# Description

The code attached is a culmination of my efforts to create a successful trading algorithm. My project began with the creation and testing of my novel volatility indicator PCI (Price Confidence Indicator). PCI is calculated by first calculating μ = ((Shares Outstanding + Volume) * Highest Value) then λ = ((Shares Outstanding - Volume_t) * Lowest Value) and Finally PCI = (Highest value - Lowest value) / (μ - λ).
The concept behind this was to imagine Mu and Lambda values as the hypothetical limit the price could have reached that day adjusted for the real price movement and volume with Mu being the Maximum and Lambda the Minimum (hence Mu for Maximum and Lambda for lowest). Then by dividing the highest-lowest value by Mu-Lambda I get a measure of market confidence in a given price, this reveals market confidence as we can consider 4 hypothetical scenarios:

| | Low Volume | High Volume |
|---|---|---|
| High (High - Low) | In this scenario PCI is high. High future market volatility is expected as the market is shifting rapidly with little volume required to generate a market movement indicating a high selling or buying pressure thus expected future changes. | In this scenario PCI could vary based on exact values but likely isn't low. High future volatility would be expected as the market is uncertain in the initial value of the stock thus causing large volumes and large market movements. |
| Low (High-Low) | In this scenario PCI will vary based on exact values but is likely high. Low future volatility is expected as the market is confident in the given price meaning there is no need for price changes or much trading volume. This is a weakness of PCI. | In this scenario PCI will be low, Low future volatility is expected as the market is confident in the current pricing since despite high volumes the pricing remains constant. |

This is the theoretical core of PCI.
The code below calculates PCI and the following outputs. Each code outputs:

- lift@topK_mean which is a measure of how much stronger PCI is at extremes
- Partial Correlation is a measure of the new information added by PCI onto information like standard deviation and ATR14
- Where IC is the internal correlation it outputs the:
    - IC mean
    - IC median
    - IC at certain percentiles
- CI which is the share of bootstraps where IC is greater than 0

Each code compares different smoothing techniques and horizons which after testing and comparing outputs was discovered to be weighted moving average of strength 15 over a 10 day horizon.

Each code also has certain control variables such as bootstrapping, and including a random PCI as a baseline.

- The file named PCI_ABSReturns_Research finds the correlation of PCI and future absolute returns
- The file named PCI_STDev_Research calculates the correlation of PCI and standard deviation
    - std_limits can be adjusted such that if std_limits = 0.9 only the 90th percentile of movements are used as data points to calculate standard deviation.

This concluded my PCI research however for future experimentation I intend to test PCI over larger time periods then a day to see if that can predict longer trends and I also want to test correlation with absolute returns and standard deviation such that high PCI and high volume is required to remedy the contradiction that occurs with low volume and low (high-low).

<u>Developing PCI into a tradeable algorithm:</u>

- First I tested if PCI had any directional qualities using a Montecarlo simulation which it did not.
- Given this to create a tradeable strategy I introduced a directional indicator named ASI
- ASI is calculated by doing the following:
    - Define a time period e.g. one day
    - ASI is calculated using daily data as one day is the time period
    - First find the number of days with positive returns and the number of days in total for a given horizon up to N days
    - Then find the number of 2 day periods with positive returns and the number of two day periods in total
    - continue this all the way to N day periods
    - Take the number of periods with positive returns and divide with total number of periods to get ASI
    - For example take a 5 day period where returns are as follows [+5, -5, -5, +3, +3]
    - There are 3 successful one day periods and 5 total periods
    - there is 1 successful period out of 4 total possible 2 day periods
    - There is 1 successful period out of 3 possible 3 day periods
    - There no successful periods out of 2 possible 4 day periods
    - There is 1 successful period out of 1 possible 5 day periods
    - Thus ASI is (3+1+1+1)/(5+4+3+2+1) =0.4

- The ASI I use for the strategy can be dubbed ASI2 or RASI which is from calculating a 15 day rolling ASI and dividing it by a 50 day rolling ASI.
    - I deduced this to be the best combination through testing ASI correlation with future returns in a similar fashion to PCI

- To combine the two indicators I did the following calculations
    - For a given ticker ASIΩ = (ASI^ASI indice) / Σ(ASIΩ)
    - For a given ticker PCIΩ = (PCI^PCI indice) / Σ(PCIΩ)
    - For a given ticker Ω = ASIΩ + PCIΩ
    - For a given ticker weighting = Ω/ΣΩ where ΣΩ can be narrowed down to only the top K percentile ASI basket

- For this I used the S&P 500 as a universe initially and then to avoid survivorship bias I used a contemporary list
- I then began to notice that the code was significantly more successful when rebalancing on Fridays than any other day with cumulative returns being 1.5X the second highest day and 6.5X the lowest day indicating something must be causing this and it is not simply a case of overfitting.
    - To investigate I found that Fridays generally have higher trading volumes. If trading volume is on average higher I theorised that PCI would have more information to outline sentiment and trends.
        - I then plotted every rebalance dates trading volume against the resultant return to see if higher volume indicated higher returns however volume had a negligible effect
    - My other theory is that since option expiration tends to be on Friday the code may be picking up on maximum pain theory to some extent.
        - Unfortunately I had insufficient access to data to graph a rebalance days resultant return against the value of option contracts expiring on that rebalance day (or the next)

- The file named Main_ASI+PCI_Backtest contains the strategy's performance using the contemporary S&P 500 universe and in the optimal set up that I have discovered

- My future research plans are:
    - Testing my option expiration theory
    - Exploring the strategy's performance in larger universes and in different geographies
    - Testing higher frequency ASI calculations e.g. hourly candles to get more incremental ASI

# Code

# PCI FInal

**Cell 1**

```
# --- Libraries ---
import datetime as dt
import numpy as np
import pandas as pd
import yfinance as yf
import statsmodels.api as sm
import matplotlib.pyplot as plt

# ---------- Parameters ----------
TICKERS = ["A", "AAPL", "ABBV", "ABNB", "ABT", "ACGL", "ACN", "ADBE",
"ADI", "ADM", "ADP", "ADSK", "AEE", "AEP", "AES", "AFL", "AIG", "AIZ",
"AJG", "AKAM", "ALB", "ALGN", "ALL", "ALLE", "AMAT", "AMCR", "AMD", "AME",
"AMGN", "AMP", "AMT", "AMZN", "ANET", "AON", "AOS", "APA", "APD", "APH",
"APO", "APP", "APTV", "ARE", "ATO", "AVB", "AVGO", "AVY", "AWK", "AXON",
"AXP", "AZO", "BA", "BAC", "BALL", "BAX", "BBY", "BDX", "BEN", "BF-B",
"BG", "BIIB", "BK", "BKNG", "BKR", "BLDR", "BLK", "BMY", "BR", "BRK-B",
"BRO", "BSX", "BX", "BXP", "C", "CAG", "CAH", "CARR", "CAT", "CB", "CBOE",
"CBRE", "CCI", "CCL", "CDNS", "CDW", "CEG", "CF", "CFG", "CHD", "CHRW",
"CHTR", "CI", "CINF", "CL", "CLX", "CMCSA", "CME", "CMG", "CMI", "CMS",
"CNC", "CNP", "COF", "COIN", "COO", "COP", "COR", "COST", "CPAY", "CPB",
"CPRT", "CPT", "CRL", "CRM", "CRWD", "CSCO", "CSGP", "CSX", "CTAS",
"CTRA", "CTSH", "CTVA", "CVS", "CVX", "D", "DAL", "DASH", "DAY", "DD",
"DDOG", "DE", "DECK", "DELL", "DG", "DGX", "DHI", "DHR", "DIS", "DLR",
"DLTR", "DOC", "DOV", "DOW", "DPZ", "DRI", "DTE", "DUK", "DVA", "DVN",
"DXCM", "EA", "EBAY", "ECL", "ED", "EFX", "EG", "EIX", "EL", "ELV", "EME",
"EMN", "EMR", "EOG", "EPAM", "EQIX", "EQR", "EQT", "ERIE", "ES", "ESS",
"ETN", "ETR", "EVRG", "EW", "EXC", "EXE", "EXPD", "EXPE", "EXR", "F",
"FANG", "FAST", "FCX", "FDS", "FDX",
          "FE", "FFIV", "FI", "FICO", "FIS", "FITB", "FOX", "FOXA", "FRT",
"FSLR", "FTNT", "FTV", "GD", "GDDY", "GE", "GEHC", "GEN", "GEV", "GILD",
"GIS", "GL", "GLW", "GM", "GNRC", "GOOG", "GOOGL", "GPC", "GPN", "GRMN",
"GS", "GWW", "HAL", "HAS", "HBAN", "HCA", "HD", "HIG", "HII", "HLT",
"HOLX", "HON", "HOOD", "HPE", "HPQ", "HRL", "HSIC", "HST", "HSY", "HUBB",
"HUM", "HWM", "IBKR", "IBM", "ICE", "IDXX", "IEX", "IFF", "INCY", "INTC",
"INTU", "INVH", "IP", "IPG", "IQV", "IR", "IRM", "ISRG", "IT", "ITW",
"IVZ", "J", "JBHT", "JBL", "JCI", "JKHY", "JNJ", "JPM", "K", "KDP", "KEY",
"KEYS", "KHC", "KIM", "KKR", "KLAC", "KMB", "KMI", "KMX", "KO", "KR",
"KVUE", "L", "LDOS", "LEN", "LH", "LHX", "LII", "LIN", "LKQ", "LLY",
"LMT", "LNT", "LOW", "LRCX", "LULU", "LUV", "LVS", "LW", "LYB", "LYV",
"MA", "MAA", "MAR", "MAS", "MCD", "MCHP", "MCK", "MCO", "MDLZ", "MDT",
```

```
"MET", "META", "MGM", "MHK", "MKC", "MLM", "MMC", "MMM", "MNST", "MO",
"MOH", "MOS", "MPC", "MPWR", "MRK", "MRNA", "MS", "MSCI", "MSFT", "MSI",
"MTB", "MTCH", "MTD", "MU", "NCLH", "NDAQ", "NDSN", "NEE", "NEM", "NFLX",
"NI", "NKE", "NOC", "NOW", "NRG", "NSC", "NTAP", "NTRS", "NUE", "NVDA",
"NVR", "NWS", "NWSA", "NXPI", "O", "ODFL", "OKE", "OMC", "ON", "ORCL",
"ORLY", "OTIS", "OXY", "PANW", "PAYC", "PAYX", "PCAR", "PCG", "PEG",
"PEP", "PFE", "PFG", "PG", "PGR", "PH", "PHM", "PKG", "PLD", "PLTR", "PM",
"PNC", "PNR", "PNW", "PODD", "POOL", "PPG", "PPL", "PRU", "PSA", "PSKY",
"PSX", "PTC", "PWR", "PYPL", "QCOM", "RCL", "REG", "REGN", "RF", "RJF",
"RL", "RMD", "ROK", "ROL", "ROP", "ROST", "RSG", "RTX", "RVTY", "SBAC",
"SBUX", "SCHW", "SHW", "SJM", "SLB", "SMCI", "SNA", "SNPS", "SO", "SOLV",
"SPG", "SPGI", "SRE", "STE", "STLD", "STT", "STX", "STZ", "SW", "SWK",
"SWKS", "SYF", "SYK", "SYY", "T", "TAP", "TDG", "TDY", "TECH", "TEL",
"TER", "TFC", "TGT", "TJX", "TKO", "TMO", "TMUS", "TPL", "TPR", "TRGP",
"TRMB", "TROW", "TRV", "TSCO", "TSLA", "TSN", "TT", "TTD", "TTWO", "TXN",
"TXT", "TYL", "UAL", "UBER", "UDR", "UHS", "ULTA", "UNH", "UNP", "UPS",
"URI", "USB", "V", "VICI", "VLO", "VLTO", "VMC", "VRSK", "VRSN", "VRTX",
"VST", "VTR", "VTRS", "VZ", "WAB", "WAT", "WBD", "WDAY", "WDC", "WEC",
"WELL", "WFC", "WM", "WMB", "WMT", "WRB", "WSM", "WST", "WTW", "WY",
"WYNN", "XEL", "XOM", "XYL", "XYZ", "YUM", "ZBH", "ZBRA", "ZTS"]  # <--
replace with your full list


TRAIN_YEARS   = 20
TEST_MONTHS   = 9
ROLL_Q_WIN    = 252
HORIZONS      = [10, 15, 20]

# --- Smoothing toggles & params ---
USE_EMA  = True
EMA_SPANS = [16, 20, 40]

USE_SMA  = True
SMA_WINDOWS = [5, 10, 15, 20, 30]

USE_WMA  = True
WMA_WINDOWS = [10, 15, 20]

USE_KAMA = True
KAMA_WINDOWS = [5, 10, 15, 20, 30]
KAMA_FAST = 2
KAMA_SLOW = 30
```

```
# Baseline windows
STD_ROLL_WINDOWS = [5, 10]

# Lift cutoffs to compute (top-q of scores vs event p90)
LIFT_QUANTILES = [0.90, 0.80, 0.70, 0.50]

# ---------- Bootstrap controls ----------
ENABLE_BOOTSTRAP = True
BOOT_ITERS = 500
BOOT_MIN_LEN = 0
BOOT_PCTS = [5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95]  #
requested set

# ---------- Helper Functions ----------
def get_shares_outstanding(ticker: yf.Ticker):
   try:
       fi = getattr(ticker, "fast_info", None)
       if fi and getattr(fi, "shares_outstanding", None):
           shares = float(fi.shares_outstanding)
       else:
           info = ticker.info
           shares = float(info.get("sharesOutstanding", 0) or 0)
       return shares if np.isfinite(shares) and shares > 0 else np.nan
   except Exception:
       return np.nan

def atr14_from_ohlc(high, low, close, window=14):
   prev_close = close.shift(1)
   tr = pd.concat([
       (high - low).abs(),
       (high - prev_close).abs(),
       (low  - prev_close).abs()
   ], axis=1).max(axis=1)
   return tr.rolling(window).mean()

def large_move_label_pct(abs_kret, q, roll_win, train_mask):
   thr = abs_kret.where(train_mask).rolling(
       roll_win, min_periods=max(10, roll_win//2)
   ).quantile(q).ffill()
   return (abs_kret >= thr).astype(float)
```

```
def lift_at_top_quantile(scores, labels, q):
    df = pd.DataFrame({"s": scores, "y": labels}).dropna()
    if len(df) == 0:
        return np.nan
    cutoff = np.quantile(df["s"], q)
    top = df[df["s"] >= cutoff]
    base = df["y"].mean()
    return np.nan if base == 0 else float(top["y"].mean() / base)

def partial_corr_multi(x, y, Z):
    data = pd.concat([x, y, Z], axis=1).dropna()
    if len(data) < 30:
        return np.nan
    Xz = sm.add_constant(data.iloc[:, 2:])
    rx = sm.OLS(data.iloc[:, 0], Xz).fit().resid
    ry = sm.OLS(data.iloc[:, 1], Xz).fit().resid
    return float(np.corrcoef(rx, ry)[0, 1])

def compute_pci2(high, low, vol, shares):
    if not np.isfinite(shares) or shares <= 0:
        return pd.Series(np.nan, index=high.index)
    mu  = ((shares + vol) * high) / shares
    lam = ((shares - vol) * low)  / shares
    denom = mu - lam
    eps = 1e-9 * np.nanmean(np.abs(denom))
    good = np.isfinite(denom) & (np.abs(denom) > eps)
    out = np.full(len(high), np.nan, dtype=float)
    hv, lv, dv = high.values, low.values, denom.values
    out[good] = (hv[good] - lv[good]) / dv[good]
    return pd.Series(out, index=high.index)

# --- Smoothing helpers ---
def ema_series(x: pd.Series, span: int) -> pd.Series:
    return x.ewm(span=span, adjust=False).mean()

def sma_series(x: pd.Series, window: int) -> pd.Series:
    return x.rolling(window, min_periods=max(2, window//2)).mean()

def wma_series(x: pd.Series, window: int) -> pd.Series:
    if window <= 1:
        return x.copy()
    weights = np.arange(1, window+1, dtype=float)
```

```
    def _wavg(vals):
        v = np.asarray(vals, dtype=float)
        if np.isnan(v).all():
            return np.nan
        mask = np.isfinite(v)
        if mask.sum() == 0:
            return np.nan
        w = weights[-mask.sum():]
        return np.dot(v[mask], w) / w.sum()
    return x.rolling(window, min_periods=max(2, window//2)).apply(_wavg,
raw=False)

def kama_series(x: pd.Series, er_window: int = 10, fast: int = 2, slow:
int = 30) -> pd.Series:
    x = x.astype(float)
    kama = np.full(len(x), np.nan, dtype=float)
    fastSC = 2.0 / (fast + 1.0)
    slowSC = 2.0 / (slow + 1.0)
    first = np.where(np.isfinite(x.values))[0]
    if len(first) == 0:
        return pd.Series(kama, index=x.index)
    start_idx = max(first[0], er_window)
    if start_idx >= er_window and
np.isfinite(x.iloc[start_idx-er_window:start_idx]).all():
        kama[start_idx] = x.iloc[start_idx-er_window:start_idx].mean()
    else:
        kama[start_idx] = x.iloc[start_idx]
    for t in range(start_idx+1, len(x)):
        px = x.iloc[t]
        if not np.isfinite(px):
            kama[t] = kama[t-1]
            continue
        if t - er_window < 0 or not np.isfinite(x.iloc[t-er_window]):
            er = 0.0
        else:
            change = abs(px - x.iloc[t-er_window])
            vol = np.abs(np.diff(x.iloc[t-er_window:t+1])).sum()
            er = 0.0 if vol == 0 or not np.isfinite(vol) else change / vol
        sc = (er * (fastSC - slowSC) + slowSC) ** 2
        kama[t] = kama[t-1] + sc * (px - kama[t-1])
    return pd.Series(kama, index=x.index)
```

```python
# ---------- Deterministic Spearman ----------
def fast_spearman(a: pd.Series, b: pd.Series) -> float:
    s = pd.concat([a, b], axis=1).dropna()
    if len(s) < 5:
        return np.nan
    ar = s.iloc[:, 0].rank(method="average").to_numpy()
    br = s.iloc[:, 1].rank(method="average").to_numpy()
    return float(np.corrcoef(ar, br)[0, 1])

# ---------- Bootstrap IC (now returns many percentiles AND gt0 share)
----------
def bootstrap_ic(x: pd.Series, y: pd.Series, n_iter: int = 500, min_len:
int = 20, pcts=None):
    """
    Nonparametric bootstrap for Spearman IC.
    Returns (mean, {pct->value}, gt0_share).
    """
    if pcts is None:
        pcts = [5, 50, 95]

    s = pd.concat([x, y], axis=1).dropna()
    n = len(s)
    if n < min_len:
        return np.nan, {p: np.nan for p in pcts}, np.nan

    # rank once (equivalent to Spearman)
    ar = s.iloc[:, 0].rank(method="average").to_numpy()
    br = s.iloc[:, 1].rank(method="average").to_numpy()

    # bootstrap indices
    idx = np.random.randint(0, n, size=(n_iter, n))
    A = ar[idx]
    B = br[idx]

    # center and compute correlation efficiently
    A = A - A.mean(axis=1, keepdims=True)
    B = B - B.mean(axis=1, keepdims=True)
    denom = (A.std(axis=1, ddof=1) * B.std(axis=1, ddof=1))
    vals = np.full(n_iter, np.nan, dtype=float)
    good = denom > 0
    if np.any(good):
```

```
        vals[good] = (A[good] * B[good]).sum(axis=1) / ((A.shape[1] - 1) *
denom[good])

    boot_mean = float(np.nanmean(vals))
    boot_pcts = {int(p): float(np.nanpercentile(vals, p)) for p in pcts}
    gt0_share = float(np.nanmean(vals > 0))  # share of alternate histories
with IC>0
    return boot_mean, boot_pcts, gt0_share

# ---------- Main Evaluation ----------
results = []

for sym in TICKERS:
    try:
        ticker = yf.Ticker(sym)
        end = dt.datetime.utcnow()
        start = end - dt.timedelta(days=int(365.25*TRAIN_YEARS))
        hist = ticker.history(start=start.date(), end=end.date(),
                              interval="1d", auto_adjust=True)

        if hist.empty:
            print(f"[{sym}] No price data.")
            continue

        df =
hist.reset_index()[["Date","Open","High","Low","Close","Volume"]].copy()
        df["Date"]   = pd.to_datetime(df["Date"])
        df["ret"]    = df["Close"].pct_change()
        df["absret"] = df["ret"].abs()

        # --- Multiple rolling-std baselines (dynamic) ---
        for w in STD_ROLL_WINDOWS:
            w = int(w)
            df[f"roll_std{w}"] = df["ret"].rolling(w).std()
            df[f"std_vol_ratio_{w}"] = df[f"roll_std{w}"] / df["Volume"]

        # Other baselines
        df["ATR14"] = atr14_from_ohlc(df["High"], df["Low"], df["Close"],
14)
        df["absret_mom5"] = df["absret"].rolling(5).sum()
        df["vol_z"] = (df["Volume"] - df["Volume"].rolling(20).mean()) /
df["Volume"].rolling(20).std()
```

```
        shares = get_shares_outstanding(ticker)
        if not np.isfinite(shares) or shares <= 0:
            print(f"[{sym}] Skipping (no shares outstanding).")
            continue

        # Raw PCI2
        df["PCI2"] = compute_pci2(df["High"], df["Low"], df["Volume"],
shares)

        # Dynamic training length if history is short
        available_years = (df["Date"].max() - df["Date"].min()).days /
365.25
        train_years_eff = min(TRAIN_YEARS, max(2, int(available_years //
2)))
        start_date, last_date = df["Date"].min(), df["Date"].max()

        # -------- Baseline feature names (include ATR14 as a baseline)
--------
        baseline_cols = ["vol_z", "ATR14"]
        baseline_cols += [f"roll_std{int(w)}" for w in STD_ROLL_WINDOWS]
        baseline_cols += [f"std_vol_ratio_{int(w)}" for w in
STD_ROLL_WINDOWS]

        while True:
            train_start = start_date
            train_end   = train_start +
pd.DateOffset(years=train_years_eff)
            test_end    = train_end + pd.DateOffset(months=TEST_MONTHS)
            if test_end > last_date:
                break

            train_mask = (df["Date"] >= train_start) & (df["Date"] <
train_end)
            test_mask  = (df["Date"] >= train_end)   & (df["Date"] <
test_end)

            # ---------- Build all smoothed PCI2 ONCE per fold ----------
            smoothed_series = {}
            if USE_EMA:
                for span in EMA_SPANS:
```

```
                    smoothed_series[f"PCI2_ema{int(span)}"] =
ema_series(df["PCI2"], int(span))
            if USE_SMA:
                for w in SMA_WINDOWS:
                    smoothed_series[f"PCI2_sma{int(w)}"] =
sma_series(df["PCI2"], int(w))
            if USE_WMA:
                for w in WMA_WINDOWS:
                    smoothed_series[f"PCI2_wma{int(w)}"] =
wma_series(df["PCI2"], int(w))
            if USE_KAMA:
                for n in KAMA_WINDOWS:
                    smoothed_series[f"PCI2_kama{int(n)}"] =
kama_series(df["PCI2"], int(n), KAMA_FAST, KAMA_SLOW)

            # ---------- Cache Z matrices for partial corr ----------
            Z_all = df.loc[test_mask, baseline_cols + ["absret_mom5"]]
            Z_by_base = {
                base: df.loc[test_mask, [c for c in baseline_cols if c !=
base] + ["absret_mom5"]]
                for base in baseline_cols
            }

            for h in HORIZONS:
                if test_mask.sum() < max(25, h+5):
                    continue

                df[f"fut_kret{h}"] = df["Close"].pct_change(h).shift(-h)
                df[f"abs_kret{h}"] = df[f"fut_kret{h}"].abs()

                # Labels for "big move" events (TRAIN-only thresholds)
                evt_p90 = large_move_label_pct(df[f"abs_kret{h}"], 0.90,
ROLL_Q_WIN, train_mask)
                evt_p80 = large_move_label_pct(df[f"abs_kret{h}"], 0.80,
ROLL_Q_WIN, train_mask)

                # Prepare slices used repeatedly
                y_test = df.loc[test_mask, f"abs_kret{h}"]
                evt_p90_test = evt_p90.loc[test_mask]
                evt_p80_test = evt_p80.loc[test_mask]

                # ------ Baselines (including ATR14) ------
```

```
            for base in baseline_cols:
                s_test = df.loc[test_mask, base]

                ic = fast_spearman(s_test, y_test)

                # Bootstrap IC diagnostics with many percentiles + gt0
share
                if ENABLE_BOOTSTRAP:
                    ic_b_mean, ic_b_pct, ic_b_gt0 =
bootstrap_ic(s_test, y_test, BOOT_ITERS, BOOT_MIN_LEN, BOOT_PCTS)
                else:
                    ic_b_mean, ic_b_pct, ic_b_gt0 = np.nan, {p: np.nan
for p in BOOT_PCTS}, np.nan

                # Lifts
                lift_vals = {}
                for q in LIFT_QUANTILES:
                    lift_vals[f"Lift@top{int(q*100)}"] =
lift_at_top_quantile(
                        s_test, evt_p90_test, q
                    )

                pc = partial_corr_multi(s_test, y_test,
Z_by_base[base])

                row = {
                    "Symbol": sym, "Horizon": h, "Score": base,
                    "IC": ic,
                    "PartialCorr": pc,
                    "Lift@p80": lift_at_top_quantile(s_test,
evt_p80_test, 0.80),
                }
                if ENABLE_BOOTSTRAP:
                    row["IC_boot_mean"] = ic_b_mean
                    for p, val in ic_b_pct.items():
                        row[f"IC_boot_p{int(p):02d}"] = val
                    row["IC_boot_gt0_share"] = ic_b_gt0

                row.update(lift_vals)
                results.append(row)

            # ------ Smoothed PCI2 ------
```

```
            for name, series in smoothed_series.items():
                s_test = series.loc[test_mask]

                ic = fast_spearman(s_test, y_test)

                if ENABLE_BOOTSTRAP:
                    ic_b_mean, ic_b_pct, ic_b_gt0 =
bootstrap_ic(s_test, y_test, BOOT_ITERS, BOOT_MIN_LEN, BOOT_PCTS)
                else:
                    ic_b_mean, ic_b_pct, ic_b_gt0 = np.nan, {p: np.nan
for p in BOOT_PCTS}, np.nan

                lift_vals = {}
                for q in LIFT_QUANTILES:
                    lift_vals[f"Lift@top{int(q*100)}"] =
lift_at_top_quantile(
                        s_test, evt_p90_test, q
                    )

                pc = partial_corr_multi(s_test, y_test, Z_all)

                row = {
                    "Symbol": sym, "Horizon": h, "Score": name,
                    "IC": ic,
                    "PartialCorr": pc,
                    "Lift@p80": lift_at_top_quantile(s_test,
evt_p80_test, 0.80),
                }
                if ENABLE_BOOTSTRAP:
                    row["IC_boot_mean"] = ic_b_mean
                    for p, val in ic_b_pct.items():
                        row[f"IC_boot_p{int(p):02d}"] = val
                    row["IC_boot_gt0_share"] = ic_b_gt0

                row.update(lift_vals)
                results.append(row)

            start_date = start_date + pd.DateOffset(months=TEST_MONTHS)

    except Exception as e:
        print(f"[{sym}] Error: {e}")
```

```
# ---------- Scorecard ----------
if len(results) == 0:
   raise ValueError("No results were collected. Check tickers or
parameters.")

resdf = pd.DataFrame(results)

# Figure out all Lift column names dynamically
lift_cols = sorted([c for c in resdf.columns if c.startswith("Lift@top")])
lift_cols_legacy = ["Lift@p80"] if "Lift@p80" in resdf.columns else []

# Identify bootstrap percentile columns that exist
boot_pct_cols = sorted([c for c in resdf.columns if
c.startswith("IC_boot_p")])

scorecards = []
for (h, score), sub in resdf.groupby(["Horizon","Score"]):
   row = {
       "Horizon": h,
       "Score": score,
       "IC_mean": sub["IC"].mean(),
       # IC median is the median of the per-fold bootstrap mean ICs
(fallback to raw IC median if absent)
       "IC_median": sub["IC_boot_mean"].median() if "IC_boot_mean" in
sub.columns else sub["IC"].median(),
       "IC>0_share": (sub["IC"] > 0).mean(),
       # NEW: average share of bootstrapped alternate histories with IC>0
across folds
       "bootIC>0_share": sub["IC_boot_gt0_share"].mean() if
"IC_boot_gt0_share" in sub.columns else np.nan,
       "PartialCorr_mean": sub["PartialCorr"].mean(),
       "N_obs": len(sub)
   }
   # Aggregate bootstrap percentiles across folds (mean, like previous
lo/hi handling)
   for c in boot_pct_cols:
       row[f"{c}_mean"] = sub[c].mean()

   for lc in lift_cols:
       row[f"{lc}_mean"] = sub[lc].mean()
   for lc in lift_cols_legacy:
       row[f"{lc}_mean"] = sub[lc].mean()
```

```
    scorecards.append(row)

scorecard_df = pd.DataFrame(scorecards).sort_values(["Horizon","Score"])
print(scorecard_df.round(3))

# ---------- Plotting ----------
def plot_metric(df, metric, title):
    plt.figure(figsize=(10,5))
    for score in sorted(df["Score"].unique()):
        subset = df[df["Score"]==score].sort_values("Horizon")
        if metric in subset.columns:
            plt.plot(subset["Horizon"], subset[metric], marker='o',
label=score)
    plt.title(title)
    plt.xlabel("Horizon (days)")
    plt.ylabel(metric)
    plt.legend(bbox_to_anchor=(1.05,1), loc='upper left')
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

# Plot the IC median (based on bootstrap means)
plot_metric(scorecard_df, "IC_median", "IC_median (median of bootstrap
means) by Horizon")

# ---------- Save Results ----------
# Save scorecard to Excel
scorecard_df.to_excel("scorecard_PCI2_multi_smoothing.xlsx", index=False)

# Keep detailed per-fold results as CSV
resdf.to_csv("fold_results_PCI2_multi_smoothing.csv", index=False)
```

**Cell 2**

```
# --- Libraries ---
import datetime as dt
import numpy as np
import pandas as pd
import yfinance as yf
```

```
import statsmodels.api as sm
from scipy.stats import spearmanr  # kept for compatibility/reference
import matplotlib.pyplot as plt

# ---------- Parameters ----------
TICKERS = ["NVDA", "AAPL", "MSFT", "AMZN", "META", "AVGO", "GOOGL",
"GOOG", "TSLA", "BRK-B", "WMT", "JPM", "ORCL", "LLY", "V", "MA", "XOM",
"NFLX", "JNJ", "PLTR", "COST", "AMD", "ABBV", "BAC", "HD", "PG", "UNH",
"GE", "CVX", "KO", "IBM", "CSCO", "WFC", "MS", "AXP", "MU", "PM", "CAT",
"TMUS", "CRM"]

TRAIN_YEARS   = 20  # max years to request; will auto-shrink if not
available
TEST_MONTHS   = 9
ROLL_Q_WIN    = 252  # label window for event thresholds (built on TRAIN
only)
HORIZONS      = [10, 15, 20, 30, 40, 100]

# (Keep these if you use them elsewhere, but EMA now uses EMA_SPANS only)
SMOOTH_SPANS  = [16, 20, 40]
STD_ROLL_WINDOWS = [5, 10]
LIFT_QUANTILES  = [0.90, 0.80, 0.70, 0.50]
STD_LIMITS      = [0.0]  # 0.0 = use the full test set for IC/PC (no top-q
filter)

# --- Smoothing strategy toggles/params ---
USE_EMA  = True
EMA_SPANS = [16, 20, 40]

USE_SMA  = True
SMA_WINDOWS = [5, 10, 15, 20, 30]

USE_WMA  = True
WMA_WINDOWS = [5, 10, 15, 20, 30]

USE_KAMA = True
KAMA_WINDOWS = [5, 10, 15, 20, 30]
KAMA_FAST = 2
KAMA_SLOW = 30

# --- Bootstrap diagnostics ---
ENABLE_BOOTSTRAP = True
```

```
BOOT_ITERS = 500
BOOT_PCTS = list(range(5, 100, 5))  # 5,10,15,...,95

# ---------- Helper Functions ----------

def get_shares_outstanding(ticker: yf.Ticker):
    try:
        fi = getattr(ticker, "fast_info", None)
        if fi and getattr(fi, "shares_outstanding", None):
            shares = float(fi.shares_outstanding)
        else:
            info = ticker.info
            shares = float(info.get("sharesOutstanding", 0) or 0)
        return shares if np.isfinite(shares) and shares > 0 else np.nan
    except Exception:
        return np.nan

def atr14_from_ohlc(high, low, close, window=14):
    prev_close = close.shift(1)
    tr = pd.concat([
        (high - low).abs(),
        (high - prev_close).abs(),
        (low  - prev_close).abs()
    ], axis=1).max(axis=1)
    return tr.rolling(window).mean()

def large_move_label_pct(series, q, roll_win, train_mask):
    """TRAIN-only rolling quantile threshold -> label high (1) vs not
(0)."""
    thr = series.where(train_mask).rolling(
        roll_win, min_periods=max(10, roll_win//2)
    ).quantile(q).ffill()
    return (series >= thr).astype(float)

def lift_at_top_quantile(scores, labels, q):
    s = np.asarray(scores, dtype=float)
    y = np.asarray(labels, dtype=float)
    mask = np.isfinite(s) & np.isfinite(y)
    if mask.sum() == 0:
        return np.nan
    s = s[mask]; y = y[mask]
    if y.size == 0:
```

```
        return np.nan
    cutoff = np.quantile(s, q)
    top = y[s >= cutoff]
    if top.size == 0:
        return np.nan
    base = y.mean()
    return np.nan if base == 0 else float(top.mean() / base)

# --- Cached-QR residualizer for partial corr (identical math to OLS
residuals) ---
def _make_residualizer(Z_df: pd.DataFrame):
    Z_clean = Z_df.dropna()
    if Z_clean.empty:
        return None, None, None
    X = sm.add_constant(Z_clean).to_numpy(dtype=float)
    Q, _ = np.linalg.qr(X, mode='reduced')
    P = np.eye(Q.shape[0]) - Q @ Q.T
    valid_mask = Z_df.notna().all(axis=1).to_numpy()
    return P, valid_mask, None

def _partial_corr_with_P(s_arr, y_arr, Z_df, P, Z_valid_mask, test_len):
    if P is None or Z_valid_mask is None:
        return np.nan
    z_idx = np.where(Z_valid_mask)[0]
    pos_map = -np.ones(test_len, dtype=int)
    pos_map[z_idx] = np.arange(len(z_idx), dtype=int)

    finite_xy = np.isfinite(s_arr) & np.isfinite(y_arr)
    keep_mask = finite_xy & Z_valid_mask
    if keep_mask.sum() < 30:
        return np.nan

    idx_test = np.where(keep_mask)[0]
    idx_in_P = pos_map[idx_test]
    P_sub = P[np.ix_(idx_in_P, idx_in_P)]

    sx = s_arr[idx_test]
    sy = y_arr[idx_test]

    rx = P_sub @ sx
    ry = P_sub @ sy
```

```
    if rx.size < 5:
        return np.nan
    c = np.corrcoef(rx, ry)
    return float(c[0, 1])

def compute_pci2(high, low, vol, shares):
    """PCI2 = (High-Low)/( ((S+V)*High - (S-V)*Low)/S ), with guards."""
    if not np.isfinite(shares) or shares <= 0:
        return pd.Series(np.nan, index=high.index)
    mu  = ((shares + vol) * high) / shares
    lam = ((shares - vol) * low)  / shares
    denom = mu - lam
    eps = 1e-9 * np.nanmean(np.abs(denom))
    good = np.isfinite(denom) & (np.abs(denom) > eps)
    out = np.full(len(high), np.nan, dtype=float)
    hv, lv, dv = high.values, low.values, denom.values
    out[good] = (hv[good] - lv[good]) / dv[good]
    return pd.Series(out, index=high.index)

def forward_realized_vol(ret: pd.Series, h: int, min_frac: float = 0.8) ->
pd.Series:
    """RV over next h days: std(ret_{t+1..t+h}); reverse-rolling trick."""
    min_periods = max(2, int(h * min_frac))
    fwd_std = ret.shift(-1)[::-1].rolling(window=h,
min_periods=min_periods).std()[::-1]
    return fwd_std

# ---------- Smoothing helpers ----------
def ema_series(x: pd.Series, span: int) -> pd.Series:
    return x.ewm(span=span, adjust=False).mean()

def sma_series(x: pd.Series, window: int) -> pd.Series:
    return x.rolling(window, min_periods=max(2, window//2)).mean()

def wma_series(x: pd.Series, window: int) -> pd.Series:
    if window <= 1:
        return x.copy()
    weights = np.arange(1, window+1, dtype=float)
    def _wavg(vals):
        v = np.asarray(vals, dtype=float)
        mask = np.isfinite(v)
        if mask.sum() == 0:
```

```
            return np.nan
        w = weights[-mask.sum():]
        return np.dot(v[mask], w) / w.sum()
    return x.rolling(window, min_periods=max(2, window//2)).apply(_wavg,
raw=False)

def kama_series(x: pd.Series, er_window: int = 10, fast: int = 2, slow:
int = 30) -> pd.Series:
    x = x.astype(float)
    kama = np.full(len(x), np.nan, dtype=float)
    fastSC = 2.0 / (fast + 1.0)
    slowSC = 2.0 / (slow + 1.0)
    first = np.where(np.isfinite(x.values))[0]
    if len(first) == 0:
        return pd.Series(kama, index=x.index)
    start_idx = max(first[0], er_window)
    kama[start_idx] = np.nanmean(x.iloc[start_idx-er_window:start_idx]) if
start_idx >= er_window else x.iloc[start_idx]
    for t in range(start_idx+1, len(x)):
        px = x.iloc[t]
        if not np.isfinite(px):
            kama[t] = kama[t-1]
            continue
        if t - er_window < 0 or not np.isfinite(x.iloc[t-er_window]):
            er = 0.0
        else:
            change = abs(px - x.iloc[t-er_window])
            vol = np.abs(np.diff(x.iloc[t-er_window:t+1])).sum()
            er = 0.0 if vol == 0 or not np.isfinite(vol) else change / vol
        sc = (er * (fastSC - slowSC) + slowSC) ** 2
        kama[t] = kama[t-1] + sc * (px - kama[t-1])
    return pd.Series(kama, index=x.index)

# ---------- FAST STATS ----------
def fast_spearman(a: pd.Series, b: pd.Series) -> float:
    s = pd.concat([a, b], axis=1).dropna()
    if len(s) < 5:
        return np.nan
    ar = s.iloc[:, 0].rank(method="average").to_numpy()
    br = s.iloc[:, 1].rank(method="average").to_numpy()
    return float(np.corrcoef(ar, br)[0, 1])
```

```
def bootstrap_ic(x: pd.Series, y: pd.Series, n_iter: int = 500, min_len:
int = 20, pcts=None):
    """
    Returns:
      mean_of_bootstrap_ic, {pct: value, ...} for requested percentiles,
gt0_share
    """
    if pcts is None:
        pcts = BOOT_PCTS
    s = pd.concat([x, y], axis=1).dropna()
    n = len(s)
    if n < min_len:
        return np.nan, {p: np.nan for p in pcts}, np.nan
    ar = s.iloc[:, 0].rank(method="average").to_numpy()
    br = s.iloc[:, 1].rank(method="average").to_numpy()

    idx = np.random.randint(0, n, size=(n_iter, n))
    A = ar[idx]; B = br[idx]

    A = A - A.mean(axis=1, keepdims=True)
    B = B - B.mean(axis=1, keepdims=True)
    denom = (A.std(axis=1, ddof=1) * B.std(axis=1, ddof=1))
    vals = np.full(n_iter, np.nan, dtype=float)
    good = denom > 0
    if np.any(good):
        vals[good] = (A[good] * B[good]).sum(axis=1) / ((A.shape[1] - 1) *
denom[good])

    mean_val = float(np.nanmean(vals))
    pct_vals = np.nanpercentile(vals, pcts)
    pct_dict = {int(p): float(v) for p, v in zip(pcts, pct_vals)}
    gt0_share = float(np.nanmean(vals > 0))  # NEW: share of alternate
histories with IC>0
    return mean_val, pct_dict, gt0_share

# ---------- Main Evaluation ----------
results = []

# --- Batch download all price history once ---
end = dt.datetime.utcnow()
start = end - dt.timedelta(days=int(365.25 * TRAIN_YEARS))
all_hist = yf.download(TICKERS, start=start.date(), end=end.date(),
```

```
                    interval="1d", auto_adjust=True, group_by='ticker',
threads=True, progress=False)

for sym in TICKERS:
   try:
       # Extract per-ticker frame from the batch download; fallback to
per-ticker if missing
       if isinstance(all_hist.columns, pd.MultiIndex) and sym in
all_hist.columns.get_level_values(0):
           h = all_hist[sym].copy()
           h.columns = [c.capitalize() for c in h.columns]
           hist = h.reset_index()
       else:
           ticker_tmp = yf.Ticker(sym)
           h = ticker_tmp.history(start=start.date(), end=end.date(),
interval="1d", auto_adjust=True)
           hist = h.reset_index()

       if hist.empty:
           print(f"[{sym}] No price data.")
           continue

       df = hist[["Date","Open","High","Low","Close","Volume"]].copy()
       df["Date"]   = pd.to_datetime(df["Date"])
       df["ret"]    = df["Close"].pct_change()
       df["absret"] = df["ret"].abs()

       # --- Multiple rolling-std baselines ---
       for w in STD_ROLL_WINDOWS:
           w = int(w)
           df[f"roll_std{w}"] = df["ret"].rolling(w).std()
           df[f"std_vol_ratio_{w}"] = df[f"roll_std{w}"] / df["Volume"]

       # --- ATR14 baseline + other features ---
       df["ATR14"] = atr14_from_ohlc(df["High"], df["Low"], df["Close"],
14)
       df["absret_mom5"] = df["absret"].rolling(5).sum()
       df["vol_z"] = (df["Volume"] - df["Volume"].rolling(20).mean()) /
df["Volume"].rolling(20).std()

       # Shares outstanding (for PCI2)
       ticker = yf.Ticker(sym)
```

```
        shares = get_shares_outstanding(ticker)
        if not np.isfinite(shares) or shares <= 0:
            print(f"[{sym}] Skipping (no shares outstanding).")
            continue

        df["PCI2"] = compute_pci2(df["High"], df["Low"], df["Volume"],
shares)

        # Dynamic training length if history is short
        available_years = (df["Date"].max() - df["Date"].min()).days /
365.25
        train_years_eff = min(TRAIN_YEARS, max(2, int(available_years //
2)))
        start_date, last_date = df["Date"].min(), df["Date"].max()

        # Baselines to evaluate (ATR14 INCLUDED as a first-class baseline)
        baseline_cols = ["ATR14", "vol_z"]
        baseline_cols += [f"roll_std{int(w)}" for w in STD_ROLL_WINDOWS]
        baseline_cols += [f"std_vol_ratio_{int(w)}" for w in
STD_ROLL_WINDOWS]

        while True:
            train_start = start_date
            train_end   = train_start +
pd.DateOffset(years=train_years_eff)
            test_end    = train_end + pd.DateOffset(months=TEST_MONTHS)
            if test_end > last_date:
                break

            train_mask = (df["Date"] >= train_start) & (df["Date"] <
train_end)
            test_mask  = (df["Date"] >= train_end)   & (df["Date"] <
test_end)
            if test_mask.sum() < 5:
                start_date = start_date + pd.DateOffset(months=TEST_MONTHS)
                continue

            # ---------- Build all smoothers ONCE per fold ----------
            smoothed_series = {}
            if USE_EMA:
                for span in EMA_SPANS:
```

```
                    smoothed_series[f"PCI2_ema{int(span)}"] =
ema_series(df["PCI2"], int(span))
            if USE_SMA:
                for w in SMA_WINDOWS:
                    smoothed_series[f"PCI2_sma{int(w)}"] =
sma_series(df["PCI2"], int(w))
            if USE_WMA:
                for w in WMA_WINDOWS:
                    smoothed_series[f"PCI2_wma{int(w)}"] =
wma_series(df["PCI2"], int(w))
            if USE_KAMA:
                for n in KAMA_WINDOWS:
                    smoothed_series[f"PCI2_kama{int(n)}"] =
kama_series(df["PCI2"], int(n), KAMA_FAST, KAMA_SLOW)

            # ---------- Hoist arrays at test slice ----------
            test_len = test_mask.sum()

            # Cache y for all horizons at once
            for h in HORIZONS:
                df[f"fwd_rvol{h}"] = forward_realized_vol(df["ret"], h,
min_frac=0.8)
            y_by_h = {h: df.loc[test_mask, f"fwd_rvol{h}"].to_numpy() for h
in HORIZONS}

            # ---------- Cache Z matrices & residualizers ----------
            Z_all_df = df.loc[test_mask, baseline_cols + ["absret_mom5"]]
            P_all, Z_all_valid_mask, _ = _make_residualizer(Z_all_df)

            Z_by_base = {
                base: df.loc[test_mask, [c for c in baseline_cols if c !=
base] + ["absret_mom5"]]
                for base in baseline_cols
            }
            P_by_base = {}
            Z_valid_by_base = {}
            for base, Zdf in Z_by_base.items():
                P_b, Z_b_valid, _ = _make_residualizer(Zdf)
                P_by_base[base] = P_b
                Z_valid_by_base[base] = Z_b_valid

            # ---------- Main eval ----------
```

```
            for h in HORIZONS:
                if test_len < max(25, h + 5):
                    continue

                y_series = df[f"fwd_rvol{h}"]
                evt_labels = {q: large_move_label_pct(y_series, q,
ROLL_Q_WIN, train_mask) for q in STD_LIMITS}
                evt_p90 = evt_labels.get(0.90,
large_move_label_pct(y_series, 0.90, ROLL_Q_WIN, train_mask))
                evt_p80 = large_move_label_pct(y_series, 0.80, ROLL_Q_WIN,
train_mask)
                y_test_np = y_by_h[h]

                evt_p90_arr = evt_p90.loc[test_mask].to_numpy()
                evt_p80_arr = evt_p80.loc[test_mask].to_numpy()

                # ---------- Baselines (includes ATR14 now) ----------
                for base in baseline_cols:
                    s_test_np = df.loc[test_mask, base].to_numpy()

                    ic_primary = np.nan
                    pc_primary = np.nan
                    dyn_vals = {}

                    for i, q in enumerate(STD_LIMITS):
                        mask_q = (evt_labels[q].loc[test_mask] ==
1.0).to_numpy() if q > 0 else np.ones(test_len, dtype=bool)

                        s_sub = pd.Series(s_test_np[mask_q])
                        y_sub = pd.Series(y_test_np[mask_q])
                        if s_sub.dropna().shape[0] >= 5 and
y_sub.dropna().shape[0] >= 5:
                            ic_q = fast_spearman(s_sub, y_sub)
                            pc_q = _partial_corr_with_P(s_test_np,
y_test_np, Z_by_base[base], P_by_base[base], Z_valid_by_base[base],
test_len)
                            if ENABLE_BOOTSTRAP:
                                ic_b_mean, ic_b_pcts, ic_b_gt0 =
bootstrap_ic(s_sub, y_sub, BOOT_ITERS, pcts=BOOT_PCTS)
                            else:
                                ic_b_mean, ic_b_pcts, ic_b_gt0 = np.nan,
{p: np.nan for p in BOOT_PCTS}, np.nan
```

```
                    else:
                        ic_q = pc_q = np.nan
                        ic_b_mean, ic_b_pcts, ic_b_gt0 = np.nan, {p:
np.nan for p in BOOT_PCTS}, np.nan

                    dyn_vals[f"IC@top{int(q*100)}"] = ic_q
                    dyn_vals[f"PartialCorr@top{int(q*100)}"] = pc_q
                    if i == 0:
                        ic_primary = ic_q
                        pc_primary = pc_q
                        if ENABLE_BOOTSTRAP:
                            dyn_vals["IC_boot_mean"] = ic_b_mean
                            for p, v in ic_b_pcts.items():
                                dyn_vals[f"IC_boot_p{int(p)}"] = v
                            # NEW per-fold metric
                            dyn_vals["IC_boot_gt0_share"] = ic_b_gt0

                lift_vals = {}
                for q in LIFT_QUANTILES:
                    lift_vals[f"Lift@top{int(q*100)}"] =
lift_at_top_quantile(s_test_np, evt_p90_arr, q)

                row = {
                    "Symbol": sym, "Horizon": h, "Score": base,
                    "IC": ic_primary, "PartialCorr": pc_primary,
                    "Lift@p80": lift_at_top_quantile(s_test_np,
evt_p80_arr, 0.80),
                }
                row.update(lift_vals)
                row.update(dyn_vals)
                results.append(row)

            # ---------- Smoothed PCI2 (EMA/SMA/WMA/KAMA) ----------
            # EMA only block
            for name, series in smoothed_series.items():
                if not name.startswith("PCI2_ema"):
                    continue
                s_test_np = series.loc[test_mask].to_numpy()

                ic_primary = np.nan
                pc_primary = np.nan
                dyn_vals = {}
```

```
                    for i, q in enumerate(STD_LIMITS):
                        mask_q = (evt_labels[q].loc[test_mask] ==
1.0).to_numpy() if q > 0 else np.ones(test_len, dtype=bool)

                        s_sub = pd.Series(s_test_np[mask_q])
                        y_sub = pd.Series(y_test_np[mask_q])
                        if s_sub.dropna().shape[0] >= 5 and
y_sub.dropna().shape[0] >= 5:
                            ic_q = fast_spearman(s_sub, y_sub)
                            pc_q = _partial_corr_with_P(s_test_np,
y_test_np, Z_all_df, P_all, Z_all_valid_mask, test_len)
                            if ENABLE_BOOTSTRAP:
                                ic_b_mean, ic_b_pcts, ic_b_gt0 =
bootstrap_ic(s_sub, y_sub, BOOT_ITERS, pcts=BOOT_PCTS)
                            else:
                                ic_b_mean, ic_b_pcts, ic_b_gt0 = np.nan,
{p: np.nan for p in BOOT_PCTS}, np.nan
                        else:
                            ic_q = pc_q = np.nan
                            ic_b_mean, ic_b_pcts, ic_b_gt0 = np.nan, {p:
np.nan for p in BOOT_PCTS}, np.nan

                        dyn_vals[f"IC@top{int(q*100)}"] = ic_q
                        dyn_vals[f"PartialCorr@top{int(q*100)}"] = pc_q
                        if i == 0:
                            ic_primary = ic_q
                            pc_primary = pc_q
                            if ENABLE_BOOTSTRAP:
                                dyn_vals["IC_boot_mean"] = ic_b_mean
                                for p, v in ic_b_pcts.items():
                                    dyn_vals[f"IC_boot_p{int(p)}"] = v
                                # NEW per-fold metric
                                dyn_vals["IC_boot_gt0_share"] = ic_b_gt0

                    lift_vals = {}
                    for q in LIFT_QUANTILES:
                        lift_vals[f"Lift@top{int(q*100)}"] =
lift_at_top_quantile(s_test_np, evt_p90_arr, q)

                    row = {
                        "Symbol": sym, "Horizon": h, "Score": name,
```

```
                    "IC": ic_primary, "PartialCorr": pc_primary,
                    "Lift@p80": lift_at_top_quantile(s_test_np,
evt_p80_arr, 0.80),
                }
                row.update(lift_vals)
                row.update(dyn_vals)
                results.append(row)

            # ---------- Additional smoothing strategies (SMA/WMA/KAMA)
----------
            for prefix in ("PCI2_sma", "PCI2_wma", "PCI2_kama"):
                for name, series in smoothed_series.items():
                    if not name.startswith(prefix):
                        continue
                    s_test_np = series.loc[test_mask].to_numpy()

                    ic_primary = np.nan
                    pc_primary = np.nan
                    dyn_vals = {}

                    for i, q in enumerate(STD_LIMITS):
                        mask_q = (evt_labels[q].loc[test_mask] ==
1.0).to_numpy() if q > 0 else np.ones(test_len, dtype=bool)

                        s_sub = pd.Series(s_test_np[mask_q])
                        y_sub = pd.Series(y_test_np[mask_q])
                        if s_sub.dropna().shape[0] >= 5 and
y_sub.dropna().shape[0] >= 5:
                            ic_q = fast_spearman(s_sub, y_sub)
                            pc_q = _partial_corr_with_P(s_test_np,
y_test_np, Z_all_df, P_all, Z_all_valid_mask, test_len)
                            if ENABLE_BOOTSTRAP:
                                ic_b_mean, ic_b_pcts, ic_b_gt0 =
bootstrap_ic(s_sub, y_sub, BOOT_ITERS, pcts=BOOT_PCTS)
                            else:
                                ic_b_mean, ic_b_pcts, ic_b_gt0 =
np.nan, {p: np.nan for p in BOOT_PCTS}, np.nan
                        else:
                            ic_q = pc_q = np.nan
                            ic_b_mean, ic_b_pcts, ic_b_gt0 = np.nan,
{p: np.nan for p in BOOT_PCTS}, np.nan
```

```
                        dyn_vals[f"IC@top{int(q*100)}"] = ic_q
                        dyn_vals[f"PartialCorr@top{int(q*100)}"] = pc_q
                        if i == 0:
                            ic_primary = ic_q
                            pc_primary = pc_q
                            if ENABLE_BOOTSTRAP:
                                dyn_vals["IC_boot_mean"] = ic_b_mean
                                for p, v in ic_b_pcts.items():
                                    dyn_vals[f"IC_boot_p{int(p)}"] = v
                                # NEW per-fold metric
                                dyn_vals["IC_boot_gt0_share"] =
ic_b_gt0

                    lift_vals = {}
                    for q in LIFT_QUANTILES:
                        lift_vals[f"Lift@top{int(q*100)}"] =
lift_at_top_quantile(s_test_np, evt_p90_arr, q)

                    row = {
                        "Symbol": sym, "Horizon": h, "Score": name,
                        "IC": ic_primary, "PartialCorr": pc_primary,
                        "Lift@p80": lift_at_top_quantile(s_test_np,
evt_p80_arr, 0.80),
                    }
                    row.update(lift_vals)
                    row.update(dyn_vals)
                    results.append(row)

            start_date = start_date + pd.DateOffset(months=TEST_MONTHS)

    except Exception as e:
        print(f"[{sym}] Error: {e}")

# ---------- Scorecard ----------
if len(results) == 0:
    raise ValueError("No results were collected. Check tickers or
parameters.")

resdf = pd.DataFrame(results)

lift_cols = sorted([c for c in resdf.columns if c.startswith("Lift@top")])
lift_cols_legacy = ["Lift@p80"] if "Lift@p80" in resdf.columns else []
```

```
ic_top_cols = sorted([c for c in resdf.columns if c.startswith("IC@top")])
pc_top_cols = sorted([c for c in resdf.columns if
c.startswith("PartialCorr@top")])

scorecards = []
for (h, score), sub in resdf.groupby(["Horizon","Score"]):
   row = {
       "Horizon": h,
       "Score": score,
       "IC_mean": sub["IC"].mean(),
       # --- CHANGE: IC_median is now the median of bootstrap means (if
available)
       "IC_median": sub["IC_boot_mean"].median() if ENABLE_BOOTSTRAP and
"IC_boot_mean" in sub.columns else sub["IC"].median(),
       "IC>0_share": (sub["IC"] > 0).mean(),
       # NEW aggregated bootstrap share:
       "bootIC>0_share": sub["IC_boot_gt0_share"].mean() if
"IC_boot_gt0_share" in sub.columns else np.nan,
       "PartialCorr_mean": sub["PartialCorr"].mean(),
       "N_obs": len(sub),
   }
   # --- Bootstrap summary to scorecard: mean across folds of each
percentile
   if ENABLE_BOOTSTRAP and "IC_boot_mean" in sub.columns:
       row["IC_boot_mean"] = sub["IC_boot_mean"].mean()
       for p in BOOT_PCTS:
           col = f"IC_boot_p{int(p)}"
           if col in sub.columns:
               row[f"{col}_mean"] = sub[col].mean()

   for c in ic_top_cols:
       row[f"{c}_mean"] = sub[c].mean()
   for c in pc_top_cols:
       row[f"{c}_mean"] = sub[c].mean()
   for lc in lift_cols:
       row[f"{lc}_mean"] = sub[lc].mean()
   for lc in lift_cols_legacy:
       row[f"{lc}_mean"] = sub[lc].mean()
   scorecards.append(row)

scorecard_df = pd.DataFrame(scorecards).sort_values(["Horizon","Score"])
print(scorecard_df.round(3))
```

```
# ---------- Plotting (ONLY the median) ----------
def plot_metric(df, metric, title):
   plt.figure(figsize=(10,5))
   for score in sorted(df["Score"].unique()):
       subset = df[df["Score"]==score].sort_values("Horizon")
       if metric in subset.columns:
           plt.plot(subset["Horizon"], subset[metric], marker='o',
label=score)
   plt.title(title)
   plt.xlabel("Horizon (days)")
   plt.ylabel(metric)
   plt.legend(bbox_to_anchor=(1.05,1), loc='upper left')
   plt.grid(True, alpha=0.3)
   plt.tight_layout()
   plt.show()

# Only plot IC_median (now the median of the bootstrap means)
lbl = f"IC_median (bootstrap-mean median) by Horizon" if ENABLE_BOOTSTRAP
else "IC_median by Horizon"
plot_metric(scorecard_df, "IC_median", lbl)

# ---------- Save Results ----------
scorecard_xlsx =
"scorecard_PCI2_corr_on_high_vol_subsets_with_multi_smoothing_bootstrap.xl
sx"
scorecard_df.to_excel(scorecard_xlsx, index=False)  # lets pandas pick an
available engine (usually openpyxl)

resdf.to_csv("fold_results_PCI2_corr_on_high_vol_subsets_with_multi_smooth
ing_bootstrap.csv", index=False)
print("Done.")
```

## Strategy Final Code

```
from __future__ import annotations
# ============================
# CONFIG
# ============================
START_DATE = "2000-01-01"
```

```
END_DATE   = "2022-01-10"
# Historical S&P 500 membership (ticker, start, end)
SP500_TICKER_RANGES_URL = (

"https://raw.githubusercontent.com/fja05680/sp500/master/sp500_ticker_star
t_end.csv"
)
# ASI2 pairs
ASI2_PAIR_LONG  = (15, 50)   # S15 / L50
ASI2_PAIR_SHORT = (15, 50)
# Cross-sectional selection fractions
LONG_TOP_FRAC  = 0.01        # top X% go long
SHORT_BOT_FRAC = 0.01       # bottom Y% go short
# PCI smoothing
PCI_WMA_WINDOW = 15          # WMA window
# Ω-score parameters
ASI_INDICE       = 2.0
PCI_INDICE       = 2.0
ASI_DENOMINATOR  = 1.0
PCI_DENOMINATOR  = 3.0
# ASI narrowing?
ASI_NARROWING = True
# Capital / allocations
START_CAPITAL = 100_000.0
LONG_ALLOC    = 1
SHORT_ALLOC   = 0
# Rebalancing
REBAL_FREQ_DAYS = 14   # target spacing in trading days
LAG_DAYS        = 1    # signals lagged by 1 trading day (Thu signal → Fri
trade)
# Rebalance weekday:
#   - "Friday", "Mon", "Wednesday", etc., or integer 0=Mon,...,6=Sun
#   - Set to None to use simple "every Nth trading day" instead
REBAL_WEEKDAY   = "Friday"
# Risk-free & annualization
RF_ANNUAL = 0.03
PERIODS_PER_YEAR = 252
# Return clipping to avoid insane outliers from bad data
MAX_ABS_DAILY_RETURN = 0.5   # ±50% per day cap
# Minimum number of S&P members needed to form baseline on a given day
MIN_UNIVERSE_COUNT = 50
# Plot save
SAVE_FIG = True
FIG_PATH = "asi2_pci15_omega_ls_equity_vs_baseline.png"
# ============================
# IMPORTS
# ============================
```

```
import io
import math
import warnings
import requests
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
warnings.filterwarnings("ignore", category=FutureWarning)
# ============================
# S&P 500 MEMBERSHIP
# ============================
def fetch_sp500_ticker_ranges(url: str) -> pd.DataFrame:
    """
    Fetch historical S&P 500 membership as ticker + start/end dates.
    Expected CSV columns (case-insensitive, auto-detected):
        - ticker/symbol
        - start/startdate
        - end/enddate
    Returns DataFrame with:
        ['ticker', 'start', 'end'] (datetimes, end inclusive).
    """
    resp = requests.get(url, timeout=30)
    resp.raise_for_status()
    df = pd.read_csv(io.StringIO(resp.text))
    cols_lower = {c.lower(): c for c in df.columns}
    # ticker
    ticker_col = None
    for key, col in cols_lower.items():
        if key.startswith("ticker") or key.startswith("symbol"):
            ticker_col = col
            break
    if ticker_col is None:
        raise ValueError(f"Ticker column not found in
{df.columns.tolist()}")
    # start
    start_col = None
    for key, col in cols_lower.items():
        if key.startswith("start"):
            start_col = col
            break
    if start_col is None:
        raise ValueError(f"Start date column not found in
{df.columns.tolist()}")
    # end
    end_col = None
    for key, col in cols_lower.items():
```

```
        if key.startswith("end"):
            end_col = col
            break
    if end_col is None:
        raise ValueError(f"End date column not found in
{df.columns.tolist()}")
    out = pd.DataFrame({
        "ticker": df[ticker_col].astype(str).str.upper(),
        "start": pd.to_datetime(df[start_col]),
        "end": pd.to_datetime(df[end_col], errors="coerce"),
    })
    # Missing end => “still in index”
    out["end"] = out["end"].fillna(pd.Timestamp("2100-01-01"))
    # Map some class share tickers to Yahoo style
    mapping = {
        "BRK.B": "BRK-B",
        "BF.B": "BF-B",
        "RDS.A": "RDS-A",
        "RDS.B": "RDS-B",
    }
    out["ticker"] = out["ticker"].replace(mapping)
    return out
def build_membership_mask_from_ranges(
    ranges: pd.DataFrame,
    dates: pd.DatetimeIndex,
) -> pd.DataFrame:
    """
    Build boolean membership mask: index=dates, columns=tickers.
    mask.loc[d, t] = True if start <= d <= end.
    """
    tickers = sorted(ranges["ticker"].unique())
    mask = pd.DataFrame(False, index=dates, columns=tickers)
    for _, row in ranges.iterrows():
        t = row["ticker"]
        s = row["start"]
        e = row["end"]
        if t not in mask.columns:
            continue
        idx = (dates >= s) & (dates <= e)
        if idx.any():
            mask.loc[idx, t] = True
    return mask
# ============================
# MARKET DATA
# ============================
def fetch_ohlcv_adjclose(tickers, start, end):
    """
```

```
    Return (Adj Close, Close, High, Low, Volume)
    All: DataFrames with columns=tickers, index=date.
    """
    raw = yf.download(
        tickers=tickers,
        start=start,
        end=end,
        auto_adjust=False,
        group_by="ticker",
        progress=False,
        threads=True,
        interval="1d",
    )
    if isinstance(raw.columns, pd.MultiIndex):
        adjs, closes, highs, lows, vols = [], [], [], [], []
        top_level = set(raw.columns.get_level_values(0))
        for t in tickers:
            if t not in top_level:
                continue
            df_t = raw[t]
            s_adj = df_t.get("Adj Close", df_t.get("Close"))
            s_clo = df_t.get("Close", df_t.get("Adj Close"))
            s_hi  = df_t.get("High")
            s_lo  = df_t.get("Low")
            s_vol = df_t.get("Volume")
            adjs.append(s_adj.rename(t))
            closes.append(s_clo.rename(t))
            highs.append(s_hi.rename(t))
            lows.append(s_lo.rename(t))
            vols.append(s_vol.rename(t))
        adj_close = pd.concat(adjs, axis=1).sort_index()
        close     = pd.concat(closes, axis=1).sort_index()
        high      = pd.concat(highs, axis=1).sort_index()
        low       = pd.concat(lows, axis=1).sort_index()
        volume    = pd.concat(vols, axis=1).sort_index()
    else:
        col_adj = "Adj Close" if "Adj Close" in raw.columns else "Close"
        col_clo = "Close" if "Close" in raw.columns else col_adj
        t0 = tickers[0]
        adj_close = raw[[col_adj]].rename(columns={col_adj:
t0}).sort_index()
        close     = raw[[col_clo]].rename(columns={col_clo:
t0}).sort_index()
        high      = raw[["High"]].rename(columns={"High": t0}).sort_index()
        low       = raw[["Low"]].rename(columns={"Low": t0}).sort_index()
        volume    = raw[["Volume"]].rename(columns={"Volume":
t0}).sort_index()
```

```
    return adj_close, close, high, low, volume
def get_shares_outstanding(symbol: str) -> float:
    """
    Shares outstanding from Yahoo. Returns NaN if unavailable.
    """
    try:
        ticker = yf.Ticker(symbol)
        fi = getattr(ticker, "fast_info", None)
        shares = None
        if fi is not None and getattr(fi, "shares_outstanding", None):
            shares = float(fi.shares_outstanding)
        else:
            info = ticker.info
            shares = float(info.get("sharesOutstanding", 0) or 0)
        if np.isfinite(shares) and shares > 0:
            return shares
    except Exception:
        pass
    return np.nan
# ============================
# PCI
# ============================
def compute_pci2(high: pd.Series, low: pd.Series, vol: pd.Series, shares:
float) -> pd.Series:
    """
    PCI2 definition:
        μ  = ((shares + vol) * high) / shares
        λ  = ((shares - vol) * low)  / shares
        PCI2 = (high - low) / (μ - λ)
    """
    if not np.isfinite(shares) or shares <= 0:
        return pd.Series(np.nan, index=high.index)
    mu  = ((shares + vol) * high) / shares
    lam = ((shares - vol) * low)  / shares
    denom = mu - lam
    abs_denom = np.abs(denom.to_numpy(dtype=float))
    mask = np.isfinite(abs_denom)
    if mask.any():
        mean_abs = abs_denom[mask].mean()
        if not np.isfinite(mean_abs) or mean_abs == 0:
            eps = 1e-9
        else:
            eps = 1e-9 * mean_abs
    else:
        eps = 1e-9
    good = np.isfinite(denom) & (np.abs(denom) > eps)
    out = np.full(len(high), np.nan, dtype=float)
```

```
    hv, lv, dv = high.values, low.values, denom.values
    out[good] = (hv[good] - lv[good]) / dv[good]
    return pd.Series(out, index=high.index)
def wma_series(x: pd.Series, window: int) -> pd.Series:
    """
    Weighted moving average, used for PCI smoothing.
    """
    if window <= 1:
        return x.copy()
    weights = np.arange(1, window + 1, dtype=float)
    def _wavg(vals):
        v = np.asarray(vals, dtype=float)
        if np.isnan(v).all():
            return np.nan
        mask = np.isfinite(v)
        if mask.sum() == 0:
            return np.nan
        w = weights[-mask.sum():]
        return np.dot(v[mask], w) / w.sum()
    return x.rolling(window, min_periods=max(2, window // 2)).apply(_wavg,
raw=False)
def compute_pci_panel(high: pd.DataFrame,
                      low: pd.DataFrame,
                      volume: pd.DataFrame,
                      shares_map: dict[str, float],
                      wma_window: int = 15) -> pd.DataFrame:
    """
    For each ticker:
      - compute raw PCI2
      - smooth with WMA(window)
    """
    cols = {}
    for t in high.columns:
        sh = shares_map.get(t, np.nan)
        if not np.isfinite(sh) or sh <= 0:
            cols[t] = pd.Series(np.nan, index=high.index)
            continue
        raw_pci = compute_pci2(high[t], low[t], volume[t], sh)
        pci_smoothed = wma_series(raw_pci, wma_window)
        cols[t] = pci_smoothed
    return pd.DataFrame(cols, index=high.index)
# ============================
# ASI / ASI2
# ============================
def asi_from_window_fast(prices_1d: np.ndarray) -> float:
    """
    ASI = #positive / (#positive + #negative)
```

```
    across all subperiod returns inside the window.
    """
    P = prices_1d.astype(float)
    if P.size < 2 or np.any(~np.isfinite(P)):
        return np.nan
    R = (P[:, None] / P[None, :]) - 1.0
    tri_mask = np.triu(np.ones_like(R, dtype=bool), k=1)
    vals = R[tri_mask]
    pos = np.count_nonzero(vals > 0)
    neg = np.count_nonzero(vals < 0)
    denom = pos + neg
    return np.nan if denom == 0 else pos / denom
def compute_asi_on_date(close_wide: pd.DataFrame, date: pd.Timestamp,
lookbacks: list[int]) -> dict[int, pd.Series]:
    """
    Compute ASI for a set of lookbacks on ONE date across all tickers.
    Returns {L: Series(index=tickers)}.
    """
    out = {}
    try:
        end_loc = close_wide.index.get_loc(date)
    except KeyError:
        for L in lookbacks:
            out[L] = pd.Series(index=close_wide.columns, dtype=float)
        return out
    for L in sorted(set(lookbacks)):
        if end_loc < L - 1:
            out[L] = pd.Series(index=close_wide.columns, dtype=float)
            continue
        window = close_wide.iloc[end_loc - (L - 1): end_loc + 1]
        vals = []
        for t in close_wide.columns:
            p = window[t].to_numpy()
            if np.isnan(p).any():
                vals.append(np.nan)
            else:
                vals.append(asi_from_window_fast(p))
        out[L] = pd.Series(vals, index=close_wide.columns, dtype=float)
    return out
def build_asi2_on_rebalance_dates(close_wide: pd.DataFrame,
                                  rebal_dates: pd.DatetimeIndex,
                                  lag_days: int,
                                  pair_long: tuple[int, int],
                                  pair_short: tuple[int, int]) ->
tuple[pd.DataFrame, pd.DataFrame]:
    """
    Build ASI2 panels ONLY on the (lagged) rebalance dates.
```

```
    """
    needed = sorted({pair_long[0], pair_long[1], pair_short[0],
pair_short[1]})
    asi2_long_rows, asi2_short_rows = [], []
    valid_index = []
    for d in rebal_dates:
        try:
            loc = close_wide.index.get_loc(d)
        except KeyError:
            continue
        lag_loc = loc - lag_days
        if lag_loc < 0:
            continue
        d_sig = close_wide.index[lag_loc]
        by_L = compute_asi_on_date(close_wide, d_sig, needed)
        asi2_long_row = by_L.get(pair_long[0],
pd.Series(index=close_wide.columns)) / \
                        by_L.get(pair_long[1],
pd.Series(index=close_wide.columns))
        asi2_short_row = by_L.get(pair_short[0],
pd.Series(index=close_wide.columns)) / \
                         by_L.get(pair_short[1],
pd.Series(index=close_wide.columns))
        asi2_long_rows.append(asi2_long_row.rename(d))
        asi2_short_rows.append(asi2_short_row.rename(d))
        valid_index.append(d)
    if not valid_index:
        empty_index = pd.DatetimeIndex([], name="date")
        return (pd.DataFrame(index=empty_index, columns=close_wide.columns,
dtype=float),
                pd.DataFrame(index=empty_index, columns=close_wide.columns,
dtype=float))
    asi2_long_df  = pd.DataFrame(asi2_long_rows)
    asi2_short_df = pd.DataFrame(asi2_short_rows)
    return asi2_long_df, asi2_short_df
# ============================
# GENERAL UTILITIES
# ============================
def sharpe_annualized(daily_returns: pd.Series, rf_annual: float = 0.0,
periods_per_year: int = 252) -> float:
    r = daily_returns.dropna()
    if r.empty:
        return float("nan")
    rf_daily = (1.0 + rf_annual) ** (1.0 / periods_per_year) - 1.0
    excess = r - rf_daily
    sd = excess.std(ddof=1)
    if sd == 0 or math.isnan(sd):
```

```
        return float("nan")
    return (excess.mean() / sd) * math.sqrt(periods_per_year)
def compute_return_stats(
    daily_returns: pd.Series,
    rf_annual: float = 0.0,
    periods_per_year: int = 252,
) -> dict[str, float]:
    """
    Compute daily standard deviation, daily geometric mean and annualized
Sharpe
    for a series of daily returns.
    """
    r = daily_returns.dropna()
    if r.empty:
        return {
            "std_daily": float("nan"),
            "geo_daily": float("nan"),
            "sharpe_annual": float("nan"),
        }
    # Daily standard deviation (arithmetic returns)
    std_daily = r.std(ddof=1)
    # Daily geometric mean: (Π (1+r))^(1/N) - 1
    geo_daily = (1.0 + r).prod() ** (1.0 / len(r)) - 1.0
    # Annualized Sharpe using existing helper
    sharpe = sharpe_annualized(r, rf_annual=rf_annual,
periods_per_year=periods_per_year)
    return {
        "std_daily": float(std_daily),
        "geo_daily": float(geo_daily),
        "sharpe_annual": float(sharpe),
    }
def max_drawdown(equity_curve: pd.Series) -> float:
    ec = equity_curve.dropna()
    if ec.empty:
        return float("nan")
    peak = ec.cummax()
    dd = ec / peak - 1.0
    return float(dd.min())
def alpha_beta_excess(port_ret: pd.Series, mkt_ret: pd.Series,
                      rf_annual: float, periods_per_year: int = 252):
    rf_daily = (1.0 + rf_annual) ** (1.0 / periods_per_year) - 1.0
    df = pd.concat([port_ret, mkt_ret], axis=1, join="inner").dropna()
    if df.empty:
        return np.nan, np.nan, np.nan
    p = df.iloc[:, 0] - rf_daily
    m = df.iloc[:, 1] - rf_daily
    var_m = m.var(ddof=1)
```

```
    if var_m == 0 or np.isnan(var_m):
        return np.nan, np.nan, np.nan
    cov_mp = np.cov(m, p, ddof=1)[0, 1]
    beta = cov_mp / var_m
    alpha_daily = p.mean() - beta * m.mean()
    alpha_annual = (1.0 + alpha_daily) ** periods_per_year - 1.0
    return float(alpha_annual), float(beta), float(alpha_daily)
def _normalize_or_empty(w: pd.Series) -> pd.Series:
    s = w.sum()
    if s is None or not np.isfinite(s) or s <= 0:
        return pd.Series(dtype=float)
    return w / s
def _equal_weight_index(idx: pd.Index) -> pd.Series:
    if len(idx) == 0:
        return pd.Series(dtype=float)
    return pd.Series(1.0 / len(idx), index=idx, dtype=float)
def _safe_power_and_normalize(series: pd.Series, power: float) ->
pd.Series | None:
    if series is None or series.empty:
        return None
    s = series.astype(float).replace([np.inf, -np.inf], np.nan)
    s = s.clip(lower=0.0)
    if s.isna().all():
        return None
    if power == 0:
        s = pd.Series(1.0, index=s.index)
    else:
        s = s ** power
    s = s.replace([np.inf, -np.inf], np.nan)
    if s.isna().all():
        return None
    total = s.sum()
    if not np.isfinite(total) or total <= 0:
        return None
    return s / total
def compute_omega_series(asi_series: pd.Series,
                         pci_series: pd.Series,
                         asi_power: float,
                         pci_power: float,
                         asi_denominator: float,
                         pci_denominator: float) -> pd.Series | None:
    idx = asi_series.index.union(pci_series.index)
    asi_norm = _safe_power_and_normalize(asi_series, asi_power)
    pci_norm = _safe_power_and_normalize(pci_series, pci_power)
    if asi_norm is None and pci_norm is None:
        return None
    if asi_norm is None:
```

```
        asi_omega = pd.Series(0.0, index=idx)
    else:
        asi_omega = asi_norm.reindex(idx).fillna(0.0) /
float(asi_denominator)
    if pci_norm is None:
        pci_omega = pd.Series(0.0, index=idx)
    else:
        pci_omega = pci_norm.reindex(idx).fillna(0.0) /
float(pci_denominator)
    omega = asi_omega.add(pci_omega, fill_value=0.0)
    omega = omega.replace([np.inf, -np.inf], np.nan)
    if omega.isna().all():
        return None
    return omega
def sanitize_returns_df(df: pd.DataFrame,
                        max_abs_return: float = MAX_ABS_DAILY_RETURN) ->
pd.DataFrame:
    """
    Clip extreme daily returns to avoid insane compounding from bad data.
    """
    return df.clip(lower=-max_abs_return, upper=max_abs_return)
def sanitize_returns_series(s: pd.Series,
                            max_abs_return: float = MAX_ABS_DAILY_RETURN)
-> pd.Series:
    return s.clip(lower=-max_abs_return, upper=max_abs_return)
# ============================
# WEEKDAY LOGIC
# ============================
def _weekday_to_int(wd) -> int | None:
    """
    Convert weekday spec to integer:
      - 0=Mon, 1=Tue, ..., 6=Sun
      - Accepts 'Monday', 'Mon', 'fri', etc.
      - If wd is None, returns None (no weekday restriction).
    """
    if wd is None:
        return None
    if isinstance(wd, int):
        if 0 <= wd <= 6:
            return wd
        raise ValueError("rebal_weekday integer must be between 0 (Monday)
and 6 (Sunday).")
    wd_str = str(wd).strip().lower()
    mapping = {
        "monday": 0, "mon": 0,
        "tuesday": 1, "tue": 1, "tues": 1,
        "wednesday": 2, "wed": 2,
```

```
        "thursday": 3, "thu": 3, "thur": 3, "thurs": 3,
        "friday": 4, "fri": 4,
        "saturday": 5, "sat": 5,
        "sunday": 6, "sun": 6,
    }
    if wd_str not in mapping:
        raise ValueError(f"Unrecognized weekday: {wd!r}")
    return mapping[wd_str]
def compute_weekday_rebalance_dates(
    all_dates: pd.DatetimeIndex,
    rebalance_freq_days: int,
    rebal_weekday_int: int,
) -> pd.DatetimeIndex:
    """
    Build a sequence of rebalance dates restricted to a given weekday.
    Logic:
      1) First rebalance = first date whose weekday == rebal_weekday_int.
      2) Each subsequent rebalance i:
           - target index = last_rebal_index + rebalance_freq_days
           - among FUTURE dates with that weekday, pick the one whose
             index is closest to 'target index' (in trading days),
             while strictly moving forward in time.
    """
    all_dates = pd.DatetimeIndex(all_dates)
    if len(all_dates) == 0:
        return all_dates[:0]
    weekday_vals = all_dates.weekday
    weekday_indices = np.where(weekday_vals == rebal_weekday_int)[0]
    if len(weekday_indices) == 0:
        raise ValueError("No trading days match the requested rebalance
weekday in the date range.")
    rebal_indices: list[int] = []
    # First rebalance: first such weekday
    first_idx = int(weekday_indices[0])
    rebal_indices.append(first_idx)
    # Subsequent rebalances
    while True:
        last_idx = rebal_indices[-1]
        target_idx = last_idx + rebalance_freq_days
        if target_idx >= len(all_dates):
            break
        future_wd = weekday_indices[weekday_indices > last_idx]
        if len(future_wd) == 0:
            break
        closest_idx = int(future_wd[np.argmin(np.abs(future_wd -
target_idx))])
        if closest_idx >= len(all_dates):
```

```
            break
        rebal_indices.append(closest_idx)
    rebal_indices = np.array(sorted(set(rebal_indices)), dtype=int)
    return all_dates[rebal_indices]
# ============================
# CROSS-SECTION WEIGHTS
# ============================
def cross_section_long_weights(asi2_row: pd.Series,
                               pci_row: pd.Series,
                               top_frac: float,
                               asi_power: float,
                               pci_power: float,
                               asi_denominator: float,
                               pci_denominator: float,
                               asi_narrowing: bool) -> pd.Series:
    asi_clean = asi2_row.dropna()
    pci_clean = pci_row.dropna()
    if asi_narrowing:
        if asi_clean.empty:
            return pd.Series(dtype=float)
        cutoff = asi_clean.quantile(1 - top_frac)
        sel = asi_clean[asi_clean >= cutoff]
        sel_idx = sel.index
        if len(sel_idx) == 0:
            return pd.Series(dtype=float)
        asi_sel = asi2_row.reindex(sel_idx)
        pci_sel = pci_row.reindex(sel_idx)
        omega = compute_omega_series(
            asi_sel, pci_sel,
            asi_power=asi_power,
            pci_power=pci_power,
            asi_denominator=asi_denominator,
            pci_denominator=pci_denominator,
        )
        if omega is None:
            return _equal_weight_index(sel_idx)
        omega = omega.reindex(sel_idx).fillna(0.0)
        if omega.sum() <= 0 or omega.isna().all():
            return _equal_weight_index(sel_idx)
        return _normalize_or_empty(omega)
    # no ASI narrowing
    if asi_clean.empty and pci_clean.empty:
        return pd.Series(dtype=float)
    all_idx = asi2_row.index.union(pci_row.index)
    asi_all = asi2_row.reindex(all_idx)
    pci_all = pci_row.reindex(all_idx)
    omega_all = compute_omega_series(
```

```
        asi_all, pci_all,
        asi_power=asi_power,
        pci_power=pci_power,
        asi_denominator=asi_denominator,
        pci_denominator=pci_denominator,
    )
    if omega_all is None:
        return _equal_weight_index(all_idx)
    omega_all = omega_all.dropna()
    if omega_all.empty:
        return _equal_weight_index(all_idx)
    cutoff = omega_all.quantile(1 - top_frac)
    sel = omega_all[omega_all >= cutoff]
    if sel.empty:
        return _equal_weight_index(omega_all.index)
    return _normalize_or_empty(sel)
def cross_section_short_weights(asi2_row: pd.Series,
                                pci_row: pd.Series,
                                bot_frac: float,
                                asi_power: float,
                                pci_power: float,
                                asi_denominator: float,
                                pci_denominator: float,
                                asi_narrowing: bool) -> pd.Series:
    asi_clean = asi2_row.dropna()
    pci_clean = pci_row.dropna()
    if asi_narrowing:
        if asi_clean.empty:
            return pd.Series(dtype=float)
        cutoff = asi_clean.quantile(bot_frac)
        sel = asi_clean[asi_clean <= cutoff]
        sel_idx = sel.index
        if len(sel_idx) == 0:
            return pd.Series(dtype=float)
        asi_sel = asi2_row.reindex(sel_idx)
        pci_sel = pci_row.reindex(sel_idx)
        omega = compute_omega_series(
            asi_sel, pci_sel,
            asi_power=asi_power,
            pci_power=pci_power,
            asi_denominator=asi_denominator,
            pci_denominator=pci_denominator,
        )
        if omega is None:
            return _equal_weight_index(sel_idx)
        omega = omega.reindex(sel_idx).fillna(0.0)
        if omega.sum() <= 0 or omega.isna().all():
```

```
            return _equal_weight_index(sel_idx)
        return _normalize_or_empty(omega)
    if asi_clean.empty and pci_clean.empty:
        return pd.Series(dtype=float)
    all_idx = asi2_row.index.union(pci_row.index)
    asi_all = asi2_row.reindex(all_idx)
    pci_all = pci_row.reindex(all_idx)
    omega_all = compute_omega_series(
        asi_all, pci_all,
        asi_power=asi_power,
        pci_power=pci_power,
        asi_denominator=asi_denominator,
        pci_denominator=pci_denominator,
    )
    if omega_all is None:
        return _equal_weight_index(all_idx)
    omega_all = omega_all.dropna()
    if omega_all.empty:
        return _equal_weight_index(all_idx)
    cutoff = omega_all.quantile(bot_frac)
    sel = omega_all[omega_all <= cutoff]
    if sel.empty:
        return _equal_weight_index(omega_all.index)
    return _normalize_or_empty(sel)
# ============================
# BACKTEST
# ============================
def backtest_long_short_rebalance_only(
    adj_close: pd.DataFrame,
    close_for_asi: pd.DataFrame,
    pci_panel: pd.DataFrame,
    pair_long: tuple[int, int],
    pair_short: tuple[int, int],
    top_frac_long: float,
    bot_frac_short: float,
    long_alloc: float,
    short_alloc: float,
    rebalance_freq_days: int,
    lag_days: int,
    start_capital: float = 100_000.0,
    rf_annual: float = 0.0,
    periods_per_year: int = 252,
    asi_indice: float = 2.0,
    pci_indice: float = 2.0,
    asi_denominator: float = 1.0,
    pci_denominator: float = 1.0,
    asi_narrowing: bool = True,
```

```
    membership_mask: pd.DataFrame | None = None,
    rebal_weekday=None,
):
    """
    ASI2 + PCI Ω-score L/S backtest with point-in-time S&P 500 membership.
    - ASI signals are lagged by `lag_days` (e.g. Thu signal → Fri
rebalance).
    - PCI panel is lagged by the same `lag_days` so PCI is also from the
prior day.
    """
    # Daily returns
    ret = adj_close.pct_change(fill_method=None)
    ret = sanitize_returns_df(ret)
    all_dates = ret.index
    if len(all_dates) == 0:
        raise ValueError("No return data available.")
    # Membership & universe baseline
    if membership_mask is not None:
        membership_mask = membership_mask.reindex(index=all_dates,
columns=ret.columns, fill_value=False)
        counts = membership_mask.sum(axis=1)
        mask_good = counts >= MIN_UNIVERSE_COUNT
        membership_mask = membership_mask.where(mask_good, other=False)
        ret = ret.where(mask_good, np.nan)
        mkt_ret_all = ret.where(membership_mask).mean(axis=1, skipna=True)
    else:
        mkt_ret_all = ret.mean(axis=1, skipna=True)
    mkt_ret_all = sanitize_returns_series(mkt_ret_all)
    # Rebalance dates: either every Nth trading day or weekday-restricted
    weekday_int = _weekday_to_int(rebal_weekday)
    if weekday_int is None:
        rebal_idx = np.arange(0, len(all_dates), rebalance_freq_days,
dtype=int)
        rebal_dates_all = all_dates[rebal_idx]
    else:
        rebal_dates_all = compute_weekday_rebalance_dates(
            all_dates=all_dates,
            rebalance_freq_days=rebalance_freq_days,
            rebal_weekday_int=weekday_int,

    # ASI2 on (lagged) rebalance dates
    asi2_long_df, asi2_short_df = build_asi2_on_rebalance_dates(
        close_wide=close_for_asi,
        rebal_dates=rebal_dates_all,
        lag_days=lag_days,
        pair_long=pair_long,
        pair_short=pair_short,
```

```
    )
    valid_rebals = asi2_long_df.index.intersection(asi2_short_df.index)
    if len(valid_rebals) == 0:
        raise ValueError("No valid rebalance dates after lag — not enough
history.")
    first_date = valid_rebals.min()
    ret       = ret.loc[first_date:]
    mkt_ret   = mkt_ret_all.loc[first_date:]
    # Lag PCI by the same lag_days (e.g. Thu PCI for Fri trade)
    pci_signal_panel = pci_panel.shift(lag_days)
    pci_signal_panel = pci_signal_panel.loc[first_date:]
    if membership_mask is not None:
        membership_mask = membership_mask.loc[first_date:]
    dates = ret.index
    rebal_dates = pd.DatetimeIndex([d for d in valid_rebals if d in dates])
    port_ret = pd.Series(0.0, index=dates)
    long_weight_book, short_weight_book = {}, {}
    for i, d in enumerate(rebal_dates):
        # Holding window [d, d_next)
        if i < len(rebal_dates) - 1:
            d_next = rebal_dates[i + 1]
            window = dates[(dates > d) & (dates <= d_next)] if i <
len(rebal_dates)-1 else dates[dates > d]
        else:
            window = dates[dates >= d]
        if window.empty:
            continue
        # Active S&P 500 members at rebalance date
        if membership_mask is not None and d in membership_mask.index:
            active_cols = membership_mask.columns[membership_mask.loc[d]]
            if len(active_cols) == 0:
                port_ret.loc[window] = 0.0
                continue
        else:
            active_cols = ret.columns
        row_long_full  = asi2_long_df.loc[d]  if d in asi2_long_df.index
else pd.Series(index=ret.columns, dtype=float)
        row_short_full = asi2_short_df.loc[d] if d in asi2_short_df.index
else pd.Series(index=ret.columns, dtype=float)
        pci_row_full   = pci_signal_panel.loc[d] if d in
pci_signal_panel.index else pd.Series(index=ret.columns, dtype=float)
        row_long  = row_long_full.reindex(active_cols)
        row_short = row_short_full.reindex(active_cols)
        pci_row   = pci_row_full.reindex(active_cols)
        # Weights
        wL = cross_section_long_weights(
            row_long,
```

```
            pci_row,
            top_frac=top_frac_long,
            asi_power=asi_indice,
            pci_power=pci_indice,
            asi_denominator=asi_denominator,
            pci_denominator=pci_denominator,
            asi_narrowing=asi_narrowing,
        )
        wS = cross_section_short_weights(
            row_short,
            pci_row,
            bot_frac=bot_frac_short,
            asi_power=asi_indice,
            pci_power=pci_indice,
            asi_denominator=asi_denominator,
            pci_denominator=pci_denominator,
            asi_narrowing=asi_narrowing,
        )
        long_weight_book[d]  = wL
        short_weight_book[d] = wS
        # Basket returns over holding window
        if not wL.empty:
            rL = ret[wL.index].loc[window].fillna(0.0).mul(wL,
axis=1).sum(axis=1)
        else:
            rL = pd.Series(0.0, index=window)
        if not wS.empty:
            rS = ret[wS.index].loc[window].fillna(0.0).mul(wS,
axis=1).sum(axis=1)
        else:
            rS = pd.Series(0.0, index=window)
        port_ret.loc[window] = long_alloc * rL - short_alloc * rS
    equity      = (1.0 + port_ret.fillna(0.0)).cumprod() * start_capital
    mkt_equity  = (1.0 + mkt_ret.fillna(0.0)).cumprod() * start_capital
    total_return = equity.iloc[-1] / equity.iloc[0] - 1.0
    ann_ret = (equity.iloc[-1] / equity.iloc[0]) ** (periods_per_year /
max(1, len(equity))) - 1.0
    sharpe_full = sharpe_annualized(port_ret, rf_annual=rf_annual,
periods_per_year=periods_per_year)
    mdd = max_drawdown(equity)
    alpha_ann, beta, _ = alpha_beta_excess(port_ret, mkt_ret, rf_annual,
periods_per_year)
    # ============
    # New stats (strategy & market, full period and last year)
    # ============
    # Full-period stats
    strat_stats_full = compute_return_stats(
```

```
        port_ret, rf_annual=rf_annual, periods_per_year=periods_per_year
    )
    mkt_stats_full = compute_return_stats(
        mkt_ret, rf_annual=rf_annual, periods_per_year=periods_per_year
    )
    # Last "year" of backtest = last `periods_per_year` trading days
    if len(port_ret.dropna()) >= periods_per_year:
        port_last = port_ret.dropna().iloc[-periods_per_year:]
    else:
        port_last = port_ret.dropna()
    if len(mkt_ret.dropna()) >= periods_per_year:
        mkt_last = mkt_ret.dropna().iloc[-periods_per_year:]
    else:
        mkt_last = mkt_ret.dropna()
    strat_stats_last = compute_return_stats(
        port_last, rf_annual=rf_annual, periods_per_year=periods_per_year
    )
    mkt_stats_last = compute_return_stats(
        mkt_last, rf_annual=rf_annual, periods_per_year=periods_per_year
    )
    summary = {
        "Total Return %": total_return * 100.0,
        "Ann. Return %": ann_ret * 100.0,
        "Ann. Sharpe": sharpe_full,
        "Max Drawdown %": mdd * 100.0,
        "Alpha (annual %)": (alpha_ann * 100.0) if not np.isnan(alpha_ann)
else np.nan,
        "Beta vs Universe": beta,
        "Start": str(dates.min().date()),
        "End": str(dates.max().date()),
        "Rebalance (days)": rebalance_freq_days,
        "Rebalance Weekday": str(rebal_weekday),
        "Long Alloc": long_alloc,
        "Short Alloc": short_alloc,
        "Long Pair": f"S{pair_long[0]}/L{pair_long[1]}",
        "Short Pair": f"S{pair_short[0]}/L{pair_short[1]}",
        "Top% Long": top_frac_long,
        "Bot% Short": bot_frac_short,
        "ASI Ω Power": asi_indice,
        "PCI Ω Power": pci_indice,
        "ASI Ω Denominator": asi_denominator,
        "PCI Ω Denominator": pci_denominator,
        "ASI Narrowing": asi_narrowing,
        "PCI WMA Window": PCI_WMA_WINDOW,
        "Max abs daily return clip": MAX_ABS_DAILY_RETURN,
        # Strategy stats (full period)
        "Strategy Daily Std (full)": strat_stats_full["std_daily"],
```

```
        "Strategy Daily GeoMean % (full)": strat_stats_full["geo_daily"] *
100.0,
        "Strategy Ann. Sharpe (full)": strat_stats_full["sharpe_annual"],
        # Strategy stats (last year)
        "Strategy Daily Std (last year)": strat_stats_last["std_daily"],
        "Strategy Daily GeoMean % (last year)":
strat_stats_last["geo_daily"] * 100.0,
        "Strategy Ann. Sharpe (last year)":
strat_stats_last["sharpe_annual"],
        # Market / universe stats (full period)
        "Market Daily Std (full)": mkt_stats_full["std_daily"],
        "Market Daily GeoMean % (full)": mkt_stats_full["geo_daily"] *
100.0,
        "Market Ann. Sharpe (full)": mkt_stats_full["sharpe_annual"],
        # Market / universe stats (last year)
        "Market Daily Std (last year)": mkt_stats_last["std_daily"],
        "Market Daily GeoMean % (last year)": mkt_stats_last["geo_daily"] *
100.0,
        "Market Ann. Sharpe (last year)": mkt_stats_last["sharpe_annual"],
    }
    return {
        "daily_returns": port_ret,
        "equity": equity,
        "market_returns": mkt_ret,
        "market_equity": mkt_equity,
        "summary": summary,
        "long_weights": long_weight_book,
        "short_weights": short_weight_book,
        "asi2_long_rows": asi2_long_df,
        "asi2_short_rows": asi2_short_df,
        "raw_returns": ret,
    }
# ============================
# MAIN
# ============================
if __name__ == "__main__":
    # 0) Historical membership
    print("Downloading historical S&P 500 membership ranges...")
    sp500_ranges = fetch_sp500_ticker_ranges(SP500_TICKER_RANGES_URL)
    universe_tickers = sorted(sp500_ranges["ticker"].unique())
    print(f"Universe tickers (from S&P 500 history):
{len(universe_tickers)}")
    # 1) OHLCV data
    print("Downloading OHLCV from Yahoo Finance...")
    adj_close, close, high, low, volume =
fetch_ohlcv_adjclose(universe_tickers, START_DATE, END_DATE)
    # membership mask aligned to trading days
```

```
    print("Building point-in-time membership mask...")
    membership_mask = build_membership_mask_from_ranges(sp500_ranges,
adj_close.index)
    # 2) Shares outstanding
    print("Fetching shares outstanding (this may take a bit)...")
    shares_map = {t: get_shares_outstanding(t) for t in adj_close.columns}
    print("Sample shares_outstanding:", list(shares_map.items())[:5])
    # 3) PCI
    print("Computing PCI panel (smoothed)...")
    pci_panel = compute_pci_panel(
        high=high,
        low=low,
        volume=volume,
        shares_map=shares_map,
        wma_window=PCI_WMA_WINDOW,
    )
    # 4) Backtest
    print("Running backtest...")
    results = backtest_long_short_rebalance_only(
        adj_close=adj_close,
        close_for_asi=close,
        pci_panel=pci_panel,
        pair_long=ASI2_PAIR_LONG,
        pair_short=ASI2_PAIR_SHORT,
        top_frac_long=LONG_TOP_FRAC,
        bot_frac_short=SHORT_BOT_FRAC,
        long_alloc=LONG_ALLOC,
        short_alloc=SHORT_ALLOC,
        rebalance_freq_days=REBAL_FREQ_DAYS,
        lag_days=LAG_DAYS,
        start_capital=START_CAPITAL,
        rf_annual=RF_ANNUAL,
        periods_per_year=PERIODS_PER_YEAR,
        asi_indice=ASI_INDICE,
        pci_indice=PCI_INDICE,
        asi_denominator=ASI_DENOMINATOR,
        pci_denominator=PCI_DENOMINATOR,
        asi_narrowing=ASI_NARROWING,
        membership_mask=membership_mask,
        rebal_weekday=REBAL_WEEKDAY,
    )
    # 5) Summary
    print("\n=== Backtest Summary (ASI2 + PCI via Ω-score; PIT S&P 500)
===")
    for k, v in results["summary"].items():
        if isinstance(v, float):
```

```
            if "Return" in k or "Drawdown" in k or "Alpha" in k or
"GeoMean" in k:
                print(f"{k:>36}: {v:10.2f}")
            else:
                print(f"{k:>36}: {v:10.4f}")
        else:
            print(f"{k:>36}: {v}")
    # 6) Plot
    plt.figure(figsize=(11, 5))
    results["equity"].plot(label="ASI2 + PCI Ω L/S")
    results["market_equity"].plot(label="S&P 500 Universe (EW)")
    plt.title("Equity Curve — ASI2 + PCI Ω-Score vs S&P 500 (PIT
Membership)")
    plt.xlabel("Date")
    plt.ylabel("Equity ($)")
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.tight_layout()
    if SAVE_FIG:
        plt.savefig(FIG_PATH, dpi=160)
    plt.show(
    # 7) Save CSVs
    results["equity"].to_csv("asi2_pci15_omega_ls_equity.csv",
header=["Equity"])

results["daily_returns"].to_csv("asi2_pci15_omega_ls_daily_returns.csv",
header=["Daily Return"])
```