専門家向け解説

User: Anonymous (kbmedphys@gmail.com)
Created: 11/10/2025 21:36:18
Updated: 11/10/2025 22:10:29
Exported: 11/10/2025 22:39:48
Link: https://chatgpt.com/c/6911dc41-47d4-8323-a8e2-0c0651c4fd21

Response:

以下は、添付論文 **Bai & Perron (2003), "Computation and Analysis of Multiple Structural Change Models"**（*Journal of Applied Econometrics*, Vol.18, pp.1–22）の専門家向け詳細解説です。
この論文は、1998年の理論論文（Bai and Perron, _Econometrica_）の実証実装編であり、**複数構造変化点（multiple structural breaks）を持つ線形回帰モデルの推定・検定方法の計算的枠組み**を確立した代表的研究です。

🔷 1. 背景と目的

構造変化（structural change）問題とは、時系列データにおいて**回帰パラメータが特定の時点で不連続に変化する**場合に、その変化点（break date）を特定することを目的とします。
従来は単一の変化点を想定した手法（Chow testなど）が主流でしたが、Bai & Perron (1998, 2003) は**複数変化点を同時に推定**できる一般的手法を理論・実装の両面から提示しました。

本稿では特に次の実務的課題を扱います：

1. **ブレーク日（break date）の推定アルゴリズムの開発**
   → 動的計画法（dynamic programming）を用いて**O(T^2)** の計算量でグローバル最小値を求める。
2. **信頼区間の構築**
   → 各種誤差構造（自己相関・分散不均一）に対応。
3. **構造変化の検定とブレーク数の推定**
   → Sup-Wald, Double Maximum, Sequential tests を包括的に整理。
4. **実証応用と実装**
   → GAUSSプログラムで利用可能な計算手順を示す。

🔷 2. モデルの定式化

一般形

$$y_t = x_t'\beta + z_t'\nu_j + u_t, \quad t = T_{j-1}+1, \ldots, T_j, \quad j = 1, \ldots, m+1$$

- m ：構造変化点の数（未知）
- x_t ：変化しない説明変数
- z_t ：係数がセグメントごとに変化する変数
- β ：固定パラメータ
- ν_j ：セグメント j に特有のパラメータ

これは**部分構造変化モデル（partial structural change model）**であり、**
$p = 0$ **の場合には全ての係数が変化する**純構造変化モデル（pure structural change model）**
になります。

🔷 3. 推定アルゴリズム：動的計画法による最適分割

複数のブレーク点を探索するには、全探索では O(T^m) の計算量となり現実的でありません。
Bai & Perron は、**動的計画法（Dynamic Programming, DP）**を応用し、O(T^2) の効率的アルゴリズムを提示しました。

手順概要

1. 各可能な区間 [i, j] に対し、OLS による **SSR(i,j)**（残差平方和）を計算。
 → これにより **T(T+1)/2** の「三角行列」を構築（Figure 1参照）。

2. DPを用い、次の再帰式で最適分割を求める：

$$SSR(\{T_m, T\}) = \min_{h \le j \le T-h} [SSR(\{T_{m-1}, j\}) + SSR(j+1, T)]$$

 → 各サブサンプルで最小SSRを保存し、順次更新。

3. m 個のブレークを持つ最小SSRパーティションを決定。
 → **グローバル最小値**が保証される。

計算特性

- 計算負荷の中心はSSR行列の構築（O(T^2)）
- m の増加に対して計算時間の増加はごくわずか

- 通常のサンプルサイズ（例：T=200–1000）では実用的に高速

🔷 4. 部分構造変化モデルへの拡張

β が共通パラメータの場合、分割が未知では β の推定も難しくなります。
そこで **Sargan (1964)** に基づく反復最適化法を導入します：

1. 初期値 $\beta^{(0)}$ を固定し、DPでブレーク点を推定。
2. 得られた区間に基づき、OLSで β, ν を同時更新。
3. 収束まで反復（通常は1〜2回で収束）。

初期値には「全係数変化モデル」を用いて推定した値を採用することで局所解を回避。

🔷 5. 信頼区間の構築

（1）係数推定値

ブレーク推定誤差は漸近的に無視可能（T倍速収束）であるため、
通常の OLS と同様に √T-漸近正規分布を持つ：

$$\sqrt{T}(\hat{\theta} - \theta_0) \xrightarrow{d} N(0, V^{-1}\Omega V^{-1})$$

ここで、Ωは系列相関・異分散を許す一般化分散。
HAC推定（Andrews, 1991）を利用。

（2）ブレーク日（Break Date）の分布

推定されたブレーク点 $\hat{T}_i$ は、真値 T_i^0 から定数オーダーで収束する：

$$(\hat{T}_i - T_i^0) \Rightarrow \arg\max_s V_i(s)$$

- $V_i(s)$ ：ブラウン運動（Weiner過程）を用いた確率過程
- 分布は変化幅（Δ_i）や分散構造に依存
- 対称的な二側Brownian motion型の分布として近似可能

これを用いて**整数区間での信頼区間**（例えば±k期間）を構築します。

🔷 6. 構造変化検定

(1) SupF 検定（no break vs k breaks）

$$F_T(\tau_1, ..., \tau_k) = \frac{(SSR_0 - SSR_k)/kq}{SSR_k/(T-(k+1)q)}$$

- 帰無仮説：構造変化なし
- 対立仮説：k個の変化あり
- 最小SSRを達成するブレーク点におけるF値を取る（SupF）

---

(2) Double Maximum 検定

ブレーク数 m が未知の場合：

- **UDmax**：均一重み

- **WDmax**：p値等重み

$$UD = \max_{1 \le m \le M} F_T(m)$$

→ $M = 5$ 程度で十分。ε（トリミング率）=0.05〜0.25 の臨界値を提供。

---

(3) Sequential 検定 (supF(ℓ+1|ℓ))

順次的に「ℓ個 vs ℓ+1個のブレーク」を比較。
実際の推定ではこの手法（逐次追加法）が最も安定。

🔷 7. ブレーク数の決定

情報量基準に基づく選択も可能：

- **BIC**（Yao, 1988）：過大推定しがち
- **LWZ**（Liu–Wu–Zidek, 1997）：過少推定傾向
- **Sequential supF test**：実務上最も堅牢

🔷 8. 実践的推奨事項（Bai & Perron 2000 シミュレーションに基づく）

条件	推奨トリミング ε	推奨方法
同分散・無相関	0.05	SupF / Sequential
異分散または系列相関あり	≥0.15–0.20	Sequential test 推奨
小標本 (T<100)	≥0.15	BICとSequential併用
大標本 (T>200)	0.10 以下可	DPアルゴリズム安定

🔷 9. 意義と応用

本論文は、構造変化モデルの理論と実務を統合した決定的研究であり：

- **マクロ経済のレジーム転換分析**（例：景気転換点）
- **金融時系列のパラメータ変動検出**（例：ボラティリティ体制）
- **政策効果の時点特定**（例：制度改革前後の回帰差）

など、多くの応用研究の標準手法となっています。

🔷 10. まとめ：技術的要点

項目	内容
モデル	部分／純構造変化モデル
推定法	動的計画法によるグローバル最適化
計算量	$O(T^2)$（従来の $O(T^m)$ に比べ劇的に改善）
信頼区間	Brownian-motionベースの近似分布
検定	SupF, Double Max, Sequential
実装	GAUSSコード付属、現在はR/Pythonにも移植（`strucchange`, `ruptures` など）

ご希望であれば、
この論文を基にした **Python実装（DPアルゴリズム＋SupF検定）** の完全スクリプトも作成可能です。
ご要望ありますか？

Response:

思考時間: 4m 29s

以下で、

1. **Bai & Perron 型のブレークポイントモデル（純構造変化）**
2. **論文 3.6 節の閾値モデル（threshold model）**

を、それぞれ

- 論文の内容を簡潔に整理
- それに対応する **Python 実装コード**

という順で示します。

---

1. 純構造変化モデル（multiple structural breaks）の実装

1.1 モデルとアルゴリズムの要点

論文の基本形（純構造変化）は

$$y_t = z_t' \nu_j + u_t, \quad t = T_{j-1}+1, \dots, T_j, \quad j = 1, \dots, m+1$$

- m ：ブレーク数（未知）
- 各セグメント j ごとに係数 ν_j が異なる
- ブレーク点 $T_1, \dots, T_m$ を **OLS の残差平方和（SSR）を最小化する分割**として求める J of Applied Econometrics - 200… 。

**重要ポイント（論文）

J of Applied Econometrics - 200…

**

- 全ての可能な区間 $[i, j]$ に対して OLS の SSR を計算し、**三角行列**として保持。
- 任意の m 分割の SSR は、これら区間 SSR の線形和で書ける。
- そこで、動的計画法（Dynamic Programming）で

  $$SSR_k(1..t) = \min_s \{SSR_{k-1}(1..s) + SSR(s+1..t)\}$$

  を再帰的に計算し、**O(T²)** でグローバル最小の分割を得る J of Applied Econometrics - 200… 。

以下の実装は：

- **純構造変化モデルのみ**（すべての係数がセグメントごとに変化）
- 共分散構造の推定や HAC は入れていない（単純な OLS ベースの推定）
- 最小セグメント長 h を強制（trimming 相当）

1.2 Python 実装コード

```python
import numpy as np

def ols_segment_SSR_precompute(y, Z):
    """
    y: shape (T,)
    Z: shape (T, q)
    各 t までの累積クロスプロダクトを事前計算し、
    任意区間 [i, j] の SSR を高速に計算できるようにする。
    """
    y = np.asarray(y)
    Z = np.asarray(Z)
    T, q = Z.shape

    Sy = np.zeros(T + 1)            # 累積 y'y
    SZZ = np.zeros((T + 1, q, q))   # 累積 Z'Z
    SZy = np.zeros((T + 1, q))      # 累積 Z'y

    for t in range(T):
        Sy[t+1] = Sy[t] + y[t]**2
        SZZ[t+1] = SZZ[t] + np.outer(Z[t], Z[t])
        SZy[t+1] = SZy[t] + Z[t] * y[t]

    return Sy, SZZ, SZy

def segment_SSR(i, j, Sy, SZZ, SZy, ridge=1e-12):
    """
    区間 [i, j]（両端含む）での回帰 y ~ Z の SSR を計算。
    SSR = y'y - (Z'y)' (Z'Z)^{-1} (Z'y)
    """
    Syy = Sy[j+1] - Sy[i]
    SZZ_seg = SZZ[j+1] - SZZ[i]
    SZy_seg = SZy[j+1] - SZy[i]

    # 数値安定化用のごく弱いリッジ
    SZZ_reg = SZZ_seg + ridge * np.eye(SZZ_seg.shape[0])

    try:
        beta = np.linalg.solve(SZZ_reg, SZy_seg)
    except np.linalg.LinAlgError:
        beta = np.linalg.lstsq(SZZ_reg, SZy_seg, rcond=None)[0]

    SSR = Syy - SZy_seg @ beta
    return SSR, beta

def fit_pure_structural_breaks(y, Z, m, h):
    """
    純構造変化モデル:
```

```
        y_t = z_t' * nu_j + u_t,    t in regime j
    について、Bai & Perron (2003) の動的計画法:contentReference[oaicite:3]{index=3}
に基づき
    m 個のブレークを推定する。

    Parameters
    ----------
    y : array-like, shape (T,)
        応答変数
    Z : array-like, shape (T, q)
        説明変数（すべての係数がセグメントごとに変化）
    m : int
        ブレーク数
    h : int
        最小セグメント長（各セグメントに少なくとも h 観測が必要）

    Returns
    -------
    breakpoints : list of int
        ブレークインデックス（0-based, その時点が「セグメントの終わり」）
        例: [24, 60] は
            [0..24], [25..60], [61..T-1] の 3 セグメント。
    coefs : list of ndarray
        各セグメントの OLS 係数ベクトル
    SSR_total : float
        全セグメントの SSR（最小値）
    """
    y = np.asarray(y)
    Z = np.asarray(Z)
    T = len(y)
    K = m + 1  # セグメント数

    if T < K * h:
        raise ValueError("サンプルが短すぎます: T < (m+1)*h")

    # 1. 累積クロスプロダクト
    Sy, SZZ, SZy = ols_segment_SSR_precompute(y, Z)

    # 2. 各区間 [i, j] の SSR を前計算
    SSR_mat = np.full((T, T), np.inf)
    for i in range(T):
        for j in range(i + h - 1, T):
            SSR_mat[i, j], _ = segment_SSR(i, j, Sy, SZZ, SZy)

    # 3. 動的計画法テーブル
    dp = np.full((K, T), np.inf)    # dp[k, t]: 先頭〜t を k+1 セグメントに分割した
最小 SSR
    prev = np.full((K, T), -1, int)
```

```
    # ベースケース: k = 0 （1セグメント）
    for t in range(h - 1, T):
        # 残りのセグメント (K-1) を入れるために十分な長さが残っているか
        if T - (t + 1) < (K - 1) * h:
            continue
        dp[0, t] = SSR_mat[0, t]
        prev[0, t] = -1

    # 再帰ステップ: k = 1..K-1
    for k in range(1, K):
        for t in range((k + 1) * h - 1, T):
            # 残りのセグメントに必要な長さが残っているか
            if T - (t + 1) < (K - 1 - k) * h:
                continue

            best_val = np.inf
            best_s = -1

            # 一つ前のセグメントの終端 s
            s_min = k * h - 1
            s_max = t - h
            for s in range(s_min, s_max + 1):
                if not np.isfinite(dp[k-1, s]):
                    continue
                val = dp[k-1, s] + SSR_mat[s+1, t]
                if val < best_val:
                    best_val = val
                    best_s = s

            dp[k, t] = best_val
            prev[k, t] = best_s

    # 4. 最後のセグメントは必ず T-1 で終了させる
    last_k = K - 1
    best_end = T - 1
    best_val = dp[last_k, best_end]

    if not np.isfinite(best_val):
        raise ValueError("フルサンプルをカバーする分割が見つかりません。m や h を見直して
ください。")

    # 5. バックトラックでブレークポイントを復元
    breakpoints = []
    t = best_end
    for k in range(last_k, 0, -1):
        s = prev[k, t]
        breakpoints.append(s)
```

```
        t = s
    breakpoints = sorted(breakpoints)

    # 6. 各セグメントの OLS 係数を再推定
    coefs = []
    start = 0
    for bp in breakpoints + [T - 1]:
        _, beta = segment_SSR(start, bp, Sy, SZZ, SZy)
        coefs.append(beta)
        start = bp + 1

    return breakpoints, coefs, best_val
```

簡単な使い方例（シミュレーション）

```
# 1 つのブレークを持つ単純な例
T = 200
np.random.seed(0)
Z = np.random.randn(T, 1)

true_break = 120
beta1 = 1.0
beta2 = 3.0
sigma = 0.1

y = np.empty(T)
y[:true_break]  = beta1 * Z[:true_break, 0]  + sigma *
np.random.randn(true_break)
y[true_break:]  = beta2 * Z[true_break:, 0] + sigma * np.random.randn(T -
true_break)

breaks, coefs, ssr = fit_pure_structural_breaks(y, Z, m=1, h=20)
print("推定ブレーク:", breaks)
print("推定係数:", coefs)
```

---

2. 閾値モデル（Threshold Model）の理論と実装

2.1 論文の閾値モデルの内容（3.6 節）

J of Applied Econometrics - 200…

Bai & Perron は、構造変化モデルのアルゴリズムを**閾値回帰（threshold regression）**にも拡張できることを示しています

J of Applied Econometrics - 200…

。

モデル例：

$$y_t = x_t'\beta + z_t'\nu_j + u_t, \quad \text{if}\, \gamma_{j-1} < v_t \le \gamma_j, \quad j = 1, \ldots, m+1$$

- v_t ：観測可能な閾値変数（たとえばラグ付き y や別のマクロ指標など）
- $\gamma_1, \ldots, \gamma_m$ ：未知の閾値（thresholds）
- 区間 $(\gamma_{j-1}, \gamma_j]$ が「レジーム j 」に対応

**キーアイデア

J of Applied Econometrics - 200…

**

1. v_t の昇順に並び替えた系列 $v_{t_1} \le v_{t_2} \le \cdots \le v_{t_T}$ を考える。

2. 並べ替え後の時系列 $(y_{t_r}, x_{t_r}, z_{t_r})_{r=1}^T$ の上で、
「ブレーク日による構造変化モデル」として扱うことができる。

3. 並べ替え後の index でのブレークポイント $T_1, \ldots, T_m$ から、
閾値は単に

$$\hat{\gamma}_j = v_{t_{T_j}}$$

として復元できる。

つまり、

> **「閾値モデルの推定 ＝ 閾値変数でソートした空間での多重構造変化問題」**

という帰着になり、前節の DP アルゴリズムをそのまま利用できる、というのが論文の主張です

J of Applied Econometrics - 200…

。

2.2 閾値モデルのシンプルな Python 実装

ここでは説明をシンプルにするため、

- x_t （共通係数）は無し（純閾値モデル）
- モデルは

$$y_t = z_t' \nu_j + u_t, \quad \gamma_{j-1} < v_t \leq \gamma_j$$

- 閾値変数 v_t は 1 次元

とします。
実装の流れ：

1. v_t の昇順に並べ替える（index を保持）。
2. 並べ替え後の (y, Z) に対して、**上で実装した** `fit_pure_structural_breaks` を適用。
3. 得られたブレーク位置を閾値 γ_j に変換。
4. その閾値を使って「元の時間順」に戻した上で、各レジームごとに OLS で ν_j を推定。

```python
def fit_threshold_model(y, Z, v, m, h):
    """
    閾値モデル（純閾値版）：
        y_t = z_t' * nu_j + u_t,
        gamma_{j-1} < v_t <= gamma_j でレジーム j

    を、Bai & Perron の「閾値変数でソート → ブレーク問題に帰着」という方法で推定する。

    Parameters
    ----------
    y : array-like, shape (T,)
        応答変数
    Z : array-like, shape (T, q)
        説明変数（係数はレジームごとに変化）
    v : array-like, shape (T,)
        閾値変数
    m : int
        閾値の数（= ブレーク数）
    h : int
        ソート後の「最小レジーム観測数」（各レジームに少なくとも h 点）

    Returns
    -------
    gammas : list of float
        推定された閾値（gamma_1, ..., gamma_m）
    coefs : list of ndarray
        各レジームの係数ベクトル
    ssr_sorted : float
        ソート後空間での最小 SSR（参考）
    """
    y = np.asarray(y)
    Z = np.asarray(Z)
    v = np.asarray(v)
    T = len(y)

    # 1. 閾値変数 v の昇順でソート
    idx = np.argsort(v)
    y_sorted = y[idx]
    Z_sorted = Z[idx]
    v_sorted = v[idx]

    # 2. ソート空間で純構造変化モデルとしてブレーク推定
    breakpoints_sorted, coefs_sorted, ssr_sorted = fit_pure_structural_breaks(
        y_sorted, Z_sorted, m=m, h=h
    )

    # 3. ブレーク位置から閾値を復元
```

```
    gammas = [v_sorted[bp] for bp in breakpoints_sorted]

    # 4. 閾値に基づいて、元の順序のデータをレジーム分割し係数推定
    coefs = []
    bounds = [-np.inf] + gammas + [np.inf]
    for j in range(len(bounds) - 1):
        mask = (v > bounds[j]) & (v <= bounds[j+1])
        Z_j = Z[mask]
        y_j = y[mask]

        if len(y_j) == 0:
            # 安全のため: 実務では h を大きく取ることで基本的には起こらないようにする
            coefs.append(np.full(Z.shape[1], np.nan))
            continue

        SZZ = Z_j.T @ Z_j
        SZy = Z_j.T @ y_j
        beta_j = np.linalg.lstsq(SZZ, SZy, rcond=None)[0]
        coefs.append(beta_j)

    return gammas, coefs, ssr_sorted
```

閾値モデルの簡単なシミュレーション例

```
# 閾値 v=0 で係数が変わる単純な例
T = 300
np.random.seed(1)
v = np.random.randn(T)
Z = np.random.randn(T, 1)

beta_low  = 1.0   # v <= 0 のとき
beta_high = -2.0  # v > 0 のとき
sigma = 0.1

y = np.where(v <= 0,
             beta_low  * Z[:, 0],
             beta_high * Z[:, 0]) + sigma * np.random.randn(T)

gammas, coefs, ssr = fit_threshold_model(y, Z, v, m=1, h=30)

print("推定された閾値 gamma:", gammas)
print("レジーム1係数 (v <= gamma):   ", coefs[0])
print("レジーム2係数 (v >  gamma):   ", coefs[1])
```

3. まとめと拡張のポイント

- 上記コードは論文の中核部分：
  - **O(T^2) 動的計画法によるグローバルなブレーク推定** J of Applied Econometrics - 200…
  - **閾値変数でソートして閾値モデルを構造変化問題に帰着するアイデア** J of Applied Econometrics - 200… を、そのまま Python で再現した、**最小限だけど素直な実装**です。
- 実務レベルで Bai–Perron を完全に再現するには、さらに：
  - 部分構造変化モデル（共通係数 β とセグメント特有係数 ν の両方）
  - HAC 共分散を用いた Wald 系の検定統計（supF, UDmax, WDmax など）
  - supF(ℓ+1|ℓ) によるブレーク数の逐次推定 J of Applied Econometrics - 200…
  - ブレーク日の漸近分布に基づく信頼区間
    などを実装する必要があります。

もし、

- 「共通係数 β がある partial structural change モデルも含めて実装したい」
- 「supF 検定や sequential procedure を Python で再現したい」
- 「あなたの実データ（マクロ・金融時系列）にそのまま当てはめるラッパ関数が欲しい」

といったニーズがあれば、その前提でモジュール風に整理したコードも用意できます。

Response:

思考時間: 58s

了解です。
ここでは

1. **supF 検定（H0: ブレークなし vs H1: m 個のブレーク）**
2. **supF(ℓ+1｜ℓ) を使った sequential procedure（ブレーク数の推定）**

を、前回の **純構造変化モデル用 DP 実装** を拡張する形で Python コード付きで示します。

前提：
モデルは純構造変化

$$y_t = z_t' \nu_j + u_t, \quad t \in \text{regime } j$$

で、各レジームの回帰係数 ν_j が変化するケースを扱います。

---

1. 下準備：DP と SSR 計算（前回コード＋α）

前回の DP 実装と SSR 関数に、**「与えられたブレーク集合の SSR」を計算するヘルパー**だけを追加します。
（前回コードをそのまま貼っておきます。もう書いてあれば、このブロックは飛ばしても OK です。）

```python
import numpy as np

def ols_segment_SSR_precompute(y, Z):
    """
    y: shape (T,)
    Z: shape (T, q)
    任意区間 [i, j] の SSR を高速に計算するための累積和を構築。
    """
    y = np.asarray(y)
    Z = np.asarray(Z)
    T, q = Z.shape

    Sy = np.zeros(T + 1)             # 累積 y'y
    SZZ = np.zeros((T + 1, q, q))    # 累積 Z'Z
    SZy = np.zeros((T + 1, q))       # 累積 Z'y

    for t in range(T):
        Sy[t+1] = Sy[t] + y[t]**2
        SZZ[t+1] = SZZ[t] + np.outer(Z[t], Z[t])
        SZy[t+1] = SZy[t] + Z[t] * y[t]

    return Sy, SZZ, SZy

def segment_SSR(i, j, Sy, SZZ, SZy, ridge=1e-12):
    """
    区間 [i, j]（両端含む）での回帰 y ~ Z の SSR を計算。
    SSR = y'y - (Z'y)' (Z'Z)^{-1} (Z'y)
    """
    Syy = Sy[j+1] - Sy[i]
    SZZ_seg = SZZ[j+1] - SZZ[i]
    SZy_seg = SZy[j+1] - SZy[i]

    # 数値安定化用のごく弱いリッジ
    SZZ_reg = SZZ_seg + ridge * np.eye(SZZ_seg.shape[0])

    try:
        beta = np.linalg.solve(SZZ_reg, SZy_seg)
    except np.linalg.LinAlgError:
        beta = np.linalg.lstsq(SZZ_reg, SZy_seg, rcond=None)[0]

    SSR = Syy - SZy_seg @ beta
    return SSR, beta

def fit_pure_structural_breaks(y, Z, m, h):
    """
    純構造変化モデル:
        y_t = z_t' * nu_j + u_t,    t in regime j
```

```
    について、Bai & Perron 型 DP【グローバル最小 SSR】で
    m 個のブレークを推定する。

    Parameters
    ----------
    y : (T,)
    Z : (T, q)
    m : int         ブレーク数
    h : int         最小セグメント長

    Returns
    -------
    breakpoints : list[int]
        ブレークインデックス (0-based, その index が「セグメントの終わり」)
    coefs : list[np.ndarray]
        各セグメントの OLS 係数
    SSR_total : float
        最小 SSR
    """
    y = np.asarray(y)
    Z = np.asarray(Z)
    T = len(y)
    K = m + 1  # セグメント数

    if T < K * h:
        raise ValueError("サンプルが短すぎます: T < (m+1)*h")

    # 1. 累積クロスプロダクト
    Sy, SZZ, SZy = ols_segment_SSR_precompute(y, Z)

    # 2. 各区間 [i, j] の SSR を事前計算
    SSR_mat = np.full((T, T), np.inf)
    for i in range(T):
        for j in range(i + h - 1, T):
            SSR_mat[i, j], _ = segment_SSR(i, j, Sy, SZZ, SZy)

    # 3. DP テーブル
    dp = np.full((K, T), np.inf)    # dp[k, t]: 先頭〜t を k+1 セグメントに分割した 最小 SSR
    prev = np.full((K, T), -1, int)

    # ベースケース: k = 0 (1セグメント)
    for t in range(h - 1, T):
        if T - (t + 1) < (K - 1) * h:
            continue
        dp[0, t] = SSR_mat[0, t]
        prev[0, t] = -1
```

```python
    # 再帰: k = 1..K-1
    for k in range(1, K):
        for t in range((k + 1) * h - 1, T):
            if T - (t + 1) < (K - 1 - k) * h:
                continue

            best_val = np.inf
            best_s = -1

            s_min = k * h - 1
            s_max = t - h
            for s in range(s_min, s_max + 1):
                if not np.isfinite(dp[k-1, s]):
                    continue
                val = dp[k-1, s] + SSR_mat[s+1, t]
                if val < best_val:
                    best_val = val
                    best_s = s

            dp[k, t] = best_val
            prev[k, t] = best_s

    # 最後のセグメントは T-1 で終了
    last_k = K - 1
    best_end = T - 1
    best_val = dp[last_k, best_end]
    if not np.isfinite(best_val):
        raise ValueError("フルサンプルをカバーする分割が見つかりません。m や h を見直して
ください。")

    # バックトラックでブレーク復元
    breakpoints = []
    t = best_end
    for k in range(last_k, 0, -1):
        s = prev[k, t]
        breakpoints.append(s)
        t = s
    breakpoints = sorted(breakpoints)

    # 各セグメントの係数再推定
    coefs = []
    Sy, SZZ, SZy = ols_segment_SSR_precompute(y, Z)  # 念のため再計算
    start = 0
    for bp in breakpoints + [T - 1]:
        _, beta = segment_SSR(start, bp, Sy, SZZ, SZy)
        coefs.append(beta)
        start = bp + 1
```

```
    return breakpoints, coefs, best_val

def SSR_from_breaks(y, Z, breakpoints):
    """
    与えられたブレーク集合（0-based, セグメント終端）から
    全体の SSR を計算するヘルパー。
    """
    y = np.asarray(y)
    Z = np.asarray(Z)
    T = len(y)

    Sy, SZZ, SZy = ols_segment_SSR_precompute(y, Z)
    bps = list(breakpoints)
    SSR_total = 0.0
    start = 0
    for bp in bps + [T - 1]:
        SSR_seg, _ = segment_SSR(start, bp, Sy, SZZ, SZy)
        SSR_total += SSR_seg
        start = bp + 1
    return SSR_total
```

---

2. supF 検定：H0（ブレーク無し） vs H1（m ブレーク）

2.1 理屈

純構造変化モデルで

- 制約付きモデル（H0：ブレーク無し）：
  パラメータ数 = q
- 制約無しモデル（H1：m ブレーク）：
  各レジームで q パラメータ → 合計 (m+1)q 個

残差平方和を

- SSR_0：ブレーク無し
- SSR_m：m ブレーク（最小 SSR：動的計画法で求めたもの）

とすると、通常の回帰 F 検定に対応する統計量は

$$F_{\mathrm{supF}(m)} = \frac{(SSR_0 - SSR_m)/(mq)}{SSR_m/(T - (m+1)q)}.$$

Bai–Perron の supF は「ブレーク位置で最適化した F」なので、
DP によって最小 SSR_m を使うことで、そのまま supF 統計量になります。
（p 値・臨界値は **通常の F 分布ではなく** Bai–Perron のシミュレーション表を参照する必要があります）

2.2 実装

```
def supF_0_vs_m(y, Z, m, h):
    """
    supF(m) 検定統計量を計算（H0: ブレーク無し vs H1: m ブレーク）。

    純構造変化モデルを仮定し、i.i.d. 誤差・同分散の通常 F 統計量の形で計算。
    ※ p値は Bai-Perron の臨界値表と比較するのが望ましい。

    Returns
    -------
    F_stat : float
        supF(m) の統計量
    SSR0 : float
        ブレーク無し（1 セグメント）の SSR
    SSRm : float
        m ブレークの最小 SSR（DP による）
    breakpoints_m : list[int]
        m ブレークの推定位置
    """
    y = np.asarray(y)
    Z = np.asarray(Z)
    T, q = Z.shape

    # ブレーク無しの SSR
    Sy, SZZ, SZy = ols_segment_SSR_precompute(y, Z)
    SSR0, _ = segment_SSR(0, T-1, Sy, SZZ, SZy)

    # m ブレークの SSR（グローバル最小）
    breakpoints_m, _, SSRm = fit_pure_structural_breaks(y, Z, m=m, h=h)

    df1 = m * q
    df2 = T - (m + 1) * q
    if df1 <= 0 or df2 <= 0:
        raise ValueError("自由度が負またはゼロになっています。T, m, q を確認してください。")

    F_stat = ((SSR0 - SSRm) / df1) / (SSRm / df2)
    return F_stat, SSR0, SSRm, breakpoints_m
```

3. Sequential procedure：supF(ℓ+1｜ℓ) によるブレーク数推定

3.1 考え方

Bai & Perron の sequential 法はざっくりいうと：

1. まず H0: 0 ブレーク vs H1: 1 ブレークで supF(1) を計算し、
   「何らかのブレークが存在するか」を検定。
2. 有意なら 1 ブレークモデルを受け入れ、次に
   - H0: 1 ブレーク vs H1: 2 ブレーク
     → supF(2|1) を計算。
3. supF(ℓ+1|ℓ) の計算は
   - 現在の ℓ ブレークモデルの各セグメントごとに
     「そのセグメント内でさらに 1 ブレーク入りうるか？」を supF(1) で調べ、
   - そのセグメント群の中で最大の F を supF(ℓ+1|ℓ) とする。
4. これを ℓ = 0,1,... と繰り返し、臨界値を下回ったところで停止。
   → その ℓ が推定ブレーク数。

ここでは、

- 臨界値との比較は**ユーザー側で**行う前提で
- Python 関数は
  「各 ℓ での supF(ℓ+1|ℓ) と、対応するブレーク構造」を返す

ようにします。

3.2 実装コード

```
def supF_1_within_segment(y_seg, Z_seg, h):
    """
    1つのセグメント上で、
    H0: ブレーク無し vs H1: 1 ブレーク
    の supF(1) を計算する補助関数。

    Returns
    -------
    F_stat : float
    SSR0_seg : float
    SSR1_seg : float
    local_break : int or None
        セグメント内のブレーク位置 (0-based, セグメント内 index)
    """
    y_seg = np.asarray(y_seg)
    Z_seg = np.asarray(Z_seg)
    n = len(y_seg)
    q = Z_seg.shape[1]

    if n < 2 * h or n <= 2 * q:
        # ブレークを入れられるだけの長さがない
        return np.nan, np.nan, np.nan, None

    # セグメント内での SSR0
    Sy, SZZ, SZy = ols_segment_SSR_precompute(y_seg, Z_seg)
    SSR0_seg, _ = segment_SSR(0, n-1, Sy, SZZ, SZy)

    # セグメント内で 1 ブレーク DP
    try:
        bps_local, _, SSR1_seg = fit_pure_structural_breaks(y_seg, Z_seg, m=1,
h=h)
    except ValueError:
        return np.nan, np.nan, np.nan, None

    df1 = 1 * q
    df2 = n - (1 + 1) * q  # (1+1)セグメント
    if df1 <= 0 or df2 <= 0:
        return np.nan, SSR0_seg, SSR1_seg, None

    F_stat = ((SSR0_seg - SSR1_seg) / df1) / (SSR1_seg / df2)
    local_break = bps_local[0] if len(bps_local) > 0 else None
    return F_stat, SSR0_seg, SSR1_seg, local_break

def sequential_break_selection(y, Z, max_breaks, h):
    """
    supF(ℓ+1|ℓ) に基づく sequential procedure を実装。
```

```
    手順:
        - ℓ = 0,1,2,... と増やしながら、
          ・ℓ ブレークモデルを DP で推定
          ・各セグメントで supF(1) を計算して supF(ℓ+1|ℓ) を求める
        - ここでは「いつ止めるか」はユーザーが臨界値と比較して判断する前提で、
          各 ℓ に対する情報をすべて返す。

    Parameters
    ----------
    y : (T,)
    Z : (T, q)
    max_breaks : int
        探索する最大ブレーク数 M
    h : int
        最小セグメント長

    Returns
    -------
    results : list[dict]
        各 ℓ (0..max_breaks-1) について以下を含む辞書のリスト:
        - 'ell' : ℓ
        - 'breaks_ell' : ℓ ブレークの推定ブレーク位置 (list[int])
        - 'SSR_ell' : ℓ ブレークモデルの最小 SSR
        - 'supF_ell_plus_1' : supF(ℓ+1|ℓ) の値
        - 'candidate_new_break' : supF を達成する候補追加ブレーク位置 (global index)
        - 'segment_index' : どのセグメントに追加されたか (0-based)
    """
    y = np.asarray(y)
    Z = np.asarray(Z)
    T = len(y)
    q = Z.shape[1]

    results = []

    # ℓ = 0: ブレークなし
    Sy_full, SZZ_full, SZy_full = ols_segment_SSR_precompute(y, Z)
    SSR0, _ = segment_SSR(0, T-1, Sy_full, SZZ_full, SZy_full)
    breaks_ell = []  # ℓ=0 なのでブレークなし

    SSR_ell = SSR0
    results.append({
        'ell': 0,
        'breaks_ell': [],
        'SSR_ell': SSR0,
        'supF_ell_plus_1': None,
        'candidate_new_break': None,
        'segment_index': None
    })
```

```python
    # ℓ = 0..max_breaks-1 で supF(ℓ+1|ℓ) を計算
    for ell in range(0, max_breaks):
        # まず ℓ ブレークモデルのグローバルな推定 (ℓ>=1 のとき)
        if ell > 0:
            breaks_ell, _, SSR_ell = fit_pure_structural_breaks(y, Z, m=ell,
h=h)

        # セグメント境界を構成 (0-based index の終端)
        if len(breaks_ell) == 0:
            seg_ends = [T - 1]
            seg_starts = [0]
        else:
            seg_ends = breaks_ell + [T - 1]
            seg_starts = [0] + [bp + 1 for bp in breaks_ell]

        # 各セグメントで supF(1) を計算
        best_F = -np.inf
        best_new_break = None
        best_seg_idx = None

        for seg_idx, (s, e) in enumerate(zip(seg_starts, seg_ends)):
            y_seg = y[s:e+1]
            Z_seg = Z[s:e+1, :]

            F_seg, SSR0_seg, SSR1_seg, local_break =
supF_1_within_segment(y_seg, Z_seg, h=h)
            if np.isnan(F_seg) or local_break is None:
                continue

            if F_seg > best_F:
                best_F = F_seg
                best_seg_idx = seg_idx
                # セグメント内 index → 全体 index に変換
                best_new_break = s + local_break

        results[ell]['supF_ell_plus_1'] = best_F if best_F != -np.inf else None
        results[ell]['candidate_new_break'] = best_new_break
        results[ell]['segment_index'] = best_seg_idx

        # 次のループでは ell を 1 増やし、fit_pure_structural_breaks で ℓ+1 ブレーク
を推定する想定。
        # 実際に「止める」かどうかは、ユーザーが results[ell]['supF_ell_plus_1'] と
        # Bai-Perron の臨界値表を比較して判断する想定。

    return results
```

4. 使い方イメージ

4.1 supF(0 vs m) の例

```
T = 200
np.random.seed(0)
Z = np.random.randn(T, 1)
true_break = 120
beta1, beta2 = 1.0, 3.0
sigma = 0.2

y = np.empty(T)
y[:true_break]  = beta1 * Z[:true_break, 0]  + sigma *
np.random.randn(true_break)
y[true_break:]  = beta2 * Z[true_break:, 0] + sigma * np.random.randn(T -
true_break)

F_sup, SSR0, SSR1, breaks_m = supF_0_vs_m(y, Z, m=1, h=20)
print("supF(1) =", F_sup)
print("推定ブレーク位置:", breaks_m)
```

4.2 sequential procedure の例

```
results = sequential_break_selection(y, Z, max_breaks=3, h=20)

for res in results:
    ell = res['ell']
    print(f"\nell = {ell}")
    print("  ブレーク位置 (ell個):", res['breaks_ell'])
    print("  SSR_ell:", res['SSR_ell'])
    print("  supF(ell+1 | ell):", res['supF_ell_plus_1'])
    print("  候補追加ブレーク位置:", res['candidate_new_break'])
```

ここで得られた `supF_ell_plus_1` を、Bai & Perron (2003) や関連の補論で示されている **臨界値表（trim=ε, q 次元などに依存）** と比較して、

- supF(1|0) が臨界値を超える → 少なくとも 1 ブレークあり
- supF(2|1) が臨界値を超える → 2 ブレークに増やす
- supF(3|2) が臨界値を下回る → 推定ブレーク数は 2

といった具合にブレーク数を決めるイメージです。

---

5. 補足

- 上記の実装は
  - 純構造変化モデル（全係数がレジームごとに変化）
  - i.i.d. 誤差・同分散という単純なケース
    を想定した **基本形** です。
- Bai–Perron 論文本来は、
  - 部分構造変化（共通係数 β＋変化する係数 v）
  - シリアル相関・異分散に対する HAC ロバストな Wald 統計
  - trimming ε の調整
    などを含んでいますが、それらは「共分散行列推定」と「制約付き回帰行列の構成」がかなり重たくなるので、
    必要でしたらそこまで拡張したバージョンも段階的に書いていけます。

もし次のステップとして、

- partial structural change モデル（β 共通＋v セグメント別）での sequential procedure
- あなたのマクロ／金融データに直接適用できるヘルパー関数群

が欲しければ、その前提で設計します。

Response:

思考が停止しました

Response:

非常に良い発想です。
Bai–Perron型の**構造変化／閾値回帰モデル**を「インフレ率（π_t）」と「セクターリターン（$r_{i,t}$）」の関係に適用し、**マクロ環境に応じてセクターごとのベータを動的に切り替える運用モデル**を構築することができます。以下で、専門家向けに体系的な「投資運用手法」として整理します。

🔷 概要：インフレ体制別セクター・ローテーション戦略

基本アイデア

- インフレ率（または期待インフレ率）を**マクロ状態変数**とし、

- 各セクター（例：Energy, Financials, IT, Utilities 等）のリターンを

$$r_{i,t} = \alpha_i + \beta_{i,j}\pi_t + \varepsilon_{i,t}, \quad t \in \text{Regime } j$$

  のように回帰。

- ただし「Regime j」は **インフレ率または時点による regime segmentation** により決まる：
  - **構造変化モデル**：時点 $T_1, T_2, \ldots$ で回帰係数が変化。
  - **閾値モデル**：インフレ率が閾値 $\gamma_1, \gamma_2, \ldots$ を超えると係数が変化。

これにより、「インフレレジームごとのセクター感応度（β）」を明示的にモデル化できます。

🔷 モデル構築のステップ

1. データセット

- 月次（または週次）データ：
  - $r_{i,t}$ ：各セクターインデックスの超過リターン
  - π_t ：CPIやPCEから計算したインフレ率、または break-even inflation
  - オプションで x_t ：金利、成長率、失業率など他のマクロ変数

---

2. 推定モデル

(A) 構造変化モデル

$$r_{i,t} = \alpha_{i,j} + \beta_{i,j}\pi_t + \varepsilon_{i,t}, \quad t \in (T_{j-1}, T_j]$$

ブレーク時点 T_j を Bai–Perron の DP により推定。

- 結果として各セクターごとに複数の regime が得られる。
- 各 regime の傾き β_{i,j} が「インフレ感応度」。

(B) 閾値モデル

$$r_{i,t} = \alpha_i + \begin{cases} \beta_i^{(L)}\pi_t + \varepsilon_{i,t}, & \pi_t \le \gamma \\ \beta_i^{(H)}\pi_t + \varepsilon_{i,t}, & \pi_t > \gamma \end{cases}$$

γ は未知の閾値として最尤推定。
インフレ率が高い局面と低い局面でセクター感応度が変化する構造。

---

3. 投資ステップ（セクターローテーション）

1. **Regime 推定**
   - 最新期のインフレ率または regime 確率に基づき、現在の regime ĵ を特定。
2. **セクター別ベータの活用**
   - 各セクターの $\beta_{i,\hat{j}}$ を使用して、**マクロに対するレバレッジ感応度**を推定。
   - 例：高インフレ局面では Energy β↑、IT β↓。
3. **ポートフォリオ最適化**
   - 各セクターの期待超過リターンを

     $$E[r_i|\pi_t, \hat{j}] = \alpha_{i,\hat{j}} + \beta_{i,\hat{j}}\pi_t$$

     で予測。
   - 共分散行列 Σ̂ と合わせて、

     $$\max_w \ w'\mu_{\hat{j}} - \frac{\lambda}{2}w'\hat{\Sigma}w$$

     を解く（平均分散型最適化）。
4. **リバランス**
   - 毎月（または四半期）末に、最新のインフレ regime 判定に基づきポートフォリオを更新。

🔷 Python 実装の概略

以下の疑似コードは、Energy と IT の2セクターで閾値回帰＋動的最適化を行う例です。

```python
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

# ---- Step 1. データ読み込み ----
df = pd.read_csv("sector_inflation_data.csv", parse_dates=["date"])
pi = df["inflation"].values
r_energy = df["ret_energy"].values
r_it = df["ret_it"].values

# ---- Step 2. 閾値推定（例: 1つの閾値）----
def fit_threshold_regression(y, x, h=20):
    x_sorted_idx = np.argsort(x)
    x_sorted, y_sorted = x[x_sorted_idx], y[x_sorted_idx]
    T = len(y)
    best_gamma, best_ssr = None, np.inf
    for i in range(h, T-h):
        gamma = x_sorted[i]
        low_mask = x <= gamma
        high_mask = x > gamma
        if sum(low_mask) < h or sum(high_mask) < h:
            continue
        model_low = LinearRegression().fit(x[low_mask].reshape(-1,1),
y[low_mask])
        model_high = LinearRegression().fit(x[high_mask].reshape(-1,1),
y[high_mask])
        ssr = np.sum((y[low_mask]-
model_low.predict(x[low_mask].reshape(-1,1)))**2) \
            + np.sum((y[high_mask]-
model_high.predict(x[high_mask].reshape(-1,1)))**2)
        if ssr < best_ssr:
            best_ssr, best_gamma = ssr, gamma
            best_params = {
                "beta_low": model_low.coef_[0],
                "alpha_low": model_low.intercept_,
                "beta_high": model_high.coef_[0],
                "alpha_high": model_high.intercept_
            }
    return best_gamma, best_params

gamma_E, pars_E = fit_threshold_regression(r_energy, pi)
gamma_IT, pars_IT = fit_threshold_regression(r_it, pi)

print("Energy閾値:", gamma_E, pars_E)
print("IT閾値:", gamma_IT, pars_IT)

# ---- Step 3. 現在のレジームを判定 ----
```

```python
pi_t = pi[-1]
regime_E = "High" if pi_t > gamma_E else "Low"
regime_IT = "High" if pi_t > gamma_IT else "Low"

# ---- Step 4. 各セクターの期待リターンを予測 ----
def expected_return(pi_val, pars, regime):
    if regime == "High":
        return pars["alpha_high"] + pars["beta_high"] * pi_val
    else:
        return pars["alpha_low"] + pars["beta_low"] * pi_val

mu_E = expected_return(pi_t, pars_E, regime_E)
mu_IT = expected_return(pi_t, pars_IT, regime_IT)
mu_vec = np.array([mu_E, mu_IT])

# 共分散推定（単純に過去分散で代用）
rets = np.vstack([r_energy, r_it])
Sigma = np.cov(rets)

# ---- Step 5. 平均分散最適化（λ = リスク許容度）----
lam = 5.0
Sigma_inv = np.linalg.inv(Sigma)
w_opt = Sigma_inv @ mu_vec / (lam)
w_opt /= np.sum(np.abs(w_opt))  # 重みを正規化

print("推奨ウェイト:", w_opt)
```

🔷 戦略の特徴と拡張

観点	内容
マクロ適応性	インフレ体制ごとに係数を切り替えるため、**スタグフレーション局面など非線形効果を吸収**できる。
ポートフォリオ分散効果	各セクター β の符号が regime により反転するため、**リスク分散が体制依存的に変動**。
更新頻度	月次／四半期ごとの CPI 公表タイミングで regime 判定しリバランス。
拡張	- 部分構造変化モデル（β だけ変化）
- 複数マクロ変数（π, GDP growth, rates）を含む多次元閾値モデル	
- 確率的 regime 判定（例：HMMやDynamic Factorとの組み合わせ）	

🔷 発展方向

1. **Regime Probability に基づく確率加重ポートフォリオ**
   - 確率的 regime モデル（例：マルコフスイッチング）と組み合わせ、

     $$w_t = \sum_j p_{j,t}\, w_j$$

     とすることでリバランスをスムーズに。

2. **Expected Utility 最適化への拡張**
   - 投資家効用関数を regime 別に設定し、

     $$\max E[U(W_{t+1})|j_t]$$

     を数値的に最適化。

3. **Quantum / RL 強化**
   - 閾値切替を離散状態とみなし、**Regime-aware Reinforcement Learning** でポリシー（投資比率）を動的最適化する拡張も可能。

次に進める場合、以下2方向が考えられます：

1. 上記手法を**実データ（例：Fama-Frenchセクター＋米CPI）**で回す完全Pythonノートブック
2. 閾値または構造変化の**確率的 regime モデル化（Markov switching とのハイブリッド）**

どちらを進めたいですか？

Response:

思考時間: 26s

了解です。
ここでは

1. **実データを使った「インフレ×セクター」閾値／ブレーク＋ポートフォリオ運用の Python ノートブック雛形**
2. **確率的レジーム（Markov switching / HMM）と組み合わせたハイブリッド運用モデル**

の両方をまとめて整理します。
どちらも「そのままノートブックに貼れる」イメージで書きます（実行やAPIアクセスはユーザー側で調整ください）。

---

1. 実データで回す Python ノートブック構成

1.1 全体フロー

例として：

- インフレ率：FRED の CPI（CPIAUCSL）から月次前月比／前年比を計算
- セクター：ETF
  - XLE: Energy
  - XLF: Financials
  - XLK: Technology
  - XLU: Utilities
    （好みで他を追加）

を使うことにします。

フロー：

1. データ取得（`pandas_datareader`＋`yfinance` など）
2. 月次化＆インフレ率の計算
3. 各セクターごとに **閾値回帰（または構造変化）** を推定
4. 各月において
    - 現在のインフレ（またはレジーム）から各セクターの期待リターンを推定
    - 共分散行列を推定し Markowitz 最適化でウエイト決定
5. ローリングでバックテスト

以下は**閾値回帰（thresholdモデル）**で実装した例です（Bai–Perron型ブレークに差し替えることも容易です）。

1.2 データ取得と前処理

```python
import pandas as pd
import numpy as np
import yfinance as yf
from pandas_datareader import data as pdr

# yfinance を pandas_datareader に組み込む
yf.pdr_override()

start = "2000-01-01"
end   = "2025-01-01"

# 1. CPI (FRED)
cpi = pdr.DataReader("CPIAUCSL", "fred", start, end)
cpi = cpi.resample("M").last()  # 月末値
cpi.columns = ["CPI"]

# インフレ率: 前月比 or 前年同月比
infl_mom = cpi["CPI"].pct_change()                # 前月比
infl_yoy = cpi["CPI"].pct_change(12)              # 前年同月比
infl = infl_yoy.dropna()                          # ここでは前年比を採用

# 2. セクターETF価格（調整終値）
tickers = ["XLE", "XLF", "XLK", "XLU"]
px = yf.download(tickers, start=start, end=end)["Adj Close"]
px_m = px.resample("M").last()

# リターン（logリターン）
rets = np.log(px_m / px_m.shift(1)).dropna()
rets.columns = tickers

# インフレと結合（共通期間）
df = pd.concat([infl, rets], axis=1, join="inner")
df.columns = ["inflation"] + tickers
df = df.dropna()
df.head()
```

1.3 閾値回帰（1セクターあたり）

先ほどの簡易版を、汎用関数として定義し直します。

```python
from sklearn.linear_model import LinearRegression

def fit_threshold_regression_1d(y, x, h=24):
    """
    y: セクターリターン (T,)
    x: インフレ率 (T,)
    h: 各レジームの最小サンプル数

    1つの閾値 γ を探索して、
    r_t = α_L + β_L x_t (x <= γ)
    r_t = α_H + β_H x_t (x >  γ)
    を OLS でフィット。
    """
    y = np.asarray(y)
    x = np.asarray(x)
    T = len(y)

    sort_idx = np.argsort(x)
    x_sorted = x[sort_idx]
    y_sorted = y[sort_idx]

    best_ssr = np.inf
    best_gamma = None
    best_pars = None

    # h .. T-h の位置を閾値候補とする
    for i in range(h, T - h):
        gamma = x_sorted[i]

        low_mask = x <= gamma
        high_mask = x > gamma

        if low_mask.sum() < h or high_mask.sum() < h:
            continue

        X_low = x[low_mask].reshape(-1, 1)
        y_low = y[low_mask]
        X_high = x[high_mask].reshape(-1, 1)
        y_high = y[high_mask]

        model_low = LinearRegression().fit(X_low, y_low)
        model_high = LinearRegression().fit(X_high, y_high)

        ssr_low = np.sum((y_low - model_low.predict(X_low))**2)
        ssr_high = np.sum((y_high - model_high.predict(X_high))**2)
        ssr = ssr_low + ssr_high
```

```python
            if ssr < best_ssr:
                best_ssr = ssr
                best_gamma = gamma
                best_pars = {
                    "alpha_low":  model_low.intercept_,
                    "beta_low":   model_low.coef_[0],
                    "alpha_high": model_high.intercept_,
                    "beta_high":  model_high.coef_[0],
                }

    return best_gamma, best_pars, best_ssr
```

複数セクターに適用：

```python
infl_series = df["inflation"]
sector_cols = tickers

threshold_models = {}

for sec in sector_cols:
    y_sec = df[sec]
    gamma, pars, ssr = fit_threshold_regression_1d(y_sec, infl_series, h=24)
    threshold_models[sec] = {
        "gamma": gamma,
        "pars": pars,
        "ssr": ssr
    }

threshold_models
```

1.4 Regime に応じた期待リターンと Markowitz 最適化

期待リターンの計算

```python
def expected_return_threshold(pi_t, gamma, pars):
    if pi_t > gamma:
        return pars["alpha_high"] + pars["beta_high"] * pi_t
    else:
        return pars["alpha_low"] + pars["beta_low"] * pi_t
```

各時点 t で、 `pi_t` を用いて各セクターの期待リターンベクトル μ_t を計算：

```python
def compute_mu_vector(pi_t, models, sector_list):
    mu = []
    for sec in sector_list:
        gamma = models[sec]["gamma"]
        pars = models[sec]["pars"]
        mu_sec = expected_return_threshold(pi_t, gamma, pars)
        mu.append(mu_sec)
    return np.array(mu)
```

単純な平均分散最適化（無リスク資産なし）

$$\max_w \ w'\mu - \frac{\lambda}{2} w'\Sigma w, \quad \sum_i w_i = 1$$

Lagrange の1階条件から解析的解：

```
w^\* \propto \Sigma^{-1} \mu
```

ここでは簡単に

- Σ：過去 `window` ヶ月の共分散
- λ：スケールとして後で正規化するので省略し、重みは $\Sigma^{-1}\mu$ を正規化

```python
def mean_variance_weights(mu, Sigma):
    """
    mu: (N,)
    Sigma: (N,N)
    出力: w (N,), sum(|w|)=1 でスケール
    """
    Sigma_inv = np.linalg.inv(Sigma)
    w_raw = Sigma_inv @ mu
    if np.allclose(w_raw, 0):
        # 安全側：等重み
        return np.ones_like(w_raw) / len(w_raw)
    w = w_raw / np.sum(np.abs(w_raw))
    return w
```

1.5 ローリング・バックテスト

- 推定ウィンドウ：例 10年（120ヶ月）

- 月 t で
  1. 過去 120ヶ月 (t-120..t-1) から threshold モデルと共分散を推定
  2. 月 t のインフレ率 π_t から μ_t を計算
  3. $w_t = f(\mu_t, \Sigma_t)$ を決定
  4. 月 t+1 のセクターリターン r_{t+1} からポートフォリオリターンを計算

```
window = 120  # 10年ローリング
dates = df.index
n = len(df)
sector_cols = tickers

port_rets = []
port_dates = []
weights_hist = []

for t in range(window, n-1):
    hist = df.iloc[t-window:t]          # 推定期間
    pi_hist = hist["inflation"]
    rets_hist = hist[sector_cols]

    # 1. 新たに threshold モデル推定（実務では頻度を下げても良い）
    models_t = {}
    for sec in sector_cols:
        gamma, pars, ssr = fit_threshold_regression_1d(rets_hist[sec], pi_hist,
h=24)
        models_t[sec] = {"gamma": gamma, "pars": pars}

    # 2. 共分散推定
    Sigma_t = rets_hist.cov().values

    # 3. 月 t のインフレ π_t で μ_t を計算
    pi_t = df["inflation"].iloc[t]
    mu_t = compute_mu_vector(pi_t, models_t, sector_cols)

    # 4. 最適ウェイト
    w_t = mean_variance_weights(mu_t, Sigma_t)
    weights_hist.append(w_t)

    # 5. 翌月 t+1 の実現リターン
    r_next = df[sector_cols].iloc[t+1].values
    port_ret = np.dot(w_t, r_next)
    port_rets.append(port_ret)
    port_dates.append(dates[t+1])

bt = pd.Series(port_rets, index=port_dates, name="port_ret")

# 比較用: セクター等重みポートフォリオ
ew = df[sector_cols].mean(axis=1)
ew_bt = ew.loc[bt.index]

summary = pd.DataFrame({
    "strategy": bt,
    "equal_weight": ew_bt
```

```
})

summary.cumsum().plot(title="Cumulative Returns")
```

この枠組みをベースに、Bai–Perron の「時点によるブレーク」（DP 実装）に差し替えることもできます（インフレ系列に対してブレークを入れて regime を推定するか、各セクター回帰に直接ブレークを入れるか、など設計自由度があります）。

---

2. 確率的レジーム（Markov switching / HMM）とのハイブリッド運用

次に、**「インフレレジームが確率的に推移する」**モデルと組み合わせる案です。

2.1 コンセプト

2段階アプローチにするのが実装しやすいです：

1. **インフレの Markov switching モデル**
   - 例：2状態（低インフレ / 高インフレ）の MarkovRegression を推定
   - 各時点 t ごとに状態確率 $P(S_t = j|\text{data})$ を得る
2. **各レジームでのセクター回帰**
   - レジーム j ごとに
   $$r_{i,t} = \alpha_{i,j} + \beta_{i,j}\pi_t + \varepsilon_{i,t}$$
   - 実務的には、「状態 j の事後確率が高い期間」を抽出して OLS を行うか、EM的に重み付き回帰を行う。
3. **ポートフォリオ構築**
   - 各時点 t の regime 確率 $p_{j,t}$ を用いて
   $$\mu_i(t) = \sum_j p_{j,t}(\alpha_{i,j} + \beta_{i,j}\pi_t)$$
   - 共分散は全体または regime 加重で推定し、同様に MV 最適化。

2.2 インフレの Markov switching 回帰（statsmodels）

statsmodels にある `MarkovRegression` を用いた例です：

```
import statsmodels.api as sm

# インフレ series (df["inflation"]) を2状態のマルコフ回帰にフィット
pi = df["inflation"].values

# 単純な平均ゼロ周りのMSAR(1) なども可能だが、ここでは定数のみのRegime Meanモデルを例示
mod = sm.tsa.MarkovRegression(pi, k_regimes=2, trend='c',
switching_variance=True)
res = mod.fit(em_iter=10, search_reps=20)  # EM + ランダム初期値探索

print(res.summary())

# 各時点の状態の事後確率
smoothed_probs = res.smoothed_marginal_probabilities
df["p_regime0"] = smoothed_probs[0]
df["p_regime1"] = smoothed_probs[1]
```

これで、各月 t で「低インフレ（Regime 0）」と「高インフレ（Regime 1）」の確率が得られます。

2.3 Regime別セクター回帰（重み付きOLS）

レジーム j の「重み付きOLS」を行います：

$$\min_{\alpha_{i,j},\beta_{i,j}} \sum_t p_{j,t}(r_{i,t} - \alpha_{i,j} - \beta_{i,j}\pi_t)^2$$

これは、`sqrt(p_{j,t})` を重みとした加重回帰で実装できます。

```python
from sklearn.linear_model import LinearRegression

def weighted_regression(y, x, w):
    """
    y, x: (T,)
    w: (T,)  非負ウェイト
    """
    y = np.asarray(y)
    x = np.asarray(x)
    w = np.asarray(w)
    mask = w > 1e-6
    y, x, w = y[mask], x[mask], w[mask]
    X = x.reshape(-1, 1)

    # sklearn の sample_weight を使用
    model = LinearRegression().fit(X, y, sample_weight=w)
    alpha = model.intercept_
    beta = model.coef_[0]
    return alpha, beta

regime_models = {sec: {} for sec in sector_cols}
for sec in sector_cols:
    y_sec = df[sec].values
    x_pi = df["inflation"].values

    for j in [0, 1]:  # 2レジームの場合
        w_j = df[f"p_regime{j}"].values
        alpha_j, beta_j = weighted_regression(y_sec, x_pi, w_j)
        regime_models[sec][j] = {"alpha": alpha_j, "beta": beta_j}

regime_models
```

2.4 確率レジームを用いた期待リターンと運用

regime混合型の期待リターン

$$\mu_i(t) = \sum_{j=0}^{1} p_{j,t}(\alpha_{i,j} + \beta_{i,j}\pi_t)$$

```python
def mu_regime_mixture(t_idx, df, sector_cols, regime_models):
    pi_t = df["inflation"].iloc[t_idx]
    p0 = df["p_regime0"].iloc[t_idx]
    p1 = df["p_regime1"].iloc[t_idx]

    mu = []
    for sec in sector_cols:
        a0, b0 = regime_models[sec][0]["alpha"], regime_models[sec][0]["beta"]
        a1, b1 = regime_models[sec][1]["alpha"], regime_models[sec][1]["beta"]
        mu_i = p0 * (a0 + b0 * pi_t) + p1 * (a1 + b1 * pi_t)
        mu.append(mu_i)
    return np.array(mu)
```

ローリング・バックテスト（確率レジーム版）

簡略化のため「レジームモデルは全期間で1回推定」とし、
ウィンドウ共分散＋レジーム混合期待リターンで MV 最適化する例を示します。

```python
window = 120
dates = df.index
n = len(df)

port_rets_ms = []
port_dates_ms = []
weights_ms_hist = []

for t in range(window, n-1):
    # 共分散：過去windowの生データから
    rets_hist = df[sector_cols].iloc[t-window:t]
    Sigma_t = rets_hist.cov().values

    # 確率レジーム混合の期待リターン
    mu_t = mu_regime_mixture(t, df, sector_cols, regime_models)

    # MVウエイト
    w_t = mean_variance_weights(mu_t, Sigma_t)
    weights_ms_hist.append(w_t)

    r_next = df[sector_cols].iloc[t+1].values
    port_ret = np.dot(w_t, r_next)
    port_rets_ms.append(port_ret)
    port_dates_ms.append(dates[t+1])

bt_ms = pd.Series(port_rets_ms, index=port_dates_ms, name="port_ret_ms")

# 先ほどの threshold戦略、equal weight と比較
comparison = pd.concat([bt, bt_ms, ew_bt], axis=1)
comparison.columns = ["threshold_strategy", "ms_strategy", "equal_weight"]
comparison.cumsum().plot(title="Cumulative Returns: Threshold vs Markov
Switching vs EW")
```

---

3. 今後の発展アイデア

- **partial structural change**
  β（インフレ感応度）のみレジームごとに変えるモデル

$$r_{i,t} = \alpha_i + \beta_{i,j}\pi_t + \varepsilon_{i,t}$$

  を Bai–Perron の partial モデルとして実装（Sargan 型反復＋DP）。

- **多因子拡張**
  インフレに加え、金利・成長率・金融条件指数などを含むマルチファクター回帰に
  閾値や構造変化を導入。
- **RLとの統合**
  regimeごとに異なる報酬構造を持つ MDP として定式化し、
  Regime-aware RL（たとえば、状態空間にレジーム指標を含める）でポートフォリオを最適化。

このあたりから先は、

- 実データセット（あなたが普段使われているセクター・インデックスや CPI 系列）
- バックテスト期間
- ベンチマーク（TOPIX sectors / MSCI sectors など）

を前提に、もう少し**研究ノートブック＋論文草稿**に近い形で整えていくイメージになると思います。

もし次のステップとして、

- 「Bai–Perron の DP ブレークを**インフレ系列に直接入れる**バージョン」
- 「partial structural change（α共通＋βレジーム別）を Python 実装」

をやりたい場合、そこにフォーカスしたコードと数式整理も用意できます。

Response:

思考時間: 43s

了解です。
さっき話した「次のステップ」2つを、ちゃんと **Python の関数レベル**まで落として実装します：

1. **インフレ系列に Bai–Perron DP でブレークを入れる関数**
2. **そのブレークを使った partial structural change（α 共通＋ β レジーム別）回帰の実装**

全部「そのままノートブックに貼れる」形で書きます。

---

0. 前提：DP 構造変化アルゴリズム（再掲）

まずはこれをベースにします（すでに一度出したものですが、ここでセルとして完結させておきます）。

```python
import numpy as np

# ---------- DP 用の前処理と区間 SSR ----------

def ols_segment_SSR_precompute(y, Z):
    """
    y: shape (T,)
    Z: shape (T, q)
    任意区間 [i, j] の SSR を高速に計算するための累積和を構築。
    """
    y = np.asarray(y)
    Z = np.asarray(Z)
    T, q = Z.shape

    Sy = np.zeros(T + 1)             # 累積 y'y
    SZZ = np.zeros((T + 1, q, q))    # 累積 Z'Z
    SZy = np.zeros((T + 1, q))       # 累積 Z'y

    for t in range(T):
        Sy[t+1]    = Sy[t]    + y[t]**2
        SZZ[t+1]   = SZZ[t]   + np.outer(Z[t], Z[t])
        SZy[t+1]   = SZy[t]   + Z[t] * y[t]

    return Sy, SZZ, SZy

def segment_SSR(i, j, Sy, SZZ, SZy, ridge=1e-12):
    """
    区間 [i, j]（両端含む）での回帰 y ~ Z の SSR を計算。
    SSR = y'y - (Z'y)' (Z'Z)^{-1} (Z'y)
    """
    Syy     = Sy[j+1]    - Sy[i]
    SZZ_seg = SZZ[j+1]   - SZZ[i]
    SZy_seg = SZy[j+1]   - SZy[i]

    # 数値安定化用のごく弱いリッジ
    SZZ_reg = SZZ_seg + ridge * np.eye(SZZ_seg.shape[0])

    try:
        beta = np.linalg.solve(SZZ_reg, SZy_seg)
    except np.linalg.LinAlgError:
        beta = np.linalg.lstsq(SZZ_reg, SZy_seg, rcond=None)[0]

    SSR = Syy - SZy_seg @ beta
    return SSR, beta

# ---------- 純構造変化モデルの DP 推定 ----------
```

```python
def fit_pure_structural_breaks(y, Z, m, h):
    """
    純構造変化モデル:
        y_t = z_t' * nu_j + u_t,    t in regime j
    について、Bai–Perron 型 DP で m 個のブレークを推定。

    Parameters
    ----------
    y : (T,)
    Z : (T, q)
    m : int          ブレーク数
    h : int          最小セグメント長

    Returns
    -------
    breakpoints : list[int]
        ブレークインデックス (0-based, その index が「セグメントの終わり」)
    coefs : list[np.ndarray]
        各セグメントの OLS 係数
    SSR_total : float
        最小 SSR
    """
    y = np.asarray(y)
    Z = np.asarray(Z)
    T = len(y)
    K = m + 1  # セグメント数

    if T < K * h:
        raise ValueError("サンプルが短すぎます: T < (m+1)*h")

    # 1. 累積クロスプロダクト
    Sy, SZZ, SZy = ols_segment_SSR_precompute(y, Z)

    # 2. 各区間 [i, j] の SSR を事前計算
    SSR_mat = np.full((T, T), np.inf)
    for i in range(T):
        for j in range(i + h - 1, T):
            SSR_mat[i, j], _ = segment_SSR(i, j, Sy, SZZ, SZy)

    # 3. DP テーブル
    dp    = np.full((K, T), np.inf)    # dp[k, t]: 先頭〜t を k+1 セグメントに分割し
た最小 SSR
    prev = np.full((K, T), -1, int)

    # ベースケース: k = 0 (1セグメント)
    for t in range(h - 1, T):
        if T - (t + 1) < (K - 1) * h:
            continue
```

```
        dp[0, t]    = SSR_mat[0, t]
        prev[0, t] = -1

    # 再帰: k = 1..K-1
    for k in range(1, K):
        for t in range((k + 1) * h - 1, T):
            if T - (t + 1) < (K - 1 - k) * h:
                continue

            best_val = np.inf
            best_s   = -1

            s_min = k * h - 1
            s_max = t - h
            for s in range(s_min, s_max + 1):
                if not np.isfinite(dp[k-1, s]):
                    continue
                val = dp[k-1, s] + SSR_mat[s+1, t]
                if val < best_val:
                    best_val = val
                    best_s   = s

            dp[k, t]    = best_val
            prev[k, t] = best_s

    # 最後のセグメントは T-1 で終了
    last_k   = K - 1
    best_end = T - 1
    best_val = dp[last_k, best_end]
    if not np.isfinite(best_val):
        raise ValueError("フルサンプルをカバーする分割が見つかりません。m や h を見直してください。")

    # バックトラックでブレーク復元
    breakpoints = []
    t = best_end
    for k in range(last_k, 0, -1):
        s = prev[k, t]
        breakpoints.append(s)
        t = s
    breakpoints = sorted(breakpoints)

    # 各セグメントの係数再推定
    coefs = []
    Sy, SZZ, SZy = ols_segment_SSR_precompute(y, Z)  # 念のため再計算
    start = 0
    for bp in breakpoints + [T - 1]:
        _, beta = segment_SSR(start, bp, Sy, SZZ, SZy)
```

```
        coefs.append(beta)
        start = bp + 1

    return breakpoints, coefs, best_val
```

---

1. インフレ系列に Bai–Perron ブレークを入れる

1.1 モデル設定

インフレ率 π_t に対して、単純な「平均シフト」モデルを考えます：

$$\pi_t = \mu_j + u_t, \quad t \in (T_{j-1}, T_j]$$

- 説明変数 Z は定数項のみ：Z_t = [1]
- これを上の DP 関数 `fit_pure_structural_breaks` にそのまま入れると、**インフレ平均のレジーム（高インフレ期 / 低インフレ期 etc.）** が抽出されます。

さらに、**ブレーク数 m を自動選択するために BIC を使った簡易選択**も実装します。

1.2 実装コード

```python
def bic_for_breaks(SSR, T, k_params):
    """
    単純な BIC: T*log(SSR/T) + k_params*log(T)
    k_params: 推定パラメータ数
             純構造変化モデル: (m+1)*q など
    """
    sigma2_hat = SSR / T
    return T * np.log(sigma2_hat) + k_params * np.log(T)

def detect_breaks_inflation(pi, max_breaks=5, h=12):
    """
    インフレ系列 pi に対して Bai-Perron 型 DP でブレークを検出し、
    BIC 最小のブレーク数を選ぶ。

    モデル:
        pi_t = mu_j + u_t    (jごとに平均シフト)
    Parameters
    ----------
    pi : array-like, (T,)
        インフレ率系列
    max_breaks : int
        試す最大ブレーク数 M
    h : int
        最小セグメント長 (たとえば月次なら 12 = 1年 など)

    Returns
    -------
    best_m : int
        BIC によって選ばれたブレーク数
    best_breaks : list[int]
        ブレーク位置 (0-based, index がそのまま使える)
    regimes : ndarray (T,)
        各時点のレジーム番号 0..best_m
    summary : list[dict]
        m=0..max_breaks についての (m, breaks, BIC, SSR, coefs) 情報
    """
    pi = np.asarray(pi)
    T = len(pi)
    Z = np.ones((T, 1))  # 定数のみ
    q = 1                # パラメータ数 per segment

    summary = []

    # m = 0 (ブレークなし)
    Sy, SZZ, SZy = ols_segment_SSR_precompute(pi, Z)
    SSR0, beta0 = segment_SSR(0, T-1, Sy, SZZ, SZy)
    bic0 = bic_for_breaks(SSR0, T, k_params=(1 * q))  # セグメント1つ
```

```
    summary.append({
        "m": 0,
        "breaks": [],
        "SSR": SSR0,
        "BIC": bic0,
        "coefs": [beta0],
    })

    # m = 1..max_breaks
    for m in range(1, max_breaks + 1):
        # (m+1)*q 個の平均パラメータ
        try:
            breaks_m, coefs_m, SSRm = fit_pure_structural_breaks(pi, Z, m=m,
h=h)
        except ValueError:
            # サンプル長・h の制約で不可能なとき
            break
        bic_m = bic_for_breaks(SSRm, T, k_params=((m+1) * q))
        summary.append({
            "m": m,
            "breaks": breaks_m,
            "SSR": SSRm,
            "BIC": bic_m,
            "coefs": coefs_m,
        })

    # BIC 最小を選択
    best = min(summary, key=lambda d: d["BIC"])
    best_m      = best["m"]
    best_breaks = best["breaks"]

    # レジーム番号ラベルを構成
    regimes = np.zeros(T, dtype=int)
    start = 0
    regime_id = 0
    for bp in best_breaks + [T - 1]:
        regimes[start:bp+1] = regime_id
        regime_id += 1
        start = bp + 1

    return best_m, best_breaks, regimes, summary
```

1.3 使い方例

```
# pi_series: pandas.Series など
best_m, breaks, regimes, summary = detect_breaks_inflation(pi_series.values,
                                                            max_breaks=5,
                                                            h=12)

print("選ばれたブレーク数 m* =", best_m)
print("ブレーク位置 (インデックス):", breaks)
print("最初の20期のレジームラベル:", regimes[:20])
```

ここで得られた `regimes` は「インフレレジーム」を表す整数ラベル（0,1,2,...）です。
このレジームをそのまま「マクロの regime」として、次の partial structural change に使います。

---

2. partial structural change（α共通＋βレジーム別）の実装

2.1 モデル設定

セクター i のリターン $r_{i,t}$ とインフレ率 π_t の関係を

$$r_{i,t} = \alpha_i + \beta_{i,j}\,\pi_t + \varepsilon_{i,t}, \quad t \in \text{Regime } j$$

とします。ここでは：

- **α_i はレジーム共通**（partial structural change）
- **β_{i,j} はレジームごとに変わる**

インフレレジーム j は、上で検出した `regimes` を使います（共通ブレーク・共通レジーム）。

2.2 OLS 表現

Regime が K = m+1 個あるとすると、デザイン行列は

- 列0：常に1（共通 α）
- 列1..K：π_t × I(regime_t = j)

とすれば、

$$y = \alpha \cdot 1 + \sum_{j=0}^{K-1} \beta_j (\pi_t \cdot I_{t,j})$$

という線形モデルになります。

2.3 実装コード（単一セクター）

```python
import numpy as np

def fit_partial_structural_change_sector(pi, r, regimes):
    """
    モデル:
        r_t = α + β_j * pi_t  (t in regime j)
    を OLS で推定（ブレーク位置/レジームは既知とする）。

    Parameters
    ----------
    pi : (T,)
        インフレ率系列
    r : (T,)
        対象セクターのリターン
    regimes : (T,)
        レジームラベル (0..K-1), detect_breaks_inflation で得たもの

    Returns
    -------
    alpha : float
        共通切片
    betas : ndarray shape (K,)
        各レジームのインフレ感応度 β_j
    """
    pi = np.asarray(pi)
    r  = np.asarray(r)
    reg = np.asarray(regimes)
    T = len(r)
    assert len(pi) == T and len(reg) == T

    K = reg.max() + 1  # レジーム数
    # X の列: [1, pi*I(regime=0), pi*I(regime=1), ...]
    X = np.zeros((T, K + 1))
    X[:, 0] = 1.0
    for j in range(K):
        mask = (reg == j)
        X[mask, j+1] = pi[mask]

    # OLS
    beta_hat, *_ = np.linalg.lstsq(X, r, rcond=None)
    alpha = beta_hat[0]
    betas = beta_hat[1:]
    return alpha, betas
```

2.4 複数セクターをまとめて推定

```python
def fit_partial_structural_change_all_sectors(pi, df_sectors, regimes):
    """
    複数セクターに対して一括で
        r_{i,t} = α_i + β_{i,j} * pi_t
    を推定するヘルパー。

    Parameters
    ----------
    pi : (T,)
    df_sectors : pandas.DataFrame
        各列がセクターリターン
    regimes : (T,)

    Returns
    -------
    results : dict
        {sector_name: {"alpha": α_i, "betas": β_i_vec}} の形式
    """
    import pandas as pd
    assert len(pi) == len(df_sectors)

    results = {}
    for sec in df_sectors.columns:
        r = df_sectors[sec].values
        alpha_i, betas_i = fit_partial_structural_change_sector(pi, r, regimes)
        results[sec] = {
            "alpha": alpha_i,
            "betas": betas_i
        }
    return results
```

2.5 使い方例

```
# 例: df_infl = df["inflation"], df_sectors = df[["XLE","XLF","XLK","XLU"]]

pi_arr = df_infl.values
best_m, breaks, regimes, summary = detect_breaks_inflation(pi_arr,
                                                           max_breaks=5,
                                                           h=12)


print("選ばれたインフレレジーム数:", best_m+1)
print("ブレーク index:", breaks)

results_partial = fit_partial_structural_change_all_sectors(
    pi_arr,
    df_sectors=df[["XLE","XLF","XLK","XLU"]],
    regimes=regimes
)

for sec, par in results_partial.items():
    print(f"\n=== {sec} ===")
    print("alpha:", par["alpha"])
    print("betas (regimeごと):", par["betas"])
```

ここで得られるものは：

- 各セクター i ごとに
  - `alpha_i`：マクロ共通の平均（レジーム非依存）
  - `betas_i[j]`：インフレレジーム j におけるインフレ感応度

---

3. この partial モデルをポートフォリオ運用に組み込む

上で推定したパラメータに基づき、時点 t における **レジーム別期待リターン**を計算してポートフォリオを組めます。

3.1 レジーム別期待リターン

「時点 t のインフレ π_t とレジームラベル reg_t」がわかれば

$$E[r_{i,t+1}|\pi_t, reg_t = j] \approx \alpha_i + \beta_{i,j}\pi_t$$

とおいて、これを μ_i として使います。

```python
def expected_returns_partial(pi_t, reg_t, partial_params, sector_list):
    """
    partial structural change パラメータから、時点 t の
    セクター期待リターンベクトル μ_t を計算。

    partial_params: fit_partial_structural_change_all_sectors の結果
    sector_list: セクター名のリスト
    """
    mu = []
    for sec in sector_list:
        par = partial_params[sec]
        alpha = par["alpha"]
        betas = par["betas"]
        beta_reg = betas[reg_t]
        mu_i = alpha + beta_reg * pi_t
        mu.append(mu_i)
    return np.array(mu)
```

3.2 ローリング・バックテストの雛形

インフレレジームは「detect_breaks_inflation を全期間で1回推定」として、
ウィンドウ共分散＋ partial モデルでMV最適化を行う簡易版の例です：

```python
import pandas as pd

def mean_variance_weights(mu, Sigma):
    Sigma_inv = np.linalg.inv(Sigma)
    w_raw = Sigma_inv @ mu
    if np.allclose(w_raw, 0):
        return np.ones_like(w_raw) / len(w_raw)
    return w_raw / np.sum(np.abs(w_raw))

def backtest_partial_structural(pi_series, df_sectors, regimes,
                                window=120, sector_list=None):
    """
    partial structural change モデルを用いた簡易バックテスト。

    - インフレレジーム・パラメータは全期間で固定
    - 共分散は過去 window で更新
    - 各月 t で:
        μ_t = α_i + β_{i,reg_t} * π_t
        Σ_t = Cov_{t-window..t-1}(r)
        w_t = MV(μ_t, Σ_t)
    """
    if sector_list is None:
        sector_list = list(df_sectors.columns)

    pi_arr = pi_series.values
    regimes_arr = np.asarray(regimes)
    T = len(pi_arr)

    partial_params = fit_partial_structural_change_all_sectors(
        pi_arr, df_sectors[sector_list], regimes_arr
    )

    dates = df_sectors.index
    n = len(df_sectors)

    port_rets = []
    port_dates = []
    weights_hist = []

    for t in range(window, n-1):
        # 共分散 (t-window .. t-1)
        rets_hist = df_sectors[sector_list].iloc[t-window:t]
        Sigma_t = rets_hist.cov().values

        # 時点 t のインフレとレジーム
        pi_t = pi_arr[t]
        reg_t = regimes_arr[t]
```

```python
        # partial モデルによる μ_t
        mu_t = expected_returns_partial(pi_t, reg_t,
                                        partial_params,
                                        sector_list)

        # MV 最適ウェイト
        w_t = mean_variance_weights(mu_t, Sigma_t)
        weights_hist.append(w_t)

        # 翌月 t+1 リターン
        r_next = df_sectors[sector_list].iloc[t+1].values
        port_ret = np.dot(w_t, r_next)
        port_rets.append(port_ret)
        port_dates.append(dates[t+1])

    bt = pd.Series(port_rets, index=port_dates, name="port_ret_partial")
    return bt, np.array(weights_hist), partial_params
```

使い方：

```python
# df: [inflation, XLE, XLF, XLK, XLU] が入った DataFrame と仮定
pi_series  = df["inflation"]
df_sectors = df[["XLE","XLF","XLK","XLU"]]

# ① インフレレジーム推定
best_m, breaks, regimes, summary = detect_breaks_inflation(pi_series.values,
                                                           max_breaks=5,
                                                           h=12)

# ② partial structural change + バックテスト
bt_partial, w_hist, params_partial = backtest_partial_structural(
    pi_series, df_sectors, regimes, window=120
)

# 比較用に等重みなどと一緒にプロットも可能
ew = df_sectors.mean(axis=1)
ew_bt = ew.loc[bt_partial.index]

import matplotlib.pyplot as plt

pd.concat([bt_partial, ew_bt], axis=1).cumsum().plot(
    title="Cumulative Returns: Partial Structural Change vs Equal Weight"
)
plt.show()
```

まとめ

- **ステップ1**：インフレ系列そのものに Bai–Perron 型 DP を適用し、
  「高インフレ期／低インフレ期」などの **インフレレジーム** を抽出しました。

- **ステップ2**：そのインフレレジームを使って、

$$r_{i,t} = \alpha_i + \beta_{i,j}\pi_t$$

  という **partial structural change（α 共通＋β レジーム別）モデル** を OLS で推定し、
  得られたレジーム別 β を使ってセクターポートフォリオ運用を行うバックテスト枠組みを実装しました。

この枠組みをベースに、

- レジームごとの λ（リスク許容度）を変える
- total risk を一定に保つようなターゲット・ボラティリティ制御
- FTRL や RL によるレジーム別ポリシー最適化

など、かなりいろいろ遊べると思います。

必要であれば、

- このコードをクラス構造（`InflationRegimeModel`, `PartialSCPortfolio` など）に整理
- 学会用スライドや論文の「モデル・推定・バックテスト」図を描くコード

もまとめて書きます。

Response:

思考時間: 7s

まず結論から言うと、これまであなたと一緒にやってきた実装は、大きく分けて次の 4 レイヤーに分かれています：

1. **Bai–Perron 型「構造変化（複数ブレーク）」の数値実装**
2. **supF / sequential procedure によるブレーク数の検定ロジック**
3. **閾値モデル・Markov switching モデルなど、マクロ・レジームの推定**
4. **インフレ×セクター回帰（レジーム別 β）を用いたセクターポートフォリオ運用**

以下、数式とコード対応を整理しながら、一つずつ説明します。

1. Bai–Perron 型「純構造変化モデル」の実装

1.1 モデル

最も基本となるモデルは「純構造変化モデル」です：

$$y_t = z_t' \nu_j + u_t, \quad t = T_{j-1}+1, \dots, T_j, \quad j = 1, \dots, m+1$$

- y_t ：スカラー応答変数
- z_t ： q 次元の説明変数ベクトル
- $\nu_j \in \mathbb{R}^q$ ：レジーム j に固有の回帰係数
- ブレーク点 $T_1 < \cdots < T_m$ （未知）
- 各セグメントの範囲は $(T_{j-1}, T_j]$ 、 $T_0 = 0, T_{m+1} = T$

このとき、「どこにブレークを入れるか」によって残差平方和（SSR）が変わります。
目的はブレーク位置と係数を選んで、全セグメントの SSR 合計を最小化することです：

$$\min_{T_1,\dots,T_m,\{\nu_j\}} \sum_{j=1}^{m+1} \sum_{t=T_{j-1}+1}^{T_j} (y_t - z_t' \nu_j)^2.$$

1.2 区間ごとの OLS と SSR の前計算

ある区間 $[i, j]$ について OLS をすると、

- 区間内の Z 行列と y ベクトルを

$$Z_{i:j} = \begin{bmatrix} z_i' \\ \vdots \\ z_j' \end{bmatrix}, \quad y_{i:j} = \begin{bmatrix} y_i \\ \vdots \\ y_j \end{bmatrix}$$

- OLS 解は

$$\hat{\nu}_{i,j} = (Z_{i:j}' Z_{i:j})^{-1} Z_{i:j}' y_{i:j}$$

このとき SSR は

$$\mathrm{SSR}(i, j) = y_{i:j}' y_{i:j} - (Z_{i:j}' y_{i:j})' (Z_{i:j}' Z_{i:j})^{-1} (Z_{i:j}' y_{i:j}).$$

コードでは、これを高速に計算するために**累積クロスプロダクト**を使っています：

```
Sy[t+1]   = Sy[t]   + y[t]**2
SZZ[t+1]  = SZZ[t]  + np.outer(Z[t], Z[t])
SZy[t+1]  = SZy[t]  + Z[t] * y[t]
```

区間 [i,j] に対する合計は差分で表現できます：

$$\begin{aligned} S_{yy}(i,j) &= Sy[j+1] - Sy[i], \\ S_{ZZ}(i,j) &= SZZ[j+1] - SZZ[i], \\ S_{Zy}(i,j) &= SZy[j+1] - SZy[i]. \end{aligned}$$

これをそのまま SSR の式に代入する関数が `segment_SSR(i, j, Sy, SZZ, SZy)` です。

1.3 動的計画法（Dynamic Programming, DP）

全時点 $1, \dots, T$ を m 個のブレークで $m+1$ セグメントに分けるとき、
全体の SSR は各セグメントの $\mathrm{SSR}(i,j)$ の和になります。

しかし全探索すると組み合わせ数が爆発するため、
Bai–Perron は**動的計画法で O(T²)** まで落とします。

DP の再帰は概念的には次のようになります：

- $C(k,t)$ ：先頭から時点 t までを **k+1 セグメント**に分割したときの最小 SSR
- 最後のセグメント終端が t のとき、その一つ前のセグメントの終端を s とすると

$$C(k,t) = \min_s \{C(k-1,s) + \mathrm{SSR}(s+1,t)\}$$

境界条件は

- $k=0$ （1セグメント）のとき

$$C(0,t) = \mathrm{SSR}(0,t).$$

コードでは `dp[k, t]` が $C(k,t)$ に対応し、
`prev[k, t]` が最適な s（＝一つ前のセグメントの終端）を覚えているテーブルです。

最終的に `dp[m, T-1]` が「m 個のブレークを持つときの最小 SSR」であり、
`prev` を後ろから辿ることでブレーク位置（インデックス）を復元しています。

さらに、各セグメントで再度 OLS をかけ直して $\hat{\nu}_j$ を求めるのが `fit_pure_structural_breaks` の最後の部分です。

---

2. supF 検定と sequential procedure

2.1 supF(0 vs m) 検定

構造変化があるかどうかを検定するために、
「ブレークなし vs m ブレーク」の F 検定を行いました。

モデル

- H0（制約あり）：ブレークなし

$$y_t = z_t' \nu + u_t \quad (\nu \text{は全期間共通})$$

  → SSR_0、パラメータ数 q

- H1（制約なし）：m ブレーク

$$y_t = z_t' \nu_j + u_t, \quad j = 1, \ldots, m+1$$

  → SSR_m、パラメータ数 $(m+1)q$

クラシカルな F 検定の形で

$$F_{\text{supF}(m)} = \frac{(SSR_0 - SSR_m)/(mq)}{SSR_m/(T-(m+1)q)}.$$

ここで重要なのは、SSR_m が**ブレーク位置について最適化された値**であることです。
これを DP で求めているので、統計量は「supF」と呼ばれます。

コードでは `supF_0_vs_m(y, Z, m, h)` が以下を返します：

- `F_stat`：上式の F 値
- `SSR0`：ブレークなしの SSR
- `SSRm`：m ブレークの最小 SSR
- `breakpoints_m`：そのときのブレーク位置

2.2 supF(ℓ+1 | ℓ) と sequential procedure

ブレーク数 m が未知のとき、
Bai–Perron の推奨は「逐次的（sequential）」な検定です。

1. まず supF(1) で **ブレークが全く無いか**をチェック
2. 有意なら 1 ブレークモデルを採用し、
   各セグメント内でさらに「もう1つブレークを追加できるか？」を調べる
   → supF(2|1)
3. supF(ℓ+1|ℓ) が有意な限り ℓ を増やし、
   はじめて有意でなくなった ℓ を最終的なブレーク数とする。

supF(ℓ+1|ℓ) の計算の中身

- まず「ℓ ブレークモデル」を DP で推定し、
  セグメント (s_j, e_j) が得られます。
- その**各セグメント内**で「0 ブレーク vs 1 ブレーク」の supF を計算： セグメント長 n_j の中で

$$F^{(j)} = \frac{(SSR_{0,j} - SSR_{1,j})/q}{SSR_{1,j}/(n_j - 2q)}$$

- その最大値

$$\text{supF}(\ell + 1|\ell) = \max_j F^{(j)}$$

  を採用。

コードでは

- `supF_1_within_segment(y_seg, Z_seg, h)`
  → 個々のセグメント内で supF(1) を計算
- `sequential_break_selection(y, Z, max_breaks, h)`
  → ℓ=0,1,… に対して
  - ℓブレークモデル（DP）
  - 各セグメント内 supF(1)
  - supF(ℓ+1|ℓ) と「どこに次のブレークを入れるべきか」候補

を返すようにしています。
実際に「どこで止めるか」は、この supF(ℓ+1|ℓ) を Bai–Perron の臨界値表と比較して判断する、という設計になっています。

3. 閾値モデル・Markov switching・インフレレジーム

3.1 閾値モデル（threshold regression）

インフレ率 π_t とセクターリターン $r_{i,t}$ の関係を**非線形に**捉えるため、

$$r_t = \begin{cases} \alpha_L + \beta_L \pi_t + \varepsilon_t, & \pi_t \le \gamma \\ \alpha_H + \beta_H \pi_t + \varepsilon_t, & \pi_t > \gamma \end{cases}$$

のような**1閾値モデル**を実装しました。

推定の仕方

1. 閾値候補 γ を、インフレ率のソート済み系列から選ぶ
（最低サンプル数 h を保証するように）。

2. 各候補 γ について

    - Low regime: $\{t : \pi_t \le \gamma\}$ で OLS → $\hat{\alpha}_L, \hat{\beta}_L$
    - High regime: $\{t : \pi_t > \gamma\}$ で OLS → $\hat{\alpha}_H, \hat{\beta}_H$

3. 2つの OLS の SSR 合計を

$$SSR(\gamma) = SSR_L(\gamma) + SSR_H(\gamma)$$

   とし、γ を走査して SSR 最小の閾値 $\hat{\gamma}$ を求める。

これをコード化したのが `fit_threshold_regression_1d(y, x, h)` で、

- `y` : セクターリターン
- `x` : インフレ率 π_t
- `h` : 各レジーム最低サンプル数

を入力として、

- 最適閾値 $\hat{\gamma}$
- パラメータ $\hat{\alpha}_L, \hat{\beta}_L, \hat{\alpha}_H, \hat{\beta}_H$

を返します。

3.2 Markov switching（確率的レジーム）

より滑らかにレジームを表現するため、インフレ率 π_t に対して
マルコフスイッチングモデル（MSモデル）をあてはめました。

単純な例として、2レジームの平均付きモデル：

$$\pi_t = \mu_{S_t} + \varepsilon_t, \quad S_t \in \{0, 1\}$$

- S_t ：時点 t のレジーム（低インフレ / 高インフレ）
- S_t はマルコフ連鎖：

$$P(S_t = j | S_{t-1} = i) = p_{ij}$$

`statsmodels.tsa.MarkovRegression` を使い、

- k_regimes=2
- trend='c'（レジームごとに定数）
- switching_variance=True

などを指定して EM で推定。
その結果として、各時点 t ごとに

$$P(S_t = 0 | \text{data}), \quad P(S_t = 1 | \text{data})$$

という**平滑化されたレジーム確率（smoothed probabilities）**が得られます。
これを `df["p_regime0"], df["p_regime1"]` として保存しました。

3.3 weighted regression による「レジーム別 β」

レジーム確率を用いて、セクターリターン $r_{i,t}$ とインフレ π_t の関係を
重みつき回帰で推定しました。

レジーム j に対するパラメータ $(\alpha_{i,j}, \beta_{i,j})$ を

$$\min_{\alpha_{i,j}, \beta_{i,j}} \sum_t p_{j,t} \left(r_{i,t} - \alpha_{i,j} - \beta_{i,j} \pi_t\right)^2$$

として求めると、
これは「サンプル重み $w_t = p_{j,t}$ を使った OLS」と等価です。

実装は `weighted_regression(y, x, w)` で、

- `y` : セクターリターン
- `x` : インフレ
- `w` : レジーム j の確率 $p_{j,t}$

として LinearRegression の `sample_weight=w` で推定しています。

これによって、各レジーム j ごとに $(\hat{\alpha}_{i,j}, \hat{\beta}_{i,j})$ が得られ、
「低インフレ期／高インフレ期でのインフレ感応度」が推定されます。

---

4. インフレレジーム×セクター回帰による運用ロジック

最後に、これらのモデルを用いて**ポートフォリオ運用**を組み立てました。

4.1 閾値モデルを用いた期待リターン

単純な 1 閾値モデルでは、時点 t のインフレ π_t に応じて

$$E[r_{i,t+1}|\pi_t] \approx \begin{cases} \alpha_{i,L} + \beta_{i,L}\pi_t, & \pi_t \le \hat{\gamma}_i \\ \alpha_{i,H} + \beta_{i,H}\pi_t, & \pi_t > \hat{\gamma}_i \end{cases}$$

という形で**セクター i の期待リターン**を定義しました。

コードでは

```
def expected_return_threshold(pi_t, gamma, pars):
    if pi_t > gamma:
        return pars["alpha_high"] + pars["beta_high"] * pi_t
    else:
        return pars["alpha_low"] + pars["beta_low"] * pi_t
```

という形で実装し、各時点で μ ベクトルとしてまとめています。

4.2 Markov switching 混合型の期待リターン

MS モデルの場合は、レジーム確率で混合した期待値：

$$\mu_i(t) = \sum_{j=0}^{1} P(S_t = j|\text{data})\left(\alpha_{i,j} + \beta_{i,j}\pi_t\right).$$

これをコード化したのが `mu_regime_mixture` で、
`p_regime0, p_regime1` と `regime_models[sec][j]["alpha"], ["beta"]` を用いて計算しています。

4.3 Bai–Perron によるインフレレジーム + partial structural change

さらに進んだステップとして、

1. **インフレ系列 π_t 自体に Bai–Perron DP を適用**し、
   平均シフトモデル

   $$\pi_t = \mu_j + u_t, \quad t \in \text{Regime } j$$

   でブレーク位置とレジーム番号 `regimes` を推定しました。

   - 定数のみの回帰、Z_t = [1]
   - m=0..M について DP で SSR_m を求め、
     BIC

     $$\text{BIC} = T\log(SSR/T) + k\log T$$

     が最小の m を選択。
   - これが `detect_breaks_inflation` です。

2. そのレジームラベル `regimes_t = 0,1,…` を用いて、セクター i について

   $$r_{i,t} = \alpha_i + \beta_{i,j}\pi_t + \varepsilon_{i,t}, \quad t \in \text{Regime } j$$

   という **partial structural change** モデルを OLS で推定しました。

   - デザイン行列 X は

     $$X_t = (1,\ \pi_t 1\{reg_t = 0\},\ \pi_t 1\{reg_t = 1\}, \dots)$$

   - OLS 解

$$\hat{\theta} = (\hat{\alpha}_i, \hat{\beta}_{i,0}, \dots, \hat{\beta}_{i,K-1})$$

を求める関数が `fit_partial_structural_change_sector` です。
これを全セクターに対して適用するのが
`fit_partial_structural_change_all_sectors`。

3. この partial モデルを用いて、時点 t のインフレ π_t と
レジーム `reg_t` に基づく**セクター期待リターン**：

$$\mu_i(t) = \alpha_i + \beta_{i,reg_t} \cdot \pi_t$$

を計算し、MV 最適化の目的関数に入れました。

4.4 平均分散最適化（Markowitz）の部分

期待リターンベクトル $\mu_t \in \mathbb{R}^N$ 、
共分散行列 $\Sigma_t \in \mathbb{R}^{N\times N}$ が与えられたとき、
単純な平均分散モデルでは

$$\max_w \; w'\mu_t - \frac{\lambda}{2} w'\Sigma_t w \quad \text{s.t.} \; \sum_i w_i = 1$$

という問題を解きます。
解析的には、ラグランジュ乗数を使うと

```
w^\* \propto \Sigma_t^{-1} \mu_t
```

という形の解が得られます（本当は制約つきで正規化が必要ですが、形として）。

コードでは

```
Sigma_inv = np.linalg.inv(Sigma)
w_raw = Sigma_inv @ mu
w = w_raw / np.sum(np.abs(w_raw))
```

とし、「L1 ノルムで正規化」することで

- 比率の比は平均分散理論に対応
- 合計は 1（絶対値基準）になるようにスケール調整

という簡易な実装を採用しました。

共分散 Σ_t は、過去 `window` 期間（例：120ヶ月）の実現リターンから

$$\hat{\Sigma}_t = \widehat{\mathrm{Cov}}(r_{t-window}, \ldots, r_{t-1})$$

として推定しています。

4.5 バックテストの流れ

バックテストは概ね次のようなステップで構成しました：

1. t = window, …, T-2 についてループ
2. 推定ウィンドウ（t-window .. t-1）から
   - 共分散 Σ_t
   - （必要なら）閾値モデルや partial モデルのパラメータ
3. 時点 t で利用可能なインフレ情報（ π_t とレジーム）から
   - 閾値／partial／MS モデルにより期待リターン μ_t を計算
4. (μ_t, Σ_t) をもとに MV 最適化 → 重み w_t
5. 翌期 t+1 の実現リターン r_{t+1} を用いて
   ポートフォリオリターン $w_t' r_{t+1}$ を記録
6. 累積和（累積リターン）を取ることで
   等重みポートフォリオ等と比較し、
   「レジーム依存 α,β を利用した戦略がどの程度アウトパフォームするか」を評価

という構造です。

もし、

- 数式をもう少し「論文の節立て（モデル・推定・検定・運用戦略）」に合わせて整理したい
- 今のコードをモジュール（class ベース）にして `inflation_regime.py`, `sector_model.py`, `portfolio_backtest.py` みたいに分割したい
- スライド用に図（インフレレジーム推定結果、β のレジーム別棒グラフ、パフォーマンス曲線）を描く Python コードもほしい

といったニーズがあれば、その前提で「論文草稿 or スライド草案」レベルまでまとめることもできます。
