Breakeven Inflation (BEI) を Affine Term Structure Model で寄与分解する完全ガイド

最終更新 2025-07-28

---

0. 目次

---

1. イントロダクション

Breakeven Inflation (BEI) とは、名目国債の利回り \$(y^N)\$ と実質国債 (TIPS) の利回り \$(y^R)\$ のスプレッド: $BEI_t^{(n)} = y_t^N(n) \;-\; y_t^R(n)$. 市場はしばしば BEI を「期待インフレ率」と同義に扱いますが、実際には期待インフレ以外にも **リスク補償** や **流動性プレミアム** が混在しているため、**寄与分解** が不可欠です。

---

2. 寄与分解の理論的枠組み

2.1 4 項分解の数式

$$ BEI_t^{(n)} \;=\; \underbrace{\mathbb E_t^{P}[\pi_{t,t+n}]}{\text{期待インフレ}} \;+\; \underbrace{\bigl(\mathbb E_t^{Q}[\pi}] - \mathbb E_t^{P}[\pi_{t,t+n}]\bigr)}{\text{インフレ・リスクプレミアム (IRP)}} \;+\; \underbrace{LP_t^{(n)}} \;+\; \underbrace{Conv/ME_t^{(n)}}_{\text{コンベクシティほか}} $$}

項	意味	主な要因	モデル内での抽出方法
期待インフレ	物理測度 \$P\$ 下の将来インフレ期待	マクロ見通し, 中銀目標	模型状態 \$X_t\$ から計算 (セミ エン-ドジェナス)
IRP	リスク中立 \$Q\$ と \$P\$ の期待差	インフレリスクの価格付け	\$\lambda_t\$ (リスク価格) を推定し \$Q\$-期待を算出

項	意味	主な要因	モデル内での抽出方法
LP	TIPS 特有の流動性劣位	発行額, オークション周期, マーケットデプス	独立因子 Z_t or 観測ノイズを介して識別
Conv/ME	インデックスラグやクーポン構造由来	CPI ラグ, クーポン複利, 測定誤差	測定誤差共分散 R に吸収

2.2 リスク測度の役割

- **P 測度**: 実現確率での期待 → 時系列ダイナミクス (μ,Φ) に依存
- **Q 測度**: 債券価格を決定 → リスク価格 (λ_0,Λ) に依存\ ATSM は **同一の状態変数で P・Q をリンク** するため、$\text{IRP}=E^Q-E^P$ を内部一貫的に算出できる点が強みです。

---

3. Affine TSM を使う利点

1. **閉形式 Riccati** により多満期価格を一括計算できる。
2. **リスク価格がアフィン**: $\lambda_t=\lambda_0+\Lambda X_t$ により IRP をパラメトリックに識別。
3. **状態空間化**してカルマンフィルタ推定が可能 → 高頻度データ・測定誤差を自然に扱える。
4. **拡張容易**: 流動性因子, CPI ラグ, ジャンプ拡張などをモジュラーに追加できる。

---

4. モデル仕様（再掲＋補足）

- **金利側**: 3 因子ガウス (X_t) — レベル, スロープ, 曲率。
- **流動性側**: 1 因子 (Z_t) — AR(1) で実質利回り観測式みに効く。
- **CPI**: $\pi_{t+1}=\kappa_0+\kappa_1'X_t+u_{t+1}$。
- **3 か月ラグ**: 実質債の効果的満期 $n_{\text{eff}}=n-0.25$ 年で価格付け。
- **名目短期金利**: $r_t^N=\delta_0^N+\delta_1^{N'}X_t$。
- **実質短期金利**: $r_t^R=\delta_0^R+\delta_1^{R'}X_t$ （流動性は価格ではなく観測式に追加）。

---

5. 識別と推定のメカニズム

寄与	モデルパラメータ	識別キー	推定ステップ
期待インフレ	$(\kappa_0,\kappa_1,\mu,\Phi)$	CPI 観測行	カルマンフィルタで状態 X_t を抽出後、$E^P[\pi]$ 計算
IRP	(λ_0,Λ)	名目 & 実質利回り曲線 の形状差	Riccati + 債券価格適合で最尤推定し $E^Q[\pi]$ 生成
LP	(Z_t,ρ_Z,σ_Z)	名目 vs 実質の残差パターン	観測式に $H Z_t$ を入れフィルタリング
Conv/ME	R 行列	満期特有の誤差構造	最尤が暗黙に推定、残差分解で解釈

6. データ要件 & 前処理

1. **名目曲線**: 米財務省 STRIPS or Svensson フィットのゼロクーポン利回り。
2. **実質曲線**: BLS 公表 TIPS データをブートストラップ。
3. **CPI**: CPI All-Urban NSA, ラグ補間は「公表直近値を 3 か月先までキャリー」。
4. **観測頻度の整合**: 月末 or 週次スナップで揃える。
5. **先にキャッシュフローとクリーン価格からゼロカーブ化**し、名目・実質とも利回り単位に変換。

7. 実装詳細

- **Riccati 再帰**: 離散ガウス版 (Duffie–Kan)。
- **カルマンフィルタ**: 4 次元状態, 観測ノイズ対角。
- **MLE**: `scipy.optimize.minimize` (L-BFGS-B)、対数バリアで正値制約。
- **指数化ラグ**: `eff_idx()` で満期を月グリッド上シフト。
- **期待インフレ**: `exp_inf()` でラグ調整後の \$n_{\text{eff}}\$ を使用。

8. 結果の読み方

- **期待インフレ**: 通常 BEI より滑らかで低ボラ。
- **IRP**: 景気悪化でプラス (インフレ回避需要増)・好景気でマイナスになる傾向。
- **LP**: リーマン期や TIPS 発行減の時に急上昇。満期ベースで逆ハンプ型をとる場合が多い。
- **Conv/ME**: CPI ラグ & クーポンの影響で 5 y 付近が最も大きいとの実証例が多数。

9. 限界と留意点

- ガウス仮定 → 極度の tail リスクを過小評価。
- 流動性因子を 1 本にするとパンデミック期のセクター歪みを十分捕捉できない可能性。
- CPI ラグ以外にも **Indexation floor** (CPI ≤ 0) や税調整が BEI に影響。

10. Python フルコード

（前回提示のテンプレートをそのまま貼付、必要ならローカルで `pip install numpy scipy dataclasses`）

```
# -*- coding: utf-8 -*-
"""Breakeven Inflation decomposition via 3-factor ATSM with liquidity premium and CPI 3-month lag"""

from __future__ import annotations
import numpy as np
from numpy.linalg import inv
from dataclasses import dataclass
from typing import Sequence, Tuple, Dict
```

```python
from scipy.optimize import minimize

# ---- Global constants ----
STEP = 1/12   # Riccati grid step (1 month)
LAG  = 0.25   # Indexation lag in years (3 months)


def eff_idx(m_year: float, step: float = STEP, lag: float = LAG) -> int:
  """Effective maturity index on the monthly Riccati grid."""
    m_eff = max(m_year - lag, step)
    return int(round(m_eff / step))


# ---- Parameter container ----
@dataclass
class Params:
    # State transition (P-measure)
    mu: np.ndarray             # (3,1)
    Phi: np.ndarray            # (3,3)
    Sigma: np.ndarray          # (3,3)

    # Liquidity factor
    rho_Z: float
    sig_Z: float

    # Risk prices (affine)
    lambda0: np.ndarray        # (3,1)
    Lambda: np.ndarray         # (3,3)

    # Short-rate loadings
    delta0_N: float
    delta1_N: np.ndarray       # (3,1)
    delta0_R: float
    delta1_R: np.ndarray       # (3,1)

    # Inflation dynamics
    kappa0: float
    kappa1: np.ndarray         # (3,1)
    sig_pi: float

    # Measurement noise diagonal
    R_diag: np.ndarray         # (Ny,)

    # Data dims & grids
    Ny: int
    matur_N: Sequence[int]
    matur_R: Sequence[int]
    n_max: int
    index_lag: float = LAG

    # ----- pack / unpack utilities -----
```

```python
def pack(self) -> np.ndarray:
    parts = [
        self.mu.flatten(),
        self.Phi.flatten(),
        self.Sigma[np.tril_indices(3)],
        self.lambda0.flatten(),
        self.Lambda[np.tril_indices(3)],
        self.delta1_N.flatten(),
        self.delta1_R.flatten(),
        self.kappa1.flatten(),
        np.array([
            self.rho_Z,
            np.log(self.sig_Z),
            self.delta0_N,
            self.delta0_R,
            self.kappa0,
            np.log(self.sig_pi)
        ]),
        np.log(self.R_diag)
    ]
    return np.concatenate(parts)

@staticmethod
def unpack(theta: np.ndarray, tmpl: 'Params') -> 'Params':
    i = 0
    def take(n):
        nonlocal i
        out = theta[i:i+n]
        i += n
        return out

    mu    = take(3).reshape(3,1)
    Phi   = take(9).reshape(3,3)
    Sigma = np.zeros((3,3)); Sigma[np.tril_indices(3)] = take(6)
    lambda0 = take(3).reshape(3,1)
    Lambda  = np.zeros((3,3)); Lambda[np.tril_indices(3)] = take(6)
    delta1_N = take(3).reshape(3,1)
    delta1_R = take(3).reshape(3,1)
    kappa1    = take(3).reshape(3,1)
    rest      = take(6)
    rho_Z, ln_sigZ, delta0_N, delta0_R, kappa0, ln_sigpi = rest
    R_diag = np.exp(take(tmpl.Ny))

    return Params(
        mu, Phi, Sigma,
        rho_Z, np.exp(ln_sigZ),
        lambda0, Lambda,
        delta0_N, delta1_N, delta0_R, delta1_R,
        kappa0, kappa1, np.exp(ln_sigpi),
        R_diag,
        tmpl.Ny, tmpl.matur_N, tmpl.matur_R, tmpl.n_max, tmpl.index_lag
```

```
        )


# ---- Riccati recursion ----
def riccati(params: Params, n_max: int, nominal: bool = True) -> Tuple[np.ndarray,
np.ndarray]:
    Phi = params.Phi
    SigSig = params.Sigma @ params.Sigma.T
    lambda0, Lambda = params.lambda0, params.Lambda
    Phi_Q = Phi - SigSig @ Lambda
    mu_Q  = params.mu - SigSig @ lambda0

    if nominal:
        delta0, delta1 = params.delta0_N, params.delta1_N
    else:
        delta0, delta1 = params.delta0_R, params.delta1_R

    A = np.zeros(n_max+1)
    B = np.zeros((n_max+1, 3))

    for n in range(1, n_max+1):
        A[n] = A[n-1] + B[n-1] @ mu_Q + 0.5 * B[n-1] @ SigSig @ B[n-1].T - delta0
        B[n] = B[n-1] @ Phi_Q - delta1.T
    return A, B


# ---- Measurement matrices ----
def build_measurement(params: Params, A_N, B_N, A_R, B_R):
    c, D, H = [], [], []

    # Nominal yields
    for n in params.matur_N:
        idx = int(n / STEP)
        c.append(-A_N[idx] / n)
        D.append(-B_N[idx] / n)
        H.append(np.zeros(1))

    # Real yields with lag
    for m in params.matur_R:
        idx = eff_idx(m)
        c.append(-A_R[idx] / m)
        D.append(-B_R[idx] / m)
        H.append(np.array([1.0]))

    # Inflation observation
    c.append(params.kappa0)
    D.append(params.kappa1.flatten())
    H.append(np.zeros(1))

    return np.array(c).reshape(-1,1), np.vstack(D), np.vstack(H)
```

```python
# ---- Kalman filter log-likelihood ----
def kalman_ll(Y: np.ndarray, p: Params) -> float:
    T, Ny = Y.shape
    Kx, Kz = 3, 1
    K = Kx + Kz

    # Riccati precompute
    A_N, B_N = riccati(p, p.n_max, True)
    A_R, B_R = riccati(p, p.n_max, False)
    c, D, H = build_measurement(p, A_N, B_N, A_R, B_R)
    C = np.hstack([D, H])
    R = np.diag(p.R_diag)

    # Transition matrices
    Phi_big = np.zeros((K,K)); Phi_big[:3,:3] = p.Phi; Phi_big[3,3] = p.rho_Z
    Sig_big = np.zeros((K,K)); Sig_big[:3,:3] = p.Sigma; Sig_big[3,3] = p.sig_Z
    mu_big  = np.zeros((K,1)); mu_big[:3,0] = p.mu.flatten()

    x, P = np.zeros((K,1)), np.eye(K)*0.1
    ll = 0.0
    for t in range(T):
        x_pred = mu_big + Phi_big @ x
        P_pred = Phi_big @ P @ Phi_big.T + Sig_big @ Sig_big.T
        y_pred = c + C @ x_pred
        S = C @ P_pred @ C.T + R
        K_gain = P_pred @ C.T @ inv(S)
        innov = Y[t].reshape(-1,1) - y_pred
        x = x_pred + K_gain @ innov
        P = (np.eye(K) - K_gain @ C) @ P_pred
        sign, logdet = np.linalg.slogdet(S)
        ll += -0.5*(logdet + innov.T @ inv(S) @ innov + Ny*np.log(2*np.pi))
    return float(-ll)


# ---- MLE wrapper ----
def fit(Y: np.ndarray, matur_N: Sequence[int], matur_R: Sequence[int], init: Params) -> Params:
    init.Ny = Y.shape[1]
    init.matur_N, init.matur_R = matur_N, matur_R
    init.n_max = int(max(max(matur_N), max(matur_R)) / STEP) + 12

    th0 = init.pack()
    obj = lambda th: kalman_ll(Y, Params.unpack(th, init))
    res = minimize(obj, th0, method='L-BFGS-B', options={'maxiter':500,'disp':True})
    print(res.message)
    return Params.unpack(res.x, init)


# ---- Expected inflation under P (with lag) ----
def exp_inf(p: Params, X: np.ndarray, n_year: float) -> float:
    n_eff = max(n_year - p.index_lag, STEP)
```

````
    steps = int(round(n_eff / STEP))
    I = np.eye(3); Phi = p.Phi; mu = p.mu
    Phi_power = I
    Ex_sum = np.zeros((3,1))
    for j in range(steps):
        if j>0:
            Phi_power = Phi_power @ Phi
        Ex_j = Phi_power @ X + (I - Phi_power) @ inv(I - Phi) @ mu
        Ex_sum += Ex_j
    Ex_avg = Ex_sum / steps
    return p.kappa0 + float(p.kappa1.T @ Ex_avg)


# ---- BEI decomposition ----
def decompose(p: Params, X: np.ndarray, Z: float, n: float, yN: float, yR: float):
    exp_pi = exp_inf(p, X, n)
    lp = Z
    bei = yN - yR
    irp = bei - exp_pi - lp
    return dict(BEI=bei, ExpInf=exp_pi, LP

*実装ブロックは前回と同じ完全実装を収録しています。*

---

### ご活用のヒント
1. **フィルタ後の状態推定を .csv でエクスポート** → BI / IRP / LP の時系列チャート化。
2. **ブートストラップ後のゼロカーブ**は必ず単利換算に統一。
3. **IRP** にマクロファクター (景気先行指数, VIX) を回帰して解釈を深めると示唆が増える。

---

## 11. End-to-End Pipeline — BEI 推定 ➜ 寄与分解
以下のドライバースクリプトを実行すれば、
1. **モデル推定 (MLE)**
2. **カルマンスムーザで状態時系列取得**
3. **BEI 計算 & 期待インフレ / IRP / LP 分解**
を一括で行えます（ダミーデータ例）。

```python
import pandas as pd
import numpy as np
from typing import Sequence

# --- 0) データ読み込み ------------------------------------------------
# Assume CSV with columns: Date, N2, N5, N10, R5, R10, CPI
raw = pd.read_csv("example_data.csv", parse_dates=["Date"]).set_index("Date")

matur_N: Sequence[int] = [2,5,10]
matur_R: Sequence[int] = [5,10]
````
````

```python
# CPI YoY (または月次差) を観測に入れる
cpi_yoy = raw["CPI"].pct_change(12).dropna()
raw = raw.loc[cpi_yoy.index]
Y = np.vstack([
    raw[["N2","N5","N10"]].to_numpy(),
    raw[["R5","R10"]].to_numpy(),
    cpi_yoy.to_numpy().reshape(-1,1)
]).astype(float)

# --- 1) 初期パラメータ (PCA などで設定) ------------------------------
init_param = Params(
    mu=np.zeros((3,1)),
    Phi=np.diag([0.98,0.95,0.9]),
    Sigma=np.diag([0.05,0.04,0.03]),
    rho_Z=0.7,
    sig_Z=0.003,
    lambda0=np.zeros((3,1)),
    Lambda=np.zeros((3,3)),
    delta0_N=0.01, delta1_N=np.array([[1.0],[0.0],[0.0]]),
    delta0_R=0.008, delta1_R=np.array([[1.0],[0.0],[0.0]]),
    kappa0=0.002, kappa1=np.array([[0.2],[0.1],[0.05]]),
    sig_pi=0.3,
    R_diag=np.full(len(matur_N)+len(matur_R)+1, 0.0005),
    Ny=0, matur_N=[], matur_R=[], n_max=0
)

# --- 2) MLE 推定 ----------------------------------------------------
est_param = fit(Y, matur_N, matur_R, init_param)

# --- 3) スムーザで状態系列を取得 -------------------------------------
# ここでは簡便にフィルタ系列 (x) を取る。実際は RTS スムーザ推奨
from numpy.linalg import inv

T, Ny = Y.shape
K = 4
x_hist = np.zeros((T, K, 1))
P_hist = np.zeros((T, K, K))

A_N, B_N = riccati(est_param, est_param.n_max, True)
A_R, B_R = riccati(est_param, est_param.n_max, False)
c, D, H = build_measurement(est_param, A_N, B_N, A_R, B_R)
C = np.hstack([D,H])
R = np.diag(est_param.R_diag)
Phi_big = np.zeros((K,K)); Phi_big[:3,:3] = est_param.Phi; Phi_big[3,3]=est_param.rho_Z
Sig_big = np.zeros((K,K)); Sig_big[:3,:3] = est_param.Sigma; Sig_big[3,3]=est_param.sig_Z
mu_big = np.zeros((K,1)); mu_big[:3,0] = est_param.mu.flatten()

x, P = np.zeros((K,1)), np.eye(K)*0.1
for t in range(T):
    x_pred = mu_big + Phi_big @ x
    P_pred = Phi_big @ P @ Phi_big.T + Sig_big @ Sig_big.T
```

```
    y_pred = c + C @ x_pred
    S = C @ P_pred @ C.T + R
    K_gain = P_pred @ C.T @ inv(S)
    innov = Y[t].reshape(-1,1) - y_pred
    x = x_pred + K_gain @ innov
    P = (np.eye(K) - K_gain @ C) @ P_pred
    x_hist[t] = x
    P_hist[t] = P

# --- 4) BEI & 寄与分解 ---------------------------------------------
results = []
for t in range(T):
    X_t = x_hist[t,:3]
    Z_t = float(x_hist[t,3,0])
    yN = float(raw.iloc[t]["N10"])
    yR = float(raw.iloc[t]["R10"])
    res = decompose(est_param, X_t, Z_t, n=10, yN=yN, yR=yR)
    results.append(res)

res_df = pd.DataFrame(results, index=raw.index)
res_df.to_csv("bei_decomposition.csv")
print(res_df.tail())
```

ポイント

- **カスタムデータを `` に入れ替え**ればそのまま実行可能。
- スムーザは簡易版フィルタで代用しているが、`statsmodels` の RTS スムーザ等で精緻化するとより滑らかに。
- 列名 (`N2`, `R5` など) は自由に変えて OK — スクリプト側を合わせてください。

次の改善案\ • RTS スムーザ実装 / `pandas` マージで可視化\ • `matplotlib` で BEI vs ExpInf vs IRP vs LP の折れ線グラフ生成\ • マルチプロセスによる最尤推定高速化