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

# Read CSV and return a DataFrame with ['REF_AREA','TIME_PERIOD','OBS_VALUE']
def read_cli_csv(csv_path):
    df_raw = pd.read_csv(csv_path, header=None, skipinitialspace=True)
    df = df_raw[[4, 22, 24]].copy()
    df.columns = ['REF_AREA','TIME_PERIOD','OBS_VALUE']
    df.dropna(subset=['TIME_PERIOD','OBS_VALUE'], inplace=True)
    df['TIME_PERIOD'] = pd.to_datetime(df['TIME_PERIOD'], format='%Y-%m', errors='coerce')
    df.dropna(subset=['TIME_PERIOD'], inplace=True)
    df['OBS_VALUE'] = pd.to_numeric(df['OBS_VALUE'], errors='coerce')
    df.dropna(subset=['OBS_VALUE'], inplace=True)
    df.sort_values(['REF_AREA','TIME_PERIOD'], inplace=True)
    df.reset_index(drop=True, inplace=True)
    return df

# Detect and interpolate outliers using a rolling Median Absolute Deviation
def remove_outliers_rolling_mad(series, window=12, threshold=3.5):
    s = series.copy()

    def rolling_mad(x):
        m = np.median(x)
        return np.median(np.abs(x - m))

    rolling_median = s.rolling(window=window, center=True).median()
    rolling_mad_series = s.rolling(window=window, center=True).apply(rolling_mad, raw=True)
    median_vals = rolling_median.values
    mad_vals = rolling_mad_series.values

    outlier_mask = np.zeros(len(s), dtype=bool)
    for i in range(len(s)):
        if not np.isnan(median_vals[i]) and not np.isnan(mad_vals[i]) and mad_vals[i] != 0:
            diff = abs(s.iloc[i] - median_vals[i])
            if diff > threshold * mad_vals[i]:
                outlier_mask[i] = True

    s[outlier_mask] = np.nan
    s = s.interpolate(method='linear', limit_direction='both')
    return s

# Single Bry-Boschan run to detect local peaks/troughs with constraints
def bry_boschan_once(series, min_phase=5, min_cycle=15):
    idx = series.index
    vals = series.values
    n = len(vals)

    potential = []
    for i in range(1, n - 1):
        if vals[i] > vals[i - 1] and vals[i] > vals[i + 1]:
            potential.append((i, 'peak'))
        elif vals[i] < vals[i - 1] and vals[i] < vals[i + 1]:
            potential.append((i, 'trough'))
    potential.sort(key=lambda x: x[0])

    refined = []
    for pt in potential:
        if not refined:
            refined.append(pt)
        else:
            pr_i, pr_tp = refined[-1]
            cu_i, cu_tp = pt
            if cu_tp == pr_tp:
                if pr_tp == 'peak':
                    if vals[cu_i] > vals[pr_i]:
                        refined[-1] = pt
```

Gestion des pics ou creux consécutifs

Faudrait pas mettre ça à la fin plutôt ?

```
                else:
                    if vals[cu_i] < vals[pr_i]:
                        refined[-1] = pt
            else:
                refined.append(pt)

    phase_f = []
    i = 0
    while i < len(refined):
        if not phase_f:
            phase_f.append(refined[i])
            i += 1
        else:
            p_i, p_tp = phase_f[-1]
            c_i, c_tp = refined[i]
            dist = c_i - p_i
            if dist < min_phase:
                if p_tp == 'peak':
                    if vals[c_i] > vals[p_i]:
                        phase_f[-1] = (c_i, c_tp)
                else:
                    if vals[c_i] < vals[p_i]:
                        phase_f[-1] = (c_i, c_tp)
                i += 1
            else:
                phase_f.append(refined[i])
                i += 1

    final = []
    j = 0
    while j < len(phase_f):
        if not final:
            final.append(phase_f[j])
            j += 1
        else:
            pr_i, pr_tp = final[-1]
            cu_i, cu_tp = phase_f[j]
            if cu_tp == pr_tp:
                dist2 = cu_i - pr_i
                if dist2 < min_cycle:
                    if pr_tp == 'peak':
                        if vals[cu_i] > vals[pr_i]:
                            final[-1] = (cu_i, cu_tp)
                    else:
                        if vals[cu_i] < vals[pr_i]:
                            final[-1] = (cu_i, cu_tp)
                    j += 1
                else:
                    final.append(phase_f[j])
                    j += 1
            else:
                final.append(phase_f[j])
                j += 1

    return [(idx[p], tp) for (p, tp) in final]

# Compute Phase Average Trend (PAT)
def compute_pat_trend(series, turning_points):
    n = len(series)
    idx = series.index
    vals = series.values

    if not turning_points:
        return pd.Series(np.full(n, np.mean(vals)), index=idx)

    turning_idx = [0]
```

```
    idx_map = {dt: i for i, dt in enumerate(idx)}
    for (dt, _) in turning_points:
        if dt in idx_map:
            turning_idx.append(idx_map[dt])
    if turning_idx[-1] != n - 1:
        turning_idx.append(n - 1)
    turning_idx.sort()

    phase_means = []
    for k in range(len(turning_idx) - 1):
        s_i = turning_idx[k]
        e_i = turning_idx[k + 1]
        seg = vals[s_i:e_i + 1]
        pm = np.mean(seg)
        phase_means.append(pm)

    smooth_pm = []
    if len(phase_means) < 3:
        smooth_pm = phase_means
    else:
        for i in range(len(phase_means)):
            if i == 0:
                val = (phase_means[0] + phase_means[1]) / 2.0
            elif i == len(phase_means) - 1:
                val = (phase_means[i] + phase_means[i - 1]) / 2.0
            else:
                val = (phase_means[i - 1] + phase_means[i] + phase_means[i + 1]) / 3.0
            smooth_pm.append(val)

    pat_trend = np.zeros(n)
    for k in range(len(turning_idx) - 1):
        s_i = turning_idx[k]
        e_i = turning_idx[k + 1]
        pat_trend[s_i:e_i + 1] = smooth_pm[k]

    return pd.Series(pat_trend, index=idx)

# Merge or remove very short cycles iteratively
def prune_short_cycles(tps, cycle_series, min_length=9, min_amplitude=0.5, max_passes=5):
    if not tps:
        return tps

    idx_map = {dt: i for i, dt in enumerate(cycle_series.index)}

    def single_pass(tps_in):
        if len(tps_in) < 2:
            return tps_in, False
        out = []
        changed = False
        i = 0
        while i < len(tps_in):
            if not out:
                out.append(tps_in[i])
                i += 1
            else:
                prev_t, prev_tp = out[-1]
                curr_t, curr_tp = tps_in[i]
                dist = (curr_t.year - prev_t.year)*12 + (curr_t.month - prev_t.month)

                amp_ok = True
                if min_amplitude is not None and prev_tp != curr_tp:
                    if (prev_t in idx_map) and (curr_t in idx_map):
                        i1 = idx_map[prev_t]
                        i2 = idx_map[curr_t]
                        c1 = cycle_series.iloc[i1]
                        c2 = cycle_series.iloc[i2]
```

```
                        amplitude = abs(c2 - c1)
                        if amplitude < min_amplitude:
                            amp_ok = False

                if dist < min_length or not amp_ok:
                    changed = True
                    i += 1
                else:
                    out.append(tps_in[i])
                    i += 1
        return out, changed

    current = tps[:]
    for _ in range(max_passes):
        new_tps, changed = single_pass(current)
        if not changed:
            return new_tps
        current = new_tps
    return current

# Iteratively compute turning points with PAT and prune short cycles
def detect_turning_points_pat(
    series,
    min_phase=5,
    min_cycle=15,
    max_iter=3,
    final_min_phase=9,
    final_min_amp=0.5,
    final_max_passes=5
):
    raw_vals = series.copy()
    old_tps = None
    final_tps = None

    for _ in range(max_iter):
        tps_raw = bry_boschan_once(raw_vals, min_phase=min_phase, min_cycle=min_cycle)
        pat_series = compute_pat_trend(raw_vals, tps_raw)
        cyc = raw_vals - pat_series
        tps_cyc = bry_boschan_once(cyc, min_phase=min_phase, min_cycle=min_cycle)

        if old_tps is not None:
            if len(tps_cyc) == len(old_tps):
                same_count = sum(1 for (a, b) in zip(tps_cyc, old_tps) if a == b)
                if same_count == len(tps_cyc):
                    final_tps = tps_cyc
                    break
        old_tps = tps_cyc
        final_tps = tps_cyc

    if not final_tps:
        final_tps = old_tps if old_tps else []

    cyc = raw_vals - compute_pat_trend(raw_vals, final_tps)
    final_tps = prune_short_cycles(
        tps=final_tps,
        cycle_series=cyc,
        min_length=final_min_phase,
        min_amplitude=final_min_amp,
        max_passes=final_max_passes
    )

    return final_tps

def main():
    csv_file = "CLI.csv"
    df_all = read_cli_csv(csv_file)
```

```
    if df_all.empty:
        print("No data or parse error.")
        return

    MIN_PHASE = 5
    MIN_CYCLE = 15
    MAX_ITER = 5
    MAD_WINDOW = 12
    MAD_THRESHOLD = 3.5
    FINAL_MIN_PHASE = 9
    FINAL_MIN_AMPLITUDE = 0.5
    FINAL_MAX_PASSES = 5

    country_list = df_all['REF_AREA'].dropna().unique()
    results = {}

    for c in country_list:
        sub = df_all[df_all['REF_AREA'] == c]
        if sub.empty:
            print(f"No data for {c}, skip.")
            continue

        s = sub.set_index('TIME_PERIOD')['OBS_VALUE'].sort_index()
        s_clean = remove_outliers_rolling_mad(s, window=MAD_WINDOW, threshold=MAD_THRESHOLD)

        tps_final = detect_turning_points_pat(
            series=s_clean,
            min_phase=MIN_PHASE,
            min_cycle=MIN_CYCLE,
            max_iter=MAX_ITER,
            final_min_phase=FINAL_MIN_PHASE,
            final_min_amp=FINAL_MIN_AMPLITUDE,
            final_max_passes=FINAL_MAX_PASSES
        )

        results[c] = tps_final
        print(f"\n--- {c} ---")
        if tps_final:
            for (dt, tp) in tps_final:
                print(dt.strftime('%Y-%m'), tp)
        else:
            print("No turning points found.")

if __name__ == "__main__":
    main()
```