Static No-Arbitrage Constraints in Option Markets

Theoretical Framework and Algorithmic Implementation

Financial Engineering Research

December 18, 2025

1 Introduction

In quantitative finance, the principle of no-arbitrage is fundamental for pricing and risk management. Static no-arbitrage conditions are constraints on option prices that must hold regardless of the underlying price dynamics, provided the market does not allow for risk-free profits. This report details the primary constraints—Monotonicity, Vertical Spreads, and Convexity—and provides a robust implementation for data cleaning.

2 Theoretical Conditions

2.1 Monotonicity

Option prices must be monotonic functions of their strike prices. This is derived from the fact that an option with a more favorable strike must be worth at least as much as one with a less favorable strike.

- **Call Options:** $C(K, T)$ must be non-increasing in K. For $K_1 < K_2$:

$$C(K_1, T) \geq C(K_2, T)$$

- **Put Options:** $P(K, T)$ must be non-decreasing in K. For $K_1 < K_2$:

$$P(K_1, T) \leq P(K_2, T)$$

2.2 Vertical Spreads (Slope)

The rate of change of an option price with respect to its strike price is bounded. A vertical spread represents the purchase of one option and the sale of another with a different strike.

For Call options, the maximum value of a vertical spread $C(K_1) - C(K_2)$ is the present value of the difference in strikes:

$$0 \leq C(K_1) - C(K_2) \leq e^{-rT}(K_2 - K_1)$$

Violation of this indicates that the binary risk-free payoff is mispriced relative to the underlying options.

2.3 Convexity (Butterfly Spreads)

The option price surface must be convex with respect to the strike price. This ensures that butterfly spreads—a portfolio consisting of one long call at K_1, two short calls at K_2, and one long call at K_3 (where $K_2 = \frac{K_1+K_3}{2}$)—have non-negative value.

Mathematically, for any $K_1 < K_2 < K_3$:

$$C(K_2) \leq \frac{K_3 - K_2}{K_3 - K_1} C(K_1) + \frac{K_2 - K_1}{K_3 - K_1} C(K_3)$$

If this condition is violated, the implied probability density function (PDF) derived from the second derivative of the option price would be negative, which is economically impossible.

3 Production Implementation

The following Python implementation utilizes `pandas` and `numpy` to identify and filter arbitrage violations from a standard market data format.

```python
import pandas as pd
import numpy as np

def clean_option_arbitrage(df: pd.DataFrame, ticker: str) -> pd.DataFrame:
    """
    Scans a dataframe for static arbitrage violations and removes bad rows.
    Required Columns: 'strike', 'option_type', 'mid_price'
    """
    # Split into Calls and Puts
    calls = df[df['option_type'] == 'call'].sort_values('strike').copy()
    puts = df[df['option_type'] == 'put'].sort_values('strike').copy()

    def scan_monotonicity(data, is_call=True):
        indices_to_drop = set()
        prices = data['mid_price'].values
        strikes = data['strike'].values

        for i in range(1, len(prices)):
            if is_call:
                # Calls: Price must decrease as strike increases
                if prices[i] > prices[i-1]:
                    indices_to_drop.add(data.index[i])
            else:
                # Puts: Price must increase as strike increases
                if prices[i] < prices[i-1]:
                    indices_to_drop.add(data.index[i])
        return indices_to_drop

    def scan_convexity(data):
        indices_to_drop = set()
        prices = data['mid_price'].values
        strikes = data['strike'].values

        # Check triplets for convexity (Butterfly)
        for i in range(1, len(prices) - 1):
            k1, k2, k3 = strikes[i-1], strikes[i], strikes[i+1]
            p1, p2, p3 = prices[i-1], prices[i], prices[i+1]

            # Linear interpolation (chord) value
```

```
            lambda_val = (k3 - k2) / (k3 - k1)
            interpolated = lambda_val * p1 + (1 - lambda_val) * p3

            if p2 > interpolated + 1e-5: # Using epsilon for float safety
                indices_to_drop.add(data.index[i])
        return indices_to_drop

    # Execute Scans
    call_drops = scan_monotonicity(calls, True) | scan_convexity(calls)
    put_drops = scan_monotonicity(puts, False) | scan_convexity(puts)

    total_drops = call_drops | put_drops
    return df.drop(index=total_drops)

# Usage Example
# df_clean = clean_option_arbitrage(market_df, 'AAPL')
```

Listing 1: Production Arbitrage Scanner

4 Conclusion

Maintaining an arbitrage-free surface is critical for the stability of volatility surface calibration and Greek calculations. The algorithms provided ensure that market data is internally consistent before being passed to downstream pricing engines.