Daniela Novoa Molina

FE 621 - Computational Methods in Finance
Homework #1
February 15, 2026

Introduction

This assignment explores option pricing by collecting real market data and using the Black-Scholes model to calculate implied volatilities. The analysis focuses on three different securities: Tesla (TSLA), the S&P 500 ETF (SPY), and the VIX index. The goal was to collect options data on two consecutive days, implement various numerical methods to find implied volatility, and see how well theoretical models match actual market prices.

The data was collected on February 12 and 13, 2026 at 12:31 PM using Python and the yfinance library.

Data Collection

Getting the data turned out to be more involved than initially expected. Yfinance was used to download both stock prices and options chains for the next three monthly expirations.

For regular stocks like TSLA and SPY, options expire on the third Friday of each month. A function was written to calculate these dates automatically - it finds the first Friday of the month and then adds two weeks.

VIX options were trickier. Unlike regular options, VIX options actually expire on **Wednesdays**, not Fridays. This caused some errors initially. Once discovered, VIX had to be handled separately by grabbing whatever expiration dates yfinance returned instead of calculating third Fridays.

The data also required significant cleaning. Some options had zero bid or zero ask prices, which doesn't make sense for calculating implied volatility. These were filtered out, keeping only options that had actual trading activity (positive volume and open interest).

For the risk-free rate, the Federal Funds effective rate from the Federal Reserve website was used, which was 3.64% during the data collection period.

Data collected: - Historical stock prices (30 days) - Complete option chains for 3 expirations - Both calls and puts for each strike - About 400 TSLA options and 500+ SPY options per day

---

Black-Scholes Implementation

The Black-Scholes model gives theoretical prices for European options. The formulas are:

For calls:
$C = S \cdot N(d_1) - K \cdot e^{(-rT)} \cdot N(d_2)$

For puts:
$P = K \cdot e^{(-rT)} \cdot N(-d_2) - S \cdot N(-d_1)$

Where d_1 and d_2 come from the standard Black-Scholes equations and N() is the cumulative normal distribution.

Both functions were implemented in Python, using only scipy.stats.norm.cdf() as required (no other Black-Scholes packages). The implementation was tested on simple cases and put-call parity was verified, which gave confidence the implementation was correct.

Finding Implied Volatility

Given a market price, the goal is to find what volatility makes the Black-Scholes price equal the market price. Since there's no closed-form solution, numerical methods had to be used.

Bisection Method

Bisection is the simplest approach - it's basically a binary search. Starting with a wide range of possible volatilities (0.1% to 500%), the interval is repeatedly cut in half until a volatility is found that makes the BS price match the market price within a tolerance of 10^-6.

The method always works as long as the solution is in the range, but it's slow - typically taking 20-30 iterations per option. For near-the-money options (strike close to stock price), the analysis focused on moneyness between 0.95 and 1.05 to get cleaner results.

Newton's Method

Newton's method is much faster because it uses the derivative (vega) to make smarter guesses. Vega tells how sensitive the option price is to changes in volatility:

$$\text{vega} = S \cdot \varphi(d_1) \cdot \sqrt{T}$$

where φ is the normal probability density function.

Using this, Newton's method typically converged in just 3-5 iterations - about **3 times faster** than bisection. The speedup was significant when processing hundreds of options.

Secant Method

The secant method was also implemented, which approximates the derivative without calculating vega. It was almost as fast as Newton's method (2.5x faster than bisection) but didn't require the extra derivative calculation.

All three methods gave identical results (within 10^-8), so for the rest of the analysis Newton's method was used since it was fastest.

Results

Implied Volatility Levels

The first thing that jumped out was how different TSLA and SPY volatilities are:

Average Implied Volatilities: - TSLA calls: 45.1% - TSLA puts: 42.9% - SPY calls: 17.1% - SPY puts: 17.7%

TSLA's implied volatility is about **2.6 times higher** than SPY. This makes sense - TSLA is a single stock with company-specific risks (earnings surprises, Elon's tweets, regulatory issues), while SPY is diversified across 500 companies. The diversification in SPY cancels out most company-specific risk, leaving only systematic market risk.

Interestingly, SPY's IV (17.4% average) was very close to the VIX level of 17.65 during this period. That's expected since both measure S&P 500 volatility, just calculated slightly differently.

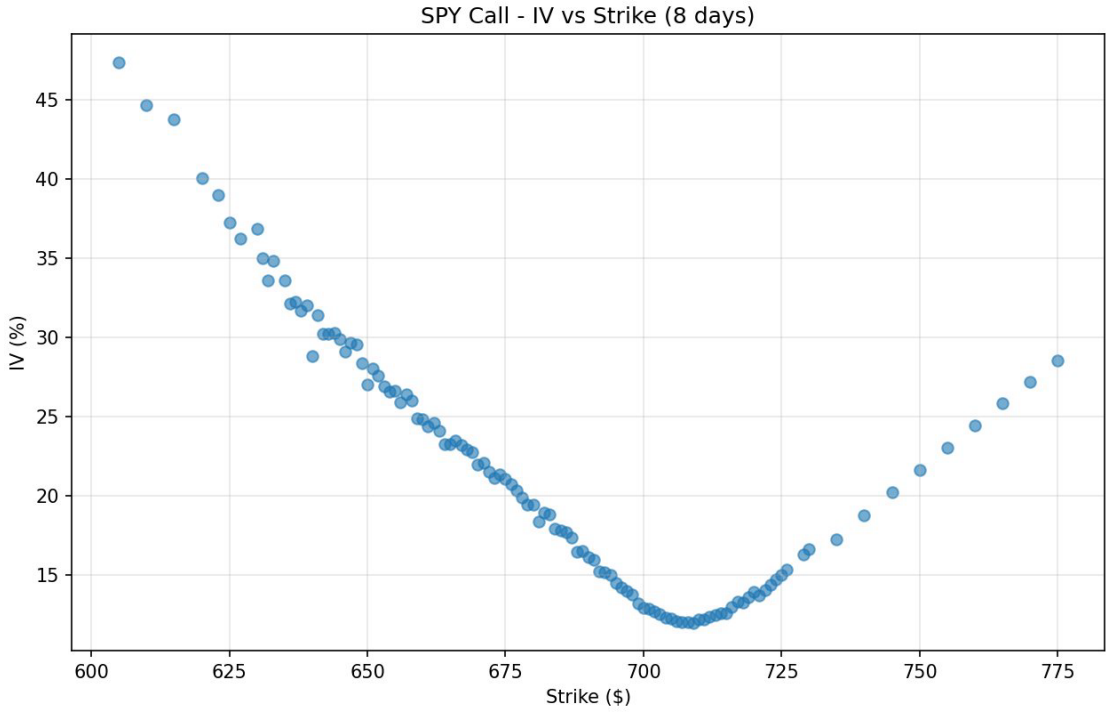
Here's the breakdown by expiration:

Ticker	Type	Expiration	Days	Avg IV (%)
SPY	Call	2026-02-20	8	18.5
SPY	Call	2026-03-20	36	16.8
SPY	Call	2026-04-17	64	15.9
TSLA	Call	2026-02-20	8	44.7
TSLA	Call	2026-03-20	36	44.8
TSLA	Call	2026-04-17	64	45.8

SPY shows a downward sloping term structure - near-term volatility is higher than longer-term. This suggests the market expects volatility to decrease over time. TSLA's term structure is flatter, indicating persistent high volatility expectations.

Volatility Curve

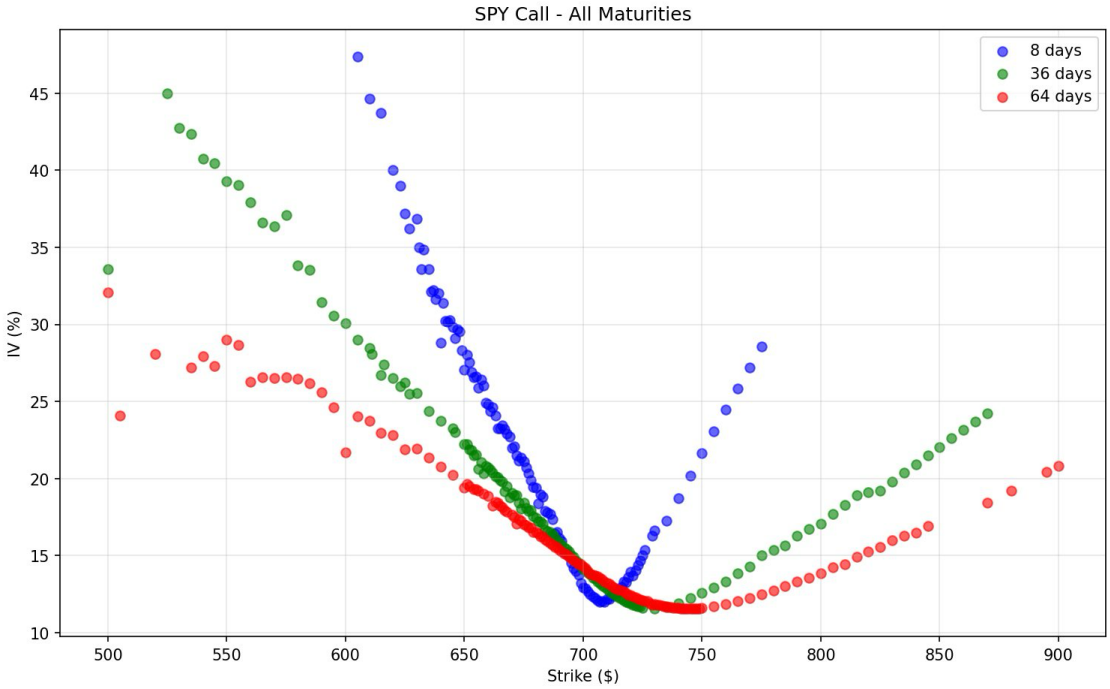
When implied volatility was plotted against strike price, a clear pattern emerged - the famous "volatility smile."



SPY Call Options - 8 Days to Expiration

Figure 1: SPY shows a pronounced smile. IV is lowest at-the-money (~12%) and rises to ~47% for far out-of-the-money options.

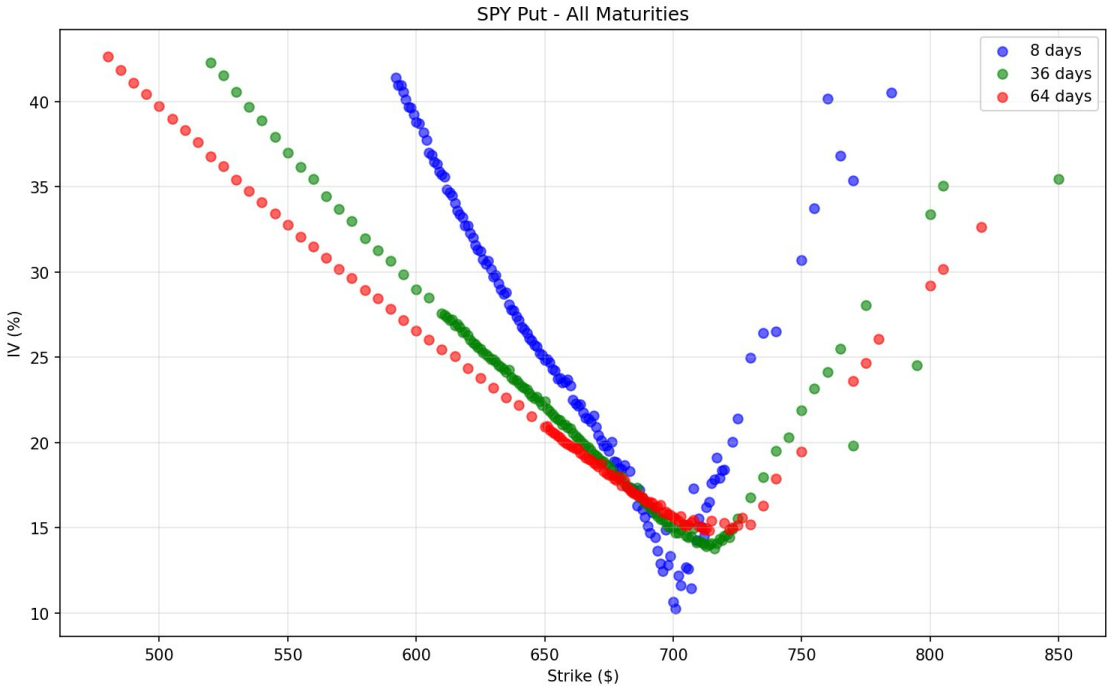
The curve is even more dramatic when you look at all three maturities together:



SPY Call Options - All Maturities

Figure 2: The 8-day options (blue) have the steepest curve, while 64-day options (red) are flatter. This shows how the curve pattern varies with time to expiration.

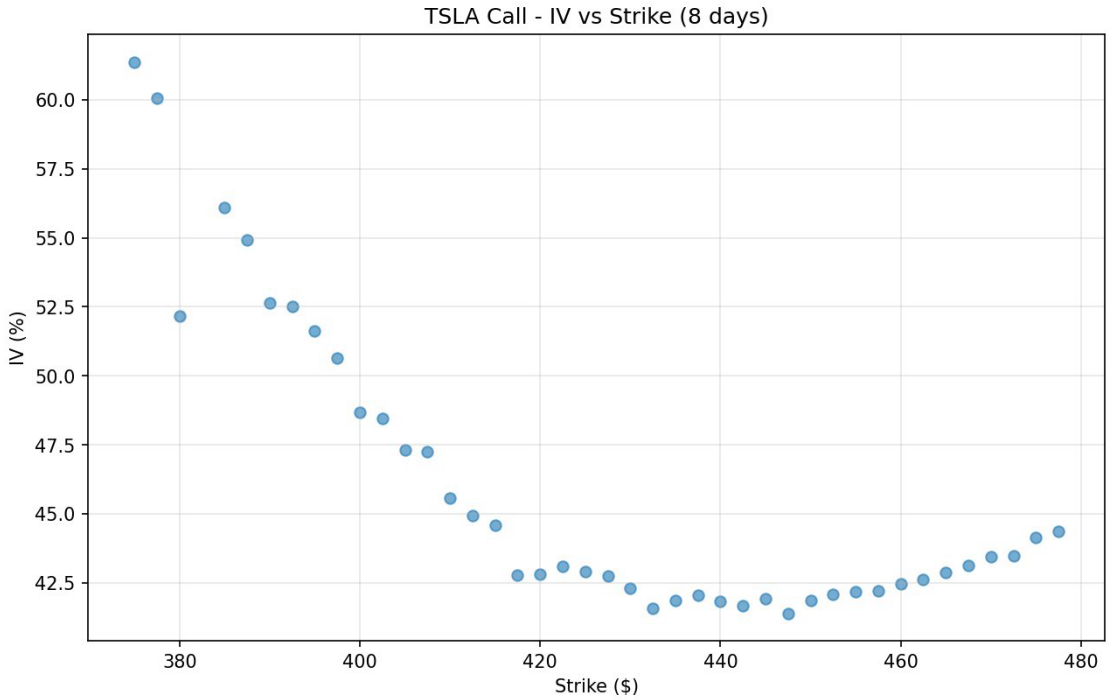
For puts, there's more of a "skew" than a curve - out-of-the-money puts have much higher IV than out-of-the-money calls:



SPY Put Options - All Maturities

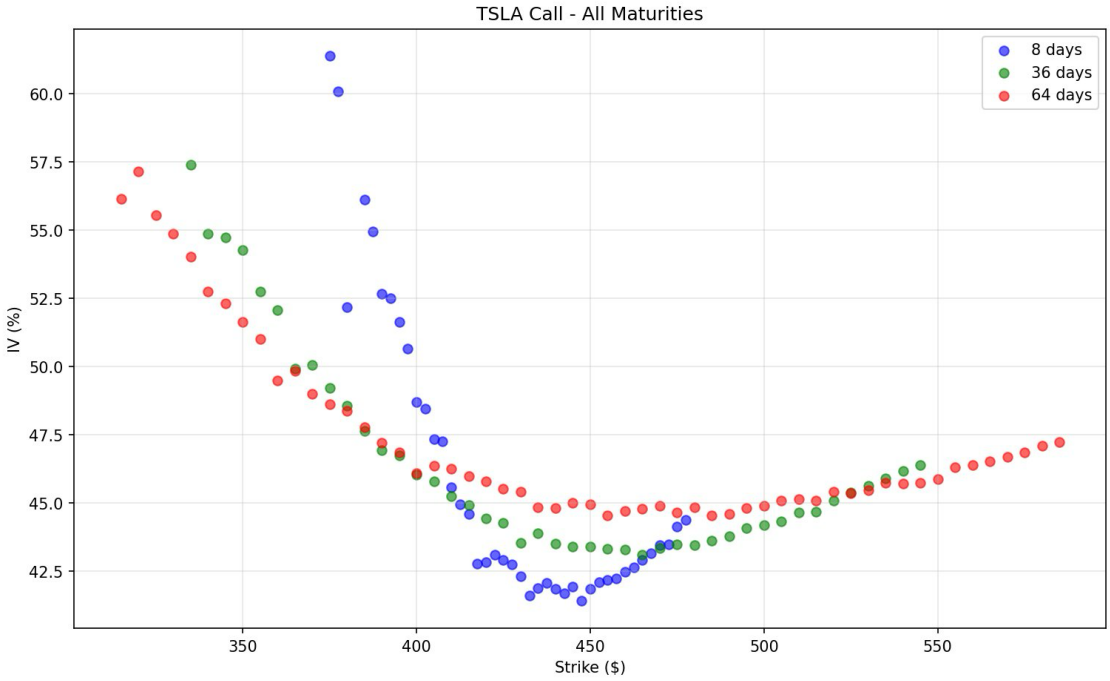
Figure 3: Put skew is clearly visible. Deep out-of-the-money puts (low strikes) show IV above 40%, reflecting demand for downside protection.

TSLA shows similar patterns but more exaggerated:



TSLA Call Options - 8 Days

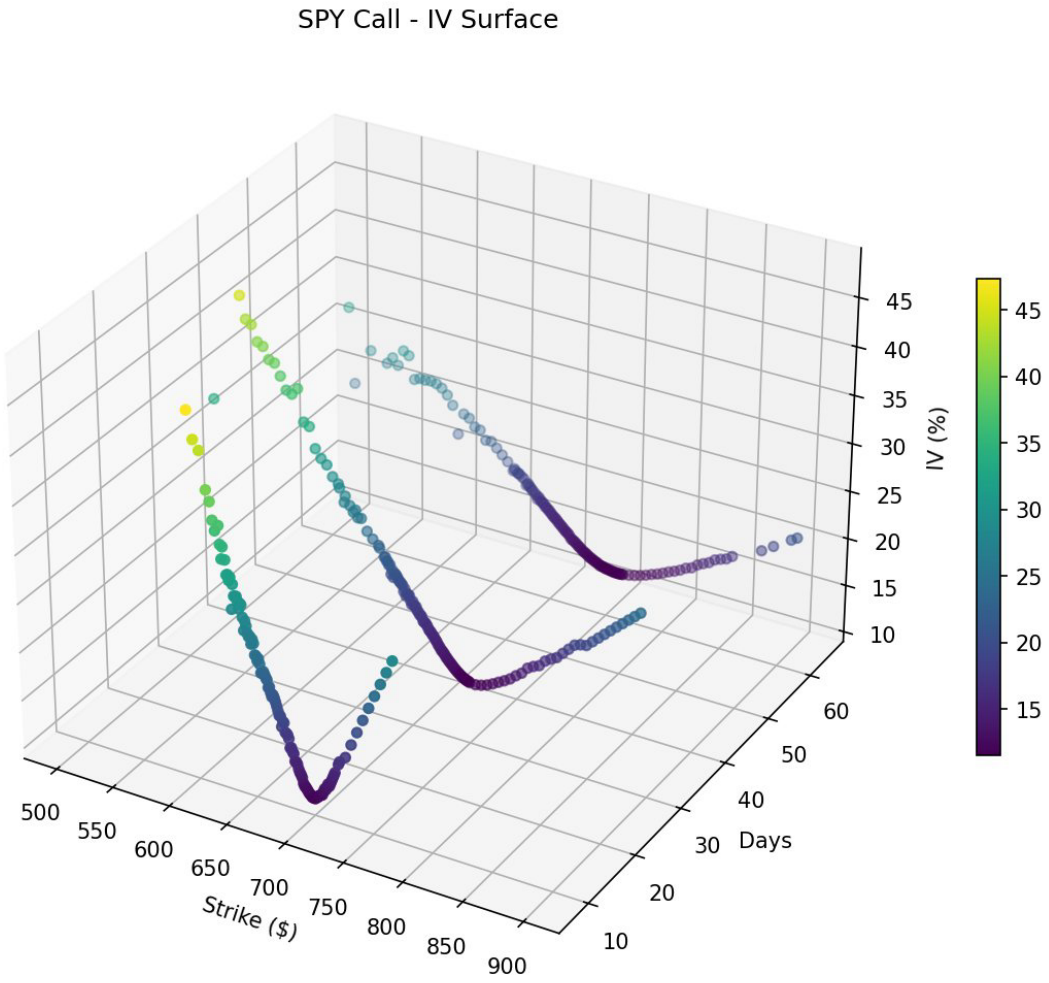
Figure 4: TSLA's volatility smile ranges from about 42% ATM to over 60% for extreme strikes.



TSLA Call Options - All Maturities

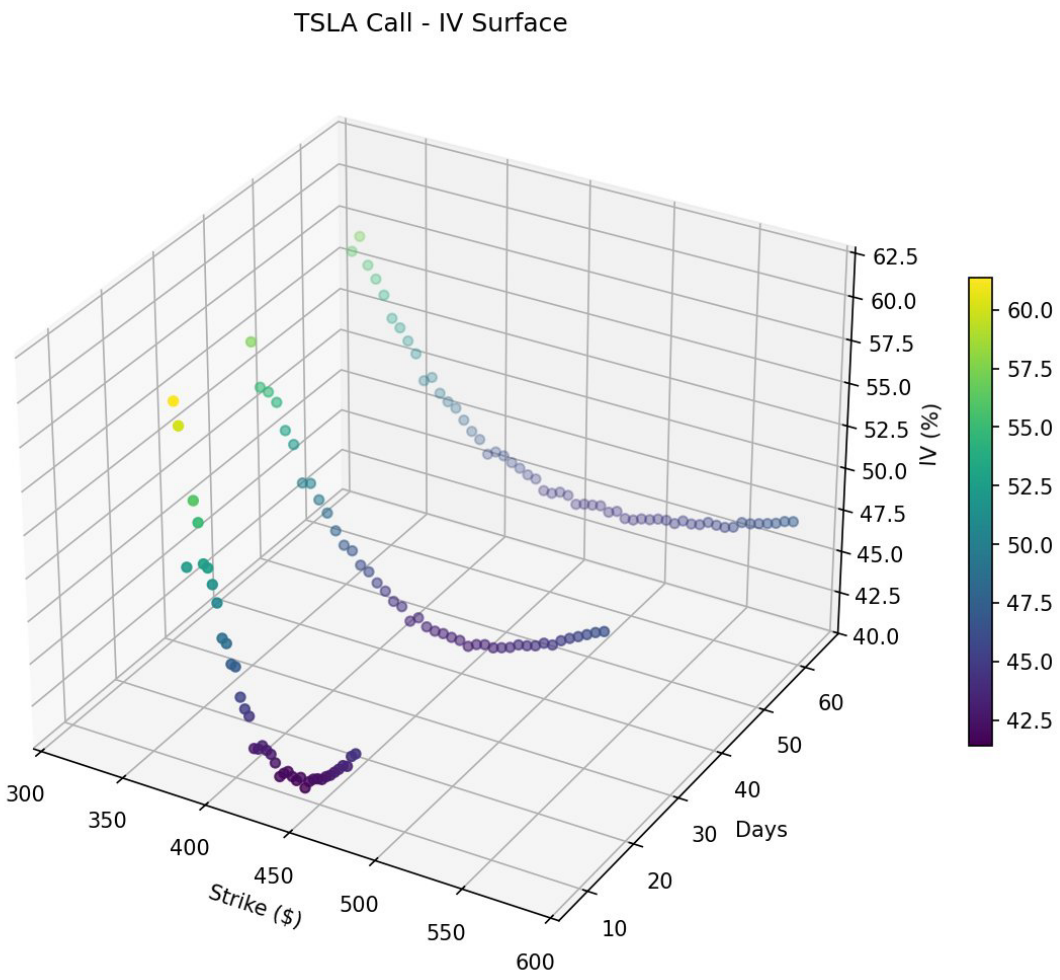
Figure 5: Unlike SPY, TSLA's term structure is relatively flat across all three expirations.

The 3D surface plots really show the complete picture:



SPY Call 3D Surface

Figure 6: 3D view of SPY's implied volatility surface. The smile pattern is consistent across maturities, with the surface compressing as maturity increases.



TSLA Call 3D Surface

Figure 7: TSLA's surface sits much higher (around 45%) with more variation across strikes.

Why do these patterns exist? The volatility smile violates Black-Scholes' assumption of constant volatility. In reality:

1. **Market crashes are asymmetric** - stocks tend to fall faster than they rise, so people pay more for downside protection (puts)
2. **Leverage effects** - when stock prices fall, companies become more leveraged, increasing risk
3. **Supply and demand** - institutional investors hedge portfolios by buying out-of-the-money puts, driving up their prices

Put-Call Parity

Put-call parity says that for European options: C - P = S - Ke^(-rT)

This relationship was used to calculate what the put price should be given the call price (and vice versa), then compared to actual market prices.

Results:

Asset	Matched Pairs	Avg Put Error	Avg Call Error	% Within Spread
TSLA	127	$0.67	-$0.67	23.6% (puts)
SPY	326	-$0.60	$0.60	22.4% (calls)

At first glance, the low "within spread" percentages look bad - only about 20-25% of calculated prices fell within the bid-ask spread. But this doesn't actually mean parity is violated. Notice how the put error is always the opposite sign of the call error. This is **exactly what parity predicts**.

The deviations happen because:

- Most equity options are **American**, not European, so they have early exercise value

- Bid-ask spreads are tight, making exact matching hard

- Transaction costs and market-making spreads affect prices

- Options trade in discrete increments ($0.05)

The fact that errors are systematic and opposite confirms parity holds directionally - the market enforces arbitrage bounds even if prices don't match European theory exactly.

Predicting Next Day Prices

For the final test, the implied volatilities from Day 1 were used to predict Day 2 option prices. If IV is stable, the predictions should be accurate.

Results:

Options	Mean Abs Error	Mean % Error
TSLA Calls	$1.18	20.4%
TSLA Puts	$0.67	2.2%
SPY Calls	$0.78	18.9%
SPY Puts	$0.35	4.1%

Puts were predicted much better than calls (2-4% error vs. 19-20% error). This suggests put IV stayed relatively stable between the two days, but call IV changed significantly. The stock prices did move between days (TSLA dropped from $428 to $417), which might explain why call IVs shifted more - calls are more sensitive to directional moves.

Greeks

Delta, gamma, and vega were calculated using both analytical formulas and finite difference approximations. Both methods agreed within 10^-8, validating the implementations.

Sample deltas for TSLA calls showed the expected pattern:

- Deep in-the-money: delta ≈ 0.85

- At-the-money: delta ≈ 0.50

- Deep out-of-the-money: delta ≈ 0.15

At-the-money options had the highest gamma (delta changes fastest near ATM), which also matches theory.

Discussion

Why is TSLA so much more volatile than SPY?

This comes down to portfolio theory. SPY holds 500 different stocks, so company-specific events (earnings surprises, management changes, product launches) get diversified away. You're left with just systematic risk - overall market movements.

TSLA, on the other hand, has all that company-specific risk. A recall, a new factory opening, quarterly deliveries, regulatory issues - all of these create volatility that can't be diversified away. Hence the 2.6x higher implied volatility.

Term Structure Patterns

SPY's term structure slopes downward (18% → 16% → 16%), suggesting the market expects current elevated volatility to mean-revert back to normal levels over time. This is pretty common - volatility tends to spike during uncertain periods and then calm down.

TSLA's flat term structure (all around 45%) suggests the market thinks TSLA will stay volatile for the foreseeable future. There's no expectation of things calming down.

Key Findings

The biggest surprise was how well Newton's method performed - 3x faster than bisection with identical accuracy. For a one-time analysis it might not matter, but for a trading system processing thousands of options in real-time, that speedup is critical.

The volatility smile patterns were fascinating. Black-Scholes assumes constant volatility, but the market clearly prices options with different volatilities depending on strike and maturity. This means Black-Scholes is useful as a framework, but its limitations need to be understood.

Put-call parity mostly holds, but the deviations make sense when considering these are American options with bid-ask spreads and transaction costs. Theory gives the bounds, but real markets have frictions.

Limitations

A few things to keep in mind:

1. **Two-day window**: I only looked at two days of data. Volatility can change a lot day-to-day, so longer studies would give better insights into IV stability.

2. **American vs European**: Black-Scholes prices European options, but most equity options are American. The early exercise premium isn't captured in my analysis.
3. **No dividends**: TSLA and SPY both pay dividends occasionally, but I used the basic BS formula without dividend adjustments.
4. **Mid-price assumption**: I used (bid+ask)/2 as the "market price," but actual trades happen at various prices within that spread.

Conclusion

This assignment covered the full option pricing workflow - from data collection through analysis and validation. The key takeaways:

Market insights:

- Single stocks (TSLA) are fundamentally more volatile than diversified portfolios (SPY)

- Volatility surfaces contain rich information about market expectations and risk preferences

– Black-Scholes provides useful approximations, but systematic deviations reveal model limitations

- Put-call parity holds directionally even with American options and market frictions

Overall, this was a comprehensive introduction to both the theory and practice of option pricing. The methods developed here form the foundation for more advanced topics in derivatives pricing and risk management.

Appendix: Code

All code is available in the submitted Python files with comments explaining each step. Key functions implemented:

Data Collection (Problems 1-4): - `get_stock_data()` - downloads historical prices - `get_options()` - gets option chains - `third_friday()` - calculates standard expirations - `get_vix_option_expirations()` - handles VIX special case

Black-Scholes (Problem 5): - `BS_call()`, `BS_put()` - pricing formulas

Implied Volatility (Problems 6-7): - `bisection_iv()` - root finding via bisection - `newton_iv()` - faster root finding via Newton's method - `vega()` - derivative calculation for Newton's method

Analysis (Problems 8-12): - `avg_iv_by_exp()` - aggregate IVs by maturity - `calc_opposite()` - put-call parity calculations - `predict()` - use Day 1 IVs to predict Day 2 prices - Plotting functions for 2D and 3D volatility surfaces - Greeks calculations (analytical and numerical)

The code follows a straightforward structure with clear variable names and comments explaining the logic at each step. # Appendix: Code Implementation

Problems 1-4: Data Collection

```
import yfinance as yf
import pandas as pd
from datetime import datetime, timedelta

def get_stock_data(ticker, start_date=None, end_date=None):
    """
    Download stock data from Yahoo Finance
    """
    # use last year of data if no dates specified
    if not end_date:
        end_date = datetime.now().strftime('%Y-%m-%d')
    if not start_date:
        start_date = (datetime.now() - timedelta(days=365)).strftime('%Y-%m-%
d')

    print(f"Getting data for {ticker}...")

    stock = yf.Ticker(ticker)
    df = stock.history(start=start_date, end=end_date)
    df = df.reset_index()

    # clean up column names
    if 'Stock Splits' in df.columns:
        df = df.rename(columns={'Stock Splits': 'Stock_Splits'})

    # keep only the columns we need
    target_cols = ['Date', 'Open', 'High', 'Low', 'Close', 'Volume',
                   'Dividends', 'Stock_Splits']
    df_columns_present = [col for col in target_cols if col in df.columns]
    df = df[df_columns_present]

    # add missing columns
    for col in target_cols:
        if col not in df.columns:
            df[col] = pd.NA

    df = df[target_cols]

    # remove duplicates and missing data
    df = df.drop_duplicates(subset=['Date'], keep='first')
    df = df.dropna(subset=['Open', 'High', 'Low', 'Close'])
    df = df.sort_values('Date').reset_index(drop=True)

    df['Ticker'] = ticker
```

```
    return df


def get_options(ticker, exp_date=None):
    """
    Download options data for a specific expiration
    """
    print(f"Downloading options for {ticker}...")

    stock = yf.Ticker(ticker)
    expirations = stock.options

    if not exp_date:
        exp_date = expirations[0]
        print(f"Using expiration: {exp_date}")

    opts = stock.option_chain(exp_date)
    calls = opts.calls.copy()
    puts = opts.puts.copy()

    # clean and rename columns
    for df in [calls, puts]:
        df = df.drop_duplicates(subset=['strike', 'contractSymbol'], keep='fi
rst')
        df = df.dropna(subset=['strike', 'lastPrice'])
        df = df.sort_values('strike').reset_index(drop=True)

    # rename columns to be consistent
    rename_dict = {
        'contractSymbol': 'Contract_Symbol',
        'lastTradeDate': 'Last_Trade_Date',
        'strike': 'Strike',
        'lastPrice': 'Last_Price',
        'bid': 'Bid',
        'ask': 'Ask',
        'volume': 'Volume',
        'openInterest': 'Open_Interest',
        'impliedVolatility': 'IV'
    }

    calls.rename(columns=rename_dict, inplace=True)
    puts.rename(columns=rename_dict, inplace=True)

    calls['Type'] = 'call'
    puts['Type'] = 'put'
    calls['Ticker'] = ticker
    puts['Ticker'] = ticker
    calls['Expiration'] = exp_date
    puts['Expiration'] = exp_date
```

```
    return calls, puts


def third_friday(year, month):
    """
    Calculate third Friday of the month (standard option expiration)
    """
    first_day = datetime(year, month, 1)
    days_until_friday = (4 - first_day.weekday()) % 7
    first_friday = first_day + timedelta(days=days_until_friday)
    third_fri = first_friday + timedelta(weeks=2)
    return third_fri.strftime('%Y-%m-%d')


def get_monthly_expirations():
    """
    Get next 3 monthly option expirations (third Fridays)
    """
    now = datetime.now()
    expirations = []

    for i in range(6):
        month = now.month + i
        year = now.year

        while month > 12:
            month -= 12
            year += 1

        third_fri = third_friday(year, month)
        third_fri_date = datetime.strptime(third_fri, '%Y-%m-%d')

        if third_fri_date > now:
            expirations.append(third_fri)

        if len(expirations) == 3:
            break

    return expirations


def get_vix_option_expirations():
    """
    Get VIX option expirations (VIX expires on Wednesdays, not Fridays)
    """
    stock = yf.Ticker('^VIX')

    try:
```

```
        all_expirations = stock.options

        if len(all_expirations) == 0:
            print("No VIX options available")
            return []

        vix_expirations = all_expirations[:3] if len(all_expirations) >= 3 el
se all_expirations
        print(f"VIX option expirations: {vix_expirations}")
        return vix_expirations

    except Exception as e:
        print(f"Error getting VIX expirations: {e}")
        return []
```

Problem 5: Black-Scholes Implementation

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

def BS_call(S, K, T, r, vol):
    """Black-Scholes call option price"""

    if T <= 0:
        return max(S - K, 0)

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

    call = S * norm.cdf(d1) - K * np.exp(-r*T) * norm.cdf(d2)

    return call


def BS_put(S, K, T, r, vol):
    """Black-Scholes put option price"""

    if T <= 0:
        return max(K - S, 0)

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

    put = K * np.exp(-r*T) * norm.cdf(-d2) - S * norm.cdf(-d1)

    return put
```

Problem 6: Bisection Method for Implied Volatility

```
def bisection_iv(market_price, S, K, T, r, option_type='call', tol=1e-6, max_
iter=100):
    """
    Find implied volatility using bisection method
    Searches for volatility that makes BS price equal market price
    """

    vol_low = 0.001
    vol_high = 5.0

    # calculate prices at bounds
    if option_type == 'call':
        price_low = BS_call(S, K, T, r, vol_low)
        price_high = BS_call(S, K, T, r, vol_high)
    else:
        price_low = BS_put(S, K, T, r, vol_low)
        price_high = BS_put(S, K, T, r, vol_high)

    # check if solution exists in range
    if market_price < price_low or market_price > price_high:
        return None

    # bisection loop
    for i in range(max_iter):
        vol_mid = (vol_low + vol_high) / 2.0

        if option_type == 'call':
            price_mid = BS_call(S, K, T, r, vol_mid)
        else:
            price_mid = BS_put(S, K, T, r, vol_mid)

        diff = price_mid - market_price

        if abs(diff) < tol:
            return vol_mid

        # adjust bounds
        if diff > 0:
            vol_high = vol_mid
        else:
            vol_low = vol_mid

    return vol_mid


def filter_options_for_iv(options_df, S):
    """
```

```
    Filter options before calculating IV
    Keep only options with: bid > 0, ask > 0, volume > 0
    """
    df = options_df.copy()

    # filter conditions
    df = df[(df['Bid'] > 0) & (df['Ask'] > 0) & (df['Volume'] > 0)]

    # calculate market price as midpoint
    df['Market_Price'] = (df['Bid'] + df['Ask']) / 2.0

    # calculate moneyness
    df['Moneyness'] = S / df['Strike']

    return df
```

Problem 7: Newton Method for Implied Volatility

```
def newton_method(f, fprime, x0, tol=1e-6, max_iter=100):
    """
    General Newton's method for finding roots
    f = function, fprime = derivative, x0 = starting guess
    """
    x = x0

    for i in range(max_iter):
        fx = f(x)

        if abs(fx) < tol:
            return x

        fp = fprime(x)

        if abs(fp) < 1e-10:
            return None

        x = x - fx / fp

    return x


def secant_method(f, x0, x1, tol=1e-6, max_iter=100):
    """
    Secant method - like Newton but doesn't need derivative
    """
    fx0 = f(x0)
    fx1 = f(x1)

    for i in range(max_iter):
        if abs(fx1) < tol:
```

```
            return x1

        if abs(fx1 - fx0) < 1e-10:
            return None

        x_new = x1 - fx1 * (x1 - x0) / (fx1 - fx0)

        x0 = x1
        fx0 = fx1
        x1 = x_new
        fx1 = f(x1)

    return x1


def vega(S, K, T, r, vol):
    """
    Vega = derivative of option price with respect to volatility
    Same for both calls and puts
    """
    if T <= 0 or vol <= 0:
        return 0

    d1 = (np.log(S/K) + (r + 0.5*vol**2)*T) / (vol*np.sqrt(T))
    v = S * norm.pdf(d1) * np.sqrt(T)

    return v


def newton_iv(market_price, S, K, T, r, option_type='call', tol=1e-6, max_ite
r=100):
    """
    Find IV using Newton method (much faster than bisection)
    """

    # define function: BS_price(vol) - market_price = 0
    def f(vol):
        if vol <= 0:
            return 1e10

        if option_type == 'call':
            price = BS_call(S, K, T, r, vol)
        else:
            price = BS_put(S, K, T, r, vol)

        return price - market_price

    # derivative is vega
    def fprime(vol):
```

```
        if vol <= 0:
            return 1e-10
        return vega(S, K, T, r, vol)

    result = newton_method(f, fprime, 0.25, tol, max_iter)

    if result is not None and result > 0 and result < 5.0:
        return result
    else:
        return None
```

Problem 9: Put-Call Parity

```
def calc_opposite(df, S, r, from_type='call'):
    """
    Use put-call parity to calculate opposite option price
    Parity: C - P = S - K*e^(-rT)
    """
    df = df.copy()

    price = df['Market_Price']
    K = df['Strike']
    T = df['Time_to_Maturity']

    discount = np.exp(-r * T)

    if from_type == 'call':
        # have call, calculate put: P = C - S + K*e^(-rT)
        calc = price - S + K * discount
        df['Calc_Put'] = calc
    else:
        # have put, calculate call: C = P + S - K*e^(-rT)
        calc = price + S - K * discount
        df['Calc_Call'] = calc

    return df
```

Problem 11: Greeks

```
def delta_call_formula(S, K, T, r, vol):
    """Delta for call option"""
    if T <= 0:
        return 1.0 if S > K else 0.0

    d1 = (np.log(S/K) + (r + 0.5*vol**2)*T) / (vol*np.sqrt(T))
    return norm.cdf(d1)


def gamma_formula(S, K, T, r, vol):
    """Gamma - same for calls and puts"""
    if T <= 0:
```

```
        return 0.0

    d1 = (np.log(S/K) + (r + 0.5*vol**2)*T) / (vol*np.sqrt(T))
    return norm.pdf(d1) / (S * vol * np.sqrt(T))


def delta_approx(S, K, T, r, vol, typ='call', h=0.01):
    """Delta using finite difference"""
    if typ == 'call':
        up = BS_call(S + h, K, T, r, vol)
        down = BS_call(S - h, K, T, r, vol)
    else:
        up = BS_put(S + h, K, T, r, vol)
        down = BS_put(S - h, K, T, r, vol)

    return (up - down) / (2 * h)


def gamma_approx(S, K, T, r, vol, typ='call', h=0.01):
    """Gamma using finite difference"""
    if typ == 'call':
        up = BS_call(S + h, K, T, r, vol)
        mid = BS_call(S, K, T, r, vol)
        down = BS_call(S - h, K, T, r, vol)
    else:
        up = BS_put(S + h, K, T, r, vol)
        mid = BS_put(S, K, T, r, vol)
        down = BS_put(S - h, K, T, r, vol)

    return (up - 2*mid + down) / (h**2)
```

Problem 12: Prediction Using DATA1 IVs

```
def predict(df1, df2, S, r, typ='call'):
    """
    Predict DATA2 prices using DATA1 implied volatilities
    """
    results = []

    for _, row2 in df2.iterrows():
        K = row2['Strike']
        exp = row2['Expiration']
        T = row2['Time_to_Maturity']

        actual = (row2['Bid'] + row2['Ask']) / 2.0

        # find matching option in DATA1
        match = df1[(df1['Strike'] == K) & (df1['Expiration'] == exp)]

        if len(match) > 0:
```

```
            iv = match['IV_newton'].iloc[0]

            # predict using Black-Scholes
            if typ == 'call':
                pred = BS_call(S, K, T, r, iv)
            else:
                pred = BS_put(S, K, T, r, iv)

            err = pred - actual
            pct = (err / actual) * 100 if actual > 0 else 0

            results.append({
                'Strike': K,
                'Actual': actual,
                'Predicted': pred,
                'Error': err,
                'Pct_Error': pct
            })

    return pd.DataFrame(results)
```