curve flattens for options far from the money and steepens for options near the money, with the steepness accentuated as expiration approaches. The gamma curve is most pronounced near the money, indicating significant changes in delta for small movements in the underlying's price, especially as the option approaches maturity.

Problem #2 - Root Finding Algorithms

1. Write pseudocode for using the bisection method and Newton's method to iteratively compute the square root of a positive number a. Remember to state how you choose the initial search interval for the bisection method, and the initial guess for Newton's method.

- *Hint: Finding the square root of a is equivalent to finding the positive root of the function* $f(x) = x^2 - a$.

Bisection Method Pseudocode

The bisection method requires selecting an initial interval $[x_0, x_1]$ that contains the root.

1. *Initial Interval Selection*:
   - Choose `x_0 = 0` and `x_1 = a` if `a >= 1` or `x_1 = 1` if `a < 1`. This ensures the square root of `a` is within the interval `[x_0, x_1]`.
2. *Check Midpoint*:
   - Compute the midpoint `m = (x_0 + x_1) / 2` and evaluate `f(m) = m^2 - a`.
3. *Update Interval*:
   - If `f(m) = 0`, then `m` is the square root, and the process stops.
   - If `f(m) * f(x_0) < 0`, the root is in the interval `[x_0, m]`. Set `x_1 = m`.
   - Otherwise, the root is in the interval `[m, x_1]`. Set `x_0 = m`.
4. *Convergence Check*:
   - Repeat steps 2 and 3 until the interval `[x_0, x_1]` is sufficiently small, e.g., `|x_1 - x_0| < epsilon`, where `epsilon` is a small number representing the desired accuracy.
5. *Result*:
   - The square root of `a` is approximately `(x_0 + x_1) / 2` after convergence.

Newton's Method Pseudocode

1. *Initial Guess*:
   - Choose `x_0 = a / 2` if `a >= 1` or `x_0 = 1` if `a < 1`. This is a reasonable starting point for most positive numbers `a`.
2. *Iteration*:
   - Apply the Newton iteration formula to refine the guess: `x_{n+1} = x_n - (f(x_n) / f'(x_n)) = x_n - ((x_n^2 - a) / (2x_n)) = (x_n + (a / x_n)) / 2`.
3. *Convergence Check*:
   - Repeat step 2 until the change in successive estimates is below a threshold, e.g., `|x_{n+1} - x_n| < epsilon`, where `epsilon` is the desired precision.
4. *Result*:

- The square root of `a` is `x_n` when the process converges.

2. Implement your procedures in part (a) with tolerance $\epsilon = 10^{-6}$.
   1. How many iterations do the algorithms need for $a = 0.5$ and $a = 2$?
   2. To visualize the convergence, plot $|err_n|$ as a function of n ($|err_n|$ should converge to zero), and $\log(|err_n|)$ as a function of $\log(n)$, where $err_n = \sqrt{a} - x_n$ is the error on the n-th iteration of the algorithm.
   3. Which algorithm seems to converge faster? Is what you observe in line with theoretical results about the rate of convergence of the two methods?

2bi.

[ ]:
```python
def sqrt_bisection(a: float, epsilon: float = 1e-6) -> float:
    if a < 0:
        raise ValueError("a must be non-negative")
    elif a == 0 or a == 1:
        return a

    lower, upper = (0, a) if a > 1 else (a, 1)

    while upper - lower > epsilon:
        midpoint = (lower + upper) / 2
        if midpoint**2 == a:
            return midpoint
        elif midpoint**2 < a:
            lower = midpoint
        else:
            upper = midpoint

    return (lower + upper) / 2

print(sqrt_bisection(0.5))
print(sqrt_bisection(2))
```

```
0.7071070671081543
1.4142136573791504
```

[ ]:
```python
def sqrt_newton(a: float, epsilon: float = 1e-6) -> float:
    if a < 0:
        raise ValueError("a must be non-negative")
    elif a == 0 or a == 1:
        return a

    x_n = a / 2  # Initial guess
    while True:
        x_next = 0.5 * (x_n + a / x_n)
        if abs(x_next - x_n) < epsilon:
            return x_next
        x_n = x_next
```

```
print(sqrt_newton(0.5))
print(sqrt_newton(2))
```

```
0.7071067811865475
1.414213562373095
```

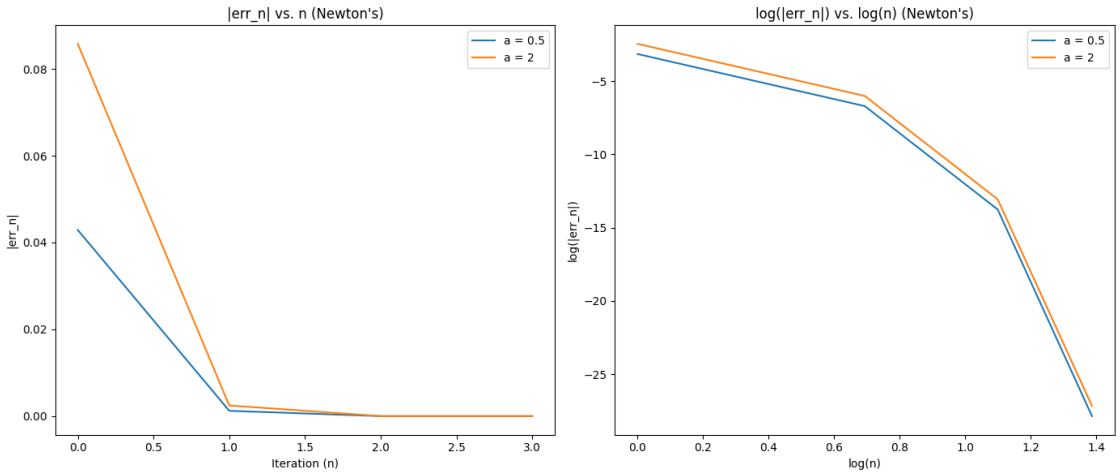
[ ]:
```
def sqrt_bisection_iterations(a: float, epsilon: float = 1e-6) -> int:
    if a == 0 or a == 1:
        return 0  # No iterations needed
    lower, upper = (0, a) if a > 1 else (a, 1)
    iterations = 0
    while upper - lower > epsilon:
        midpoint = (lower + upper) / 2
        if midpoint**2 < a:
            lower = midpoint
        else:
            upper = midpoint
        iterations += 1
    return iterations

def sqrt_newton_iterations(a: float, epsilon: float = 1e-6) -> int:
    if a == 0 or a == 1:
        return 0  # No iterations needed
    x_n = a / 2
    iterations = 0
    while True:
        x_next = 0.5 * (x_n + a / x_n)
        if abs(x_next - x_n) < epsilon:
            break
        x_n = x_next
        iterations += 1
    return iterations

# Calculate and print the number of iterations for each method and value of a.
for a in [0.5, 2]:
    bisection_iters = sqrt_bisection_iterations(a)
    newton_iters = sqrt_newton_iterations(a)
    print(f"For a = {a}: Bisection method iterations = {bisection_iters},␣
 ↪Newton's method iterations = {newton_iters}")
```

```
For a = 0.5: Bisection method iterations = 19, Newton's method iterations = 5
For a = 2: Bisection method iterations = 21, Newton's method iterations = 4
```

2bii.

[ ]:
```
def sqrt_newton_with_errors(a: float, epsilon: float = 1e-6):
    if a <= 0:
        raise ValueError("a must be positive")
```

```
    x_n = a if a < 1 else a / 2  # Initial guess
    errors = []  # To track errors

    while True:
        x_next = 0.5 * (x_n + a / x_n)
        err_n = np.abs(np.sqrt(a) - x_next)
        errors.append(err_n)
        if err_n < epsilon:
            break
        x_n = x_next

    return errors

# Compute errors for a = 0.5 and a = 2
errors_05 = sqrt_newton_with_errors(0.5)
errors_2 = sqrt_newton_with_errors(2)

# Plot |err_n| as a function of n
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
plt.plot(errors_05, label='a = 0.5')
plt.plot(errors_2, label='a = 2')
plt.xlabel('Iteration (n)')
plt.ylabel('|err_n|')
plt.title("|err_n| vs. n (Newton's)")
plt.legend()

# Plot log(|err_n|) vs log(n)
plt.subplot(1, 2, 2)
n_05 = np.arange(1, len(errors_05) + 1)
n_2 = np.arange(1, len(errors_2) + 1)
plt.plot(np.log(n_05), np.log(errors_05), label='a = 0.5')
plt.plot(np.log(n_2), np.log(errors_2), label='a = 2')
plt.xlabel('log(n)')
plt.ylabel('log(|err_n|)')
plt.title("log(|err_n|) vs. log(n) (Newton's)")
plt.legend()

plt.tight_layout()
plt.show()
```



2biii. When comparing the convergence rates of Newton's method and the bisection method for finding roots, Newton's method generally converges faster than the bisection method under ideal conditions. This is because Newton's method is a form of open method that uses the function's derivative to iteratively estimate the root, potentially doubling the number of correct digits at each step (quadratic convergence) when close to the root and the function behaves well. In contrast, the bisection method is a bracketing method that linearly narrows down the interval containing the root by repeatedly bisecting it and selecting the subinterval that contains the root, leading to linear convergence.

The faster convergence of Newton's method comes with caveats, such as the requirement for the initial guess to be relatively close to the actual root and the function to be reasonably well-behaved (continuous and differentiable) near the root. If these conditions are not met, Newton's method may fail to converge or may converge to a wrong root. The bisection method, while slower, is guaranteed to converge as long as the function changes sign over the interval and is continuous, making it a more robust choice in certain situations.

The observed convergence rates in practical applications should align with these theoretical expectations. Newton's method should converge more rapidly to the root in well-conditioned scenarios, offering a significant advantage in computational efficiency. However, its performance can vary significantly if the initial guess is poor or the function does not meet the necessary smoothness criteria. The bisection method, while inherently slower due to its linear convergence rate, provides a more consistent and reliable convergence behavior, especially in less ideal conditions.

In summary, while Newton's method generally converges faster and is preferred for its efficiency when applicable, the bisection method offers a reliable alternative that guarantees convergence under broader conditions, aligning with theoretical understandings of their respective convergence rates.

3. *Extra credit:* Show theoretically that Newton's method has *quadratic order of convergence*. Specifically, denote by x^* a solution of $f(x) = 0$ for some "nice" function f, and let $\epsilon_n = |x_n - x^*|$ be the error on the n-th iteration of Newton's algorithm. Show that

$$\epsilon_{n+1} \leq K\epsilon_n^2,$$

for some constant $K > 0$, which is the definition of quadratic order of convergence.

- Remark: For comparison, the error of the bisection method satisfies

$$\epsilon_{n+1} \leq \frac{1}{2}\epsilon_n,$$

where $\epsilon_n = |x_n - x^*|$ is the error on the n-th iteration, and $x_n = (b_n - a_n)/2$ is the half-width of the n-th interval (a_n, b_n). This is called linear order of convergence.

To demonstrate that Newton's method has quadratic order of convergence, we consider the Newton iteration formula and apply Taylor expansion of $f(x)$ around a root x^*, assuming f is twice differentiable in the neighborhood of x^*.

The iteration formula for Newton's method is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Let $\epsilon_n = |x_n - x^*|$ denote the error on the n-th iteration. Our aim is to show that $\epsilon_{n+1} \leq K\epsilon_n^2$ for some constant $K > 0$.

Taylor's theorem allows us to expand $f(x)$ around x^*:

$$f(x_n) = f(x^*) + f'(x^*)(x_n - x^*) + \frac{f''(\xi_n)}{2}(x_n - x^*)^2$$

where ξ_n is some point between x_n and x^*. Given x^* is a root of $f(x) = 0$, this simplifies to:

$$f(x_n) = f'(x^*)\epsilon_n + \frac{f''(\xi_n)}{2}\epsilon_n^2$$

Substituting the Taylor expansion back into Newton's iteration formula yields:

$$x_{n+1} = x_n - \frac{f'(x^*)\epsilon_n + \frac{f''(\xi_n)}{2}\epsilon_n^2}{f'(x_n)}$$

Assuming $f'(x^*) \neq 0$ and that $f'(x_n)$ approximates $f'(x^*)$ closely as n increases, we get:

$$\epsilon_{n+1} = |x_{n+1} - x^*| \approx \left| -\frac{f''(\xi_n)}{2f'(x^*)}\epsilon_n^2 \right|$$

This demonstrates that ϵ_{n+1} is proportional to ϵ_n^2, indicative of quadratic convergence. Here, $K = \left|\frac{f''(\xi_n)}{2f'(x^*)}\right|$ is a constant, showcasing the efficiency of Newton's method for well-behaved functions, as it implies the precision roughly doubles with each iteration.