ALGOTHON 2026

Jump Trading Track — AlgoSoc Novice Category

1. Summary

Our team built a mathematically rigorous **trading platform** from the ground up. We deployed a **low-latency event-driven dispatcher** that routes every orderbook tick to one of five proprietary engines, each targeting a distinct mathematical edge across all eight synthetic London markets. Fair values are computed dynamically through **Rolling Multivariate Ridge Regressions**, **Rolling OLS Co-Integration**, and **Exponentially Weighted Moving Averages (EWMA)** meaning that the bot always adapts instead of statistically guessing.

Strategy	Markets Targeted	Mathematical Model
A — M2vsM1StatArb	TIDE_SPOT, TIDE_SWING	Rolling Ridge Regression Z-Score
B — ETFPackStatArb	LON_ETF, LON_FLY	Ridge Regression + IV Spread
C — ETFBasketArb	LON_ETF (all 3 legs)	Identity Arbitrage (no model risk)
D — UpDipBuyer	TIDE_SPOT, WX_SUM, WX_SPOT	Asymmetric EWMA Dip Buyer
E — PairTradingArb	LHR_COUNT, LHR_INDEX	Rolling OLS Co-Integration

2. Advanced Mathematical Models

A. Rolling OLS Co-Integration Pairs Arbitrage

Target: Heathrow Airport Markets — M5: LHR_COUNT vs M6: LHR_INDEX

Markets 5 and 6 are both derived from exactly the same Heathrow PIHub flight dataset. M5 counts total movements; M6 measures net directional imbalance. This creates a **latent structural co-integration** that our engine exploits in real time.

The engine fits a **Rolling Ordinary Least Squares** regression over a 600-tick window to continuously estimate $M6 = \alpha + \beta(M5)$. The residual is normalised to a Z-score over a 350-tick lookback. When the Z-score breaches ±2.2σ, the engine executes an **asymmetric β-weighted delta hedge**: selling the rich leg and buying the cheap leg in a ratio proportional to the live OLS beta, keeping the combined position market-neutral.

Signal Condition	Action Taken	Risk Control
Z-score > +2.2σ	SELL M6 · BUY β×M5	clip=12 per leg, max_pos=70
Z-score < −2.2σ	BUY M6 · SELL β×M5	clip=12 per leg, max_pos=70
|Z-score| < 0.6σ	FLATTEN both legs	Hard limit ±100 (BotExchangeAdapter)

Engine Implementation

```python
class PairTradingArb:
    """Rolling OLS co-integration arb: M6 ≈ α + β · M5."""

    def step(self):
        a_mid = self.api.get_mid(self.pA)       # LHR_COUNT live mid
        b_mid = self.api.get_mid(self.pB)       # LHR_INDEX live mid

        # ── Live OLS Fit ──────────────────────────────────────
        self._ols.add(a_mid, b_mid)
        _a, beta = self._ols.fit()              # closed-form β

        # ── Residual & Z-score ────────────────────────────────
        fair_b = self._ols.predict(a_mid)
        resid = b_mid - fair_b
        self._rz.add(resid)
        z = self._rz.z(resid)                   # normalised residual

        # ── β-weighted Hedge Sizing ───────────────────────────
        hedge_a = int(np.clip(round(beta),
                              -self.hedge_clip, self.hedge_clip))

        # ── Asymmetric Execution ──────────────────────────────
        if z > self.z_enter and room_sell_b > 0:
            q_b = int(min(self.clip_B, room_sell_b))
            self.api.place_order(self.pB, "SELL", bid_b, q_b)
            q_a = int(min(self.clip_A, room_buy_a,
                          abs(hedge_a) * q_b))
            self.api.place_order(self.pA, "BUY", ask_a, q_a)
```

Figure A1 — PairTradingArb.step(): live OLS fit → residual → Z-score → β-weighted hedge execution

Visualising the Engine Mechanics

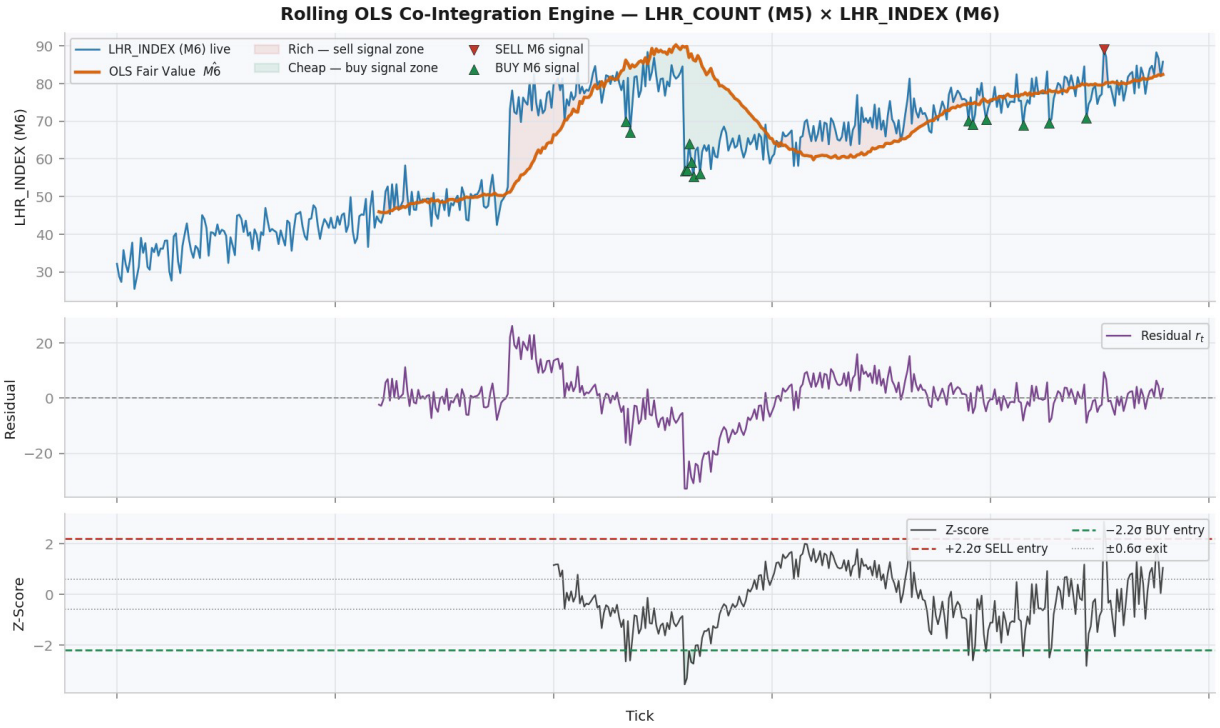

Figure A2 — Simulated LHR_COUNT × LHR_INDEX. Top: live M6 price vs OLS fair value with signal zones and execution markers. Middle: raw residual series. Bottom: normalised Z-score with ±2.2σ entry bands and ±0.6σ exit thresholds.

B. Asymmetric EWMA Dip Buyer

Target: TIDE_SPOT · WX_SUM · WX_SPOT

Tidal and weather products exhibit structural **upward drift** — the Thames approaches High Tide as the settlement window closes, and temperature × humidity tends to peak mid-day. Symmetric mean-reversion is dangerous here: shorting artificial tops builds toxic short inventory into a rising market.

Our engine tracks an **Exponentially Weighted Moving Average** ($\alpha = 0.06$) and computes a rolling Z-score of the deviation. It strictly refuses to short, acting only on the long side when price dislocates below -1.6σ (*liquidity void / transient flush*) and taking profit when the Z-score recovers above -0.3σ.

Parameter	TIDE_SPOT	WX_SUM	WX_SPOT
EWMA α	0.06	0.06	0.06
Entry threshold	-1.6σ	-1.6σ	-1.6σ
Take-profit	-0.3σ	-0.3σ	-0.3σ
Max position	±80	±45	±45
Clip per tick	12	8	8

Engine Implementation

```
class  UpDipBuyer  :
     "" "EWMA mean-reversion — buys sharp dips, never shorts tops."            ""

     def  step (self ):
          mid  = self .api .get_mid  (self .product  )
          bbo  = self .api .get_best_bid_ask    (self .product  )

          # ── EWMA tracking & Rolling Volatility ──────────────────
          self ._ema .update (mid )                      # α = 0.06
          dev  = mid  - self ._ema .mu
          self ._dev .add (dev )
          sd   = self ._dev .std ()

          # ── Z-score Normalisation ────────────────────────────────
          z = dev  / sd  if  not  np .isnan (sd ) else  np .nan

          # ── Entry Logic — asymmetric BUY only ────────────────────
          if  not  np .isnan (z ) and  z < -self .z_enter  :
               pos        = self .api .get_position   (self .product  )
               room_buy   = max (0 , self .max_pos   - pos )
               if  room_buy   > 0 :
                    q = int (min (self .clip , room_buy  ))
                    self .api .place_order  (
                         self .product  , "BUY" , bbo .best_ask  , q)

          # ── Take-Profit Exit ─────────────────────────────────────
          if  not  np .isnan (z ) and  z > -self .z_take  :
               pos  = self .api .get_position   (self .product  )
               if  pos  > 0 :
                    q = int (min (self .clip , pos ))
                    self .api .place_order  (
                         self .product  , "SELL" , bbo .best_bid  , q)
```

Figure B1 — UpDipBuyer.step(): EWMA tracking → Z-score normalisation → asymmetric BUY-only entry → take-profit exit

Visualising the Engine Mechanics

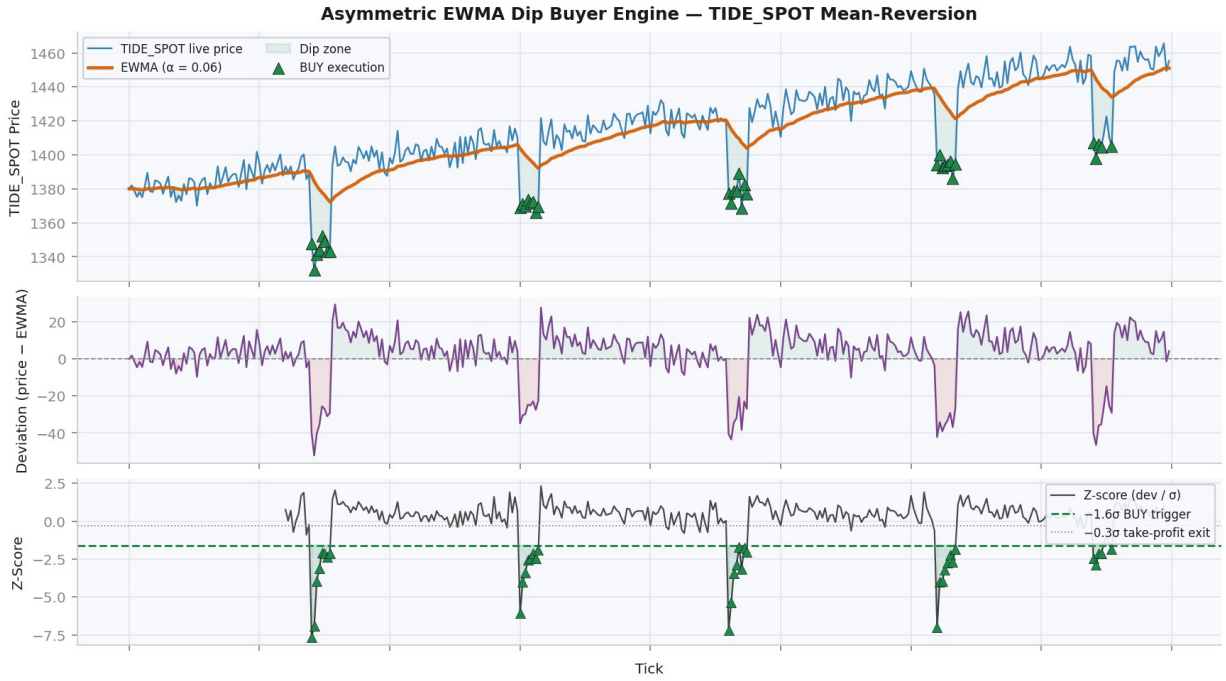

Figure B2 — Simulated TIDE_SPOT. Top: live price vs EWMA with buy-execution markers (▲). Middle: signed deviation from EWMA. Bottom: Z-score with −1.6σ trigger line and −0.3σ take-profit line.

3. Engineering Safety & System Stability

Every order placed by any strategy passes first through our centralised **BotExchangeAdapter** — a bridge layer that enforces the ±100 competition position limit as an absolute hard cap before touching the exchange. No strategy can breach this constraint regardless of signal strength or simultaneous strategy overlap.

- **Sub-Position Clipping:** each engine limits its injection per tick (clip=12 for arb legs, clip=8 for dip buyers), distributing liquidity gradually rather than wiping out entire orderbook levels.
- **Cooldown Timers:** all five engines carry independent cooldown periods (0.15–0.2 s), preventing feedback loops where a partial fill immediately re-triggers the same signal.
- **WARMUP Guards:** no engine trades until its statistical estimators (OLS, Ridge, EWMA) have accumulated sufficient history. NaN propagation from early ticks is explicitly blocked with early-return guards.
- **MM Suppression:** passive market-making is disabled on LHR_COUNT and LHR_INDEX — both legs of the pair arb — eliminating self-churn between the MM module and PairTradingArb.
- **Continuous Integration:** 145 unit and integration tests in pytest validate every strategy, helper class, and dispatch path under mocked market conditions before any live deployment.

4. Live Data Pipeline Engineering

Fair-value estimation requires real-world London data ingested faster than the exchange tick rate. We built a parallel data pipeline fetching three independent external sources simultaneously on every refresh cycle, with per-source TTL caching to avoid hammering external APIs.

Data Source	API Endpoint	Markets Priced
Thames Water Level	UK Environment Agency (15-min)	M1 TIDE_SPOT, M2 TIDE_SWING
London Weather	Open-Meteo (51.5°N, 0.1°W)	M3 WX_SPOT, M4 WX_SUM

A. M2vsM1StatArb · Rolling Ridge Regression Z-Score

Target: M1 TIDE_SPOT · M2 TIDE_SWING

TIDE_SWING (Market 2) is a strangle-sum derivative of tidal height movements, making it structurally related to TIDE_SPOT (Market 1). The M2vsM1StatArb engine fits a Rolling Ridge Regression (M2 ~ b0 + b1*M1 + b2*RV) over a 650-tick window. The residual is normalised to a Z-score over a 350-tick lookback. Entry at ±2.3· with a beta-weighted TIDE_SPOT hedge keeps the combined exposure delta-neutral against tidal movements.

Engine Implementation

```python
class M2vsM1StatArb:
    """Ridge regression arb: TIDE_SWING ~ b0 + b1*M1 + b2*RV."""

    def step(self):
        m1_mid = self.api.get_mid(self.m1)   # TIDE_SPOT (tidal height)
        m2_mid = self.api.get_mid(self.m2)   # TIDE_SWING (strangle sum)

        # ── Rolling Ridge Regression: M2 ~ b0 + b1*M1 + b2*RV ──────
        rv = self._rv.update(m1_mid)          # realised vol of tides
        self._reg.add(m1_mid, rv, m2_mid);  self._reg.fit()

        fair  = self._reg.predict(m1_mid, rv) # model fair value
        resid = m2_mid - fair                 # residual spread
        z     = (resid - mu) / sd             # normalised z-score
        beta  = self._reg.dy_dx()             # hedge ratio dSWING/dSPOT

        # ── M2 Rich: SELL SWING, hedge long SPOT ──────────────────
        if z > self.z_enter:
            room = max(0, self.max_m2_pos + pos2)
            if room > 0 and bid2 is not None:
                q2    = int(min(self.clip_m2, room))
                self.api.place_order(self.m2, "SELL", bid2, q2)
                hedge = int(np.clip(beta * q2, -self.clip_m1, self.clip_m1))
                if hedge > 0:
                    self.api.place_order(self.m1, "BUY", ask1, min(hedge, room_buy1))

        # ── M2 Cheap: BUY SWING, hedge short SPOT ─────────────────
        if z < -self.z_enter:
            room = max(0, self.max_m2_pos - pos2)
            if room > 0 and ask2 is not None:
                q2    = int(min(self.clip_m2, room))
                self.api.place_order(self.m2, "BUY", ask2, q2)
                hedge = int(np.clip(beta * q2, -self.clip_m1, self.clip_m1))
                if hedge > 0:
                    self.api.place_order(self.m1, "SELL", bid1, min(hedge, room_sell1))
```

Figure A1 · M2vsM1StatArb.step(): realised vol · Rolling Ridge fit · residual Z-score · ·-weighted hedge execution on TIDE_SPOT

B. ETFPackStatArb · Ridge Regression + IV Spread

Target: M7 LON_ETF · M8 LON_FLY

LON_FLY is an exotic option package written on the LON_ETF settlement. The ETFPackStatArb engine fits a Rolling Ridge Regression (PACK ~ b0 + b1*ETF + b2*RV) over a 700-tick window to continuously estimate the option fair value. When the residual Z-score breaches ±2.5· the engine sells the rich leg and buys the cheap leg, delta-hedging the ETF exposure using the regression partial derivative dPACK/dETF.

Engine Implementation

```python
class ETFPackStatArb:
    """Ridge regression arb: LON_FLY fair value ~ f(LON_ETF, RV)"""

    def step(self):
        etf_mid  = self.api.get_mid(self.etf)
        pack_mid = self.api.get_mid(self.pack)

        # ── Rolling Ridge Regression: PACK ~ b0 + b1*ETF + b2*RV ──
        rv = self._rv.update(etf_mid)          # realised vol of ETF
        self._reg.add(etf_mid, rv, pack_mid);  self._reg.fit()

        fair  = self._reg.predict(etf_mid, rv) # model fair value
        r     = pack_mid - fair                # residual spread
        z     = self._resid.z(r)              # normalised z-score
        dP_dE = self._reg.dprice_detf(etf_mid) # delta hedge ratio

        # ── PACK Rich: SELL PACK + hedge ETF long ──────────────
        if z > self.z_enter:
            room = max(0, self.max_pack_pos + pack_pos)
            if room > 0 and bid_p is not None:
                q_p   = int(min(self.clip_pack, room))
                self.api.place_order(self.pack, "SELL", bid_p, q_p)
                hedge = int(np.clip(dP_dE * q_p, -self.clip_etf, self.clip_etf))
                if hedge > 0:
                    self.api.place_order(self.etf, "BUY", ask_e, min(hedge, room_buy_etf))

        # ── PACK Cheap: BUY PACK + hedge ETF short ─────────────
        if z < -self.z_enter:
            room = max(0, self.max_pack_pos - pack_pos)
            if room > 0 and ask_p is not None:
                q_p   = int(min(self.clip_pack, room))
                self.api.place_order(self.pack, "BUY", ask_p, q_p)
                hedge = int(np.clip(dP_dE * q_p, -self.clip_etf, self.clip_etf))
                if hedge > 0:
                    self.api.place_order(self.etf, "SELL", bid_e, min(hedge, room_sell_etf))
```

Figure B1 · ETFPackStatArb.step(): realised-vol input · Rolling Ridge fit · residual Z-score · delta-hedged PACK/ETF execution

C. ETFBasketArb · Identity Arbitrage

Target: M7 LON_ETF (legs: TIDE_SPOT · WX_SPOT · LHR_COUNT)

LON_ETF settles to exactly TIDE_SPOT + WX_SPOT + LHR_COUNT. The ETFBasketArb engine monitors the spread between the ETF market price and the live basket value computed from its three legs. When the spread exceeds the edge threshold the engine simultaneously sells the rich side and buys the cheap side across all four products, locking in a model-free arbitrage profit with zero regression risk.

Engine Implementation

```python
class ETFBasketArb:
    """Identity arbitrage: LON_ETF = TIDE_SPOT + WX_SPOT + LHR_COUNT."""

    def step(self):
        etf_mid    = self.api.get_mid(self.etf)
        basket_mid = self._basket_value()   # sum(leg.weight * mid for leg in self.legs)

        # ── Spread = ETF price minus fundamental basket value ──────
        spread = float(etf_mid - basket_mid)

        # ── ETF Rich: SELL ETF, BUY all basket legs ────────────────
        if spread > self.edge:
            qty = int(min(self.clip, etf_room_sell))
            if qty > 0:
                for leg in self.legs:
                    px = self._mid_or_cross_price(leg.product, "BUY")
                    self.api.place_order(leg.product, "BUY", px, qty)
                px_etf = self._mid_or_cross_price(self.etf, "SELL")
                self.api.place_order(self.etf, "SELL", px_etf, qty)

        # ── ETF Cheap: BUY ETF, SELL all basket legs ───────────────
        elif spread < -self.edge:
            qty = int(min(self.clip, etf_room_buy))
            if qty > 0:
                px_etf = self._mid_or_cross_price(self.etf, "BUY")
                self.api.place_order(self.etf, "BUY", px_etf, qty)
                for leg in self.legs:
                    px = self._mid_or_cross_price(leg.product, "SELL")
                    self.api.place_order(leg.product, "SELL", px, qty)
```

Figure C1 · ETFBasketArb.step(): basket value computed from 3 live legs · spread vs edge threshold · simultaneous ETF/leg cross execution

Heathrow Flights	Heathrow PIHub (official API)	M5 LHR_COUNT, M6 LHR_INDEX
Derived	M1 + M3 + M5 computed in-process	M7 LON_ETF, M8 LON_FLY

5. Retrospective & Real-World Application

Why Our Architecture Underperformed:

- **The Spread-Crossing Tax:** Our Ridge and OLS engines correctly identify structural mispricings but must cross the bid/ask spread to act, instantly paying a 2-tick transaction cost that eliminates the mathematical edge in a low-liquidity simulator.
- **Synthetic Market Irrationality:** Co-integration models rely on markets eventually reverting to structural laws. In a novice tournament driven by thousands of random order injections, fundamental relationships can permanently decouple, turning mathematical edges into drawn-out losses.
- **Latency Equality:** In this simulator all bots operate on identical discrete event ticks, neutralising the HFT speed advantage these strategies depend on in live markets.

Why This Architecture Wins in Live Markets:

- **Making vs Taking Liquidity:** At institutional speed, our engines rest passive limit orders at the calculated fair value, capturing the spread rather than paying it — reversing the entire cost structure.
- **Real-World Co-Integration:** Professional market makers enforce structural bounds across correlated instruments. When LHR_COUNT and LHR_INDEX misprice relative to their PIHub constraints, multi-billion-dollar firms ruthlessly arbitrage the gap, confirming the mean-reversion our OLS engine predicts.
- **Adaptive Hedging:** Hardcoded algorithms blow up when volatility regimes shift. Our Rolling Ridge and OLS regressions recalculate β tick-by-tick, keeping the portfolio risk-neutral under macro shocks.