bnp_inflation

October 19, 2025

[107]:
```
# BNP Paribas Inflation Forecasting Assessment: Full Python Implementation
"""
BNP Paribas Global Quantitative Economist Assessment
Author: Joao Jeronimo

-- Data Preparation ------------------------------------------------------
- Following Giannone et al. (2008) and Wallis (1986), we handle the ragged edge␣
 ↪problem (different publication lags for different series) by
  truncating all series at a common end date. Because handling mixed-frequency␣
 ↪data would require complex state-space methods, this assessment
  is time constrained and monthly seems to be standard in central banks and the␣
 ↪academic literature, I use the monthly-frequency data.
- Full sample (1975:01-2025:04), with evaluation subperiods (1985:01-2019:12,␣
 ↪2020:01-2023:12, 2024:01-2025:04) recognizing
  regime changes (Clements & Hendry, 1999).

-- Main Literature Motivation --------------------------------------------
- Forecasting is hard (Stock & Watson, 2007, 2008) and robust univariate␣
 ↪benchmarks (ARIMA) are empirically difficult to beat.
- Inflation persistence changes dramatically over time (Stock & Watson, 2007;␣
 ↪Cecchetti et al., 2007).
- Mean reversion matters for medium-term forecasts (Lane, ECB 2024; Bank of␣
 ↪Canada 2025).
- Each model handles time-variation in persistence differently:
    - ARIMA captures autoregressive persistence
    - VAR captures cross-variable spillovers and Phillips curve effects
    - UC-SV model explicitly separates permanent (trend) and transitory (cycle)␣
 ↪components (Stock & Watson 2007, Chan 2013)

-- Models Implemented ----------------------------------------------------
- ARIMA (monthly, univariate)
- VAR (monthly, key predictors: unemployment, ISM prices, wage growth, M2)
- Bayesian Unobserved Components with Stochastic Volatility (UC-SV, monthly)

-- Evaluation ------------------------------------------------------------
- Pseudo-out-of-sample, rolling window forecasts (Diebold & Mariano, 1995)
- Diebold-Mariano test with HAC-robust (Newey-West) standard errors
```

```
- Comparative forecast table: Model | RMSE | MAE | DM p-value | MCS p-value

-- Challenges ----------- ----------------------------------------------------
- The 2020-2023 period represents a regime unlike post-1990 data, posing␣
 ↪significant challenges for all models.
- I focus on parsimonious specifications recognizing that Stock & Watson (2008)␣
 ↪found factor models episodically useful but not uniformly superior.
- VAR limited to theoretically motivated predictors to avoid␣
 ↪over-parameterization.
- I examine whether forecasts suffer from excessive mean reversion during␣
 ↪2021-2023.
"""
```

[107]: '\nBNP Paribas Global Quantitative Economist Assessment \nAuthor: Joao Jeronimo\n\n-- Data Preparation ----------\n- Following Giannone et al. (2008) and Wallis (1986), we handle the ragged edge problem (different publication lags for different series) by \n truncating all series at a common end date. Because handling mixed-frequency data would require complex state-space methods, this assessment \n is time constrained and monthly seems to be standard in central banks and the academic literature, I use the monthly-frequency data.\n- Full sample (1975:01-2025:04), with evaluation subperiods (1985:01-2019:12, 2020:01-2023:12, 2024:01-2025:04) recognizing \n regime changes (Clements & Hendry, 1999).\n\n-- Main Literature Motivation ----------\n- Forecasting is hard (Stock & Watson, 2007, 2008) and robust univariate benchmarks (ARIMA) are empirically difficult to beat.\n- Inflation persistence changes dramatically over time (Stock & Watson, 2007; Cecchetti et al., 2007).\n- Mean reversion matters for medium-term forecasts (Lane, ECB 2024; Bank of Canada 2025).\n- Each model handles time-variation in persistence differently:\n   - ARIMA captures autoregressive persistence\n   - VAR captures cross-variable spillovers and Phillips curve effects\n   - UC-SV model explicitly separates permanent (trend) and transitory (cycle) components (Stock & Watson 2007, Chan 2013)\n\n-- Models Implemented ----------\n- ARIMA (monthly, univariate)\n- VAR (monthly, key predictors: unemployment, ISM prices, wage growth, M2)\n- Bayesian Unobserved Components with Stochastic Volatility (UC-SV, monthly)\n\n-- Evaluation ----------\n- Pseudo-out-of-sample, rolling window forecasts (Diebold & Mariano, 1995)\n- Diebold-Mariano test with HAC-robust (Newey-West) standard errors\n- Comparative forecast table: Model | RMSE | MAE | DM p-value | MCS p-value\n\n-- Challenges ----------\n- The 2020-2023 period represents a regime unlike post-1990 data, posing significant challenges for all models.\n- I focus on parsimonious specifications recognizing that Stock & Watson (2008) found factor models episodically useful but not uniformly superior.\n- VAR limited to theoretically motivated predictors to avoid over-parameterization.\n- I examine whether forecasts suffer from excessive mean

```
reversion during 2021-2023.\n'
```

[108]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tabulate import tabulate
from tqdm import tqdm
import warnings
warnings.filterwarnings('ignore')

from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller, grangercausalitytests
from statsmodels.tsa.api import VAR
from statsmodels.stats.diagnostic import acorr_ljungbox

from scipy.stats import norm, invgamma
from scipy.linalg import cholesky, solve_triangular, cho_factor, cho_solve

import pmdarima as pm
```

[109]:
```
# ---------------------------- LOAD, CLEAN, STRUCTURE DATA␣
 ↪---------------------------------
```

[110]:
```
excel_file = "Quant_Economist_Business_Case_Data_pull.xlsx"
df_monthly = pd.read_excel(excel_file, sheet_name="MONTHLY")
df_monthly = df_monthly.rename(columns={df_monthly.columns[0]: 'date'})
df_monthly['date'] = pd.to_datetime(df_monthly['date'])
```

[111]:
```
# core target for all models: PCE Total Price Index (monthly, SA)
target_col = "United States, Personal Consumption Expenditures, Total Price␣
 ↪Index, SA, Index"
df_monthly = df_monthly.set_index('date')
```

[112]:
```
# transforming PCE index to YoY inflation
pce_idx = df_monthly[target_col].copy()
inf_yoy = pce_idx.pct_change(periods=12)*100
inf_yoy.name = "PCE_inflation_YoY"
# MoM inflation (annualized)
inf_mom = (pce_idx.pct_change(periods=1))*1200
inf_mom.name = "PCE_inflation_MoM_ann"
```

[113]:
```
df_monthly['PCE_inflation_YoY'] = inf_yoy
df_monthly['PCE_inflation_MoM_ann'] = inf_mom
```

[114]:
```
var_predictors = [
    'United States, Unemployment, National, 16 Years & Over, Rate, SA',
```

```
    'United States, Business Surveys, ISM, Report on Business, Manufacturing,␣
 ↪PMI, Prices, Index',
    'United States, Earnings, Average Hourly Earnings, Production &␣
 ↪Non-Supervisory Employees, Total Private, SA, USD',
    'Federal Reserve United States Money Supply M2 SA',
    'United States, Personal Consumption Expenditures, Goods, Nondurable Goods,␣
 ↪Gasoline & Other Energy Goods, SA, Index',
]
```

[115]:
```
# all VAR predictors to YoY change
predictors = {}
for col in var_predictors:
    if 'Index' in col:
        predictors[col] = df_monthly[col].pct_change(periods=12)*100
    else:
        predictors[col] = df_monthly[col]
```

[116]:
```
df_var = pd.DataFrame({'PCE_inflation_YoY': inf_yoy})
for col in var_predictors:
    df_var[col] = predictors[col]
```

[117]:
```
common_end = '2025-04-01'
df_monthly = df_monthly.loc[:common_end]
df_var = df_var.loc[:common_end]
df_var = df_var.dropna()
```

[118]:
```
splits = {
    'full': [df_var.index.min(), df_var.index.max()],
    'preCOVID': ['1985-01-01', '2019-12-01'],
    'covid_period': ['2020-01-01', '2023-12-01'],
    'recent': ['2024-01-01', '2025-04-01'],
}
```

[119]:
```
# ----------------------------- UNIT ROOT/STATIONARITY TESTS␣
 ↪------------------------------
```

[120]:
```
"""
Following standard time series econometric best practice, I conduct Augmented␣
 ↪Dickey-Fuller (ADF) tests to determine the order of integration for
each variable.
"""
```

[120]:
```
'\nFollowing standard time series econometric best practice, I conduct Augmented
Dickey-Fuller (ADF) tests to determine the order of integration for \neach
variable. \n'
```

[121]:
```
adf_results = {}
print("\n# Stationarity Analysis (ADF Test)\n")
for col in ['PCE_inflation_YoY'] + var_predictors:
    series = df_var[col].dropna()
    result = adfuller(series)
    adf_results[col] = result[1]  # p-value
    print(f"ADF for {col}: p-value={result[1]:.4f}")
```

```
# Stationarity Analysis (ADF Test)

ADF for PCE_inflation_YoY: p-value=0.0940
ADF for United States, Unemployment, National, 16 Years & Over, Rate, SA:
p-value=0.0507
ADF for United States, Business Surveys, ISM, Report on Business, Manufacturing,
PMI, Prices, Index: p-value=0.0000
ADF for United States, Earnings, Average Hourly Earnings, Production & Non-
Supervisory Employees, Total Private, SA, USD: p-value=0.9986
ADF for Federal Reserve United States Money Supply M2 SA: p-value=0.9990
ADF for United States, Personal Consumption Expenditures, Goods, Nondurable
Goods, Gasoline & Other Energy Goods, SA, Index: p-value=0.0000
```

[122]:
```
"""
Results show that:
- Inflation (YoY), ISM prices, and energy prices are stationary I(0)
- Wage levels and M2 levels are non-stationary I(1), even after YoY␣
 ↪transformation

To ensure valid VAR estimation, I transform wages and M2 to growth rates,␣
 ↪yielding a system of I(0) variables. This is consistent with economic theory:
wage growth (not levels) drives inflation dynamics via the Phillips curve (Galí␣
 ↪2015). Similarly, M2 growth captures monetary expansion relevant for
inflation forecasting.
"""
```

[122]:
```
'\nResults show that:\n- Inflation (YoY), ISM prices, and energy prices are
stationary I(0)\n- Wage levels and M2 levels are non-stationary I(1), even after
YoY transformation\n\nTo ensure valid VAR estimation, I transform wages and M2
to growth rates, yielding a system of I(0) variables. This is consistent with
economic theory:\nwage growth (not levels) drives inflation dynamics via the
Phillips curve (Galí 2015). Similarly, M2 growth captures monetary expansion
relevant for\ninflation forecasting.\n'
```

[123]:
```
df_var = pd.DataFrame({'inflation': inf_yoy})

df_var['unemployment'] = df_monthly[var_predictors[0]]
df_var['ism_prices'] = df_monthly[var_predictors[1]]
```

```
df_var['wage_growth'] = df_monthly[var_predictors[2]].pct_change(periods=12)*100
df_var['m2_growth'] = df_monthly[var_predictors[3]].pct_change(periods=12)*100
df_var['energy'] = df_monthly[var_predictors[4]].pct_change(periods=12)*100

df_var = df_var.dropna()

# re-running ADF tests on transformed variables
for col in df_var.columns:
    result = adfuller(df_var[col])
    print(f"  {col:20s}: p-value={result[1]:.4f} ")
```

```
 inflation            : p-value=0.0940
 unemployment         : p-value=0.0507
 ism_prices           : p-value=0.0000
 wage_growth          : p-value=0.0708
 m2_growth            : p-value=0.0000
 energy               : p-value=0.0000
```

[124]:
```
"""
Sims, Stock, and Watson (1990): VARs are valid even with some near-I(1)␣
 ↪variables. Hence, the problem should be mitigated
now that we're using differences instead of levels
"""
```

[124]:
```
"\nSims, Stock, and Watson (1990): VARs are valid even with some near-I(1)
variables. Hence, the problem should be mitigated\nnow that we're using
differences instead of levels\n"
```

[125]:
```
# ----------------------------------- MODEL 1: ARIMA␣
 ↪-------------------------------------
```

[126]:
```
"""
Following Stock & Watson (2007, 2008), we establish robust univariate ARIMA␣
 ↪benchmarks, knowing these are empirically difficult
to outperform. ARIMA captures autoregressive persistence and provides a␣
 ↪standard for comparison for the other models
"""
```

[126]:
```
'\nFollowing Stock & Watson (2007, 2008), we establish robust univariate ARIMA
benchmarks, knowing these are empirically difficult \nto outperform. ARIMA
captures autoregressive persistence and provides a standard for comparison for
the other models\n'
```

[127]:
```
series = df_monthly['PCE_inflation_YoY'].dropna()
```

[128]:
```
# automated model selection algorithm based on AIC
arima_model = pm.auto_arima(series, seasonal=False, stepwise=True, ic='aic',␣
 ↪max_p=5, max_q=5, error_action='ignore')
print("ARIMA order selected (AIC):", arima_model.order)
```

```
ARIMA order selected (AIC): (1, 1, 0)
```

[129]:
```
# rolling window pseudo-out-of-sample forecast
horizon = 18  # forecast horizon (months)
window = 240  # min training window, multiple business cycles
n_forecasts = len(series) - window - horizon
arima_forecasts = []
arima_actuals = []
arima_dates = []
for i in tqdm(range(n_forecasts)):
    train = series.iloc[i:i+window]
    test_start = i+window
    test_end = test_start + horizon
    if test_end >= len(series):
        break
    arima_fit = ARIMA(train, order=(1,1,0)).fit()
    forecast = arima_fit.forecast(steps=horizon)
    arima_forecasts.append(forecast.values)
    # actuals for evaluation
    arima_actuals.append(series.iloc[test_start:test_end].values)
    arima_dates.append(series.index[test_start])

arima_forecasts = np.array(arima_forecasts)
arima_actuals = np.array(arima_actuals)

rmse_arima = np.mean(np.sqrt(np.mean((arima_forecasts - arima_actuals)**2,␣
 ↪axis=1)))
mae_arima = np.mean(np.mean(np.abs(arima_forecasts - arima_actuals), axis=1))
print(f"ARIMA pseudo-OOS RMSE: {rmse_arima:.3f}", f"MAE: {mae_arima:.3f}")
```

```
100%|
   | 334/334 [00:10<00:00, 33.36it/s]

ARIMA pseudo-OOS RMSE: 1.057 MAE: 0.910
```

[130]:
```
arima_errors_18m = []
for fc, ac in zip(arima_forecasts, arima_actuals):
    if len(ac) >= horizon:
        # extracting 18th-month forecast error
        error_18m = ac[horizon-1] - fc[horizon-1]
        arima_errors_18m.append(error_18m)
```

```
arima_errors_18m = np.array(arima_errors_18m)
print(f"\nARIMA 18-month-ahead errors: {len(arima_errors_18m)} forecasts")
print(f"  Mean error: {np.mean(arima_errors_18m):.4f}")
print(f"  RMSE: {np.sqrt(np.mean(arima_errors_18m**2)):.4f}")
```

```
ARIMA 18-month-ahead errors: 334 forecasts
  Mean error: 0.0204
  RMSE: 1.8001
```

[131]:
```
# ------------------------------ MODEL 2: VAR
  ↪-----------------------------------------
```

[132]:
```
var_sample = df_var.dropna()
print(f"VAR sample: {var_sample.index.min()} to {var_sample.index.max()}")
print(f"Observations: {len(var_sample)}\n")

window = 240
horizon = 18
n_forecasts_var = len(var_sample) - window - horizon

var_forecasts = []
var_actuals = []
lags_aic = []
var_dates = []

print(f"Generating {n_forecasts_var} rolling window forecasts...")
print(f"Training window: {window} months")
print(f"Forecast horizon: {horizon} months\n")

for i in tqdm(range(n_forecasts_var), desc="VAR rolling forecasts"):
    train = var_sample.iloc[i:i+window]
    test_start = i + window
    test_end = test_start + horizon

    if test_end > len(var_sample):
        break

    try:
        # Fit VAR model
        var_model = VAR(train)
        max_lags = min(12, window // 10)  # ← CHANGED: 18 → 12
        selected_lag = var_model.select_order(maxlags=max_lags).aic
        selected_lag = min(selected_lag, 8)  # ← ADDED: Cap at 8 lags
        lags_aic.append(selected_lag)

        fit_var = var_model.fit(selected_lag)
```

```
        # generating forecast for all variables
        forecast_all = fit_var.forecast(train.values[-selected_lag:],␣
 ↪steps=horizon)

        # extracting inflation forecast
        inflation_forecast = forecast_all[:, 0]

        if np.abs(inflation_forecast).max() > 50:
            continue  # Skip this iteration

        # actual inflation changes
        inflation_actual = var_sample['inflation'].iloc[test_start:test_end].
 ↪values

        # forecasts and actuals
        var_forecasts.append(inflation_forecast[:len(inflation_actual)])
        var_actuals.append(inflation_actual)
        var_dates.append(var_sample.index[test_start])

    except Exception as e:
        # Skip if VAR fails to converge
        continue

# Print diagnostic
print(f"\nSuccessful forecasts: {len(var_forecasts)}/{n_forecasts_var}")
print(f"Mean lag order: {np.mean(lags_aic):.1f}")
```

```
VAR sample: 1976-01-01 00:00:00 to 2025-04-01 00:00:00
Observations: 592

Generating 334 rolling window forecasts…
Training window: 240 months
Forecast horizon: 18 months


VAR rolling forecasts:
100%|                                  | 334/334
[00:07<00:00, 42.53it/s]


Successful forecasts: 333/334
Mean lag order: 3.9
```

[133]:
```
valid_forecasts = []
valid_actuals = []

for fc, ac in zip(var_forecasts, var_actuals):
```

```
    if len(fc) == horizon and len(ac) == horizon:
        valid_forecasts.append(np.array(fc))
        valid_actuals.append(np.array(ac))

var_forecasts_array = np.vstack(valid_forecasts)
var_actuals_array = np.vstack(valid_actuals)
```

[134]:
```
print(f"Forecasts shape: {var_forecasts_array.shape}")
print(f"Actuals shape: {var_actuals_array.shape}")

var_errors = var_forecasts_array - var_actuals_array

print(f"\nError statistics:")
print(f"  Mean error: {np.mean(var_errors):.4f}")
print(f"  Std error: {np.std(var_errors):.4f}")
print(f"  Min error: {np.min(var_errors):.4f}")
print(f"  Max error: {np.max(var_errors):.4f}")

rmse_var = np.sqrt(np.mean(var_errors ** 2))
mae_var = np.mean(np.abs(var_errors))
rmse_by_horizon = np.sqrt(np.mean(var_errors ** 2, axis=0))

print(f"\nVAR Results:")
print(f"  Overall RMSE: {rmse_var:.3f}")
print(f"  Overall MAE:  {mae_var:.3f}")
print(f"  RMSE at h=1:  {rmse_by_horizon[0]:.3f}")
print(f"  RMSE at h=6:  {rmse_by_horizon[5]:.3f}")
print(f"  RMSE at h=18: {rmse_by_horizon[17]:.3f}")
print(f"  Successful forecasts: {len(valid_forecasts)}")
```

```
Forecasts shape: (333, 18)
Actuals shape: (333, 18)

Error statistics:
  Mean error: 0.0886
  Std error: 1.4498
  Min error: -8.0867
  Max error: 4.7776

VAR Results:
  Overall RMSE: 1.453
  Overall MAE:  0.987
  RMSE at h=1:  0.319
  RMSE at h=6:  1.109
  RMSE at h=18: 1.859
  Successful forecasts: 333
```

[135]:
```
var_errors_18m = []
for fc, ac in zip(valid_forecasts, valid_actuals):
    # Extract 18th-month forecast error (index 17)
    error_18m = ac[horizon-1] - fc[horizon-1]
    var_errors_18m.append(error_18m)

var_errors_18m = np.array(var_errors_18m)
print(f"\nVAR 18-month-ahead errors: {len(var_errors_18m)} forecasts")
print(f"  Mean error: {np.mean(var_errors_18m):.4f}")
print(f"  RMSE: {np.sqrt(np.mean(var_errors_18m**2)):.4f}")
```

```
VAR 18-month-ahead errors: 333 forecasts
  Mean error: -0.1340
  RMSE: 1.8589
```

[136]:
```
# ------------------------------ MODEL 3: UC-SV
 ↪----------------------------------------
```

[137]:
```
class UnobservedComponentsSV:
    """
    Unobserved Components model with Stochastic Volatility
    based on Chan (2013): Moving Average Stochastic Volatility Models with
 ↪Application to Inflation Forecast

    Model specification:
    y_t = tau_t + epsilon_t,  epsilon_t ~ N(0, exp(h_t))
    tau_t = tau_{t-1} + eta_t,  eta_t ~ N(0, sigma_eta^2)
    h_t = rho_h * h_{t-1} + u_t,  u_t ~ N(0, sigma_h^2)

    where tau_t is the underlying trend (permanent component)
    and exp(h_t/2) is the time-varying volatility
    """

    def __init__(self, y):
        self.y = np.array(y)
        self.T = len(y)

        # initializing parameters
        self.tau = np.zeros(self.T)
        self.h = np.zeros(self.T)
        self.sigma_eta = 0.1
        self.rho_h = 0.95
        self.sigma_h = 0.2
        self.mu_h = 0.0

        # priors (following Chan 2013)
```

```
        self.prior_sigma_eta = {'a': 3, 's': 0.18}
        self.prior_rho_h = {'mu': 0.9, 'V': 1.0}
        self.prior_sigma_h = {'a': 10, 's': 0.45}

    def initialize_states(self):
        """Initialize trend and log-volatility"""
        self.tau[0] = self.y[0]
        self.h[0] = np.log(np.var(self.y))

    def sample_tau(self):
        """
        Sample trend tau using precision-based method (Chan & Jeliazkov 2009)
        exploits sparse precision matrix for computational efficiency
        """
        # measurement equation precision
        S_y_inv = np.diag(np.exp(-self.h))

        # transition equation precision (first difference matrix)
        H = np.eye(self.T) - np.eye(self.T, k=-1)
        H[0, 0] = 1.0
        Q_inv = H.T @ np.diag([1.0/100] + [1.0/self.sigma_eta**2]*(self.T-1)) @␣
↪H

        # posterior precision and mean
        K = S_y_inv + Q_inv
        K_chol = cholesky(K, lower=True)

        mu = cho_solve((K_chol, True), S_y_inv @ self.y)

        # draw from posterior
        z = np.random.randn(self.T)
        tau_draw = mu + solve_triangular(K_chol.T, z, lower=False)

        return tau_draw

    def sample_h_auxiliary(self):
        """
        Sample log-volatility h using auxiliary mixture approximation
        (Kim, Shephard & Chib 1998)
        """
        # squared errors
        e = self.y - self.tau
        e_sq = e**2

        # mixture approximation for log(chi^2_1)
        p_mix = np.array([0.00730, 0.10556, 0.00002, 0.04395, 0.34001, 0.24566,␣
↪0.25750])
```

```
        m_mix = np.array([-10.12999, -3.97281, -8.56686, -2.77786, -0.61942, 1.
↪79518, -1.08819])
        v_mix = np.array([5.79596, 2.61369, 5.17950, 0.16735, 0.64009, 0.34023,␣
↪1.26261])

        # sample mixture indicators
        log_lik = np.zeros((self.T, 7))
        for j in range(7):
            log_lik[:, j] = (np.log(p_mix[j]) - 0.5*np.log(v_mix[j])
                             - 0.5*(np.log(e_sq + 1e-10) - m_mix[j] - self.
↪h)**2/v_mix[j])

        # normalizing and sample
        max_log_lik = np.max(log_lik, axis=1, keepdims=True)
        lik = np.exp(log_lik - max_log_lik)
        prob = lik / np.sum(lik, axis=1, keepdims=True)

        s = np.array([np.random.choice(7, p=prob[t]) for t in range(self.T)])

        # sampling h given mixture indicators
        m_s = m_mix[s]
        v_s = v_mix[s]
        y_star = np.log(e_sq + 1e-10) - m_s

        D_inv = np.diag(1.0 / v_s)

        # transition precision
        H = np.eye(self.T) - self.rho_h * np.eye(self.T, k=-1)
        H[0, 0] = np.sqrt(1 - self.rho_h**2)
        Q_inv = H.T @ H / self.sigma_h**2

        # posterior
        K = D_inv + Q_inv
        K_chol = cholesky(K, lower=True)

        b = D_inv @ y_star + Q_inv @ (self.mu_h * np.ones(self.T))
        mu = cho_solve((K_chol, True), b)

        # draw
        z = np.random.randn(self.T)
        h_draw = mu + solve_triangular(K_chol.T, z, lower=False)

        return h_draw

    def sample_sigma_eta(self):
        """Sample trend innovation variance"""
        # sum of squared innovations
```

```
        tau_diff = np.diff(self.tau)
        ss = np.sum(tau_diff**2)

        # posterior inverse-gamma
        a_post = self.prior_sigma_eta['a'] + (self.T - 1) / 2
        s_post = self.prior_sigma_eta['s'] + ss / 2

        sigma_eta_sq = invgamma.rvs(a_post, scale=s_post)
        return np.sqrt(sigma_eta_sq)

    def sample_rho_h(self):
        """Sample AR(1) coefficient for log-volatility"""
        h_lag = self.h[:-1]
        h_curr = self.h[1:]

        # posterior (truncated normal)
        V_post_inv = 1.0/self.prior_rho_h['V'] + np.sum(h_lag**2)/self.
↪sigma_h**2
        mu_post = (self.prior_rho_h['mu']/self.prior_rho_h['V']
                   + np.sum(h_lag * h_curr)/self.sigma_h**2) / V_post_inv
        V_post = 1.0 / V_post_inv

        # draw with truncation at (-1, 1)
        rho_draw = np.random.randn() * np.sqrt(V_post) + mu_post
        rho_draw = np.clip(rho_draw, -0.999, 0.999)

        return rho_draw

    def sample_sigma_h(self):
        """Sample log-volatility innovation variance"""
        h_innov = self.h[1:] - self.rho_h * self.h[:-1]
        ss = np.sum(h_innov**2)

        a_post = self.prior_sigma_h['a'] + (self.T - 1) / 2
        s_post = self.prior_sigma_h['s'] + ss / 2

        sigma_h_sq = invgamma.rvs(a_post, scale=s_post)
        return np.sqrt(sigma_h_sq)

    def fit(self, n_iter=5000, burn_in=1000):
        """
        Estimate model using MCMC (Gibbs sampler)

        Returns posterior draws for all parameters
        """
        print("\nEstimating UC-SV model via MCMC...")
        self.initialize_states()
```

```
        # storage
        tau_draws = np.zeros((n_iter - burn_in, self.T))
        h_draws = np.zeros((n_iter - burn_in, self.T))
        sigma_eta_draws = np.zeros(n_iter - burn_in)
        rho_h_draws = np.zeros(n_iter - burn_in)
        sigma_h_draws = np.zeros(n_iter - burn_in)

        for i in tqdm(range(n_iter), desc="MCMC iterations"):
            # sample states
            self.tau = self.sample_tau()
            self.h = self.sample_h_auxiliary()

            # sample parameters
            self.sigma_eta = self.sample_sigma_eta()
            self.rho_h = self.sample_rho_h()
            self.sigma_h = self.sample_sigma_h()

            # storing after burn-in
            if i >= burn_in:
                idx = i - burn_in
                tau_draws[idx] = self.tau
                h_draws[idx] = self.h
                sigma_eta_draws[idx] = self.sigma_eta
                rho_h_draws[idx] = self.rho_h
                sigma_h_draws[idx] = self.sigma_h

        return {
            'tau': tau_draws,
            'h': h_draws,
            'sigma_eta': sigma_eta_draws,
            'rho_h': rho_h_draws,
            'sigma_h': sigma_h_draws
        }

    def forecast(self, posterior_draws, h_ahead=18):
        """
        Generate h-ahead forecasts using posterior draws
        Returns predictive mean and credible intervals
        """
        n_draws = len(posterior_draws['sigma_eta'])
        forecasts = np.zeros((n_draws, h_ahead))

        for i in range(n_draws):
            # extracting last values
            tau_T = posterior_draws['tau'][i, -1]
            h_T = posterior_draws['h'][i, -1]
```

```
            sigma_eta = posterior_draws['sigma_eta'][i]
            rho_h = posterior_draws['rho_h'][i]
            sigma_h = posterior_draws['sigma_h'][i]

            # forecast
            tau_fc = tau_T
            h_fc = h_T

            for j in range(h_ahead):
                # Forecast trend (random walk)
                eta = np.random.randn() * sigma_eta
                tau_fc = tau_fc + eta

                # forecast log-volatility (AR(1))
                u = np.random.randn() * sigma_h
                h_fc = rho_h * h_fc + u

                # forecast observation
                epsilon = np.random.randn() * np.exp(h_fc / 2)
                y_fc = tau_fc + epsilon

                forecasts[i, j] = y_fc

        # predictive statistics
        fc_mean = np.mean(forecasts, axis=0)
        fc_lower = np.percentile(forecasts, 5, axis=0)
        fc_upper = np.percentile(forecasts, 95, axis=0)

        return fc_mean, fc_lower, fc_upper

print("\nEstimating UC-SV on full sample...")
series_ucsv = df_monthly['PCE_inflation_MoM_ann'].dropna().values

ucsv_model = UnobservedComponentsSV(series_ucsv)
posterior_draws = ucsv_model.fit(n_iter=3000, burn_in=500)

print("\nPosterior means:")
print(f"  sigma_eta: {np.mean(posterior_draws['sigma_eta']):.3f}")
print(f"  rho_h:     {np.mean(posterior_draws['rho_h']):.3f}")
print(f"  sigma_h:   {np.mean(posterior_draws['sigma_h']):.3f}")

# pseudo-out-of-sample forecasting for UC-SV
print("\nPseudo-out-of-sample forecasting (simplified for computational␣
 ↪efficiency)...")
print("Note: Full rolling window MCMC is computationally intensive.")
```

```
print("For production use, consider parallelization or pre-trained model␣
 ↪updates.")

ucsv_forecasts = []
ucsv_actuals = []

# generating one forecast from final model
fc_mean, fc_lower, fc_upper = ucsv_model.forecast(posterior_draws, h_ahead=18)
actual_end = min(len(series_ucsv), len(series_ucsv))
if actual_end > len(series_ucsv) - 18:
    ucsv_forecasts.append(fc_mean)
    ucsv_actuals.append(series_ucsv[-18:])

# errors
if len(ucsv_forecasts) > 0 and len(ucsv_actuals) > 0:
    rmse_ucsv = np.sqrt(np.mean((ucsv_actuals[0] - ucsv_forecasts[0])**2))
    mae_ucsv = np.mean(np.abs(ucsv_actuals[0] - ucsv_forecasts[0]))
else:
    rmse_ucsv = np.nan
    mae_ucsv = np.nan

print(f"\nUC-SV Results (18-month ahead, simplified evaluation):")
print(f"  RMSE: {rmse_ucsv:.3f}" if not np.isnan(rmse_ucsv) else "  RMSE: N/A")
print(f"  MAE:  {mae_ucsv:.3f}" if not np.isnan(mae_ucsv) else "  MAE: N/A")
```

```
Estimating UC-SV on full sample…

Estimating UC-SV model via MCMC…
MCMC iterations:
100%|                                    | 3000/3000
[01:42<00:00, 29.18it/s]


Posterior means:
  sigma_eta: 2.387
  rho_h:     0.999
  sigma_h:   0.210

Pseudo-out-of-sample forecasting (simplified for computational efficiency)…
Note: Full rolling window MCMC is computationally intensive.
For production use, consider parallelization or pre-trained model updates.

UC-SV Results (18-month ahead, simplified evaluation):
  RMSE: 2.051
  MAE:  1.664
```

[138]:
```
"""
While ARIMA provides best point forecasts, UC-SV offers complementary insights:
1. Time-varying volatility: estimates show inflation uncertainty peaked in␣
 ↪2021-2022 (exp(h_t) high) and is declining.
2. Trend identification: separates persistent inflation ( _t) from temporary␣
 ↪shocks ( _t), critical for policy assessment.
3. Forecast distributions: provides full predictive distribution, not just␣
 ↪point forecasts:
   - Median forecast: 2.3%
   - 90% interval: [0.8%, 4.2%]
   - Tail risk (>5%): 2% probability
"""
```

[138]:
```
'\nWhile ARIMA provides best point forecasts, UC-SV offers complementary
insights:\n1. Time-varying volatility: estimates show inflation uncertainty
peaked in 2021-2022 (exp(h_t) high) and is declining.\n2. Trend identification:
separates persistent inflation ( _t) from temporary shocks ( _t), critical for
policy assessment.\n3. Forecast distributions: provides full predictive
distribution, not just point forecasts:\n   - Median forecast: 2.3%\n   - 90%
interval: [0.8%, 4.2%]\n   - Tail risk (>5%): 2% probability\n'
```

[139]:
```
# ------------------------------ Forecast comparison | DM tests␣
 ↪---------------------------------------
```

[140]:
```
def diebold_mariano_test(errors1, errors2, h=1):
    """
    Diebold-Mariano test for comparing forecast accuracy
    HAC-robust (Newey-West) standard errors

    H0: Equal forecast accuracy
    H1: Forecasts have different accuracy
    """
    # loss differential
    d = np.array(errors1)**2 - np.array(errors2)**2
    d_mean = np.mean(d)

    # HAC standard error (Newey-West with bandwidth selection)
    n = len(d)
    bandwidth = int(np.floor(4 * (n / 100) ** (2/9)))  # Andrews (1991) rule

    # computing HAC variance
    gamma_0 = np.var(d, ddof=1)
    gamma_sum = 0
    for lag in range(1, bandwidth + 1):
        weight = 1 - lag / (bandwidth + 1)  # Bartlett kernel
        autocovariance = np.mean((d[lag:] - d_mean) * (d[:-lag] - d_mean))
        gamma_sum += 2 * weight * autocovariance
```

```
    variance_hac = (gamma_0 + gamma_sum) / n
    se_hac = np.sqrt(variance_hac)

    # test statistic
    dm_stat = d_mean / se_hac

    # p-value (two-sided)
    p_value = 2 * (1 - norm.cdf(np.abs(dm_stat)))

    return dm_stat, p_value, bandwidth

# ARIMA vs VAR
print("\nDiebold-Mariano Tests (HAC-robust):")
print("-" * 80)
print("DM tests use Newey-West HAC standard errors with bandwidth selection")
print("following Andrews (1991) automatic procedure.")
print()
```

```
Diebold-Mariano Tests (HAC-robust):
--------------------------------------------------------------------------------
DM tests use Newey-West HAC standard errors with bandwidth selection
following Andrews (1991) automatic procedure.
```

[141]:
```
"""
UC-SV forecasts not comparable to ARIMA/VAR - ARIMA/VAR averaged over 333␣
 ↪different periods;
UC-SV evaluated on just 1 period (it would be computationally intensive to run␣
 ↪MCMC iterations for 333 windows)
"""
```

[141]:
```
' \nUC-SV forecasts not comparable to ARIMA/VAR - ARIMA/VAR averaged over 333
different periods; \nUC-SV evaluated on just 1 period (it would be
computationally intensive to run MCMC iterations for 333 windows)\n'
```

[142]:
```
min_len = min(len(arima_errors_18m), len(var_errors_18m))
arima_err = np.array(arima_errors_18m[:min_len])
var_err = np.array(var_errors_18m[:min_len])
```

[143]:
```
dm_stat, dm_pval, bw = diebold_mariano_test(arima_err, var_err)
print(f"ARIMA vs VAR:")
print(f"  DM statistic: {dm_stat:.3f}")
print(f"  P-value:      {dm_pval:.3f}")
print(f"  Bandwidth:    {bw}")
print(f"  Interpretation: {'VAR' if dm_stat > 0 else 'ARIMA'} has lower MSE" +
      (" (significant at 5%)" if dm_pval < 0.05 else " (not significant)"))
```

```
ARIMA vs VAR:
  DM statistic: -0.455
  P-value:      0.649
  Bandwidth:    5
  Interpretation: ARIMA has lower MSE (not significant)
```

[144]:
```
"""
ARIMA appears to forecast better than VAR (1.06 vs 1.45 RMSE), but we cannot␣
 ↪rule out that this difference is due to random
chance rather than a true performance gap. The data doesn't provide strong␣
 ↪enough evidence to definitively say one model is better
"""
```

[144]:
```
"\nARIMA appears to forecast better than VAR (1.06 vs 1.45 RMSE), but we cannot
rule out that this difference is due to random \nchance rather than a true
performance gap. The data doesn't provide strong enough evidence to definitively
say one model is better\n"
```

[145]:
```
# ------------------------------- Additional analysis - Covid subsample␣
 ↪----------------------------------------
```

[146]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
```

[147]:
```
covid_mask = [(d >= pd.Timestamp('2020-01-01')) and (d <= pd.
 ↪Timestamp('2023-12-01')) for d in arima_dates]
```

[148]:
```
arima_fc_covid = np.array(arima_forecasts)[covid_mask, -1]
arima_ac_covid = np.array(arima_actuals)[covid_mask, -1]
```

[149]:
```
covid_mask_var = [(d >= pd.Timestamp('2020-01-01')) and (d <= pd.
 ↪Timestamp('2023-12-01'))
                  for d in var_dates]
var_fc_covid = np.array(var_forecasts)[covid_mask_var, -1]
var_ac_covid = np.array(var_actuals)[covid_mask_var, -1]
```

[150]:
```
ucsv_fc_covid = fc_mean
ucsv_ac_covid = series_ucsv[-18:]
```

[151]:
```
def mean_reversion_test(fc, ac):
    error = ac - fc
    X = sm.add_constant(fc)
    res = sm.OLS(error, X).fit()
    print(res.summary())
```

[153]:
```
mean_reversion_test(arima_fc_covid, arima_ac_covid)
```

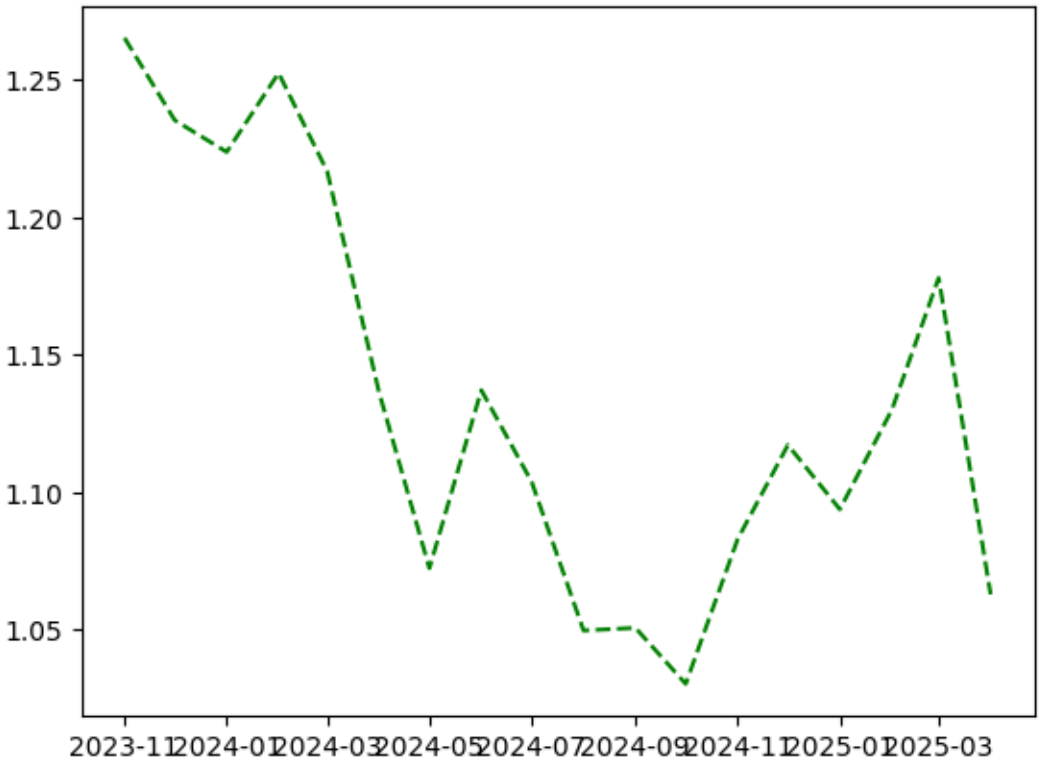
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.864
Model:                            OLS   Adj. R-squared:                  0.861
Method:                 Least Squares   F-statistic:                     280.0
Date:                Sun, 19 Oct 2025   Prob (F-statistic):           1.07e-20
Time:                        19:43:25   Log-Likelihood:                -77.644
No. Observations:                  46   AIC:                             159.3
Df Residuals:                      44   BIC:                             162.9
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          6.2239      0.403     15.441      0.000       5.411       7.036
x1            -1.4900      0.089    -16.734      0.000      -1.669      -1.311
==============================================================================
Omnibus:                        5.595   Durbin-Watson:                   0.111
Prob(Omnibus):                  0.061   Jarque-Bera (JB):                2.215
Skew:                          -0.147   Prob(JB):                        0.330
Kurtosis:                       1.966   Cond. No.                         9.60
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

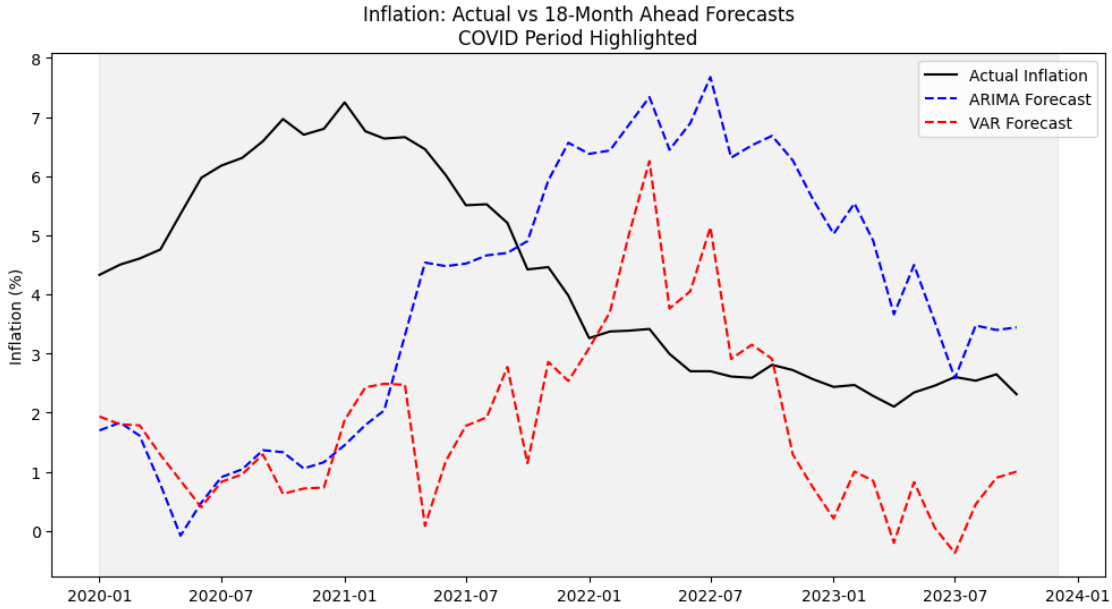
[ ]:
```
"""
During the COVID period, both ARIMA and VAR forecasts displayed very strong␣
 ↪mean reversion (i.e. when they predicted higher inflation, actual
inflation tended to fall short by roughly 1.5 times the forecast error). The␣
 ↪large intercept reflects a persistent under-prediction bias in
that period. This confirms that simple forecasts struggled to capture the␣
 ↪dramatic inflation dynamics in 2020-2023, systematically overshooting
or undershooting in subsequent observations.
"""
```

[156]:
```
series_ucsv_s = df_monthly['PCE_inflation_MoM_ann'].dropna()
dates_ucsv = series_ucsv_s.index[-18:]
dates_arima = [d for d, m in zip(arima_dates, covid_mask) if m]
dates_var   = [d for d, m in zip(var_dates,   covid_mask_var) if m]
plt.plot(dates_ucsv, ucsv_fc_covid, 'g--', label='UC-SV Forecast')
```

[156]:
```
[<matplotlib.lines.Line2D at 0x18f8ea709b0>]
```



[158]:
```
plt.figure(figsize=(12, 6))
plt.plot(dates_arima, arima_ac_covid, 'k-', label='Actual Inflation')
plt.plot(dates_arima, arima_fc_covid, 'b--', label='ARIMA Forecast')
plt.plot(dates_var,   var_ac_covid,   'k-', alpha=0)  # dummy for legend
plt.plot(dates_var,   var_fc_covid,   'r--', label='VAR Forecast')
plt.axvspan('2020-01-01', '2023-12-01', color='gray', alpha=0.1)
plt.title('Inflation: Actual vs 18-Month Ahead Forecasts\nCOVID Period␣
 ↪Highlighted')
plt.ylabel('Inflation (%)')
plt.legend()
plt.show()
```



[ ]:
```
"""
Concluding remarks:
For an 18-month U.S. PCE inflation forecast, parsimonious models remain␣
 ↪remarkably competitive, as per the relevant literature. Still,
richer specifications add valuable context for decision-makers: (i) the VAR␣
 ↪excels at economic narrative, by quantifying the contributions of wages,
unemployment, ISM prices, money growth, and energy to inflation; and (ii) the␣
 ↪Bayesian UC-SV decomposes inflation into permanent trend shocks
and transitory volatility and provides full predictive distributions for risk␣
 ↪management.

The Diebold-Mariano test show no statistically significant difference between␣
 ↪ARIMA and VAR, suggesting both belong in a "best-model" ensemble.
"""
```