```
"""
Σ-metrics Engine
Teoría Cromodinámica Sincrónica (TCDS)

Calcula métricas de coherencia desde series temporales:
- LI (Locking Index): Estabilidad de fase/amplitud
- R: Correlación temporal
- RMSE_SL: Error de ajuste slope-lock
- κΣ: Curvatura de coherencia (rigidez)
- ΔH: Cambio de entropía normalizada (Shannon)

Autor: Genaro Carrasco Ozuna
ORCID: 0009-0005-6358-9910
"""

import numpy as np
from scipy import signal, stats
from typing import Dict, Tuple
import warnings

warnings.filterwarnings('ignore')


def compute_locking_index(signal_array: np.ndarray, window_size: int = None) -> float:
    """
    Calcula el Locking Index (LI).

    Mide estabilidad de fase/amplitud usando la varianza normalizada
    de la señal en ventanas deslizantes.

    Args:
        signal_array: Serie temporal 1D
        window_size: Tamaño de ventana (default: len(signal)/10)

    Returns:
        LI ∈ [0, 1]: 1 = máximo locking, 0 = ruido completo
    """
    if len(signal_array) < 10:
        return 0.0

    if window_size is None:
        window_size = max(10, len(signal_array) // 10)

    # Normalizar señal
    sig_norm = (signal_array - np.mean(signal_array)) / (np.std(signal_array) + 1e-10)

    # Calcular varianza en ventanas
    n_windows = len(sig_norm) // window_size
    if n_windows < 2:
        return 0.0

    variances = []
    for i in range(n_windows):
        window = sig_norm[i*window_size:(i+1)*window_size]
        variances.append(np.var(window))

    # LI = 1 - coef_variación(varianzas)
    # Alta estabilidad → varianzas constantes → CV bajo → LI alto
    mean_var = np.mean(variances)
    std_var = np.std(variances)
```

```
    if mean_var < 1e-10:
        return 0.0

    cv = std_var / mean_var
    li = max(0.0, min(1.0, 1.0 - cv))

    return li


def compute_correlation_r(signal_array: np.ndarray, lag: int = 1) -> float:
    """
    Calcula correlación temporal R(t).

    Mide autocorrelación con lag especificado.

    Args:
        signal_array: Serie temporal 1D
        lag: Desplazamiento temporal (default: 1)

    Returns:
        R ∈ [-1, 1]: correlación de Pearson
    """
    if len(signal_array) < lag + 2:
        return 0.0

    sig1 = signal_array[:-lag]
    sig2 = signal_array[lag:]

    if len(sig1) < 2:
        return 0.0

    r, _ = stats.pearsonr(sig1, sig2)

    if np.isnan(r):
        return 0.0

    return abs(r)  # Tomamos valor absoluto para medir coherencia


def compute_rmse_sl(signal_array: np.ndarray) -> float:
    """
    Calcula RMSE de ajuste slope-lock.

    Mide cuán bien la señal sigue una tendencia lineal.
    Bajo RMSE_SL → alta predictibilidad estructural.

    Args:
        signal_array: Serie temporal 1D

    Returns:
        RMSE_SL ≥ 0: error cuadrático medio normalizado
    """
    if len(signal_array) < 3:
        return 1.0

    x = np.arange(len(signal_array))

    # Ajuste lineal
    try:
        slope, intercept = np.polyfit(x, signal_array, 1)
        y_pred = slope * x + intercept
```

```
        # RMSE normalizado por desviación estándar
        residuals = signal_array - y_pred
        rmse = np.sqrt(np.mean(residuals**2))
        std_signal = np.std(signal_array)

        if std_signal < 1e-10:
            return 0.0

        rmse_sl = rmse / std_signal

        return min(1.0, rmse_sl)  # Cap a 1.0 para normalización

    except:
        return 1.0


def compute_kappa_sigma(signal_array: np.ndarray) -> float:
    """
    Calcula κΣ (curvatura de coherencia).

    Mide rigidez estructural mediante segunda derivada.
    Alto κΣ → alta rigidez → coherencia estructural fuerte.

    Args:
        signal_array: Serie temporal 1D

    Returns:
        κΣ ≥ 0: curvatura promedio normalizada
    """
    if len(signal_array) < 3:
        return 0.0

    # Segunda derivada discreta
    second_deriv = np.diff(signal_array, n=2)

    # κΣ = promedio(|d²Σ/dt²|) / std(Σ)
    std_signal = np.std(signal_array)

    if std_signal < 1e-10:
        return 0.0

    kappa = np.mean(np.abs(second_deriv)) / std_signal

    return kappa


def compute_delta_h(signal_array: np.ndarray, n_bins: int = 10) -> float:
    """
    Calcula ΔH (cambio de entropía normalizada de Shannon).

    Compara entropía entre primera y segunda mitad de la señal.
    ΔH < 0 → reducción entrópica (orden emergente)
    ΔH > 0 → incremento entrópico (desorden)

    Args:
        signal_array: Serie temporal 1D
        n_bins: Número de bins para histograma (default: 10)

    Returns:
        ΔH ∈ (-∞, ∞): cambio normalizado de entropía
    """
    if len(signal_array) < 20:
        return 0.0
```

```
    # Dividir en dos mitades
    mid = len(signal_array) // 2
    first_half = signal_array[:mid]
    second_half = signal_array[mid:]

    def entropy_shannon(data):
        """Calcula entropía de Shannon normalizada"""
        if len(data) < 2:
            return 0.0

        # Histograma
        hist, _ = np.histogram(data, bins=n_bins)
        hist = hist[hist > 0]  # Eliminar bins vacíos

        if len(hist) == 0:
            return 0.0

        # Probabilidades
        probs = hist / np.sum(hist)

        # Entropía de Shannon
        h = -np.sum(probs * np.log2(probs + 1e-10))

        # Normalizar por máxima entropía (log2(n_bins))
        h_norm = h / np.log2(n_bins)

        return h_norm

    h_pre = entropy_shannon(first_half)
    h_post = entropy_shannon(second_half)

    delta_h = h_post - h_pre

    return delta_h


def compute_sigma_metrics(
    signal_array: np.ndarray,
    window_size: int = None,
    lag: int = 1,
    n_bins_entropy: int = 10
) -> Dict[str, float]:
    """
    Calcula todas las Σ-metrics de una ventana temporal.

    Args:
        signal_array: Serie temporal 1D
        window_size: Tamaño ventana para LI (default: auto)
        lag: Desplazamiento para R (default: 1)
        n_bins_entropy: Bins para ΔH (default: 10)

    Returns:
        Dict con claves: 'LI', 'R', 'RMSE_SL', 'κΣ', 'ΔH'
    """
    if len(signal_array) < 10:
        return {
            'LI': 0.0,
            'R': 0.0,
            'RMSE_SL': 1.0,
            'κΣ': 0.0,
            'ΔH': 0.0
        }
```

```
    metrics = {
        'LI': compute_locking_index(signal_array, window_size),
        'R': compute_correlation_r(signal_array, lag),
        'RMSE_SL': compute_rmse_sl(signal_array),
        'κΣ': compute_kappa_sigma(signal_array),
        'ΔH': compute_delta_h(signal_array, n_bins_entropy)
    }

    return metrics


# Test de auto-validación
if __name__ == "__main__":
    print("=== Test Σ-metrics Engine ===\n")

    # Señal Q-driven: sinusoide estable con ruido bajo
    t = np.linspace(0, 10, 500)
    signal_q = np.sin(2 * np.pi * 1.5 * t) + 0.05 * np.random.randn(len(t))

    # Señal φ-driven: ruido gaussiano
    signal_phi = np.random.randn(500)

    # Señal borderline: mezcla
    signal_border = 0.6 * np.sin(2 * np.pi * t) + 0.4 * np.random.randn(len(t))

    test_signals = {
        "Q-driven (sinusoide limpia)": signal_q,
        "φ-driven (ruido blanco)": signal_phi,
        "Borderline (mezcla)": signal_border
    }

    for name, sig in test_signals.items():
        metrics = compute_sigma_metrics(sig)
        print(f"[{name}]")
        for k, v in metrics.items():
            print(f"  {k}: {v:.4f}")
        print()
```