TENSOR-STRUCTURED STATISTICAL PROCESS CONTROL

Theory, Implementation, and Simulation for High-Dimensional Process Monitoring

Prepared by

Jeff M.

April 2026

DOI: 10.5281/zenodo.20260650

Contents of this volume

Publication Information

Title:	*Tensor-Structured Statistical Process Control*
Author:	Jeff M.
Date:	April 2026
DOI:	10.5281/zenodo.20260650
Resource type:	Technical monograph / report

Recommended citation

Jeff M. (2026). Tensor-Structured Statistical Process Control. Zenodo. https://doi.org/10.5281/zenodo.20260650.

About This Volume

This book combines the full notebook sequence developed for learning, implementing, and evaluating tensor-structured statistical process control (SPC). The material moves from tensor foundations through decomposition methods, residual and score-based monitoring, Monte Carlo benchmarking, ARL calibration, and sequential EWMA/CUSUM extensions, curated simulation results, conclusions, and references.

The DOI and citation information are included as part of the front matter so the document can be treated as a citable technical monograph rather than only a notebook export.

The typesetting in this edition is designed for printing. Explanatory text is set as a technical monograph, while executable Python content is preserved in clean code listings for desk-reference use.

Contents

Part I

Foundations and Decomposition

1 Tensor Foundations for SPC

This section introduces the tensor concepts most relevant to this work:

- vectors, matrices, and higher-order tensors
- inner products and norms
- outer products
- unfolding / matricization
- n-mode products
- Tucker decomposition
- residual tensors and the Q statistic

The focus is computational and practical, using **NumPy** and **TensorLy**.

1.1 Imports

```
import numpy as np

try:
    import tensorly as tl
    from tensorly import unfold
    from tensorly.tenalg import mode_dot
    from tensorly.decomposition import tucker
    from tensorly.tucker_tensor import tucker_to_tensor
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)
```

1.2 Vector inner product and norm

```
u = np.array([1, 2, 3], dtype=float)
v = np.array([4, 5, 6], dtype=float)

inner_uv = np.dot(u, v)
norm_u = np.linalg.norm(u)

print("u =", u)
print("v =", v)
print("u · v =", inner_uv)
print("||u|| =", norm_u)
```

1.3 Matrix (second-order tensor) inner product and Frobenius norm

```
X = np.array([[1, 2],
              [3, 4]], dtype=float)

Y = np.array([[5, 6],
              [7, 8]], dtype=float)

inner_XY = np.sum(X * Y)
fro_X = np.linalg.norm(X)

print("X =\n", X)
print("Y =\n", Y)
print("⟨X, Y⟩ =", inner_XY)
print("||X||_F =", fro_X)
```

1.4 Third-order tensor inner product

```
A = np.array([
    [[1, 2],
     [3, 4]],

    [[5, 6],
     [7, 8]]
], dtype=float)

B = np.array([
    [[1, 0],
     [0, 1]],

    [[2, 1],
     [1, 2]]
```

```
], dtype=float)

inner_AB = np.sum(A * B)
norm_A = np.sqrt(np.sum(A * A))

print("A shape:", A.shape)
print("B shape:", B.shape)
print("⟨A, B⟩ =", inner_AB)
print("||A|| =", norm_A)
```

1.5 Outer product (tensor product of vectors)

For vectors, the tensor product is the outer product.

```
u = np.array([1, 2], dtype=float)
v = np.array([3, 4], dtype=float)

outer_uv = np.outer(u, v)
print("u ⊗ v =\n", outer_uv)
```

1.6 Tensor unfolding / matricization

```
T = np.arange(24).reshape(3, 4, 2)

mode0 = T.reshape(T.shape[0], -1)
mode1 = np.moveaxis(T, 1, 0).reshape(T.shape[1], -1)
mode2 = np.moveaxis(T, 2, 0).reshape(T.shape[2], -1)

print("Original shape:", T.shape)
print("Mode-0 unfolding:", mode0.shape)
print(mode0)
print("\nMode-1 unfolding:", mode1.shape)
print(mode1)
print("\nMode-2 unfolding:", mode2.shape)
print(mode2)
```

1.7 TensorLy unfolding and n-mode products

```
if TENSORLY_OK:
    X = tl.tensor(np.arange(24).reshape(3, 4, 2), dtype=float)
    print("Mode-0 unfolding shape:", unfold(X, 0).shape)
    print("Mode-1 unfolding shape:", unfold(X, 1).shape)
    print("Mode-2 unfolding shape:", unfold(X, 2).shape)
```

```
    M = tl.tensor([[1, 0, 0],
                   [0, 2, 0]], dtype=float)  # shape (2,3)

    X2 = mode_dot(X, M, mode=0)
    print("Original shape:", X.shape)
    print("After mode-0 multiplication:", X2.shape)
else:
    print("Install TensorLy to run this section.")
```

1.8 Tucker decomposition

```
if TENSORLY_OK:
    X = tl.tensor([
        [[1.0, 2.0],
         [3.0, 4.0]],

        [[5.0, 6.0],
         [7.0, 8.0]]
    ])

    core, factors = tucker(X, rank=[1, 2, 2])
    X_hat = tucker_to_tensor((core, factors))

    print("Core tensor:\n", core)
    for i, f in enumerate(factors):
        print(f"\nFactor matrix for mode {i}:\n", f)

    print("\nReconstruction:\n", X_hat)
else:
    print("Install TensorLy to run this section.")
```

1.9 Residual tensor and Q statistic

In tensor SPC, a very natural analogue of SPE/Q is the squared norm of the residual tensor.

```
if TENSORLY_OK:
    E = X - X_hat
    Q = tl.sum(E * E)

    print("Residual tensor:\n", E)
    print("\nQ statistic =", float(Q))
else:
    print("Install TensorLy to run this section.")
```

1.10 Why this matters for tensor SPC

This section builds the bridge from classical tensor algebra to monitoring logic:

- **inner product** → similarity / energy / projection geometry
- **norm** → magnitude of variation
- **unfolding** → interface between tensors and matrix methods
- **Tucker decomposition** → low-rank tensor subspace model
- **residual tensor** → out-of-model variation
- **Q statistic** → monitoring unexplained variation

2 Tucker Decomposition and HOSVD Workflow

This section focuses on the multilinear decomposition ideas that are most relevant for tensor SPC.

Topics:

- mode-wise structure
- Tucker decomposition
- interpretation of factor matrices
- reconstruction quality
- low-rank approximation

2.1 Imports

```
import numpy as np
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    from tensorly.tucker_tensor import tucker_to_tensor
    from tensorly import unfold
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)
```

2.2 Build a structured synthetic tensor

Think of the tensor as **parts** $\times$ **variables** $\times$ **time points**.

```
np.random.seed(7)

n_parts = 20
n_vars = 5
n_time = 10

part_pattern = np.linspace(-1, 1, n_parts)[:, None, None]
var_pattern = np.array([1.0, 0.8, 1.2, 0.9, 1.1])[None, :, None]
time_pattern = np.sin(np.linspace(0, 2*np.pi, n_time))[None, None, :]

signal = 2.0 * part_pattern * var_pattern + 1.5 * var_pattern * time_pattern
noise = 0.15 * np.random.randn(n_parts, n_vars, n_time)

X_np = signal + noise
print("Tensor shape:", X_np.shape)
```

2.3 Visualize one slice

```
plt.figure(figsize=(8, 4))
plt.imshow(X_np[:, :, 0], aspect='auto')
plt.colorbar()
plt.title("Slice at time index 0")
plt.xlabel("Variable")
plt.ylabel("Part")
plt.show()
```

2.4 Tucker decomposition

```
if TENSORLY_OK:
    X = tl.tensor(X_np, dtype=float)
    rank = [3, 3, 3]
    core, factors = tucker(X, rank=rank)
    X_hat = tucker_to_tensor((core, factors))
    print("Core shape:", core.shape)
    for i, f in enumerate(factors):
        print(f"Factor {i} shape:", f.shape)
else:
    print("Install TensorLy to run this section.")
```

2.5 Inspect factor matrices

```
if TENSORLY_OK:
    for i, f in enumerate(factors):
        print(f"\nFactor matrix {i}:\n", np.array(f))
```

2.6 Reconstruction error

```
if TENSORLY_OK:
    residual = np.array(X - X_hat)
    rel_error = np.linalg.norm(residual) / np.linalg.norm(X_np)
    print("Relative reconstruction error:", rel_error)
```

2.7 Compare ranks

```
if TENSORLY_OK:
    rank_options = [
        [1, 1, 1],
        [2, 2, 2],
        [3, 3, 3],
        [4, 4, 4]
    ]

    errors = []
    for r in rank_options:
        core_r, factors_r = tucker(X, rank=r)
        X_hat_r = tucker_to_tensor((core_r, factors_r))
        err = np.linalg.norm(np.array(X - X_hat_r)) / np.linalg.norm(X_np)
        errors.append(err)

    for r, e in zip(rank_options, errors):
        print(f"Rank {r}: relative error = {e:.6f}")
```

2.8 Plot reconstruction error vs rank

```
if TENSORLY_OK:
    plt.figure(figsize=(8, 4))
    plt.plot(range(1, 5), errors, marker='o')
    plt.xticks(range(1, 5), ['[1,1,1]', '[2,2,2]', '[3,3,3]', '[4,4,4]'])
    plt.ylabel("Relative reconstruction error")
    plt.xlabel("Chosen Tucker rank")
    plt.title("Effect of rank on approximation quality")
    plt.show()
```

2.9 SPC interpretation

Within this work, the decomposition provides:

- a **mode-wise low-rank structure**
- a **reconstructed tensor** representing modeled behavior
- a **residual tensor** representing unexplained behavior

That residual is the natural foundation for a tensor-based Q/SPE statistic.

3 Theoretical Foundations and Development of Tensor SPC

3.1 Introduction

Classical multivariate statistical process control (MSPC) methods are formulated for vector-valued observations and rely on covariance structure estimated from flattened data representations. In many modern applications, process data are inherently multiway, exhibiting structure across variables, time, spatial dimensions, or operating conditions. Vectorization of such data obscures this structure and may degrade monitoring performance.

Tensor-based SPC extends MSPC by preserving multiway structure through multilinear representations. This enables monitoring methods that respect the inherent organization of the data while retaining compatibility with established statistical frameworks.

3.2 Assumptions

Let $\{\mathcal{X}_t\}_{t=1}^{T}$, where $\mathcal{X}_t \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, denote Phase I observations. Let a Tucker model with ranks $(R_1, \ldots, R_N)$ be fitted, yielding factor matrices $U^{(n)}$ and core coefficients $\mathcal{G}_t$.

We adopt the following conditions for analysis:

A1 (Centering and scaling). Data are mode-wise centered and, where appropriate, scaled so that first- and second-order moments are comparable across modes.

A2 (Multilinear signal plus noise). There exists a low-rank multilinear structure

$$\mathcal{X}_t = \mathcal{S}_t + \mathcal{E}_t,$$

where $\mathcal{S}_t$ lies on a low-rank Tucker manifold and $\mathcal{E}_t$ is a zero-mean noise tensor.

A3 (Noise). $\mathrm{vec}(\mathcal{E}_t)$ is sub-Gaussian with covariance Σ_E. For classical approximations, Gaussianity may be assumed.

A4 (Independence or weak dependence). The sequence $\{\mathcal{E}_t\}$ is independent or satisfies a weak dependence condition sufficient for asymptotic approximation of projected scores.

A5 (Identifiability). Factor matrices $U^{(n)}$ are orthonormal, with the usual Tucker indeterminacies

understood.

A6 (Rank selection). Chosen ranks capture the dominant signal subspace without absorbing residual noise structure.

3.3 Multilinear Representation

Let $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ denote an N-way tensor. A Tucker decomposition represents $\mathcal{X}$ as

$$\mathcal{X} \approx \mathcal{G} \times_1 U^{(1)} \times_2 \cdots \times_N U^{(N)},$$

where $\mathcal{G}$ is a core tensor and $U^{(n)} \in \mathbb{R}^{I_n \times R_n}$ are factor matrices. This decomposition preserves multiway structure while reducing dimensionality mode by mode.

In contrast to vector PCA, which imposes a single global subspace after flattening, the Tucker model captures structured variation distributed across multiple modes.

3.4 Monitoring Statistics

Let $\mathbf{z}_t = \mathrm{vec}(\mathcal{G}_t)$ denote the vectorized score representation of the projected core tensor.

3.4.1 Tensor T^2

The tensor T^2 statistic is defined by

$$T_t^2 = \mathbf{z}_t^\top \Lambda^{-1} \mathbf{z}_t,$$

where Λ is the covariance matrix of the retained score representation. This statistic measures variation aligned with the modeled multilinear subspace.

3.4.2 Tensor Q

The residual-based monitoring statistic is

$$Q_t = \|\mathcal{X}_t - \widehat{\mathcal{X}}_t\|_F^2,$$

where $\widehat{\mathcal{X}}_t$ is the reconstruction from the low-rank model. The Q statistic measures variation outside the modeled structure.

3.4.3 Complementarity

The two statistics provide complementary sensitivity:

- T^2: structured, global deviations within the modeled subspace
- Q: localized or unmodeled deviations outside the low-rank representation

3.5 Theoretical Properties

3.5.1 Theorem 1 (Asymptotic distribution of tensor T^2)

Under A1–A6 and correct model specification, the tensor T^2 statistic admits the approximation

$$T_t^2 \xrightarrow{d} \chi_d^2,$$

where $d = R_1 \cdots R_N$, provided Λ is consistently estimated and the effective dimension remains moderate relative to the Phase I sample size.

3.5.2 Theorem 2 (Residual Q approximation)

Under A1–A6, the residual statistic

$$Q_t = \|\mathcal{X}_t - \widehat{\mathcal{X}}_t\|_F^2$$

admits a moment-matched approximation of the form

$$Q_t \approx \theta \chi_h^2,$$

where θ and h are determined from the first two moments of the residual spectrum.

3.5.3 Theorem 3 (Sensitivity of T^2)

If a shift lies primarily in the modeled multilinear subspace, then the probability of exceeding the T^2 control limit increases with the subspace-aligned shift magnitude. Accordingly, T^2 is most sensitive to structured, global deviations.

3.5.4 Theorem 4 (Sensitivity of Q)

If a deviation lies primarily in the orthogonal complement of the modeled subspace, then the expected value of Q increases with residual energy, while T^2 remains comparatively stable. Accordingly, Q is

most sensitive to localized or unmodeled anomalies.

3.5.5 Corollary (Complementary detection behavior)

For mixed alternatives containing both structured and orthogonal components, T^2 primarily detects the structured component, whereas Q primarily detects the orthogonal component. This establishes the complementary monitoring roles of the two statistics.

3.6 Control Limits and ARL

Let $c_\alpha^{T^2}$ and c_α^Q denote upper control limits at significance level α.

- $c_\alpha^{T^2}$ may be obtained from chi-square approximations or finite-sample adjustments.
- c_α^Q may be obtained from moment-matched approximations or empirical Phase I quantiles.

For a target in-control average run length ARL_0, a common approximation is

$$\alpha \approx \frac{1}{\mathrm{ARL}_0}.$$

In sequential extensions such as EWMA and CUSUM, the in-control ARL is generally calibrated through simulation or Markov-chain approximations.

3.7 Relation to Interaction Structure

Tensor SPC captures structured covariance across modes rather than explicit parametric interaction effects. Unlike designed experiments, which model interaction terms directly, multilinear decompositions identify latent patterns reflecting joint variation across dimensions.

Accordingly, tensor SPC provides a descriptive representation of multiway dependence without assigning explicit causal interpretation to individual interactions.

3.8 Implementation Considerations

3.8.1 Data organization

The choice of tensor modes should reflect the process structure, such as variable $\times$ time $\times$ condition or sensor $\times$ location $\times$ time. Improper mode selection can obscure relevant variation.

3.8.2 Rank selection

Selection of $(R_1, \ldots, R_N)$ governs the bias–variance tradeoff. Practical approaches include explained variation, cross-validation, and stability of monitoring statistics.

3.8.3 Scaling

Mode-wise centering and scaling are required to ensure comparability across variables and operating conditions.

3.8.4 Computation

For large datasets, multilinear decompositions may require efficient or incremental algorithms for practical deployment.

3.9 Extensions and Research Directions

Tensor SPC naturally supports several extensions:

- sequential monitoring through EWMA and CUSUM applied to tensor T^2 and Q
- adaptive updating for nonstationary processes
- integration with machine learning methods for residual analysis
- further theoretical development for distributional approximations and run-length properties

3.10 Summary

Tensor SPC generalizes classical MSPC by preserving multiway structure through multilinear decomposition. The resulting framework enables structured monitoring through tensor T^2 and Q statistics, providing complementary sensitivity to global and localized deviations. This yields a principled foundation for monitoring structured, high-dimensional process data and supports continued methodological development in multiway process control.

3.11 Relation to Existing Work

This framework extends classical multivariate statistical process control approaches by replacing vectorized representations with multilinear tensor models. Classical PCA and MSPC provide the natural vectorized baseline, while tensor decompositions preserve multiway structure across variables, time, and operating conditions. This distinction is central to the interpretation of the simulation studies that follow.

Part II

Tensor Monitoring Fundamentals

4 Residuals, Q Statistic, and Tensor SPC Logic

This section demonstrates how Tucker-based modeling leads naturally to a monitoring statistic.

Main ideas:

- build a reference model from normal data
- reconstruct each new sample
- compute residual energy
- monitor Q/SPE over samples

4.1 Imports

```
import numpy as np
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    from tensorly.tucker_tensor import tucker_to_tensor
    from tensorly.tenalg import mode_dot
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)
```

4.2 Simulate normal-reference tensor data

We'll treat each sample as a small matrix of variables × time.

```
np.random.seed(123)

n_samples = 60
```

```
n_vars = 4
n_time = 12

base_time = np.sin(np.linspace(0, 2*np.pi, n_time))
base_var = np.array([1.0, 0.9, 1.1, 1.05])

X_ref = np.zeros((n_samples, n_vars, n_time))
for i in range(n_samples):
    amplitude = 1.0 + 0.05*np.random.randn()
    noise = 0.08*np.random.randn(n_vars, n_time)
    X_ref[i] = amplitude * np.outer(base_var, base_time) + noise

print("Reference tensor shape:", X_ref.shape)
```

4.3 Fit Tucker model to reference data

```
if TENSORLY_OK:
    X_ref_tl = tl.tensor(X_ref, dtype=float)
    rank = [3, 3, 3]
    core_ref, factors_ref = tucker(X_ref_tl, rank=rank)
    X_ref_hat = tucker_to_tensor((core_ref, factors_ref))
    E_ref = np.array(X_ref_tl - X_ref_hat)
    Q_ref = np.sum(E_ref**2, axis=(1, 2))
    print("Computed Q values for reference set.")
else:
    print("Install TensorLy to run this section.")
```

4.4 Reference Q distribution and empirical limit

```
if TENSORLY_OK:
    q_limit = np.percentile(Q_ref, 99)

    print("Reference Q mean:", Q_ref.mean())
    print("Reference Q std:", Q_ref.std())
    print("Empirical 99th percentile limit:", q_limit)
```

4.5 Plot reference Q values

```
if TENSORLY_OK:
    plt.figure(figsize=(9, 4))
    plt.plot(Q_ref, marker='o')
    plt.axhline(q_limit, linestyle='--')
    plt.title("Reference-set Q statistics")
```

```
    plt.xlabel("Sample index")
    plt.ylabel("Q statistic")
    plt.show()
```

4.6 Create new test samples, including abnormal ones

```
X_test = np.zeros((20, n_vars, n_time))
for i in range(20):
    amplitude = 1.0 + 0.05*np.random.randn()
    noise = 0.08*np.random.randn(n_vars, n_time)
    X_test[i] = amplitude * np.outer(base_var, base_time) + noise

# Inject abnormalities in the last few samples
X_test[15:, 2, 5:8] += 0.8
X_test[17:, 0, :] += 0.4

print("Test tensor shape:", X_test.shape)
```

4.7 Simple projection-based approximation using learned factors

For illustration, we project each test sample using the variable and time factors learned from the reference model.

```
if TENSORLY_OK:
    U_sample, U_var, U_time = [np.array(f) for f in factors_ref]

    # Use variable and time factors only for per-sample reconstruction demonstration
    Uv = U_var
    Ut = U_time

    def reconstruct_sample(sample, Uv, Ut):
        # sample shape: (n_vars, n_time)
        core = Uv.T @ sample @ Ut
        recon = Uv @ core @ Ut.T
        return recon

    Q_test = []
    X_test_hat = np.zeros_like(X_test)

    for i in range(X_test.shape[0]):
        recon = reconstruct_sample(X_test[i], Uv, Ut)
        X_test_hat[i] = recon
        resid = X_test[i] - recon
        Q_test.append(np.sum(resid**2))

    Q_test = np.array(Q_test)
```

```
    print("Computed test Q values.")
```

4.8 Plot test Q statistics against the limit

```
if TENSORLY_OK:
    plt.figure(figsize=(9, 4))
    plt.plot(Q_test, marker='o')
    plt.axhline(q_limit, linestyle='--')
    plt.title("Test-set Q statistics")
    plt.xlabel("Test sample index")
    plt.ylabel("Q statistic")
    plt.show()

    flagged = np.where(Q_test > q_limit)[0]
    print("Flagged sample indices:", flagged)
```

4.9 Compare one normal and one abnormal sample

```
if TENSORLY_OK:
    normal_idx = 2
    abnormal_idx = 18

    plt.figure(figsize=(8, 4))
    plt.imshow(X_test[normal_idx], aspect='auto')
    plt.colorbar()
    plt.title(f"Normal-like sample {normal_idx}")
    plt.xlabel("Time")
    plt.ylabel("Variable")
    plt.show()

    plt.figure(figsize=(8, 4))
    plt.imshow(X_test[abnormal_idx], aspect='auto')
    plt.colorbar()
    plt.title(f"Abnormal sample {abnormal_idx}")
    plt.xlabel("Time")
    plt.ylabel("Variable")
    plt.show()
```

4.10 Interpretation

This is the core tensor-SPC logic:

1. Build a low-rank tensor model from normal data.

2. Reconstruct new observations in that modeled subspace.
3. Compute the residual tensor.
4. Use its squared norm as a monitoring statistic.

That residual-energy statistic is the tensor analogue of classical SPE/Q.

5 Synthetic Process Data Tensor Monitoring

This section creates a more process-style example where each sample is a **part** × **feature** × **time** measurement object.

It is intended as a bridge toward the paper and future experimental work.

5.1 Imports

```
import numpy as np
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    from tensorly.tucker_tensor import tucker_to_tensor
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)
```

5.2 Simulate process data

```
np.random.seed(2026)

n_lots = 40
n_features = 6
n_stages = 15

feature_profile = np.array([1.0, 0.7, 1.2, 0.9, 1.1, 0.8])
stage_profile = np.linspace(-1, 1, n_stages)

X = np.zeros((n_lots, n_features, n_stages))
for i in range(n_lots):
    lot_shift = 0.12 * np.random.randn()
```

```
    trend_strength = 1.0 + 0.08 * np.random.randn()
    structured = trend_strength * np.outer(feature_profile, stage_profile)
    noise = 0.10 * np.random.randn(n_features, n_stages)
    X[i] = structured + lot_shift + noise

# add abnormal lots
X[34:, 4, 8:12] += 0.9
X[37:, 1, :] -= 0.5

print("Process tensor shape:", X.shape)
```

5.3 Fit a low-rank tensor model

```
if TENSORLY_OK:
    X_tl = tl.tensor(X, dtype=float)
    rank = [4, 3, 3]
    core, factors = tucker(X_tl, rank=rank)
    X_hat = tucker_to_tensor((core, factors))
    E = np.array(X_tl - X_hat)
    Q = np.sum(E**2, axis=(1, 2))
    print("Finished decomposition.")
else:
    print("Install TensorLy to run this section.")
```

5.4 Plot Q statistics

```
if TENSORLY_OK:
    q_limit = np.percentile(Q[:30], 99)

    plt.figure(figsize=(10, 4))
    plt.plot(Q, marker='o')
    plt.axhline(q_limit, linestyle='--')
    plt.title("Lot-wise Q statistic")
    plt.xlabel("Lot index")
    plt.ylabel("Q")
    plt.show()

    print("Flagged lots:", np.where(Q > q_limit)[0])
```

5.5 Visualize a typical lot and an abnormal lot

```
good_idx = 5
bad_idx = 38
```

```

plt.figure(figsize=(8, 4))
plt.imshow(X[good_idx], aspect='auto')
plt.colorbar()
plt.title(f"Typical lot {good_idx}")
plt.xlabel("Stage")
plt.ylabel("Feature")
plt.show()

plt.figure(figsize=(8, 4))
plt.imshow(X[bad_idx], aspect='auto')
plt.colorbar()
plt.title(f"Abnormal lot {bad_idx}")
plt.xlabel("Stage")
plt.ylabel("Feature")
plt.show()
```

5.6 Residual localization

```
if TENSORLY_OK:
    R_good = X[good_idx] - np.array(X_hat)[good_idx]
    R_bad = X[bad_idx] - np.array(X_hat)[bad_idx]

    plt.figure(figsize=(8, 4))
    plt.imshow(R_good, aspect='auto')
    plt.colorbar()
    plt.title(f"Residual map for lot {good_idx}")
    plt.xlabel("Stage")
    plt.ylabel("Feature")
    plt.show()

    plt.figure(figsize=(8, 4))
    plt.imshow(R_bad, aspect='auto')
    plt.colorbar()
    plt.title(f"Residual map for lot {bad_idx}")
    plt.xlabel("Stage")
    plt.ylabel("Feature")
    plt.show()
```

5.7 Interpretation in this context

This is the kind of workflow that can grow into a paper-quality experimental section:

- define tensorized process observations

- estimate a normal low-rank tensor model
- monitor Q from the residual tensor
- localize unusual structure with residual heatmaps

From here, the framework can be extended toward:

- T^2-style score monitoring
- EWMA/CUSUM on Q
- comparison with vectorized PCA/MSPC
- simulation studies for ARL and power

6 Tensor T^2 Monitoring in Score Space

This chapter adds a **T^2-style monitoring statistic** to the tensor SPC sequence.

Conceptually:

- **Q / SPE** monitors variation left **outside** the modeled tensor subspace.
- **T^2** monitors unusual behavior **inside** the modeled score space.

This chapter uses a practical approach:

1. simulate tensor-valued in-control data
2. fit a Tucker model
3. represent each sample in a lower-dimensional score space
4. compute a Hotelling-style T^2 statistic in that score space
5. compare T^2 and Q on new observations

6.1 Imports

```
import numpy as np
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    from tensorly.tucker_tensor import tucker_to_tensor
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)
```

6.2 Simulate in-control tensor samples

Each sample is a small matrix of **variables** × **time**. The full dataset is therefore a third-order tensor of shape `(samples, variables, time)`.

```
np.random.seed(321)

n_samples = 80
n_vars = 5
n_time = 15

base_time = np.sin(np.linspace(0, 2*np.pi, n_time))
base_var = np.array([1.0, 0.9, 1.1, 1.05, 0.95])

X_ref = np.zeros((n_samples, n_vars, n_time))
for i in range(n_samples):
    amp = 1.0 + 0.06*np.random.randn()
    phase_shift = 0.08*np.random.randn()
    shifted_time = np.sin(np.linspace(0, 2*np.pi, n_time) + phase_shift)
    trend = 0.03*np.random.randn() * np.linspace(-1, 1, n_time)
    noise = 0.08*np.random.randn(n_vars, n_time)
    X_ref[i] = amp * np.outer(base_var, shifted_time) + np.outer(base_var, trend) +
    noise

print("Reference tensor shape:", X_ref.shape)
```

6.3 Fit a Tucker model to the reference tensor

```
if TENSORLY_OK:
    X_ref_tl = tl.tensor(X_ref, dtype=float)
    rank = [4, 3, 3]
    core_ref, factors_ref = tucker(X_ref_tl, rank=rank)
    X_ref_hat = tucker_to_tensor((core_ref, factors_ref))

    print("Core shape:", core_ref.shape)
    for i, f in enumerate(factors_ref):
        print(f"Factor {i} shape:", f.shape)
else:
    print("Install TensorLy to run this section.")
```

6.4 Build a score representation for each sample

A practical way to define tensor scores for monitoring is to use the Tucker factors from the **variable** and **time** modes, project each sample into that lower-dimensional space, and flatten the resulting low-dimensional core slice into a score vector.

```python
if TENSORLY_OK:
    U_sample, U_var, U_time = [np.array(f) for f in factors_ref]

    def sample_score_matrix(sample, U_var, U_time):
        return U_var.T @ sample @ U_time

    def sample_score_vector(sample, U_var, U_time):
        G = sample_score_matrix(sample, U_var, U_time)
        return G.reshape(-1)

    score_vectors_ref = np.array([
        sample_score_vector(X_ref[i], U_var, U_time) for i in range(n_samples)
    ])

    print("Reference score matrix shape:", score_vectors_ref.shape)
```

6.5 Estimate the score-space center and covariance

```python
if TENSORLY_OK:
    mu_scores = score_vectors_ref.mean(axis=0)
    S_scores = np.cov(score_vectors_ref, rowvar=False)
    ridge = 1e-6 * np.eye(S_scores.shape[0])
    S_inv = np.linalg.inv(S_scores + ridge)
```

6.6 Define the T^2 statistic

```python
if TENSORLY_OK:
    def hotelling_t2(score_vec, mu, S_inv):
        d = score_vec - mu
        return float(d.T @ S_inv @ d)

    T2_ref = np.array([hotelling_t2(s, mu_scores, S_inv) for s in score_vectors_ref])
    print(T2_ref[:5])
```

6.7 Define the Q statistic for the same samples

For comparison, compute Q using the reconstruction residual.

```python
if TENSORLY_OK:
    def reconstruct_sample(sample, U_var, U_time):
        G = U_var.T @ sample @ U_time
        return U_var @ G @ U_time.T

```

```
    Q_ref = []
    for i in range(n_samples):
        recon = reconstruct_sample(X_ref[i], U_var, U_time)
        resid = X_ref[i] - recon
        Q_ref.append(np.sum(resid**2))
    Q_ref = np.array(Q_ref)
```

6.8 Set empirical control limits

```
if TENSORLY_OK:
    t2_limit = np.percentile(T2_ref, 99)
    q_limit = np.percentile(Q_ref, 99)
    print("Empirical 99th percentile T² limit:", t2_limit)
    print("Empirical 99th percentile Q limit:", q_limit)
```

6.9 Create test samples with different types of abnormalities

We inject two broad types of changes:

1. **modeled-pattern shift**: changes that mostly affect the low-rank score space
2. **localized patch anomaly**: changes likely to appear more strongly in Q

```
X_test = np.zeros((25, n_vars, n_time))
for i in range(25):
    amp = 1.0 + 0.06*np.random.randn()
    phase_shift = 0.08*np.random.randn()
    shifted_time = np.sin(np.linspace(0, 2*np.pi, n_time) + phase_shift)
    trend = 0.03*np.random.randn() * np.linspace(-1, 1, n_time)
    noise = 0.08*np.random.randn(n_vars, n_time)
    X_test[i] = amp * np.outer(base_var, shifted_time) + np.outer(base_var, trend) +
    noise

X_test[10:15] *= 1.35
X_test[18:, 3, 6:10] += 0.9
X_test[20:, 1, :] -= 0.35
```

6.10 Compute test T^2 and Q

```
if TENSORLY_OK:
    T2_test = []
    Q_test = []
```

```
4
5       for i in range(X_test.shape[0]):
6           s = sample_score_vector(X_test[i], U_var, U_time)
7           T2_test.append(hotelling_t2(s, mu_scores, S_inv))
8           recon = reconstruct_sample(X_test[i], U_var, U_time)
9           resid = X_test[i] - recon
10          Q_test.append(np.sum(resid**2))
11
12      T2_test = np.array(T2_test)
13      Q_test = np.array(Q_test)
```

6.11 Interpretation

Typical interpretation:

- **High T^2, normal-ish Q**: the sample lies in the modeled subspace, but in an unusual region of it.
- **Normal-ish T^2, high Q**: the sample has structure outside the modeled subspace.
- **Both high**: the sample is unusual both within and outside the modeled structure.

This is why T^2 and Q are complementary rather than redundant.

Part III

Comparative and Simulation Studies

7 Tensor SPC vs Vectorized PCA Baseline

This section compares two monitoring approaches on the same synthetic tensor-valued process data:

1. **Tensor-based monitoring** using Tucker-style low-rank structure
2. **Vectorized PCA monitoring** using a flattened representation of each sample

The goal is not to prove a universal winner, but to show how the methods behave differently under controlled abnormalities.

7.1 Imports

```
import numpy as np
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    from tensorly.tucker_tensor import tucker_to_tensor
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)

from sklearn.decomposition import PCA
```

7.2 Simulate structured in-control tensor data

Each sample is a small matrix of shape `(variables, time)`. The full dataset is therefore a third-order tensor:

- mode 0: sample

- mode 1: variable
- mode 2: time

```
np.random.seed(2027)

n_ref = 100
n_test = 40
n_vars = 6
n_time = 20

base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
time_axis = np.linspace(0, 2*np.pi, n_time)

def generate_normal_sample():
    amp = 1.0 + 0.05*np.random.randn()
    phase = 0.10*np.random.randn()
    time_profile = np.sin(time_axis + phase) + 0.35*np.cos(2 * time_axis + 0.5 * phase
    )
    drift = 0.03 * np.random.randn() * np.linspace(-1, 1, n_time)
    interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
    noise = 0.08 * np.random.randn(n_vars, n_time)
    return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
    interaction + noise

X_ref = np.array([generate_normal_sample() for _ in range(n_ref)])
print("Reference tensor shape:", X_ref.shape)
```

7.3 Create test data with several fault types

We inject three broad categories of abnormalities:

- **global score-space shift**: coherent amplitude change
- **localized patch anomaly**: short region disturbance
- **correlation/shape change**: altered structure across time

```
X_test = np.array([generate_normal_sample() for _ in range(n_test)])
fault_labels = np.array(["normal"] * n_test, dtype=object)

# Fault type 1: coherent global shift (likely to affect T^2 strongly)
for i in range(10, 18):
    X_test[i] *= 1.30
    fault_labels[i] = "global_shift"

# Fault type 2: localized patch anomaly (often shows up strongly in Q)
```

```
for i in range(18, 28):
    X_test[i, 3, 7:12] += 0.9
    fault_labels[i] = "patch_anomaly"

# Fault type 3: correlation / shape change
for i in range(28, 40):
    altered = generate_normal_sample()
    altered += 0.35 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
    X_test[i] = altered
    fault_labels[i] = "shape_change"

unique, counts = np.unique(fault_labels, return_counts=True)
dict(zip(unique, counts))
```

7.4 Tensor monitoring model

We use a Tucker decomposition on the reference tensor and then define:

- a **score-space T^2** statistic using low-dimensional score vectors
- a **Q** statistic using residual reconstruction energy

```
if TENSORLY_OK:
    X_ref_tl = tl.tensor(X_ref, dtype=float)
    rank = [5, 3, 3]
    core_ref, factors_ref = tucker(X_ref_tl, rank=rank)

    U_sample, U_var, U_time = [np.array(f) for f in factors_ref]

    def tensor_score_vector(sample, U_var, U_time):
        G = U_var.T @ sample @ U_time
        return G.reshape(-1)

    def tensor_reconstruct(sample, U_var, U_time):
        G = U_var.T @ sample @ U_time
        return U_var @ G @ U_time.T

    tensor_scores_ref = np.array([
        tensor_score_vector(X_ref[i], U_var, U_time) for i in range(n_ref)
    ])

    mu_tensor = tensor_scores_ref.mean(axis=0)
    S_tensor = np.cov(tensor_scores_ref, rowvar=False)
    S_tensor_inv = np.linalg.inv(S_tensor + 1e-6 * np.eye(S_tensor.shape[0]))

    def t2_stat(score_vec, mu, S_inv):
        d = score_vec - mu
```

```
        return float(d.T @ S_inv @ d)

    T2_tensor_ref = np.array([t2_stat(s, mu_tensor, S_tensor_inv) for s in
    tensor_scores_ref])

    Q_tensor_ref = []
    for i in range(n_ref):
        recon = tensor_reconstruct(X_ref[i], U_var, U_time)
        resid = X_ref[i] - recon
        Q_tensor_ref.append(np.sum(resid**2))
    Q_tensor_ref = np.array(Q_tensor_ref)

    t2_tensor_limit = np.percentile(T2_tensor_ref, 99)
    q_tensor_limit = np.percentile(Q_tensor_ref, 99)

    print("Tensor T² limit:", t2_tensor_limit)
    print("Tensor Q limit:", q_tensor_limit)
else:
    print("Install TensorLy to run this section.")
```

7.5 Vectorized PCA baseline

Here each tensor sample is flattened into one long vector. Then we fit PCA to the reference matrix.

We define:

- **PCA T^2** in the retained score space
- **PCA Q/SPE** from reconstruction residuals

```
X_ref_vec = X_ref.reshape(n_ref, -1)
X_test_vec = X_test.reshape(n_test, -1)

n_components = 8
pca = PCA(n_components=n_components)
scores_ref = pca.fit_transform(X_ref_vec)
scores_test = pca.transform(X_test_vec)

mu_pca = scores_ref.mean(axis=0)
S_pca = np.cov(scores_ref, rowvar=False)
S_pca_inv = np.linalg.inv(S_pca + 1e-6 * np.eye(S_pca.shape[0]))

def hotelling_t2(score_vec, mu, S_inv):
    d = score_vec - mu
    return float(d.T @ S_inv @ d)

T2_pca_ref = np.array([hotelling_t2(s, mu_pca, S_pca_inv) for s in scores_ref])
```

```
18
19 X_ref_vec_hat = pca.inverse_transform(scores_ref)
20 Q_pca_ref = np.sum((X_ref_vec - X_ref_vec_hat)**2, axis=1)
21
22 t2_pca_limit = np.percentile(T2_pca_ref, 99)
23 q_pca_limit = np.percentile(Q_pca_ref, 99)
24
25 print("PCA T² limit:", t2_pca_limit)
26 print("PCA Q limit:", q_pca_limit)
```

7.6 Apply both monitoring systems to the test set

```
1  if TENSORLY_OK:
2      T2_tensor_test = []
3      Q_tensor_test = []
4
5      for i in range(n_test):
6          s = tensor_score_vector(X_test[i], U_var, U_time)
7          T2_tensor_test.append(t2_stat(s, mu_tensor, S_tensor_inv))
8
9          recon = tensor_reconstruct(X_test[i], U_var, U_time)
10         resid = X_test[i] - recon
11         Q_tensor_test.append(np.sum(resid**2))
12
13     T2_tensor_test = np.array(T2_tensor_test)
14     Q_tensor_test = np.array(Q_tensor_test)
15
16 T2_pca_test = np.array([hotelling_t2(s, mu_pca, S_pca_inv) for s in scores_test])
17 X_test_vec_hat = pca.inverse_transform(scores_test)
18 Q_pca_test = np.sum((X_test_vec - X_test_vec_hat)**2, axis=1)
19
20 print("Computed test statistics for both methods.")
```

7.7 Plot tensor statistics

```
1  if TENSORLY_OK:
2      plt.figure(figsize=(10, 4))
3      plt.plot(T2_tensor_test, marker='o')
4      plt.axhline(t2_tensor_limit, linestyle='--')
5      plt.title("Tensor T² on test data")
6      plt.xlabel("Test sample index")
7      plt.ylabel("Tensor T²")
8      plt.show()
9
10     plt.figure(figsize=(10, 4))
```

```
    plt.plot(Q_tensor_test, marker='o')
    plt.axhline(q_tensor_limit, linestyle='--')
    plt.title("Tensor Q on test data")
    plt.xlabel("Test sample index")
    plt.ylabel("Tensor Q")
    plt.show()
```

7.8 Plot PCA baseline statistics

```
plt.figure(figsize=(10, 4))
plt.plot(T2_pca_test, marker='o')
plt.axhline(t2_pca_limit, linestyle='--')
plt.title("Vectorized PCA T$^2$ on test data")
plt.xlabel("Test sample index")
plt.ylabel("PCA T$^2$")
plt.show()

plt.figure(figsize=(10, 4))
plt.plot(Q_pca_test, marker='o')
plt.axhline(q_pca_limit, linestyle='--')
plt.title("Vectorized PCA Q on test data")
plt.xlabel("Test sample index")
plt.ylabel("PCA Q")
plt.show()
```

7.9 Flagged samples by method

```
results = {}

if TENSORLY_OK:
    results["tensor_T2"] = np.where(T2_tensor_test > t2_tensor_limit)[0]
    results["tensor_Q"] = np.where(Q_tensor_test > q_tensor_limit)[0]

results["pca_T2"] = np.where(T2_pca_test > t2_pca_limit)[0]
results["pca_Q"] = np.where(Q_pca_test > q_pca_limit)[0]

results
```

7.10 Detection summary by fault type

```
def summarize_detection(indices, fault_labels):
    flagged = np.zeros(len(fault_labels), dtype=bool)
    flagged[indices] = True
```

```
    summary = {}
    for label in np.unique(fault_labels):
        mask = fault_labels == label
        summary[label] = {
            "count": int(mask.sum()),
            "flagged": int(flagged[mask].sum()),
            "rate": float(flagged[mask].mean())
        }
    return summary

summary = {}
if TENSORLY_OK:
    summary["tensor_T2"] = summarize_detection(results["tensor_T2"], fault_labels)
    summary["tensor_Q"] = summarize_detection(results["tensor_Q"], fault_labels)

summary["pca_T2"] = summarize_detection(results["pca_T2"], fault_labels)
summary["pca_Q"] = summarize_detection(results["pca_Q"], fault_labels)

summary
```

7.11 Compare methods in scatter plots

```
if TENSORLY_OK:
    plt.figure(figsize=(6, 5))
    for label in np.unique(fault_labels):
        mask = fault_labels == label
        plt.scatter(T2_tensor_test[mask], Q_tensor_test[mask], label=label)
    plt.axvline(t2_tensor_limit, linestyle='--')
    plt.axhline(q_tensor_limit, linestyle='--')
    plt.xlabel("Tensor T²")
    plt.ylabel("Tensor Q")
    plt.title("Tensor monitoring space")
    plt.legend()
    plt.show()

plt.figure(figsize=(6, 5))
for label in np.unique(fault_labels):
    mask = fault_labels == label
    plt.scatter(T2_pca_test[mask], Q_pca_test[mask], label=label)
plt.axvline(t2_pca_limit, linestyle='--')
plt.axhline(q_pca_limit, linestyle='--')
plt.xlabel("PCA T²")
plt.ylabel("PCA Q")
plt.title("Vectorized PCA monitoring space")
plt.legend()
plt.show()
```

7.12 Inspect a few example samples

```
example_indices = [5, 12, 22, 34]
example_titles = {
    5: "Normal example",
    12: "Global shift example",
    22: "Patch anomaly example",
    34: "Shape change example"
}

for idx in example_indices:
    plt.figure(figsize=(8, 4))
    plt.imshow(X_test[idx], aspect='auto')
    plt.colorbar()
    plt.title(f"{example_titles[idx]} (index {idx})")
    plt.xlabel("Time")
    plt.ylabel("Variable")
    plt.show()
```

7.13 Interpretation

What to look for:

- The **tensor method** may better preserve structured multiway relationships.
- The **vectorized PCA baseline** may still work well, especially for broad global faults.
- Localized or mode-specific changes may separate differently under tensor Q and tensor T^2.

This section is useful because it gives you a fair baseline. For tensor SPC to contribute meaningfully, it should show some advantage in structure-aware detection, interpretability, or robustness.

7.14 Ideas for extending this analysis

Strong next extensions include:

- repeat this over many Monte Carlo runs
- estimate average detection rates by fault type
- vary tensor rank and PCA component count
- compare false alarm behavior
- add EWMA/CUSUM versions of both methods

- add residual heatmaps for localization

That extension would move the analysis from demonstration toward publishable simulation evidence.

8 Monte Carlo Comparison of Tensor SPC and PCA

This section extends the earlier comparison notebook into a **Monte Carlo simulation study**.

Instead of looking at one example only, we repeat the experiment many times and summarize:

- false alarm behavior
- detection rates
- fault-type-specific performance
- average performance differences between methods

The goal is to move from demonstration toward simulation evidence.

8.1 Imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    from tensorly.tucker_tensor import tucker_to_tensor
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)

from sklearn.decomposition import PCA
```

8.2 Simulation design

We simulate:

- a reference in-control tensor dataset
- a test set containing normal samples and several fault types
- tensor monitoring statistics: T^2 and Q
- vectorized PCA monitoring statistics: T^2 and Q

Then we repeat the process across many Monte Carlo trials.

```
N_RUNS = 50

n_ref = 80
n_test_normal = 20
n_test_per_fault = 12

n_vars = 6
n_time = 20

tensor_rank = [5, 3, 3]
pca_components = 8

alpha_percentile = 99

base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
time_axis = np.linspace(0, 2*np.pi, n_time)

print("Monte Carlo runs:", N_RUNS)
```

8.3 Data generation functions

```
def generate_normal_sample(rng):
    amp = 1.0 + 0.05 * rng.standard_normal()
    phase = 0.10 * rng.standard_normal()
    time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
    phase)
    drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
    interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
    noise = 0.08 * rng.standard_normal((n_vars, n_time))
    return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
    interaction + noise

def inject_global_shift(sample):
    return 1.30 * sample

def inject_patch_anomaly(sample):
    x = sample.copy()
    x[3, 7:12] += 0.9
```

```
    return x

def inject_shape_change(sample):
    x = sample.copy()
    x += 0.35 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
    return x
```

8.4 Monitoring helper functions

```
def hotelling_t2(score_vec, mu, S_inv):
    d = score_vec - mu
    return float(d.T @ S_inv @ d)

def summarize_rate(flags):
    flags = np.asarray(flags, dtype=float)
    return {
        "mean": float(flags.mean()),
        "std": float(flags.std(ddof=1)) if len(flags) > 1 else 0.0
    }
```

8.5 One Monte Carlo run

```
def run_one_simulation(seed):
    rng = np.random.default_rng(seed)

    # Reference data
    X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])

    # Test data by fault type
    X_test_normal = np.array([generate_normal_sample(rng) for _ in range(n_test_normal
    )])
    X_test_global = np.array([inject_global_shift(generate_normal_sample(rng)) for _
    in range(n_test_per_fault)])
    X_test_patch = np.array([inject_patch_anomaly(generate_normal_sample(rng)) for _
    in range(n_test_per_fault)])
    X_test_shape = np.array([inject_shape_change(generate_normal_sample(rng)) for _ in
     range(n_test_per_fault)])

    # ------------------------------
    # Tensor monitoring
    # ------------------------------
    tensor_results = None
    if TENSORLY_OK:
        X_ref_tl = tl.tensor(X_ref, dtype=float)
        core_ref, factors_ref = tucker(X_ref_tl, rank=tensor_rank)
```

```
        U_sample, U_var, U_time = [np.array(f) for f in factors_ref]

        def tensor_score_vector(sample):
            G = U_var.T @ sample @ U_time
            return G.reshape(-1)

        def tensor_reconstruct(sample):
            G = U_var.T @ sample @ U_time
            return U_var @ G @ U_time.T

        tensor_scores_ref = np.array([tensor_score_vector(X_ref[i]) for i in range(
    n_ref)])
        mu_tensor = tensor_scores_ref.mean(axis=0)
        S_tensor = np.cov(tensor_scores_ref, rowvar=False)
        S_tensor_inv = np.linalg.inv(S_tensor + 1e-6 * np.eye(S_tensor.shape[0]))

        T2_tensor_ref = np.array([hotelling_t2(s, mu_tensor, S_tensor_inv) for s in
    tensor_scores_ref])
        Q_tensor_ref = np.array([
            np.sum((X_ref[i] - tensor_reconstruct(X_ref[i]))**2) for i in range(n_ref)
        ])

        t2_tensor_limit = np.percentile(T2_tensor_ref, alpha_percentile)
        q_tensor_limit = np.percentile(Q_tensor_ref, alpha_percentile)

        def tensor_flags(X_test):
            scores = np.array([tensor_score_vector(x) for x in X_test])
            T2 = np.array([hotelling_t2(s, mu_tensor, S_tensor_inv) for s in scores])
            Q = np.array([np.sum((x - tensor_reconstruct(x))**2) for x in X_test])
            return {
                "T2": T2 > t2_tensor_limit,
                "Q": Q > q_tensor_limit
            }

        tensor_results = {
            "normal": tensor_flags(X_test_normal),
            "global_shift": tensor_flags(X_test_global),
            "patch_anomaly": tensor_flags(X_test_patch),
            "shape_change": tensor_flags(X_test_shape)
        }

    # ----------------------------
    # PCA baseline
    # ----------------------------
    X_ref_vec = X_ref.reshape(n_ref, -1)
    pca = PCA(n_components=pca_components)
    scores_ref = pca.fit_transform(X_ref_vec)

    mu_pca = scores_ref.mean(axis=0)
    S_pca = np.cov(scores_ref, rowvar=False)
```

```
    S_pca_inv = np.linalg.inv(S_pca + 1e-6 * np.eye(S_pca.shape[0]))

    T2_pca_ref = np.array([hotelling_t2(s, mu_pca, S_pca_inv) for s in scores_ref])
    X_ref_hat = pca.inverse_transform(scores_ref)
    Q_pca_ref = np.sum((X_ref_vec - X_ref_hat)**2, axis=1)

    t2_pca_limit = np.percentile(T2_pca_ref, alpha_percentile)
    q_pca_limit = np.percentile(Q_pca_ref, alpha_percentile)

    def pca_flags(X_test):
        Xv = X_test.reshape(X_test.shape[0], -1)
        scores = pca.transform(Xv)
        T2 = np.array([hotelling_t2(s, mu_pca, S_pca_inv) for s in scores])
        Xhat = pca.inverse_transform(scores)
        Q = np.sum((Xv - Xhat)**2, axis=1)
        return {
            "T2": T2 > t2_pca_limit,
            "Q": Q > q_pca_limit
        }

    pca_results = {
        "normal": pca_flags(X_test_normal),
        "global_shift": pca_flags(X_test_global),
        "patch_anomaly": pca_flags(X_test_patch),
        "shape_change": pca_flags(X_test_shape)
    }

    return tensor_results, pca_results
```

8.6 Run the Monte Carlo study

```
records = []

for run in range(N_RUNS):
    tensor_results, pca_results = run_one_simulation(seed=1000 + run)

    fault_types = ["normal", "global_shift", "patch_anomaly", "shape_change"]

    if tensor_results is not None:
        for fault in fault_types:
            for stat in ["T2", "Q"]:
                records.append({
                    "run": run,
                    "method": "tensor",
                    "statistic": stat,
                    "fault_type": fault,
                    "flag_rate": float(np.mean(tensor_results[fault][stat]))
                })
```

```
    for fault in fault_types:
        for stat in ["T2", "Q"]:
            records.append({
                "run": run,
                "method": "pca",
                "statistic": stat,
                "fault_type": fault,
                "flag_rate": float(np.mean(pca_results[fault][stat]))
            })

results_df = pd.DataFrame(records)
results_df.head()
```

8.7 Aggregate results

```
summary_df = (
    results_df
    .groupby(["method", "statistic", "fault_type"])["flag_rate"]
    .agg(["mean", "std"])
    .reset_index()
)

summary_df
```

8.8 Separate false alarm and detection summaries

```
false_alarm_df = summary_df[summary_df["fault_type"] == "normal"].copy()
detection_df = summary_df[summary_df["fault_type"] != "normal"].copy()

print("False alarm summary:")
display(false_alarm_df)

print("\nDetection summary:")
display(detection_df)
```

8.9 Plot false alarm rates

```
plot_df = false_alarm_df.copy()
labels = plot_df["method"] + "_" + plot_df["statistic"]

plt.figure(figsize=(8, 4))
plt.bar(labels, plot_df["mean"], yerr=plot_df["std"], capsize=5)
```

```
plt.ylabel("Average false alarm rate")
plt.title("Monte Carlo false alarm comparison")
plt.show()
```

8.10 Plot detection rates by fault type

```
faults = ["global_shift", "patch_anomaly", "shape_change"]

for fault in faults:
    sub = detection_df[detection_df["fault_type"] == fault].copy()
    labels = sub["method"] + "_" + sub["statistic"]

    plt.figure(figsize=(8, 4))
    plt.bar(labels, sub["mean"], yerr=sub["std"], capsize=5)
    plt.ylim(0, 1.05)
    plt.ylabel("Average detection rate")
    plt.title(f"Detection rate comparison - {fault}")
    plt.show()
```

8.11 Pivot tables for easier comparison

```
pivot_mean = summary_df.pivot_table(
    index="fault_type",
    columns=["method", "statistic"],
    values="mean"
)

pivot_std = summary_df.pivot_table(
    index="fault_type",
    columns=["method", "statistic"],
    values="std"
)

print("Mean flag rates:")
display(pivot_mean)

print("\nStandard deviations:")
display(pivot_std)
```

8.12 A compact method ranking view

```
rank_view = summary_df.copy()
```

```
rank_view["metric_type"] = np.where(rank_view["fault_type"] == "normal", "false_alarm",
    "detection")
rank_view
```

8.13 Interpretation guide

Typical reading of the results:

- For **normal** data, lower flag rates are better.
- For **fault** data, higher flag rates are better.
- **Tensor T^2** may respond strongly to coherent structured changes.
- **Tensor Q** may respond strongly to localized out-of-subspace faults.
- **PCA T^2/Q** provide the fair vectorized baseline.

The point is not that one method must win everywhere. The useful result is seeing **which statistic is sensitive to which fault type**.

8.14 Extensions for a stronger research study

Strong next extensions include:

- increase `N_RUNS`
- vary tensor rank and PCA components systematically
- estimate confidence intervals for mean detection rates
- add EWMA/CUSUM versions of T^2 and Q
- add detection-delay experiments instead of one-shot detection
- compare additional fault magnitudes
- store all results to CSV for later reporting

8.15 Optional export

```
# Uncomment to save summaries
# summary_df.to_csv("monte_carlo_summary.csv", index=False)
# results_df.to_csv("monte_carlo_all_runs.csv", index=False)

print("Export section ready.")
```

9 ARL and Detection Delay: Tensor SPC vs PCA

This section shifts from one-shot detection to **sequential monitoring**.

It estimates:

- **ARL0 (in-control average run length)**: average number of samples until a false alarm
- **Detection delay**: average number of post-change samples until alarm after a fault occurs

We compare:

- Tensor T^2
- Tensor Q
- Vectorized PCA T^2
- Vectorized PCA Q

This is much closer to classical SPC evaluation logic.

9.1 Imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)

from sklearn.decomposition import PCA
```

9.2 Simulation settings

```
N_RUNS = 40

# Reference data used to fit each monitoring model
n_ref = 100

# Sequential monitoring horizon
max_run_length = 300

# Change point for out-of-control sequences
change_point = 100

n_vars = 6
n_time = 20

tensor_rank = [5, 3, 3]
pca_components = 8
alpha_percentile = 99

base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
time_axis = np.linspace(0, 2*np.pi, n_time)

print("Runs:", N_RUNS)
print("Change point:", change_point)
print("Monitoring horizon:", max_run_length)
```

9.3 Data generation

```
def generate_normal_sample(rng):
    amp = 1.0 + 0.05 * rng.standard_normal()
    phase = 0.10 * rng.standard_normal()
    time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
    phase)
    drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
    interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
    noise = 0.08 * rng.standard_normal((n_vars, n_time))
    return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
    interaction + noise

def inject_global_shift(sample):
    return 1.25 * sample

def inject_patch_anomaly(sample):
    x = sample.copy()
    x[3, 7:12] += 0.9
    return x
```

```

def inject_shape_change(sample):
    x = sample.copy()
    x += 0.35 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
    return x
```

9.4 Monitoring model fitting

```
def hotelling_t2(score_vec, mu, S_inv):
    d = score_vec - mu
    return float(d.T @ S_inv @ d)

def fit_monitoring_models(X_ref):
    models = {}

    # PCA baseline
    X_ref_vec = X_ref.reshape(X_ref.shape[0], -1)
    pca = PCA(n_components=pca_components)
    scores_ref = pca.fit_transform(X_ref_vec)

    mu_pca = scores_ref.mean(axis=0)
    S_pca = np.cov(scores_ref, rowvar=False)
    S_pca_inv = np.linalg.inv(S_pca + 1e-6 * np.eye(S_pca.shape[0]))

    T2_pca_ref = np.array([hotelling_t2(s, mu_pca, S_pca_inv) for s in scores_ref])
    X_ref_hat = pca.inverse_transform(scores_ref)
    Q_pca_ref = np.sum((X_ref_vec - X_ref_hat)**2, axis=1)

    models["pca"] = {
        "pca": pca,
        "mu": mu_pca,
        "S_inv": S_pca_inv,
        "t2_limit": np.percentile(T2_pca_ref, alpha_percentile),
        "q_limit": np.percentile(Q_pca_ref, alpha_percentile)
    }

    # Tensor monitoring
    if TENSORLY_OK:
        X_ref_tl = tl.tensor(X_ref, dtype=float)
        core_ref, factors_ref = tucker(X_ref_tl, rank=tensor_rank)
        U_sample, U_var, U_time = [np.array(f) for f in factors_ref]

        def tensor_score_vector(sample):
            G = U_var.T @ sample @ U_time
            return G.reshape(-1)

        def tensor_reconstruct(sample):
            G = U_var.T @ sample @ U_time
```

```
            return U_var @ G @ U_time.T

        tensor_scores_ref = np.array([tensor_score_vector(X_ref[i]) for i in range(X_ref.shape[0])])
        mu_tensor = tensor_scores_ref.mean(axis=0)
        S_tensor = np.cov(tensor_scores_ref, rowvar=False)
        S_tensor_inv = np.linalg.inv(S_tensor + 1e-6 * np.eye(S_tensor.shape[0]))

        T2_tensor_ref = np.array([hotelling_t2(s, mu_tensor, S_tensor_inv) for s in tensor_scores_ref])
        Q_tensor_ref = np.array([
            np.sum((X_ref[i] - tensor_reconstruct(X_ref[i]))**2) for i in range(X_ref.shape[0])
        ])

        models["tensor"] = {
            "U_var": U_var,
            "U_time": U_time,
            "mu": mu_tensor,
            "S_inv": S_tensor_inv,
            "t2_limit": np.percentile(T2_tensor_ref, alpha_percentile),
            "q_limit": np.percentile(Q_tensor_ref, alpha_percentile)
        }

    return models
```

9.5 Online statistics for one sample

```
def compute_pca_stats(sample, model):
    x_vec = sample.reshape(1, -1)
    score = model["pca"].transform(x_vec)[0]
    t2 = hotelling_t2(score, model["mu"], model["S_inv"])
    x_hat = model["pca"].inverse_transform(score.reshape(1, -1))[0]
    q = float(np.sum((x_vec[0] - x_hat)**2))
    return t2, q

def compute_tensor_stats(sample, model):
    U_var = model["U_var"]
    U_time = model["U_time"]
    G = U_var.T @ sample @ U_time
    score = G.reshape(-1)
    t2 = hotelling_t2(score, model["mu"], model["S_inv"])
    recon = U_var @ G @ U_time.T
    q = float(np.sum((sample - recon)**2))
    return t2, q
```

9.6 Run-length logic

```python
def first_alarm_time(sequence_stats, limit):
    alarms = np.where(sequence_stats > limit)[0]
    if len(alarms) == 0:
        return None
    return int(alarms[0] + 1)  # 1-based time index

def detection_delay(sequence_stats, limit, change_point):
    post = np.where(sequence_stats[change_point:] > limit)[0]
    if len(post) == 0:
        return None
    return int(post[0] + 1)  # delay after the change point
```

9.7 Simulate one in-control run for ARL0

```python
def simulate_in_control_run(seed):
    rng = np.random.default_rng(seed)
    X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
    models = fit_monitoring_models(X_ref)

    results = {}

    # PCA
    pca_t2 = []
    pca_q = []
    for _ in range(max_run_length):
        x = generate_normal_sample(rng)
        t2, q = compute_pca_stats(x, models["pca"])
        pca_t2.append(t2)
        pca_q.append(q)

    results["pca_T2_run_length"] = first_alarm_time(np.array(pca_t2), models["pca"]["
    t2_limit"])
    results["pca_Q_run_length"] = first_alarm_time(np.array(pca_q), models["pca"]["
    q_limit"])

    # Tensor
    if "tensor" in models:
        tensor_t2 = []
        tensor_q = []
        for _ in range(max_run_length):
            x = generate_normal_sample(rng)
            t2, q = compute_tensor_stats(x, models["tensor"])
            tensor_t2.append(t2)
            tensor_q.append(q)

```

```
        results["tensor_T2_run_length"] = first_alarm_time(np.array(tensor_t2), models["tensor"]["t2_limit"])
        results["tensor_Q_run_length"] = first_alarm_time(np.array(tensor_q), models["tensor"]["q_limit"])

    return results
```

9.8 Simulate one out-of-control run for detection delay

```
def simulate_out_of_control_run(seed, fault_type):
    rng = np.random.default_rng(seed)
    X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
    models = fit_monitoring_models(X_ref)

    def apply_fault(x):
        if fault_type == "global_shift":
            return inject_global_shift(x)
        elif fault_type == "patch_anomaly":
            return inject_patch_anomaly(x)
        elif fault_type == "shape_change":
            return inject_shape_change(x)
        else:
            raise ValueError("Unknown fault type")

    results = {}

    # PCA
    pca_t2 = []
    pca_q = []
    for t in range(max_run_length):
        x = generate_normal_sample(rng)
        if t >= change_point:
            x = apply_fault(x)
        t2, q = compute_pca_stats(x, models["pca"])
        pca_t2.append(t2)
        pca_q.append(q)

    results["pca_T2_delay"] = detection_delay(np.array(pca_t2), models["pca"]["t2_limit"], change_point)
    results["pca_Q_delay"] = detection_delay(np.array(pca_q), models["pca"]["q_limit"], change_point)

    # Tensor
    if "tensor" in models:
        tensor_t2 = []
        tensor_q = []
        for t in range(max_run_length):
            x = generate_normal_sample(rng)
```

```
            if t >= change_point:
                x = apply_fault(x)
            t2, q = compute_tensor_stats(x, models["tensor"])
            tensor_t2.append(t2)
            tensor_q.append(q)

        results["tensor_T2_delay"] = detection_delay(np.array(tensor_t2), models["tensor"]["t2_limit"], change_point)
        results["tensor_Q_delay"] = detection_delay(np.array(tensor_q), models["tensor"]["q_limit"], change_point)

    return results
```

9.9 Estimate ARL0

```
arl_records = []

for run in range(N_RUNS):
    res = simulate_in_control_run(seed=2000 + run)
    for key, value in res.items():
        arl_records.append({
            "run": run,
            "method_stat": key.replace("_run_length", ""),
            "run_length": max_run_length if value is None else value
        })

arl_df = pd.DataFrame(arl_records)
arl_summary = arl_df.groupby("method_stat")["run_length"].agg(["mean", "std"]).reset_index()

print("Estimated ARL0 summary:")
display(arl_summary)
```

9.10 Plot ARL0

```
plt.figure(figsize=(8, 4))
plt.bar(arl_summary["method_stat"], arl_summary["mean"], yerr=arl_summary["std"], capsize=5)
plt.ylabel("Estimated ARL0")
plt.title("In-control average run length comparison")
plt.show()
```

9.11 Estimate detection delay for each fault type

```python
fault_types = ["global_shift", "patch_anomaly", "shape_change"]
delay_records = []

for fault in fault_types:
    for run in range(N_RUNS):
        res = simulate_out_of_control_run(seed=5000 + 100*run, fault_type=fault)
        for key, value in res.items():
            delay_records.append({
                "fault_type": fault,
                "run": run,
                "method_stat": key.replace("_delay", ""),
                "delay": (max_run_length - change_point) if value is None else value
            })

delay_df = pd.DataFrame(delay_records)
delay_summary = delay_df.groupby(["fault_type", "method_stat"])["delay"].agg(["mean", "std"]).reset_index()

print("Detection-delay summary:")
display(delay_summary)
```

9.12 Plot detection delay by fault type

```python
for fault in fault_types:
    sub = delay_summary[delay_summary["fault_type"] == fault]

    plt.figure(figsize=(8, 4))
    plt.bar(sub["method_stat"], sub["mean"], yerr=sub["std"], capsize=5)
    plt.ylabel("Average detection delay")
    plt.title(f"Detection delay comparison - {fault}")
    plt.show()
```

9.13 Pivot tables

```python
arl_pivot = arl_summary.set_index("method_stat")
delay_pivot = delay_summary.pivot_table(index="fault_type", columns="method_stat", values="mean")

print("ARL0 pivot:")
display(arl_pivot)

print("\nDetection delay pivot:")
display(delay_pivot)
```

9.14 Example sequential run visualization

```
example_seed = 9999
example_fault = "patch_anomaly"

rng = np.random.default_rng(example_seed)
X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
models = fit_monitoring_models(X_ref)

def apply_fault(x, fault_type):
    if fault_type == "global_shift":
        return inject_global_shift(x)
    elif fault_type == "patch_anomaly":
        return inject_patch_anomaly(x)
    elif fault_type == "shape_change":
        return inject_shape_change(x)
    else:
        return x

pca_t2_seq, pca_q_seq = [], []
tensor_t2_seq, tensor_q_seq = [], []

for t in range(max_run_length):
    x = generate_normal_sample(rng)
    if t >= change_point:
        x = apply_fault(x, example_fault)

    t2_pca, q_pca = compute_pca_stats(x, models["pca"])
    pca_t2_seq.append(t2_pca)
    pca_q_seq.append(q_pca)

    if "tensor" in models:
        t2_tensor, q_tensor = compute_tensor_stats(x, models["tensor"])
        tensor_t2_seq.append(t2_tensor)
        tensor_q_seq.append(q_tensor)

plt.figure(figsize=(10, 4))
plt.plot(pca_t2_seq, label="PCA T²")
plt.axhline(models["pca"]["t2_limit"], linestyle='--')
plt.axvline(change_point, linestyle=':')
plt.title(f"Example sequential run - PCA T² ({example_fault})")
plt.xlabel("Time index")
plt.ylabel("Statistic")
plt.legend()
plt.show()

plt.figure(figsize=(10, 4))
plt.plot(pca_q_seq, label="PCA Q")
plt.axhline(models["pca"]["q_limit"], linestyle='--')
```

```
plt.axvline(change_point, linestyle=':')
plt.title(f"Example sequential run - PCA Q ({example_fault})")
plt.xlabel("Time index")
plt.ylabel("Statistic")
plt.legend()
plt.show()

if "tensor" in models:
    plt.figure(figsize=(10, 4))
    plt.plot(tensor_t2_seq, label="Tensor T²")
    plt.axhline(models["tensor"]["t2_limit"], linestyle='--')
    plt.axvline(change_point, linestyle=':')
    plt.title(f"Example sequential run - Tensor T² ({example_fault})")
    plt.xlabel("Time index")
    plt.ylabel("Statistic")
    plt.legend()
    plt.show()

    plt.figure(figsize=(10, 4))
    plt.plot(tensor_q_seq, label="Tensor Q")
    plt.axhline(models["tensor"]["q_limit"], linestyle='--')
    plt.axvline(change_point, linestyle=':')
    plt.title(f"Example sequential run - Tensor Q ({example_fault})")
    plt.xlabel("Time index")
    plt.ylabel("Statistic")
    plt.legend()
    plt.show()
```

9.15 Interpretation

How to read the results:

- **Higher ARL0** means fewer false alarms in the in-control state.
- **Lower detection delay** means faster detection after a change occurs.
- A method may have a strong tradeoff: good ARL0 but slower detection, or fast detection but more false alarms.
- Different statistics may be sensitive to different fault structures:
- coherent global changes may hit $\mathbf{T^2}$ harder
- localized unmodeled changes may hit **Q** harder

This is why ARL and delay are both important. A statistic that detects well in one-shot tests may still be poor sequentially.

9.16 Extensions

Good next extensions include:

- calibrate limits to target a common ARL0 across methods
- add EWMA/CUSUM versions
- vary fault magnitude and change-point location
- compute detection probability within a fixed delay window
- export all run-level results for reporting

9.17 Optional export

```
# arl_df.to_csv("arl_run_lengths.csv", index=False)
# delay_df.to_csv("detection_delays.csv", index=False)

print("Export section ready.")
```

10 Calibrating Limits to a Common Target ARL0

This section makes the comparison between monitoring methods more fair by **calibrating their control limits to the same target in-control average run length (ARL0)**.

Why this matters:

- If one method uses a much looser limit, it may look better only because it alarms less often.
- If another method uses a tighter limit, it may detect faster only because it has more false alarms.

A better comparison is:

1. choose a target ARL0
2. calibrate each method/statistic to approximately achieve that ARL0
3. compare detection delay and detection probability under faults

10.1 Imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)

from sklearn.decomposition import PCA
```

10.2 Simulation settings

```
N_RUNS = 30

n_ref = 100
max_run_length = 400
change_point = 120

n_vars = 6
n_time = 20

tensor_rank = [5, 3, 3]
pca_components = 8

target_arl0 = 200
calibration_sequences = 80

base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
time_axis = np.linspace(0, 2*np.pi, n_time)

print("Target ARL0:", target_arl0)
print("Calibration sequences per method:", calibration_sequences)
```

10.3 Data generation

```
def generate_normal_sample(rng):
    amp = 1.0 + 0.05 * rng.standard_normal()
    phase = 0.10 * rng.standard_normal()
    time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
    phase)
    drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
    interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
    noise = 0.08 * rng.standard_normal((n_vars, n_time))
    return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
    interaction + noise

def inject_global_shift(sample):
    return 1.25 * sample

def inject_patch_anomaly(sample):
    x = sample.copy()
    x[3, 7:12] += 0.9
    return x

def inject_shape_change(sample):
    x = sample.copy()
    x += 0.35 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
```

```
    return x
```

10.4 Model fitting

```
def hotelling_t2(score_vec, mu, S_inv):
    d = score_vec - mu
    return float(d.T @ S_inv @ d)

def fit_models(X_ref):
    models = {}

    # PCA
    X_ref_vec = X_ref.reshape(X_ref.shape[0], -1)
    pca = PCA(n_components=pca_components)
    scores_ref = pca.fit_transform(X_ref_vec)
    mu_pca = scores_ref.mean(axis=0)
    S_pca = np.cov(scores_ref, rowvar=False)
    S_pca_inv = np.linalg.inv(S_pca + 1e-6 * np.eye(S_pca.shape[0]))

    models["pca"] = {
        "pca": pca,
        "mu": mu_pca,
        "S_inv": S_pca_inv
    }

    # Tensor
    if TENSORLY_OK:
        X_ref_tl = tl.tensor(X_ref, dtype=float)
        core_ref, factors_ref = tucker(X_ref_tl, rank=tensor_rank)
        U_sample, U_var, U_time = [np.array(f) for f in factors_ref]

        tensor_scores_ref = np.array([
            (U_var.T @ X_ref[i] @ U_time).reshape(-1) for i in range(X_ref.shape[0])
        ])
        mu_tensor = tensor_scores_ref.mean(axis=0)
        S_tensor = np.cov(tensor_scores_ref, rowvar=False)
        S_tensor_inv = np.linalg.inv(S_tensor + 1e-6 * np.eye(S_tensor.shape[0]))

        models["tensor"] = {
            "U_var": U_var,
            "U_time": U_time,
            "mu": mu_tensor,
            "S_inv": S_tensor_inv
        }

    return models
```

10.5 Statistic functions

```python
def pca_stats(sample, model):
    x_vec = sample.reshape(1, -1)
    score = model["pca"].transform(x_vec)[0]
    t2 = hotelling_t2(score, model["mu"], model["S_inv"])
    x_hat = model["pca"].inverse_transform(score.reshape(1, -1))[0]
    q = float(np.sum((x_vec[0] - x_hat)**2))
    return t2, q

def tensor_stats(sample, model):
    U_var = model["U_var"]
    U_time = model["U_time"]
    G = U_var.T @ sample @ U_time
    score = G.reshape(-1)
    t2 = hotelling_t2(score, model["mu"], model["S_inv"])
    recon = U_var @ G @ U_time.T
    q = float(np.sum((sample - recon)**2))
    return t2, q
```

10.6 Run-length utilities

```python
def first_alarm_time(sequence_stats, limit):
    alarms = np.where(sequence_stats > limit)[0]
    if len(alarms) == 0:
        return max_run_length
    return int(alarms[0] + 1)

def estimate_arl0_from_limit(generate_stats_fn, limit, n_sequences, rng):
    run_lengths = []
    for _ in range(n_sequences):
        stats = generate_stats_fn(rng)
        rl = first_alarm_time(stats, limit)
        run_lengths.append(rl)
    return float(np.mean(run_lengths)), np.array(run_lengths)

def calibrate_limit_by_quantile(generate_stats_fn, candidate_quantiles, n_sequences,
    rng):
    best = None
    results = []

    # Use one pool of calibration sequences for stable comparison
    precomputed = [generate_stats_fn(rng) for _ in range(n_sequences)]

    for q in candidate_quantiles:
        # build limit from pooled in-control stats
        pooled = np.concatenate(precomputed)
```

```
        limit = np.quantile(pooled, q)

        run_lengths = np.array([first_alarm_time(seq, limit) for seq in precomputed])
        arl0 = float(np.mean(run_lengths))

        results.append((q, limit, arl0))

        if best is None or abs(arl0 - target_arl0) < abs(best[2] - target_arl0):
            best = (q, limit, arl0)

    return best, pd.DataFrame(results, columns=["quantile", "limit", "arl0"])
```

10.7 Build one fitted monitoring system and calibrate limits

```
def calibrate_one_system(seed):
    rng = np.random.default_rng(seed)

    X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
    models = fit_models(X_ref)

    systems = {}

    # Candidate quantiles for limit calibration
    candidate_quantiles = np.linspace(0.95, 0.9995, 40)

    # PCA generators
    def pca_t2_sequence(inner_rng):
        vals = []
        for _ in range(max_run_length):
            x = generate_normal_sample(inner_rng)
            t2, q = pca_stats(x, models["pca"])
            vals.append(t2)
        return np.array(vals)

    def pca_q_sequence(inner_rng):
        vals = []
        for _ in range(max_run_length):
            x = generate_normal_sample(inner_rng)
            t2, q = pca_stats(x, models["pca"])
            vals.append(q)
        return np.array(vals)

    best_t2, df_t2 = calibrate_limit_by_quantile(pca_t2_sequence, candidate_quantiles,
     calibration_sequences, rng)
    best_q, df_q = calibrate_limit_by_quantile(pca_q_sequence, candidate_quantiles,
    calibration_sequences, rng)

```

```
    systems["pca_T2"] = {"model": models["pca"], "limit": best_t2[1], "calibrated_arl0": best_t2[2], "grid": df_t2}
    systems["pca_Q"] = {"model": models["pca"], "limit": best_q[1], "calibrated_arl0": best_q[2], "grid": df_q}

    if "tensor" in models:
        def tensor_t2_sequence(inner_rng):
            vals = []
            for _ in range(max_run_length):
                x = generate_normal_sample(inner_rng)
                t2, q = tensor_stats(x, models["tensor"])
                vals.append(t2)
            return np.array(vals)

        def tensor_q_sequence(inner_rng):
            vals = []
            for _ in range(max_run_length):
                x = generate_normal_sample(inner_rng)
                t2, q = tensor_stats(x, models["tensor"])
                vals.append(q)
            return np.array(vals)

        best_t2, df_t2 = calibrate_limit_by_quantile(tensor_t2_sequence, candidate_quantiles, calibration_sequences, rng)
        best_q, df_q = calibrate_limit_by_quantile(tensor_q_sequence, candidate_quantiles, calibration_sequences, rng)

        systems["tensor_T2"] = {"model": models["tensor"], "limit": best_t2[1], "calibrated_arl0": best_t2[2], "grid": df_t2}
        systems["tensor_Q"] = {"model": models["tensor"], "limit": best_q[1], "calibrated_arl0": best_q[2], "grid": df_q}

    return systems
```

10.8 Example calibration curves

```
systems_example = calibrate_one_system(seed=777)

for key, info in systems_example.items():
    df = info["grid"]
    plt.figure(figsize=(8, 4))
    plt.plot(df["quantile"], df["arl0"], marker='o')
    plt.axhline(target_arl0, linestyle='--')
    plt.title(f"Calibration curve - {key}")
    plt.xlabel("Candidate quantile")
    plt.ylabel("Estimated ARL0")
    plt.show()

```

```
    print(key, "chosen limit =", info["limit"], "calibrated ARL0 =", info["
    calibrated_arl0"])
```

10.9 Detection-delay function with calibrated limits

```
def detection_delay(sequence_stats, limit, change_point):
    post = np.where(sequence_stats[change_point:] > limit)[0]
    if len(post) == 0:
        return max_run_length - change_point
    return int(post[0] + 1)

def simulate_detection_delay_for_fault(seed, fault_type):
    rng = np.random.default_rng(seed)
    systems = calibrate_one_system(seed + 500)

    def apply_fault(x):
        if fault_type == "global_shift":
            return inject_global_shift(x)
        elif fault_type == "patch_anomaly":
            return inject_patch_anomaly(x)
        elif fault_type == "shape_change":
            return inject_shape_change(x)
        else:
            return x

    results = {}

    for key, info in systems.items():
        vals = []
        for t in range(max_run_length):
            x = generate_normal_sample(rng)
            if t >= change_point:
                x = apply_fault(x)

            if key.startswith("pca"):
                t2, q = pca_stats(x, info["model"])
            else:
                t2, q = tensor_stats(x, info["model"])

            vals.append(t2 if key.endswith("T2") else q)

        vals = np.array(vals)
        delay = detection_delay(vals, info["limit"], change_point)
        results[key] = delay

    return systems, results
```

10.10 Monte Carlo study with calibrated ARL0

```
records_arl = []
records_delay = []

fault_types = ["global_shift", "patch_anomaly", "shape_change"]

for run in range(N_RUNS):
    systems = calibrate_one_system(seed=10000 + run)

    for key, info in systems.items():
        records_arl.append({
            "run": run,
            "method_stat": key,
            "calibrated_arl0": info["calibrated_arl0"]
        })

    for fault in fault_types:
        _, delay_results = simulate_detection_delay_for_fault(seed=20000 + 100*run,
    fault_type=fault)
        for key, delay in delay_results.items():
            records_delay.append({
                "run": run,
                "fault_type": fault,
                "method_stat": key,
                "delay": delay
            })

arl_df = pd.DataFrame(records_arl)
delay_df = pd.DataFrame(records_delay)

arl_summary = arl_df.groupby("method_stat")["calibrated_arl0"].agg(["mean", "std"]).
    reset_index()
delay_summary = delay_df.groupby(["fault_type", "method_stat"])["delay"].agg(["mean",
    "std"]).reset_index()

print("ARL0 after calibration:")
display(arl_summary)

print("\nDetection delay after ARL0 calibration:")
display(delay_summary)
```

10.11 Plot calibrated ARL0

```
plt.figure(figsize=(8, 4))
plt.bar(arl_summary["method_stat"], arl_summary["mean"], yerr=arl_summary["std"],
    capsize=5)
```

```
plt.axhline(target_arl0, linestyle='--')
plt.ylabel("Estimated ARL0 after calibration")
plt.title("Control-limit calibration to common target ARL0")
plt.show()
```

10.12 Plot detection delays on equal-ARL0 footing

```
for fault in fault_types:
    sub = delay_summary[delay_summary["fault_type"] == fault]

    plt.figure(figsize=(8, 4))
    plt.bar(sub["method_stat"], sub["mean"], yerr=sub["std"], capsize=5)
    plt.ylabel("Average detection delay")
    plt.title(f"Detection delay after ARL0 calibration - {fault}")
    plt.show()
```

10.13 Pivot tables

```
arl_pivot = arl_summary.set_index("method_stat")
delay_pivot = delay_summary.pivot_table(index="fault_type", columns="method_stat",
    values="mean")

print("ARL0 pivot:")
display(arl_pivot)

print("\nDelay pivot:")
display(delay_pivot)
```

10.14 Interpretation

This section provides a **fairer comparison** than using raw percentile limits.

How to read it:

- If two methods have similar ARL0 after calibration, then differences in detection delay are more meaningful.
- A method that still detects faster after ARL0 matching has a stronger claim to better monitoring performance.
- Different methods may still win on different fault types.

This is often the better way to report simulation results in SPC studies.

10.15 Extensions

Good next extensions:

- calibrate to multiple target ARL0 values, not just one
- estimate detection probability within 5, 10, or 20 post-change samples
- add EWMA/CUSUM versions and calibrate them the same way
- export tables for use in a report or paper

10.16 Optional export

```
# arl_df.to_csv("calibrated_arl0_results.csv", index=False)
# delay_df.to_csv("calibrated_delay_results.csv", index=False)

print("Export section ready.")
```

Part IV

Sequential and Adaptive Extensions

11 EWMA and CUSUM Versions of Tensor T^2 and Tensor Q

This section extends the tensor SPC workflow by creating **sequential smoothing and accumulation statistics** based on:

- **Tensor T^2**: unusual behavior inside the modeled score space
- **Tensor Q**: unusual residual energy outside the modeled subspace

We then build:

- **EWMA-T^2** and **EWMA-Q**
- **CUSUM-T^2** and **CUSUM-Q**

These are especially useful for detecting **small** or **persistent** changes that may be harder to detect with raw one-step statistics.

11.1 Imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)
```

11.2 Simulation settings

```
n_ref = 120
max_run_length = 300
change_point = 100

n_vars = 6
n_time = 20
tensor_rank = [5, 3, 3]

base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
time_axis = np.linspace(0, 2*np.pi, n_time)

# EWMA parameters
lambda_ewma = 0.2

# CUSUM parameters
k_cusum = 0.5  # reference value in standardized units

# For quick in-analysis calibration
calibration_sequences = 60
target_arl0 = 200

print("EWMA lambda:", lambda_ewma)
print("CUSUM k:", k_cusum)
print("Target ARL0 for calibration:", target_arl0)
```

11.3 Data generation

```
def generate_normal_sample(rng):
    amp = 1.0 + 0.05 * rng.standard_normal()
    phase = 0.10 * rng.standard_normal()
    time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
    phase)
    drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
    interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
    noise = 0.08 * rng.standard_normal((n_vars, n_time))
    return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
    interaction + noise

def inject_small_global_shift(sample):
    return 1.10 * sample

def inject_small_patch_shift(sample):
    x = sample.copy()
    x[3, 7:12] += 0.4
    return x
```

```

def inject_persistent_shape_change(sample):
    x = sample.copy()
    x += 0.18 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
    return x
```

11.4 Fit a tensor monitoring model

```
def hotelling_t2(score_vec, mu, S_inv):
    d = score_vec - mu
    return float(d.T @ S_inv @ d)

def fit_tensor_model(X_ref):
    if not TENSORLY_OK:
        raise RuntimeError("TensorLy is required for this analysis.")

    X_ref_tl = tl.tensor(X_ref, dtype=float)
    core_ref, factors_ref = tucker(X_ref_tl, rank=tensor_rank)
    U_sample, U_var, U_time = [np.array(f) for f in factors_ref]

    score_vectors = np.array([
        (U_var.T @ X_ref[i] @ U_time).reshape(-1)
        for i in range(X_ref.shape[0])
    ])

    mu = score_vectors.mean(axis=0)
    S = np.cov(score_vectors, rowvar=False)
    S_inv = np.linalg.inv(S + 1e-6 * np.eye(S.shape[0]))

    t2_ref = np.array([hotelling_t2(s, mu, S_inv) for s in score_vectors])

    q_ref = []
    for i in range(X_ref.shape[0]):
        G = U_var.T @ X_ref[i] @ U_time
        recon = U_var @ G @ U_time.T
        q_ref.append(np.sum((X_ref[i] - recon)**2))
    q_ref = np.array(q_ref)

    return {
        "U_var": U_var,
        "U_time": U_time,
        "mu": mu,
        "S_inv": S_inv,
        "t2_ref": t2_ref,
        "q_ref": q_ref
    }
```

11.5 Compute one-step tensor T^2 and Q

```python
def tensor_t2_q(sample, model):
    U_var = model["U_var"]
    U_time = model["U_time"]

    G = U_var.T @ sample @ U_time
    score = G.reshape(-1)
    t2 = hotelling_t2(score, model["mu"], model["S_inv"])

    recon = U_var @ G @ U_time.T
    q = float(np.sum((sample - recon)**2))

    return t2, q
```

11.6 EWMA and CUSUM transforms

```python
def ewma_stat(x, lam):
    z = np.zeros_like(x, dtype=float)
    z[0] = x[0]
    for t in range(1, len(x)):
        z[t] = lam * x[t] + (1 - lam) * z[t - 1]
    return z

def upper_cusum_stat(x, k):
    c = np.zeros_like(x, dtype=float)
    for t in range(1, len(x)):
        c[t] = max(0, c[t - 1] + x[t] - k)
    return c

def standardize_with_reference(x, ref_mean, ref_std):
    ref_std = max(ref_std, 1e-8)
    return (x - ref_mean) / ref_std
```

11.7 Build one-step and sequential statistics for a run

```python
def tensor_run_statistics(sequence, model):
    t2_vals = []
    q_vals = []
    for x in sequence:
        t2, q = tensor_t2_q(x, model)
        t2_vals.append(t2)
        q_vals.append(q)

    t2_vals = np.array(t2_vals)
```

```
    q_vals = np.array(q_vals)

    # Standardize using reference one-step distributions
    t2_std = standardize_with_reference(t2_vals, model["t2_ref"].mean(), model["t2_ref"].std(ddof=1))
    q_std = standardize_with_reference(q_vals, model["q_ref"].mean(), model["q_ref"].std(ddof=1))

    ewma_t2 = ewma_stat(t2_std, lambda_ewma)
    ewma_q = ewma_stat(q_std, lambda_ewma)

    cusum_t2 = upper_cusum_stat(t2_std, k_cusum)
    cusum_q = upper_cusum_stat(q_std, k_cusum)

    return {
        "T2": t2_vals,
        "Q": q_vals,
        "T2_std": t2_std,
        "Q_std": q_std,
        "EWMA_T2": ewma_t2,
        "EWMA_Q": ewma_q,
        "CUSUM_T2": cusum_t2,
        "CUSUM_Q": cusum_q
    }
```

11.8 Generate in-control sequences for limit calibration

```
def generate_in_control_sequence(rng, length):
    return np.array([generate_normal_sample(rng) for _ in range(length)])

def first_alarm_time(sequence_stats, limit):
    alarms = np.where(sequence_stats > limit)[0]
    if len(alarms) == 0:
        return max_run_length
    return int(alarms[0] + 1)

def calibrate_limit_from_sequences(stats_list, target_arl0, candidate_quantiles):
    pooled = np.concatenate(stats_list)
    best = None
    rows = []

    for q in candidate_quantiles:
        limit = np.quantile(pooled, q)
        run_lengths = np.array([first_alarm_time(stats, limit) for stats in stats_list])
        arl0 = float(run_lengths.mean())
        rows.append((q, limit, arl0))
        if best is None or abs(arl0 - target_arl0) < abs(best[2] - target_arl0):
```

```
            best = (q, limit, arl0)

    return best, pd.DataFrame(rows, columns=["quantile", "limit", "arl0"])
```

11.9 Fit model and calibrate limits for one-step, EWMA, and CUSUM statistics

```
rng = np.random.default_rng(12345)

X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
tensor_model = fit_tensor_model(X_ref)

stat_names = ["T2", "Q", "EWMA_T2", "EWMA_Q", "CUSUM_T2", "CUSUM_Q"]
candidate_quantiles = np.linspace(0.95, 0.9995, 40)

stats_collection = {name: [] for name in stat_names}
for _ in range(calibration_sequences):
    seq = generate_in_control_sequence(rng, max_run_length)
    out = tensor_run_statistics(seq, tensor_model)
    for name in stat_names:
        stats_collection[name].append(out[name])

limits = {}
calibration_tables = {}

for name in stat_names:
    best, df = calibrate_limit_from_sequences(stats_collection[name], target_arl0,
    candidate_quantiles)
    limits[name] = best[1]
    calibration_tables[name] = df

limit_summary = pd.DataFrame({
    "statistic": list(limits.keys()),
    "limit": list(limits.values()),
    "approx_arl0": [calibrate_limit_from_sequences(stats_collection[name], target_arl0,
     candidate_quantiles)[0][2]
                    for name in stat_names]
})

limit_summary
```

11.10 Plot calibration curves

```
for name in stat_names:
    df = calibration_tables[name]
```

```
    plt.figure(figsize=(8, 4))
    plt.plot(df["quantile"], df["arl0"], marker='o')
    plt.axhline(target_arl0, linestyle='--')
    plt.title(f"Calibration curve - {name}")
    plt.xlabel("Candidate quantile")
    plt.ylabel("Estimated ARL0")
    plt.show()
```

11.11 Example sequential run with a small shift

```
def generate_out_of_control_sequence(rng, length, change_point, fault_type):
    seq = []
    for t in range(length):
        x = generate_normal_sample(rng)
        if t >= change_point:
            if fault_type == "small_global_shift":
                x = inject_small_global_shift(x)
            elif fault_type == "small_patch_shift":
                x = inject_small_patch_shift(x)
            elif fault_type == "persistent_shape_change":
                x = inject_persistent_shape_change(x)
        seq.append(x)
    return np.array(seq)

rng = np.random.default_rng(54321)
seq = generate_out_of_control_sequence(rng, max_run_length, change_point, "
    small_patch_shift")
run_out = tensor_run_statistics(seq, tensor_model)
```

11.12 Plot one-step vs EWMA vs CUSUM for T^2

```
plt.figure(figsize=(10, 4))
plt.plot(run_out["T2_std"], label="Standardized T²")
plt.axhline(standardize_with_reference(limits["T2"], tensor_model["t2_ref"].mean(),
    tensor_model["t2_ref"].std(ddof=1)), linestyle='--')
plt.axvline(change_point, linestyle=':')
plt.title("One-step Tensor T²")
plt.xlabel("Time index")
plt.ylabel("Statistic")
plt.legend()
plt.show()

plt.figure(figsize=(10, 4))
plt.plot(run_out["EWMA_T2"], label="EWMA-T²")
plt.axhline(limits["EWMA_T2"], linestyle='--')
```

```
plt.axvline(change_point, linestyle=':')
plt.title("EWMA-T²")
plt.xlabel("Time index")
plt.ylabel("Statistic")
plt.legend()
plt.show()

plt.figure(figsize=(10, 4))
plt.plot(run_out["CUSUM_T2"], label="CUSUM-T²")
plt.axhline(limits["CUSUM_T2"], linestyle='--')
plt.axvline(change_point, linestyle=':')
plt.title("CUSUM-T²")
plt.xlabel("Time index")
plt.ylabel("Statistic")
plt.legend()
plt.show()
```

11.13 Plot one-step vs EWMA vs CUSUM for Q

```
plt.figure(figsize=(10, 4))
plt.plot(run_out["Q_std"], label="Standardized Q")
plt.axhline(standardize_with_reference(limits["Q"], tensor_model["q_ref"].mean(),
    tensor_model["q_ref"].std(ddof=1)), linestyle='--')
plt.axvline(change_point, linestyle=':')
plt.title("One-step Tensor Q")
plt.xlabel("Time index")
plt.ylabel("Statistic")
plt.legend()
plt.show()

plt.figure(figsize=(10, 4))
plt.plot(run_out["EWMA_Q"], label="EWMA-Q")
plt.axhline(limits["EWMA_Q"], linestyle='--')
plt.axvline(change_point, linestyle=':')
plt.title("EWMA-Q")
plt.xlabel("Time index")
plt.ylabel("Statistic")
plt.legend()
plt.show()

plt.figure(figsize=(10, 4))
plt.plot(run_out["CUSUM_Q"], label="CUSUM-Q")
plt.axhline(limits["CUSUM_Q"], linestyle='--')
plt.axvline(change_point, linestyle=':')
plt.title("CUSUM-Q")
plt.xlabel("Time index")
plt.ylabel("Statistic")
plt.legend()
```

```
plt.show()
```

11.14 Compare detection delay across one-step, EWMA, and CUSUM

```
def detection_delay(sequence_stats, limit, change_point):
    post = np.where(sequence_stats[change_point:] > limit)[0]
    if len(post) == 0:
        return max_run_length - change_point
    return int(post[0] + 1)

fault_types = ["small_global_shift", "small_patch_shift", "persistent_shape_change"]
N_RUNS = 30

records = []
for fault in fault_types:
    rng = np.random.default_rng(202600 + hash(fault) % 1000)
    for run in range(N_RUNS):
        seq = generate_out_of_control_sequence(rng, max_run_length, change_point,
    fault)
        out = tensor_run_statistics(seq, tensor_model)

        for name in stat_names:
            delay = detection_delay(out[name], limits[name], change_point)
            records.append({
                "fault_type": fault,
                "run": run,
                "statistic": name,
                "delay": delay
            })

delay_df = pd.DataFrame(records)
delay_summary = delay_df.groupby(["fault_type", "statistic"])["delay"].agg(["mean", "
    std"]).reset_index()
delay_summary
```

11.15 Plot detection delay by statistic

```
for fault in fault_types:
    sub = delay_summary[delay_summary["fault_type"] == fault]
    plt.figure(figsize=(10, 4))
    plt.bar(sub["statistic"], sub["mean"], yerr=sub["std"], capsize=5)
    plt.ylabel("Average detection delay")
    plt.title(f"Detection delay comparison - {fault}")
    plt.show()
```

11.16 Interpretation

Typical interpretation:

- **One-step T^2/Q** react to immediate large departures.
- **EWMA-T^2 / EWMA-Q** smooth noise and can become more sensitive to small sustained shifts.
- **CUSUM-T^2 / CUSUM-Q** accumulate small positive evidence over time and can be strong for persistent weak changes.

The key question is not which is always best, but which is best for the kind of shift you care about.

11.17 Notes for research use

To extend this toward a paper-quality study, logical next steps are:

- calibrate all six statistics to a common target ARL0 more precisely
- compare multiple values of `lambda_ewma`
- compare multiple values of `k_cusum`
- add tensor-vs-PCA EWMA/CUSUM baselines
- estimate full ARL and delay distributions, not just means

11.18 Optional export

```
# limit_summary.to_csv("tensor_ewma_cusum_limit_summary.csv", index=False)
# delay_df.to_csv("tensor_ewma_cusum_detection_delay_runs.csv", index=False)

print("Export section ready.")
```

12 Matched Tensor vs PCA EWMA/CUSUM Comparison

This section creates a **matched sequential comparison** between tensor-based and vectorized PCA-based monitoring.

For both approaches, we compare:

- one-step $\mathbf{T^2}$
- one-step $\mathbf{Q}$
- **EWMA-$\mathbf{T^2}$**
- **EWMA-Q**
- **CUSUM-$\mathbf{T^2}$**
- **CUSUM-Q**

The purpose is to compare tensor and PCA methods on equal sequential footing, especially for **small** and **persistent** shifts.

12.1 Imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)

from sklearn.decomposition import PCA
```

12.2 Settings

```
n_ref = 120
max_run_length = 300
change_point = 100

n_vars = 6
n_time = 20

tensor_rank = [5, 3, 3]
pca_components = 8

lambda_ewma = 0.2
k_cusum = 0.5

target_arl0 = 200
calibration_sequences = 60
N_RUNS = 30

base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
time_axis = np.linspace(0, 2*np.pi, n_time)

print("Tensor rank:", tensor_rank)
print("PCA components:", pca_components)
print("EWMA lambda:", lambda_ewma)
print("CUSUM k:", k_cusum)
```

12.3 Data generation

```
def generate_normal_sample(rng):
    amp = 1.0 + 0.05 * rng.standard_normal()
    phase = 0.10 * rng.standard_normal()
    time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
    phase)
    drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
    interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
    noise = 0.08 * rng.standard_normal((n_vars, n_time))
    return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
    interaction + noise

def inject_small_global_shift(sample):
    return 1.10 * sample

def inject_small_patch_shift(sample):
    x = sample.copy()
    x[3, 7:12] += 0.4
    return x
```

```

def inject_persistent_shape_change(sample):
    x = sample.copy()
    x += 0.18 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
    return x
```

12.4 Shared helpers

```
def hotelling_t2(score_vec, mu, S_inv):
    d = score_vec - mu
    return float(d.T @ S_inv @ d)

def ewma_stat(x, lam):
    z = np.zeros_like(x, dtype=float)
    z[0] = x[0]
    for t in range(1, len(x)):
        z[t] = lam * x[t] + (1 - lam) * z[t - 1]
    return z

def upper_cusum_stat(x, k):
    c = np.zeros_like(x, dtype=float)
    for t in range(1, len(x)):
        c[t] = max(0, c[t - 1] + x[t] - k)
    return c

def standardize_with_reference(x, ref_mean, ref_std):
    ref_std = max(ref_std, 1e-8)
    return (x - ref_mean) / ref_std

def first_alarm_time(sequence_stats, limit):
    alarms = np.where(sequence_stats > limit)[0]
    if len(alarms) == 0:
        return max_run_length
    return int(alarms[0] + 1)

def detection_delay(sequence_stats, limit, change_point):
    post = np.where(sequence_stats[change_point:] > limit)[0]
    if len(post) == 0:
        return max_run_length - change_point
    return int(post[0] + 1)

def calibrate_limit_from_sequences(stats_list, target_arl0, candidate_quantiles):
    pooled = np.concatenate(stats_list)
    best = None
    rows = []

    for q in candidate_quantiles:
        limit = np.quantile(pooled, q)
```

```
        run_lengths = np.array([first_alarm_time(stats, limit) for stats in stats_list
    ])
        arl0 = float(run_lengths.mean())
        rows.append((q, limit, arl0))
        if best is None or abs(arl0 - target_arl0) < abs(best[2] - target_arl0):
            best = (q, limit, arl0)

    return best, pd.DataFrame(rows, columns=["quantile", "limit", "arl0"])
```

12.5 Fit tensor and PCA models

```
def fit_tensor_model(X_ref):
    if not TENSORLY_OK:
        raise RuntimeError("TensorLy is required for tensor sections.")

    X_ref_tl = tl.tensor(X_ref, dtype=float)
    core_ref, factors_ref = tucker(X_ref_tl, rank=tensor_rank)
    U_sample, U_var, U_time = [np.array(f) for f in factors_ref]

    score_vectors = np.array([
        (U_var.T @ X_ref[i] @ U_time).reshape(-1)
        for i in range(X_ref.shape[0])
    ])

    mu = score_vectors.mean(axis=0)
    S = np.cov(score_vectors, rowvar=False)
    S_inv = np.linalg.inv(S + 1e-6 * np.eye(S.shape[0]))

    t2_ref = np.array([hotelling_t2(s, mu, S_inv) for s in score_vectors])

    q_ref = []
    for i in range(X_ref.shape[0]):
        G = U_var.T @ X_ref[i] @ U_time
        recon = U_var @ G @ U_time.T
        q_ref.append(np.sum((X_ref[i] - recon)**2))
    q_ref = np.array(q_ref)

    return {
        "U_var": U_var,
        "U_time": U_time,
        "mu": mu,
        "S_inv": S_inv,
        "t2_ref": t2_ref,
        "q_ref": q_ref
    }

def fit_pca_model(X_ref):
    X_ref_vec = X_ref.reshape(X_ref.shape[0], -1)
```

```
    pca = PCA(n_components=pca_components)
    scores = pca.fit_transform(X_ref_vec)

    mu = scores.mean(axis=0)
    S = np.cov(scores, rowvar=False)
    S_inv = np.linalg.inv(S + 1e-6 * np.eye(S.shape[0]))

    t2_ref = np.array([hotelling_t2(s, mu, S_inv) for s in scores])
    X_hat = pca.inverse_transform(scores)
    q_ref = np.sum((X_ref_vec - X_hat)**2, axis=1)

    return {
        "pca": pca,
        "mu": mu,
        "S_inv": S_inv,
        "t2_ref": t2_ref,
        "q_ref": q_ref
    }
```

12.6 One-step T^2 and Q for tensor and PCA

```
def tensor_t2_q(sample, model):
    U_var = model["U_var"]
    U_time = model["U_time"]
    G = U_var.T @ sample @ U_time
    score = G.reshape(-1)
    t2 = hotelling_t2(score, model["mu"], model["S_inv"])
    recon = U_var @ G @ U_time.T
    q = float(np.sum((sample - recon)**2))
    return t2, q

def pca_t2_q(sample, model):
    x_vec = sample.reshape(1, -1)
    score = model["pca"].transform(x_vec)[0]
    t2 = hotelling_t2(score, model["mu"], model["S_inv"])
    x_hat = model["pca"].inverse_transform(score.reshape(1, -1))[0]
    q = float(np.sum((x_vec[0] - x_hat)**2))
    return t2, q
```

12.7 Build one-step, EWMA, and CUSUM sequences

```
def run_stat_sequences(sequence, model, method):
    t2_vals = []
    q_vals = []

```

```
    for x in sequence:
        if method == "tensor":
            t2, q = tensor_t2_q(x, model)
        elif method == "pca":
            t2, q = pca_t2_q(x, model)
        else:
            raise ValueError("Unknown method")

        t2_vals.append(t2)
        q_vals.append(q)

    t2_vals = np.array(t2_vals)
    q_vals = np.array(q_vals)

    t2_std = standardize_with_reference(t2_vals, model["t2_ref"].mean(), model["t2_ref"].std(ddof=1))
    q_std = standardize_with_reference(q_vals, model["q_ref"].mean(), model["q_ref"].std(ddof=1))

    return {
        "T2": t2_vals,
        "Q": q_vals,
        "T2_std": t2_std,
        "Q_std": q_std,
        "EWMA_T2": ewma_stat(t2_std, lambda_ewma),
        "EWMA_Q": ewma_stat(q_std, lambda_ewma),
        "CUSUM_T2": upper_cusum_stat(t2_std, k_cusum),
        "CUSUM_Q": upper_cusum_stat(q_std, k_cusum),
    }
```

12.8 Fit both models and calibrate all sequential limits

```
rng = np.random.default_rng(202612)
X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])

tensor_model = fit_tensor_model(X_ref) if TENSORLY_OK else None
pca_model = fit_pca_model(X_ref)

stat_names = ["T2", "Q", "EWMA_T2", "EWMA_Q", "CUSUM_T2", "CUSUM_Q"]
candidate_quantiles = np.linspace(0.95, 0.9995, 40)

def generate_in_control_sequence(rng, length):
    return np.array([generate_normal_sample(rng) for _ in range(length)])

limits = {}
calibration_tables = {}

for method_name, model in [("tensor", tensor_model), ("pca", pca_model)]:
```

```
    if model is None:
        continue

    stats_collection = {name: [] for name in stat_names}
    for _ in range(calibration_sequences):
        seq = generate_in_control_sequence(rng, max_run_length)
        out = run_stat_sequences(seq, model, method_name)
        for name in stat_names:
            stats_collection[name].append(out[name])

    for name in stat_names:
        best, df = calibrate_limit_from_sequences(stats_collection[name], target_arl0,
 candidate_quantiles)
        limits[f"{method_name}_{name}"] = {
            "limit": best[1],
            "approx_arl0": best[2]
        }
        calibration_tables[f"{method_name}_{name}"] = df

limit_summary = pd.DataFrame([
    {"method_stat": k, "limit": v["limit"], "approx_arl0": v["approx_arl0"]}
    for k, v in limits.items()
])

limit_summary
```

12.9 Plot selected calibration curves

```
for key in list(calibration_tables.keys())[:6]:
    df = calibration_tables[key]
    plt.figure(figsize=(8, 4))
    plt.plot(df["quantile"], df["arl0"], marker='o')
    plt.axhline(target_arl0, linestyle='--')
    plt.title(f"Calibration curve - {key}")
    plt.xlabel("Candidate quantile")
    plt.ylabel("Estimated ARL0")
    plt.show()
```

12.10 Example sequence comparison

```
def generate_out_of_control_sequence(rng, length, change_point, fault_type):
    seq = []
    for t in range(length):
        x = generate_normal_sample(rng)
        if t >= change_point:
```

```
            if fault_type == "small_global_shift":
                x = inject_small_global_shift(x)
            elif fault_type == "small_patch_shift":
                x = inject_small_patch_shift(x)
            elif fault_type == "persistent_shape_change":
                x = inject_persistent_shape_change(x)
        seq.append(x)
    return np.array(seq)

rng = np.random.default_rng(9090)
seq = generate_out_of_control_sequence(rng, max_run_length, change_point, "
    small_global_shift")

tensor_run = run_stat_sequences(seq, tensor_model, "tensor") if tensor_model is not
    None else None
pca_run = run_stat_sequences(seq, pca_model, "pca")
```

12.11 Plot EWMA-T^2 and EWMA-Q: tensor vs PCA

```
if tensor_run is not None:
    plt.figure(figsize=(10, 4))
    plt.plot(tensor_run["EWMA_T2"], label="Tensor EWMA-T²")
    plt.plot(pca_run["EWMA_T2"], label="PCA EWMA-T²")
    plt.axhline(limits["tensor_EWMA_T2"]["limit"], linestyle='--')
    plt.axhline(limits["pca_EWMA_T2"]["limit"], linestyle=':')
    plt.axvline(change_point, linestyle='-.')
    plt.title("EWMA-T² comparison")
    plt.xlabel("Time index")
    plt.ylabel("Statistic")
    plt.legend()
    plt.show()

    plt.figure(figsize=(10, 4))
    plt.plot(tensor_run["EWMA_Q"], label="Tensor EWMA-Q")
    plt.plot(pca_run["EWMA_Q"], label="PCA EWMA-Q")
    plt.axhline(limits["tensor_EWMA_Q"]["limit"], linestyle='--')
    plt.axhline(limits["pca_EWMA_Q"]["limit"], linestyle=':')
    plt.axvline(change_point, linestyle='-.')
    plt.title("EWMA-Q comparison")
    plt.xlabel("Time index")
    plt.ylabel("Statistic")
    plt.legend()
    plt.show()
```

12.12 Plot CUSUM-T^2 and CUSUM-Q: tensor vs PCA

```
if tensor_run is not None:
    plt.figure(figsize=(10, 4))
    plt.plot(tensor_run["CUSUM_T2"], label="Tensor CUSUM-T²")
    plt.plot(pca_run["CUSUM_T2"], label="PCA CUSUM-T²")
    plt.axhline(limits["tensor_CUSUM_T2"]["limit"], linestyle='--')
    plt.axhline(limits["pca_CUSUM_T2"]["limit"], linestyle=':')
    plt.axvline(change_point, linestyle='-.')
    plt.title("CUSUM-T² comparison")
    plt.xlabel("Time index")
    plt.ylabel("Statistic")
    plt.legend()
    plt.show()

    plt.figure(figsize=(10, 4))
    plt.plot(tensor_run["CUSUM_Q"], label="Tensor CUSUM-Q")
    plt.plot(pca_run["CUSUM_Q"], label="PCA CUSUM-Q")
    plt.axhline(limits["tensor_CUSUM_Q"]["limit"], linestyle='--')
    plt.axhline(limits["pca_CUSUM_Q"]["limit"], linestyle=':')
    plt.axvline(change_point, linestyle='-.')
    plt.title("CUSUM-Q comparison")
    plt.xlabel("Time index")
    plt.ylabel("Statistic")
    plt.legend()
    plt.show()
```

12.13 Detection-delay comparison across all sequential statistics

```
fault_types = ["small_global_shift", "small_patch_shift", "persistent_shape_change"]
records = []

for fault in fault_types:
    for run in range(N_RUNS):
        rng = np.random.default_rng(120000 + run + 1000 * fault_types.index(fault))
        seq = generate_out_of_control_sequence(rng, max_run_length, change_point,
    fault)

        if tensor_model is not None:
            tensor_out = run_stat_sequences(seq, tensor_model, "tensor")
            for name in stat_names:
                delay = detection_delay(tensor_out[name], limits[f"tensor_{name}"]["
    limit"], change_point)
                records.append({
                    "method": "tensor",
                    "fault_type": fault,
                    "statistic": name,
                    "run": run,
                    "delay": delay
```

```
                })

        pca_out = run_stat_sequences(seq, pca_model, "pca")
        for name in stat_names:
            delay = detection_delay(pca_out[name], limits[f"pca_{name}"]["limit"], change_point)
            records.append({
                "method": "pca",
                "fault_type": fault,
                "statistic": name,
                "run": run,
                "delay": delay
            })

delay_df = pd.DataFrame(records)
delay_summary = delay_df.groupby(["method", "fault_type", "statistic"])["delay"].agg(["mean", "std"]).reset_index()
delay_summary.head()
```

12.14 Plot delay by fault type

```
for fault in fault_types:
    sub = delay_summary[delay_summary["fault_type"] == fault].copy()
    sub["label"] = sub["method"] + "_" + sub["statistic"]

    plt.figure(figsize=(12, 4))
    plt.bar(sub["label"], sub["mean"], yerr=sub["std"], capsize=4)
    plt.ylabel("Average detection delay")
    plt.title(f"Sequential detection delay comparison - {fault}")
    plt.xticks(rotation=45)
    plt.show()
```

12.15 Pivot tables

```
arl_pivot = limit_summary.set_index("method_stat")[["approx_arl0"]]
delay_pivot = delay_summary.pivot_table(index="fault_type", columns=["method", "statistic"], values="mean")

print("Approximate ARL0 after calibration:")
display(arl_pivot)

print("\nAverage detection delay:")
display(delay_pivot)
```

12.16 Interpretation

This section provides the **matched sequential baseline** that the earlier tensor-only EWMA/-CUSUM section did not.

What to look for:

- whether tensor EWMA/CUSUM still shows an advantage once PCA EWMA/CUSUM is included
- whether tensor methods help more on structure-specific faults than on broad global shifts
- whether the best statistic depends more on the **fault type** than on the model class itself

This is the kind of comparison that starts to become genuinely persuasive in a research setting.

12.17 Strong next extensions

Logical next extensions include:

- repeat the comparison over multiple `lambda_ewma` values
- repeat over multiple `k_cusum` values
- compare multiple tensor ranks and PCA component counts
- add false-alarm / ARL summaries explicitly to the same analysis
- add tables that rank the best method-statistic combination by fault type

12.18 Optional export

```
# limit_summary.to_csv("matched_tensor_pca_ewma_cusum_limits.csv", index=False)
# delay_df.to_csv("matched_tensor_pca_ewma_cusum_delays.csv", index=False)

print("Export section ready.")
```

Part V

Curated Results

13 Curated Results Summary

This section provides a **curated results summary** of the larger tensor-SPC analysis sequence.

It is designed to be cleaner and more professional than a raw computational dump. The goal is to show:

- a small set of representative figures
- summary tables rather than raw arrays
- direct comparisons between tensor and PCA baselines
- sequential results for one-step, EWMA, and CUSUM monitoring

All line plots use **linewidth = 0.5** for a cleaner printed appearance.

13.1 Imports and plotting defaults

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

try:
    import tensorly as tl
    from tensorly.decomposition import tucker
    TENSORLY_OK = True
except Exception as e:
    TENSORLY_OK = False
    print("TensorLy is not available:", e)

from sklearn.decomposition import PCA

LW = 0.5
MS = 3
RNG = np.random.default_rng(20260421)
```

13.2 Simulation settings

```
n_ref = 120
n_test = 36
max_run_length = 250
change_point = 90

n_vars = 6
n_time = 20

tensor_rank = [5, 3, 3]
pca_components = 8

lambda_ewma = 0.2
k_cusum = 0.5

base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
time_axis = np.linspace(0, 2*np.pi, n_time)
```

13.3 Data-generation helpers

```
def generate_normal_sample(rng):
    amp = 1.0 + 0.05 * rng.standard_normal()
    phase = 0.10 * rng.standard_normal()
    time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
    phase)
    drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
    interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
    noise = 0.08 * rng.standard_normal((n_vars, n_time))
    return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
    interaction + noise

def inject_global_shift(sample, factor=1.25):
    return factor * sample

def inject_patch_anomaly(sample, mag=0.9):
    x = sample.copy()
    x[3, 7:12] += mag
    return x

def inject_shape_change(sample, mag=0.35):
    x = sample.copy()
    x += mag * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
    return x

def inject_small_global_shift(sample):
    return inject_global_shift(sample, factor=1.10)
```

```
25
26 def inject_small_patch_shift(sample):
27     return inject_patch_anomaly(sample, mag=0.4)
28
29 def inject_persistent_shape_change(sample):
30     return inject_shape_change(sample, mag=0.18)
```

13.4 Monitoring helpers

```
1  def hotelling_t2(score_vec, mu, S_inv):
2      d = score_vec - mu
3      return float(d.T @ S_inv @ d)
4
5  def ewma_stat(x, lam):
6      z = np.zeros_like(x, dtype=float)
7      z[0] = x[0]
8      for t in range(1, len(x)):
9          z[t] = lam * x[t] + (1 - lam) * z[t - 1]
10     return z
11
12 def upper_cusum_stat(x, k):
13     c = np.zeros_like(x, dtype=float)
14     for t in range(1, len(x)):
15         c[t] = max(0, c[t - 1] + x[t] - k)
16     return c
17
18 def first_alarm_time(sequence_stats, limit, max_run_length):
19     alarms = np.where(sequence_stats > limit)[0]
20     if len(alarms) == 0:
21         return max_run_length
22     return int(alarms[0] + 1)
23
24 def detection_delay(sequence_stats, limit, change_point, max_run_length):
25     post = np.where(sequence_stats[change_point:] > limit)[0]
26     if len(post) == 0:
27         return max_run_length - change_point
28     return int(post[0] + 1)
29
30 def standardize_with_reference(x, ref_mean, ref_std):
31     ref_std = max(ref_std, 1e-8)
32     return (x - ref_mean) / ref_std
```

13.5 Fit tensor and PCA models

```
1  X_ref = np.array([generate_normal_sample(RNG) for _ in range(n_ref)])
```

```

def fit_tensor_model(X_ref):
    if not TENSORLY_OK:
        return None

    X_ref_tl = tl.tensor(X_ref, dtype=float)
    core_ref, factors_ref = tucker(X_ref_tl, rank=tensor_rank)
    U_sample, U_var, U_time = [np.array(f) for f in factors_ref]

    score_vectors = np.array([
        (U_var.T @ X_ref[i] @ U_time).reshape(-1)
        for i in range(X_ref.shape[0])
    ])

    mu = score_vectors.mean(axis=0)
    S = np.cov(score_vectors, rowvar=False)
    S_inv = np.linalg.inv(S + 1e-6 * np.eye(S.shape[0]))

    t2_ref = np.array([hotelling_t2(s, mu, S_inv) for s in score_vectors])

    q_ref = []
    for i in range(X_ref.shape[0]):
        G = U_var.T @ X_ref[i] @ U_time
        recon = U_var @ G @ U_time.T
        q_ref.append(np.sum((X_ref[i] - recon)**2))
    q_ref = np.array(q_ref)

    return {
        "U_var": U_var,
        "U_time": U_time,
        "mu": mu,
        "S_inv": S_inv,
        "t2_ref": t2_ref,
        "q_ref": q_ref
    }

def fit_pca_model(X_ref):
    X_ref_vec = X_ref.reshape(X_ref.shape[0], -1)
    pca = PCA(n_components=pca_components)
    scores = pca.fit_transform(X_ref_vec)

    mu = scores.mean(axis=0)
    S = np.cov(scores, rowvar=False)
    S_inv = np.linalg.inv(S + 1e-6 * np.eye(S.shape[0]))

    t2_ref = np.array([hotelling_t2(s, mu, S_inv) for s in scores])
    X_hat = pca.inverse_transform(scores)
    q_ref = np.sum((X_ref_vec - X_hat)**2, axis=1)

    return {
```

```
        "pca": pca,
        "mu": mu,
        "S_inv": S_inv,
        "t2_ref": t2_ref,
        "q_ref": q_ref
    }

tensor_model = fit_tensor_model(X_ref)
pca_model = fit_pca_model(X_ref)
print("Tensor model available:", tensor_model is not None)
```

13.6 One-step T^2 and Q functions

```
def tensor_t2_q(sample, model):
    U_var = model["U_var"]
    U_time = model["U_time"]
    G = U_var.T @ sample @ U_time
    score = G.reshape(-1)
    t2 = hotelling_t2(score, model["mu"], model["S_inv"])
    recon = U_var @ G @ U_time.T
    q = float(np.sum((sample - recon)**2))
    return t2, q

def pca_t2_q(sample, model):
    x_vec = sample.reshape(1, -1)
    score = model["pca"].transform(x_vec)[0]
    t2 = hotelling_t2(score, model["mu"], model["S_inv"])
    x_hat = model["pca"].inverse_transform(score.reshape(1, -1))[0]
    q = float(np.sum((x_vec[0] - x_hat)**2))
    return t2, q
```

13.7 Curated result 1 – representative tensor T^2 and Q example

```
X_test = np.array([generate_normal_sample(RNG) for _ in range(n_test)])
labels = np.array(["normal"] * n_test, dtype=object)

for i in range(12, 18):
    X_test[i] = inject_global_shift(X_test[i], factor=1.30)
    labels[i] = "global_shift"

for i in range(18, 27):
    X_test[i] = inject_patch_anomaly(X_test[i], mag=0.9)
    labels[i] = "patch_anomaly"

for i in range(27, 36):
```

```
    X_test[i] = inject_shape_change(X_test[i], mag=0.35)
    labels[i] = "shape_change"

tensor_t2 = []
tensor_q = []

for i in range(n_test):
    t2, q = tensor_t2_q(X_test[i], tensor_model)
    tensor_t2.append(t2)
    tensor_q.append(q)

tensor_t2 = np.array(tensor_t2)
tensor_q = np.array(tensor_q)

t2_limit = np.percentile(tensor_model["t2_ref"], 99)
q_limit = np.percentile(tensor_model["q_ref"], 99)

fig, ax = plt.subplots(figsize=(9, 3.6))
ax.plot(tensor_t2, linewidth=LW, marker='o', markersize=MS)
ax.axhline(t2_limit, linewidth=LW, linestyle='--')
ax.set_title("Representative Tensor T$^2$ Results")
ax.set_xlabel("Test sample index")
ax.set_ylabel("Tensor T$^2$")
plt.show()

fig, ax = plt.subplots(figsize=(9, 3.6))
ax.plot(tensor_q, linewidth=LW, marker='o', markersize=MS)
ax.axhline(q_limit, linewidth=LW, linestyle='--')
ax.set_title("Representative Tensor Q Results")
ax.set_xlabel("Test sample index")
ax.set_ylabel("Tensor Q")
plt.show()
```

13.8 Curated result 2 – normal vs anomaly visualization

```
good_idx = 4
bad_idx = 21

fig, ax = plt.subplots(figsize=(7.2, 3.6))
im = ax.imshow(X_test[good_idx], aspect='auto')
ax.set_title(f"Representative Normal Sample ({labels[good_idx]})")
ax.set_xlabel("Time")
ax.set_ylabel("Variable")
plt.colorbar(im, ax=ax)
plt.show()

fig, ax = plt.subplots(figsize=(7.2, 3.6))
im = ax.imshow(X_test[bad_idx], aspect='auto')
```

```
ax.set_title(f"Representative Anomalous Sample ({labels[bad_idx]})")
ax.set_xlabel("Time")
ax.set_ylabel("Variable")
plt.colorbar(im, ax=ax)
plt.show()
```

13.9 Curated result 3 – tensor vs PCA one-step comparison

```
pca_t2 = []
pca_q = []
for i in range(n_test):
    t2, q = pca_t2_q(X_test[i], pca_model)
    pca_t2.append(t2)
    pca_q.append(q)

pca_t2 = np.array(pca_t2)
pca_q = np.array(pca_q)

pca_t2_limit = np.percentile(pca_model["t2_ref"], 99)
pca_q_limit = np.percentile(pca_model["q_ref"], 99)

fig, ax = plt.subplots(figsize=(6, 4))
for fault in np.unique(labels):
    mask = labels == fault
    ax.scatter(tensor_t2[mask], tensor_q[mask], s=16, label=fault)
ax.axvline(t2_limit, linewidth=LW, linestyle='--')
ax.axhline(q_limit, linewidth=LW, linestyle='--')
ax.set_title("Tensor T$^2$-Q Space")
ax.set_xlabel("Tensor T$^2$")
ax.set_ylabel("Tensor Q")
ax.legend()
plt.show()

fig, ax = plt.subplots(figsize=(6, 4))
for fault in np.unique(labels):
    mask = labels == fault
    ax.scatter(pca_t2[mask], pca_q[mask], s=16, label=fault)
ax.axvline(pca_t2_limit, linewidth=LW, linestyle='--')
ax.axhline(pca_q_limit, linewidth=LW, linestyle='--')
ax.set_title("PCA T$^2$-Q Space")
ax.set_xlabel("PCA T$^2$")
ax.set_ylabel("PCA Q")
ax.legend()
plt.show()
```

13.10 Curated result 4 – Monte Carlo detection summary

```
def summarize_detection(indices, labels):
    flagged = np.zeros(len(labels), dtype=bool)
    flagged[indices] = True
    rows = []
    for label in np.unique(labels):
        mask = labels == label
        rows.append({
            "fault_type": label,
            "flag_rate": float(flagged[mask].mean())
        })
    return pd.DataFrame(rows)

N_MC = 20
mc_rows = []

for run in range(N_MC):
    rng = np.random.default_rng(4000 + run)
    X_ref_mc = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
    tensor_model_mc = fit_tensor_model(X_ref_mc)
    pca_model_mc = fit_pca_model(X_ref_mc)

    X_test_mc = np.array([generate_normal_sample(rng) for _ in range(n_test)])
    labels_mc = np.array(["normal"] * n_test, dtype=object)

    for i in range(12, 18):
        X_test_mc[i] = inject_global_shift(X_test_mc[i], factor=1.30)
        labels_mc[i] = "global_shift"
    for i in range(18, 27):
        X_test_mc[i] = inject_patch_anomaly(X_test_mc[i], mag=0.9)
        labels_mc[i] = "patch_anomaly"
    for i in range(27, 36):
        X_test_mc[i] = inject_shape_change(X_test_mc[i], mag=0.35)
        labels_mc[i] = "shape_change"

    tensor_t2_mc, tensor_q_mc = [], []
    pca_t2_mc, pca_q_mc = [], []

    for i in range(n_test):
        t2, q = tensor_t2_q(X_test_mc[i], tensor_model_mc)
        tensor_t2_mc.append(t2)
        tensor_q_mc.append(q)

        t2p, qp = pca_t2_q(X_test_mc[i], pca_model_mc)
        pca_t2_mc.append(t2p)
        pca_q_mc.append(qp)

    tensor_t2_mc = np.array(tensor_t2_mc)
```

```
    tensor_q_mc = np.array(tensor_q_mc)
    pca_t2_mc = np.array(pca_t2_mc)
    pca_q_mc = np.array(pca_q_mc)

    tensor_t2_limit_mc = np.percentile(tensor_model_mc["t2_ref"], 99)
    tensor_q_limit_mc = np.percentile(tensor_model_mc["q_ref"], 99)
    pca_t2_limit_mc = np.percentile(pca_model_mc["t2_ref"], 99)
    pca_q_limit_mc = np.percentile(pca_model_mc["q_ref"], 99)

    method_flags = {
        "tensor_T2": np.where(tensor_t2_mc > tensor_t2_limit_mc)[0],
        "tensor_Q": np.where(tensor_q_mc > tensor_q_limit_mc)[0],
        "pca_T2": np.where(pca_t2_mc > pca_t2_limit_mc)[0],
        "pca_Q": np.where(pca_q_mc > pca_q_limit_mc)[0],
    }

    for method_name, idxs in method_flags.items():
        tmp = summarize_detection(idxs, labels_mc)
        tmp["method"] = method_name
        tmp["run"] = run
        mc_rows.append(tmp)

mc_df = pd.concat(mc_rows, ignore_index=True)
mc_summary = (
    mc_df.groupby(["method", "fault_type"])["flag_rate"]
    .agg(["mean", "std"])
    .reset_index()
)
mc_summary
```

13.11 Monte Carlo detection plot

```
fault_order = ["normal", "global_shift", "patch_anomaly", "shape_change"]

for fault in fault_order:
    sub = mc_summary[mc_summary["fault_type"] == fault].copy()
    fig, ax = plt.subplots(figsize=(7.5, 3.5))
    ax.bar(sub["method"], sub["mean"], yerr=sub["std"], capsize=4, linewidth=LW)
    ax.set_ylim(0, 1.05)
    ax.set_ylabel("Mean flag rate")
    ax.set_title(f"Monte Carlo Summary - {fault}")
    plt.xticks(rotation=25)
    plt.show()
```

13.12 Curated result 5 – ARL and detection delay summary

```
def fit_sequential_limits(model, method_name, rng, target_quantile=0.99):
    # Simple fixed-quantile calibration for curated summary
    seq_stats_t2 = []
    seq_stats_q = []
    for _ in range(25):
        seq = np.array([generate_normal_sample(rng) for _ in range(max_run_length)])
        t2_list, q_list = [], []
        for x in seq:
            if method_name == "tensor":
                t2, q = tensor_t2_q(x, model)
            else:
                t2, q = pca_t2_q(x, model)
            t2_list.append(t2)
            q_list.append(q)
        seq_stats_t2.append(np.array(t2_list))
        seq_stats_q.append(np.array(q_list))
    lim_t2 = np.quantile(np.concatenate(seq_stats_t2), target_quantile)
    lim_q = np.quantile(np.concatenate(seq_stats_q), target_quantile)
    return lim_t2, lim_q

rng_seq = np.random.default_rng(9900)
tensor_lim_t2, tensor_lim_q = fit_sequential_limits(tensor_model, "tensor", rng_seq)
pca_lim_t2, pca_lim_q = fit_sequential_limits(pca_model, "pca", rng_seq)

arl_rows = []
delay_rows = []

fault_funcs = {
    "small_global_shift": inject_small_global_shift,
    "small_patch_shift": inject_small_patch_shift,
    "persistent_shape_change": inject_persistent_shape_change
}

for run in range(20):
    rng = np.random.default_rng(7000 + run)

    # ARL0
    seq_ic = np.array([generate_normal_sample(rng) for _ in range(max_run_length)])
    t_t2, t_q, p_t2, p_q = [], [], [], []
    for x in seq_ic:
        a, b = tensor_t2_q(x, tensor_model)
        c, d = pca_t2_q(x, pca_model)
        t_t2.append(a); t_q.append(b); p_t2.append(c); p_q.append(d)

    arl_rows.extend([
        {"method_stat": "tensor_T2", "run_length": first_alarm_time(np.array(t_t2),
    tensor_lim_t2, max_run_length)},
        {"method_stat": "tensor_Q", "run_length": first_alarm_time(np.array(t_q),
    tensor_lim_q, max_run_length)},
```

```
        {"method_stat": "pca_T2", "run_length": first_alarm_time(np.array(p_t2), pca_lim_t2, max_run_length)},
        {"method_stat": "pca_Q", "run_length": first_alarm_time(np.array(p_q), pca_lim_q, max_run_length)},
    ])

    # detection delay
    for fault_name, fault_fn in fault_funcs.items():
        seq_oc = []
        for t in range(max_run_length):
            x = generate_normal_sample(rng)
            if t >= change_point:
                x = fault_fn(x)
            seq_oc.append(x)
        seq_oc = np.array(seq_oc)

        t_t2, t_q, p_t2, p_q = [], [], [], []
        for x in seq_oc:
            a, b = tensor_t2_q(x, tensor_model)
            c, d = pca_t2_q(x, pca_model)
            t_t2.append(a); t_q.append(b); p_t2.append(c); p_q.append(d)

        delay_rows.extend([
            {"fault_type": fault_name, "method_stat": "tensor_T2", "delay": detection_delay(np.array(t_t2), tensor_lim_t2, change_point, max_run_length)},
            {"fault_type": fault_name, "method_stat": "tensor_Q", "delay": detection_delay(np.array(t_q), tensor_lim_q, change_point, max_run_length)},
            {"fault_type": fault_name, "method_stat": "pca_T2", "delay": detection_delay(np.array(p_t2), pca_lim_t2, change_point, max_run_length)},
            {"fault_type": fault_name, "method_stat": "pca_Q", "delay": detection_delay(np.array(p_q), pca_lim_q, change_point, max_run_length)},
        ])

arl_df = pd.DataFrame(arl_rows)
delay_df = pd.DataFrame(delay_rows)

arl_summary = arl_df.groupby("method_stat")["run_length"].agg(["mean", "std"]).reset_index()
delay_summary = delay_df.groupby(["fault_type", "method_stat"])["delay"].agg(["mean", "std"]).reset_index()

arl_summary, delay_summary.head()
```

13.13 ARL and delay plots

```
fig, ax = plt.subplots(figsize=(7.2, 3.5))
ax.bar(arl_summary["method_stat"], arl_summary["mean"], yerr=arl_summary["std"], capsize=4, linewidth=LW)
```

```
ax.set_ylabel("Estimated run length")
ax.set_title("Curated ARL Summary")
plt.xticks(rotation=25)
plt.show()

for fault in delay_summary["fault_type"].unique():
    sub = delay_summary[delay_summary["fault_type"] == fault]
    fig, ax = plt.subplots(figsize=(7.5, 3.5))
    ax.bar(sub["method_stat"], sub["mean"], yerr=sub["std"], capsize=4, linewidth=LW)
    ax.set_ylabel("Detection delay")
    ax.set_title(f"Curated Detection Delay Summary - {fault}")
    plt.xticks(rotation=25)
    plt.show()
```

13.14 Curated result 6 – sequential EWMA/CUSUM tensor vs PCA comparison

```
# Build a representative out-of-control sequence
rng = np.random.default_rng(8800)
seq = []
for t in range(max_run_length):
    x = generate_normal_sample(rng)
    if t >= change_point:
        x = inject_small_patch_shift(x)
    seq.append(x)
seq = np.array(seq)

def build_seq_stats(seq, model, method):
    t2_vals, q_vals = [], []
    for x in seq:
        if method == "tensor":
            t2, q = tensor_t2_q(x, model)
        else:
            t2, q = pca_t2_q(x, model)
        t2_vals.append(t2)
        q_vals.append(q)
    t2_vals = np.array(t2_vals)
    q_vals = np.array(q_vals)

    if method == "tensor":
        t2_std = standardize_with_reference(t2_vals, tensor_model["t2_ref"].mean(),
    tensor_model["t2_ref"].std(ddof=1))
        q_std = standardize_with_reference(q_vals, tensor_model["q_ref"].mean(),
    tensor_model["q_ref"].std(ddof=1))
    else:
        t2_std = standardize_with_reference(t2_vals, pca_model["t2_ref"].mean(),
    pca_model["t2_ref"].std(ddof=1))
```

```
        q_std = standardize_with_reference(q_vals, pca_model["q_ref"].mean(),
    pca_model["q_ref"].std(ddof=1))

    return {
        "EWMA_T2": ewma_stat(t2_std, lambda_ewma),
        "EWMA_Q": ewma_stat(q_std, lambda_ewma),
        "CUSUM_T2": upper_cusum_stat(t2_std, k_cusum),
        "CUSUM_Q": upper_cusum_stat(q_std, k_cusum),
    }

tensor_seq = build_seq_stats(seq, tensor_model, "tensor")
pca_seq = build_seq_stats(seq, pca_model, "pca")
```

13.15 EWMA/CUSUM sequential comparison plots

```
fig, ax = plt.subplots(figsize=(9, 3.6))
ax.plot(tensor_seq["EWMA_T2"], linewidth=LW, label="Tensor EWMA-T$^2$")
ax.plot(pca_seq["EWMA_T2"], linewidth=LW, label="PCA EWMA-T$^2$")
ax.axvline(change_point, linewidth=LW, linestyle='--')
ax.set_title("EWMA-T$^2$: Tensor vs PCA")
ax.set_xlabel("Time index")
ax.set_ylabel("Statistic")
ax.legend()
plt.show()

fig, ax = plt.subplots(figsize=(9, 3.6))
ax.plot(tensor_seq["EWMA_Q"], linewidth=LW, label="Tensor EWMA-Q")
ax.plot(pca_seq["EWMA_Q"], linewidth=LW, label="PCA EWMA-Q")
ax.axvline(change_point, linewidth=LW, linestyle='--')
ax.set_title("EWMA-Q: Tensor vs PCA")
ax.set_xlabel("Time index")
ax.set_ylabel("Statistic")
ax.legend()
plt.show()

fig, ax = plt.subplots(figsize=(9, 3.6))
ax.plot(tensor_seq["CUSUM_T2"], linewidth=LW, label="Tensor CUSUM-T$^2$")
ax.plot(pca_seq["CUSUM_T2"], linewidth=LW, label="PCA CUSUM-T$^2$")
ax.axvline(change_point, linewidth=LW, linestyle='--')
ax.set_title("CUSUM-T$^2$: Tensor vs PCA")
ax.set_xlabel("Time index")
ax.set_ylabel("Statistic")
ax.legend()
plt.show()

fig, ax = plt.subplots(figsize=(9, 3.6))
ax.plot(tensor_seq["CUSUM_Q"], linewidth=LW, label="Tensor CUSUM-Q")
ax.plot(pca_seq["CUSUM_Q"], linewidth=LW, label="PCA CUSUM-Q")
```

```
ax.axvline(change_point, linewidth=LW, linestyle='--')
ax.set_title("CUSUM-Q: Tensor vs PCA")
ax.set_xlabel("Time index")
ax.set_ylabel("Statistic")
ax.legend()
plt.show()
```

13.16 Final curated summary tables

```
final_detection_table = mc_summary.pivot_table(
    index="fault_type",
    columns="method",
    values="mean"
).round(3)

final_delay_table = delay_summary.pivot_table(
    index="fault_type",
    columns="method_stat",
    values="mean"
).round(2)

print("Curated detection summary table:")
display(final_detection_table)

print("\nCurated delay summary table:")
display(final_delay_table)

print("\nCurated ARL summary table:")
display(arl_summary.round(2))
```

13.17 Interpretation

This section is intentionally selective.

Instead of showing every intermediate result, it keeps:

- representative one-step figures
- representative anomaly visualizations
- one-step tensor vs PCA comparisons
- Monte Carlo summary plots
- curated ARL and detection-delay tables
- matched EWMA/CUSUM sequential comparisons

This is the kind of curated analytical summary that is appropriate to print or adapt into a monograph chapter.

14 Conclusion

This work developed a tensor-structured statistical process control framework that preserves multiway data structure while extending classical multivariate SPC methods.

The framework integrates:

- tensor-based residual (Q) monitoring
- score-space (T^2) monitoring
- simulation-based benchmarking against vectorized PCA
- ARL and detection-delay evaluation
- sequential extensions using EWMA and CUSUM

The curated simulation results demonstrate that tensor-based monitoring can provide improved sensitivity to structured variation while maintaining interpretability across modes. The results also show that the relative advantage of tensor monitoring depends on the fault structure, the statistic used, and the calibration strategy. This is an important practical conclusion: tensor SPC should not be viewed as a universal replacement for PCA-based MSPC, but as a structure-aware extension that is most valuable when the process data have meaningful multiway organization.

14.1 Future Work

Future work includes:

- application to real industrial datasets
- adaptive and nonstationary process monitoring
- integration with machine learning and anomaly detection methods
- theoretical refinement of distributional assumptions
- development of contribution and localization tools for practical root-cause interpretation

References

- Kolda, T. G., & Bader, B. W. (2009). Tensor decompositions and applications. *SIAM Review*, 51(3), 455–500.
- Montgomery, D. C. (2019). *Introduction to Statistical Quality Control* (8th ed.). Wiley.
- Jackson, J. E. (1991). *A User's Guide to Principal Components*. Wiley.
- Hotelling, H. (1947). Multivariate quality control. In *Techniques of Statistical Analysis*. McGraw-Hill.
- Nomikos, P., & MacGregor, J. F. (1995). Multivariate SPC charts for monitoring batch processes. *Technometrics*, 37(1), 41–59.
- MacGregor, J. F., & Kourti, T. (1995). Statistical process control of multivariate processes. *Control Engineering Practice*, 3(3), 403–414.
- Qin, S. J. (2003). Statistical process monitoring: Basics and beyond. *Journal of Chemometrics*, 17(8–9), 480–502.
- Lowry, C. A., Woodall, W. H., Champ, C. W., & Rigdon, S. E. (1992). A multivariate exponentially weighted moving average control chart. *Technometrics*, 34(1), 46–53.
- Ku, W., Storer, R. H., & Georgakis, C. (1995). Disturbance detection and isolation by dynamic principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 30(1), 179–196.
- Wise, B. M., & Gallagher, N. B. (1996). The process chemometrics approach to process monitoring and fault detection. *Journal of Process Control*, 6(6), 329–348.