Adaptive Filter Strategy Pack Manual

This guide provides an overview of each trading strategy included in the package, with descriptions of their objectives, key components, parameters, logic, and notes on their unique features.

A Note on Backtesting: All strategies presented are for educational and illustrative purposes. The Python scripts provide a framework for backtesting these concepts. Past performance is not indicative of future results. Thorough testing, optimization, risk management, and consideration of transaction costs (slippage, commissions) are crucial before deploying any trading strategy with real capital. The code snippets included in the manual are illustrative and extracted from the provided standalone Python scripts.

Table of Contents

---

1. Adaptive Rectified Linear Filter Strategy

1.1. Overview and Objective

Overview:
This strategy implements an adaptive filter, specifically an Exponential Moving Average (EMA), where the smoothing period is dynamically adjusted based on market trend strength. The Average Directional Index (ADX) is used to quantify trend strength. A stronger trend (higher ADX) results in a shorter EMA period (making the filter more responsive), while a weaker trend (lower ADX) uses a longer EMA period (making the filter smoother). This "Adaptive Rectified Linear Filter" (ARL) then serves as a dynamic baseline for generating trading signals.

Objective:
The primary objective is to trade crossovers of the closing price with this adaptively smoothed price filter.

- A **long** position is initiated when the previous day's close crosses above the previous day's filtered price.
- A **short** position is initiated when the previous day's close crosses below the previous day's filtered price.
  The strategy incorporates an ATR-based trailing stop-loss for risk management and accounts for trading commissions.

1.2. Key Indicators and Components

The strategy relies on several indicators and calculated series:

- **ADX (Average Directional Index - `adx_col_name`):** Measures trend strength. Calculated using the `ta` library.

```
# ADX Calculation
high = df['High']
low = df['Low']
close = df['Close']
df[adx_col_name] = ta.trend.ADXIndicator(high, low, close,
window=adx_period).adx()
df[adx_col_name].fillna(method='bfill', inplace=True) # Fill initial ADX
NaNs
```

- **Normalized ADX (`normalized_adx_col`):** ADX values are clipped within a defined range (`adx_map_min` , `adx_map_max`) and then normalized to a 0-1 scale. This normalized value determines the adaptiveness of the filter.

```
adx_clipped = df[adx_col_name].clip(adx_map_min, adx_map_max)
df[normalized_adx_col] = (adx_clipped - adx_map_min) / (adx_map_max -
adx_map_min)
df[normalized_adx_col].fillna(0, inplace=True)
df[normalized_adx_col] = np.clip(df[normalized_adx_col], 0, 1)
```

- **Adaptive EMA Period (`adaptive_ema_period_col`):** The period for the adaptive EMA, interpolated between `period_min_filt` (for high ADX) and `period_max_filt` (for low ADX) based on the *previous day's* normalized ADX.

```
# High Normalized ADX (strong trend) -> period_min_filt (faster EMA)
# Low Normalized ADX (weak trend) -> period_max_filt (slower EMA)
df[adaptive_ema_period_col] = period_max_filt -
df[normalized_adx_col].shift(1) * (period_max_filt - period_min_filt)
df[adaptive_ema_period_col] =
np.round(df[adaptive_ema_period_col]).fillna( (period_min_filt +
period_max_filt) / 2 ).astype(int)
df[adaptive_ema_period_col] = np.clip(df[adaptive_ema_period_col],
period_min_filt, period_max_filt)
```

- **Adaptive Rectified Linear Filter / Adaptive EMA (`filtered_price_arl_col`):** This is the core adaptive filter, calculated iteratively. The smoothing factor `Alpha_Adaptive` changes daily based on `adaptive_ema_period_col` .

```
df['Alpha_Adaptive'] = 2 / (df[adaptive_ema_period_col] + 1)
df[filtered_price_arl_col] = np.nan
# ... (Seeding the first value) ...
first_valid_alpha_idx = df['Alpha_Adaptive'].first_valid_index()
df.loc[first_valid_alpha_idx, filtered_price_arl_col] =
df.loc[first_valid_alpha_idx, 'Close'] # Seed
start_loc_for_ema_loop = df.index.get_loc(first_valid_alpha_idx)

for i_loop in range(start_loc_for_ema_loop + 1, len(df)):
    idx_today = df.index[i_loop]; idx_prev = df.index[i_loop-1]
    alpha_val = df.loc[idx_today, 'Alpha_Adaptive']
    current_close_val = df.loc[idx_today, 'Close']
    prev_filtered_price_val = df.loc[idx_prev, filtered_price_arl_col]
    if any(pd.isna(val) for val in [alpha_val, current_close_val,
prev_filtered_price_val]):
        df.loc[idx_today, filtered_price_arl_col] =
prev_filtered_price_val
    else:
        df.loc[idx_today, filtered_price_arl_col] = alpha_val *
current_close_val + (1 - alpha_val) * prev_filtered_price_val
df[filtered_price_arl_col].fillna(method='ffill', inplace=True);
df[filtered_price_arl_col].fillna(method='bfill', inplace=True)
```

- **Average True Range (ATR - `atr_col_name_sl`):** Used for calculating the trailing stop-loss.

```
df['H-L_sl'] = df['High'] - df['Low']; df['H-PC_sl'] = (df['High'] -
df['Close'].shift(1)).abs(); df['L-PC_sl'] = (df['Low']  -
df['Close'].shift(1)).abs()
df['TR_sl'] = df[['H-L_sl','H-PC_sl','L-PC_sl']].max(axis=1);
df[atr_col_name_sl] = df['TR_sl'].rolling(window=atr_window_sl).mean()
```

1.3. Script Parameters

The main configurable parameters are defined at the beginning of the script:

```
# --- Parameters ---
ticker = "ETH-USD"
start_date_str = "2020-01-01"
end_date_str = "2024-12-31"

adx_period = 14
# ADX values to map to period range
adx_map_min = 20  # ADX at or below this gets period_max_filt
adx_map_max = 40  # ADX at or above this gets period_min_filt

# EMA period range for the adaptive filter
period_min_filt = 7 # Faster EMA for strong ADX
period_max_filt = 30 # Slower EMA for weak ADX

atr_window_sl = 14
```

```
atr_multiplier_sl = 1.0
commission_bps_per_side = 1.0 # Basis points per side for commission
```

(Note: `TRADING_DAYS_PER_YEAR`, `verbose`, and `plot_results` are also present).

1.4. Data Handling

- **Data Download:** Daily OHLCV data is downloaded using `yfinance`. User preferences for `auto_adjust=False` and `droplevel` are applied.

  ```
  # --- 1. Download Data ---
  df_raw = yf.download([ticker], start=start_date_str, end=end_date_str,
  auto_adjust=False, progress=False)
  # ... (droplevel and column selection logic) ...
  df = df[['Open', 'High', 'Low', 'Close', 'Volume']].copy()
  ```

1.5. Trading Logic (within the main backtesting loop)

The strategy iterates daily. Signals are based on the *previous day's* close versus the *previous day's* filtered price. Trades are executed at the *current day's* `Open`.

1.5.1. Signal Generation Conditions

A target position (`target_pos`) is determined based on the previous day's relationship between the close and the adaptive filter:

```
# (Inside backtesting loop, after stop-loss check)
    target_pos = 0
    if prev_close > prev_filtered_price: target_pos = 1  # Signal Long
    elif prev_close < prev_filtered_price: target_pos = -1 # Signal Short
```

1.5.2. Entry Conditions

If the `target_pos` is different from the `current_pos` (position held at the start of the day), a new trade or a flip is considered. Entry is at `today_open`.

```
# (If target_pos != current_pos)
            if current_pos != 0 and pd.notna(current_entry_gross): # It's a
flip
                # ... (calculate pnl_exit_leg_net for the closing part of
the flip) ...

            current_pos = target_pos; current_entry_gross = today_open # Set
new position and entry price
```

```
            if current_pos == 1: # Entering New Long
                # ... (calculate pnl_entry_leg_day, set init_ts and
current_ts) ...
            elif current_pos == -1: # Entering New Short
                # ... (calculate pnl_entry_leg_day, set init_ts and
current_ts) ...
```

1.5.3. Exit Conditions

The primary exit mechanism is the ATR Trailing Stop-Loss. Positions are also exited if an opposing signal is generated (flip).

- **ATR Trailing Stop-Loss Check (highest priority):**

```
# (At the start of the daily loop logic for an active position)
if current_pos != 0 and pd.notna(current_ts) and
pd.notna(current_entry_gross):
    exit_price_sl_gross = 0; stopped_out = False
    if current_pos == 1 and today_low <= current_ts:
        exit_price_sl_gross = min(today_open, current_ts); stopped_out =
True
    elif current_pos == -1 and today_high >= current_ts:
        exit_price_sl_gross = max(today_open, current_ts); stopped_out =
True
    if stopped_out:
        # ... (calculate pnl_for_day including commission, set
current_pos to 0) ...
        action_this_bar = True # Flag that a stop occurred
```

- **Trailing Stop Adjustment (if holding and not stopped):**
  The stop is updated at the end of the day based on `today_close` and `today_atr_sl`.

```
# (If holding and not stopped or flipped)
elif current_pos != 0: # Holding
    if current_pos == 1:
        # ... (pnl_for_day calculation for holding) ...
        current_ts = max(current_ts, today_close - atr_multiplier_sl *
today_atr_sl)
    else: # current_pos == -1
        # ... (pnl_for_day calculation for holding) ...
        current_ts = min(current_ts, today_close + atr_multiplier_sl *
today_atr_sl)
```

1.5.4. Position Sizing, P&L, and Commission Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).

- **Commissions:** A commission rate (`comm_rate_val = commission_bps_per_side / 10000.0`) is applied to each side of a trade (entry and exit).
    - When entering a new position (from flat or flip): `pnl_entry_leg_day` calculations subtract `comm_rate_val`.
    - When exiting a position (due to stop or flip): `pnl_for_day` or `pnl_exit_leg_net` calculations subtract `comm_rate_val`.

```
# Example for stop-loss P&L
if current_pos == 1: pnl_for_day = (exit_price_sl_gross /
current_entry_gross) - 1 - comm_rate_val
else: pnl_for_day = -((exit_price_sl_gross / current_entry_gross) - 1) -
comm_rate_val

# Example for new entry P&L contribution for the day
if current_pos == 1:
    pnl_entry_leg_day = ((today_close / current_entry_gross) - 1) -
comm_rate_val
```

- **P&L Calculation (`Strategy_Daily_Return`):** Calculated daily, accounting for commissions and whether the action was a stop, new entry, flip, or hold.

1.6. Performance Evaluation

Daily strategy returns are used to compute various performance metrics.

- **Cumulative Returns & Alignment:** Standard calculation for strategy and buy & hold, with alignment.
- **Performance Metrics Function:** A loop iterates through strategy and benchmark returns to print:
    - Cumulative Return
    - Annualized Return
    - Annualized Volatility
    - Sharpe Ratio
    - Max Drawdown (`_calculate_max_drawdown_local` helper function is used).

```
# (Inside the performance metrics loop)
avg_r,std_r=returns_data.mean(),returns_data.std();
ann_r,ann_v=avg_r*TRADING_DAYS_PER_YEAR,std_r*np.sqrt(TRADING_DAYS_PER_Y
EAR);
sharpe=ann_r/ann_v if ann_v > 1e-7 else np.nan;
cum_r=(1+returns_data).prod();
mdd=_calculate_max_drawdown_local(returns_data) # Max Drawdown
calculation
print(f"\n--- {label_name} ---\nCum Ret: {cum_r:.2f}x | Ann Ret:
```

```
{ann_r:.2%} | Ann Vol: {ann_v:.2%} | Sharpe: {sharpe:.2f} | Max DD:
{mdd:.2%}")
```

1.7. Plotting Results

The script generates six separate plots if `plot_results` is true:

1. **Price & Adaptive Filter (ARL):** Shows Close price, the `filtered_price_arl_col`, and active Trailing Stops.
2. **ADX Trend Strength:** Displays the ADX value along with `adx_map_min` and `adx_map_max` lines.
3. **Adaptive EMA Period:** Shows the calculated `adaptive_ema_period_col` over time.
4. **Strategy Position:** Step plot of Long (1), Short (-1), or Flat (0) positions.
5. **ATR for Stop Loss:** Plots the `atr_col_name_sl`.
6. **Cumulative Performance (Log Scale):** Compares strategy returns to Buy & Hold.

```
# --- 5. Plotting Results (Separate Figures) ---
if plot_results and not df_analysis.empty and len(df_analysis) > 5:
    # Figure 1: Price & Adaptive Filter (ARL)
    plt.figure(figsize=(14, 7)); # ... plotting code ...; plt.show()
    # Figure 2: ADX Trend Strength
    plt.figure(figsize=(14, 7)); # ... plotting code ...; plt.show()
    # Figure 3: Adaptive EMA Period
    plt.figure(figsize=(14, 7)); # ... plotting code ...; plt.show()
    # Figure 4: Strategy Position
    plt.figure(figsize=(14, 7)); # ... plotting code ...; plt.show()
    # Figure 5: ATR for Stop Loss
    plt.figure(figsize=(14,7)); # ... plotting code ...; plt.show()
    # Figure 6: Cumulative Performance
    plt.figure(figsize=(14, 7)); # ... plotting code ...; plt.show()
```

1.8. Unique Features & Notes

- **Adaptive Filtering:** The core feature is the EMA-like filter whose responsiveness changes with trend strength (ADX). This allows the filter to be quick in fast markets and smooth in slower ones.
- **Iterative Calculation:** The adaptive filter is calculated iteratively due to the daily changing alpha, which is uncommon for standard EMA calculations in vectorized libraries but necessary here.
- **Commission Modeling:** Explicitly deducts commissions per side, providing a more realistic backtest.
- **Trend Strength Based Adaptation:** Uses ADX not as a direct entry signal, but as a meta-parameter to control another indicator (the filter).

- **Daily Timeframe:** The strategy operates on daily data.

---

2. Adaptive Savitzky-Golay Filter Strategy

2.1. Overview and Objective

Overview:
This strategy employs an **Adaptive Savitzky-Golay (SG) filter** to smooth the price series. The Savitzky-Golay filter is a type of polynomial filter that can be effective for smoothing data while preserving features like local maxima and minima better than simpler moving averages. The "adaptive" nature of this strategy comes from dynamically adjusting the **window length** of the SG filter based on recent market volatility conditions.

Objective:
The strategy aims to generate trading signals by identifying crossovers between the closing price and this adaptively filtered price series. The core idea is that:

- In **calm markets** (low recent volatility percentile), a **smaller SG filter window** is used, making the filter more responsive to price changes.
- In **choppy markets** (high recent volatility percentile), a **larger SG filter window** is used, providing more smoothing to filter out noise.
  Trades are initiated based on the previous day's close crossing the previous day's SG filtered price, with an ATR-based trailing stop-loss for risk management and accounting for commissions.

2.2. Key Concepts: Savitzky-Golay Filter

The **Savitzky-Golay filter** is a digital filter that fits successive sub-sets of adjacent data points with a low-degree polynomial by the method of linear least squares. It's effective for smoothing a noisy signal to better represent the underlying trend without overly distorting it. Key parameters for an SG filter are:

- **Window Length:** The number of data points used to fit the polynomial. Must be odd.
- **Polynomial Order:** The degree of the polynomial used for fitting. Must be less than the window length.

In this strategy, the **window length** is adapted daily.

2.3. Key Indicators and Components

The strategy utilizes several calculated series and functions:

- **Historical Volatility (`volatility_col`):** Standard deviation of daily returns over `vol_window` .

```
df[daily_return_col] = df['Close'].pct_change()
df[volatility_col] =
df[daily_return_col].rolling(window=vol_window).std()
```

- **Volatility Percentile (`vol_percentile_col`):** The rolling percentile rank of the historical volatility over `vol_percentile_window` . This value (0.0 to 1.0) indicates how the current volatility compares to its recent history.

```
df[vol_percentile_col] =
df[volatility_col].rolling(window=vol_percentile_window)\
    .apply(lambda x: pd.Series(x).rank(pct=True).iloc[-1] if not x.empty
and pd.notna(x.iloc[-1]) else np.nan, raw=False)
df[vol_percentile_col].fillna(method='ffill', inplace=True);
df[vol_percentile_col].fillna(0.5, inplace=True)
```

- **Adaptive SG Window (`adaptive_sg_window_col`):** The window length for the Savitzky-Golay filter, determined by the *previous day's* `vol_percentile_col` . Low volatility percentile maps to `sg_min_window` , high percentile maps to `sg_max_window` . The `adjust_window` helper function ensures the window is odd and respects the polynomial order.

```
# Low vol percentile (calm, 0.0) -> sg_min_window
# High vol percentile (choppy, 1.0) -> sg_max_window
df['Adaptive_SG_Window_Float'] = sg_min_window +
df[vol_percentile_col].shift(1) * (sg_max_window - sg_min_window)

def adjust_window(w_float, p_order, w_min, w_max): # Helper function
    # ... (logic to ensure odd, within bounds, and > polyorder) ...
    return final_w

df[adaptive_sg_window_col] = df['Adaptive_SG_Window_Float'].apply(
    lambda w: adjust_window(w, sg_polyorder, sg_min_window,
sg_max_window)
).fillna(sg_min_window).astype(int)
```

- **Adaptive Savitzky-Golay Filtered Price (`filtered_price_sg_col`):** The price series smoothed by the SG filter with a dynamically changing window length. Calculated iteratively using `scipy.signal.savgol_filter` .

```
df[filtered_price_sg_col] = np.nan
# ... (logic for start_loc_for_sg_loop and seeding first value) ...

for i_loop in range(start_loc_for_sg_loop, len(df)):
    idx_today = df.index[i_loop]
    current_sg_window = df.loc[idx_today, adaptive_sg_window_col]
    if pd.isna(current_sg_window) or i_loop - current_sg_window + 1 < 0
:
        df.loc[idx_today, filtered_price_sg_col] =
df.loc[df.index[i_loop-1], filtered_price_sg_col]
        continue

    price_segment = df['Close'].iloc[i_loop - current_sg_window + 1 :
i_loop + 1].values

    if len(price_segment) < current_sg_window or len(price_segment) <=
sg_polyorder :
        df.loc[idx_today, filtered_price_sg_col] =
df.loc[df.index[i_loop-1], filtered_price_sg_col]
        continue

    try:
        smoothed_segment = savgol_filter(price_segment,
window_length=current_sg_window, polyorder=sg_polyorder)
        df.loc[idx_today, filtered_price_sg_col] = smoothed_segment[-1]
# Get the last point of the smoothed segment
    except ValueError:
        df.loc[idx_today, filtered_price_sg_col] =
df.loc[df.index[i_loop-1], filtered_price_sg_col]
df[filtered_price_sg_col].fillna(method='ffill', inplace=True);
df[filtered_price_sg_col].fillna(method='bfill', inplace=True)
```

- **Average True Range (ATR - `atr_col_name_sl`):** Used for the trailing stop-loss.

```
df['H-L_sl'] = df['High'] - df['Low']; df['H-PC_sl'] = (df['High'] -
df['Close'].shift(1)).abs(); df['L-PC_sl'] = (df['Low']  -
df['Close'].shift(1)).abs()
df['TR_sl'] = df[['H-L_sl','H-PC_sl','L-PC_sl']].max(axis=1);
df[atr_col_name_sl] = df['TR_sl'].rolling(window=atr_window_sl).mean()
```

2.4. Script Parameters (Function Arguments)

The core logic is encapsulated in the `run_adaptive_sg_filter_backtest` function, which accepts the following parameters:

```
def run_adaptive_sg_filter_backtest(
    ticker="BTC-USD",
    start_date_str="2024-01-01",
    end_date_str="2024-12-31",
    vol_window=20,
    vol_percentile_window=100,
    sg_polyorder=2,          # Must be less than sg_min_window
    sg_min_window=5,          # Must be odd and > sg_polyorder
    sg_max_window=51,         # Must be odd and > sg_polyorder
    atr_window_sl=14,
    atr_multiplier_sl=2.0,
    commission_bps_per_side=1.0,
    plot_results=True,
    verbose=True
):
# ... (parameter validation for sg_min_window, sg_max_window, sg_polyorder)
...
```

The `if __name__ == '__main__':` block provides an example of how to call this function with specific parameter values.

2.5. Data Handling

- **Data Download:** Daily OHLCV data is fetched via `yfinance`. Standard `droplevel` and column selection logic is applied.

  ```
  # --- 1. Download Data ---
  df_raw = yf.download([ticker], start=start_date_str, end=end_date_str,
  auto_adjust=False, progress=False)
  # ... (error handling, droplevel, column selection) ...
  ```

2.6. Trading Logic (within `run_adaptive_sg_filter_backtest`)

The strategy iterates daily. Signals are derived from the *previous day's* close versus the *previous day's* SG filtered price. Trades are executed at the *current day's* `Open`.

2.6.1. Signal Generation Conditions

A target position (`target_pos`) is determined by comparing `prev_close` to `prev_filtered_price`:

```
# (Inside backtesting loop, after stop-loss check)
    target_pos = 0
    if prev_close > prev_filtered_price: target_pos = 1  # Signal Long
    elif prev_close < prev_filtered_price: target_pos = -1 # Signal Short
```

2.6.2. Entry Conditions

If `target_pos` indicates a new trade or a flip from an existing position, the trade is entered at `today_open`. Commission is applied to the entry price.

```
# (If target_pos != 0 and target_pos != current_pos)
            if current_pos != 0 and pd.notna(current_entry_gross): # It's a
flip
                # ... (calculate pnl_exit_leg_net for the closing part of
the flip, including commission) ...

            current_pos = target_pos; current_entry_gross = today_open # Set
new position and gross entry price
            cost_entry = current_entry_gross * comm_rate_per_side #
Calculate entry commission

            if current_pos == 1: # Entering New Long
                effective_entry_costed = current_entry_gross + cost_entry
                pnl_entry_leg_day = (today_close / effective_entry_costed) -
1 if effective_entry_costed !=0 else 0
                # ... (set init_ts and current_ts) ...
            elif current_pos == -1: # Entering New Short
                effective_entry_costed = current_entry_gross - cost_entry
                pnl_entry_leg_day = -((today_close / effective_entry_costed)
- 1) if effective_entry_costed !=0 else 0
                # ... (set init_ts and current_ts) ...
```

2.6.3. Exit Conditions

Primarily through an ATR Trailing Stop-Loss. Positions also close if an opposite signal generates a flip.

- **ATR Trailing Stop-Loss Check (highest priority):**

```
# (At the start of the daily loop logic for an active position)
if current_pos != 0 and pd.notna(current_ts) and
pd.notna(current_entry_gross):
    exit_price_sl_gross = 0; stopped_out = False
    if current_pos == 1 and today_low <= current_ts:
        exit_price_sl_gross = min(today_open, current_ts); stopped_out =
True
    elif current_pos == -1 and today_high >= current_ts:
        exit_price_sl_gross = max(today_open, current_ts); stopped_out =
True
    if stopped_out:
        cost_exit = exit_price_sl_gross * comm_rate_per_side
        if current_pos == 1: pnl_for_day = ((exit_price_sl_gross -
cost_exit) / current_entry_gross) - 1
        else: pnl_for_day = -(((exit_price_sl_gross + cost_exit) /
current_entry_gross) - 1)
        current_pos = 0; action_this_bar = True # Flag that a stop
occurred
```

- **Trailing Stop Adjustment (if holding and not stopped):** Updated daily.

```
# (If holding and not stopped or flipped)
elif current_pos != 0: # Holding
    if current_pos == 1:
        # ... (pnl_for_day calculation for holding) ...
        current_ts = max(current_ts, today_close - atr_multiplier_sl *
today_atr_sl)
    else: # current_pos == -1
        # ... (pnl_for_day calculation for holding) ...
        current_ts = min(current_ts, today_close + atr_multiplier_sl *
today_atr_sl)
```

2.6.4. Position Sizing, P&L, and Commission Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **Commissions (`comm_rate_per_side`):** Calculated as `commission_bps_per_side / 10000.0` .
    - For entries: `cost_entry = current_entry_gross * comm_rate_per_side` . `effective_entry_costed` is `current_entry_gross + cost_entry` (long) or `current_entry_gross - cost_entry` (short).
    - For stop-loss exits: `cost_exit = exit_price_sl_gross * comm_rate_per_side` . P&L is based on `(exit_price_sl_gross - cost_exit)` or `(exit_price_sl_gross + cost_exit)` .
    - For flip exits: Similar logic applied to the exit leg at `today_open` .
- **P&L Calculation (`Strategy_Daily_Return`):** Daily P&L considers gross prices and then adjusts for commissions for entries/exits. For holding days, it's based on `today_close / prev_close` .

2.7. Performance Evaluation

The script calculates and stores several performance metrics in a dictionary.

- **Max Drawdown:** A helper function `calculate_max_drawdown` is defined and used.

```
def calculate_max_drawdown(returns_series):
    # ... (implementation as provided in the script) ...
    return max_drawdown_val if pd.notna(max_drawdown_val) else 0.0
```

- **Metrics Calculation (`_calc_and_store_metrics` local function):**
This function computes Cumulative Return, Annualized Return, Annualized Volatility, Sharpe Ratio, and Max Drawdown.
`python # (Inside run_adaptive_sg_filter_backtest, after backtest loop) def _calc_and_store_metrics(returns, name, tdpy, results_dict_ref): # ... (Check`

`for sufficient data) ... avg_r,std_r=returns.mean(),returns.std(); ann_r,ann_v=avg_r*tdpy,std_r*np.sqrt(tdpy); sharpe=ann_r/ann_v if ann_v > 1e-7 else np.nan; cum_r=(1+returns).prod(); mdd=calculate_max_drawdown(returns) # Using the external helper results_dict_ref[name] = { 'Cumulative Return':cum_r, 'Annualized Return':ann_r, 'Annualized Volatility':ann_v, 'Sharpe Ratio':sharpe, 'Max Drawdown':mdd } if verbose: print(f"\n--- {name} ---\nCum Ret: {cum_r:.2f}x | ...") # Called for strategy and benchmark _calc_and_store_metrics(strat_returns, strategy_lbl, TRADING_DAYS_PER_YEAR, metrics_results) _calc_and_store_metrics(bh_returns, f"{ticker} Buy & Hold", TRADING_DAYS_PER_YEAR, metrics_results)`
The script returns a dictionary `metrics_results`.

2.8. Plotting Results

If `plot_results` is true, six separate plots are generated:

1. **Price & Adaptive Savitzky-Golay Filter:** Shows Close price, the `filtered_price_sg_col`, and active Trailing Stops.
2. **Historical Volatility & Rolling Percentile:** Displays raw volatility and its rolling percentile on a twin axis.
3. **Adaptive Savitzky-Golay Window Length:** Plots the `adaptive_sg_window_col` (number of bars).
4. **Strategy Position:** Step plot of Long (1), Short (-1), or Flat (0) positions.
5. **ATR for Stop Loss:** Shows the `atr_col_name_sl`.
6. **Cumulative Performance (Log Scale):** Compares strategy returns to Buy & Hold.

```
# --- Plotting ---
if plot_results and not df_analysis.empty and len(df_analysis) > 5:
    # fig, axs = plt.subplots(6, 1, ...)
    # axs[0]: Price & Adaptive SG Filter
    # axs[1]: Historical Volatility & Rolling Percentile
    # axs[2]: Adaptive SG Window Length
    # axs[3]: Strategy Position
    # axs[4]: ATR for Stop Loss
    # axs[5]: Cumulative Performance
    # ... (detailed plotting calls for each) ...
    plt.show()
```

2.9. Unique Features & Notes

- **Adaptive Savitzky-Golay Filter:** The key feature is the use of an SG filter whose window length adapts to market volatility (percentile rank). This differs from the previous strategy which adapted EMA period based on ADX.

- **Volatility-Based Window Sizing:** Calm markets (low vol percentile) use a shorter SG window, aiming for more responsiveness. Choppy markets (high vol percentile) use a longer SG window for increased smoothing.
- **Iterative Filtering:** The SG filter is applied iteratively because the window length can change daily.
- **Scipy Dependency:** Relies on `scipy.signal.savgol_filter`.
- **Commission Modeling:** Explicitly includes and applies commissions per side.
- **Parameter Constraints:** `sg_polyorder` must be less than `sg_min_window`, and window lengths must be odd. The `adjust_window` function helps manage these constraints.
- **Daily Timeframe:** Operates on daily data.

---

3. Dynamic Wavelet-Filter Hybrid Strategy

3.1. Overview and Objective

Overview:
This strategy utilizes the Discrete Wavelet Transform (DWT) to decompose the price series into different frequency components within a rolling window. It then reconstructs a filtered price signal by selectively using and modifying these components. Specifically, it aims to keep the lower-frequency approximation, adjust mid-frequency detail coefficients based on a volatility-derived gain, and discard the highest-frequency details.

Objective:
The objective is to create an adaptive filter that responds to changing market volatility by dynamically adjusting the influence of mid-band wavelet coefficients.

- When recent volatility is high, the "gain" applied to these mid-band coefficients is increased, potentially making the filter more sensitive to these components.
- When recent volatility is low, the gain is decreased.
  Trading signals are then generated based on crossovers of the closing price with this dynamically filtered price series. An ATR-based trailing stop-loss is used for risk management.

3.2. Key Concepts: Discrete Wavelet Transform (DWT) & Coefficient Weighting

- **Discrete Wavelet Transform (DWT):** The DWT decomposes a signal (here, a window of closing prices) into hierarchal sets of coefficients:
  - **Approximation Coefficients (cA):** Represent the low-frequency, trend-like part of the signal at the chosen decomposition level.

  - **Detail Coefficients (cD):** Represent the higher-frequency parts. For a multi-level decomposition (e.g., level `N`), you get `cD_N, cD_N-1, ..., cD_1`, where `cD1` are the highest frequency details.
- **Coefficient Weighting:** This strategy modifies the wavelet coefficients before reconstruction. It specifically targets the "mid-band" detail coefficients (interpreted as `cD2` in a 2-level decomposition context as suggested by the script's problem description, though parameter `dwt_level` is 3 in the provided code which would cause an error with the current coefficient unpacking `cA2, cD2, cD1 = coeffs`). The `cD2` coefficients are scaled by a `volatility_gain_col`. The highest frequency details (`cD1`) are zeroed out, and the approximation (`cA2`) is preserved. The filtered signal is then reconstructed using these modified coefficients.
  - **Note on `dwt_level`:** The script has a parameter `dwt_level = 3` but the coefficient unpacking `cA2, cD2, cD1 = coeffs` and subsequent modification `coeffs_modified = [cA2, cD2_adjusted, cD1_zeroed]` are structured for a 2-level decomposition. For these lines to execute correctly, `dwt_level` should be set to 2. The manual will proceed assuming the logic intended for a 2-level effective processing based on the code's structure for coefficient manipulation.

3.3. Key Indicators and Components

- **Historical Volatility (`volatility_col`):** Rolling standard deviation of daily returns.

```
df[daily_return_col] = df['Close'].pct_change()
df[volatility_col] =
df[daily_return_col].rolling(window=vol_window_for_gain).std()
```

- **Normalized Volatility (`normalized_vol_col`):** Volatility normalized to a 0-1 range over `vol_norm_window_for_gain`.

```
rolling_min_vol =
df[volatility_col].rolling(window=vol_norm_window_for_gain).min()
rolling_max_vol =
df[volatility_col].rolling(window=vol_norm_window_for_gain).max()
range_vol = rolling_max_vol - rolling_min_vol
df[normalized_vol_col] = ((df[volatility_col] - rolling_min_vol) /
range_vol.replace(0, np.nan)).fillna(0.5).clip(0,1)
```

- **Volatility-Based Gain (`volatility_gain_col`):** Interpolated between `gain_min` and `gain_max` based on the *previous day's* normalized volatility. This gain is applied to the `cD2` wavelet coefficients.

```
df[volatility_gain_col] = gain_min + df[normalized_vol_col].shift(1) *
(gain_max - gain_min)
```

```
df[volatility_gain_col].fillna((gain_min + gain_max) / 2, inplace=True)
```

- **Adaptive Wavelet Filtered Price (`filtered_price_wavelet_col`):** Calculated iteratively by applying DWT to a rolling window of closing prices, modifying coefficients, and then performing an inverse DWT.

```
# (Inside the iterative loop for i_loop)
current_price_window = df['Close'].iloc[i_loop - dwt_processing_window +
1 : i_loop + 1].values
# ... (window length check) ...
current_gain = df.loc[idx_today, volatility_gain_col]
# ... (gain NaN check) ...

try:
    # Assuming dwt_level allows unpacking into cA2, cD2, cD1 (i.e.,
dwt_level=2)
    coeffs = pywt.wavedec(current_price_window, wavelet_type,
level=dwt_level) # dwt_level parameter
    cA2, cD2, cD1 = coeffs # This unpacking implies level=2 processing

    cD2_adjusted = cD2 * current_gain
    cD1_zeroed = np.zeros_like(cD1)

    coeffs_modified = [cA2, cD2_adjusted, cD1_zeroed] # Structure for
level 2 reconstruction

    filtered_segment = pywt.waverec(coeffs_modified, wavelet_type)

    # Use the last point of the reconstructed segment
    df.loc[idx_today, filtered_price_wavelet_col] =
filtered_segment[len(current_price_window)-1]

except ValueError as e: # Handle errors from DWT/IDWT
    df.loc[idx_today, filtered_price_wavelet_col] =
df.loc[df.index[i_loop-1], filtered_price_wavelet_col] if i_loop > 0
else np.nan

df[filtered_price_wavelet_col].fillna(method='ffill', inplace=True)
df[filtered_price_wavelet_col].fillna(method='bfill', inplace=True)
```

- **Average True Range (ATR - `atr_col_name_sl`):** For the trailing stop-loss.

```
df['H-L_sl'] = df['High'] - df['Low']; df['H-PC_sl'] = (df['High'] -
df['Close'].shift(1)).abs(); #...
df['TR_sl'] = df[['H-L_sl','H-PC_sl','L-PC_sl']].max(axis=1);
df[atr_col_name_sl] = df['TR_sl'].rolling(window=atr_window_sl).mean()
```

3.4. Script Parameters

Key parameters defined at the script's start:

```
# --- Parameters ---
ticker = "BTC-USD"
start_date = "2024-01-01" # Example of a shorter period
end_date = "2024-12-31"

# Wavelet Parameters
wavelet_type = 'db4'
dwt_level = 3           # DWT decomposition level (Note: code structure for
coeff modification implies level 2 processing)
dwt_processing_window = 90 # Rolling window for DWT

# Volatility-Based Gain Parameters
vol_window_for_gain = 30
vol_norm_window_for_gain = 90
gain_min = 0.5             # Min gain for cD2 (low vol)
gain_max = 1.5             # Max gain for cD2 (high vol)

# ATR Trailing Stop Parameters
atr_window_sl = 14
atr_multiplier_sl = 1.0

TRADING_DAYS_PER_YEAR = 252 # Adjusted for crypto if detected
```

3.5. Data Handling

- **Data Download:** Standard daily OHLCV data download using `yfinance`.

  ```
  # --- 1. Download Data ---
  df_raw = yf.download([ticker], start=start_date, end=end_date,
  auto_adjust=False, progress=False)
  # ... (droplevel and column selection) ...
  ```

3.6. Trading Logic (within the main backtesting loop)

The strategy iterates daily. Signals are based on the *previous day's* close vs. filtered price; trades execute at the *current day's* `Open`. The backtest loop structure and P&L calculations are similar to previous adaptive filter examples. **Note:** `comm_rate_per_side` is hardcoded to `0.0` in this script's loop, meaning no commissions are applied in this specific implementation, though the P&L logic can accommodate it.

3.6.1. Signal Generation Conditions

A target position (`target_pos`) is determined by comparing `prev_close` to `prev_filtered_price` (the wavelet-filtered price):

```
# (Inside backtesting loop, after stop-loss check)
    target_pos = 0
    if prev_close > prev_filtered_price: target_pos = 1  # Signal Long
    elif prev_close < prev_filtered_price: target_pos = -1 # Signal Short
```

3.6.2. Entry Conditions

If `target_pos` suggests a new trade or a flip, entry occurs at `today_open` .

```
# (If target_pos != 0 and target_pos != current_pos)
            if current_pos != 0 and pd.notna(current_entry_gross): # It's a
flip
                # ... (calculate pnl_exit_leg_net for the closing part of
the flip) ...

            current_pos = target_pos; current_entry_gross = today_open

            if current_pos == 1: # Entering New Long
                pnl_entry_leg_day = ((today_close / current_entry_gross) -
1) - comm_rate_per_side # comm_rate_per_side is 0
                # ... (set init_ts and current_ts) ...
            elif current_pos == -1: # Entering New Short
                pnl_entry_leg_day = (-((today_close / current_entry_gross) -
1)) - comm_rate_per_side # comm_rate_per_side is 0
                # ... (set init_ts and current_ts) ...
```

3.6.3. Exit Conditions

Primarily managed by an ATR Trailing Stop-Loss. Positions also close upon an opposing signal (flip).

- **ATR Trailing Stop-Loss Check:**

```
# (At the start of the daily loop logic for an active position)
if current_pos != 0 and pd.notna(current_ts) and
pd.notna(current_entry_gross):
    # ... (stop-loss hit logic as in previous scripts) ...
    if stopped_out:
        # ... (P&L calculation for stop-loss, including
comm_rate_per_side which is 0) ...
        current_pos = 0; action_this_bar = True
```

- **Trailing Stop Adjustment:** Updated daily if holding.

```
# (If holding and not stopped or flipped)
elif current_pos != 0:
    # ... (stop adjustment logic as in previous scripts) ...
```

3.6.4. Position Sizing, P&L, and Commission Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **Commissions:** `comm_rate_per_side` is initialized to `0.0` within the backtest loop of this specific script. While the P&L logic includes subtraction of `comm_rate_per_side`, it effectively means **no commissions are applied** in this version unless that variable is changed.
- **P&L Calculation (** `Strategy_Daily_Return` **):** Daily P&L based on stop-outs, flips, new entries, or holding.

3.7. Performance Evaluation

Standard performance metrics are calculated from daily strategy returns.

- **Max Drawdown:** The `calculate_max_drawdown` helper function is available.
- **Metrics Calculation (** `_calc_and_store_metrics` **local function):** Similar to previous scripts, computes and prints/stores Cumulative Return, Annualized Return, etc.

```
# (After backtest loop)
def _calc_and_store_metrics(returns, name, tdpy, results_dict_ref):
    # ... (implementation as in previous script) ...
strategy_lbl = f"DynWaveletFilter (W:{wavelet_type},Lvl:
{dwt_level},ProcW:{dwt_processing_window},Gain:[{gain_min}-
{gain_max}],SL:{atr_multiplier_sl}x{atr_window_sl})"
_calc_and_store_metrics(strat_returns, strategy_lbl,
TRADING_DAYS_PER_YEAR, metrics_results)
# ... (benchmark metrics) ...
```

3.8. Plotting Results

Six plots are generated for visualization:

1. **Price & Adaptive Wavelet Filter:** Close price, the `filtered_price_wavelet_col`, and Trailing Stops.
2. **Historical Volatility & Normalized Volatility:** Raw volatility and its 0-1 normalized version.
3. **Volatility-Based Gain for cD2:** The `volatility_gain_col` applied to mid-band coefficients.
4. **Strategy Position:** Long/Short/Flat positions.

5. **ATR for Stop Loss.**
6. **Cumulative Performance (Log Scale):** Strategy vs. Buy & Hold.

```python
# --- 5. Plotting ---
if not df_analysis.empty and len(df_analysis) > 5:
    fig, axs = plt.subplots(6, 1, figsize=(15, 28), sharex=True)
    # axs[0]: Price & Adaptive Wavelet Filter
    # axs[1]: Historical Volatility & Normalized Volatility
    # axs[2]: Volatility-Based Gain
    # axs[3]: Strategy Position
    # axs[4]: ATR for Stop Loss
    # axs[5]: Cumulative Performance
    # ... (detailed plotting calls) ...
    plt.show()
```

3.9. Unique Features & Notes

- **Dynamic Wavelet Filtering:** The core is the DWT applied to rolling windows, with dynamic adjustment of mid-band detail coefficient weights based on overall market volatility.
- **Volatility-Sensitive Gain:** Higher recent volatility leads to a higher gain applied to selected wavelet coefficients, potentially making the filter more responsive or emphasizing certain frequency components during such times.
- **Complexity:** Involves advanced signal processing (DWT) and iterative calculation of the filtered price.
- `PyWavelets` **Dependency:** Requires the `PyWavelets` library.
- **Coefficient Selection &** `dwt_level`**:** The script's logic for modifying coefficients (`cA2, cD2, cD1 = coeffs`) is specific to a 2-level DWT. If `dwt_level` parameter is set to a value other than 2 (e.g., the default 3 in parameters), the coefficient unpacking will cause an error. For the current code to work as intended with the specified coefficient manipulation, `dwt_level` should be 2.
- **Iterative Application:** The wavelet filtering is applied to a moving window of data, and the filtered price is extracted point-by-point.
- **Commissions:** While the P&L structure supports commissions, they are effectively set to zero in this script's backtesting loop by `comm_rate_per_side = 0.0`.

---

4. FRAMA (Fractal Adaptive Moving Average) Crossover Strategy

4.1. Overview and Objective

Overview:
This strategy implements a **Fractal Adaptive Moving Average (FRAMA)** concept by dynamically selecting an Exponential Moving Average (EMA) based on the prevailing market regime, as estimated by the **Hurst exponent**. The Hurst exponent helps to classify the market as trending, mean-reverting, or in a more random state. Based on this classification, the strategy switches between short, medium, or long period EMAs to act as its adaptive filter.

Objective:
The primary goal is to trade crossovers of the closing price with this Hurst-adapted EMA (termed `FRAMA_H`).

- In **trending markets** (high Hurst exponent), a faster EMA is used to quickly capture trend continuations.
- In **mean-reverting markets** (low Hurst exponent), a slower EMA is used, potentially aiming to filter noise and capture reversions, though the crossover logic remains directional.
- In **intermediate (random/undetermined) markets**, a medium-period EMA is used. An ATR-based trailing stop-loss is employed for risk management.

4.2. Key Concepts: Hurst Exponent & FRAMA

- **Hurst Exponent (H):** A measure used to classify time series.
  - **H > 0.5:** Indicates a persistent or trending series (positive autocorrelation). The strategy uses `H > h_trending_threshold` (e.g., 0.6) for this regime.
  - **H < 0.5:** Indicates an anti-persistent or mean-reverting series (negative autocorrelation). The strategy uses `H < h_reverting_threshold` (e.g., 0.4) for this regime.
  - **H = 0.5:** Suggests a random walk (no correlation).
    The Hurst exponent is calculated on a rolling window of log-returns using the `hurst` library.
- **FRAMA (Fractal Adaptive Moving Average - `FRAMA_H`):** In this context, FRAMA is not a specific complex formula but rather a conceptual approach where the active EMA (from a set of predefined EMAs) is selected based on the calculated Hurst exponent regime.

4.3. Key Indicators and Components

- **Hurst Exponent (`hurst_col`):** Calculated on rolling windows of log-returns.

```
# Compute Hurst on log-returns
def get_hurst(series):
    """
    series: 1D array of log-returns, length >=100
    returns: H
    """
    if len(series) < 100 or np.std(series) == 0:
        return np.nan
    H, c, data = compute_Hc(series, kind='change', simplified=True)
    return H

log_rets = np.log(df['Close']).diff().dropna()
df.loc[log_rets.index, hurst_col] = (
    log_rets
      .rolling(window=hurst_lookback_window)
      .apply(get_hurst, raw=False)
)
df[hurst_col].fillna(method='ffill', inplace=True)
df[hurst_col].fillna(0.5, inplace=True) # Default to neutral if still
NaN
```

- **Exponential Moving Averages (EMAs):** Three EMAs with different periods are calculated.

```
df[ema_short_col]  = df['Close'].ewm(span=ema_short_window,
adjust=False).mean()
df[ema_medium_col] = df['Close'].ewm(span=ema_medium_window,
adjust=False).mean()
df[ema_long_col]   = df['Close'].ewm(span=ema_long_window,
adjust=False).mean()
```

- **FRAMA_H (`frama_h_col`):** The adaptively selected EMA based on the *previous day's* Hurst exponent (`h_shift`).

```
h_shift = df[hurst_col].shift(1) # Use lagged Hurst to avoid lookahead
df[frama_h_col] = np.select(
    [h_shift > h_trending_threshold,  # Condition for trending
     h_shift < h_reverting_threshold], # Condition for mean-reverting
    [df[ema_short_col], df[ema_long_col]], # Choices for these
conditions
    default=df[ema_medium_col]          # Default choice
)
```

- **Average True Range (ATR - `atr_col_name_sl`):** For the trailing stop-loss.

```
df['H-L_sl']      = df['High'] - df['Low']
df['H-PC_sl']     = (df['High'] - df['Close'].shift(1)).abs()
df['L-PC_sl']     = (df['Low']  - df['Close'].shift(1)).abs()
df['TR_sl']       = df[['H-L_sl','H-PC_sl','L-PC_sl']].max(axis=1)
df[atr_col_name_sl] = df['TR_sl'].rolling(window=atr_window_sl).mean()
```

4.4. Script Parameters

Configurable parameters defined at the script's start:

```
# --- Parameters ---
ticker = "BTC-USD"
start_date = "2021-01-01"
end_date = "2024-12-31"

# Hurst Exponent Parameters
hurst_lookback_window = 128  # Must be >=100 for compute_Hc

h_trending_threshold = 0.6
h_reverting_threshold = 0.4

ema_short_window = 7
ema_medium_window = 30
ema_long_window = 90

# ATR Trailing Stop Parameters
atr_window_sl = 14
atr_multiplier_sl = 1.0

TRADING_DAYS_PER_YEAR = 365 # Adjusted for crypto
```

4.5. Data Handling

- **Data Download:** Standard daily OHLCV data download using `yfinance`.

  ```
  # --- 1. Download Data ---
  df_raw = yf.download(
      [ticker], start=start_date, end=end_date,
      auto_adjust=False, progress=False
  )
  # ... (droplevel and column selection logic) ...
  ```

4.6. Trading Logic (within the main backtesting loop)

The strategy iterates daily. Signals derive from the *previous day's* close versus the *previous day's* `FRAMA_H`. Trades execute at the *current day's* `Open`.

4.6.1. Signal Generation Conditions

A trading signal (`signal`) is generated based on the crossover:

```
# (Inside backtest loop, if pos==0, i.e., currently flat)
    signal = 1 if pc > ph else (-1 if pc < ph else 0)
    # pc = prev_close, ph = prev_frama_h_col
```

4.6.2. Entry Conditions

If flat (`pos==0`) and a non-zero `signal` is generated, a position is entered at `today` 's `Open` price (`o`).

```
# (Inside backtest loop, if pos==0 and signal!=0)
    if signal != 0:
        pos = signal
        ep = o # entry_price = today's Open
        if pos == 1: # Long entry
            pnl = (c / ep) - 1 # P&L for the entry day (Close / Open)
            init_ts = ep - atr_multiplier_sl * pa # pa = prev_atr_sl
            ts = max(init_ts, c - atr_multiplier_sl * ta) # ta =
today_atr_sl
        else: # Short entry
            pnl = -((c / ep) - 1)
            init_ts = ep + atr_multiplier_sl * pa
            ts = min(init_ts, c + atr_multiplier_sl * ta)
```

4.6.3. Exit Conditions

Primarily via an ATR Trailing Stop-Loss. Positions are implicitly exited if an opposing signal leads to a flip (though this script enters only when flat).

- **ATR Trailing Stop-Loss Check (highest priority):**

  ```
  # (At the start of the daily loop logic for an active position)
  if pos == 1 and l <= ts: # l = today_low, ts = active_ts
      exit_price = min(o, ts) # o = today_open
      pnl = (exit_price / ep) - 1 # ep = entry_price
      pos, ep, ts = 0, np.nan, np.nan # Go flat
  elif pos == -1 and h >= ts: # h = today_high
      exit_price = max(o, ts)
      pnl = -((exit_price / ep) - 1)
      pos, ep, ts = 0, np.nan, np.nan # Go flat
  ```

- **Trailing Stop Adjustment (if holding and not stopped):**

  ```
  # (If holding position, else block after stop check)
  else: # Not stopped
      if pos == 1: # Holding long
          pnl = (c / pc) - 1 # c = today_close, pc = prev_close
  ```

```
        ts = max(ts, c - atr_multiplier_sl * ta) # ta = today_atr_sl
    else: # Holding short
        pnl = -((c / pc) - 1)
        ts = min(ts, c + atr_multiplier_sl * ta)
```

4.6.4. Position Sizing & P&L Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **P&L Calculation (`Strategy_Daily_Return`):** Calculated daily (`pnl`).
    - On entry: P&L from `today_open` to `today_close` .
    - If stopped out: P&L from entry price to stop-loss exit price.
    - If holding: P&L from `prev_close` to `today_close` .
- **Commissions:** This script's backtest loop does not explicitly deduct commissions in its P&L calculations.

4.7. Performance Evaluation

A local `metrics` function calculates and prints key performance statistics.

```
# --- 4. Performance ---
def metrics(returns, name):
    r = returns.dropna()
    μ, σ = r.mean(), r.std()
    ann_r = μ * TRADING_DAYS_PER_YEAR
    ann_v = σ * np.sqrt(TRADING_DAYS_PER_YEAR)
    sr = ann_r / ann_v if ann_v > 1e-8 else np.nan
    cum = (1 + r).prod()
    print(f"\n--- {name} ---")
    print(f"Cumulative: {cum:.2f}x  |  Ann. Ret: {ann_r:.2%}  |  Ann. Vol:
{ann_v:.2%}  |  Sharpe: {sr:.2f}")

metrics(df_analysis['Strategy_Daily_Return'].iloc[1:], "FRAMA_H Strategy")
metrics(bh.iloc[1:], f"{ticker} Buy & Hold")
```

Metrics include Cumulative Return, Annualized Return, Annualized Volatility, and Sharpe Ratio.

4.8. Plotting Results

The script generates five plots for visualization:

1. **Price & FRAMA_H:** Close price and the adaptively selected `FRAMA_H` line.
2. **Hurst Exponent:** The calculated rolling Hurst exponent with a 0.5 reference line.
3. **Position:** Step plot showing Long/Short/Flat positions.
4. **ATR SL:** The value of the ATR used for stop-loss calculation.

5. **Cumulative Returns:** Log-scaled comparison of strategy returns vs. Buy & Hold.

```
# --- 5. Plots (optional) ---
# Plot 1: Price & FRAMA_H
plt.figure(figsize=(10, 6)); # ... plotting code ...; plt.show()
# Plot 2: Hurst Exponent
plt.figure(figsize=(10, 6)); # ... plotting code ...; plt.show()
# Plot 3: Position
plt.figure(figsize=(10, 6)); # ... plotting code ...; plt.show()
# Plot 4: ATR SL
plt.figure(figsize=(10, 6)); # ... plotting code ...; plt.show()
# Plot 5: Cumulative Returns
plt.figure(figsize=(10, 6)); # ... plotting code ...; plt.show()
```

4.9. Unique Features & Notes

- **Hurst-Based Adaptation:** The core innovation is adapting the EMA period based on the Hurst exponent's indication of market regime (trending, mean-reverting, or random).
- **Regime-Specific EMA Choice:** Uses a shorter EMA for trending markets, a longer EMA for mean-reverting markets, and a medium EMA otherwise.
- `hurst` **Library Dependency:** This strategy requires the `hurst` library (`pip install hurst`).
- **Computational Cost of Hurst:** Rolling Hurst exponent calculation can be computationally intensive, especially over long datasets or with large windows.
- **Hurst Window Requirement:** The `compute_Hc` function typically requires a window of at least 100 data points for reliable estimation. The script uses `hurst_lookback_window = 128`.
- **No Explicit Commissions in Loop:** The P&L calculations in the provided backtest loop do not factor in trading commissions.

---

5. Gopalakrishnan RAVI-Driven Adaptive EMA Filter Strategy

5.1. Overview and Objective

Overview:
This strategy implements an adaptive Exponential Moving Average (EMA) where the smoothing period is dynamically adjusted based on the **Range Action Verification Index (RAVI)**. The RAVI, developed by Tushar Chande, measures the percentage difference between a short-term and a long-term moving average to identify whether the market is in a trending or a ranging phase.

Objective:
The primary objective is to adapt the responsiveness of the EMA filter to current market conditions as indicated by RAVI:

- When RAVI is **low** (suggesting a ranging or consolidating market), the strategy uses a **longer EMA period** for more smoothing.
- When RAVI is **high** (suggesting a trending market), the strategy uses a **shorter EMA period** to make the filter more responsive.
  Trading signals are generated based on crossovers of the closing price with this RAVI-driven adaptive EMA. The strategy includes an ATR-based trailing stop-loss and accounts for trading commissions.

5.2. Key Concepts: Range Action Verification Index (RAVI)

The **Range Action Verification Index (RAVI)** is calculated as the absolute percentage difference between a short-term Simple Moving Average (SMA) and a long-term SMA of the price.

$$RAVI = \frac{|SMA_{short} - SMA_{long}|}{SMA_{long}} \times 100$$

- **Low RAVI values** (typically below a certain threshold like 0.5% to 1%) suggest that the market is in a trading range or consolidation phase, as the short-term and long-term averages are close together.
- **High RAVI values** (typically above a threshold like 3%) suggest that the market is trending, as the short-term average is diverging significantly from the long-term average.

This strategy uses these RAVI values to adjust the EMA filter's smoothing.

5.3. Key Indicators and Components

- **RAVI (`ravi_col_name`):** Calculated using short and long SMAs of the 'Close' price.

```
# Calculate RAVI
sma_short = df['Close'].rolling(window=ravi_short_ma_period).mean()
sma_long = df['Close'].rolling(window=ravi_long_ma_period).mean()
df[ravi_col_name] = (np.abs(sma_short - sma_long) /
sma_long.replace(0,np.nan)) * 100
df[ravi_col_name].fillna(method='bfill', inplace=True)
```

- **Normalized RAVI (`normalized_ravi_col`):** RAVI values are clipped within a defined percentage range (`ravi_map_min_pct` , `ravi_map_max_pct`) and then normalized to a 0-1 scale.

```
ravi_clipped = df[ravi_col_name].clip(ravi_map_min_pct,
ravi_map_max_pct)
df[normalized_ravi_col] = (ravi_clipped - ravi_map_min_pct) /
```

```
(ravi_map_max_pct - ravi_map_min_pct + 1e-9) # Add epsilon
df[normalized_ravi_col].fillna(0, inplace=True)
df[normalized_ravi_col] = np.clip(df[normalized_ravi_col], 0, 1)
```

- **Adaptive EMA Period (`adaptive_ema_period_col`):** The period for the adaptive EMA, interpolated between `period_filter_min` (for high RAVI) and `period_filter_max` (for low RAVI) based on the *previous day's* normalized RAVI.

```
# High Normalized RAVI (strong trend) -> period_filter_min (faster EMA)
# Low Normalized RAVI (weak trend) -> period_filter_max (slower EMA)
df[adaptive_ema_period_col] = period_filter_max -
df[normalized_ravi_col].shift(1) * (period_filter_max -
period_filter_min)
df[adaptive_ema_period_col] =
np.round(df[adaptive_ema_period_col]).fillna( (period_filter_min +
period_filter_max) / 2 ).astype(int)
df[adaptive_ema_period_col] = np.clip(df[adaptive_ema_period_col],
period_filter_min, period_filter_max)
```

- **RAVI-Driven Adaptive EMA (`filtered_price_ravi_col`):** This is the adaptive filter, calculated iteratively. Its smoothing factor `Alpha_Adaptive_RAVI` changes daily based on `adaptive_ema_period_col` .

```
df['Alpha_Adaptive_RAVI'] = 2 / (df[adaptive_ema_period_col] + 1)
df[filtered_price_ravi_col] = np.nan
# ... (Seeding the first value) ...
first_valid_alpha_idx = df['Alpha_Adaptive_RAVI'].first_valid_index()
df.loc[first_valid_alpha_idx, filtered_price_ravi_col] =
df.loc[first_valid_alpha_idx, 'Close'] # Seed

for i_loop in range(start_loc_for_ema_loop + 1, len(df)):
    # ... (EMA calculation as in prior adaptive EMA script, using
'Alpha_Adaptive_RAVI') ...
        df.loc[idx_today, filtered_price_ravi_col] = alpha_val *
current_close_val + (1 - alpha_val) * prev_filtered_price_val
# ... (fillna logic for filtered_price_ravi_col) ...
```

- **Average True Range (ATR - `atr_col_name_sl_val`):** Used for the trailing stop-loss.

```
df['H-L_sl'] = df['High'] - df['Low']; df['H-PC_sl'] = (df['High'] -
df['Close'].shift(1)).abs(); #...
df['TR_sl'] = df[['H-L_sl','H-PC_sl','L-PC_sl']].max(axis=1);
df[atr_col_name_sl_val] =
df['TR_sl'].rolling(window=atr_window_sl).mean()
```

5.4. Script Parameters (Function Arguments)

The strategy is encapsulated in `run_ravi_adaptive_ema_backtest`, which takes these main arguments:

```
def run_ravi_adaptive_ema_backtest(
    ticker="BTC-USD",
    start_date_str="2023-01-01",
    end_date_str="2025-12-31",
    ravi_short_ma_period=7,     # Short MA for RAVI
    ravi_long_ma_period=65,     # Long MA for RAVI
    ravi_map_min_pct=0.5,       # RAVI at or below this -> period_filter_max
    ravi_map_max_pct=3.0,       # RAVI at or above this -> period_filter_min
    period_filter_min=10,       # Faster EMA for high RAVI
    period_filter_max=100,      # Slower EMA for low RAVI
    atr_window_sl=14,
    atr_multiplier_sl=2.0,
    commission_bps_per_side=1.0,
    verbose=True
):
# ... (parameter validation) ...
```

5.5. Data Handling

- **Data Download:** Daily OHLCV data is fetched via `yfinance`, with standard processing.

  ```
  # --- 1. Download Data ---
  df_raw = yf.download([ticker], start=start_date_str, end=end_date_str,
  auto_adjust=False, progress=False)
  # ... (error handling, droplevel, column selection) ...
  ```

5.6. Trading Logic (within `run_ravi_adaptive_ema_backtest`)

Trades are based on the *previous day's* close versus the RAVI-adaptive EMA, executed at the *current day's* `Open`.

5.6.1. Signal Generation Conditions

A target position (`target_pos`) is determined by comparing `prev_close` to `prev_filtered_price` (the RAVI-driven EMA):

```
# (Inside backtesting loop, after stop-loss check)
    target_pos = 0
    if prev_close > prev_filtered_price: target_pos = 1  # Signal Long
    elif prev_close < prev_filtered_price: target_pos = -1 # Signal Short
```

5.6.2. Entry Conditions

If `target_pos` implies a new trade or a flip, entry occurs at `today_open`. Commissions are applied.

```
# (If target_pos != 0 and target_pos != current_pos)
            if current_pos != 0 and pd.notna(current_entry_gross): # It's a
flip
                # ... (calculate pnl_exit_leg_net for the closing part of
the flip, including commission) ...

            current_pos = target_pos; current_entry_gross = today_open

            if current_pos == 1: # Entering New Long
                pnl_entry_leg_day = ((today_close / current_entry_gross) -
1) - comm_rate_val
                # ... (set init_ts and current_ts) ...
            elif current_pos == -1: # Entering New Short
                pnl_entry_leg_day = (-((today_close / current_entry_gross) -
1)) - comm_rate_val
                # ... (set init_ts and current_ts) ...
```

5.6.3. Exit Conditions

The main exit is via an ATR Trailing Stop-Loss. Positions also close if an opposite signal causes a flip.

- **ATR Trailing Stop-Loss Check:**

```
# (At the start of the daily loop logic for an active position)
if current_pos != 0 and pd.notna(current_ts) and
pd.notna(current_entry_gross):
    # ... (stop-loss hit logic as in prior adaptive EMA script) ...
    if stopped_out:
        if current_pos == 1: pnl_for_day = (exit_price_sl_gross /
current_entry_gross) - 1 - comm_rate_val
        else: pnl_for_day = -((exit_price_sl_gross /
current_entry_gross) - 1) - comm_rate_val
        current_pos = 0; action_this_bar = True
```

- **Trailing Stop Adjustment:** Updated daily if holding.

```
# (If holding and not stopped or flipped)
elif current_pos != 0:
    # ... (stop adjustment logic as in prior adaptive EMA script) ...
```

5.6.4. Position Sizing, P&L, and Commission Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **Commissions (`comm_rate_val`):** Calculated as `commission_bps_per_side / 10000.0` and applied to entries and exits.

  ```
  # Example commission on stop-loss exit:
  # pnl_for_day = (exit_price_sl_gross / current_entry_gross) - 1 -
  comm_rate_val
  # Example commission on new entry P&L for the day:
  # pnl_entry_leg_day = ((today_close / current_entry_gross) - 1) -
  comm_rate_val
  ```

- **P&L Calculation (`Strategy_Daily_Return`):** Daily P&L reflecting entries, exits, holds, and commissions.

5.7. Performance Evaluation

The script calculates and stores metrics in a dictionary, and can print a summary.

- **Max Drawdown:** Uses the `calculate_max_drawdown` helper.
- **Metrics Calculation (`_calc_metrics_for_ravi_ema`):** Computes key metrics.

  ```
  # (Inside run_ravi_adaptive_ema_backtest)
  def _calc_metrics_for_ravi_ema(returns, name, tdpy, results_dict_ref):
      # ... (calculates Cum Ret, Ann Ret, Ann Vol, Sharpe, Max Drawdown)
  ...
      results_dict_ref[name] = { ... metrics ... }
  strategy_lbl = f"RAVI-AdaptiveEMA(RAVIs:
  {ravi_short_ma_period}/{ravi_long_ma_period},P:[{period_filter_min}-
  {period_filter_max}],Comm:{commission_bps_per_side}bps)"
  _calc_metrics_for_ravi_ema(strat_returns, strategy_lbl,
  TRADING_DAYS_PER_YEAR, metrics_results)
  # ... (benchmark metrics) ...
  ```

- **Metrics Summary Printing (`_print_metrics_summary`):** Formats and prints the calculated metrics.

  ```
  # (After metrics calculation loop)
  if verbose:
      print("--- 4. Performance Metrics ---")
      for strategy_name_key, metrics_val in metrics_results.items():
          if strategy_name_key == 'num_trades' or strategy_name_key ==
  'parameters': continue
          _print_metrics_summary(metrics_val, strategy_name_key)
  ```

5.8. Plotting Results (via `plot_ravi_adaptive_ema_results` function)

A separate function generates six plots:

1. **Price & RAVI-Driven Adaptive EMA:** With Trailing Stops.
2. **Range Action Verification Index (RAVI) & Normalized Value:** RAVI with its mapping thresholds, and normalized RAVI on a twin axis.
3. **Adaptive EMA Period:** Shows the dynamically changing EMA period.
4. **Strategy Position:** Long/Short/Flat positions.
5. **ATR for Stop Loss.**
6. **Cumulative Performance (Log Scale):** Strategy vs. Buy & Hold.

```
def plot_ravi_adaptive_ema_results(df_analysis, strategy_params):
    # ... (plotting logic for 6 distinct figures) ...
    plt.show() # Called after each figure configuration in the script
```

5.9. Unique Features & Notes

- **RAVI-Driven Adaptation:** Uses the Range Action Verification Index (RAVI) to gauge market state (trending vs. ranging) and adapt the EMA filter's smoothing period accordingly.
- **Opposite Adaptation to Volatility Percentile:** Unlike the Savitzky-Golay example where high vol percentile led to longer windows (more smoothing), here high RAVI (trending) leads to *shorter* EMA periods (less smoothing, more responsiveness), and low RAVI (ranging) leads to *longer* EMA periods.
- **Iterative EMA:** The adaptive EMA is calculated iteratively due to the daily changes in its period.
- **Commission Modeling:** Explicitly includes commissions in P&L calculations.
- **Modular Structure:** The main backtesting logic, parameter definitions, and plotting are well-encapsulated in functions.
- **Daily Timeframe:** Operates on daily data.

---

6. Kalman-Noise-Adaptive Trend Filter Strategy

6.1. Overview and Objective

Overview:
This strategy employs a **Kalman filter** to estimate the underlying "true" price of an asset as a latent state. The Kalman filter is a recursive algorithm that uses a series of measurements observed over time (containing noise and other inaccuracies) and produces estimates of

unknown variables that tend to be more precise than those based on a single measurement alone. A key feature of this strategy is the **adaptive tuning** of the filter's process noise covariance (Q) and measurement noise covariance (R) based on observed market volatility.

Objective:
The primary objective is to generate a smoothed, adaptive trend line (the Kalman filtered price) and trade crossovers of the closing price with this filter.

- The **process noise (Q)** is adapted using rolling realized volatility, allowing the filter to expect more state variance when volatility is high.
- The **measurement noise (R)** can be adaptive (based on the variance of price around a short EMA) or fixed, influencing how much the filter trusts the current observed price. The strategy aims to capture trends by going long when the price crosses above the Kalman filter and short when it crosses below, with positions managed by an ATR-based trailing stop-loss and accounting for commissions.

6.2. Key Concepts: Kalman Filter

A Kalman filter is an optimal recursive data processing algorithm. It works in a two-step process:

1. **Predict:** The filter predicts the current state variables along with their uncertainties.
2. **Update:** The filter updates these predictions using a weighted average, with more weight being given to estimates with higher certainty. The weights, known as the Kalman Gain, are dynamically calculated at each step to minimize the estimated error covariance.

Key components in a simple (univariate price tracking) Kalman filter:

- **State (x̂):** The estimated true price. Assumed to follow a random walk in this model ($x_k = x_{k-1} + w_k$, where w_k is process noise).
- **Error Covariance (P):** A measure of the estimated accuracy of the state estimate.
- **Process Noise Covariance (Q):** Represents the uncertainty or noise in the process model itself (i.e., how much the true price can change from one step to the next, independent of measurement). This strategy adapts Q.
- **Measurement (z):** The observed price (e.g., closing price). Assumed to be $z_k = x_k + v_k$, where v_k is measurement noise.
- **Measurement Noise Covariance (R):** Represents the uncertainty or noise in the measurement. This strategy can adapt R or use a fixed value.
- **Kalman Gain (K):** Determines how much the prediction is corrected by the current measurement.

6.3. Key Indicators and Components

- **Realized Volatility (`realized_vol_col`):** Standard deviation of daily returns, used to adapt Q.

```
df[daily_return_col] = df['Close'].pct_change()
df[realized_vol_col] =
df[daily_return_col].rolling(window=realized_vol_window_for_q).std()
```

- **Adaptive Process Noise Covariance (Q - `q_adaptive_col`):** Scaled by the square of the *previous day's* realized volatility.

```
df[q_adaptive_col] = q_scale_factor * (df[realized_vol_col].shift(1) **
2)
df[q_adaptive_col].fillna(method='bfill', inplace=True)
df[q_adaptive_col].replace(0, 1e-9, inplace=True) # Ensure Q is not zero
df[q_adaptive_col].fillna(df[q_adaptive_col].mean() if not
df[q_adaptive_col].empty else 1e-7, inplace=True)
```

- **Adaptive Measurement Noise Covariance (R - `r_adaptive_col`):** Can be adaptive or fixed. If adaptive, it's based on the rolling mean of squared differences between 'Close' and a short EMA of 'Close', lagged.

```
if r_fixed_value_fallback is not None:
    df[r_adaptive_col] = r_fixed_value_fallback
else:
    short_ema_for_noise = df['Close'].ewm(span=r_ema_period_for_noise,
adjust=False).mean()
    measurement_noise_raw_sq = (df['Close'] - short_ema_for_noise)**2
    df[r_adaptive_col] =
measurement_noise_raw_sq.rolling(window=r_calc_window).mean().shift(1)
    df[r_adaptive_col].fillna(method='bfill', inplace=True)
    df[r_adaptive_col].replace(0, 1e-9, inplace=True)
    df[r_adaptive_col].fillna(df[r_adaptive_col].mean() if not
df[r_adaptive_col].empty else 1e-5, inplace=True)
```

- **Kalman Filtered Price (`kalman_filter_col`):** Calculated iteratively using the Kalman filter equations.

```
# Kalman Filter Calculation Loop
df[kalman_filter_col] = np.nan
x_hat = df['Close'].iloc[0] if not df.empty else 0 # Initial state
estimate
P = df[daily_return_col].var() if not df[daily_return_col].empty and
pd.notna(df[daily_return_col].var()) else 1.0 # Initial error covariance
# ... (logic to find start_kalman_loop_idx and seed first Kalman value)
...

for i_kf in range(start_kalman_loop_idx, len(df)):
    idx = df.index[i_kf]
    Q_current = df.loc[idx, q_adaptive_col]
```

```
        R_current = df.loc[idx, r_adaptive_col]
        z_measured = df.loc[idx, 'Close']
        # ... (NaN checks) ...

        # Predict
        x_hat_predict = x_hat  # State is random walk: x_t = x_{t-1} + w
        P_predict = P + Q_current

        # Update
        innovation = z_measured - x_hat_predict
        S = P_predict + R_current
        if S < 1e-9 : K = 0.0 # Avoid division by zero
        else: K = P_predict / S # Kalman Gain

        x_hat = x_hat_predict + K * innovation
        P = (1 - K) * P_predict

        df.loc[idx, kalman_filter_col] = x_hat
    df[kalman_filter_col].fillna(method='ffill', inplace=True)
```

- **Average True Range (ATR - `atr_col_name_sl_val`):** For the trailing stop-loss.

```
df['H-L_sl'] = df['High'] - df['Low']; df['H-PC_sl'] = (df['High'] -
df['Close'].shift(1)).abs(); #...
df['TR_sl'] = df[['H-L_sl','H-PC_sl','L-PC_sl']].max(axis=1);
df[atr_col_name_sl_val] =
df['TR_sl'].rolling(window=atr_window_sl).mean()
```

6.4. Script Parameters (Function Arguments)

The core logic is in `run_kalman_adaptive_noise_backtest`, accepting these parameters:

```
def run_kalman_adaptive_noise_backtest(
    ticker,
    start_date_str,
    end_date_str,
    realized_vol_window_for_q, # Window for realized vol driving Q
    q_scale_factor,            # Factor to scale squared realized vol for Q
    r_calc_window,             # Window for rolling variance for R (if
adaptive)
    r_ema_period_for_noise,    # Short EMA period for measurement noise base
(if adaptive)
    r_fixed_value_fallback,    # Optional fixed R, if None R is adaptive
    atr_window_sl,
    atr_multiplier_sl,
    commission_bps_per_side,
    plot_results,
    verbose
):
# ...
```

6.5. Data Handling

- **Data Download:** Standard daily OHLCV data download.

```
# --- 1. Download Data ---
df_raw = yf.download([ticker], start=start_date_str, end=end_date_str,
auto_adjust=False, progress=False)
# ... (error handling, droplevel, column selection) ...
```

6.6. Trading Logic (within `run_kalman_adaptive_noise_backtest`)

Trades are based on the *previous day's* close vs. the Kalman filtered price, executed at the *current day's* `Open`.

6.6.1. Kalman Filter State Estimation

The Kalman filter iteratively estimates the price (`x_hat`) as shown in section 6.3.

6.6.2. Signal Generation Conditions

A target position (`target_pos`) is determined by comparing `prev_close` to `prev_filtered_price` (the Kalman filtered price):

```
# (Inside backtesting loop, after stop-loss check)
    target_pos = 0
    if prev_close > prev_filtered_price: target_pos = 1  # Signal Long
    elif prev_close < prev_filtered_price: target_pos = -1 # Signal Short
```

6.6.3. Entry Conditions

If `target_pos` indicates a new trade or a flip, entry occurs at `today_open`. Commissions are applied.

```
# (If target_pos != 0 and target_pos != current_pos)
        if current_pos != 0 and pd.notna(current_entry_gross): # It's a
flip
                # ... (calculate pnl_exit_leg_net for the closing part of
the flip, including commission) ...

        current_pos = target_pos; current_entry_gross = today_open
```

```
            if current_pos == 1: # Entering New Long
                pnl_entry_leg_day = ((today_close / current_entry_gross) -
1) - comm_rate_val
                # ... (set init_ts and current_ts) ...
            elif current_pos == -1: # Entering New Short
                pnl_entry_leg_day = (-((today_close / current_entry_gross) -
1)) - comm_rate_val
                # ... (set init_ts and current_ts) ...
```

6.6.4. Exit Conditions

Primarily via an ATR Trailing Stop-Loss. Positions also close if an opposite signal causes a flip.

- **ATR Trailing Stop-Loss Check:** (Logic identical to previous adaptive filter scripts)
- **Trailing Stop Adjustment:** (Logic identical to previous adaptive filter scripts)

6.6.5. Position Sizing, P&L, and Commission Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **Commissions (`comm_rate_val`):** Applied as `commission_bps_per_side / 10000.0` to entry and exit legs.
- **P&L Calculation (`Strategy_Daily_Return`):** Daily P&L reflecting entries, exits, holds, and commissions.

6.7. Performance Evaluation

Metrics are calculated and stored, with options for printing a summary.

- **Max Drawdown:** Uses the `calculate_max_drawdown` helper.
- **Metrics Calculation (`_calc_metrics_for_kalman`):** Computes key metrics.

```
# (Inside run_kalman_adaptive_noise_backtest)
def _calc_metrics_for_kalman(returns, name, tdpy, results_dict_ref): #
Unique name for this script's metrics calc
    # ... (calculates Cum Ret, Ann Ret, Ann Vol, Sharpe, Max Drawdown)
...
    results_dict_ref[name] = { ... metrics ... }
strategy_lbl = f"KalmanAdaptNoise(QvolW:{realized_vol_window_for_q},QSF:
{q_scale_factor},RnoiseW:{r_calc_window},Comm:
{commission_bps_per_side}bps)"
_calc_metrics_for_kalman(strat_returns, strategy_lbl,
TRADING_DAYS_PER_YEAR, metrics_results)
# ... (benchmark metrics) ...
```

- **Metrics Summary Printing (`_print_metrics_summary`):** Formats and prints.

6.8. Plotting Results (via `plot_kalman_adaptive_noise_results` function)

A separate function generates six plots:

1. **Price & Kalman Adaptive Noise Filter:** With Trailing Stops.
2. **Adaptive Process Noise (Q) & Realized Volatility:** Shows Q and the realized volatility that drives its adaptation.
3. **Adaptive Measurement Noise (R):** If R is adaptive.
4. **Strategy Position.**
5. **ATR for Stop Loss.**
6. **Cumulative Performance (Log Scale).**

```
def plot_kalman_adaptive_noise_results(df_analysis, strategy_params):
    # ... (plotting logic for 6 distinct figures) ...
    plt.show() # Called after each figure configuration in the script
```

6.9. Unique Features & Notes

- **Kalman Filter Core:** Uses a Kalman filter for price tracking, which is a sophisticated method for signal extraction.
- **Adaptive Q and R:** The process noise (Q) is adapted based on realized volatility, and measurement noise (R) can also be adaptive or fixed. This allows the filter to adjust its confidence in its own predictions versus new measurements based on market conditions.
- **State Estimation Approach:** Models price as a latent state to be estimated, differing from direct smoothing of prices with traditional MAs or SG filters.
- **Iterative Calculation:** The Kalman filter is inherently iterative.
- **Parameter Sensitivity:** Kalman filter parameters, especially `q_scale_factor` and how R is determined, can be very sensitive and require careful tuning.
- **Commission Modeling:** Explicitly included.
- **Daily Timeframe:** Designed for daily data.

---

7. Spectral-Slope Adaptive Filter Strategy

7.1. Overview and Objective

Overview:

This strategy employs spectral analysis to adapt the smoothing period of an Exponential

Moving Average (EMA). It calculates the slope of the log-log Power Spectral Density (PSD) of the price series over a rolling window. This "spectral slope" provides an indication of the underlying characteristics of the price movements (e.g., trending, noisy, or mean-reverting behavior). The calculated slope is then used to dynamically adjust the EMA's period.

Objective:
The primary objective is to create an adaptive EMA filter that changes its responsiveness based on the spectral characteristics of the price data:

- When the spectral slope indicates a **stronger trend** (typically a steeper negative slope), a **shorter EMA period** is used to make the filter more reactive.
- When the spectral slope suggests **more noise or weaker trends** (a flatter slope, closer to zero), a **longer EMA period** is used for increased smoothing.
  Trading signals are generated from crossovers of the closing price with this spectrally-adapted EMA. The strategy includes an ATR-based trailing stop-loss and accounts for commissions.

7.2. Key Concepts: Price Spectrum & Spectral Slope

- **Price Spectrum (Power Spectral Density - PSD):** The PSD of a time series describes how the power (or variance) of the signal is distributed over different frequencies. This strategy uses **Welch's method** (`scipy.signal.welch`) to estimate the PSD of a rolling window of detrended price data.
- **Log-Log Spectral Slope:** When the logarithm of the PSD is plotted against the logarithm of frequency, the slope of this line can characterize the time series.
  For example, a pure random walk (Brownian motion) often exhibits a spectral slope of around -2 (PSD $\propto 1/f^2$).
  Slopes steeper than -2 (e.g., -2.5, -3) can indicate more persistent, trending behavior (power concentrated at lower frequencies).
  * Slopes flatter than -2 (e.g., -1, -0.5) can indicate more anti-persistent or noisy behavior.
  This strategy uses this calculated slope to adapt its filter.

7.3. Key Indicators and Components

- **Spectral Slope (`spectral_slope_col_name`):** Calculated on rolling windows of closing prices using the `_calculate_spectral_slope` helper function. This function detrends the price segment, applies Welch's method for PSD estimation, and then performs a linear regression on the log-log PSD vs. frequency plot to find the slope.

```
def _calculate_spectral_slope(price_segment_values):
    if len(price_segment_values) < spectrum_nperseg / 2 or
np.std(price_segment_values) < 1e-9 :
        return np.nan
    try:
        segment_detrended = signal.detrend(price_segment_values)
        if np.std(segment_detrended) < 1e-9: return np.nan
        freqs, psd = signal.welch(segment_detrended, fs=1.0,
nperseg=min(len(segment_detrended), spectrum_nperseg),
scaling='density',nfft=max(spectrum_nperseg, len(segment_detrended)))

        valid_indices = np.where((freqs > 1e-6) & (psd > 1e-9))[0]
        if len(valid_indices) < 2: return np.nan
        log_freqs, log_psd = np.log10(freqs[valid_indices]),
np.log10(psd[valid_indices])
        if np.std(log_freqs) < 1e-6 or np.std(log_psd) < 1e-6: return
np.nan
        slope, _, _, _, _ = stats.linregress(log_freqs, log_psd)
        return slope
    except (ValueError, FloatingPointError): return np.nan

df[spectral_slope_col_name] =
df['Close'].rolling(window=spectrum_window).apply(_calculate_spectral_sl
ope, raw=True)
df[spectral_slope_col_name].fillna(method='ffill', inplace=True)
df[spectral_slope_col_name].fillna(-2.0, inplace=True) # Fallback to
Brownian slope
```

- **Adaptive EMA Period (`adaptive_ema_period_col_name`):** The EMA period is interpolated between `period_filter_min` and `period_filter_max` based on the *previous day's* normalized spectral slope. Steeper negative slopes (more trending) map to `period_filter_min`, while flatter slopes (noisier) map to `period_filter_max`.

```
clipped_slope = df[spectral_slope_col_name].clip(slope_map_trend,
slope_map_noise) # slope_map_trend is more negative
norm_slope = (clipped_slope - slope_map_trend) / (slope_map_noise -
slope_map_trend + 1e-9)
df[adaptive_ema_period_col_name] = period_filter_min + norm_slope *
(period_filter_max - period_filter_min)
df[adaptive_ema_period_col_name] =
np.round(df[adaptive_ema_period_col_name].shift(1)).fillna((period_filte
r_min + period_filter_max) / 2).astype(int)
df[adaptive_ema_period_col_name] =
np.clip(df[adaptive_ema_period_col_name], period_filter_min,
period_filter_max)
```

- **Spectral-Slope Adaptive EMA (`filtered_price_spectral_col_name`):** The EMA calculated iteratively with its period (and thus alpha) changing daily based on `adaptive_ema_period_col_name`.

```
df['Alpha_Adaptive_Spectral'] = 2 / (df[adaptive_ema_period_col_name] +
1)
df[filtered_price_spectral_col_name] = np.nan
```

```
# ... (Seeding the first value) ...
first_valid_alpha_idx =
df['Alpha_Adaptive_Spectral'].first_valid_index()
df.loc[first_valid_alpha_idx, filtered_price_spectral_col_name] =
df.loc[first_valid_alpha_idx, 'Close']

for i_loop in range(start_loc_for_ema_loop + 1, len(df)):
    # ... (EMA calculation using 'Alpha_Adaptive_Spectral', as in prior
adaptive EMA scripts) ...
        df.loc[idx_today, filtered_price_spectral_col_name] = alpha_val
* current_close_val + (1 - alpha_val) * prev_filtered_price_val
# ... (fillna logic for filtered_price_spectral_col_name) ...
```

- **Average True Range (ATR - `atr_col_name_sl_val`):** For the trailing stop-loss.

```
df['H-L_sl'] = df['High'] - df['Low']; df['H-PC_sl'] = (df['High'] -
df['Close'].shift(1)).abs(); #...
df['TR_sl'] = df[['H-L_sl','H-PC_sl','L-PC_sl']].max(axis=1);
df[atr_col_name_sl_val] =
df['TR_sl'].rolling(window=atr_window_sl).mean()
```

7.4. Script Parameters (Function Arguments)

The core logic is within `run_spectral_slope_adaptive_filter_backtest`:

```
def run_spectral_slope_adaptive_filter_backtest(
    ticker="BTC-USD",
    start_date_str="2020-01-01",
    end_date_str="2024-12-31",
    spectrum_window=128,        # Rolling window for calculating price
spectrum
    spectrum_nperseg=64,        # Length of each segment for Welch's method
    slope_map_trend = -2.5,     # Slope indicating strongest trend (maps to
period_filter_min)
    slope_map_noise = -1.0,     # Slope indicating noisiest/weakest trend
(maps to period_filter_max)
    period_filter_min = 10,     # Shortest EMA period
    period_filter_max = 100,    # Longest EMA period
    atr_window_sl=14,
    atr_multiplier_sl=2.0,
    commission_bps_per_side=1.0,
    plot_results=True,
    verbose=True
):
# ...
```

7.5. Data Handling

- **Data Download:** Standard daily OHLCV data fetch using `yfinance`.

```
# --- 1. Download Data ---
df_raw = yf.download([ticker], start=start_date_str, end=end_date_str,
auto_adjust=False, progress=False)
# ... (error handling, droplevel, column selection) ...
```

7.6. Trading Logic (within `run_spectral_slope_adaptive_filter_backtest`)

Daily iteration. Signals from *previous day's* close vs. spectral-EMA; trades at *current day's* `Open`.

7.6.1. Spectral Slope Calculation

Performed by the `_calculate_spectral_slope` helper function within a rolling apply, as shown in section 7.3.

7.6.2. Adaptive EMA Period Calculation

Based on mapping the lagged spectral slope to the defined period range, as shown in section 7.3.

7.6.3. Iterative Adaptive EMA Calculation

Calculated daily using the adaptive period, as shown in section 7.3.

7.6.4. Signal Generation Conditions

Target position (`target_pos`) from `prev_close` vs. `prev_filtered_price` (the spectral-EMA):

```
# (Inside backtesting loop, after stop-loss check)
    target_pos = 0
    if prev_close > prev_filtered_price: target_pos = 1  # Signal Long
    elif prev_close < prev_filtered_price: target_pos = -1 # Signal Short
```

7.6.5. Entry Conditions

If `target_pos` suggests a new trade or a flip, entry is at `today_open`. Commissions applied.

```
# (If target_pos != 0 and target_pos != current_pos)
            if current_pos != 0 and pd.notna(current_entry_gross): # It's a
```

```
flip
                # ... (calculate pnl_exit_leg_net with commission) ...

            current_pos = target_pos; current_entry_gross = today_open

            if current_pos == 1: # Entering New Long
                pnl_entry_leg_day = ((today_close / current_entry_gross) -
1) - comm_rate_val
                # ... (set init_ts and current_ts) ...
            elif current_pos == -1: # Entering New Short
                pnl_entry_leg_day = (-((today_close / current_entry_gross) -
1)) - comm_rate_val
                # ... (set init_ts and current_ts) ...
```

7.6.6. Exit Conditions

Mainly via ATR Trailing Stop-Loss. Flips also cause exits.

- **ATR Trailing Stop-Loss Check:** (Logic identical to previous adaptive filter scripts, includes commission)
- **Trailing Stop Adjustment:** (Logic identical to previous adaptive filter scripts)

7.6.7. Position Sizing, P&L, and Commission Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **Commissions (`comm_rate_val`):** Applied as `commission_bps_per_side / 10000.0` to entries/exits.
- **P&L Calculation (`Strategy_Daily_Return`):** Daily P&L reflecting trades and commissions.

7.7. Performance Evaluation

Metrics are calculated and can be printed, stored in `metrics_results`.

- **Max Drawdown:** Uses the `calculate_max_drawdown` helper.
- **Metrics Calculation (`_calc_metrics_for_spectral`):** Computes key metrics.

  ```
  # (Inside run_spectral_slope_adaptive_filter_backtest)
  def _calc_metrics_for_spectral(returns, name, tdpy, results_dict_ref):
      # ... (calculates Cum Ret, Ann Ret, Ann Vol, Sharpe, Max Drawdown)
  ...
      results_dict_ref[name] = { ... metrics ... }
  strategy_lbl = f"SpectralSlopeEMA(SpecW:{spectrum_window},P:
  [{period_filter_min}-{period_filter_max}],Comm:
  {commission_bps_per_side}bps)"
  _calc_metrics_for_spectral(strat_returns, strategy_lbl,
  ```

```
    TRADING_DAYS_PER_YEAR, metrics_results)
    # ... (benchmark metrics) ...
```

- **Metrics Summary Printing (`_print_metrics_summary`):** Formats and prints.

7.8. Plotting Results (via `plot_spectral_slope_adaptive_filter_results` function)

A separate function generates six plots:

1. **Price & Spectral Adaptive EMA:** With Trailing Stops.
2. **Log-Log Spectral Slope (Rolling):** With mapping thresholds and a -2.0 (Brownian) reference.
3. **Adaptive EMA Period:** Shows the dynamically changing EMA period.
4. **Strategy Position.**
5. **ATR for Stop Loss.**
6. **Cumulative Performance (Log Scale).**

```
def plot_spectral_slope_adaptive_filter_results(df_analysis,
strategy_params):
    # ... (plotting logic for 6 distinct figures) ...
    plt.show() # Called after each figure configuration in the script
```

7.9. Unique Features & Notes

- **Spectral Analysis for Adaptation:** Uses the slope of the price spectrum (a sophisticated signal processing technique) to gauge market characteristics and adapt filter bandwidth.
- **Adaptive EMA Period:** The EMA period dynamically changes based on whether the market appears more trending (steeper negative spectral slope) or noisy (flatter spectral slope).
- `scipy.signal.welch` **&** `scipy.stats.linregress` **Dependency:** Requires `scipy` for these calculations.
- **Computational Cost:** Calculating spectral slope on a rolling basis can be computationally intensive.
- **Parameter Sensitivity:** Parameters related to spectrum calculation (`spectrum_window` , `spectrum_nperseg`) and slope mapping (`slope_map_trend` , `slope_map_noise`) are critical and likely require careful tuning.
- **Iterative EMA:** The adaptive EMA is calculated iteratively.
- **Commission Modeling:** Explicitly handled.
- **Daily Timeframe.**

8. Time-Decay Adaptive Exponential MA (TD-AEMA) Strategy

8.1. Overview and Objective

Overview:
This strategy implements an adaptive Exponential Moving Average (EMA) where its smoothing factor, traditionally denoted as alpha (α), is dynamically adjusted based on recent market volatility. The adaptation mechanism uses an exponential decay function: alpha decreases (leading to a slower EMA) as an Exponentially Weighted Moving Average (EWMA) of historical volatility increases.

Objective:
The primary goal is to create an EMA filter that becomes more conservative (smoother, longer effective period) during periods of higher volatility and more responsive (faster, shorter effective period) during lower volatility. The strategy aims to:

- Adapt the filter's smoothing factor $\alpha(t)$ using the formula: $\alpha(t) = \alpha_0 \times \exp(-\lambda \cdot \sigma_{t-1})$, where σ_{t-1} is the lagged EWMA of historical volatility, α_0 is a base (maximum) alpha, and λ is a decay parameter.
- Generate trading signals based on crossovers of the closing price with this Time-Decay Adaptive EMA (TD-AEMA).
- Manage risk using an ATR-based trailing stop-loss and account for trading commissions.

8.2. Key Concepts: Time-Decay Alpha Adaptation

The core of this adaptive filter is the dynamic adjustment of the EMA's smoothing factor, alpha (α). In a standard EMA, $\alpha = 2/(Period + 1)$. Here, α is not fixed but varies with volatility:

$$\alpha(t) = \alpha_0 \times \exp(-\lambda \cdot \sigma_{t-1})$$

Where:

- $\alpha(t)$: The adaptive alpha for the current day `t`.
- α_0: A base alpha, corresponding to the fastest EMA (shortest period, `period_ema_min_for_alpha0`), which occurs when volatility (σ) is zero.
- $\exp$: The exponential function.
- $-\lambda$: A negative decay parameter (`lambda_decay_param`). A larger positive λ means alpha decays more rapidly as volatility increases.
- σ_{t-1}: The EWMA of historical volatility, calculated up to the previous day (`ewma_vol_col.shift(1)`).

Effectively:

- As σ_{t-1} (EWMA volatility) **increases**, $\exp(-\lambda \cdot \sigma_{t-1})$ **decreases**, leading to a **smaller** $\alpha(t)$. A smaller alpha means a longer effective EMA period (more smoothing).
- As σ_{t-1} **decreases**, $\exp(-\lambda \cdot \sigma_{t-1})$ **increases** (approaching 1), leading to a **larger** $\alpha(t)$ (up to α_0). A larger alpha means a shorter effective EMA period (less smoothing, faster response).
  The calculated alpha is also capped to correspond to a maximum effective period.

8.3. Key Indicators and Components

- **Historical Volatility (`hist_vol_col`):** Rolling standard deviation of daily returns.

  ```
  df[daily_return_col] = df['Close'].pct_change()
  df[hist_vol_col] =
  df[daily_return_col].rolling(window=vol_calc_window).std()
  ```

- **EWMA of Historical Volatility (`ewma_vol_col`):** An exponentially weighted moving average of `hist_vol_col` , representing σ_t.

  ```
  df[ewma_vol_col] = df[hist_vol_col].ewm(span=vol_ema_period,
  adjust=False).mean()
  df[ewma_vol_col].fillna(method='bfill', inplace=True);
  df[ewma_vol_col].fillna(0, inplace=True)
  ```

- **Adaptive Alpha (`adaptive_alpha_col`):** Calculated daily using the time-decay formula based on the *previous day's* `ewma_vol_col` .

  ```
  alpha_0 = 2 / (period_ema_min_for_alpha0 + 1)
  df[adaptive_alpha_col] = alpha_0 * np.exp(-lambda_decay_param *
  df[ewma_vol_col].shift(1))

  # Cap alpha
  alpha_min_effective = 2 / (period_ema_max_cap + 1)
  df[adaptive_alpha_col] =
  df[adaptive_alpha_col].clip(lower=alpha_min_effective, upper=alpha_0)
  df[adaptive_alpha_col].fillna(alpha_0, inplace=True)
  ```

- **Effective EMA Period (`effective_period_col`):** The EMA period that the current `adaptive_alpha_col` corresponds to.

  ```
  df[effective_period_col] = (2 / df[adaptive_alpha_col]) - 1
  df[effective_period_col] =
  np.round(df[effective_period_col]).astype(int)
  ```

- **Time-Decay Adaptive EMA (TD-AEMA - `td_aema_filter_col`):** The filtered price series calculated iteratively using the `adaptive_alpha_col`.

```
df[td_aema_filter_col] = np.nan
# ... (Seeding the first value) ...
first_valid_alpha_idx = df[adaptive_alpha_col].first_valid_index()
df.loc[first_valid_alpha_idx, td_aema_filter_col] =
df.loc[first_valid_alpha_idx, 'Close']

for i_loop in range(start_loc_for_ema_loop + 1, len(df)):
    idx_today = df.index[i_loop]; idx_prev = df.index[i_loop-1]
    alpha_val = df.loc[idx_today, adaptive_alpha_col] # Alpha for
today's EMA
    current_close_val = df.loc[idx_today, 'Close']
    prev_filtered_price_val = df.loc[idx_prev, td_aema_filter_col]
    # ... (NaN checks) ...
    df.loc[idx_today, td_aema_filter_col] = alpha_val *
current_close_val + (1 - alpha_val) * prev_filtered_price_val
# ... (fillna logic for td_aema_filter_col) ...
```

- **Average True Range (ATR - `atr_col_name_sl_val`):** For the trailing stop-loss.

```
df['H-L_sl'] = df['High'] - df['Low']; df['H-PC_sl'] = (df['High'] -
df['Close'].shift(1)).abs(); #...
df['TR_sl'] = df[['H-L_sl','H-PC_sl','L-PC_sl']].max(axis=1);
df[atr_col_name_sl_val] =
df['TR_sl'].rolling(window=atr_window_sl).mean()
```

8.4. Script Parameters (Function Arguments)

The strategy is contained within `run_time_decay_adaptive_ema_backtest`:

```
def run_time_decay_adaptive_ema_backtest(
    ticker="SPY",
    start_date_str="2010-01-01",
    end_date_str="2024-12-31",
    vol_calc_window=20,          # Window for raw historical volatility
    vol_ema_period=10,           # Period for EWMA of historical volatility
    period_ema_min_for_alpha0=10,# Corresponds to alpha_0 (fastest EMA)
    lambda_decay_param=50.0,     # Sensitivity of alpha decay to volatility
    period_ema_max_cap=200,      # Max effective period (min alpha)
    atr_window_sl=14,
    atr_multiplier_sl=2.0,
    commission_bps_per_side=1.0,
    verbose=True
):
# ...
```

8.5. Data Handling

- **Data Download:** Standard daily OHLCV data fetch.

  ```
  # --- 1. Download Data ---
  df_raw = yf.download([ticker], start=start_date_str, end=end_date_str,
  auto_adjust=False, progress=False)
  # ... (error handling, droplevel, column selection) ...
  ```

8.6. Trading Logic (within `run_time_decay_adaptive_ema_backtest`)

Daily iteration. Signals from *previous day's* close vs. TD-AEMA; trades at *current day's* `Open`.

8.6.1. Adaptive Alpha Calculation

As detailed in section 8.3, `adaptive_alpha_col` is calculated based on lagged EWMA of volatility.

8.6.2. Iterative TD-AEMA Calculation

The `td_aema_filter_col` is computed iteratively using the daily `adaptive_alpha_col`, as shown in section 8.3.

8.6.3. Signal Generation Conditions

Target position (`target_pos`) from `prev_close` vs. `prev_filtered_price` (the TD-AEMA):

```
# (Inside backtesting loop, after stop-loss check)
    target_pos = 0
    if prev_close > prev_filtered_price: target_pos = 1  # Signal Long
    elif prev_close < prev_filtered_price: target_pos = -1 # Signal Short
```

8.6.4. Entry Conditions

If `target_pos` suggests a new trade or a flip, entry is at `today_open`. Commissions applied.

```
# (If target_pos != 0 and target_pos != current_pos)
            if current_pos != 0 and pd.notna(current_entry_gross): # It's a
flip
                # ... (calculate pnl_exit_leg_net with commission) ...
```

```
            current_pos = target_pos; current_entry_gross = today_open

            if current_pos == 1: # Entering New Long
                pnl_entry_leg_day = ((today_close / current_entry_gross) -
1) - comm_rate_val
                # ... (set init_ts and current_ts) ...
            elif current_pos == -1: # Entering New Short
                pnl_entry_leg_day = (-((today_close / current_entry_gross) -
1)) - comm_rate_val
                # ... (set init_ts and current_ts) ...
```

8.6.5. Exit Conditions

Mainly via ATR Trailing Stop-Loss. Flips also cause exits.

- **ATR Trailing Stop-Loss Check:** (Identical logic to previous adaptive filter scripts, includes commission)
- **Trailing Stop Adjustment:** (Identical logic to previous adaptive filter scripts)

8.6.6. Position Sizing, P&L, and Commission Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **Commissions (`comm_rate_val`):** Applied as `commission_bps_per_side / 10000.0`.
- **P&L Calculation (`Strategy_Daily_Return`):** Daily P&L reflecting trades and commissions.

8.7. Performance Evaluation

Standard metrics are calculated.

- **Max Drawdown:** Uses `calculate_max_drawdown` helper.
- **Metrics Calculation (`_calc_metrics_for_tdaema`):** Computes key performance indicators.

```
# (Inside run_time_decay_adaptive_ema_backtest)
def _calc_metrics_for_tdaema(returns, name, tdpy, results_dict_ref): #
Unique name
    # ... (standard calculations as in previous scripts) ...
    results_dict_ref[name] = { ... metrics ... }
strategy_lbl = f"TD-AEMA(V:{vol_calc_window},Ve:{vol_ema_period},P0:
{period_ema_min_for_alpha0},L:{lambda_decay_param},Pcap:
{period_ema_max_cap},Comm:{commission_bps_per_side}bps)"
_calc_metrics_for_tdaema(strat_returns, strategy_lbl,
TRADING_DAYS_PER_YEAR, metrics_results)
# ... (benchmark metrics) ...
```

- **Metrics Summary Printing (`_print_metrics_summary`):** Formats and prints results.

8.8. Plotting Results (via `plot_time_decay_adaptive_ema_results` function)

A separate function generates six plots:

1. **Price & Time-Decay Adaptive EMA:** With Trailing Stops.
2. **Historical Volatility and its EWMA (σ_t).**
3. **Adaptive Alpha & Resulting Effective EMA Period:** Shows the dynamic alpha and its corresponding EMA period.
4. **Strategy Position.**
5. **ATR for Stop Loss.**
6. **Cumulative Performance (Log Scale).**

```
def plot_time_decay_adaptive_ema_results(df_analysis, strategy_params):
    # ... (plotting logic for 6 distinct figures) ...
    plt.show() # Called after each figure configuration in the script
```

8.9. Unique Features & Notes

- **Exponential Decay Alpha:** The EMA's smoothing factor α is directly modulated by an exponential function of EWMA volatility. This provides a continuous adaptation mechanism.
- **Volatility Sensitivity Control (λ):** The `lambda_decay_param` offers explicit control over how sensitively the EMA's responsiveness changes with volatility.
- **Bounded Adaptation:** The effective EMA period is implicitly bounded by `period_ema_min_for_alpha0` (when volatility approaches zero) and explicitly by `period_ema_max_cap` (via alpha clipping).
- **Iterative EMA:** The TD-AEMA is calculated iteratively due to the daily changing alpha.
- **Commission Modeling:** Explicitly included.
- **Daily Timeframe.**

---

9. VIDYA (CMO-Adaptive EMA) Crossover Strategy

9.1. Overview and Objective

Overview:
VIDYA, or Variable Index Dynamic Average, is a type of adaptive moving average where the smoothing factor is dynamically adjusted based on a measure of market momentum or volatility. This specific implementation uses the **Chande Momentum Oscillator (CMO)** to

adapt the period of an Exponential Moving Average (EMA). The magnitude (absolute value) of the CMO determines the EMA's responsiveness.

Objective:
The primary goal is to create an EMA filter that adapts its smoothing based on the strength of price momentum:

- When momentum is **strong** (high absolute CMO value), a **shorter EMA period** is used, making the filter more sensitive and quicker to react to price changes.
- When momentum is **weak** (low absolute CMO value), a **longer EMA period** is used, resulting in a smoother filter that is less reactive to minor fluctuations.
  Trading signals are generated from crossovers of the closing price with this CMO-adapted EMA (VIDYA filter). The strategy includes an ATR-based trailing stop-loss and accounts for commissions.

9.2. Key Concepts: Chande Momentum Oscillator (CMO) & VIDYA

- **Chande Momentum Oscillator (CMO):** Developed by Tushar Chande, the CMO measures pure momentum. It is calculated as $(S_u - S_d)/(S_u + S_d) \times 100$, where S_u is the sum of upward price movements over a period, and S_d is the sum of downward price movements over the same period. CMO values range from -100 to +100. High absolute values (e.g., > 50 or < -50) indicate strong momentum, while values near 0 suggest weak momentum or a ranging market. This strategy uses the *absolute value* of CMO, scaled to a 0-1 range, to drive the adaptation.
- **VIDYA (Variable Index Dynamic Average):** In this implementation, VIDYA refers to an EMA whose smoothing period (and thus its alpha) is dynamically adjusted based on the normalized absolute CMO.

9.3. Key Indicators and Components

- **Chande Momentum Oscillator (CMO - `cmo_col_name`):** Calculated using `pandas_ta.momentum.cmo`.

```
# Chande Momentum Oscillator (CMO)
df[cmo_col_name] = ta.momentum.cmo(df['Close'], length=cmo_period)
df[cmo_col_name].fillna(0, inplace=True) # Fill initial CMO NaNs with 0
```

- **Normalized Absolute CMO (`norm_abs_cmo_col`):** The absolute value of CMO is taken and normalized to a 0-1 range (where 1 represents maximum momentum strength, typically |CMO|=100).

```
abs_cmo = df[cmo_col_name].abs()
df[norm_abs_cmo_col] = (abs_cmo / 100).clip(0, 1)
```

- **Adaptive VIDYA Period (`adaptive_vidya_period_col`):** The EMA period is interpolated between `period_vidya_min` (for high normalized absolute CMO) and `period_vidya_max` (for low normalized absolute CMO), based on the *previous day's* `norm_abs_cmo_col`.

```
# High NormAbsCMO (strong momentum) -> period_vidya_min (faster EMA)
# Low NormAbsCMO (weak momentum) -> period_vidya_max (slower EMA)
df[adaptive_vidya_period_col] = period_vidya_max -
df[norm_abs_cmo_col].shift(1) * (period_vidya_max - period_vidya_min)
df[adaptive_vidya_period_col] =
np.round(df[adaptive_vidya_period_col]).fillna( (period_vidya_min +
period_vidya_max) / 2 ).astype(int)
df[adaptive_vidya_period_col] = np.clip(df[adaptive_vidya_period_col],
period_vidya_min, period_vidya_max)
```

- **VIDYA Filter (`vidya_filter_col`):** The CMO-adaptive EMA, calculated iteratively. Its smoothing factor `Alpha_Adaptive_VIDYA` changes daily.

```
df['Alpha_Adaptive_VIDYA'] = 2 / (df[adaptive_vidya_period_col] + 1)
df[vidya_filter_col] = np.nan
# ... (Seeding the first value) ...
first_valid_alpha_idx = df['Alpha_Adaptive_VIDYA'].first_valid_index()
df.loc[first_valid_alpha_idx, vidya_filter_col] =
df.loc[first_valid_alpha_idx, 'Close']

for i_loop in range(start_loc_for_ema_loop + 1, len(df)):
    idx_today = df.index[i_loop]; idx_prev = df.index[i_loop-1]
    alpha_val = df.loc[idx_today, 'Alpha_Adaptive_VIDYA']
    current_close_val = df.loc[idx_today, 'Close']
    prev_filtered_price_val = df.loc[idx_prev, vidya_filter_col]
    # ... (NaN checks) ...
    df.loc[idx_today, vidya_filter_col] = alpha_val * current_close_val
+ (1 - alpha_val) * prev_filtered_price_val
# ... (fillna logic for vidya_filter_col) ...
```

- **Average True Range (ATR - `atr_col_name_sl_val`):** For the trailing stop-loss.

```
df['H-L_sl'] = df['High'] - df['Low']; df['H-PC_sl'] = (df['High'] -
df['Close'].shift(1)).abs(); #...
df['TR_sl'] = df[['H-L_sl','H-PC_sl','L-PC_sl']].max(axis=1);
```

```
df[atr_col_name_sl_val] =
df['TR_sl'].rolling(window=atr_window_sl).mean()
```

9.4. Script Parameters (Function Arguments)

The strategy is encapsulated in `run_vidya_crossover_backtest`:

```
def run_vidya_crossover_backtest(
    ticker="SPY",
    start_date_str="2010-01-01",
    end_date_str="2024-12-31",
    cmo_period=14,
    # Adaptive EMA period range driven by CMO magnitude
    period_vidya_min=10, # Faster EMA for high abs(CMO)
    period_vidya_max=60, # Slower EMA for low abs(CMO)
    atr_window_sl=14,
    atr_multiplier_sl=2.0,
    commission_bps_per_side=1.0,
    verbose=True
):
# ... (parameter validation) ...
```

9.5. Data Handling

- **Data Download:** Standard daily OHLCV data fetch using `yfinance`.

  ```
  # --- 1. Download Data ---
  df_raw = yf.download([ticker], start=start_date_str, end=end_date_str,
  auto_adjust=False, progress=False)
  # ... (error handling, droplevel, column selection) ...
  ```

9.6. Trading Logic (within `run_vidya_crossover_backtest`)

Daily iteration. Signals from *previous day's* close vs. VIDYA filter; trades at *current day's* `Open`.

9.6.1. Adaptive EMA Period Calculation (VIDYA logic)

The `adaptive_vidya_period_col` is determined based on the lagged normalized absolute CMO, as detailed in section 9.3.

9.6.2. Iterative VIDYA Filter Calculation

The `vidya_filter_col` is computed iteratively using the daily `Alpha_Adaptive_VIDYA`, as shown in section 9.3.

9.6.3. Signal Generation Conditions

Target position (`target_pos`) from `prev_close` vs. `prev_filtered_price` (the VIDYA filter):

```
# (Inside backtesting loop, after stop-loss check)
    target_pos = 0
    if prev_close > prev_filtered_price: target_pos = 1  # Signal Long
    elif prev_close < prev_filtered_price: target_pos = -1 # Signal Short
```

9.6.4. Entry Conditions

If `target_pos` suggests a new trade or a flip, entry is at `today_open` . Commissions applied.

```
# (If target_pos != 0 and target_pos != current_pos)
            if current_pos != 0 and pd.notna(current_entry_gross): # It's a
flip
                # ... (calculate pnl_exit_leg_net with commission) ...

            current_pos = target_pos; current_entry_gross = today_open

            if current_pos == 1: # Entering New Long
                pnl_entry_leg_day = ((today_close / current_entry_gross) -
1) - comm_rate_val
                # ... (set init_ts and current_ts) ...
            elif current_pos == -1: # Entering New Short
                pnl_entry_leg_day = (-((today_close / current_entry_gross) -
1)) - comm_rate_val
                # ... (set init_ts and current_ts) ...
```

9.6.5. Exit Conditions

Mainly via ATR Trailing Stop-Loss. Flips also cause exits.

- **ATR Trailing Stop-Loss Check:** (Identical logic to previous adaptive filter scripts, includes commission)
- **Trailing Stop Adjustment:** (Identical logic to previous adaptive filter scripts)

9.6.6. Position Sizing, P&L, and Commission Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **Commissions (`comm_rate_val`):** Applied as `commission_bps_per_side / 10000.0` .
- **P&L Calculation (`Strategy_Daily_Return`):** Daily P&L reflecting trades and commissions.

9.7. Performance Evaluation

Standard metrics are calculated and can be printed.

- **Max Drawdown:** Uses the `calculate_max_drawdown` helper.
- **Metrics Calculation (`_calc_metrics_for_vidya`):** Computes key performance indicators.

  ```
  # (Inside run_vidya_crossover_backtest)
  def _calc_metrics_for_vidya(returns, name, tdpy, results_dict_ref):
      # ... (standard calculations as in previous scripts) ...
      results_dict_ref[name] = { ... metrics ... }
  strategy_lbl = f"VIDYA(CMO:{cmo_period},P:[{period_vidya_min}-
  {period_vidya_max}],Comm:{commission_bps_per_side}bps)"
  _calc_metrics_for_vidya(strat_returns, strategy_lbl,
  TRADING_DAYS_PER_YEAR, metrics_results)
  # ... (benchmark metrics) ...
  ```

- **Metrics Summary Printing (`_print_metrics_summary`):** Formats and prints results.

9.8. Plotting Results (via `plot_vidya_results_separate_windows` function)

A separate function generates six plots:

1. **Price & VIDYA Filter:** With Trailing Stops.
2. **Chande Momentum Oscillator (CMO) & Normalized Absolute CMO:** CMO values and its normalized magnitude on a twin axis.
3. **Adaptive VIDYA Period:** Shows the dynamically changing EMA period.
4. **Strategy Position.**
5. **ATR for Stop Loss.**
6. **Cumulative Performance (Log Scale).**

```
def plot_vidya_results_separate_windows(df_analysis, strategy_params):
    # ... (plotting logic for 6 distinct figures) ...
    plt.show() # Called after each figure configuration in the script
```

9.9. Unique Features & Notes

- **Momentum-Driven Adaptation:** Uses the Chande Momentum Oscillator (CMO) magnitude to adapt the EMA's smoothing period, making the filter faster in strong momentum environments and slower in weak momentum ones.
- `pandas_ta` **Dependency:** Relies on the `pandas_ta` library for the CMO calculation.
- **Iterative EMA:** The VIDYA is an adaptive EMA calculated iteratively.
- **Commission Modeling:** Explicitly includes commissions.
- **Daily Timeframe.**

10. Volume-Weighted Adaptive EMA (VW-AEMA) Crossover Strategy

10.1. Overview and Objective

Overview:
This strategy implements an adaptive Exponential Moving Average (EMA) where the smoothing period is dynamically adjusted based on recent trading volume. The core idea is that high trading volume often accompanies significant price moves and might warrant a more responsive filter. Conversely, low volume might suggest a less convicted market, where a smoother filter is preferable.

Objective:
The primary objective is to create an EMA filter that adapts its responsiveness to volume conditions:

- When the **previous day's volume is higher** than its own EMA (Exponential Moving Average of volume), a **shorter EMA period** is used for the price filter, making it more sensitive to recent price action.
- When the **previous day's volume is lower** than or equal to its EMA, a **longer, base EMA period** is used, resulting in a smoother price filter.
  Trading signals are generated from crossovers of the closing price with this Volume-Weighted Adaptive EMA (VW-AEMA). The strategy includes an ATR-based trailing stop-loss and accounts for trading commissions.

10.2. Key Concepts: Volume-Based Adaptation

The "weighting" in this strategy comes from the selection of the EMA period based on volume activity. Instead of directly incorporating volume into the EMA formula (like a traditional VWMA), this strategy uses a binary condition (volume above/below its EMA) to switch between two predefined EMA periods for the price data. This effectively gives more weight to recent prices (by using a shorter EMA) during periods identified as "high volume."

10.3. Key Indicators and Components

- **Volume EMA (`vol_ema_col`):** An Exponential Moving Average of the daily trading volume.

  ```
  # Volume EMA
  df[vol_ema_col] = df['Volume'].ewm(span=volume_ema_period_param,
  adjust=False).mean()
  ```

- **Adaptive EMA Period (`adaptive_period_col`):** This period switches between `short_ema_period_param` and `base_ema_period_param` based on whether the *previous day's* volume was above its EMA.

  ```
  # Determine Adaptive EMA Period (based on lagged Volume condition)
  high_volume_condition = df['Volume'].shift(1) > df[vol_ema_col].shift(1)
  df[adaptive_period_col] = np.where(high_volume_condition,
  short_ema_period_param, base_ema_period_param)
  df[adaptive_period_col].fillna(base_ema_period_param, inplace=True) #
  Fill initial NaNs
  df[adaptive_period_col] = df[adaptive_period_col].astype(int)
  ```

- **Volume-Weighted Adaptive EMA (VW-AEMA Filter - `vw_aema_filter_col`):** The EMA of the closing price, calculated iteratively with its period (and thus alpha) changing daily based on `adaptive_period_col`.

  ```
  df['Alpha_Adaptive_VW'] = 2 / (df[adaptive_period_col] + 1) # Alpha uses
  current day's adaptive period
  df[vw_aema_filter_col] = np.nan
  # ... (Seeding the first value) ...
  first_valid_alpha_idx = df['Alpha_Adaptive_VW'].first_valid_index()
  df.loc[first_valid_alpha_idx, vw_aema_filter_col] =
  df.loc[first_valid_alpha_idx, 'Close']

  for i_loop in range(start_loc_for_ema_loop + 1, len(df)):
      idx_today = df.index[i_loop]; idx_prev = df.index[i_loop-1]
      alpha_val = df.loc[idx_today, 'Alpha_Adaptive_VW']
      current_close_val = df.loc[idx_today, 'Close']
      prev_filtered_price_val = df.loc[idx_prev, vw_aema_filter_col]
      # ... (NaN checks) ...
      df.loc[idx_today, vw_aema_filter_col] = alpha_val *
  current_close_val + (1 - alpha_val) * prev_filtered_price_val
  # ... (fillna logic for vw_aema_filter_col) ...
  ```

- **Average True Range (ATR - `atr_col_name_sl`):** For the trailing stop-loss.

  ```
  df['H-L_sl'] = df['High'] - df['Low']; df['H-PC_sl'] = (df['High'] -
  df['Close'].shift(1)).abs(); #...
  df['TR_sl'] = df[['H-L_sl','H-PC_sl','L-PC_sl']].max(axis=1);
  df[atr_col_name_sl] =
  df['TR_sl'].rolling(window=atr_window_sl_param).mean()
  ```

10.4. Script Parameters (Function Arguments)

The strategy is encapsulated in `run_vw_aema_backtest` :

```
def run_vw_aema_backtest(
    ticker="SPY",
    start_date_str="2010-01-01",
    end_date_str="2024-12-31",
    volume_ema_period_param=20,
    base_ema_period_param=50,
    short_ema_period_param=20, # Must be < base_ema_period_param
    atr_window_sl_param=14,
    atr_multiplier_sl_param=2.0,
    commission_bps_per_side_param=1.0,
    plot_results=True,
    verbose=True
):
# ... (parameter validation: short_ema_period_param < base_ema_period_param)
...
```

10.5. Data Handling

- **Data Download:** Standard daily OHLCV data fetch. Volume is crucial, so a check `df['Volume'] = df['Volume'].replace(0, 1)` is included to prevent division by zero or issues if volume is reported as zero.

  ```
  # --- 1. Download Data ---
  df_raw = yf.download([ticker], start=start_date_str, end=end_date_str,
  auto_adjust=False, progress=False)
  # ... (error handling, droplevel, column selection) ...
  df['Volume'] = df['Volume'].replace(0, 1) # Avoid issues with zero
  volume
  ```

10.6. Trading Logic (within `run_vw_aema_backtest`)

Daily iteration. Signals from *previous day's* close vs. VW-AEMA; trades at *current day's* `Open`.

10.6.1. Adaptive EMA Period Calculation

The `adaptive_period_col` is determined based on the lagged volume condition (volume vs. its EMA), as detailed in section 10.3.

10.6.2. Iterative VW-AEMA Filter Calculation

The `vw_aema_filter_col` is computed iteratively using the daily `Alpha_Adaptive_VW` (derived from `adaptive_period_col`), as shown in section 10.3.

10.6.3. Signal Generation Conditions

Target position (`target_pos`) from `prev_close` vs. `prev_filtered_price` (the VW-AEMA):

```
# (Inside backtesting loop, after stop-loss check)
    target_pos = 0
    if prev_close > prev_filtered_price: target_pos = 1  # Signal Long
    elif prev_close < prev_filtered_price: target_pos = -1 # Signal Short
```

10.6.4. Entry Conditions

If `target_pos` suggests a new trade or a flip, entry is at `today_open` . Commissions applied.

```
# (If target_pos != 0 and target_pos != current_pos)
            if current_pos != 0 and pd.notna(current_entry_gross): # It's a
flip
                # ... (calculate pnl_exit_leg_net with commission) ...

            current_pos = target_pos; current_entry_gross = today_open

            if current_pos == 1: # Entering New Long
                pnl_entry_leg_day = ((today_close / current_entry_gross) -
1) - comm_rate_val
                # ... (set init_ts and current_ts using
atr_multiplier_sl_param) ...
            elif current_pos == -1: # Entering New Short
                pnl_entry_leg_day = (-((today_close / current_entry_gross) -
1)) - comm_rate_val
                # ... (set init_ts and current_ts using
atr_multiplier_sl_param) ...
```

10.6.5. Exit Conditions

Mainly via ATR Trailing Stop-Loss. Flips also cause exits.

- **ATR Trailing Stop-Loss Check:** (Identical logic to previous adaptive filter scripts, includes commission)
- **Trailing Stop Adjustment:** (Identical logic to previous adaptive filter scripts, using `atr_multiplier_sl_param`)

10.6.6. Position Sizing, P&L, and Commission Calculation

- **Position Sizing:** Binary (1 for long, -1 for short, 0 for flat).
- **Commissions (`comm_rate_val`):** Applied as `commission_bps_per_side_param / 10000.0` .
- **P&L Calculation (`Strategy_Daily_Return`):** Daily P&L reflecting trades and commissions.

10.7. Performance Evaluation

Standard metrics are calculated.

- **Max Drawdown:** Uses the `calculate_max_drawdown` helper.
- **Metrics Calculation (`_calc_metrics_for_vw_aema`):** Computes key performance indicators.

```
# (Inside run_vw_aema_backtest)
def _calc_metrics_for_vw_aema(returns, name, tdpy, results_dict_ref): #
Unique name
    # ... (standard calculations as in previous scripts) ...
    results_dict_ref[name] = { ... metrics ... }
strategy_lbl = f"VW-AEMA(VolEMA:{volume_ema_period_param},P:
[{short_ema_period_param}/{base_ema_period_param}],Comm:
{commission_bps_per_side_param}bps)"
_calc_metrics_for_vw_aema(strat_returns, strategy_lbl,
TRADING_DAYS_PER_YEAR, metrics_results)
# ... (benchmark metrics) ...
```

- **Metrics Summary Printing (`_print_metrics` - note: name changed from _print_metrics_summary in this script):** Formats and prints results.

10.8. Plotting Results (via `plot_vw_aema_results` function)

A separate function generates six plots:

1. **Price & Volume-Weighted Adaptive EMA (VW-AEMA) Filter:** With Trailing Stops.
2. **Volume and its EMA.**
3. **Adaptive EMA Period:** Shows the switching between short and base periods.
4. **Strategy Position.**
5. **ATR for Stop Loss.**
6. **Cumulative Performance (Log Scale).**

```
def plot_vw_aema_results(df_analysis, strategy_params):
    # ... (plotting logic for 6 distinct figures, using corrected param
names for labels) ...
    plt.show() # Called after each figure configuration in the script
```

10.9. Unique Features & Notes

- **Volume-Driven Adaptation:** The EMA period is directly switched based on whether the previous day's volume was above its own EMA, making the filter more responsive during high-volume periods.

- **Binary Period Switch:** Unlike some other adaptive filters that interpolate periods, this one makes a binary choice between a short and a base period.
- **Iterative EMA:** The VW-AEMA is calculated iteratively due to the daily potential change in its underlying period.
- **Commission Modeling:** Explicitly included.
- **Daily Timeframe.**
