IMC Prosperity 3 Documentation

GitHub Repository: https://github.com/angus4718/imc-prosperity-3-public

Angus Cheung[1], Vincent Cheng[2]

May 2025

[1]LinkedIn: https://www.linkedin.com/in/anguscheung, GitHub: https://github.com/angus4718
[2]LinkedIn: https://www.linkedin.com/in/vincent-chengkc, GitHub: https://github.com/Raunald

Contents

Chapter 1

Introduction

1.1 About This Report

Prosperity is a 15-day trading simulation game designed by IMC, a global market-making firm, to challenge participants' skills in algorithmic trading and strategy development in Python. Prosperity introduces players to a fictional archipelago where they trade various products and compete to maximize SeaShells - the in-game currency.

This report documents our algorithmic trading strategies and does not contain analysis of the manual trading part of the competition given that we did not perform well in this area. The trading strategies are shown in pseudo-code instead of Python as we wish to highlight the logic rather than the syntax.

For more details, visit the Prosperity 3 Wiki: Prosperity Wiki.

1.2 Our Team

We are a two-person team named "Ding Crab". We ranked 28^{th} in algorithmic trading and 44^{th} overall.

Angus is a final-year undergraduate student at the Chinese University of Hong Kong, majoring in Quantitative Finance. He also holds a minor in Mathematics.

Vincent is a final-year undergraduate student at the Chinese University of Hong Kong, majoring in Quantitative Finance and Risk Management Science.

1.3 Game Mechanics

The 15 days of simulation of Prosperity are divided into 5 rounds. Each round lasts 72 hours. At the end of every round - before the timer runs out - all teams will have to submit their algorithmic and manual trades to be processed. The algorithms will then participate in a full day of trading against the Prosperity trading bots. Note that all algorithms are trading separately, there is no interaction between the algorithms of different teams. When a new round starts, the results of the previous round will be disclosed and the leaderboard will be updated accordingly. During the game, teams can always visit previous rounds in the dashboard to review information and results. But once a round is closed, teams can no longer change their submitted trades for that round. When round 5 ends, the final results will be processed and the winner of the Prosperity trading challenge will be announced.

On this exchange, the algorithm will trade against a number of bots. At the beginning of each round, it

is disclosed which new products will be available for trading on that day. Sample data for these products is provided that players can use to get a better understanding of the price dynamics of these products, and consequently build a better algorithm for trading them. At round N, order book and price series data for days $N-1$, $N-2$, and $N-3$ will be provided. While most days will feature new products, the old products will also still be tradable in the rounds after which they are introduced.

The timestamps on each day runs from 0 to 1,000,000 at increments of 100. At each timestamp, the algorithm receives the current order book. The algorithm decides how to place trades to act on the current order book. Afterwards, the algorithm's submitted trades get executed first (if bots choose to match the orders). Then, the bots trade between themselves. The algorithm is also fed the trades among the bots between the current timestamp and the previous timestamp.

1.4 Fair Price Exploit

The platform allows an infinite number of Python file submissions and returns the log of the algorithm's performance on the first 100,000 timestamps of an unseen hypothetical trading day (which is not the actual data on which the final submission is evaluated). We realized very early on that if we submit one buy order for each product at time 0 and hold till the end. The log gives us the unrealized P&L by product at each timestamp from which we can infer the internal fair price.

$$\text{Internal Fair Price} = \text{Unrealized P\&L} + \text{Buy Price} \tag{1.1}$$

Using this exploit, we were able to find the best estimate for the fair price comparing the average squared losses between the estimates and the actual fair price.

We tried starting with one sell order instead to see if the internal fair price calculation depends on our positions. The fair prices are identical in both cases.

Chapter 2

Round 1 - Market Making

Round 1 introduced three products - Rainforest Resin, Kelp, and Squid Ink.

2.1 Rainforest Resin

Position limit: 50.

Rainforest Resin is the simplest product in the competition. Its fair value is permanently fixed at 10,000 SeaShells. As observed from Figure 2.1 and Figure 2.2, bots quote bid and ask prices around the fair price with a very wide spread.

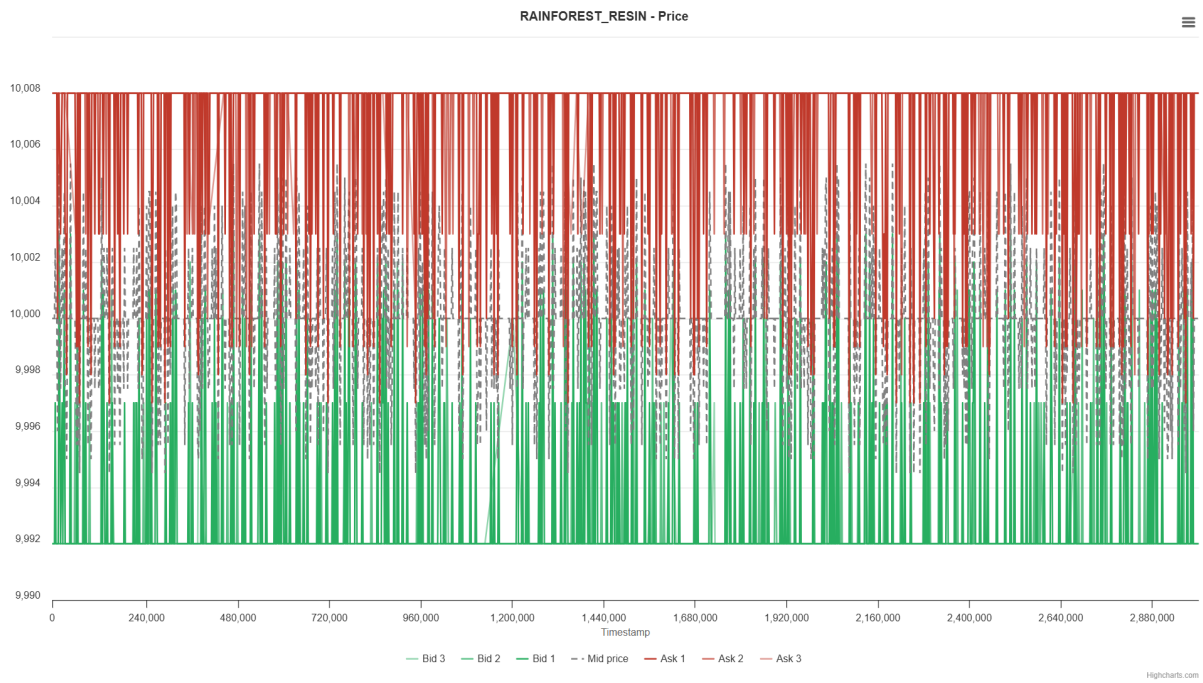


Figure 2.1: Rainforest Resin Price Chart

RAINFOREST_RESIN order depth

Bid volume	Price	Ask volume
	10,008	31
	↑ 16 ↓	
31	9,992	

Figure 2.2: Rainforest Resin Order Book

The first trade idea from a market-taking perspective is to buy whenever the ask crosses below 10,000 and sell whenever the bid crosses above 10,000. We implement the market-taking strategy as follows.

Algorithm 1 Market Taking Algorithm

Initialize max_buy_amount and max_sell_amount as max_size
if sell orders exist **then**
    Sort sell order prices into `asks`
    for each price in `asks` **do**
        if price $\leq$ fair_buying_price **then**
            Calculate amount_to_buy based on available size and limits
            if amount_to_buy > 0 **then**
                Execute buy order and update max_buy_amount
if buy orders exist **then**
    Sort buy order prices into `bids` (descending)
    for each price in `bids` **do**
        if price $\geq$ fair_selling_price **then**
            Calculate amount_to_sell based on available size and limits
            if amount_to_sell > 0 **then**
                Execute sell order and update max_sell_amount

Note that `max_size` is identical to `limit` for Rainforest Resin.

Although this strategy almost guarantees profit, this setup happens too infrequently to make enough profit. Another trade idea is to make market by submitting bid and ask orders based on the current order book. We place a bid at one seashell above the current market bid, and an ask at one seashell below the current market ask. In the meantime, we make sure that our bid must be less than or equal to 9,999 SeaShells, and our ask must be greater than or equal to 10,001 SeaShells. We implement the market-making strategy as follows.

Algorithm 2 Market Making Algorithm

Initialize remaining_buy_capacity and remaining_sell_capacity based on position and limits
if buy orders exist **then**
    Sort buy order prices into `bids`
    max_buy_price $\leftarrow$ highest price below zero_ev_bid + 1
    buy_price $\leftarrow$ min(max_buy_price, pos_ev_bid)
    if remaining_buy_capacity > 0 **then**
        Execute buy order at buy_price for remaining_buy_capacity
if sell orders exist **then**
    Sort sell order prices into `asks`
    min_sell_price $\leftarrow$ lowest price above zero_ev_ask $-$ 1
    sell_price $\leftarrow$ max(min_sell_price, pos_ev_ask)
    if remaining_sell_capacity > 0 **then**
        Execute sell order at sell_price for remaining_sell_capacity

For Rainforest Resin, `zero_ev_bid` $=$ `zero_ev_ask` $= 10,000$.

Lastly, we also realize that some trading opportunities may not be captured if our position is stuck at the limit. Therefore, before we implement the market-making strategy, we have an additional step to offload excessive inventory at the fair price. This brings our position closer to zero whenever possible and allow us to capture more trading opportunities.

Algorithm 3 Inventory Offload Algorithm

```
if position > 0 and fair_selling_price exists in buy orders then
    Calculate max_sell_amount based on positions and limits
    amount_to_sell ← min(position, volume of buy_orders[fair_selling_price], max_sell_amount)
    if amount_to_sell > 0 then
        Execute sell order at fair_selling_price for amount_to_sell
if position < 0 and fair_buying_price exists in sell orders then
    Calculate max_buy_amount based on positions and limits
    amount_to_buy ← min(-position, -volume of sell_orders[fair_buying_price], max_buy_amount)
    if amount_to_buy > 0 then
        Execute buy order at fair_buying_price for amount_to_buy
```

This concludes the three-step strategy for Rainforest Resin. We start with the market taking algorithm to trade on deviations from the fair price, then offload excessive inventory at the fair price if possible, and lastly quoting bid and ask orders in the market.

2.2 Kelp

Position limit: 50.

Kelp's price moves around but remains relatively stable. As seen from Figure 2.3 and Figure 2.4, the bid-ask spread is much tighter for Kelp, but there are no extreme spikes in the price movements. Therefore, we settled on some metrics based on the current order book to estimate Kelp's fair price. We also noticed that there are often small-volume orders before the popular bid and popular ask prices. We define the popular bid/ask price as the bid/ask price with the highest volume. As a result, when we estimate the fair price for Kelp, we use the **popular mid price**, which better reflects the consensus price levels in the market.

$$\text{Popular Mid Price} = \frac{\text{Popular Bid Price} + \text{Popular Ask Price}}{2} \tag{2.1}$$

Other than the fair price determination, the strategy for Kelp is similar to that of Rainforest Resin.

Figure 2.3: Kelp Price Chart

KELP order depth

Bid volume	Price	Ask volume
	2,046	27
	2,045	2
	↑ 2 ↓	
1	2,043	
29	2,042	

Figure 2.4: Kelp Order Book

2.3 Squid Ink

Position Limit: 50.

Squid Ink is much more volatile than the other two products mentioned above. From Figure 2.5 and Figure 2.6, there are occasional extreme spikes in the price followed by a rapid reversal. The bid-ask spread is tighter. Combined with the volatile prices, this leaves less room for profitable market making.

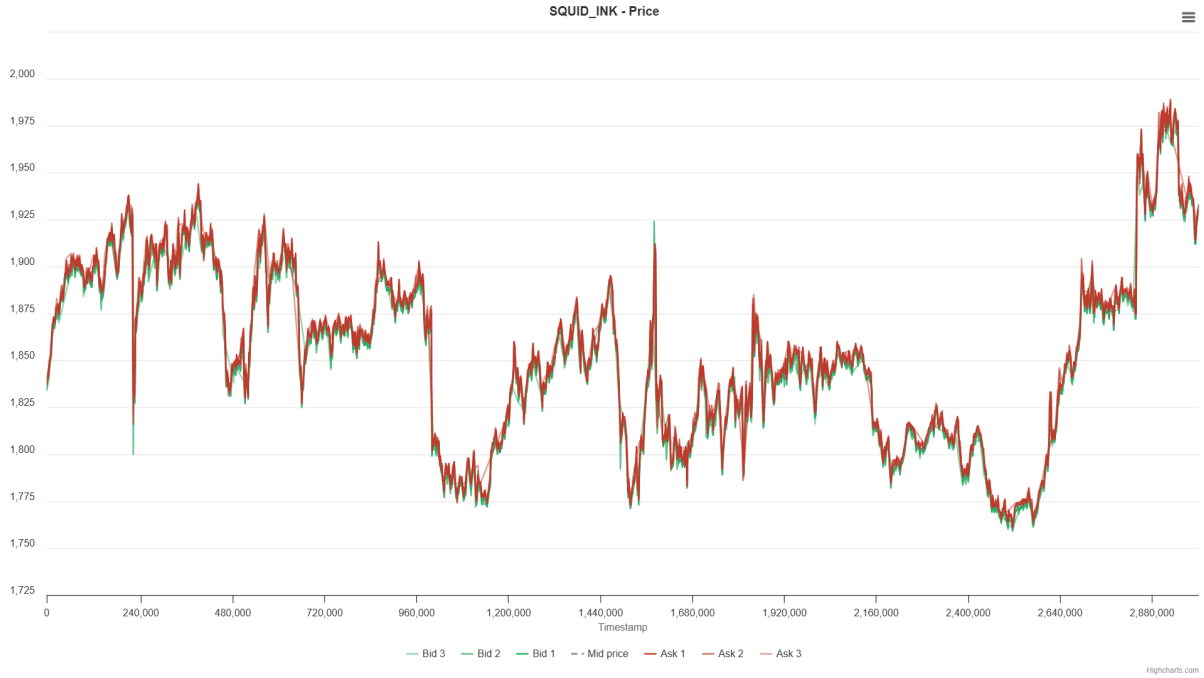


Figure 2.5: Squid Ink Price Chart

SQUID_INK order depth

Bid volume	Price	Ask volume
	1,895	29
	1,894	8
	↑ 2 ↓	
28	1,892	

Figure 2.6: Squid Ink Order Book

To capture this mean-reversion behavior, we implemented the following algorithm.

Algorithm 4 Mean-Reversion Algorithm

[H]

if `len(price_array)` $<$ `period` **then**
  return ▷ Not enough data to calculate moving average
moving_average $\leftarrow$ `sum(price_array[-period:]) / period`
if `fair_price` $<$ `moving_average` $-$ `spread` **then** ▷ Buy if fair price is below threshold
  best_ask $\leftarrow$ `min(sell_orders)`
  amount_to_buy $\leftarrow$ `min(-volume of sell_orders[best_ask], limit - position - total amount already bought)`
  if `amount_to_buy` > 0 **then**
    Execute buy order at `best_ask` **for** `amount_to_buy`
else if `fair_price` $>$ `moving_average` $+$ `spread` **then** ▷ Sell if fair price is above threshold
  best_bid $\leftarrow$ `max(buy_orders)`
  amount_to_sell $\leftarrow$ `min(volume of buy_orders[best_bid], limit + position - total amount already sold)`
  if `amount_to_sell` > 0 **then**
    Execute sell order at `best_bid` **for** `amount_to_sell`

The remaining work was to figure out the optimal values for `period` and `spread`. To test different pairs on the Squid Ink price series on Day -2, Day -1, and Day 0, we implemented a simplified back-test.

Listing 2.1: Squid Ink Simulation

```
for fair in ink_mid_price_series:
    returns = fair.diff()
    pos = (fair < fair.rolling(period).mean() - spread).astype(int) - \
          (fair > fair.rolling(period).mean() + spread).astype(int)
    trading_costs = pos.diff().abs() * 3 / 2
    (returns * pos.shift(1) - trading_costs).cumsum().plot()
```

For `period` $= 100$ and `spread` $= 30$, we obtained the following graph (Figure 2.7). We tried different values but found that there is no optimal pair that consistently catches the spikes. We thought of using z-scores but to no avail as well.

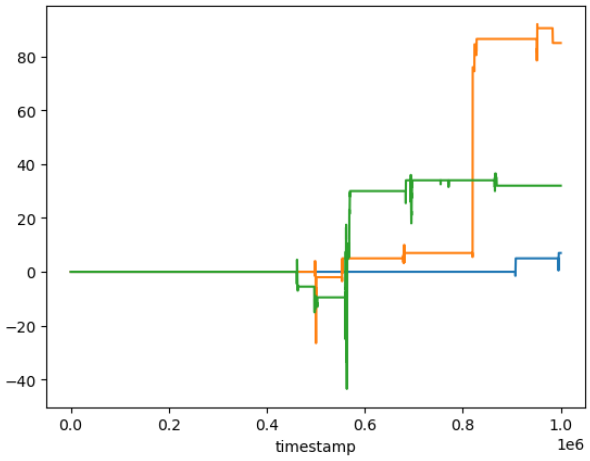


Figure 2.7: Squid Ink Simulation

Chapter 3

Round 2 - Index Arbitrage

Round 2 introduced two picnic baskets. Picnic Basket 1 contains six Croissants, three Jams, and One Djembe, while Picnic Basket 2 contains four Croissants and two Jams. Additionally, participants can trade the underlying products individually. The position limits are as follows:

- Picnic Basket 1: 60
- Picnic Basket 2: 100
- Croissant: 250
- Jam: 350
- Djembe: 60

The obvious strategy here is to trade on the deviation between the baskets' prices and the synthetic baskets' prices calculated from the underlyings' prices.

$$\text{Synthetic Price of Picnic Basket 1} = 6\times\text{Price of Croissant}+3\times\text{Price of Jam}+1\times\text{Price of Djembe} \tag{3.1}$$

$$\text{Synthetic Price of Picnic Basket 2} = 4 \times \text{Price of Croissant} + 2 \times \text{Price of Jam} \tag{3.2}$$

We define the two spreads as follows.

$$\text{Spread 1} = \text{Price of Picnic Basket 1} - \text{Synthetic Price of Picnic Basket 1} \tag{3.3}$$

$$\text{Spread 2} = \text{Price of Picnic Basket 2} - \text{Synthetic Price of Picnic Basket 2} \tag{3.4}$$

There is also a third spread which includes both baskets. Spread 3 is merely a linear combination of Spread 1 and Spread 2. We initially did not focus on Spread 3 as Spread 1 and Spread 2 would have contained all information about the deviation between actual prices and synthetic prices.

$$\text{Spread 3} = \text{Price of Picnic Basket 1} - \frac{3}{2} \times \text{Price of Picnic Basket 2} - \text{Price of Djembe} \tag{3.5}$$

$$\text{Spread 3} = \text{Spread 1} - \frac{3}{2} \times \text{Spread 2} \tag{3.6}$$

Using the sample data provided for Days -1, 0, and 1, we obtain the following graphs for the spreads (Figure 3.1, Figure 3.2, and Figure 3.3).

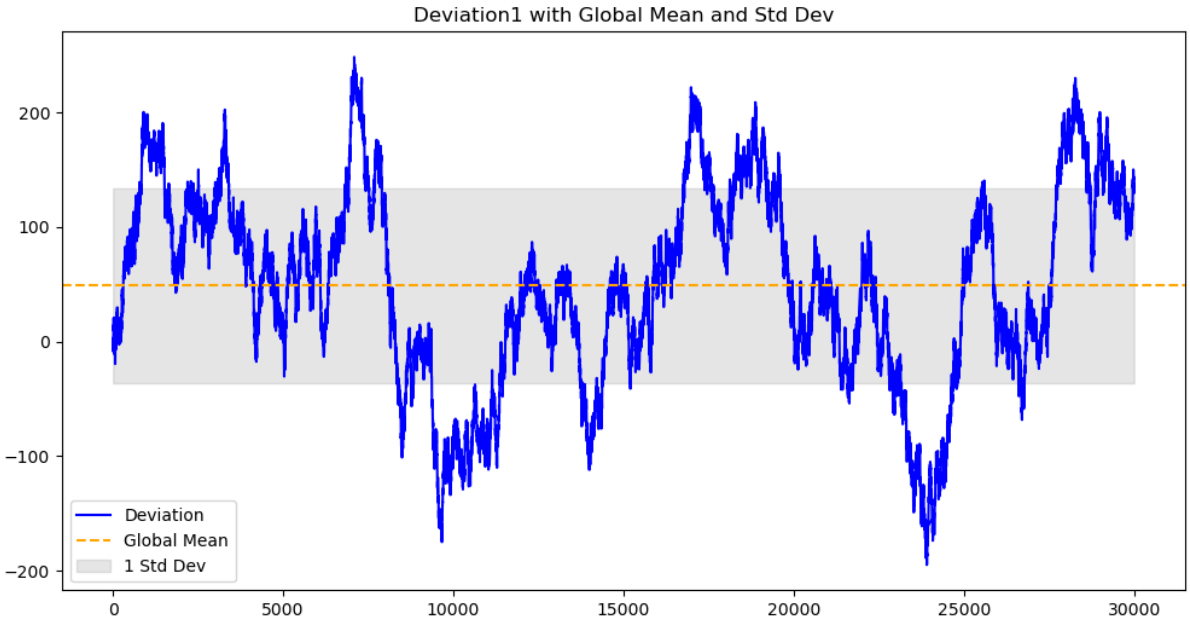


Figure 3.1: Spread 1 Chart

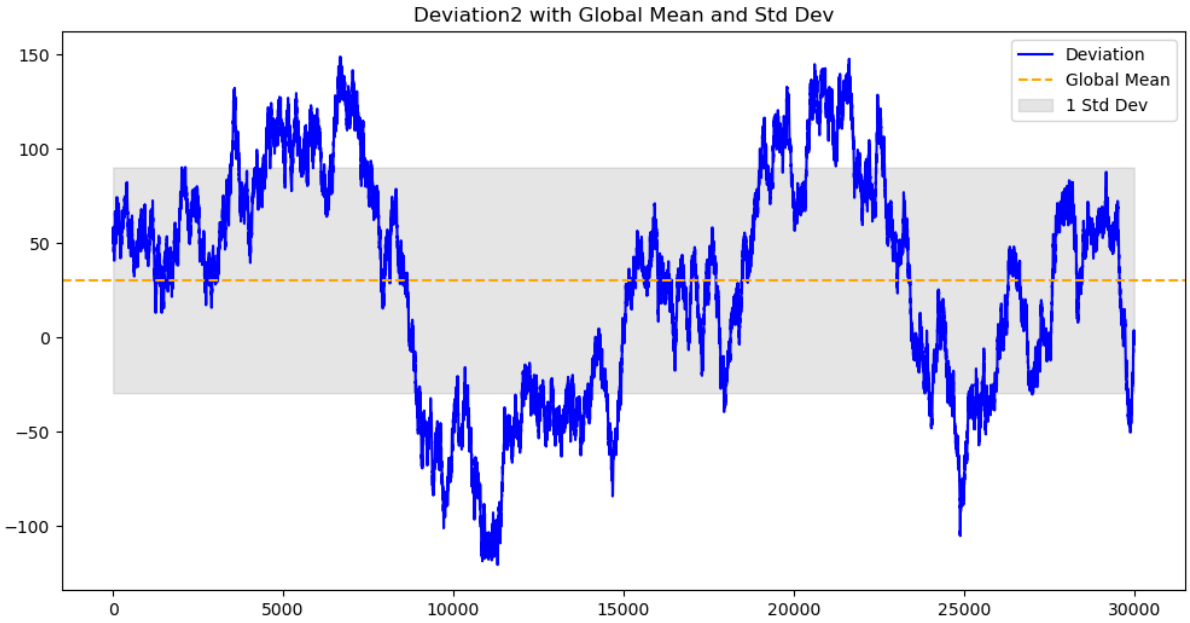


Figure 3.2: Spread 2 Chart

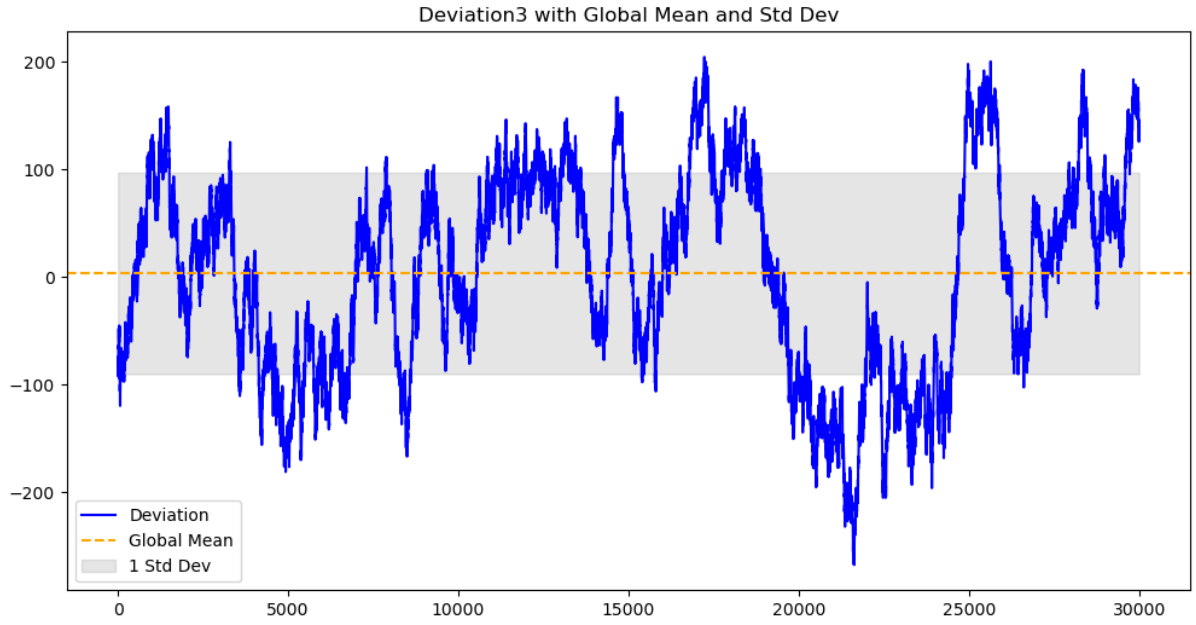


Figure 3.3: Spread 3 Chart

We initially tried pair trading strategies between the actual baskets and the synthetic baskets. However, some issues arose.

Firstly, the position limits on the individual underlyings are too little. Either we reduce the positions on the baskets to fully hedge our positions, which causes unused capacity for the baskets, or we fully utilize the position limits on the baskets, which causes our positions to be partially unhedged. Moreover, in backtesting, we realized that our profit is mostly contributed by the baskets, and our positions in the underlyings are always losing. Therefore, we decided to use the spreads as a signal to long/ short the baskets, but not to trade the underlyings.

We determine our desired positions in the baskets at each timestamp using the following algorithm.

Algorithm 5 Get Desired Positions of Baskets

function GET_DESIRED_POSITION(spread1, spread2, spread3)
  if symbol = `PICNIC_BASKET1` **then**
    $res \leftarrow -(\tanh(\frac{\text{spread1}}{85}) + \tanh(\frac{\text{spread3}}{95}))$
    $res \leftarrow \max(\min(res, 1), -1) \cdot \text{limit}$
    return round(res)
  else if symbol = `PICNIC_BASKET2` **then**
    $res \leftarrow -(\tanh(\frac{\text{spread2}}{65}) - \tanh(\frac{\text{spread3}}{95}) \cdot \frac{3}{2})$
    $res \leftarrow \max(\min(res, 1), -1) \cdot \text{limit}$
    return round(res)
  else
    return 0

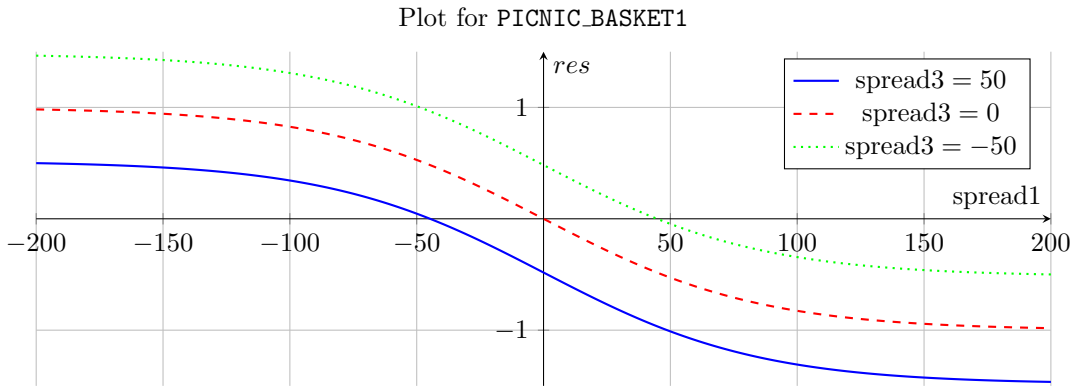
Note: We decided to remove spread 3 from the calculation of basket 2 position as we deemed this much more profitable during backtest. The new formula is simply

$$-\tanh\left(\frac{\text{spread2}}{65}\right) \tag{3.7}$$

Plotting the Function `get_desired_position`

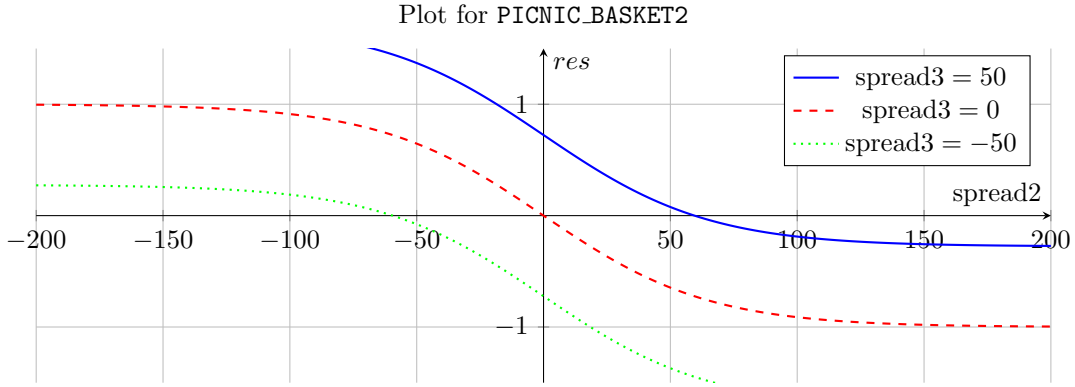
Case 1: `PICNIC_BASKET1`

$$\textbf{Plot of } res = -\left(\tanh\left(\frac{\textbf{spread1}}{85}\right) + \tanh\left(\frac{\textbf{spread3}}{95}\right)\right)$$



Case 2: `PICNIC_BASKET2`

$$\textbf{Plot of } res = -\left(\tanh\left(\frac{\textbf{spread2}}{65}\right) - \tanh\left(\frac{\textbf{spread3}}{95}\right) \cdot \frac{3}{2}\right)$$



We also wrote this `get_mid` function to get the price estimate that most closely resembles the internal fair price of the system. (See Section 1.4).

Algorithm 6 get_mid

```
function GET_MID(order_depth, method)
    Initialize bid_prices, ask_prices, bid_volumes, ask_volumes as lists of size 3
    for i, (price, volume) in the top 3 sorted buy orders (descending prices) do
        bid_prices[i] ← price
        bid_volumes[i] ← volume
    for i, (price, volume) in the top 3 sorted sell orders (ascending prices) do
        ask_prices[i] ← price
        ask_volumes[i] ← -volume
    if method = "impact_weighted_mid_price2" then
        total_bid_volume ← sum(bid_volumes)
        total_ask_volume ← sum(ask_volumes)
        weighted_bid ← sum(price × volume for price, volume in zip(bid_prices,
bid_volumes)) / total_bid_volume
        weighted_ask ← sum(price × volume for price, volume in zip(ask_prices,
ask_volumes)) / total_ask_volume
        return (weighted_bid × total_ask_volume + weighted_ask ×
total_bid_volume)/(total_bid_volume + total_ask_volume)
    else if method = "weighted_mid_price" then
        return (bid_prices[0] × ask_volumes[0] + ask_prices[0] ×
bid_volumes[0])/(bid_volumes[0] + ask_volumes[0])
    else if method = "popular_simple_mid_price" then
        popular_bid ← bid_prices[index(max(bid_volumes))]
        popular_ask ← ask_prices[index(max(ask_volumes))]
        return (popular_bid + popular_ask)/2
    else
        Raise Exception
```

With the analysis and tools above, we implement the trading logic for the baskets below.

Algorithm 7 Basket Trading Algorithm

procedure ACT(state)
    if any symbol not in state.order_depths for symbols in {"CROISSANTS", "JAMS", "DJEMBES", "PICNIC_BASKET1", "PICNIC_BASKET2"} **then**
        return
    if any buy or sell orders are missing for symbols in {"CROISSANTS", "JAMS", "DJEMBES", "PICNIC_BASKET1", "PICNIC_BASKET2"} **then**
        return
    `position ← state.position.get(self.symbol, 0)`
    `croissants_mid ← get_mid(state.order_depths["CROISSANTS"], "impact_weighted_mid_price2")`
    `jams_mid ← get_mid(state.order_depths["JAMS"], "weighted_mid_price")`
    `djembes_mid ← get_mid(state.order_depths["DJEMBES"], "weighted_mid_price")`
    `picnic_basket1_mid ← get_mid(state.order_depths["PICNIC_BASKET1"], "popular_simple_mid_price")`
    `picnic_basket2_mid ← get_mid(state.order_depths["PICNIC_BASKET2"], "weighted_mid_price")`
    `spread_1 ← picnic_basket1_mid - (croissants_mid × 6 + jams_mid × 3 + djembes_mid × 1)`
    `spread_2 ← picnic_basket2_mid - (croissants_mid × 4 + jams_mid × 2)`
    `spread_3 ← picnic_basket1_mid - (picnic_basket2_mid × 3 / 2 + djembes_mid)`
    `desired_position ← get_desired_position(spread_1, spread_2, spread_3)`
    `amount_to_buy ← max(0, desired_position - position)`
    `amount_to_sell ← max(0, position - desired_position)`
    `fair_price ← {"PICNIC_BASKET1": picnic_basket1_mid, "PICNIC_BASKET2": picnic_basket2_mid}[self.symbol]`
    `zero_ev_bid, zero_ev_ask ← floor(fair_price), ceil(fair_price)`
    `pos_ev_bid ← zero_ev_bid if zero_ev_bid < fair_price else zero_ev_bid - 1`
    `pos_ev_ask ← zero_ev_ask if zero_ev_ask > fair_price else zero_ev_ask + 1`
    if `amount_to_buy > 0` **then**
        if `amount_to_buy > position_threshold` **then**
            Execute aggressive buy strategy
        else
            Execute normal buy strategy
    else if `amount_to_sell > 0` **then**
        if `amount_to_sell > position_threshold` **then**
            Execute aggressive sell strategy
        else
            Execute normal sell strategy
    else
        Execute neutral strategy
    Log trading details

The aggressive buy (sell) strategy is implemented by simply shifting the fair price estimate down (up) such that the algorithm more likely buys (sells) even if the prices are slightly unfavorable.

The neutral strategy is simply the market taking strategy and the market making strategy introduced in round 1.

Chapter 4

Round 3 - Options Trading

Round 3 introduced Volcanic Rocks and five Volcanic Rock Vouchers at strike prices 9,500, 9,750, 10,000, 10,250, and 10,500. The vouchers will give the holder the right but not obligation to buy a Volcanic Rock at the strike price at the expiry timestamp. At beginning of Round 1, all the Vouchers have 7 trading days to expire. By end of Round 5, vouchers will have 2 trading days left to expire.

The position limits are as follows:

- Volcanic Rock: 400
- Volcanic Rock Voucher 9500: 200
- Volcanic Rock Voucher 9750: 200
- Volcanic Rock Voucher 10000: 200
- Volcanic Rock Voucher 10250: 200
- Volcanic Rock Voucher 10500: 200

A day before the end of this round, the moderators posted this hint on the wiki, which was of great help to our algorithm design:

> Hello everyone, hope you're enjoying the `VOLCANIC_ROCK` vouchers and the variety of trading strategies these new products introduce. While digging for the rock, Archipelago residents found some ancient mathematics sharing insights into `VOLCANIC_ROCK` voucher trading. Here's what the message with obscure and advanced mathematics read:
>
> **Message begins:**
>
> *I have discovered a strategy which will make ArchiCapital the biggest trading company ever. Here's how my thesis goes,*
>
> t : Timestamp $\quad S_t$: Voucher Underlying Price at t $\quad K$: Strike $\quad$ TTE: Remaining Time till expiry at t
>
> $$V_t : \text{Voucher price of strike } K \text{ at } t$$
>
> *Compute,*
>
> $$m_t = \frac{\log(K/S_t)}{\sqrt{\text{TTE}}}$$
> $$v_t = \text{BlackScholes ImpliedVol}(S_t, V_t, K, \text{TTE})$$

For each t, plot v_t vs m_t and fit a parabolic curve to filter random noise.

This fitted $v_t(m_t)$ allows me to evaluate opportunities between different strikes. I also call fitted $v_t(m_t = 0)$ the base IV and I have identified interesting patterns in the time series of base IV.

Message ends.

4.1 Volatility Smile

Using the hint, we generate two volatility smiles for the bid and ask prices respectively. We use the Black-Scholes formula for a call option as shown below. Note that Φ is the cumulative distribution function (CDF) of the standard normal distribution.

$$C(S, K, T, r, \sigma) = S\Phi(d_1) - Ke^{-rT}\Phi(d_2), \tag{4.1}$$

$$\begin{aligned} d_1 &= \frac{\ln(S/K) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}, \\ d_2 &= d_1 - \sigma\sqrt{T}. \end{aligned}$$

We then reverse-engineer the implied volatility σ using Brent's method.

Listing 4.1: Implied Volatility Calculation

```
def implied_volatility(row, product, actual_price, spot, strike, time_to_expiry
    , r=0):
    def difference(volatility):
        price = black_scholes_call(spot, strike, time_to_expiry, volatility, r)
        return price - actual_price

    try:
        implied_vol = brentq(difference, 1e-3, 3.0, xtol=1e-10)
    except:
        return 0

    return implied_vol
```

This function is applied on the highest bid price and the lowest ask price of each voucher at each timestamp. After discarding erroneous data (where implied volatility cannot be calculated), we plot the following graph.

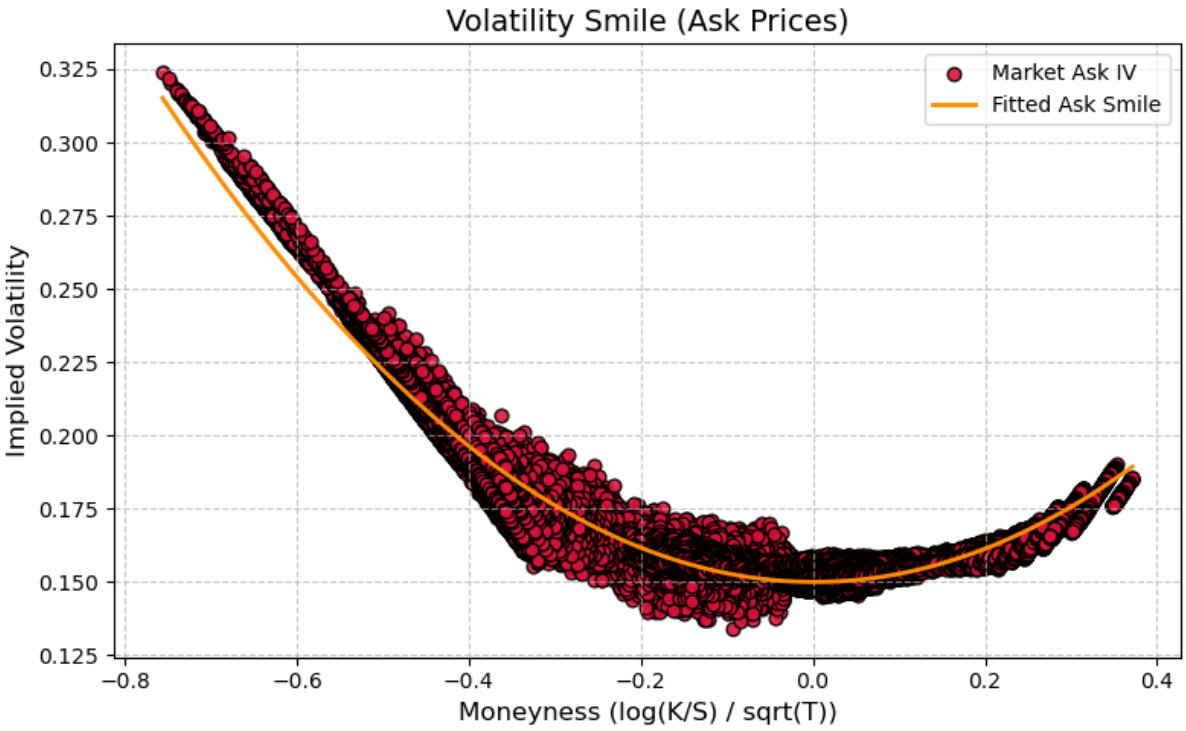


Figure 4.1: Volatility Smile (Ask Prices)

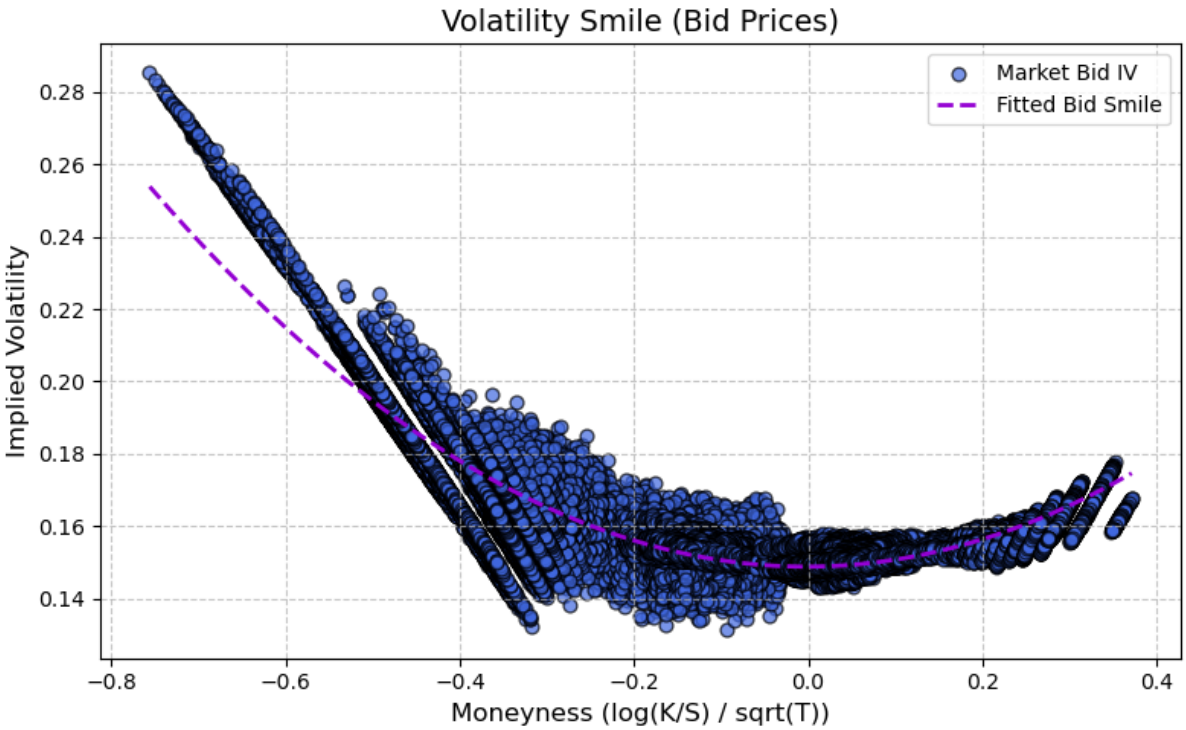


Figure 4.2: Volatility Smile (Bid Prices)

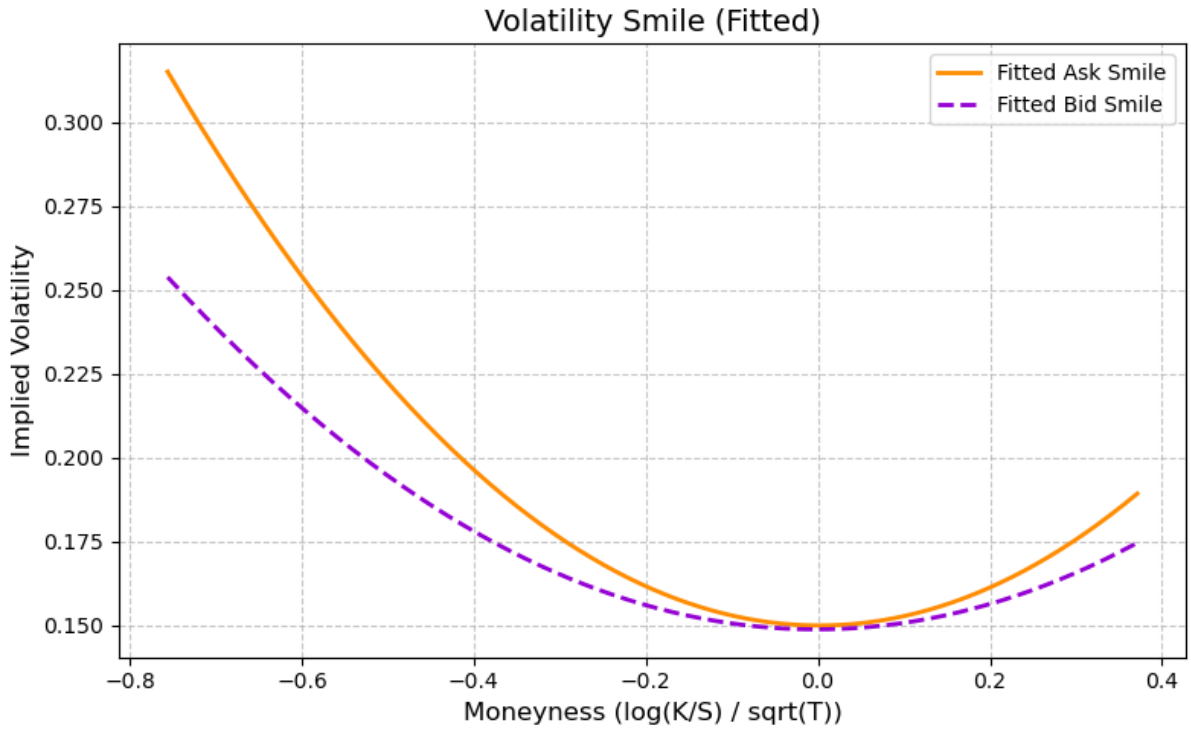


Figure 4.3: Volatility Smile (Fitted Curves)

We use the fitted parameters below for our trading strategy.

- **Model coefficients for ASK:**
  - a (quadratic term): 0.2878490683651303
  - b (linear term): -0.0009201058370376721
  - c (intercept): 0.1499510693474374
- **Model coefficients for BID:**
  - a (quadratic term): 0.1850111314490967
  - b (linear term): 0.0008529599232086579
  - c (intercept): 0.14879176125529844

We initially decided to not delta-hedge our positions given that the competition evaluates only the P&L and we decided to gamble. However, in the actual evaluation, there is a sharp spike in the price movement of the underlying Volcanic Rock which caused a huge loss for us. The following framework shows our strategy with delta hedge included.

Algorithm 8 Option Trading Strategy

procedure ACT(state)
  Initialize: `symbol, limit, window, strike, z_score_threshold`
  `underlying_symbol` ← `"VOLCANIC_ROCK"`
  `position` ← `state.position.get(self.symbol, 0)`
  if `underlying_symbol not in state.order_depths` **then**
    return
  if `symbol not in state.order_depths or missing buy/sell orders` **then**
    return
  Update underlying price:
  `underlying_order_depth` ← `state.order_depths[underlying_symbol]`
  `underlying_bid` ← $\max$(`underlying_order_depth.buy_orders`)
  `underlying_ask` ← $\min$(`underlying_order_depth.sell_orders`)
  `underlying_price` ← (`underlying_bid` + `underlying_ask`)/2
  Append `underlying_price` to `underlying_price_history`
  Retain the last `5 prices` in `underlying_price_history`
  Calculate time-to-expiry and moneyness:
  `tte` ← (`days_left`/365) − (`state.timestamp`/`timestamps_per_year`)
  `moneyness` ← $\log$(`strike`/`underlying_price`)/$\sqrt{\texttt{tte}}$
  Compute theoretical prices:
  if `order_depth.sell_orders exists` **then**
    `best_ask` ← $\min$(`order_depth.sell_orders`)
    `theoretical_iv_ask` ← $c + b \cdot$ `moneyness` $+ a \cdot$ `moneyness`^{2} `(from ask_params)`
    `theoretical_ask_price` ← `Black-Scholes call price using theoretical_iv_ask`
  if `order_depth.buy_orders exists` **then**
    `best_bid` ← $\max$(`order_depth.buy_orders`)
    `theoretical_iv_bid` ← $c + b \cdot$ `moneyness` $+ a \cdot$ `moneyness`^{2} `(from bid_params)`
    `theoretical_bid_price` ← `Black-Scholes call price using theoretical_iv_bid`
  Determine mid prices:
  if `no sell orders` **then**
    `mid_price` ← `best_bid` + 1
    `theoretical_mid_price` ← `theoretical_bid_price` + 1
    `theoretical_iv_mid` ← `theoretical_iv_bid`
  else if `no buy orders` **then**
    `mid_price` ← `best_ask` − 1
    `theoretical_mid_price` ← `theoretical_ask_price` − 1
    `theoretical_iv_mid` ← `theoretical_iv_ask`
  else
    `mid_price` ← (`best_ask` + `best_bid`)/2
    `theoretical_mid_price` ← (`theoretical_ask_price` + `theoretical_bid_price`)/2
    `theoretical_iv_mid` ← (`theoretical_iv_ask` + `theoretical_iv_bid`)/2
  Append `mid_price` to `price_array` and retain the last `window` elements
  if `len(price_array) < window` **then**
    return
  Mean reversion logic:
  `mean_reversion_taker(state, self, symbol, limit, price_array, theoretical_mid_price, window, z_score_threshold, 1)`
  Delta hedging:
  `current_position` ← `(state.position.get(symbol, 0) + total_buying_amount - total_selling_amount)`
  `delta` ← `Black-Scholes delta using underlying_price, strike, tte, theoretical_iv_mid`
  `total_delta` ← `current_position` · `delta`
  `position_to_fill` ← `total_delta` − `current underlying position`
  if `position_to_fill > 0` **then**
    `size` ← $\min$(`position_to_fill`, `limit`)
    Execute buy order for underlying
  else if `position_to_fill < 0` **then**
    `size` ← $\min$(−`position_to_fill`, `limit`)
    Execute sell order for underlying

Chapter 5

Round 4 - Factor Trading

Round 4 introduced Magnificent Macarons along with a bunch of data on sunlight index, sugar prices, import and export tariffs, and transport fees. We are allowed to trade in both the domestic and the foreign market (Pristine Cuisine). To purchase 1 unit of Magnificent Macaron from Pristine Cuisine, we will purchase at the foreign ask price, pay transport fee and import tariff. To sell 1 unit of Magnificent Macaron to Pristine Cuisine, we will sell at the foreign bid price, pay transport fee and export tariff. For every 1 unit of Magnificent Macaron net long position, storage cost of 0.1 Seashells per timestamp will be applied for the duration that position is held. No storage cost applicable to net short position.

Magnificent Macarons have a position limit of 75 and a conversion limit of 10. Conversion is the mechanism for buying from and sell to the foreign market.

Upon reviewing GitHub repositories posted by top-ranking teams, there seems to be an arbitrage opportunity between the domestic and the foreign markets. (See the 2nd place's solution). However, we did not discover this opportunity. We played with the observation data (primarily the sunlight index and sugar prices) to try to find some explanatory power for Magnificent Macaron prices. All regression techniques were futile and we ended up using a very simple logic.

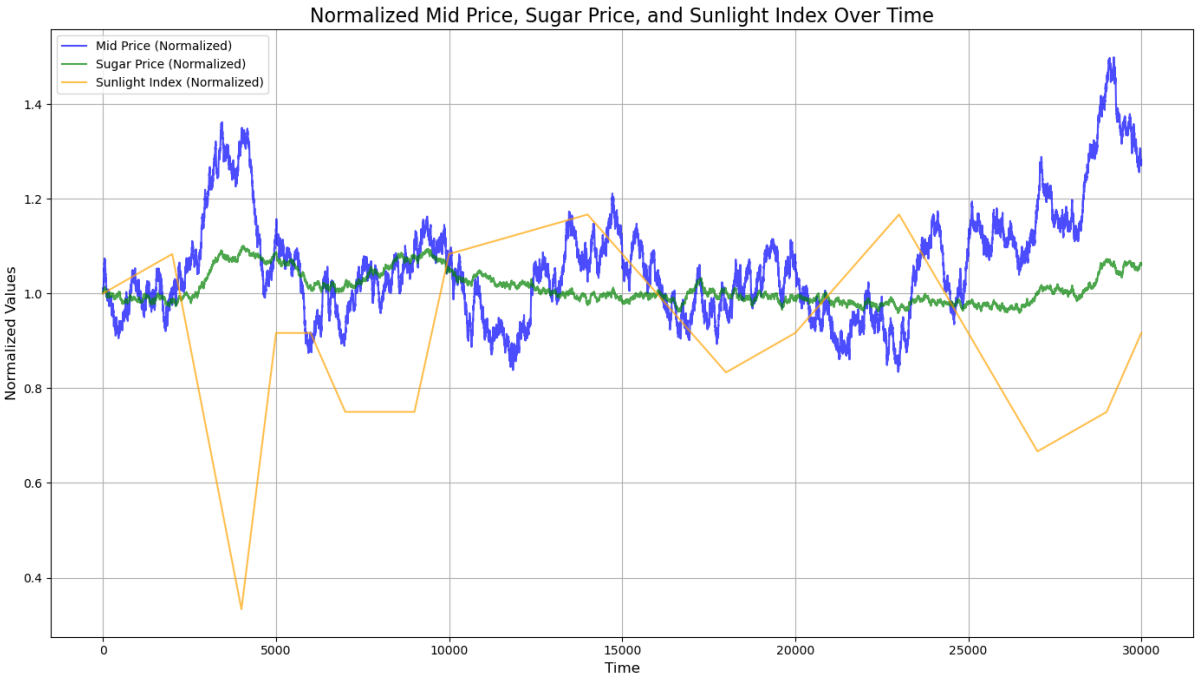


Figure 5.1: Macaron Factor Graph

We identify the current trend of sunlight index and sugar price using the current value minus the previous value. If sunlight index goes below 50:

- Sunlight uptrend + sugar downtrend: Sell signal
- Sunlight downtrend + sugar uptrend: Buy signal

As all outstanding positions at the end of the day will be force-converted, this will potentially result in a huge loss if we do not unwind our positions earlier. Therefore, when the timestamp is greater than 999,000 (i.e., the last 10 timestamps of the day), the algorithm solely unwinds all positions either in the domestic or the foreign market, whichever is more favorable.

In summary, we deployed the following algorithm.

Algorithm 9 Macaron Trading Strategy

```
procedure ACT(state)
    Initialize: position ← Macaron current position
    obs ← Macaron observations
    if obs is None then
        return
    order_depth ← Macaron order book
    if order_depth is None then
        return
    Extract sugar price and sunlight index:
    sugar_price ← obs.sugarPrice
    sunlight ← obs.sunlightIndex
    Initialize previous values if not set:
    if self.previous_sugar is None then
        self.previous_sugar ← sugar_price
    if self.previous_sunlight is None then
        self.previous_sunlight ← sunlight
    Update rolling price array:
    Append sugar_price to self.price_array
    if len(self.price_array) > 100 then
        Remove the oldest price from self.price_array
    Calculate fair price:
    self.fair_price ← self.popular_price_calculator(order_depth)
    Determine buy/sell amounts:
    buy_amount ← min(self.limit - position, self.conversion_limit)
    sell_amount ← min(self.limit + position, self.conversion_limit)
    Calculate local and foreign prices:
    foreign_ask ← obs.askPrice + obs.transportFees - obs.importTariff
    foreign_bid ← obs.bidPrice - obs.transportFees - obs.exportTariff
    Get local bid/ask from order_depth
    Detect sunlight trend:
    sunlight_change ← sunlight - self.previous_sunlight
    if sunlight_change > 0 then
        self.current_trend_sunlight ← 'uptrend'
    else if sunlight_change < 0 then
        self.current_trend_sunlight ← 'downtrend'
    Detect sugar price trend:
    sugar_change ← sugar_price - self.previous_sugar
    if sugar_change > 0 then
        self.current_trend_sugar ← 'uptrend'
    else if sugar_change < 0 then
        self.current_trend_sugar ← 'downtrend'
    Handle position unwinding:
    if state.timestamp > 999000 then
        Unwind at local or foreign market by comparing prices
    Trade based on trends:
    if sunlight <= self.sunlight_threshold then
        if self.current_trend_sunlight == 'uptrend' and self.current_trend_sugar ==
'downtrend' and sell_amount > 0 then
            Sell based on local or foreign prices
        else if self.current_trend_sunlight == 'downtrend' and self.current_trend_sugar
== 'uptrend' and buy_amount > 0 then
            Buy based on local or foreign prices
    Update sunlight and sugar price values
```

Chapter 6

Round 5 - Counterparty Information

The final round introduced no new products. Instead, we can now see the parties behind every trade in the market. We identified that Olivia was buying the dip and selling the tip in Squid Ink and Croissant. We implemented the strategy to follow Olivia's Squid Ink trades all-in if the signal is detected, and fall back to our original strategy otherwise. Effectively, we take the entire bid order book if a sell signal is detected, and take the entire ask order book if a buy signal is detected.

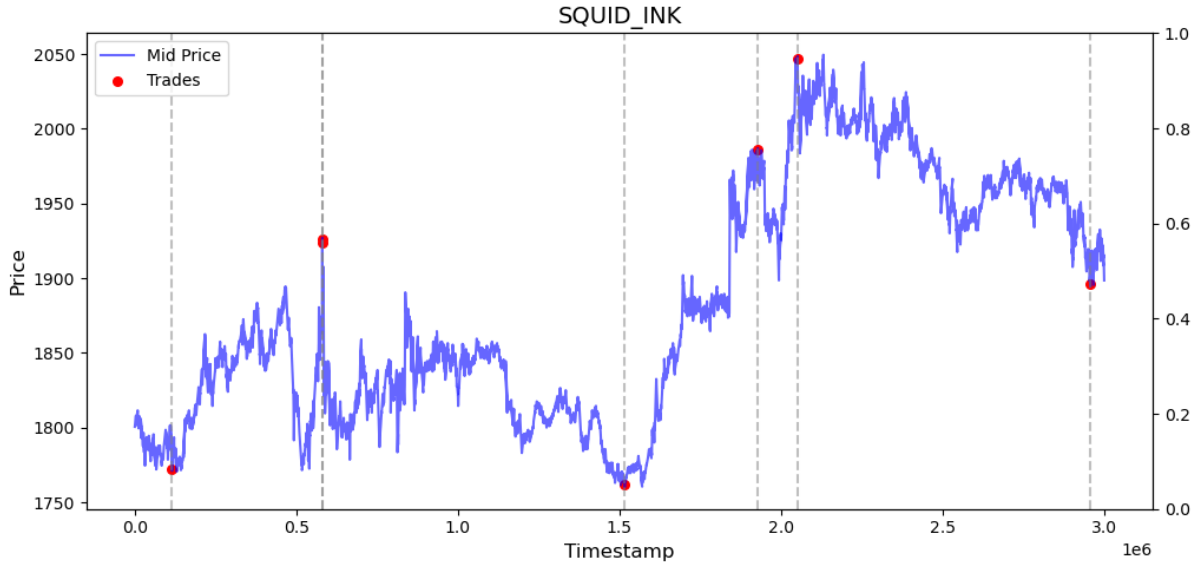


Figure 6.1: Olivia Squid Ink Trades

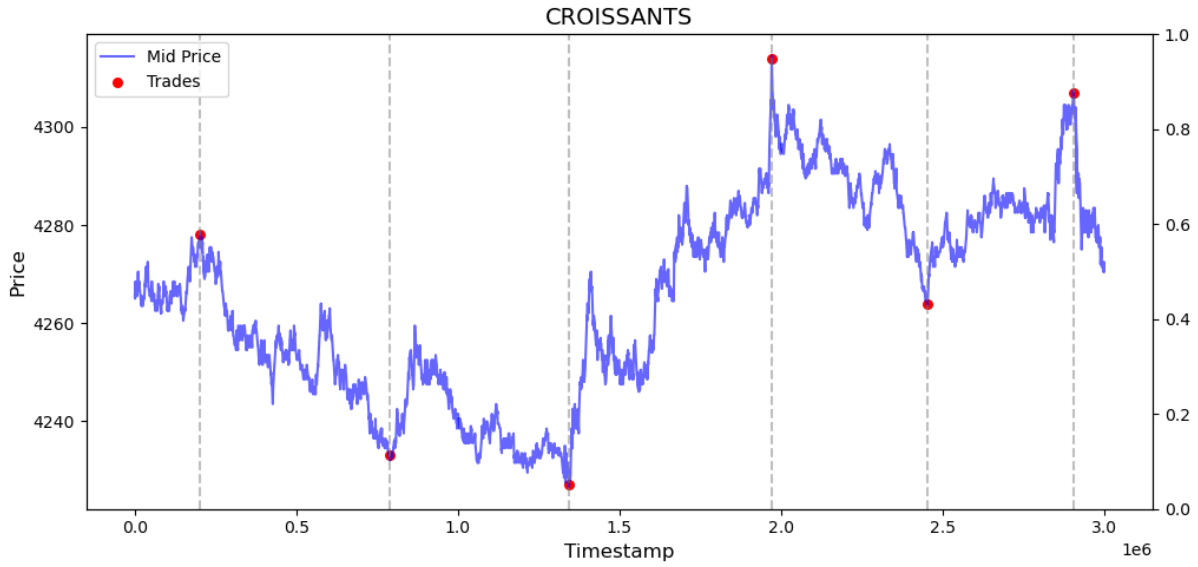


Figure 6.2: Olivia Croissant Trades