Appendix A: Reproducible Code for the Structural Singularization Benchmark

Companion appendix to "Detecting Structural Singularization in Complex Systems via Finite-Time Spectral Entropy and Relaxation Lag"

A.1 Overview

This appendix provides the reproducible validation pipeline for the finite-time spectral entropy and relaxation lag framework introduced in the main text. The code computes the observable effective rank Φ_obs, the relaxation proxy R_obs, the trend-based fragility signature, and a classical early warning signal (EWS) baseline (variance and lag-1 autocorrelation on the leading principal component). Φ_obs and R_obs are observable, finite-window proxies for the underlying structural and dynamical quantities; their values depend on the estimation window, the τ grid, and the regularization choices documented below. The same pipeline is applied without modification to all benchmark systems.

The following design choices apply throughout:

- The same estimation pipeline is used for every system; no per-system tuning is performed.
- Random seeds are fixed for each simulator to make results reproducible.
- Outputs include Φ_obs, R_obs, the fragility synchronization flag, the classical EWS baseline, and a helper for ROC/AUC evaluation when ground-truth labels are supplied.
- The code provides the reproducible validation framework; it is not claimed to reproduce specific AUC values reported in the main text unless executed end-to-end with matched experimental conditions.

A.2 Required Libraries

The pipeline depends on standard scientific Python libraries. NumPy and SciPy provide linear algebra and signal processing primitives; scikit-learn supplies the Ledoit–Wolf shrinkage estimator and ROC utilities; Matplotlib is used for plotting only.

```
import numpy as np
import scipy.linalg as la
from scipy.signal import correlate
from scipy.stats import kendalltau
from sklearn.covariance import LedoitWolf
from sklearn.metrics import roc_auc_score, roc_curve
```

```
import matplotlib.pyplot as plt
```

A.3 Core Estimators

The core routine standardizes the multivariate input, estimates the covariance with Ledoit–Wolf shrinkage, regularizes it for invertibility, and applies a symmetric whitening transform. For each lag τ in a user-supplied grid it forms the lagged cross-covariance, computes the whitened operator, and extracts (i) Φ_obs as the normalized spectral entropy of the operator's nonnegative eigenvalues and (ii) R_obs as a scalar relaxation proxy derived from the spectral norm of the lagged covariance relative to the regularized covariance.

```
def compute_phi_r_obs(X, tau_grid=None, epsilon=1e-6):
    """
    Compute observable finite-time rank Phi_obs and relaxation proxy R_obs
    from a multivariate time series X with shape (n_samples, n_features).
    """
    X = np.asarray(X, dtype=float)
    X = (X - X.mean(axis=0, keepdims=True)) / (X.std(axis=0, keepdims=True) + 1e-12)

    n_samples, n_features = X.shape

    if tau_grid is None:
        tau_grid = np.arange(1, min(30, n_samples // 10))

    lw = LedoitWolf().fit(X)
    Sigma = lw.covariance_
    Sigma_reg = Sigma + epsilon * np.eye(n_features)

    eigvals, eigvecs = la.eigh(Sigma_reg)
    eigvals = np.maximum(eigvals, epsilon)
    Sigma_inv_sqrt = eigvecs @ np.diag(1.0 / np.sqrt(eigvals)) @ eigvecs.T

    phi_values = []
    r_values = []

    for tau in tau_grid:
        X1 = X[tau:]
        X0 = X[:-tau]

        C_tau = (X1.T @ X0) / (len(X1) - 1)

        M_tau = Sigma_inv_sqrt @ C_tau @ C_tau.T @ Sigma_inv_sqrt
        M_tau = 0.5 * (M_tau + M_tau.T)

        lambdas = np.maximum(la.eigvalsh(M_tau), 0.0)

        if lambdas.sum() <= 1e-12:
```

```
            phi = np.nan
        else:
            p = lambdas / lambdas.sum()
            phi = np.exp(-np.sum(p * np.log(p + 1e-12)))
            phi = phi / n_features

        spectral_norm = la.svdvals(C_tau)[0]
        sigma_norm = la.svdvals(Sigma_reg)[0]
        r_obs = 1.0 - spectral_norm / (sigma_norm + 1e-12)

        phi_values.append(phi)
        r_values.append(r_obs)

    return np.asarray(tau_grid), np.asarray(phi_values), np.asarray(r_values)
```

A.4 Fragility Signature

The fragility signature is a threshold-free, trend-based detector. It uses Kendall's τ to test for monotone change in Φ_obs and R_obs across the evaluated index (typically a lag grid or a sequence of rolling windows). Consistent with the main manuscript, the criterion fires when Φ_obs trends downward (structural compression: the active dynamical subspace contracts) while R_obs trends upward (dynamical slowing: the characteristic relaxation time grows). The function returns the τ statistics, their p-values, and a boolean flag indicating this joint Φ↓ / R↑ pattern. Because Φ_obs and R_obs are observable proxies estimated from finite windows, the synchronization flag is to be interpreted as an empirical fragility signature rather than as a claim about the underlying operator.

```
def fragility_signature(phi_series, r_series):
    """
    Compute trend synchronization between structural compression and
    dynamical slowing.

    A fragile pattern is indicated when Phi decreases while R increases
    over the evaluated scale or time window. Phi_obs and R_obs are
    observable proxies, so the criterion is interpreted as a finite-time
    empirical signature rather than an exact dynamical statement.
    """
    idx = np.arange(len(phi_series))

    # Kendall trend tests: monotone decrease in Phi_obs (structural
    # compression) and monotone increase in R_obs (relaxation lag).
    tau_phi, p_phi = kendalltau(idx, phi_series, nan_policy="omit")
    tau_r, p_r = kendalltau(idx, r_series, nan_policy="omit")

    # Phase-synchronization criterion: Phi decreasing AND R increasing.
    synchronized = (tau_phi < 0) and (tau_r > 0)
```

```
    return {
        "kendall_phi": tau_phi,
        "kendall_r": tau_r,
        "p_phi": p_phi,
        "p_r": p_r,
        "synchronized": synchronized
    }
```

Note: Earlier versions used a symmetric trend condition (Φ↓ and R↓), which is inconsistent with the theoretical framework. The corrected formulation (Φ↓, R↑) reflects the separation of structural compression and dynamical slowing.

A.5 Benchmark Systems

Four numerical benchmarks are included. Each generator returns an array of shape (n_samples, n_features) and uses an explicit seed. The systems span (i) a slow-fold-like Ornstein–Uhlenbeck system whose leading eigenvalue drifts toward zero, (ii) a Hopf-like system whose leading pair of eigenvalues crosses the imaginary axis, (iii) a non-normal stable system that supports transient amplification without bifurcating, and (iv) a noise-induced switching system in a double-well potential.

A.5.1 Fold-like OU system

The diagonal element associated with the slow direction is annealed linearly toward zero across the trajectory, mimicking the local structure of an approach to a fold bifurcation in an otherwise Ornstein–Uhlenbeck system.

```
def simulate_fold_ou(n_samples=5000, n_features=10, seed=1):
    rng = np.random.default_rng(seed)
    X = np.zeros((n_samples, n_features))

    for t in range(1, n_samples):
        mu = 0.8 - 0.75 * (t / n_samples)
        A = -np.eye(n_features)
        A[0, 0] = -mu
        noise = 0.05 * rng.standard_normal(n_features)
        X[t] = X[t-1] + A @ X[t-1] * 0.01 + noise

    return X
```

A.5.2 Hopf-like system

A 2×2 oscillatory block embedded in a higher-dimensional damped system. The real part of the leading eigenvalue pair crosses zero as the control parameter μ is annealed.

```
def simulate_hopf(n_samples=5000, n_features=10, seed=2):
    rng = np.random.default_rng(seed)
    X = np.zeros((n_samples, n_features))

    for t in range(1, n_samples):
        mu = -0.8 + 0.75 * (t / n_samples)
        omega = 1.5

        A = -np.eye(n_features)
        A[0, 0] = mu
        A[1, 1] = mu
        A[0, 1] = -omega
        A[1, 0] = omega

        noise = 0.05 * rng.standard_normal(n_features)
        X[t] = X[t-1] + A @ X[t-1] * 0.01 + noise

    return X
```

A.5.3 Non-normal stable system

An upper-bidiagonal operator with strong off-diagonal coupling produces a highly non-normal but spectrally stable dynamics. The eigenvalues all lie at −0.6, but the pseudospectrum extends substantially into the right half plane, generating large transient amplification with no bifurcation.

```
def simulate_non_normal(n_samples=5000, n_features=10, seed=3):
    rng = np.random.default_rng(seed)
    X = np.zeros((n_samples, n_features))

    A = -0.6 * np.eye(n_features)
    for i in range(n_features - 1):
        A[i, i + 1] = 6.0

    dt = 0.005

    for t in range(1, n_samples):
        noise = 0.05 * rng.standard_normal(n_features)
        X[t] = X[t-1] + A @ X[t-1] * dt + noise

    return X
```

A.5.4 Noise-induced switching

A scalar double-well component (drift $x - x^3$) coupled to additional stable noise channels. Transitions between wells are driven by stochastic fluctuations rather than by a parameter change, providing a control case in which classical CSD indicators are not expected to fire prior to the switch.

```
def simulate_noise_induced(n_samples=5000, n_features=10, seed=4):
    rng = np.random.default_rng(seed)
    X = np.zeros((n_samples, n_features))
    dt = 0.01

    for t in range(1, n_samples):
        x0 = X[t-1, 0]
        drift0 = x0 - x0**3
        noise = 0.10 * rng.standard_normal(n_features)

        X[t] = X[t-1]
        X[t, 0] += drift0 * dt + noise[0]
        X[t, 1:] += -0.7 * X[t-1, 1:] * dt + noise[1:]

    return X
```

A.6 Rolling Evaluation

The rolling-window driver applies the core estimator and the fragility signature on overlapping segments of the trajectory and returns centered timestamps, mean Φ_obs, mean R_obs, and a binary synchronization flag for each window.

```
def rolling_metrics(X, window=500, step=50, tau_grid=None):
    times = []
    phi_mean = []
    r_mean = []
    sync_flags = []

    for start in range(0, len(X) - window, step):
        stop = start + window
        Xw = X[start:stop]

        taus, phi, r = compute_phi_r_obs(Xw, tau_grid=tau_grid)
        sig = fragility_signature(phi, r)

        times.append((start + stop) // 2)
        phi_mean.append(np.nanmean(phi))
        r_mean.append(np.nanmean(r))
        sync_flags.append(sig["synchronized"])

    return {
        "time": np.asarray(times),
```

```
        "phi": np.asarray(phi_mean),
        "r": np.asarray(r_mean),
        "sync": np.asarray(sync_flags, dtype=int)
    }
```

A.7 Classical EWS Baseline

For comparison with the proposed framework, a classical EWS baseline is computed on the leading principal-component score. Within each rolling window the routine returns sample variance and lag-1 autocorrelation, the two most widely used CSD indicators.

```
def classical_ews_baseline(X, window=500, step=50):
    times = []
    variance = []
    ar1 = []

    pc1_series = first_pc_score(X)

    for start in range(0, len(pc1_series) - window, step):
        stop = start + window
        y = pc1_series[start:stop]
        y = y - y.mean()

        var = np.var(y)

        if np.std(y[:-1]) < 1e-12 or np.std(y[1:]) < 1e-12:
            rho = np.nan
        else:
            rho = np.corrcoef(y[:-1], y[1:])[0, 1]

        times.append((start + stop) // 2)
        variance.append(var)
        ar1.append(rho)

    return {
        "time": np.asarray(times),
        "variance": np.asarray(variance),
        "ar1": np.asarray(ar1)
    }



def first_pc_score(X):
    X = (X - X.mean(axis=0)) / (X.std(axis=0) + 1e-12)
    U, S, Vt = la.svd(X, full_matrices=False)
    return U[:, 0] * S[0]
```

A.8 ROC/AUC Evaluation

When binary labels are available (for example, a per-window indicator of proximity to a tipping point), AUC can be computed directly from any scalar EWS or fragility score. The helper guards against degenerate label distributions and missing values.

```
def evaluate_auc(signal, labels):
    mask = np.isfinite(signal) & np.isfinite(labels)
    if len(np.unique(labels[mask])) < 2:
        return np.nan
    return roc_auc_score(labels[mask], signal[mask])
```

A.9 Plotting

A minimal plotting helper is provided. It overlays Φ_obs and R_obs against window-centered time and is intended for inspection rather than publication-grade rendering.

```
def plot_metrics(result, title="Structural metrics"):
    fig, ax = plt.subplots(figsize=(8, 4))
    ax.plot(result["time"], result["phi"], label="Phi_obs")
    ax.plot(result["time"], result["r"], label="R_obs")
    ax.set_title(title)
    ax.set_xlabel("time")
    ax.set_ylabel("observable")
    ax.legend()
    fig.tight_layout()
    return fig
```

A.10 Reproducibility Script

The following entry point runs the pipeline on all four benchmark systems, prints per-system summary statistics, and writes one figure per system. It serves as a single-command reproducibility test of the appendix.

```
if __name__ == "__main__":
    systems = {
        "Fold": simulate_fold_ou(seed=1),
        "Hopf": simulate_hopf(seed=2),
        "Non-normal": simulate_non_normal(seed=3),
        "Noise-induced": simulate_noise_induced(seed=4),
    }

    for name, X in systems.items():
        result = rolling_metrics(X, window=500, step=50)
        print(name)
        print("Mean Phi_obs:", np.nanmean(result["phi"]))
        print("Mean R_obs:", np.nanmean(result["r"]))
        print("Sync rate:", np.mean(result["sync"]))
```

```
        fig = plot_metrics(result, title=name)
        fig.savefig(f"{name.lower().replace(' ', '_')}_metrics.png", dpi=200)
```

This code listing is intended as a self-contained companion to the main manuscript. It defines the estimators, simulators, and evaluation utilities required to reproduce the validation framework; quantitative outputs depend on simulation length, window size, and the specific tau grid in use, and should be reported alongside any AUC or detection statistics derived from this code.