
TENSOR-STRUCTURED STATISTICAL PROCESS CONTROL

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



TENSOR-STRUCTURED STATISTICAL PROCESS CONTROL

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

Developed by

Jeff M.

April 2026

Prepared as a technical reference volume.

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.

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

| | | |
|----------|--|-----------|
| I | Foundations and Decomposition | 1 |
| 1 | Tensor Foundations for SPC | 2 |
| | 1. Imports | 2 |
| | 2. Vector inner product and norm | 2 |
| | 3. Matrix (second-order tensor) inner product and Frobenius norm | 3 |
| | 4. Third-order tensor inner product | 3 |
| | 5. Outer product (tensor product of vectors) | 4 |
| | 6. Tensor unfolding / matricization | 4 |
| | 7. TensorLy unfolding and n-mode products | 4 |
| | 8. Tucker decomposition | 5 |
| | 9. Residual tensor and Q statistic | 5 |
| | 10. Why this matters for tensor SPC | 6 |
| 2 | Tucker Decomposition and HOSVD Workflow | 7 |
| | 1. Imports | 7 |
| | 2. Build a structured synthetic tensor | 7 |
| | 3. Visualize one slice | 8 |
| | 4. Tucker decomposition | 8 |
| | 5. Inspect factor matrices | 8 |
| | 6. Reconstruction error | 9 |
| | 7. Compare ranks | 9 |
| | 8. Plot reconstruction error vs rank | 9 |
| | 9. SPC interpretation | 10 |
| 3 | Theoretical Foundations and Development of Tensor SPC | 11 |
| | Introduction | 11 |
| | Assumptions | 11 |
| | Multilinear Representation | 12 |
| | Monitoring Statistics | 12 |
| | Theoretical Properties | 12 |
| | Control Limits | 13 |
| | Relation to Interaction Structure | 13 |
| | Implementation Considerations | 13 |
| | Extensions | 14 |
| | Summary | 14 |

| | | |
|------------|--|-----------|
| II | Tensor Monitoring Fundamentals | 15 |
| 4 | Residuals, Q Statistic, and Tensor SPC Logic | 16 |
| | 1. Imports | 16 |
| | 2. Simulate normal-reference tensor data | 16 |
| | 3. Fit Tucker model to reference data | 17 |
| | 4. Reference Q distribution and empirical limit | 17 |
| | 5. Plot reference Q values | 17 |
| | 6. Create new test samples, including abnormal ones | 18 |
| | 7. Simple projection-based approximation using learned factors | 18 |
| | 8. Plot test Q statistics against the limit | 19 |
| | 9. Compare one normal and one abnormal sample | 19 |
| | 10. Interpretation | 19 |
| 5 | Synthetic Process Data Tensor Monitoring | 21 |
| | 1. Imports | 21 |
| | 2. Simulate process data | 21 |
| | 3. Fit a low-rank tensor model | 22 |
| | 4. Plot Q statistics | 22 |
| | 5. Visualize a typical lot and an abnormal lot | 22 |
| | 6. Residual localization | 23 |
| | 7. Interpretation in this context | 23 |
| 6 | Tensor T^2 Monitoring in Score Space | 25 |
| | 1. Imports | 25 |
| | 2. Simulate in-control tensor samples | 26 |
| | 3. Fit a Tucker model to the reference tensor | 26 |
| | 4. Build a score representation for each sample | 26 |
| | 5. Estimate the score-space center and covariance | 27 |
| | 6. Define the T^2 statistic | 27 |
| | 7. Define the Q statistic for the same samples | 27 |
| | 8. Set empirical control limits | 28 |
| | 9. Create test samples with different types of abnormalities | 28 |
| | 10. Compute test T^2 and Q | 28 |
| | 11. Interpretation | 29 |
| III | Comparative and Simulation Studies | 30 |
| 7 | Tensor SPC vs Vectorized PCA Baseline | 31 |
| | 1. Imports | 31 |
| | 2. Simulate structured in-control tensor data | 31 |
| | 3. Create test data with several fault types | 32 |
| | 4. Tensor monitoring model | 33 |
| | 5. Vectorized PCA baseline | 34 |
| | 6. Apply both monitoring systems to the test set | 35 |
| | 7. Plot tensor statistics | 35 |

| | |
|--|-----------|
| 8. Plot PCA baseline statistics | 36 |
| 9. Flagged samples by method | 36 |
| 10. Detection summary by fault type | 36 |
| 11. Compare methods in scatter plots | 37 |
| 12. Inspect a few example samples | 38 |
| 13. Interpretation | 38 |
| 14. Ideas for extending this notebook | 38 |
| 8 Monte Carlo Comparison of Tensor SPC and PCA | 40 |
| 1. Imports | 40 |
| 2. Simulation design | 40 |
| 3. Data generation functions | 41 |
| 4. Monitoring helper functions | 42 |
| 5. One Monte Carlo run | 42 |
| 6. Run the Monte Carlo study | 44 |
| 7. Aggregate results | 45 |
| 8. Separate false alarm and detection summaries | 45 |
| 9. Plot false alarm rates | 45 |
| 10. Plot detection rates by fault type | 46 |
| 11. Pivot tables for easier comparison | 46 |
| 12. A compact method ranking view | 46 |
| 13. Interpretation guide | 47 |
| 14. Extensions for a stronger research study | 47 |
| 9 ARL and Detection Delay: Tensor SPC vs PCA | 48 |
| 1. Imports | 48 |
| 2. Simulation settings | 49 |
| 3. Data generation | 49 |
| 4. Monitoring model fitting | 50 |
| 5. Online statistics for one sample | 51 |
| 6. Run-length logic | 52 |
| 7. Simulate one in-control run for ARL0 | 52 |
| 8. Simulate one out-of-control run for detection delay | 53 |
| 9. Estimate ARL0 | 54 |
| 10. Plot ARL0 | 54 |
| 11. Estimate detection delay for each fault type | 54 |
| 12. Plot detection delay by fault type | 55 |
| 13. Pivot tables | 55 |
| 14. Example sequential run visualization | 56 |
| 15. Interpretation | 57 |
| 16. Extensions | 58 |
| 10 Calibrating Limits to a Common Target ARL0 | 59 |
| 1. Imports | 59 |
| 2. Simulation settings | 60 |
| 3. Data generation | 60 |

| | |
|--|-----------|
| 4. Model fitting | 61 |
| 5. Statistic functions | 62 |
| 6. Run-length utilities | 62 |
| 7. Build one fitted monitoring system and calibrate limits | 63 |
| 8. Example calibration curves | 64 |
| 9. Detection-delay function with calibrated limits | 65 |
| 10. Monte Carlo study with calibrated ARL0 | 66 |
| 11. Plot calibrated ARL0 | 66 |
| 12. Plot detection delays on equal-ARL0 footing | 67 |
| 13. Pivot tables | 67 |
| 14. Interpretation | 67 |
| 15. Extensions | 68 |
| IV Sequential and Adaptive Extensions | 69 |
| 11 EWMA and CUSUM Versions of Tensor T^2 and Tensor Q | 70 |
| 1. Imports | 70 |
| 2. Simulation settings | 71 |
| 3. Data generation | 71 |
| 4. Fit a tensor monitoring model | 72 |
| 5. Compute one-step tensor T^2 and Q | 73 |
| 6. EWMA and CUSUM transforms | 73 |
| 7. Build one-step and sequential statistics for a run | 73 |
| 8. Generate in-control sequences for limit calibration | 74 |
| 9. Fit model and calibrate limits for one-step, EWMA, and CUSUM statistics | 75 |
| 10. Plot calibration curves | 75 |
| 11. Example sequential run with a small shift | 76 |
| 12. Plot one-step vs EWMA vs CUSUM for T^2 | 76 |
| 13. Plot one-step vs EWMA vs CUSUM for Q | 77 |
| 14. Compare detection delay across one-step, EWMA, and CUSUM | 78 |
| 15. Plot detection delay by statistic | 78 |
| 16. Interpretation | 79 |
| 17. Notes for research use | 79 |
| 12 Matched Tensor vs PCA EWMA/CUSUM Comparison | 80 |
| 1. Imports | 80 |
| 2. Settings | 81 |
| 3. Data generation | 81 |
| 4. Shared helpers | 82 |
| 5. Fit tensor and PCA models | 83 |
| 6. One-step T^2 and Q for tensor and PCA | 84 |
| 7. Build one-step, EWMA, and CUSUM sequences | 84 |
| 8. Fit both models and calibrate all sequential limits | 85 |
| 9. Plot selected calibration curves | 86 |
| 10. Example sequence comparison | 86 |

| | |
|---|-----------|
| 11. Plot EWMA- T^2 and EWMA-Q: tensor vs PCA | 87 |
| 12. Plot CUSUM- T^2 and CUSUM-Q: tensor vs PCA | 87 |
| 13. Detection-delay comparison across all sequential statistics | 88 |
| 14. Plot delay by fault type | 89 |
| 15. Pivot tables | 89 |
| 16. Interpretation | 90 |
| 17. Strong next extensions | 90 |
| | |
| V Curated Results | 93 |
| Curated Results Summary | 94 |

Part I

Foundations and Decomposition

Tensor Foundations for SPC

This notebook 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. Imports

```
1 import numpy as np
2
3 try:
4     import tensorly as tl
5     from tensorly import unfold
6     from tensorly.tenalg import mode_dot
7     from tensorly.decomposition import tucker
8     from tensorly.tucker_tensor import tucker_to_tensor
9     TENSORLY_OK = True
10 except Exception as e:
11     TENSORLY_OK = False
12     print("TensorLy is not available:", e)
```

2. Vector inner product and norm

```
1 u = np.array([1, 2, 3], dtype=float)
2 v = np.array([4, 5, 6], dtype=float)
3
4 inner_uv = np.dot(u, v)
5 norm_u = np.linalg.norm(u)
6
7 print("u =", u)
8 print("v =", v)
9 print("u · v =", inner_uv)
10 print("||u|| =", norm_u)
```

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

```
1 X = np.array([[1, 2],
2               [3, 4]], dtype=float)
3
4 Y = np.array([[5, 6],
5               [7, 8]], dtype=float)
6
7 inner_XY = np.sum(X * Y)
8 fro_X = np.linalg.norm(X)
9
10 print("X =\n", X)
11 print("Y =\n", Y)
12 print("<X, Y> =", inner_XY)
13 print("||X||_F =", fro_X)
```

4. Third-order tensor inner product

```
1 A = np.array([
2     [[1, 2],
3      [3, 4]],
4
5     [[5, 6],
6      [7, 8]]
7 ], dtype=float)
8
9 B = np.array([
10    [[1, 0],
11     [0, 1]],
12
13    [[2, 1],
14     [1, 2]]
15 ], dtype=float)
16
```

```

17 inner_AB = np.sum(A * B)
18 norm_A = np.sqrt(np.sum(A * A))
19
20 print("A shape:", A.shape)
21 print("B shape:", B.shape)
22 print("<A, B> =", inner_AB)
23 print("||A|| =", norm_A)

```

5. Outer product (tensor product of vectors)

For vectors, the tensor product is the outer product.

```

1 u = np.array([1, 2], dtype=float)
2 v = np.array([3, 4], dtype=float)
3
4 outer_uv = np.outer(u, v)
5 print("u ⊗ v =\n", outer_uv)

```

6. Tensor unfolding / matricization

```

1 T = np.arange(24).reshape(3, 4, 2)
2
3 mode0 = T.reshape(T.shape[0], -1)
4 mode1 = np.moveaxis(T, 1, 0).reshape(T.shape[1], -1)
5 mode2 = np.moveaxis(T, 2, 0).reshape(T.shape[2], -1)
6
7 print("Original shape:", T.shape)
8 print("Mode-0 unfolding:", mode0.shape)
9 print(mode0)
10 print("\nMode-1 unfolding:", mode1.shape)
11 print(mode1)
12 print("\nMode-2 unfolding:", mode2.shape)
13 print(mode2)

```

7. TensorLy unfolding and n-mode products

```

1 if TENSORLY_OK:
2     X = tl.tensor(np.arange(24).reshape(3, 4, 2), dtype=float)
3     print("Mode-0 unfolding shape:", unfold(X, 0).shape)
4     print("Mode-1 unfolding shape:", unfold(X, 1).shape)
5     print("Mode-2 unfolding shape:", unfold(X, 2).shape)
6
7     M = tl.tensor([[1, 0, 0],

```

```

8         [0, 2, 0]], dtype=float) # shape (2,3)
9
10     X2 = mode_dot(X, M, mode=0)
11     print("Original shape:", X.shape)
12     print("After mode-0 multiplication:", X2.shape)
13 else:
14     print("Install TensorLy to run this section.")

```

8. Tucker decomposition

```

1 if TENSORLY_OK:
2     X = tl.tensor([
3         [[1.0, 2.0],
4          [3.0, 4.0]],
5
6         [[5.0, 6.0],
7          [7.0, 8.0]]
8     ])
9
10    core, factors = tucker(X, rank=[1, 2, 2])
11    X_hat = tucker_to_tensor((core, factors))
12
13    print("Core tensor:\n", core)
14    for i, f in enumerate(factors):
15        print(f"\nFactor matrix for mode {i}:\n", f)
16
17    print("\nReconstruction:\n", X_hat)
18 else:
19    print("Install TensorLy to run this section.")

```

9. Residual tensor and Q statistic

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

```

1 if TENSORLY_OK:
2     E = X - X_hat
3     Q = tl.sum(E * E)
4
5     print("Residual tensor:\n", E)
6     print("\nQ statistic =", float(Q))
7 else:
8     print("Install TensorLy to run this section.")

```

10. Why this matters for tensor SPC

This notebook 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

Tucker Decomposition and HOSVD ²

Workflow

This notebook 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

1. Imports

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 try:
5     import tensorly as tl
6     from tensorly.decomposition import tucker
7     from tensorly.tucker_tensor import tucker_to_tensor
8     from tensorly import unfold
9     TENSORLY_OK = True
10 except Exception as e:
11     TENSORLY_OK = False
12     print("TensorLy is not available:", e)
```

2. Build a structured synthetic tensor

Think of the tensor as **parts** × **variables** × **time points**.

```

1 np.random.seed(7)
2
3 n_parts = 20
4 n_vars = 5
5 n_time = 10
6
7 part_pattern = np.linspace(-1, 1, n_parts)[: , None, None]
8 var_pattern = np.array([1.0, 0.8, 1.2, 0.9, 1.1])[None, :, None]
9 time_pattern = np.sin(np.linspace(0, 2*np.pi, n_time))[None, None, :]
10
11 signal = 2.0 * part_pattern * var_pattern + 1.5 * var_pattern * time_pattern
12 noise = 0.15 * np.random.randn(n_parts, n_vars, n_time)
13
14 X_np = signal + noise
15 print("Tensor shape:", X_np.shape)

```

3. Visualize one slice

```

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

```

4. Tucker decomposition

```

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

```

5. Inspect factor matrices

```

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

```

6. Reconstruction error

```

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

```

7. Compare ranks

```

1 if TENSORLY_OK:
2     rank_options = [
3         [1, 1, 1],
4         [2, 2, 2],
5         [3, 3, 3],
6         [4, 4, 4]
7     ]
8
9     errors = []
10    for r in rank_options:
11        core_r, factors_r = tucker(X, rank=r)
12        X_hat_r = tucker_to_tensor((core_r, factors_r))
13        err = np.linalg.norm(np.array(X - X_hat_r)) / np.linalg.norm(X_np)
14        errors.append(err)
15
16    for r, e in zip(rank_options, errors):
17        print(f"Rank {r}: relative error = {e:.6f}")

```

8. Plot reconstruction error vs rank

```

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

```

9. SPC interpretation

Within this framework, the decomposition gives you:

- 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.

Theoretical Foundations and Development of Tensor SPC

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.

Assumptions

Let $\{\mathcal{X}_t\}_{t=1}^T$, where $\mathcal{X}_t \in \mathbb{R}^{I_1 \times \dots \times I_N}$, denote Phase I observations.

- **A1 (Centering and scaling).** Data are mode-wise centered and scaled.
- **A2 (Signal + noise).**

$$\mathcal{X}_t = \mathcal{S}_t + \mathcal{E}_t$$
- **A3 (Noise).** $\text{vec}(\mathcal{E}_t)$ is sub-Gaussian or Gaussian.
- **A4 (Independence).** Observations are independent or weakly dependent.
- **A5 (Identifiability).** Factor matrices are orthonormal.
- **A6 (Rank selection).** Rank captures signal without overfitting noise.

Multilinear Representation

A Tucker decomposition is given by

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

This preserves multiway structure while reducing dimensionality.

Monitoring Statistics

Let $\mathbf{z}_t = \text{vec}(\mathcal{G}_t)$.

Tensor T^2

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

Tensor Q

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

Complementarity

- T^2 : detects structured variation
- Q : detects residual/unmodeled variation

Theoretical Properties

Theorem 1 (Tensor T^2 Distribution)

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

Theorem 2 (Residual Q Approximation)

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

Theorem 3 (Sensitivity of T^2)

T^2 is sensitive to structured shifts in the modeled subspace.

Theorem 4 (Sensitivity of Q)

Q is sensitive to deviations outside the modeled structure.

Corollary

T^2 and Q provide complementary detection behavior.

Control Limits

$$\alpha \approx \frac{1}{ARL_0}$$

Control limits may be obtained from χ^2 approximations or empirical estimates.

Relation to Interaction Structure

Tensor SPC captures structured covariance across modes rather than explicit parametric interaction effects. It identifies latent joint variation without assigning causal interpretation.

Implementation Considerations**Data organization**

Structure data as variable \times time \times condition (or similar).

Rank selection

Balance signal capture vs noise.

Scaling

Apply consistent centering and scaling.

Computation

Use efficient decomposition methods for large data.

Extensions

- EWMA and CUSUM extensions
- Adaptive / nonstationary models
- Integration with machine learning

Summary

Tensor SPC generalizes MSPC by preserving multiway structure and enabling complementary monitoring through T^2 and Q statistics.

Part II

Tensor Monitoring Fundamentals

Residuals, Q Statistic, and Tensor SPC Logic

This notebook shows 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

1. Imports

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 try:
5     import tensorly as tl
6     from tensorly.decomposition import tucker
7     from tensorly.tucker_tensor import tucker_to_tensor
8     from tensorly.tenalg import mode_dot
9     TENSORLY_OK = True
10 except Exception as e:
11     TENSORLY_OK = False
12     print("TensorLy is not available:", e)
```

2. Simulate normal-reference tensor data

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

```
1 np.random.seed(123)
2
3 n_samples = 60
```

```

4 n_vars = 4
5 n_time = 12
6
7 base_time = np.sin(np.linspace(0, 2*np.pi, n_time))
8 base_var = np.array([1.0, 0.9, 1.1, 1.05])
9
10 X_ref = np.zeros((n_samples, n_vars, n_time))
11 for i in range(n_samples):
12     amplitude = 1.0 + 0.05*np.random.randn()
13     noise = 0.08*np.random.randn(n_vars, n_time)
14     X_ref[i] = amplitude * np.outer(base_var, base_time) + noise
15
16 print("Reference tensor shape:", X_ref.shape)

```

3. Fit Tucker model to reference data

```

1 if TENSORLY_OK:
2     X_ref_t1 = tl.tensor(X_ref, dtype=float)
3     rank = [3, 3, 3]
4     core_ref, factors_ref = tucker(X_ref_t1, rank=rank)
5     X_ref_hat = tucker_to_tensor((core_ref, factors_ref))
6     E_ref = np.array(X_ref_t1 - X_ref_hat)
7     Q_ref = np.sum(E_ref**2, axis=(1, 2))
8     print("Computed Q values for reference set.")
9 else:
10    print("Install TensorLy to run this section.")

```

4. Reference Q distribution and empirical limit

```

1 if TENSORLY_OK:
2     q_limit = np.percentile(Q_ref, 99)
3
4     print("Reference Q mean:", Q_ref.mean())
5     print("Reference Q std:", Q_ref.std())
6     print("Empirical 99th percentile limit:", q_limit)

```

5. Plot reference Q values

```

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

```

```

6 plt.xlabel("Sample index")
7 plt.ylabel("Q statistic")
8 plt.show()

```

6. Create new test samples, including abnormal ones

```

1 X_test = np.zeros((20, n_vars, n_time))
2 for i in range(20):
3     amplitude = 1.0 + 0.05*np.random.randn()
4     noise = 0.08*np.random.randn(n_vars, n_time)
5     X_test[i] = amplitude * np.outer(base_var, base_time) + noise
6
7 # Inject abnormalities in the last few samples
8 X_test[15:, 2, 5:8] += 0.8
9 X_test[17:, 0, :] += 0.4
10
11 print("Test tensor shape:", X_test.shape)

```

7. Simple projection-based approximation using learned factors

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

```

1 if TENSORLY_OK:
2     U_sample, U_var, U_time = [np.array(f) for f in factors_ref]
3
4     # Use variable and time factors only for per-sample reconstruction demonstration
5     Uv = U_var
6     Ut = U_time
7
8     def reconstruct_sample(sample, Uv, Ut):
9         # sample shape: (n_vars, n_time)
10        core = Uv.T @ sample @ Ut
11        recon = Uv @ core @ Ut.T
12        return recon
13
14    Q_test = []
15    X_test_hat = np.zeros_like(X_test)
16
17    for i in range(X_test.shape[0]):
18        recon = reconstruct_sample(X_test[i], Uv, Ut)
19        X_test_hat[i] = recon
20        resid = X_test[i] - recon
21        Q_test.append(np.sum(resid**2))
22
23    Q_test = np.array(Q_test)

```

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

8. Plot test Q statistics against the limit

```
1 if TENSORLY_OK:
2     plt.figure(figsize=(9, 4))
3     plt.plot(Q_test, marker='o')
4     plt.axhline(q_limit, linestyle='--')
5     plt.title("Test-set Q statistics")
6     plt.xlabel("Test sample index")
7     plt.ylabel("Q statistic")
8     plt.show()
9
10    flagged = np.where(Q_test > q_limit)[0]
11    print("Flagged sample indices:", flagged)
```

9. Compare one normal and one abnormal sample

```
1 if TENSORLY_OK:
2     normal_idx = 2
3     abnormal_idx = 18
4
5     plt.figure(figsize=(8, 4))
6     plt.imshow(X_test[normal_idx], aspect='auto')
7     plt.colorbar()
8     plt.title(f"Normal-like sample {normal_idx}")
9     plt.xlabel("Time")
10    plt.ylabel("Variable")
11    plt.show()
12
13    plt.figure(figsize=(8, 4))
14    plt.imshow(X_test[abnormal_idx], aspect='auto')
15    plt.colorbar()
16    plt.title(f"Abnormal sample {abnormal_idx}")
17    plt.xlabel("Time")
18    plt.ylabel("Variable")
19    plt.show()
```

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.

Synthetic Process Data Tensor Monitoring

This notebook creates a more process-style example where each sample is a **part** \times **feature** \times **time** measurement object.

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

1. Imports

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 try:
5     import tensorly as tl
6     from tensorly.decomposition import tucker
7     from tensorly.tucker_tensor import tucker_to_tensor
8     TENSORLY_OK = True
9 except Exception as e:
10    TENSORLY_OK = False
11    print("TensorLy is not available:", e)
```

2. Simulate process data

```
1 np.random.seed(2026)
2
3 n_lots = 40
4 n_features = 6
5 n_stages = 15
6
7 feature_profile = np.array([1.0, 0.7, 1.2, 0.9, 1.1, 0.8])
8 stage_profile = np.linspace(-1, 1, n_stages)
9
10 X = np.zeros((n_lots, n_features, n_stages))
11 for i in range(n_lots):
12     lot_shift = 0.12 * np.random.randn()
```

```

13     trend_strength = 1.0 + 0.08 * np.random.randn()
14     structured = trend_strength * np.outer(feature_profile, stage_profile)
15     noise = 0.10 * np.random.randn(n_features, n_stages)
16     X[i] = structured + lot_shift + noise
17
18     # add abnormal lots
19     X[34:, 4, 8:12] += 0.9
20     X[37:, 1, :] -= 0.5
21
22     print("Process tensor shape:", X.shape)

```

3. Fit a low-rank tensor model

```

1  if TENSORLY_OK:
2      X_t1 = tl.tensor(X, dtype=float)
3      rank = [4, 3, 3]
4      core, factors = tucker(X_t1, rank=rank)
5      X_hat = tucker_to_tensor((core, factors))
6      E = np.array(X_t1 - X_hat)
7      Q = np.sum(E**2, axis=(1, 2))
8      print("Finished decomposition.")
9  else:
10     print("Install TensorLy to run this section.")

```

4. Plot Q statistics

```

1  if TENSORLY_OK:
2      q_limit = np.percentile(Q[:30], 99)
3
4      plt.figure(figsize=(10, 4))
5      plt.plot(Q, marker='o')
6      plt.axhline(q_limit, linestyle='--')
7      plt.title("Lot-wise Q statistic")
8      plt.xlabel("Lot index")
9      plt.ylabel("Q")
10     plt.show()
11
12     print("Flagged lots:", np.where(Q > q_limit)[0])

```

5. Visualize a typical lot and an abnormal lot

```

1  good_idx = 5
2  bad_idx = 38

```

```
3
4 plt.figure(figsize=(8, 4))
5 plt.imshow(X[good_idx], aspect='auto')
6 plt.colorbar()
7 plt.title(f"Typical lot {good_idx}")
8 plt.xlabel("Stage")
9 plt.ylabel("Feature")
10 plt.show()
11
12 plt.figure(figsize=(8, 4))
13 plt.imshow(X[bad_idx], aspect='auto')
14 plt.colorbar()
15 plt.title(f"Abnormal lot {bad_idx}")
16 plt.xlabel("Stage")
17 plt.ylabel("Feature")
18 plt.show()
```

6. Residual localization

```
1 if TENSORLY_OK:
2     R_good = X[good_idx] - np.array(X_hat)[good_idx]
3     R_bad = X[bad_idx] - np.array(X_hat)[bad_idx]
4
5     plt.figure(figsize=(8, 4))
6     plt.imshow(R_good, aspect='auto')
7     plt.colorbar()
8     plt.title(f"Residual map for lot {good_idx}")
9     plt.xlabel("Stage")
10    plt.ylabel("Feature")
11    plt.show()
12
13    plt.figure(figsize=(8, 4))
14    plt.imshow(R_bad, aspect='auto')
15    plt.colorbar()
16    plt.title(f"Residual map for lot {bad_idx}")
17    plt.xlabel("Stage")
18    plt.ylabel("Feature")
19    plt.show()
```

7. Interpretation in this context

This workflow establishes a structure for 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, you can extend toward:

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

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

1. Imports

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 try:
5     import tensorly as tl
6     from tensorly.decomposition import tucker
7     from tensorly.tucker_tensor import tucker_to_tensor
8     TENSORLY_OK = True
9 except Exception as e:
10     TENSORLY_OK = False
11     print("TensorLy is not available:", e)
```

2. Simulate in-control tensor samples

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

```

1  np.random.seed(321)
2
3  n_samples = 80
4  n_vars = 5
5  n_time = 15
6
7  base_time = np.sin(np.linspace(0, 2*np.pi, n_time))
8  base_var = np.array([1.0, 0.9, 1.1, 1.05, 0.95])
9
10 X_ref = np.zeros((n_samples, n_vars, n_time))
11 for i in range(n_samples):
12     amp = 1.0 + 0.06*np.random.randn()
13     phase_shift = 0.08*np.random.randn()
14     shifted_time = np.sin(np.linspace(0, 2*np.pi, n_time) + phase_shift)
15     trend = 0.03*np.random.randn() * np.linspace(-1, 1, n_time)
16     noise = 0.08*np.random.randn(n_vars, n_time)
17     X_ref[i] = amp * np.outer(base_var, shifted_time) + np.outer(base_var, trend) +
18         noise
19 print("Reference tensor shape:", X_ref.shape)

```

3. Fit a Tucker model to the reference tensor

```

1  if TENSORLY_OK:
2     X_ref_t1 = tl.tensor(X_ref, dtype=float)
3     rank = [4, 3, 3]
4     core_ref, factors_ref = tucker(X_ref_t1, rank=rank)
5     X_ref_hat = tucker_to_tensor((core_ref, factors_ref))
6
7     print("Core shape:", core_ref.shape)
8     for i, f in enumerate(factors_ref):
9         print(f"Factor {i} shape:", f.shape)
10 else:
11     print("Install TensorLy to run this section.")

```

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.

```

1  if TENSORLY_OK:
2      U_sample, U_var, U_time = [np.array(f) for f in factors_ref]
3
4      def sample_score_matrix(sample, U_var, U_time):
5          return U_var.T @ sample @ U_time
6
7      def sample_score_vector(sample, U_var, U_time):
8          G = sample_score_matrix(sample, U_var, U_time)
9          return G.reshape(-1)
10
11     score_vectors_ref = np.array([
12         sample_score_vector(X_ref[i], U_var, U_time) for i in range(n_samples)
13     ])
14
15     print("Reference score matrix shape:", score_vectors_ref.shape)

```

5. Estimate the score-space center and covariance

```

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

```

6. Define the T^2 statistic

```

1  if TENSORLY_OK:
2      def hotelling_t2(score_vec, mu, S_inv):
3          d = score_vec - mu
4          return float(d.T @ S_inv @ d)
5
6      T2_ref = np.array([hotelling_t2(s, mu_scores, S_inv) for s in score_vectors_ref])
7      print(T2_ref[:5])

```

7. Define the Q statistic for the same samples

For comparison, compute Q using the reconstruction residual.

```

1  if TENSORLY_OK:
2      def reconstruct_sample(sample, U_var, U_time):
3          G = U_var.T @ sample @ U_time
4          return U_var @ G @ U_time.T
5

```

```

6     Q_ref = []
7     for i in range(n_samples):
8         recon = reconstruct_sample(X_ref[i], U_var, U_time)
9         resid = X_ref[i] - recon
10        Q_ref.append(np.sum(resid**2))
11    Q_ref = np.array(Q_ref)

```

8. Set empirical control limits

```

1  if TENSORLY_OK:
2      t2_limit = np.percentile(T2_ref, 99)
3      q_limit = np.percentile(Q_ref, 99)
4      print("Empirical 99th percentile T2 limit:", t2_limit)
5      print("Empirical 99th percentile Q limit:", q_limit)

```

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

```

1  X_test = np.zeros((25, n_vars, n_time))
2  for i in range(25):
3      amp = 1.0 + 0.06*np.random.randn()
4      phase_shift = 0.08*np.random.randn()
5      shifted_time = np.sin(np.linspace(0, 2*np.pi, n_time) + phase_shift)
6      trend = 0.03*np.random.randn() * np.linspace(-1, 1, n_time)
7      noise = 0.08*np.random.randn(n_vars, n_time)
8      X_test[i] = amp * np.outer(base_var, shifted_time) + np.outer(base_var, trend) +
9          noise
10
10 X_test[10:15] *= 1.35
11 X_test[18:, 3, 6:10] += 0.9
12 X_test[20:, 1, :] -= 0.35

```

10. Compute test T^2 and Q

```

1  if TENSORLY_OK:
2      T2_test = []
3      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)
```

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

Tensor SPC vs Vectorized PCA Baseline

This notebook 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.

1. Imports

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 try:
5     import tensorly as tl
6     from tensorly.decomposition import tucker
7     from tensorly.tucker_tensor import tucker_to_tensor
8     TENSORLY_OK = True
9 except Exception as e:
10     TENSORLY_OK = False
11     print("TensorLy is not available:", e)
12
13 from sklearn.decomposition import PCA
```

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

```

1 np.random.seed(2027)
2
3 n_ref = 100
4 n_test = 40
5 n_vars = 6
6 n_time = 20
7
8 base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
9 time_axis = np.linspace(0, 2*np.pi, n_time)
10
11 def generate_normal_sample():
12     amp = 1.0 + 0.05*np.random.randn()
13     phase = 0.10*np.random.randn()
14     time_profile = np.sin(time_axis + phase) + 0.35*np.cos(2 * time_axis + 0.5 * phase
15     )
16     drift = 0.03 * np.random.randn() * np.linspace(-1, 1, n_time)
17     interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
18     noise = 0.08 * np.random.randn(n_vars, n_time)
19     return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
20     interaction + noise
21
22 X_ref = np.array([generate_normal_sample() for _ in range(n_ref)])
23 print("Reference tensor shape:", X_ref.shape)

```

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

```

1 X_test = np.array([generate_normal_sample() for _ in range(n_test)])
2 fault_labels = np.array(["normal"] * n_test, dtype=object)
3
4 # Fault type 1: coherent global shift (likely to affect T2 strongly)
5 for i in range(10, 18):
6     X_test[i] *= 1.30
7     fault_labels[i] = "global_shift"
8
9 # Fault type 2: localized patch anomaly (often shows up strongly in Q)

```

```

10 for i in range(18, 28):
11     X_test[i, 3, 7:12] += 0.9
12     fault_labels[i] = "patch_anomaly"
13
14 # Fault type 3: correlation / shape change
15 for i in range(28, 40):
16     altered = generate_normal_sample()
17     altered += 0.35 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
18     X_test[i] = altered
19     fault_labels[i] = "shape_change"
20
21 unique, counts = np.unique(fault_labels, return_counts=True)
22 dict(zip(unique, counts))

```

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

```

1 if TENSORLY_OK:
2     X_ref_t1 = t1.tensor(X_ref, dtype=float)
3     rank = [5, 3, 3]
4     core_ref, factors_ref = tucker(X_ref_t1, rank=rank)
5
6     U_sample, U_var, U_time = [np.array(f) for f in factors_ref]
7
8     def tensor_score_vector(sample, U_var, U_time):
9         G = U_var.T @ sample @ U_time
10        return G.reshape(-1)
11
12    def tensor_reconstruct(sample, U_var, U_time):
13        G = U_var.T @ sample @ U_time
14        return U_var @ G @ U_time.T
15
16    tensor_scores_ref = np.array([
17        tensor_score_vector(X_ref[i], U_var, U_time) for i in range(n_ref)
18    ])
19
20    mu_tensor = tensor_scores_ref.mean(axis=0)
21    S_tensor = np.cov(tensor_scores_ref, rowvar=False)
22    S_tensor_inv = np.linalg.inv(S_tensor + 1e-6 * np.eye(S_tensor.shape[0]))
23
24    def t2_stat(score_vec, mu, S_inv):
25        d = score_vec - mu

```

```

26     return float(d.T @ S_inv @ d)
27
28     T2_tensor_ref = np.array([t2_stat(s, mu_tensor, S_tensor_inv) for s in
29                               tensor_scores_ref])
30
31     Q_tensor_ref = []
32     for i in range(n_ref):
33         recon = tensor_reconstruct(X_ref[i], U_var, U_time)
34         resid = X_ref[i] - recon
35         Q_tensor_ref.append(np.sum(resid**2))
36     Q_tensor_ref = np.array(Q_tensor_ref)
37
38     t2_tensor_limit = np.percentile(T2_tensor_ref, 99)
39     q_tensor_limit = np.percentile(Q_tensor_ref, 99)
40
41     print("Tensor T2 limit:", t2_tensor_limit)
42     print("Tensor Q limit:", q_tensor_limit)
43 else:
44     print("Install TensorLy to run this section.")

```

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

```

1 X_ref_vec = X_ref.reshape(n_ref, -1)
2 X_test_vec = X_test.reshape(n_test, -1)
3
4 n_components = 8
5 pca = PCA(n_components=n_components)
6 scores_ref = pca.fit_transform(X_ref_vec)
7 scores_test = pca.transform(X_test_vec)
8
9 mu_pca = scores_ref.mean(axis=0)
10 S_pca = np.cov(scores_ref, rowvar=False)
11 S_pca_inv = np.linalg.inv(S_pca + 1e-6 * np.eye(S_pca.shape[0]))
12
13 def hotelling_t2(score_vec, mu, S_inv):
14     d = score_vec - mu
15     return float(d.T @ S_inv @ d)
16
17 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)

```

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. 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))

```

```

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

```

8. Plot PCA baseline statistics

```

1 plt.figure(figsize=(10, 4))
2 plt.plot(T2_pca_test, marker='o')
3 plt.axhline(t2_pca_limit, linestyle='--')
4 plt.title("Vectorized PCA T2 on test data")
5 plt.xlabel("Test sample index")
6 plt.ylabel("PCA T2")
7 plt.show()
8
9 plt.figure(figsize=(10, 4))
10 plt.plot(Q_pca_test, marker='o')
11 plt.axhline(q_pca_limit, linestyle='--')
12 plt.title("Vectorized PCA Q on test data")
13 plt.xlabel("Test sample index")
14 plt.ylabel("PCA Q")
15 plt.show()

```

9. Flagged samples by method

```

1 results = {}
2
3 if TENSORLY_OK:
4     results["tensor_T2"] = np.where(T2_tensor_test > t2_tensor_limit)[0]
5     results["tensor_Q"] = np.where(Q_tensor_test > q_tensor_limit)[0]
6
7 results["pca_T2"] = np.where(T2_pca_test > t2_pca_limit)[0]
8 results["pca_Q"] = np.where(Q_pca_test > q_pca_limit)[0]
9
10 results

```

10. Detection summary by fault type

```

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

```

```

4     summary = {}
5     for label in np.unique(fault_labels):
6         mask = fault_labels == label
7         summary[label] = {
8             "count": int(mask.sum()),
9             "flagged": int(flagged[mask].sum()),
10            "rate": float(flagged[mask].mean())
11        }
12     return summary
13
14     summary = {}
15     if TENSORLY_OK:
16         summary["tensor_T2"] = summarize_detection(results["tensor_T2"], fault_labels)
17         summary["tensor_Q"] = summarize_detection(results["tensor_Q"], fault_labels)
18
19     summary["pca_T2"] = summarize_detection(results["pca_T2"], fault_labels)
20     summary["pca_Q"] = summarize_detection(results["pca_Q"], fault_labels)
21
22     summary

```

11. Compare methods in scatter plots

```

1     if TENSORLY_OK:
2         plt.figure(figsize=(6, 5))
3         for label in np.unique(fault_labels):
4             mask = fault_labels == label
5             plt.scatter(T2_tensor_test[mask], Q_tensor_test[mask], label=label)
6             plt.axvline(t2_tensor_limit, linestyle='--')
7             plt.axhline(q_tensor_limit, linestyle='--')
8             plt.xlabel("Tensor T2")
9             plt.ylabel("Tensor Q")
10            plt.title("Tensor monitoring space")
11            plt.legend()
12            plt.show()
13
14     plt.figure(figsize=(6, 5))
15     for label in np.unique(fault_labels):
16         mask = fault_labels == label
17         plt.scatter(T2_pca_test[mask], Q_pca_test[mask], label=label)
18     plt.axvline(t2_pca_limit, linestyle='--')
19     plt.axhline(q_pca_limit, linestyle='--')
20     plt.xlabel("PCA T2")
21     plt.ylabel("PCA Q")
22     plt.title("Vectorized PCA monitoring space")
23     plt.legend()
24     plt.show()

```

12. Inspect a few example samples

```
1 example_indices = [5, 12, 22, 34]
2 example_titles = {
3     5: "Normal example",
4     12: "Global shift example",
5     22: "Patch anomaly example",
6     34: "Shape change example"
7 }
8
9 for idx in example_indices:
10     plt.figure(figsize=(8, 4))
11     plt.imshow(X_test[idx], aspect='auto')
12     plt.colorbar()
13     plt.title(f"{example_titles[idx]} (index {idx})")
14     plt.xlabel("Time")
15     plt.ylabel("Variable")
16     plt.show()
```

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 notebook provides a fair baseline. If tensor SPC is to be meaningful in this work, it should demonstrate advantages in structure-aware detection, interpretability, or robustness.

14. Ideas for extending this notebook

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

This would transition the notebook from demonstration to publishable simulation evidence.

Monte Carlo Comparison of Tensor SPC and PCA

This notebook 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.

1. Imports

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 try:
6     import tensorly as tl
7     from tensorly.decomposition import tucker
8     from tensorly.tucker_tensor import tucker_to_tensor
9     TENSORLY_OK = True
10 except Exception as e:
11     TENSORLY_OK = False
12     print("TensorLy is not available:", e)
13
14 from sklearn.decomposition import PCA
```

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.

```
1 N_RUNS = 50
2
3 n_ref = 80
4 n_test_normal = 20
5 n_test_per_fault = 12
6
7 n_vars = 6
8 n_time = 20
9
10 tensor_rank = [5, 3, 3]
11 pca_components = 8
12
13 alpha_percentile = 99
14
15 base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
16 time_axis = np.linspace(0, 2*np.pi, n_time)
17
18 print("Monte Carlo runs:", N_RUNS)
```

3. Data generation functions

```
1 def generate_normal_sample(rng):
2     amp = 1.0 + 0.05 * rng.standard_normal()
3     phase = 0.10 * rng.standard_normal()
4     time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
5         phase)
6     drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
7     interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
8     noise = 0.08 * rng.standard_normal((n_vars, n_time))
9     return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
10     interaction + noise
11
12
13 def inject_global_shift(sample):
14     return 1.30 * sample
15
16
17 def inject_patch_anomaly(sample):
18     x = sample.copy()
19     x[3, 7:12] += 0.9
```

```

16     return x
17
18 def inject_shape_change(sample):
19     x = sample.copy()
20     x += 0.35 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
21     return x

```

4. Monitoring helper functions

```

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 summarize_rate(flags):
6     flags = np.asarray(flags, dtype=float)
7     return {
8         "mean": float(flags.mean()),
9         "std": float(flags.std(ddof=1)) if len(flags) > 1 else 0.0
10    }

```

5. One Monte Carlo run

```

1 def run_one_simulation(seed):
2     rng = np.random.default_rng(seed)
3
4     # Reference data
5     X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
6
7     # Test data by fault type
8     X_test_normal = np.array([generate_normal_sample(rng) for _ in range(n_test_normal)
9                               ])
9     X_test_global = np.array([inject_global_shift(generate_normal_sample(rng)) for _
10                              in range(n_test_per_fault)])
10    X_test_patch = np.array([inject_patch_anomaly(generate_normal_sample(rng)) for _
11                             in range(n_test_per_fault)])
11    X_test_shape = np.array([inject_shape_change(generate_normal_sample(rng)) for _ in
12                             range(n_test_per_fault)])
12
13    # -----
14    # Tensor monitoring
15    # -----
16    tensor_results = None
17    if TENSORLY_OK:
18        X_ref_t1 = tl.tensor(X_ref, dtype=float)
19        core_ref, factors_ref = tucker(X_ref_t1, rank=tensor_rank)

```

```

20     U_sample, U_var, U_time = [np.array(f) for f in factors_ref]
21
22     def tensor_score_vector(sample):
23         G = U_var.T @ sample @ U_time
24         return G.reshape(-1)
25
26     def tensor_reconstruct(sample):
27         G = U_var.T @ sample @ U_time
28         return U_var @ G @ U_time.T
29
30     tensor_scores_ref = np.array([tensor_score_vector(X_ref[i]) for i in range(
n_ref)])
31     mu_tensor = tensor_scores_ref.mean(axis=0)
32     S_tensor = np.cov(tensor_scores_ref, rowvar=False)
33     S_tensor_inv = np.linalg.inv(S_tensor + 1e-6 * np.eye(S_tensor.shape[0]))
34
35     T2_tensor_ref = np.array([hotelling_t2(s, mu_tensor, S_tensor_inv) for s in
tensor_scores_ref])
36     Q_tensor_ref = np.array([
37         np.sum((X_ref[i] - tensor_reconstruct(X_ref[i]))**2) for i in range(n_ref)
38     ])
39
40     t2_tensor_limit = np.percentile(T2_tensor_ref, alpha_percentile)
41     q_tensor_limit = np.percentile(Q_tensor_ref, alpha_percentile)
42
43     def tensor_flags(X_test):
44         scores = np.array([tensor_score_vector(x) for x in X_test])
45         T2 = np.array([hotelling_t2(s, mu_tensor, S_tensor_inv) for s in scores])
46         Q = np.array([np.sum((x - tensor_reconstruct(x))**2) for x in X_test])
47         return {
48             "T2": T2 > t2_tensor_limit,
49             "Q": Q > q_tensor_limit
50         }
51
52     tensor_results = {
53         "normal": tensor_flags(X_test_normal),
54         "global_shift": tensor_flags(X_test_global),
55         "patch_anomaly": tensor_flags(X_test_patch),
56         "shape_change": tensor_flags(X_test_shape)
57     }
58
59     # -----
60     # PCA baseline
61     # -----
62     X_ref_vec = X_ref.reshape(n_ref, -1)
63     pca = PCA(n_components=pca_components)
64     scores_ref = pca.fit_transform(X_ref_vec)
65
66     mu_pca = scores_ref.mean(axis=0)
67     S_pca = np.cov(scores_ref, rowvar=False)

```

```

68 S_pca_inv = np.linalg.inv(S_pca + 1e-6 * np.eye(S_pca.shape[0]))
69
70 T2_pca_ref = np.array([hotelling_t2(s, mu_pca, S_pca_inv) for s in scores_ref])
71 X_ref_hat = pca.inverse_transform(scores_ref)
72 Q_pca_ref = np.sum((X_ref_vec - X_ref_hat)**2, axis=1)
73
74 t2_pca_limit = np.percentile(T2_pca_ref, alpha_percentile)
75 q_pca_limit = np.percentile(Q_pca_ref, alpha_percentile)
76
77 def pca_flags(X_test):
78     Xv = X_test.reshape(X_test.shape[0], -1)
79     scores = pca.transform(Xv)
80     T2 = np.array([hotelling_t2(s, mu_pca, S_pca_inv) for s in scores])
81     Xhat = pca.inverse_transform(scores)
82     Q = np.sum((Xv - Xhat)**2, axis=1)
83     return {
84         "T2": T2 > t2_pca_limit,
85         "Q": Q > q_pca_limit
86     }
87
88 pca_results = {
89     "normal": pca_flags(X_test_normal),
90     "global_shift": pca_flags(X_test_global),
91     "patch_anomaly": pca_flags(X_test_patch),
92     "shape_change": pca_flags(X_test_shape)
93 }
94
95 return tensor_results, pca_results

```

6. Run the Monte Carlo study

```

1 records = []
2
3 for run in range(N_RUNS):
4     tensor_results, pca_results = run_one_simulation(seed=1000 + run)
5
6     fault_types = ["normal", "global_shift", "patch_anomaly", "shape_change"]
7
8     if tensor_results is not None:
9         for fault in fault_types:
10             for stat in ["T2", "Q"]:
11                 records.append({
12                     "run": run,
13                     "method": "tensor",
14                     "statistic": stat,
15                     "fault_type": fault,
16                     "flag_rate": float(np.mean(tensor_results[fault][stat]))
17                 })

```

```
18
19     for fault in fault_types:
20         for stat in ["T2", "Q"]:
21             records.append({
22                 "run": run,
23                 "method": "pca",
24                 "statistic": stat,
25                 "fault_type": fault,
26                 "flag_rate": float(np.mean(pca_results[fault][stat]))
27             })
28
29 results_df = pd.DataFrame(records)
30 results_df.head()
```

7. Aggregate results

```
1 summary_df = (
2     results_df
3     .groupby(["method", "statistic", "fault_type"])["flag_rate"]
4     .agg(["mean", "std"])
5     .reset_index()
6 )
7
8 summary_df
```

8. Separate false alarm and detection summaries

```
1 false_alarm_df = summary_df[summary_df["fault_type"] == "normal"].copy()
2 detection_df = summary_df[summary_df["fault_type"] != "normal"].copy()
3
4 print("False alarm summary:")
5 display(false_alarm_df)
6
7 print("\nDetection summary:")
8 display(detection_df)
```

9. Plot false alarm rates

```
1 plot_df = false_alarm_df.copy()
2 labels = plot_df["method"] + "_" + plot_df["statistic"]
3
4 plt.figure(figsize=(8, 4))
5 plt.bar(labels, plot_df["mean"], yerr=plot_df["std"], capsize=5)
```

```

6 plt.ylabel("Average false alarm rate")
7 plt.title("Monte Carlo false alarm comparison")
8 plt.show()

```

10. Plot detection rates by fault type

```

1 faults = ["global_shift", "patch_anomaly", "shape_change"]
2
3 for fault in faults:
4     sub = detection_df[detection_df["fault_type"] == fault].copy()
5     labels = sub["method"] + "_" + sub["statistic"]
6
7     plt.figure(figsize=(8, 4))
8     plt.bar(labels, sub["mean"], yerr=sub["std"], capsize=5)
9     plt.ylim(0, 1.05)
10    plt.ylabel("Average detection rate")
11    plt.title(f"Detection rate comparison - {fault}")
12    plt.show()

```

11. Pivot tables for easier comparison

```

1 pivot_mean = summary_df.pivot_table(
2     index="fault_type",
3     columns=["method", "statistic"],
4     values="mean"
5 )
6
7 pivot_std = summary_df.pivot_table(
8     index="fault_type",
9     columns=["method", "statistic"],
10    values="std"
11 )
12
13 print("Mean flag rates:")
14 display(pivot_mean)
15
16 print("\nStandard deviations:")
17 display(pivot_std)

```

12. A compact method ranking view

```

1 rank_view = summary_df.copy()

```

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

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**.

14. Extensions for a stronger research study

Strong next steps 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

ARL and Detection Delay: Tensor SPC vs PCA

This notebook 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.

1. Imports

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 try:
6     import tensorly as tl
7     from tensorly.decomposition import Tucker
8     TENSORLY_OK = True
9 except Exception as e:
10     TENSORLY_OK = False
11     print("TensorLy is not available:", e)
12
13 from sklearn.decomposition import PCA
```

2. Simulation settings

```
1 N_RUNS = 40
2
3 # Reference data used to fit each monitoring model
4 n_ref = 100
5
6 # Sequential monitoring horizon
7 max_run_length = 300
8
9 # Change point for out-of-control sequences
10 change_point = 100
11
12 n_vars = 6
13 n_time = 20
14
15 tensor_rank = [5, 3, 3]
16 pca_components = 8
17 alpha_percentile = 99
18
19 base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
20 time_axis = np.linspace(0, 2*np.pi, n_time)
21
22 print("Runs:", N_RUNS)
23 print("Change point:", change_point)
24 print("Monitoring horizon:", max_run_length)
```

3. Data generation

```
1 def generate_normal_sample(rng):
2     amp = 1.0 + 0.05 * rng.standard_normal()
3     phase = 0.10 * rng.standard_normal()
4     time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
5         phase)
6     drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
7     interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
8     noise = 0.08 * rng.standard_normal((n_vars, n_time))
9     return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
10         interaction + noise
11
12 def inject_global_shift(sample):
13     return 1.25 * sample
14
15 def inject_patch_anomaly(sample):
16     x = sample.copy()
17     x[3, 7:12] += 0.9
18     return x
```

```

17
18 def inject_shape_change(sample):
19     x = sample.copy()
20     x += 0.35 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
21     return x

```

4. Monitoring model fitting

```

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 fit_monitoring_models(X_ref):
6     models = {}
7
8     # PCA baseline
9     X_ref_vec = X_ref.reshape(X_ref.shape[0], -1)
10    pca = PCA(n_components=pca_components)
11    scores_ref = pca.fit_transform(X_ref_vec)
12
13    mu_pca = scores_ref.mean(axis=0)
14    S_pca = np.cov(scores_ref, rowvar=False)
15    S_pca_inv = np.linalg.inv(S_pca + 1e-6 * np.eye(S_pca.shape[0]))
16
17    T2_pca_ref = np.array([hotelling_t2(s, mu_pca, S_pca_inv) for s in scores_ref])
18    X_ref_hat = pca.inverse_transform(scores_ref)
19    Q_pca_ref = np.sum((X_ref_vec - X_ref_hat)**2, axis=1)
20
21    models["pca"] = {
22        "pca": pca,
23        "mu": mu_pca,
24        "S_inv": S_pca_inv,
25        "t2_limit": np.percentile(T2_pca_ref, alpha_percentile),
26        "q_limit": np.percentile(Q_pca_ref, alpha_percentile)
27    }
28
29    # Tensor monitoring
30    if TENSORLY_OK:
31        X_ref_t1 = tl.tensor(X_ref, dtype=float)
32        core_ref, factors_ref = tucker(X_ref_t1, rank=tensor_rank)
33        U_sample, U_var, U_time = [np.array(f) for f in factors_ref]
34
35        def tensor_score_vector(sample):
36            G = U_var.T @ sample @ U_time
37            return G.reshape(-1)
38
39        def tensor_reconstruct(sample):
40            G = U_var.T @ sample @ U_time

```

```

41         return U_var @ G @ U_time.T
42
43     tensor_scores_ref = np.array([tensor_score_vector(X_ref[i]) for i in range(
X_ref.shape[0])])
44     mu_tensor = tensor_scores_ref.mean(axis=0)
45     S_tensor = np.cov(tensor_scores_ref, rowvar=False)
46     S_tensor_inv = np.linalg.inv(S_tensor + 1e-6 * np.eye(S_tensor.shape[0]))
47
48     T2_tensor_ref = np.array([hotelling_t2(s, mu_tensor, S_tensor_inv) for s in
tensor_scores_ref])
49     Q_tensor_ref = np.array([
50         np.sum((X_ref[i] - tensor_reconstruct(X_ref[i]))**2) for i in range(X_ref.
shape[0])
51     ])
52
53     models["tensor"] = {
54         "U_var": U_var,
55         "U_time": U_time,
56         "mu": mu_tensor,
57         "S_inv": S_tensor_inv,
58         "t2_limit": np.percentile(T2_tensor_ref, alpha_percentile),
59         "q_limit": np.percentile(Q_tensor_ref, alpha_percentile)
60     }
61
62     return models

```

5. Online statistics for one sample

```

1 def compute_pca_stats(sample, model):
2     x_vec = sample.reshape(1, -1)
3     score = model["pca"].transform(x_vec)[0]
4     t2 = hotelling_t2(score, model["mu"], model["S_inv"])
5     x_hat = model["pca"].inverse_transform(score.reshape(1, -1))[0]
6     q = float(np.sum((x_vec[0] - x_hat)**2))
7     return t2, q
8
9 def compute_tensor_stats(sample, model):
10    U_var = model["U_var"]
11    U_time = model["U_time"]
12    G = U_var.T @ sample @ U_time
13    score = G.reshape(-1)
14    t2 = hotelling_t2(score, model["mu"], model["S_inv"])
15    recon = U_var @ G @ U_time.T
16    q = float(np.sum((sample - recon)**2))
17    return t2, q

```

6. Run-length logic

```

1 def first_alarm_time(sequence_stats, limit):
2     alarms = np.where(sequence_stats > limit)[0]
3     if len(alarms) == 0:
4         return None
5     return int(alarms[0] + 1) # 1-based time index
6
7 def detection_delay(sequence_stats, limit, change_point):
8     post = np.where(sequence_stats[change_point:] > limit)[0]
9     if len(post) == 0:
10        return None
11    return int(post[0] + 1) # delay after the change point

```

7. Simulate one in-control run for ARL0

```

1 def simulate_in_control_run(seed):
2     rng = np.random.default_rng(seed)
3     X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
4     models = fit_monitoring_models(X_ref)
5
6     results = {}
7
8     # PCA
9     pca_t2 = []
10    pca_q = []
11    for _ in range(max_run_length):
12        x = generate_normal_sample(rng)
13        t2, q = compute_pca_stats(x, models["pca"])
14        pca_t2.append(t2)
15        pca_q.append(q)
16
17    results["pca_T2_run_length"] = first_alarm_time(np.array(pca_t2), models["pca"]["t2_limit"])
18    results["pca_Q_run_length"] = first_alarm_time(np.array(pca_q), models["pca"]["q_limit"])
19
20    # Tensor
21    if "tensor" in models:
22        tensor_t2 = []
23        tensor_q = []
24        for _ in range(max_run_length):
25            x = generate_normal_sample(rng)
26            t2, q = compute_tensor_stats(x, models["tensor"])
27            tensor_t2.append(t2)
28            tensor_q.append(q)
29

```

```

30     results["tensor_T2_run_length"] = first_alarm_time(np.array(tensor_t2), models
31     ["tensor"]["t2_limit"])
31     results["tensor_Q_run_length"] = first_alarm_time(np.array(tensor_q), models["
32     tensor"]["q_limit"])
32
33     return results

```

8. Simulate one out-of-control run for detection delay

```

1  def simulate_out_of_control_run(seed, fault_type):
2      rng = np.random.default_rng(seed)
3      X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
4      models = fit_monitoring_models(X_ref)
5
6      def apply_fault(x):
7          if fault_type == "global_shift":
8              return inject_global_shift(x)
9          elif fault_type == "patch_anomaly":
10             return inject_patch_anomaly(x)
11          elif fault_type == "shape_change":
12             return inject_shape_change(x)
13          else:
14             raise ValueError("Unknown fault type")
15
16     results = {}
17
18     # PCA
19     pca_t2 = []
20     pca_q = []
21     for t in range(max_run_length):
22         x = generate_normal_sample(rng)
23         if t >= change_point:
24             x = apply_fault(x)
25         t2, q = compute_pca_stats(x, models["pca"])
26         pca_t2.append(t2)
27         pca_q.append(q)
28
29     results["pca_T2_delay"] = detection_delay(np.array(pca_t2), models["pca"]["
30     t2_limit"], change_point)
30     results["pca_Q_delay"] = detection_delay(np.array(pca_q), models["pca"]["q_limit"],
31     change_point)
31
32     # Tensor
33     if "tensor" in models:
34         tensor_t2 = []
35         tensor_q = []
36         for t in range(max_run_length):
37             x = generate_normal_sample(rng)

```

```

38         if t >= change_point:
39             x = apply_fault(x)
40             t2, q = compute_tensor_stats(x, models["tensor"])
41             tensor_t2.append(t2)
42             tensor_q.append(q)
43
44             results["tensor_T2_delay"] = detection_delay(np.array(tensor_t2), models["
tensor"]["t2_limit"], change_point)
45             results["tensor_Q_delay"] = detection_delay(np.array(tensor_q), models["tensor
"] ["q_limit"], change_point)
46
47     return results

```

9. Estimate ARL0

```

1  arl_records = []
2
3  for run in range(N_RUNS):
4      res = simulate_in_control_run(seed=2000 + run)
5      for key, value in res.items():
6          arl_records.append({
7              "run": run,
8              "method_stat": key.replace("_run_length", ""),
9              "run_length": max_run_length if value is None else value
10         })
11
12  arl_df = pd.DataFrame(arl_records)
13  arl_summary = arl_df.groupby("method_stat")["run_length"].agg(["mean", "std"]).
    reset_index()
14
15  print("Estimated ARL0 summary:")
16  display(arl_summary)

```

10. Plot ARL0

```

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

```

11. Estimate detection delay for each fault type

```

1 fault_types = ["global_shift", "patch_anomaly", "shape_change"]
2 delay_records = []
3
4 for fault in fault_types:
5     for run in range(N_RUNS):
6         res = simulate_out_of_control_run(seed=5000 + 100*run, fault_type=fault)
7         for key, value in res.items():
8             delay_records.append({
9                 "fault_type": fault,
10                "run": run,
11                "method_stat": key.replace("_delay", ""),
12                "delay": (max_run_length - change_point) if value is None else value
13            })
14
15 delay_df = pd.DataFrame(delay_records)
16 delay_summary = delay_df.groupby(["fault_type", "method_stat"])["delay"].agg(["mean",
17                                     "std"]).reset_index()
18
19 print("Detection-delay summary:")
20 display(delay_summary)

```

12. Plot detection delay by fault type

```

1 for fault in fault_types:
2     sub = delay_summary[delay_summary["fault_type"] == fault]
3
4     plt.figure(figsize=(8, 4))
5     plt.bar(sub["method_stat"], sub["mean"], yerr=sub["std"], capsize=5)
6     plt.ylabel("Average detection delay")
7     plt.title(f"Detection delay comparison - {fault}")
8     plt.show()

```

13. Pivot tables

```

1 arl_pivot = arl_summary.set_index("method_stat")
2 delay_pivot = delay_summary.pivot_table(index="fault_type", columns="method_stat",
3                                         values="mean")
4
5 print("ARL0 pivot:")
6 display(arl_pivot)
7
8 print("\nDetection delay pivot:")
9 display(delay_pivot)

```

14. Example sequential run visualization

```

1  example_seed = 9999
2  example_fault = "patch_anomaly"
3
4  rng = np.random.default_rng(example_seed)
5  X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
6  models = fit_monitoring_models(X_ref)
7
8  def apply_fault(x, fault_type):
9      if fault_type == "global_shift":
10         return inject_global_shift(x)
11     elif fault_type == "patch_anomaly":
12         return inject_patch_anomaly(x)
13     elif fault_type == "shape_change":
14         return inject_shape_change(x)
15     else:
16         return x
17
18  pca_t2_seq, pca_q_seq = [], []
19  tensor_t2_seq, tensor_q_seq = [], []
20
21  for t in range(max_run_length):
22      x = generate_normal_sample(rng)
23      if t >= change_point:
24          x = apply_fault(x, example_fault)
25
26      t2_pca, q_pca = compute_pca_stats(x, models["pca"])
27      pca_t2_seq.append(t2_pca)
28      pca_q_seq.append(q_pca)
29
30      if "tensor" in models:
31          t2_tensor, q_tensor = compute_tensor_stats(x, models["tensor"])
32          tensor_t2_seq.append(t2_tensor)
33          tensor_q_seq.append(q_tensor)
34
35  plt.figure(figsize=(10, 4))
36  plt.plot(pca_t2_seq, label="PCA T2")
37  plt.axhline(models["pca"]["t2_limit"], linestyle='--')
38  plt.axvline(change_point, linestyle=':')
39  plt.title(f"Example sequential run - PCA T2 ({example_fault})")
40  plt.xlabel("Time index")
41  plt.ylabel("Statistic")
42  plt.legend()
43  plt.show()
44
45  plt.figure(figsize=(10, 4))
46  plt.plot(pca_q_seq, label="PCA Q")
47  plt.axhline(models["pca"]["q_limit"], linestyle='--')

```

```

48 plt.axvline(change_point, linestyle=':')
49 plt.title(f"Example sequential run - PCA Q ({example_fault})")
50 plt.xlabel("Time index")
51 plt.ylabel("Statistic")
52 plt.legend()
53 plt.show()
54
55 if "tensor" in models:
56     plt.figure(figsize=(10, 4))
57     plt.plot(tensor_t2_seq, label="Tensor T2")
58     plt.axhline(models["tensor"]["t2_limit"], linestyle='--')
59     plt.axvline(change_point, linestyle=':')
60     plt.title(f"Example sequential run - Tensor T2 ({example_fault})")
61     plt.xlabel("Time index")
62     plt.ylabel("Statistic")
63     plt.legend()
64     plt.show()
65
66     plt.figure(figsize=(10, 4))
67     plt.plot(tensor_q_seq, label="Tensor Q")
68     plt.axhline(models["tensor"]["q_limit"], linestyle='--')
69     plt.axvline(change_point, linestyle=':')
70     plt.title(f"Example sequential run - Tensor Q ({example_fault})")
71     plt.xlabel("Time index")
72     plt.ylabel("Statistic")
73     plt.legend()
74     plt.show()

```

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 **T²** 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.

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

Calibrating Limits to a Common Target ARL0

This notebook 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

1. Imports

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 try:
6     import tensorly as tl
7     from tensorly.decomposition import Tucker
8     TENSORLY_OK = True
9 except Exception as e:
10    TENSORLY_OK = False
11    print("TensorLy is not available:", e)
12
13 from sklearn.decomposition import PCA
```

2. Simulation settings

```
1 N_RUNS = 30
2
3 n_ref = 100
4 max_run_length = 400
5 change_point = 120
6
7 n_vars = 6
8 n_time = 20
9
10 tensor_rank = [5, 3, 3]
11 pca_components = 8
12
13 target_arl0 = 200
14 calibration_sequences = 80
15
16 base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
17 time_axis = np.linspace(0, 2*np.pi, n_time)
18
19 print("Target ARL0:", target_arl0)
20 print("Calibration sequences per method:", calibration_sequences)
```

3. Data generation

```
1 def generate_normal_sample(rng):
2     amp = 1.0 + 0.05 * rng.standard_normal()
3     phase = 0.10 * rng.standard_normal()
4     time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
5     phase)
6     drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
7     interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
8     noise = 0.08 * rng.standard_normal((n_vars, n_time))
9     return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
10     interaction + noise
11
12
13 def inject_global_shift(sample):
14     return 1.25 * sample
15
16
17
18 def inject_patch_anomaly(sample):
19     x = sample.copy()
20     x[3, 7:12] += 0.9
21     return x
22
23
24
25 def inject_shape_change(sample):
26     x = sample.copy()
27     x += 0.35 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
```

```
21     return x
```

4. Model fitting

```
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 fit_models(X_ref):
6      models = {}
7
8      # PCA
9      X_ref_vec = X_ref.reshape(X_ref.shape[0], -1)
10     pca = PCA(n_components=pca_components)
11     scores_ref = pca.fit_transform(X_ref_vec)
12     mu_pca = scores_ref.mean(axis=0)
13     S_pca = np.cov(scores_ref, rowvar=False)
14     S_pca_inv = np.linalg.inv(S_pca + 1e-6 * np.eye(S_pca.shape[0]))
15
16     models["pca"] = {
17         "pca": pca,
18         "mu": mu_pca,
19         "S_inv": S_pca_inv
20     }
21
22     # Tensor
23     if TENSORLY_OK:
24         X_ref_tl = tl.tensor(X_ref, dtype=float)
25         core_ref, factors_ref = tucker(X_ref_tl, rank=tensor_rank)
26         U_sample, U_var, U_time = [np.array(f) for f in factors_ref]
27
28         tensor_scores_ref = np.array([
29             (U_var.T @ X_ref[i] @ U_time).reshape(-1) for i in range(X_ref.shape[0])
30         ])
31         mu_tensor = tensor_scores_ref.mean(axis=0)
32         S_tensor = np.cov(tensor_scores_ref, rowvar=False)
33         S_tensor_inv = np.linalg.inv(S_tensor + 1e-6 * np.eye(S_tensor.shape[0]))
34
35         models["tensor"] = {
36             "U_var": U_var,
37             "U_time": U_time,
38             "mu": mu_tensor,
39             "S_inv": S_tensor_inv
40         }
41
42     return models
```

5. Statistic functions

```

1 def pca_stats(sample, model):
2     x_vec = sample.reshape(1, -1)
3     score = model["pca"].transform(x_vec)[0]
4     t2 = hotelling_t2(score, model["mu"], model["S_inv"])
5     x_hat = model["pca"].inverse_transform(score.reshape(1, -1))[0]
6     q = float(np.sum((x_vec[0] - x_hat)**2))
7     return t2, q
8
9 def tensor_stats(sample, model):
10    U_var = model["U_var"]
11    U_time = model["U_time"]
12    G = U_var.T @ sample @ U_time
13    score = G.reshape(-1)
14    t2 = hotelling_t2(score, model["mu"], model["S_inv"])
15    recon = U_var @ G @ U_time.T
16    q = float(np.sum((sample - recon)**2))
17    return t2, q

```

6. Run-length utilities

```

1 def first_alarm_time(sequence_stats, limit):
2     alarms = np.where(sequence_stats > limit)[0]
3     if len(alarms) == 0:
4         return max_run_length
5     return int(alarms[0] + 1)
6
7 def estimate_arl0_from_limit(generate_stats_fn, limit, n_sequences, rng):
8     run_lengths = []
9     for _ in range(n_sequences):
10        stats = generate_stats_fn(rng)
11        rl = first_alarm_time(stats, limit)
12        run_lengths.append(rl)
13    return float(np.mean(run_lengths)), np.array(run_lengths)
14
15 def calibrate_limit_by_quantile(generate_stats_fn, candidate_quantiles, n_sequences,
16                                rng):
17     best = None
18     results = []
19
20     # Use one pool of calibration sequences for stable comparison
21     precomputed = [generate_stats_fn(rng) for _ in range(n_sequences)]
22
23     for q in candidate_quantiles:
24         # build limit from pooled in-control stats
25         pooled = np.concatenate(precomputed)

```

```

25     limit = np.quantile(pooled, q)
26
27     run_lengths = np.array([first_alarm_time(seq, limit) for seq in precomputed])
28     arl0 = float(np.mean(run_lengths))
29
30     results.append((q, limit, arl0))
31
32     if best is None or abs(arl0 - target_arl0) < abs(best[2] - target_arl0):
33         best = (q, limit, arl0)
34
35     return best, pd.DataFrame(results, columns=["quantile", "limit", "arl0"])

```

7. Build one fitted monitoring system and calibrate limits

```

1  def calibrate_one_system(seed):
2      rng = np.random.default_rng(seed)
3
4      X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
5      models = fit_models(X_ref)
6
7      systems = {}
8
9      # Candidate quantiles for limit calibration
10     candidate_quantiles = np.linspace(0.95, 0.9995, 40)
11
12     # PCA generators
13     def pca_t2_sequence(inner_rng):
14         vals = []
15         for _ in range(max_run_length):
16             x = generate_normal_sample(inner_rng)
17             t2, q = pca_stats(x, models["pca"])
18             vals.append(t2)
19         return np.array(vals)
20
21     def pca_q_sequence(inner_rng):
22         vals = []
23         for _ in range(max_run_length):
24             x = generate_normal_sample(inner_rng)
25             t2, q = pca_stats(x, models["pca"])
26             vals.append(q)
27         return np.array(vals)
28
29     best_t2, df_t2 = calibrate_limit_by_quantile(pca_t2_sequence, candidate_quantiles,
30         calibration_sequences, rng)
31     best_q, df_q = calibrate_limit_by_quantile(pca_q_sequence, candidate_quantiles,
32         calibration_sequences, rng)

```

```

32 systems["pca_T2"] = {"model": models["pca"], "limit": best_t2[1], "calibrated_arl0":
    best_t2[2], "grid": df_t2}
33 systems["pca_Q"] = {"model": models["pca"], "limit": best_q[1], "calibrated_arl0":
    best_q[2], "grid": df_q}
34
35 if "tensor" in models:
36     def tensor_t2_sequence(inner_rng):
37         vals = []
38         for _ in range(max_run_length):
39             x = generate_normal_sample(inner_rng)
40             t2, q = tensor_stats(x, models["tensor"])
41             vals.append(t2)
42         return np.array(vals)
43
44     def tensor_q_sequence(inner_rng):
45         vals = []
46         for _ in range(max_run_length):
47             x = generate_normal_sample(inner_rng)
48             t2, q = tensor_stats(x, models["tensor"])
49             vals.append(q)
50         return np.array(vals)
51
52     best_t2, df_t2 = calibrate_limit_by_quantile(tensor_t2_sequence,
    candidate_quantiles, calibration_sequences, rng)
53     best_q, df_q = calibrate_limit_by_quantile(tensor_q_sequence,
    candidate_quantiles, calibration_sequences, rng)
54
55     systems["tensor_T2"] = {"model": models["tensor"], "limit": best_t2[1], "
    calibrated_arl0": best_t2[2], "grid": df_t2}
56     systems["tensor_Q"] = {"model": models["tensor"], "limit": best_q[1], "
    calibrated_arl0": best_q[2], "grid": df_q}
57
58     return systems

```

8. Example calibration curves

```

1 systems_example = calibrate_one_system(seed=777)
2
3 for key, info in systems_example.items():
4     df = info["grid"]
5     plt.figure(figsize=(8, 4))
6     plt.plot(df["quantile"], df["arl0"], marker='o')
7     plt.axhline(target_arl0, linestyle='--')
8     plt.title(f"Calibration curve - {key}")
9     plt.xlabel("Candidate quantile")
10    plt.ylabel("Estimated ARLO")
11    plt.show()
12

```

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

9. Detection-delay function with calibrated limits

```
1  def detection_delay(sequence_stats, limit, change_point):  
2      post = np.where(sequence_stats[change_point:] > limit)[0]  
3      if len(post) == 0:  
4          return max_run_length - change_point  
5      return int(post[0] + 1)  
6  
7  def simulate_detection_delay_for_fault(seed, fault_type):  
8      rng = np.random.default_rng(seed)  
9      systems = calibrate_one_system(seed + 500)  
10  
11     def apply_fault(x):  
12         if fault_type == "global_shift":  
13             return inject_global_shift(x)  
14         elif fault_type == "patch_anomaly":  
15             return inject_patch_anomaly(x)  
16         elif fault_type == "shape_change":  
17             return inject_shape_change(x)  
18         else:  
19             return x  
20  
21     results = {}  
22  
23     for key, info in systems.items():  
24         vals = []  
25         for t in range(max_run_length):  
26             x = generate_normal_sample(rng)  
27             if t >= change_point:  
28                 x = apply_fault(x)  
29  
30             if key.startswith("pca"):  
31                 t2, q = pca_stats(x, info["model"])  
32             else:  
33                 t2, q = tensor_stats(x, info["model"])  
34  
35             vals.append(t2 if key.endswith("T2") else q)  
36  
37     vals = np.array(vals)  
38     delay = detection_delay(vals, info["limit"], change_point)  
39     results[key] = delay  
40  
41     return systems, results
```

10. Monte Carlo study with calibrated ARL0

```
1 records_arl = []
2 records_delay = []
3
4 fault_types = ["global_shift", "patch_anomaly", "shape_change"]
5
6 for run in range(N_RUNS):
7     systems = calibrate_one_system(seed=10000 + run)
8
9     for key, info in systems.items():
10        records_arl.append({
11            "run": run,
12            "method_stat": key,
13            "calibrated_arl0": info["calibrated_arl0"]
14        })
15
16    for fault in fault_types:
17        _, delay_results = simulate_detection_delay_for_fault(seed=20000 + 100*run,
18            fault_type=fault)
19        for key, delay in delay_results.items():
20            records_delay.append({
21                "run": run,
22                "fault_type": fault,
23                "method_stat": key,
24                "delay": delay
25            })
26
27    arl_df = pd.DataFrame(records_arl)
28    delay_df = pd.DataFrame(records_delay)
29
30    arl_summary = arl_df.groupby("method_stat")["calibrated_arl0"].agg(["mean", "std"]).
31        reset_index()
32    delay_summary = delay_df.groupby(["fault_type", "method_stat"])["delay"].agg(["mean",
33        "std"]).reset_index()
34
35    print("ARL0 after calibration:")
36    display(arl_summary)
37
38    print("\nDetection delay after ARL0 calibration:")
39    display(delay_summary)
```

11. Plot calibrated ARL0

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

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

12. Plot detection delays on equal-ARL0 footing

```
1 for fault in fault_types:
2     sub = delay_summary[delay_summary["fault_type"] == fault]
3
4     plt.figure(figsize=(8, 4))
5     plt.bar(sub["method_stat"], sub["mean"], yerr=sub["std"], capsize=5)
6     plt.ylabel("Average detection delay")
7     plt.title(f"Detection delay after ARL0 calibration - {fault}")
8     plt.show()
```

13. Pivot tables

```
1 arl_pivot = arl_summary.set_index("method_stat")
2 delay_pivot = delay_summary.pivot_table(index="fault_type", columns="method_stat",
3     values="mean")
4
5 print("ARL0 pivot:")
6 display(arl_pivot)
7
8 print("\nDelay pivot:")
9 display(delay_pivot)
```

14. Interpretation

This notebook gives 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.

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

Part IV

Sequential and Adaptive Extensions

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

This notebook 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.

1. Imports

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 try:
6     import tensorly as tl
7     from tensorly.decomposition import Tucker
8     TENSORLY_OK = True
9 except Exception as e:
10    TENSORLY_OK = False
11    print("TensorLy is not available:", e)
```

2. Simulation settings

```
1 n_ref = 120
2 max_run_length = 300
3 change_point = 100
4
5 n_vars = 6
6 n_time = 20
7 tensor_rank = [5, 3, 3]
8
9 base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
10 time_axis = np.linspace(0, 2*np.pi, n_time)
11
12 # EWMA parameters
13 lambda_ewma = 0.2
14
15 # CUSUM parameters
16 k_cusum = 0.5 # reference value in standardized units
17
18 # For quick in-notebook calibration
19 calibration_sequences = 60
20 target_arl0 = 200
21
22 print("EWMA lambda:", lambda_ewma)
23 print("CUSUM k:", k_cusum)
24 print("Target ARL0 for calibration:", target_arl0)
```

3. Data generation

```
1 def generate_normal_sample(rng):
2     amp = 1.0 + 0.05 * rng.standard_normal()
3     phase = 0.10 * rng.standard_normal()
4     time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
5     phase)
6     drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
7     interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
8     noise = 0.08 * rng.standard_normal((n_vars, n_time))
9     return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
10     interaction + noise
11
12
13 def inject_small_global_shift(sample):
14     return 1.10 * sample
15
16
17 def inject_small_patch_shift(sample):
18     x = sample.copy()
19     x[3, 7:12] += 0.4
20     return x
```

```

17
18 def inject_persistent_shape_change(sample):
19     x = sample.copy()
20     x += 0.18 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
21     return x

```

4. Fit a tensor monitoring model

```

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 fit_tensor_model(X_ref):
6     if not TENSORLY_OK:
7         raise RuntimeError("TensorLy is required for this notebook.")
8
9     X_ref_t1 = tl.tensor(X_ref, dtype=float)
10    core_ref, factors_ref = tucker(X_ref_t1, rank=tensor_rank)
11    U_sample, U_var, U_time = [np.array(f) for f in factors_ref]
12
13    score_vectors = np.array([
14        (U_var.T @ X_ref[i] @ U_time).reshape(-1)
15        for i in range(X_ref.shape[0])
16    ])
17
18    mu = score_vectors.mean(axis=0)
19    S = np.cov(score_vectors, rowvar=False)
20    S_inv = np.linalg.inv(S + 1e-6 * np.eye(S.shape[0]))
21
22    t2_ref = np.array([hotelling_t2(s, mu, S_inv) for s in score_vectors])
23
24    q_ref = []
25    for i in range(X_ref.shape[0]):
26        G = U_var.T @ X_ref[i] @ U_time
27        recon = U_var @ G @ U_time.T
28        q_ref.append(np.sum((X_ref[i] - recon)**2))
29    q_ref = np.array(q_ref)
30
31    return {
32        "U_var": U_var,
33        "U_time": U_time,
34        "mu": mu,
35        "S_inv": S_inv,
36        "t2_ref": t2_ref,
37        "q_ref": q_ref
38    }

```

5. Compute one-step tensor T^2 and Q

```
1 def tensor_t2_q(sample, model):
2     U_var = model["U_var"]
3     U_time = model["U_time"]
4
5     G = U_var.T @ sample @ U_time
6     score = G.reshape(-1)
7     t2 = hotelling_t2(score, model["mu"], model["S_inv"])
8
9     recon = U_var @ G @ U_time.T
10    q = float(np.sum((sample - recon)**2))
11
12    return t2, q
```

6. EWMA and CUSUM transforms

```
1 def ewma_stat(x, lam):
2     z = np.zeros_like(x, dtype=float)
3     z[0] = x[0]
4     for t in range(1, len(x)):
5         z[t] = lam * x[t] + (1 - lam) * z[t - 1]
6     return z
7
8 def upper_cusum_stat(x, k):
9     c = np.zeros_like(x, dtype=float)
10    for t in range(1, len(x)):
11        c[t] = max(0, c[t - 1] + x[t] - k)
12    return c
13
14 def standardize_with_reference(x, ref_mean, ref_std):
15    ref_std = max(ref_std, 1e-8)
16    return (x - ref_mean) / ref_std
```

7. Build one-step and sequential statistics for a run

```
1 def tensor_run_statistics(sequence, model):
2     t2_vals = []
3     q_vals = []
4     for x in sequence:
5         t2, q = tensor_t2_q(x, model)
6         t2_vals.append(t2)
7         q_vals.append(q)
8
9     t2_vals = np.array(t2_vals)
```

```

10     q_vals = np.array(q_vals)
11
12     # Standardize using reference one-step distributions
13     t2_std = standardize_with_reference(t2_vals, model["t2_ref"].mean(), model["t2_ref"]
14     ".std(ddof=1))
15     q_std = standardize_with_reference(q_vals, model["q_ref"].mean(), model["q_ref"].
16     std(ddof=1))
17
18     ewma_t2 = ewma_stat(t2_std, lambda_ewma)
19     ewma_q = ewma_stat(q_std, lambda_ewma)
20
21     cusum_t2 = upper_cusum_stat(t2_std, k_cusum)
22     cusum_q = upper_cusum_stat(q_std, k_cusum)
23
24     return {
25         "T2": t2_vals,
26         "Q": q_vals,
27         "T2_std": t2_std,
28         "Q_std": q_std,
29         "EWMA_T2": ewma_t2,
30         "EWMA_Q": ewma_q,
31         "CUSUM_T2": cusum_t2,
32         "CUSUM_Q": cusum_q
33     }

```

8. Generate in-control sequences for limit calibration

```

1 def generate_in_control_sequence(rng, length):
2     return np.array([generate_normal_sample(rng) for _ in range(length)])
3
4 def first_alarm_time(sequence_stats, limit):
5     alarms = np.where(sequence_stats > limit)[0]
6     if len(alarms) == 0:
7         return max_run_length
8     return int(alarms[0] + 1)
9
10 def calibrate_limit_from_sequences(stats_list, target_arl0, candidate_quantiles):
11     pooled = np.concatenate(stats_list)
12     best = None
13     rows = []
14
15     for q in candidate_quantiles:
16         limit = np.quantile(pooled, q)
17         run_lengths = np.array([first_alarm_time(stats, limit) for stats in stats_list
18         ])
19         arl0 = float(run_lengths.mean())
20         rows.append((q, limit, arl0))
21         if best is None or abs(arl0 - target_arl0) < abs(best[2] - target_arl0):

```

```

21         best = (q, limit, arl0)
22
23     return best, pd.DataFrame(rows, columns=["quantile", "limit", "arl0"])

```

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

```

1  rng = np.random.default_rng(12345)
2
3  X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
4  tensor_model = fit_tensor_model(X_ref)
5
6  stat_names = ["T2", "Q", "EWMA_T2", "EWMA_Q", "CUSUM_T2", "CUSUM_Q"]
7  candidate_quantiles = np.linspace(0.95, 0.9995, 40)
8
9  stats_collection = {name: [] for name in stat_names}
10 for _ in range(calibration_sequences):
11     seq = generate_in_control_sequence(rng, max_run_length)
12     out = tensor_run_statistics(seq, tensor_model)
13     for name in stat_names:
14         stats_collection[name].append(out[name])
15
16 limits = {}
17 calibration_tables = {}
18
19 for name in stat_names:
20     best, df = calibrate_limit_from_sequences(stats_collection[name], target_arl0,
21     candidate_quantiles)
22     limits[name] = best[1]
23     calibration_tables[name] = df
24
25 limit_summary = pd.DataFrame({
26     "statistic": list(limits.keys()),
27     "limit": list(limits.values()),
28     "approx_arl0": [calibrate_limit_from_sequences(stats_collection[name], target_arl0,
29     candidate_quantiles)[0][2]
30     for name in stat_names]
31 })
32 limit_summary

```

10. Plot calibration curves

```

1  for name in stat_names:
2     df = calibration_tables[name]

```

```

3     plt.figure(figsize=(8, 4))
4     plt.plot(df["quantile"], df["ar10"], marker='o')
5     plt.axhline(target_ar10, linestyle='--')
6     plt.title(f"Calibration curve - {name}")
7     plt.xlabel("Candidate quantile")
8     plt.ylabel("Estimated ARLO")
9     plt.show()

```

11. Example sequential run with a small shift

```

1  def generate_out_of_control_sequence(rng, length, change_point, fault_type):
2      seq = []
3      for t in range(length):
4          x = generate_normal_sample(rng)
5          if t >= change_point:
6              if fault_type == "small_global_shift":
7                  x = inject_small_global_shift(x)
8              elif fault_type == "small_patch_shift":
9                  x = inject_small_patch_shift(x)
10             elif fault_type == "persistent_shape_change":
11                 x = inject_persistent_shape_change(x)
12             seq.append(x)
13     return np.array(seq)
14
15  rng = np.random.default_rng(54321)
16  seq = generate_out_of_control_sequence(rng, max_run_length, change_point, "
17     small_patch_shift")
18  run_out = tensor_run_statistics(seq, tensor_model)

```

12. Plot one-step vs EWMA vs CUSUM for T^2

```

1  plt.figure(figsize=(10, 4))
2  plt.plot(run_out["T2_std"], label="Standardized  $T^2$ ")
3  plt.axhline(standardize_with_reference(limits["T2"], tensor_model["t2_ref"].mean(),
4     tensor_model["t2_ref"].std(ddof=1)), linestyle='--')
5  plt.axvline(change_point, linestyle=':')
6  plt.title("One-step Tensor  $T^2$ ")
7  plt.xlabel("Time index")
8  plt.ylabel("Statistic")
9  plt.legend()
10 plt.show()
11
12 plt.figure(figsize=(10, 4))
13 plt.plot(run_out["EWMA_T2"], label="EWMA- $T^2$ ")
14 plt.axhline(limits["EWMA_T2"], linestyle='--')

```

```

14 plt.axvline(change_point, linestyle=':')
15 plt.title("EWMA-T2")
16 plt.xlabel("Time index")
17 plt.ylabel("Statistic")
18 plt.legend()
19 plt.show()
20
21 plt.figure(figsize=(10, 4))
22 plt.plot(run_out["CUSUM_T2"], label="CUSUM-T2")
23 plt.axhline(limits["CUSUM_T2"], linestyle='--')
24 plt.axvline(change_point, linestyle=':')
25 plt.title("CUSUM-T2")
26 plt.xlabel("Time index")
27 plt.ylabel("Statistic")
28 plt.legend()
29 plt.show()

```

13. Plot one-step vs EWMA vs CUSUM for Q

```

1 plt.figure(figsize=(10, 4))
2 plt.plot(run_out["Q_std"], label="Standardized Q")
3 plt.axhline(standardize_with_reference(limits["Q"], tensor_model["q_ref"].mean(),
4     tensor_model["q_ref"].std(ddof=1)), linestyle='--')
4 plt.axvline(change_point, linestyle=':')
5 plt.title("One-step Tensor Q")
6 plt.xlabel("Time index")
7 plt.ylabel("Statistic")
8 plt.legend()
9 plt.show()
10
11 plt.figure(figsize=(10, 4))
12 plt.plot(run_out["EWMA_Q"], label="EWMA-Q")
13 plt.axhline(limits["EWMA_Q"], linestyle='--')
14 plt.axvline(change_point, linestyle=':')
15 plt.title("EWMA-Q")
16 plt.xlabel("Time index")
17 plt.ylabel("Statistic")
18 plt.legend()
19 plt.show()
20
21 plt.figure(figsize=(10, 4))
22 plt.plot(run_out["CUSUM_Q"], label="CUSUM-Q")
23 plt.axhline(limits["CUSUM_Q"], linestyle='--')
24 plt.axvline(change_point, linestyle=':')
25 plt.title("CUSUM-Q")
26 plt.xlabel("Time index")
27 plt.ylabel("Statistic")
28 plt.legend()

```

```
29 plt.show()
```

14. Compare detection delay across one-step, EWMA, and CUSUM

```

1 def detection_delay(sequence_stats, limit, change_point):
2     post = np.where(sequence_stats[change_point:] > limit)[0]
3     if len(post) == 0:
4         return max_run_length - change_point
5     return int(post[0] + 1)
6
7 fault_types = ["small_global_shift", "small_patch_shift", "persistent_shape_change"]
8 N_RUNS = 30
9
10 records = []
11 for fault in fault_types:
12     rng = np.random.default_rng(202600 + hash(fault) % 1000)
13     for run in range(N_RUNS):
14         seq = generate_out_of_control_sequence(rng, max_run_length, change_point,
15         fault)
16         out = tensor_run_statistics(seq, tensor_model)
17
18         for name in stat_names:
19             delay = detection_delay(out[name], limits[name], change_point)
20             records.append({
21                 "fault_type": fault,
22                 "run": run,
23                 "statistic": name,
24                 "delay": delay
25             })
26
27 delay_df = pd.DataFrame(records)
28 delay_summary = delay_df.groupby(["fault_type", "statistic"])["delay"].agg(["mean", "
    std"]).reset_index()
29 delay_summary

```

15. Plot detection delay by statistic

```

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

```

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.

17. Notes for research use

Advancing this work to a paper-quality study involves the following steps:

- 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

Matched Tensor vs PCA EWMA/CUSUM Comparison

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

For both approaches, we compare:

- one-step T^2
- one-step Q
- EWMA- T^2
- EWMA- Q
- CUSUM- T^2
- CUSUM- Q

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

1. Imports

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 try:
6     import tensorly as tl
7     from tensorly.decomposition import Tucker
8     TENSORLY_OK = True
9 except Exception as e:
10    TENSORLY_OK = False
11    print("TensorLy is not available:", e)
12
13 from sklearn.decomposition import PCA
```

2. Settings

```
1 n_ref = 120
2 max_run_length = 300
3 change_point = 100
4
5 n_vars = 6
6 n_time = 20
7
8 tensor_rank = [5, 3, 3]
9 pca_components = 8
10
11 lambda_ewma = 0.2
12 k_cusum = 0.5
13
14 target_arl0 = 200
15 calibration_sequences = 60
16 N_RUNS = 30
17
18 base_var = np.array([1.0, 0.9, 1.15, 0.95, 1.1, 0.85])
19 time_axis = np.linspace(0, 2*np.pi, n_time)
20
21 print("Tensor rank:", tensor_rank)
22 print("PCA components:", pca_components)
23 print("EWMA lambda:", lambda_ewma)
24 print("CUSUM k:", k_cusum)
```

3. Data generation

```
1 def generate_normal_sample(rng):
2     amp = 1.0 + 0.05 * rng.standard_normal()
3     phase = 0.10 * rng.standard_normal()
4     time_profile = np.sin(time_axis + phase) + 0.35 * np.cos(2 * time_axis + 0.5 *
5     phase)
6     drift = 0.03 * rng.standard_normal() * np.linspace(-1, 1, n_time)
7     interaction = 0.15 * np.outer(np.linspace(-1, 1, n_vars), np.sin(0.5 * time_axis))
8     noise = 0.08 * rng.standard_normal((n_vars, n_time))
9     return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) +
10     interaction + noise
11
12
13 def inject_small_global_shift(sample):
14     return 1.10 * sample
15
16
17 def inject_small_patch_shift(sample):
18     x = sample.copy()
19     x[3, 7:12] += 0.4
20     return x
```

```

17
18 def inject_persistent_shape_change(sample):
19     x = sample.copy()
20     x += 0.18 * np.outer(np.array([1, -1, 1, -1, 1, -1]), np.cos(3 * time_axis))
21     return x

```

4. Shared 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 standardize_with_reference(x, ref_mean, ref_std):
19    ref_std = max(ref_std, 1e-8)
20    return (x - ref_mean) / ref_std
21
22 def first_alarm_time(sequence_stats, limit):
23    alarms = np.where(sequence_stats > limit)[0]
24    if len(alarms) == 0:
25        return max_run_length
26    return int(alarms[0] + 1)
27
28 def detection_delay(sequence_stats, limit, change_point):
29    post = np.where(sequence_stats[change_point:] > limit)[0]
30    if len(post) == 0:
31        return max_run_length - change_point
32    return int(post[0] + 1)
33
34 def calibrate_limit_from_sequences(stats_list, target_arl0, candidate_quantiles):
35    pooled = np.concatenate(stats_list)
36    best = None
37    rows = []
38
39    for q in candidate_quantiles:
40        limit = np.quantile(pooled, q)

```

```

41     run_lengths = np.array([first_alarm_time(stats, limit) for stats in stats_list
42                             ])
43     arl0 = float(run_lengths.mean())
44     rows.append((q, limit, arl0))
45     if best is None or abs(arl0 - target_arl0) < abs(best[2] - target_arl0):
46         best = (q, limit, arl0)
47     return best, pd.DataFrame(rows, columns=["quantile", "limit", "arl0"])

```

5. Fit tensor and PCA models

```

1  def fit_tensor_model(X_ref):
2      if not TENSORLY_OK:
3          raise RuntimeError("TensorLy is required for tensor sections.")
4
5      X_ref_t1 = tl.tensor(X_ref, dtype=float)
6      core_ref, factors_ref = tucker(X_ref_t1, rank=tensor_rank)
7      U_sample, U_var, U_time = [np.array(f) for f in factors_ref]
8
9      score_vectors = np.array([
10         (U_var.T @ X_ref[i] @ U_time).reshape(-1)
11         for i in range(X_ref.shape[0])
12     ])
13
14     mu = score_vectors.mean(axis=0)
15     S = np.cov(score_vectors, rowvar=False)
16     S_inv = np.linalg.inv(S + 1e-6 * np.eye(S.shape[0]))
17
18     t2_ref = np.array([hotelling_t2(s, mu, S_inv) for s in score_vectors])
19
20     q_ref = []
21     for i in range(X_ref.shape[0]):
22         G = U_var.T @ X_ref[i] @ U_time
23         recon = U_var @ G @ U_time.T
24         q_ref.append(np.sum((X_ref[i] - recon)**2))
25     q_ref = np.array(q_ref)
26
27     return {
28         "U_var": U_var,
29         "U_time": U_time,
30         "mu": mu,
31         "S_inv": S_inv,
32         "t2_ref": t2_ref,
33         "q_ref": q_ref
34     }
35
36 def fit_pca_model(X_ref):
37     X_ref_vec = X_ref.reshape(X_ref.shape[0], -1)

```

```

38     pca = PCA(n_components=pca_components)
39     scores = pca.fit_transform(X_ref_vec)
40
41     mu = scores.mean(axis=0)
42     S = np.cov(scores, rowvar=False)
43     S_inv = np.linalg.inv(S + 1e-6 * np.eye(S.shape[0]))
44
45     t2_ref = np.array([hotelling_t2(s, mu, S_inv) for s in scores])
46     X_hat = pca.inverse_transform(scores)
47     q_ref = np.sum((X_ref_vec - X_hat)**2, axis=1)
48
49     return {
50         "pca": pca,
51         "mu": mu,
52         "S_inv": S_inv,
53         "t2_ref": t2_ref,
54         "q_ref": q_ref
55     }

```

6. One-step T^2 and Q for tensor and PCA

```

1  def tensor_t2_q(sample, model):
2      U_var = model["U_var"]
3      U_time = model["U_time"]
4      G = U_var.T @ sample @ U_time
5      score = G.reshape(-1)
6      t2 = hotelling_t2(score, model["mu"], model["S_inv"])
7      recon = U_var @ G @ U_time.T
8      q = float(np.sum((sample - recon)**2))
9      return t2, q
10
11 def pca_t2_q(sample, model):
12     x_vec = sample.reshape(1, -1)
13     score = model["pca"].transform(x_vec)[0]
14     t2 = hotelling_t2(score, model["mu"], model["S_inv"])
15     x_hat = model["pca"].inverse_transform(score.reshape(1, -1))[0]
16     q = float(np.sum((x_vec[0] - x_hat)**2))
17     return t2, q

```

7. Build one-step, EWMA, and CUSUM sequences

```

1  def run_stat_sequences(sequence, model, method):
2      t2_vals = []
3      q_vals = []
4

```

```

5     for x in sequence:
6         if method == "tensor":
7             t2, q = tensor_t2_q(x, model)
8         elif method == "pca":
9             t2, q = pca_t2_q(x, model)
10        else:
11            raise ValueError("Unknown method")
12
13        t2_vals.append(t2)
14        q_vals.append(q)
15
16    t2_vals = np.array(t2_vals)
17    q_vals = np.array(q_vals)
18
19    t2_std = standardize_with_reference(t2_vals, model["t2_ref"].mean(), model["t2_ref"]
20    ".std(ddof=1))
21    q_std = standardize_with_reference(q_vals, model["q_ref"].mean(), model["q_ref"].
22    std(ddof=1))
23
24    return {
25        "T2": t2_vals,
26        "Q": q_vals,
27        "T2_std": t2_std,
28        "Q_std": q_std,
29        "EWMA_T2": ewma_stat(t2_std, lambda_ewma),
30        "EWMA_Q": ewma_stat(q_std, lambda_ewma),
31        "CUSUM_T2": upper_cusum_stat(t2_std, k_cusum),
32        "CUSUM_Q": upper_cusum_stat(q_std, k_cusum),
33    }

```

8. Fit both models and calibrate all sequential limits

```

1  rng = np.random.default_rng(202612)
2  X_ref = np.array([generate_normal_sample(rng) for _ in range(n_ref)])
3
4  tensor_model = fit_tensor_model(X_ref) if TENSORLY_OK else None
5  pca_model = fit_pca_model(X_ref)
6
7  stat_names = ["T2", "Q", "EWMA_T2", "EWMA_Q", "CUSUM_T2", "CUSUM_Q"]
8  candidate_quantiles = np.linspace(0.95, 0.9995, 40)
9
10 def generate_in_control_sequence(rng, length):
11     return np.array([generate_normal_sample(rng) for _ in range(length)])
12
13 limits = {}
14 calibration_tables = {}
15
16 for method_name, model in [("tensor", tensor_model), ("pca", pca_model)]:

```

```

17     if model is None:
18         continue
19
20     stats_collection = {name: [] for name in stat_names}
21     for _ in range(calibration_sequences):
22         seq = generate_in_control_sequence(rng, max_run_length)
23         out = run_stat_sequences(seq, model, method_name)
24         for name in stat_names:
25             stats_collection[name].append(out[name])
26
27     for name in stat_names:
28         best, df = calibrate_limit_from_sequences(stats_collection[name], target_arl0,
29         candidate_quantiles)
29         limits[f"{method_name}_{name}"] = {
30             "limit": best[1],
31             "approx_arl0": best[2]
32         }
33         calibration_tables[f"{method_name}_{name}"] = df
34
35     limit_summary = pd.DataFrame([
36         {"method_stat": k, "limit": v["limit"], "approx_arl0": v["approx_arl0"]}
37         for k, v in limits.items()
38     ])
39
40     limit_summary

```

9. Plot selected calibration curves

```

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

```

10. Example sequence comparison

```

1     def generate_out_of_control_sequence(rng, length, change_point, fault_type):
2         seq = []
3         for t in range(length):
4             x = generate_normal_sample(rng)
5             if t >= change_point:

```

```

6         if fault_type == "small_global_shift":
7             x = inject_small_global_shift(x)
8         elif fault_type == "small_patch_shift":
9             x = inject_small_patch_shift(x)
10        elif fault_type == "persistent_shape_change":
11            x = inject_persistent_shape_change(x)
12        seq.append(x)
13    return np.array(seq)
14
15    rng = np.random.default_rng(9090)
16    seq = generate_out_of_control_sequence(rng, max_run_length, change_point, "
17        small_global_shift")
18    tensor_run = run_stat_sequences(seq, tensor_model, "tensor") if tensor_model is not
19        None else None
20    pca_run = run_stat_sequences(seq, pca_model, "pca")

```

11. Plot EWMA-T² and EWMA-Q: tensor vs PCA

```

1    if tensor_run is not None:
2        plt.figure(figsize=(10, 4))
3        plt.plot(tensor_run["EWMA_T2"], label="Tensor EWMA-T2")
4        plt.plot(pca_run["EWMA_T2"], label="PCA EWMA-T2")
5        plt.axhline(limits["tensor_EWMA_T2"]["limit"], linestyle='--')
6        plt.axhline(limits["pca_EWMA_T2"]["limit"], linestyle=':')
7        plt.axvline(change_point, linestyle='-.')
8        plt.title("EWMA-T2 comparison")
9        plt.xlabel("Time index")
10       plt.ylabel("Statistic")
11       plt.legend()
12       plt.show()
13
14       plt.figure(figsize=(10, 4))
15       plt.plot(tensor_run["EWMA_Q"], label="Tensor EWMA-Q")
16       plt.plot(pca_run["EWMA_Q"], label="PCA EWMA-Q")
17       plt.axhline(limits["tensor_EWMA_Q"]["limit"], linestyle='--')
18       plt.axhline(limits["pca_EWMA_Q"]["limit"], linestyle=':')
19       plt.axvline(change_point, linestyle='-.')
20       plt.title("EWMA-Q comparison")
21       plt.xlabel("Time index")
22       plt.ylabel("Statistic")
23       plt.legend()
24       plt.show()

```

12. Plot CUSUM-T² and CUSUM-Q: tensor vs PCA

```

1  if tensor_run is not None:
2      plt.figure(figsize=(10, 4))
3      plt.plot(tensor_run["CUSUM_T2"], label="Tensor CUSUM-T2")
4      plt.plot(pca_run["CUSUM_T2"], label="PCA CUSUM-T2")
5      plt.axhline(limits["tensor_CUSUM_T2"]["limit"], linestyle='--')
6      plt.axhline(limits["pca_CUSUM_T2"]["limit"], linestyle=':')
7      plt.axvline(change_point, linestyle='-.')
8      plt.title("CUSUM-T2 comparison")
9      plt.xlabel("Time index")
10     plt.ylabel("Statistic")
11     plt.legend()
12     plt.show()
13
14     plt.figure(figsize=(10, 4))
15     plt.plot(tensor_run["CUSUM_Q"], label="Tensor CUSUM-Q")
16     plt.plot(pca_run["CUSUM_Q"], label="PCA CUSUM-Q")
17     plt.axhline(limits["tensor_CUSUM_Q"]["limit"], linestyle='--')
18     plt.axhline(limits["pca_CUSUM_Q"]["limit"], linestyle=':')
19     plt.axvline(change_point, linestyle='-.')
20     plt.title("CUSUM-Q comparison")
21     plt.xlabel("Time index")
22     plt.ylabel("Statistic")
23     plt.legend()
24     plt.show()

```

13. Detection-delay comparison across all sequential statistics

```

1  fault_types = ["small_global_shift", "small_patch_shift", "persistent_shape_change"]
2  records = []
3
4  for fault in fault_types:
5      for run in range(N_RUNS):
6          rng = np.random.default_rng(120000 + run + 1000 * fault_types.index(fault))
7          seq = generate_out_of_control_sequence(rng, max_run_length, change_point,
8          fault)
9
10         if tensor_model is not None:
11             tensor_out = run_stat_sequences(seq, tensor_model, "tensor")
12             for name in stat_names:
13                 delay = detection_delay(tensor_out[name], limits[f"tensor_{name}"]["
14                 limit"], change_point)
15                 records.append({
16                     "method": "tensor",
17                     "fault_type": fault,
18                     "statistic": name,
19                     "run": run,
20                     "delay": delay

```

```

19         })
20
21     pca_out = run_stat_sequences(seq, pca_model, "pca")
22     for name in stat_names:
23         delay = detection_delay(pca_out[name], limits[f"pca_{name}"]["limit"],
24                                change_point)
25         records.append({
26             "method": "pca",
27             "fault_type": fault,
28             "statistic": name,
29             "run": run,
30             "delay": delay
31         })
32
33 delay_df = pd.DataFrame(records)
34 delay_summary = delay_df.groupby(["method", "fault_type", "statistic"])["delay"].agg([
35     "mean", "std"]).reset_index()
36 delay_summary.head()

```

14. Plot delay by fault type

```

1 for fault in fault_types:
2     sub = delay_summary[delay_summary["fault_type"] == fault].copy()
3     sub["label"] = sub["method"] + "_" + sub["statistic"]
4
5     plt.figure(figsize=(12, 4))
6     plt.bar(sub["label"], sub["mean"], yerr=sub["std"], capsize=4)
7     plt.ylabel("Average detection delay")
8     plt.title(f"Sequential detection delay comparison - {fault}")
9     plt.xticks(rotation=45)
10    plt.show()

```

15. Pivot tables

```

1 arl_pivot = limit_summary.set_index("method_stat")["approx_arl0"]
2 delay_pivot = delay_summary.pivot_table(index="fault_type", columns=["method", "
3     statistic"], values="mean")
4
5 print("Approximate ARL0 after calibration:")
6 display(arl_pivot)
7
8 print("\nAverage detection delay:")
9 display(delay_pivot)

```

16. Interpretation

This notebook gives the **matched sequential baseline** that the earlier tensor-only EWMA/CUSUM notebook 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.

17. Strong next extensions

Good next steps 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 notebook
- add tables that rank the best method-statistic combination by fault type

Left Intentionally Blank For Notes

Part V
Curated Results

Curated Results Summary

Curated Results Notebook — Tensor-Structured SPC

This notebook is a **curated results version** of the larger tensor-SPC notebook sequence.

It is designed to be more clean than a raw notebook 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

1. Imports and plotting defaults

```
In [1]: 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)
```

2. Simulation settings

```
In [2]: 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)
```

3. Data-generation helpers

```
In [3]: 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 *
    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_axi
    noise = 0.08 * rng.standard_normal((n_vars, n_time))
    return amp * np.outer(base_var, time_profile) + np.outer(base_var, drift) + int

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)

def inject_small_patch_shift(sample):
    return inject_patch_anomaly(sample, mag=0.4)

def inject_persistent_shape_change(sample):
    return inject_shape_change(sample, mag=0.18)
```

4. Monitoring helpers

```
In [4]: 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 first_alarm_time(sequence_stats, limit, max_run_length):
    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, max_run_length):
    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 standardize_with_reference(x, ref_mean, ref_std):
    ref_std = max(ref_std, 1e-8)
    return (x - ref_mean) / ref_std

```

5. Fit tensor and PCA models

```

In [5]: 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_t1 = tl.tensor(X_ref, dtype=float)
    core_ref, factors_ref = tucker(X_ref_t1, 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)

```

Tensor model available: True

6. One-step T^2 and Q functions

```

In [6]: 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

```

7. Curated result 1 — representative tensor T^2 and Q example

```
In [7]: 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 = []

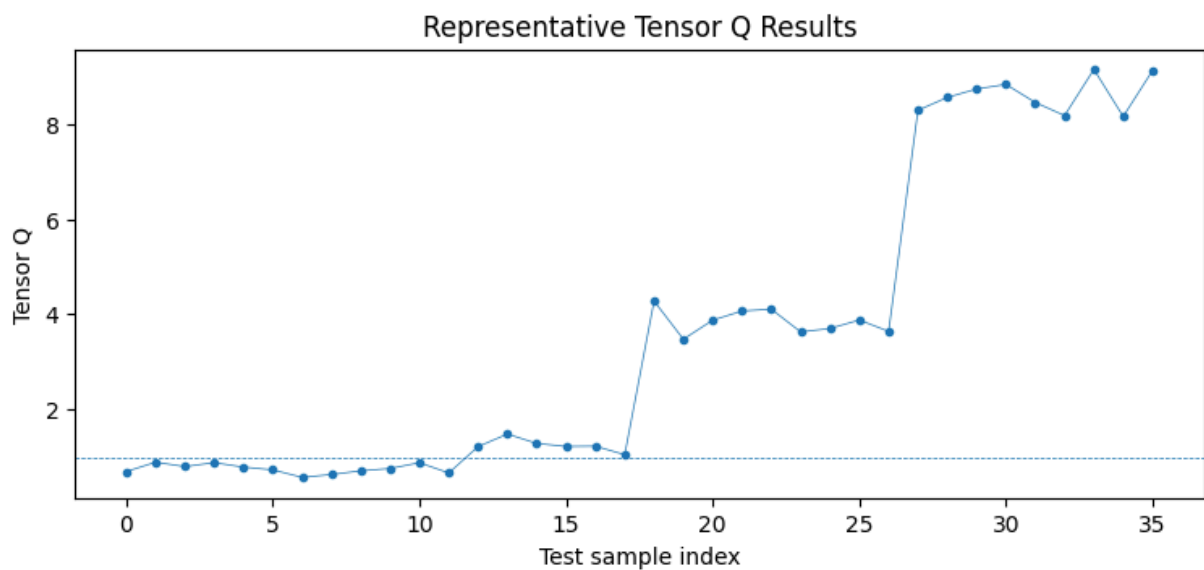
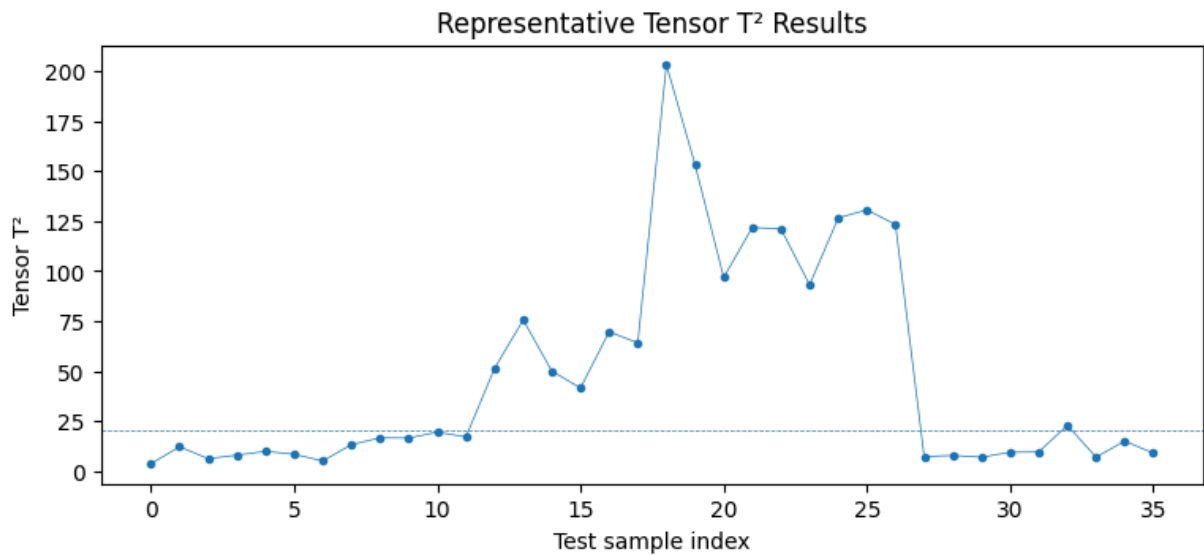
for i in range(n_test):
    t2, q = tensor_t2_q(X_test[i], tensor_model)
    tensor_t2.append(t2)
    tensor_q.append(q)

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 T2 Results")
ax.set_xlabel("Test sample index")
ax.set_ylabel("Tensor T2")
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()
```



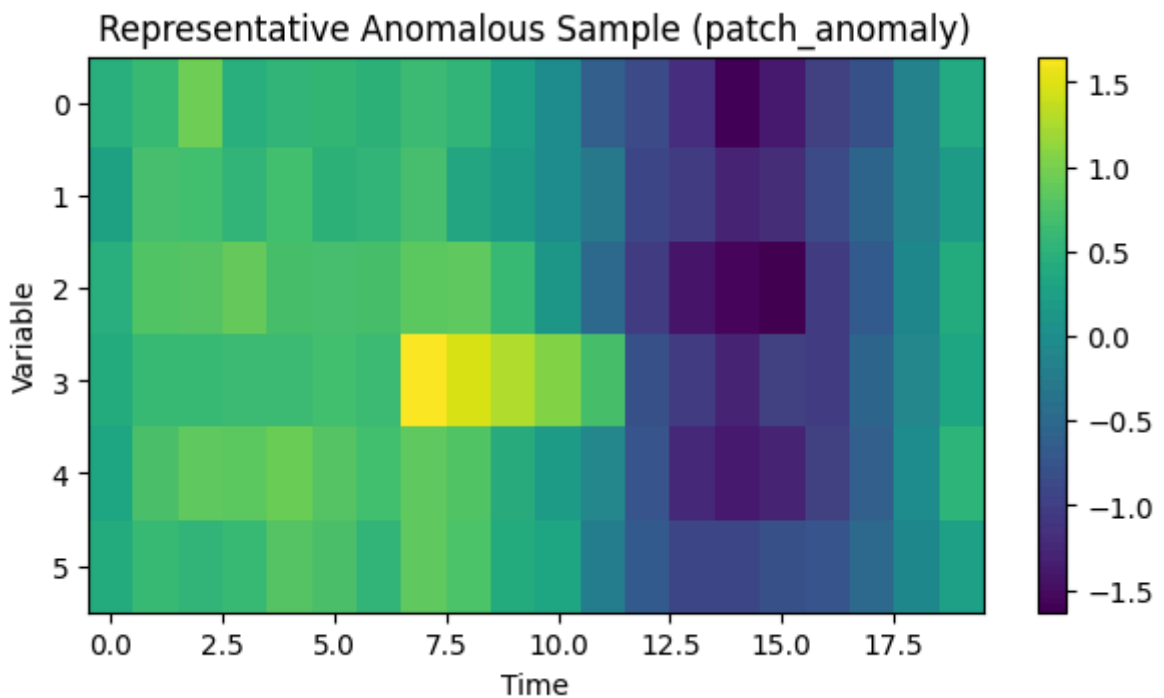
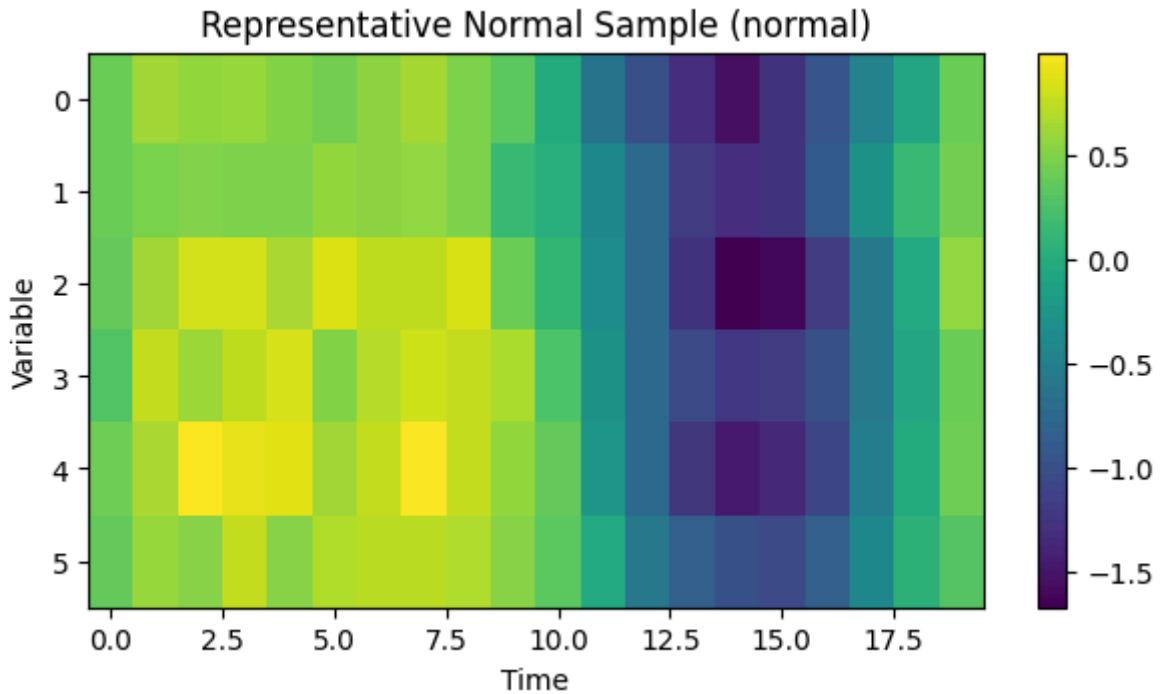
8. Curated result 2 — normal vs anomaly visualization

```
In [8]: 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()
```



9. Curated result 3 — tensor vs PCA one-step comparison

```
In [9]: 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)

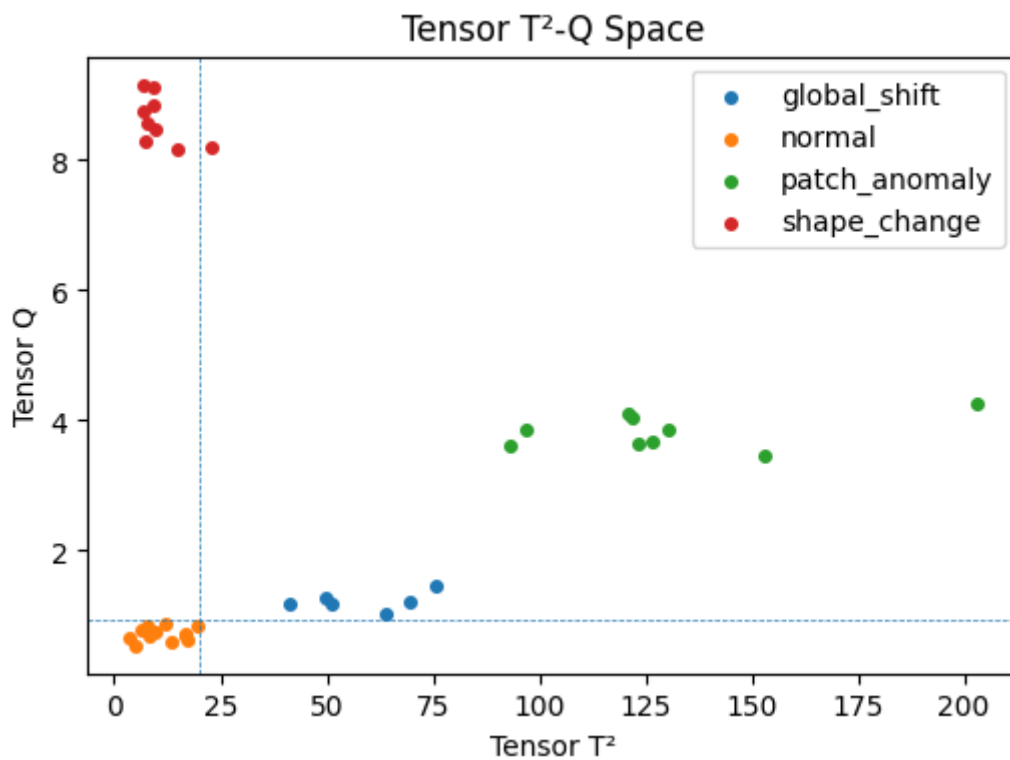
pca_t2 = np.array(pca_t2)
pca_q = np.array(pca_q)

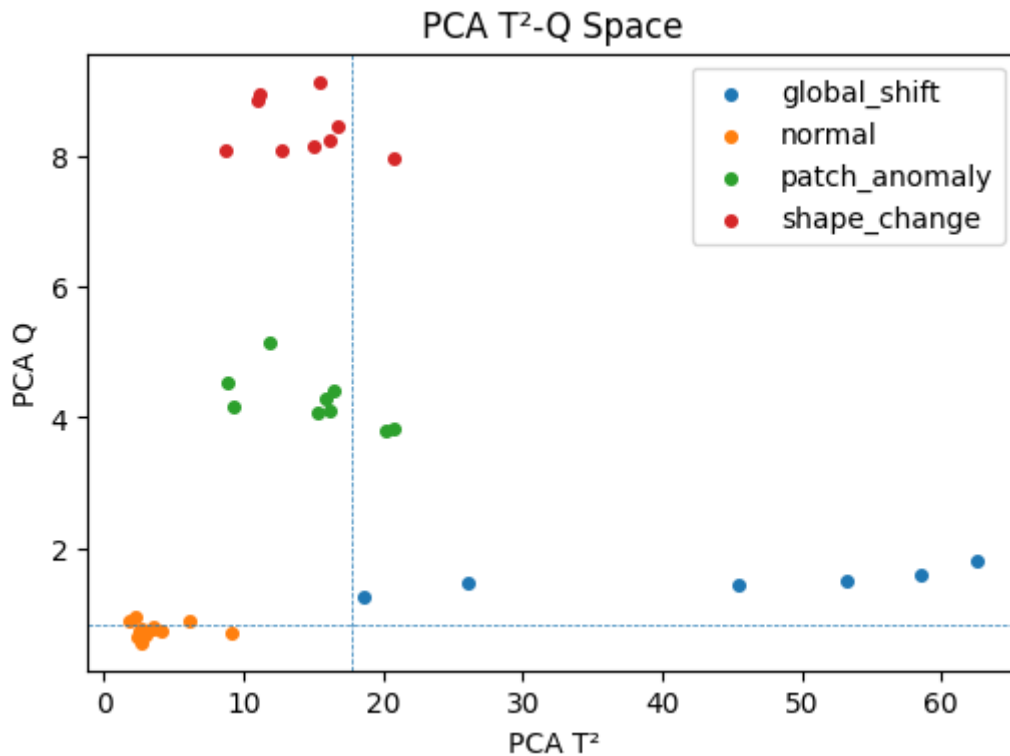
pca_t2_limit = np.percentile(pca_model["t2_ref"], 99)
pca_q_limit = np.percentile(pca_model["q_ref"], 99)

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 T2-Q Space")
ax.set_xlabel("Tensor T2")
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 T2-Q Space")
ax.set_xlabel("PCA T2")
ax.set_ylabel("PCA Q")
ax.legend()
plt.show()

```





10. Curated result 4 — Monte Carlo detection summary

```
In [10]: 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

```

Out[10]:

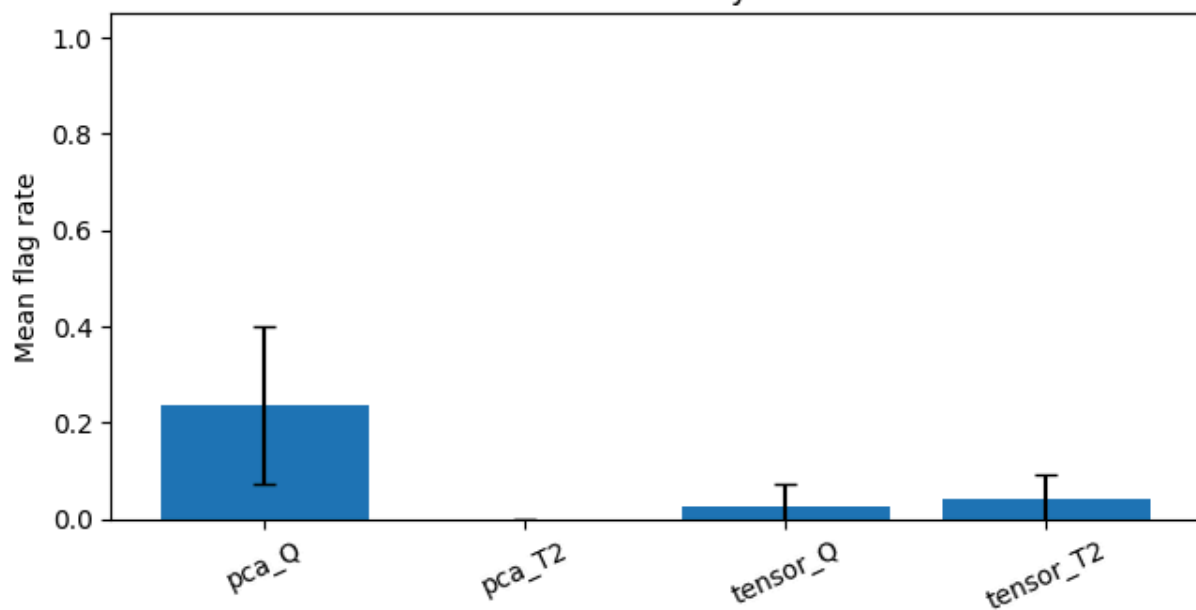
| | method | fault_type | mean | std |
|----|-----------|---------------|----------|----------|
| 0 | pca_Q | global_shift | 1.000000 | 0.000000 |
| 1 | pca_Q | normal | 0.237500 | 0.162840 |
| 2 | pca_Q | patch_anomaly | 1.000000 | 0.000000 |
| 3 | pca_Q | shape_change | 1.000000 | 0.000000 |
| 4 | pca_T2 | global_shift | 0.941667 | 0.081560 |
| 5 | pca_T2 | normal | 0.000000 | 0.000000 |
| 6 | pca_T2 | patch_anomaly | 0.094444 | 0.136202 |
| 7 | pca_T2 | shape_change | 0.577778 | 0.421329 |
| 8 | tensor_Q | global_shift | 0.950000 | 0.078360 |
| 9 | tensor_Q | normal | 0.025000 | 0.047603 |
| 10 | tensor_Q | patch_anomaly | 1.000000 | 0.000000 |
| 11 | tensor_Q | shape_change | 1.000000 | 0.000000 |
| 12 | tensor_T2 | global_shift | 0.991667 | 0.037268 |
| 13 | tensor_T2 | normal | 0.041667 | 0.050581 |
| 14 | tensor_T2 | patch_anomaly | 1.000000 | 0.000000 |
| 15 | tensor_T2 | shape_change | 0.133333 | 0.159597 |

11. Monte Carlo detection plot

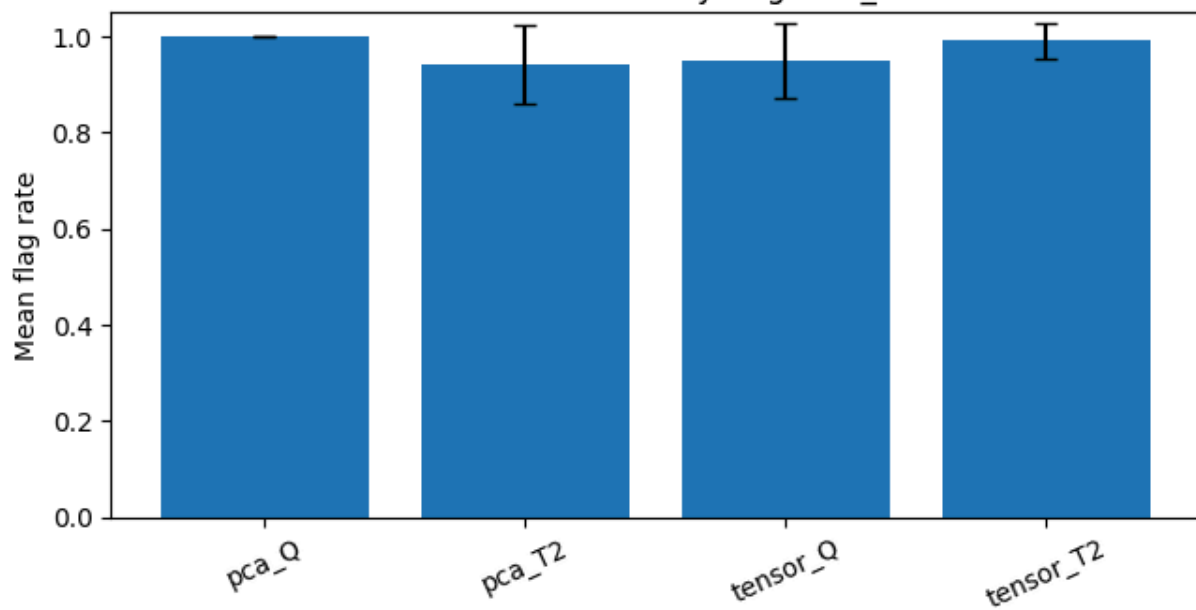
```
In [11]: fault_order = ["normal", "global_shift", "patch_anomaly", "shape_change"]

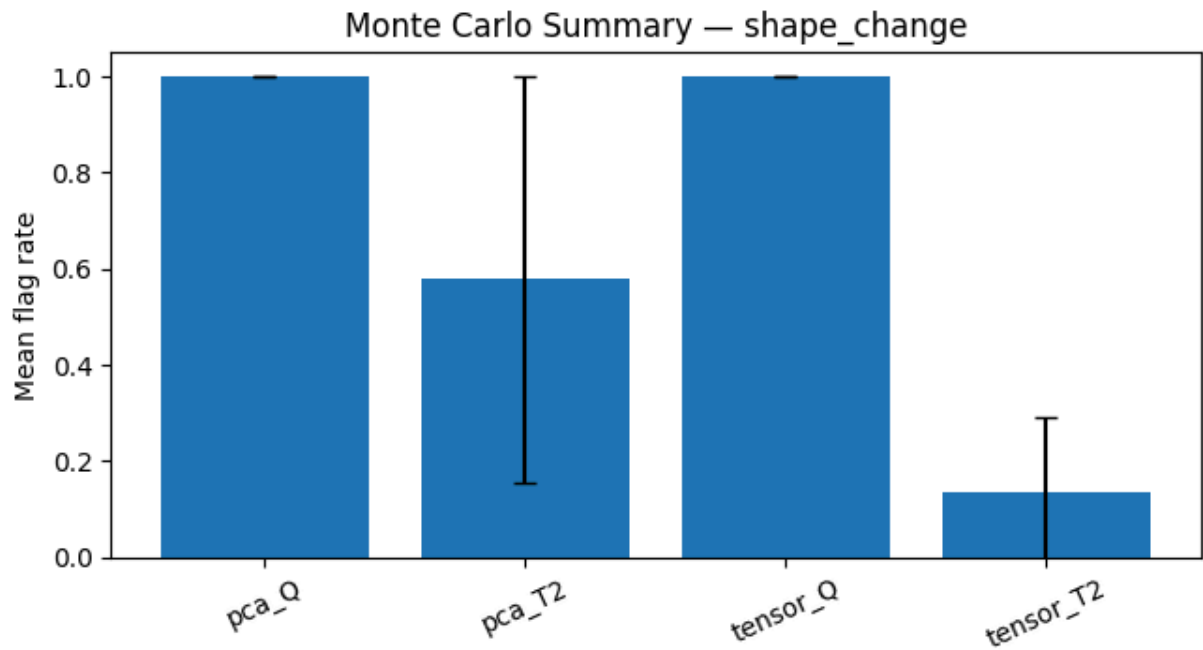
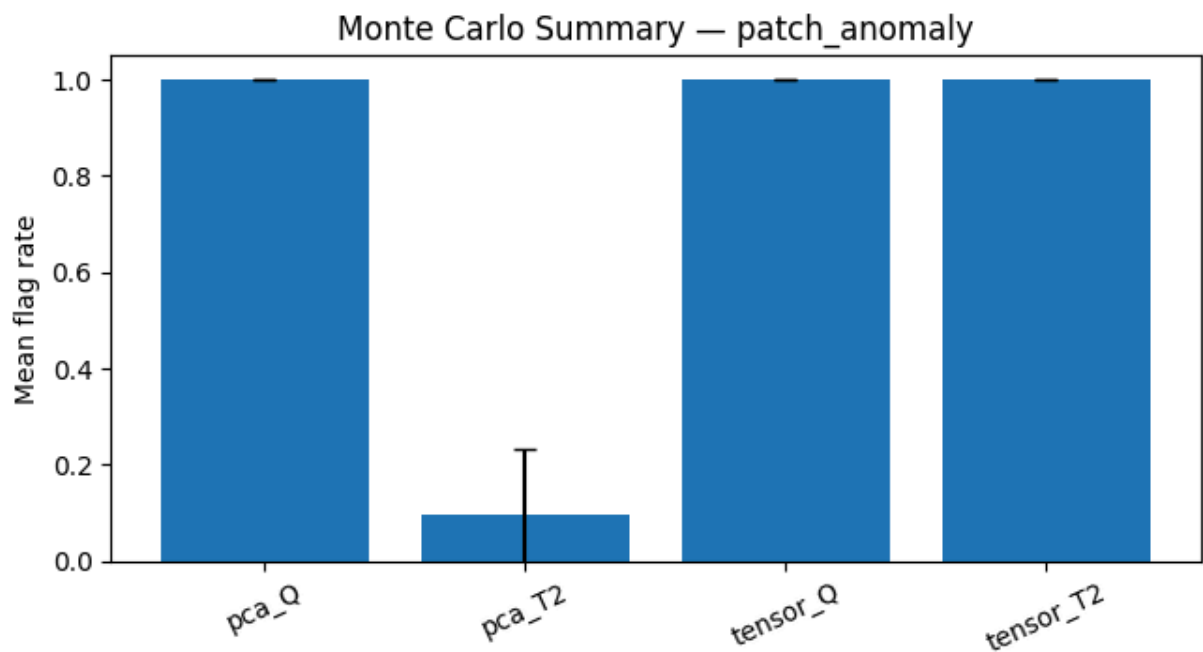
for fault in fault_order:
    sub = mc_summary[mc_summary["fault_type"] == fault].copy()
    fig, ax = plt.subplots(figsize=(7.5, 3.5))
    ax.bar(sub["method"], sub["mean"], yerr=sub["std"], capsized=4, linewidth=LW)
    ax.set_ylim(0, 1.05)
    ax.set_ylabel("Mean flag rate")
    ax.set_title(f"Monte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

Monte Carlo Summary — normal



Monte Carlo Summary — global_shift





12. Curated result 5 — ARL and detection delay summary

```
In [12]: 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),
        {"method_stat": "tensor_Q", "run_length": first_alarm_time(np.array(t_q), t
        {"method_stat": "pca_T2", "run_length": first_alarm_time(np.array(p_t2), pc
        {"method_stat": "pca_Q", "run_length": first_alarm_time(np.array(p_q), pca
    ])

    # 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": detecti
    {"fault_type": fault_name, "method_stat": "tensor_Q", "delay": detectio
    {"fault_type": fault_name, "method_stat": "pca_T2", "delay": detection_
    {"fault_type": fault_name, "method_stat": "pca_Q", "delay": detection_d
])

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

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

arl_summary, delay_summary.head()

```

```

Out[12]: ( method_stat  mean      std
0      pca_Q    75.15  86.597238
1      pca_T2    89.30  75.046722
2      tensor_Q  121.95  86.367925
3      tensor_T2  96.35  91.486712,
          fault_type method_stat  mean      std
0  persistent_shape_change      pca_Q    1.00  0.000000
1  persistent_shape_change      pca_T2   19.25  12.912153
2  persistent_shape_change      tensor_Q    1.00  0.000000
3  persistent_shape_change      tensor_T2   80.55  57.260692
4      small_global_shift      pca_Q    3.60  2.479389)

```

13. ARL and delay plots

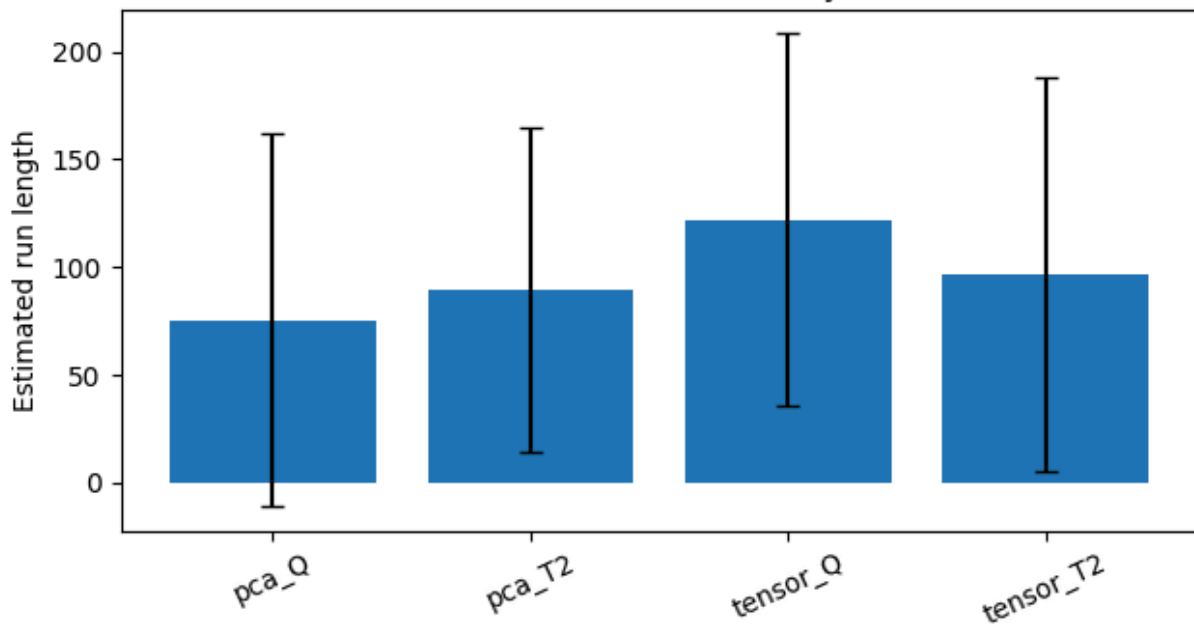
```

In [13]: fig, ax = plt.subplots(figsize=(7.2, 3.5))
ax.bar(arl_summary["method_stat"], arl_summary["mean"], yerr=arl_summary["std"], ca
ax.set_ylabel("Estimated run length")
ax.set_title("Curated ARL Summary")
plt.xticks(rotation=25)
plt.show()

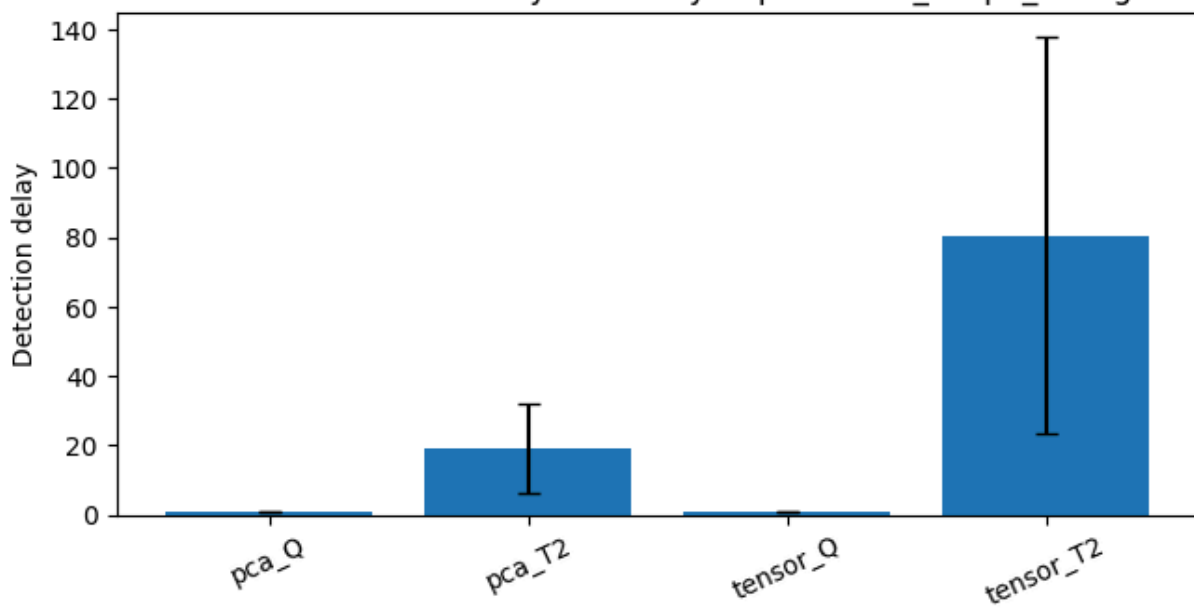
for fault in delay_summary["fault_type"].unique():
    sub = delay_summary[delay_summary["fault_type"] == fault]
    fig, ax = plt.subplots(figsize=(7.5, 3.5))
    ax.bar(sub["method_stat"], sub["mean"], yerr=sub["std"], capsize=4, linewidth=L
    ax.set_ylabel("Detection delay")
    ax.set_title(f"Curated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

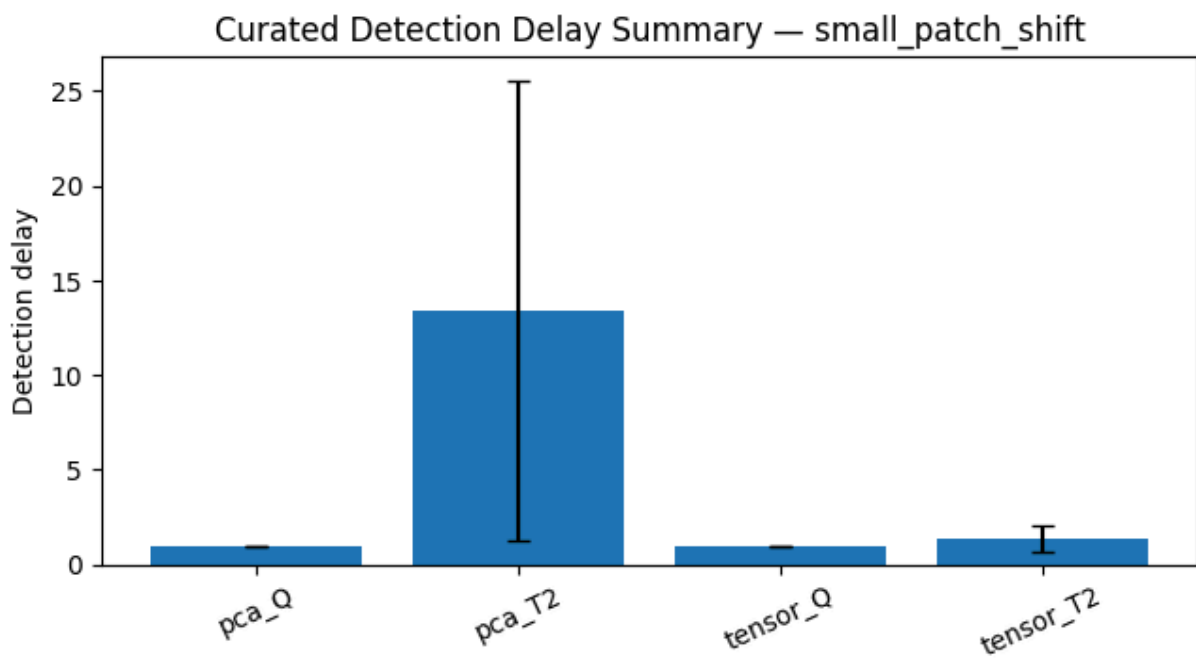
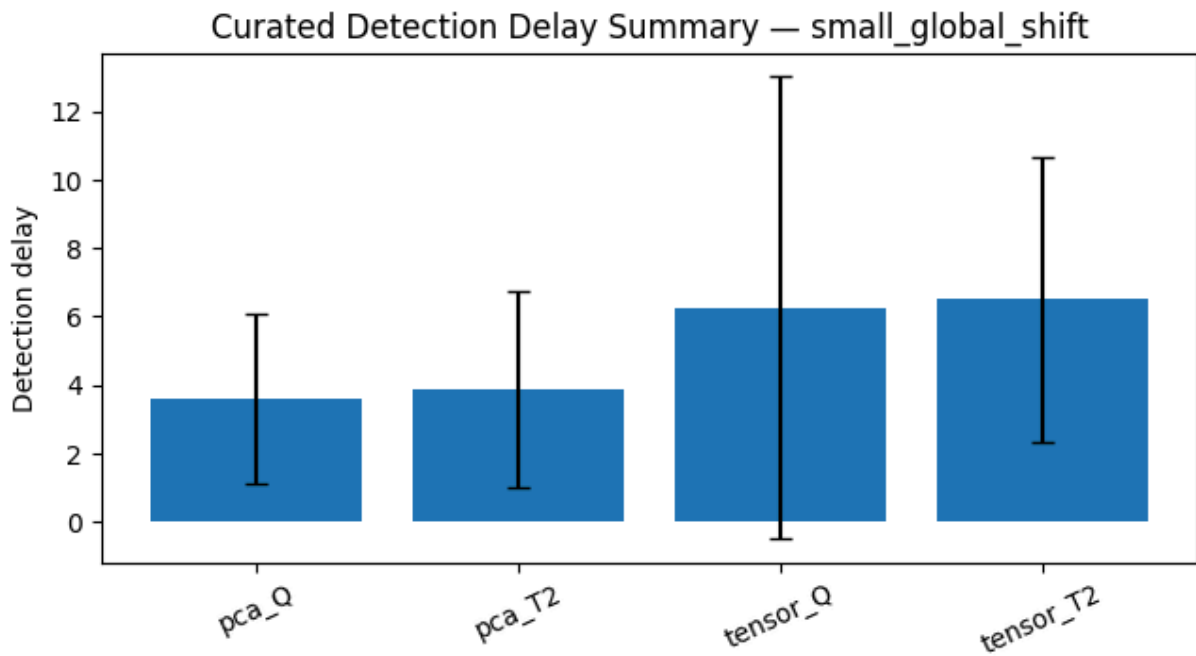
```

Curated ARL Summary



Curated Detection Delay Summary — persistent_shape_change





14. Curated result 6 — sequential EWMA/CUSUM tensor vs PCA comparison

```
In [14]: # 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(),
        q_std = standardize_with_reference(q_vals, tensor_model["q_ref"].mean(), te
    else:
        t2_std = standardize_with_reference(t2_vals, pca_model["t2_ref"].mean(), pc
        q_std = standardize_with_reference(q_vals, pca_model["q_ref"].mean(), pca_m

    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")

```

15. EWMA/CUSUM sequential comparison plots

```

In [15]: fig, ax = plt.subplots(figsize=(9, 3.6))
ax.plot(tensor_seq["EWMA_T2"], linewidth=LW, label="Tensor EWMA-T2")
ax.plot(pca_seq["EWMA_T2"], linewidth=LW, label="PCA EWMA-T2")
ax.axvline(change_point, linewidth=LW, linestyle='--')
ax.set_title("EWMA-T2: 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-T2")
ax.plot(pca_seq["CUSUM_T2"], linewidth=LW, label="PCA CUSUM-T2")

```

```

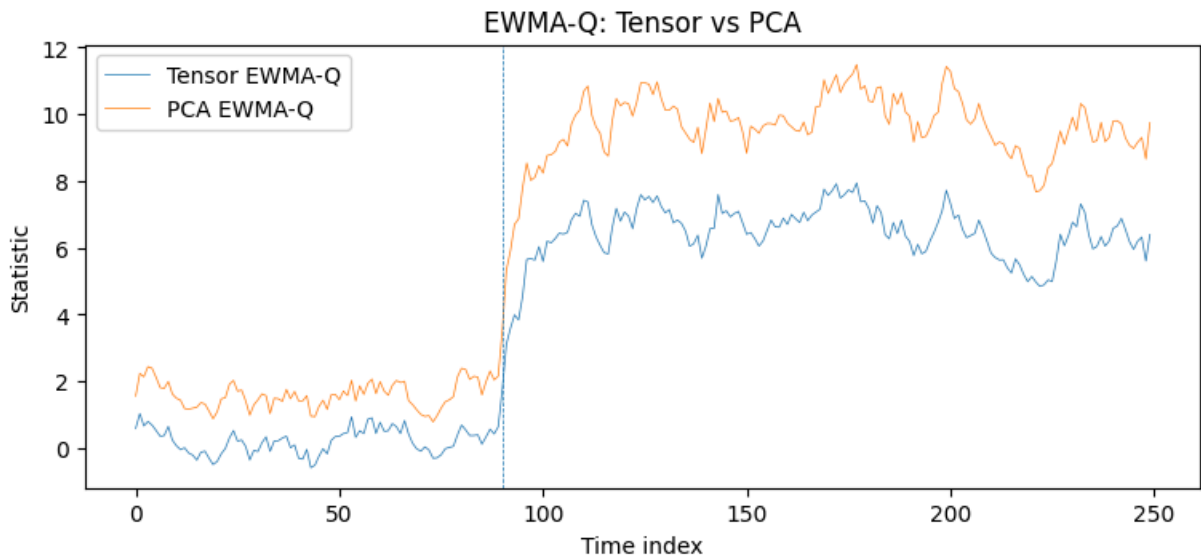
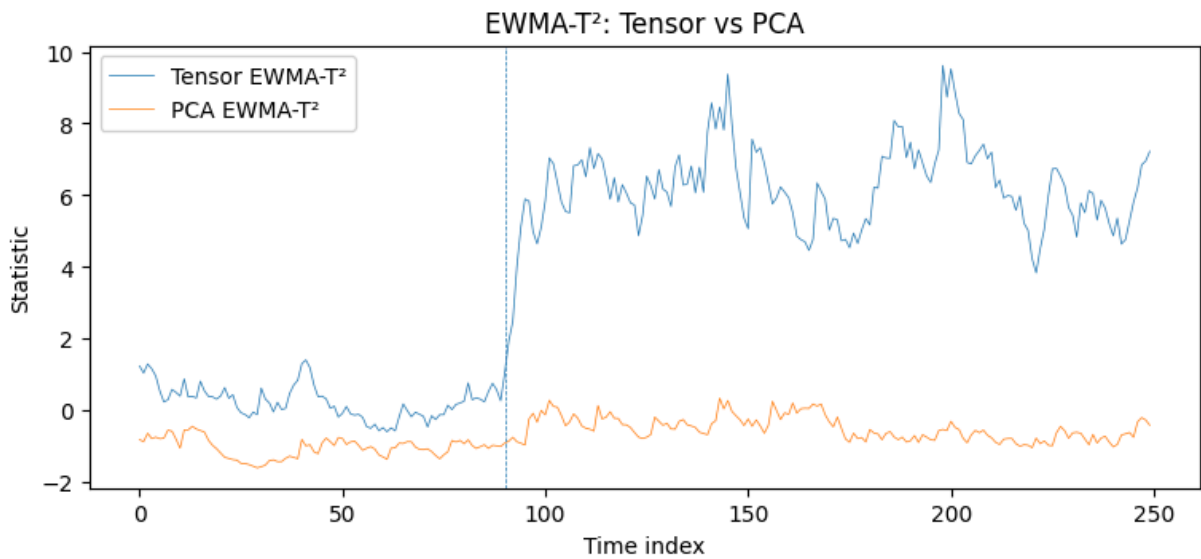
ax.axvline(change_point, linewidth=LW, linestyle='--')
ax.set_title("CUSUM-T2: Tensor vs PCA")
ax.set_xlabel("Time index")
ax.set_ylabel("Statistic")
ax.legend()
plt.show()

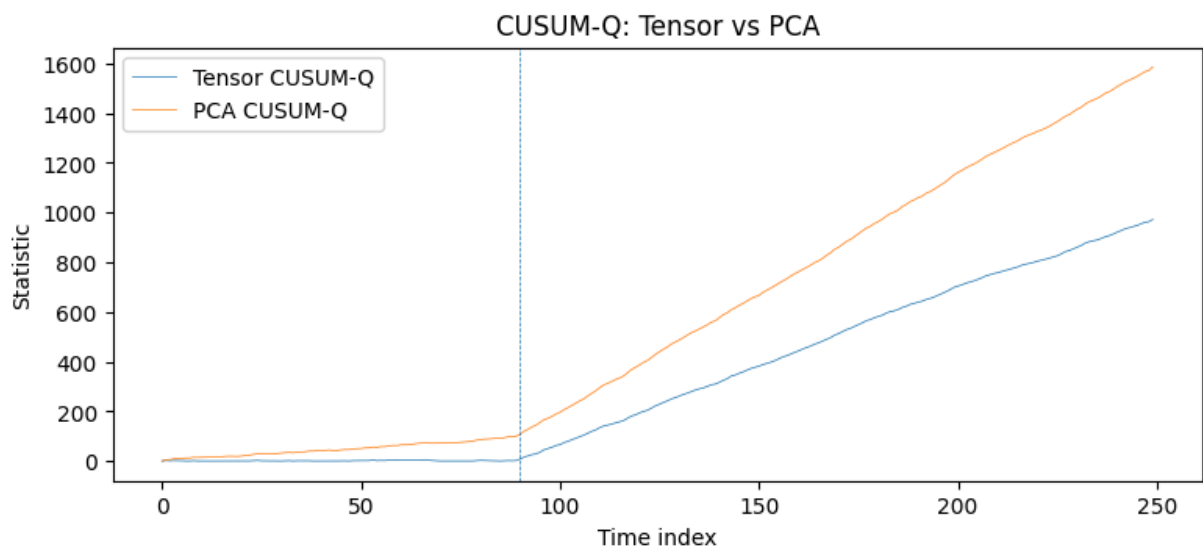
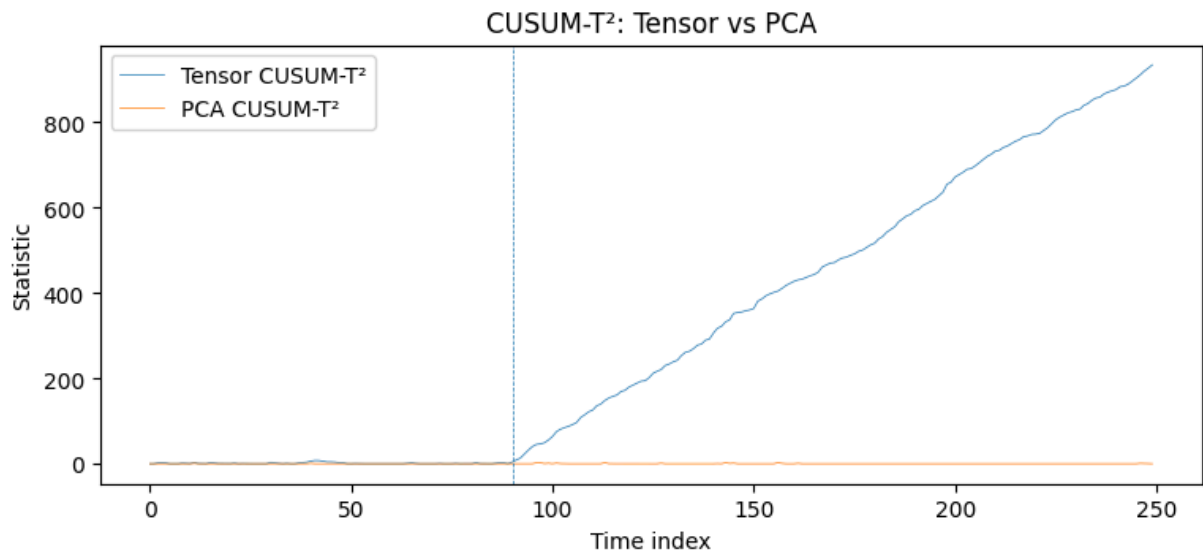
```

```

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()

```





16. Final curated summary tables

```
In [16]: 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))
```

Curated detection summary table:

| | method | pca_Q | pca_T2 | tensor_Q | tensor_T2 |
|----------------------|-------------------|--------------|---------------|-----------------|------------------|
| | fault_type | | | | |
| global_shift | 1.000 | 0.942 | 0.950 | 0.992 | |
| normal | 0.238 | 0.000 | 0.025 | 0.042 | |
| patch_anomaly | 1.000 | 0.094 | 1.000 | 1.000 | |
| shape_change | 1.000 | 0.578 | 1.000 | 0.133 | |

Curated delay summary table:

| | method_stat | pca_Q | pca_T2 | tensor_Q | tensor_T2 |
|--------------------------------|--------------------|--------------|---------------|-----------------|------------------|
| | fault_type | | | | |
| persistent_shape_change | | 1.0 | 19.25 | 1.00 | 80.55 |
| small_global_shift | | 3.6 | 3.85 | 6.25 | 6.50 |
| small_patch_shift | | 1.0 | 13.40 | 1.00 | 1.35 |

Curated ARL summary table:

| | method_stat | mean | std |
|----------|--------------------|-------------|------------|
| 0 | pca_Q | 75.15 | 86.60 |
| 1 | pca_T2 | 89.30 | 75.05 |
| 2 | tensor_Q | 121.95 | 86.37 |
| 3 | tensor_T2 | 96.35 | 91.49 |

17. Interpretation

This notebook 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 notebook that is appropriate to print or adapt into a monograph chapter.