PROTOCOLO COMPLETO DE ANÁLISIS DE ESTABILIDAD ACÚSTICA

Implementación computacional para investigación académica

```python
"""
================================================================================
PROTOCOLO COMPLETO DE ANÁLISIS DE ESTABILIDAD ACÚSTICA - 432 Hz
Versión académica corregida y optimizada
================================================================================

Referencia teórica: "Quantum Sonorology: Sound Fission and the Test of
                     Biophysical Stability of Vibrating Matter at 432/440 Hz"
Autor del estudio: Julio Alberto Solis Trejo (Liah Steer)

Implementación: Análisis completo con generación de datos, visualizaciones
                y exportación de resultados en formato académico.

Versión: 2.1 (Corregida para uso académico)
Licencia: CC BY 4.0 - Uso académico y científico permitido con atribución

Correcciones implementadas:
1. Eliminación de importaciones redundantes
2. Corrección de referencias a variables no definidas
3. Optimización de cálculos espectrales
```

4. Mejora en manejo de errores y validación

5. Documentación completa en español para contexto académico

==========

"""

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy.fft import rfft, rfftfreq
from scipy.stats import shapiro, ttest_ind, mannwhitneyu
import pandas as pd
import seaborn as sns
from datetime import datetime
import json
import warnings
from typing import Dict, List, Tuple, Optional, Union
import sys

warnings.filterwarnings('ignore')

# Configuración de estilo para gráficos científicos
sns.set_style("whitegrid")
sns.set_palette("husl")
plt.rcParams['figure.dpi'] = 100
plt.rcParams['savefig.dpi'] = 300
```

```
plt.rcParams['font.size'] = 10
plt.rcParams['axes.labelsize'] = 11
plt.rcParams['axes.titlesize'] = 12
plt.rcParams['legend.fontsize'] = 9

#
================================================================
========
# CLASE PRINCIPAL: Analizador de Estabilidad Acústica
#
================================================================
========

class AnalizadorEstabilidadAcustica:
    """
    Implementación computacional del protocolo FSE/FDT para análisis de
    estabilidad acústica según el estudio de Solis Trejo (2024).

    Protocolos implementados:
    1. FSE (Fisión por Saturación de Energía): Evalúa la energía requerida
       para inducir fisión acústica mediante incremento sistemático de ganancia.

    2. FDT (Fisión por Desincronización Temporal): Determina el umbral temporal
       de desincronización necesario para pérdida de coherencia.

    Versión 2.1 - Corregida y optimizada para publicación académica.
    """
```

```
# Constantes del protocolo (basadas en estándares de audio profesional)
DURACION_FADE_S = 0.05
VENTANA_BUSQUEDA_HZ = 2.0
GANANCIA_MAXIMA_DB = 25.0
TOLERANCIA_TIEMPO_MS = 0.05
PASO_GANANCIA_DB = 0.1
UMBRAL_SILENCIO = 1e-10
UMBRAL_SNR = 3.0
OBJETIVO_COHERENCIA = 0.50
TOLERANCIA_CONVERGENCIA = 0.05
UMBRAL_THD_PORCENTAJE = 1.0
UMBRAL_CORRELACION = 0.70

# Parámetros estadísticos
ALFA_SIGNIFICANCIA = 0.05
MINIMO_REPLICAS = 3

def __init__(self, frecuencia_muestreo: int = 48000, nivel_rms_db: float = -18.0):
    """
    Inicializa el analizador con parámetros validados.

    Args:
        frecuencia_muestreo: Tasa de muestreo en Hz (estándar: 48000 Hz)
        nivel_rms_db: Nivel RMS en dBFS (-18 dB = margen estándar)
```

```
    Raises:
        ValueError: Si los parámetros son inválidos
    """
    # Validación de parámetros
    if frecuencia_muestreo <= 0:
        raise ValueError("La frecuencia de muestreo debe ser positiva")

    if nivel_rms_db > 0:
        raise ValueError("El nivel RMS debe ser negativo en escala dBFS")

    self.frecuencia_muestreo = int(frecuencia_muestreo)
    self.nivel_rms_db = float(nivel_rms_db)
    self.resultados = {}
    self.frecuencia_nyquist = self.frecuencia_muestreo / 2

    # Información del sistema
    print("=" * 80)
    print(" ANALIZADOR DE ESTABILIDAD ACÚSTICA - PROTOCOLO FSE/FDT v2.1")
    print("=" * 80)
    print(f" Frecuencia de muestreo: {self.frecuencia_muestreo} Hz")
    print(f" Frecuencia de Nyquist:  {self.frecuencia_nyquist} Hz")
    print(f" Nivel RMS:              {self.nivel_rms_db} dBFS")
    print(f" Fecha de análisis:      {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print("=" * 80 + "\n")

def _calcular_fade_adaptativo(self, frecuencia: float) -> float:
```

```
    """
    Calcula duración óptima de fade según frecuencia.

    Para prevenir discontinuidades, el fade debe cubrir al menos
    3 ciclos de la frecuencia fundamental.

    Args:
        frecuencia: Frecuencia fundamental en Hz

    Returns:
        Duración de fade en segundos
    """
    periodo = 1.0 / frecuencia
    ciclos_minimos = 3
    duracion_fade = max(self.DURACION_FADE_S, periodo * ciclos_minimos)

    # Límite superior razonable (200 ms)
    return min(duracion_fade, 0.2)

def _calcular_armonicos_adaptativos(self, frecuencia_fundamental: float) -> int:
    """
    Determina número óptimo de armónicos según frecuencia y Nyquist.

    Args:
        frecuencia_fundamental: Frecuencia fundamental en Hz
```

```
    Returns:
        Número de armónicos a analizar
    """
    if frecuencia_fundamental <= 0:
        return 6  # Valor por defecto

    # Máximo número de armónicos antes de exceder Nyquist
    max_armonicos = int(self.frecuencia_nyquist / frecuencia_fundamental) - 1

    # Limitar entre 3 y 10 armónicos para análisis significativo
    n_armonicos = max(3, min(10, max_armonicos))

    return n_armonicos

def generar_tono_puro(
    self,
    frecuencia: float,
    duracion: float = 5.0,
    aplicar_fade: bool = True
) -> Tuple[np.ndarray, np.ndarray]:
    """
    Genera tono sinusoidal puro según especificaciones del protocolo.

    Características técnicas:
    - Forma de onda: Sinusoidal pura (THD < 0.01%)
    - Precisión: ±0.001 Hz garantizada
```

```
- Fade-in/out: Curva logarítmica adaptativa

Args:
    frecuencia: Frecuencia fundamental en Hz
    duracion: Duración total en segundos
    aplicar_fade: Aplicar fade-in/out para prevenir clicks

Returns:
    tuple: (senal_audio, array_tiempo)

Raises:
    ValueError: Si los parámetros son inválidos
"""
# Validación de parámetros
if frecuencia <= 0 or frecuencia > self.frecuencia_nyquist:
    raise ValueError(
        f"Frecuencia debe estar entre 0 y {self.frecuencia_nyquist} Hz. "
        f"Recibido: {frecuencia} Hz"
    )

if duracion <= 0:
    raise ValueError("Duración debe ser positiva")

# Generación de array temporal
n_muestras = int(self.frecuencia_muestreo * duracion)
t = np.linspace(0, duracion, n_muestras, endpoint=False, dtype=np.float64)
```

```
# Generación de onda sinusoidal pura (fase inicial en 0)
audio = np.sin(2 * np.pi * frecuencia * t)

# Normalización a nivel RMS especificado
rms_lineal = 10 ** (self.nivel_rms_db / 20)
rms_actual = np.sqrt(np.mean(audio ** 2))

if rms_actual < self.UMBRAL_SILENCIO:
    raise ValueError("Señal generada es silencio (error crítico)")

audio = audio * (rms_lineal / rms_actual)

# Aplicar fade-in/out adaptativo
if aplicar_fade:
    duracion_fade = self._calcular_fade_adaptativo(frecuencia)
    muestras_fade = int(duracion_fade * self.frecuencia_muestreo)

    # Asegurar que fade no excede mitad de la señal
    if muestras_fade > len(audio) // 2:
        muestras_fade = len(audio) // 4

    if muestras_fade > 0:
        # Curva logarítmica para transición natural
        fade_in = np.logspace(-3, 0, muestras_fade, base=10.0)
        fade_in = fade_in / fade_in[-1]  # Normalizar a [0, 1]
```

```
            fade_out = np.logspace(0, -3, muestras_fade, base=10.0)
            fade_out = fade_out / fade_out[0]

            audio[:muestras_fade] *= fade_in
            audio[-muestras_fade:] *= fade_out

    return audio, t

def calcular_thd(
    self,
    audio: np.ndarray,
    frecuencia_fundamental: float,
    n_armonicos: Optional[int] = None
) -> Dict[str, Union[float, List[float]]]:
    """
    Calcula Distorsión Armónica Total (THD) según protocolo.

    Fórmula: THD = √(H₂² + H₃² + ... + Hₙ²) / H₁

    Umbral crítico: THD > 1.0%
    Justificación: Límite de detección en oyentes entrenados

    Args:
        audio: Señal de audio
        frecuencia_fundamental: Frecuencia fundamental en Hz
```

```
    n_armonicos: Número de armónicos (None = adaptativo)

Returns:
    dict: Resultados de análisis THD
"""
if len(audio) == 0:
    raise ValueError("Array de audio está vacío")

if np.all(np.abs(audio) < self.UMBRAL_SILENCIO):
    raise ValueError("Audio es silencio completo")

# Determinar número de armónicos
if n_armonicos is None:
    n_armonicos = self._calcular_armonicos_adaptativos(frecuencia_fundamental)

N = len(audio)

# Transformada rápida de Fourier optimizada
ventana = np.hanning(N)
resultado_fft = rfft(audio * ventana)
frecuencias = rfftfreq(N, 1 / self.frecuencia_muestreo)

# Magnitud espectral
magnitud = np.abs(resultado_fft)

# Extraer magnitudes de armónicos
```

```
armonicos = []
frecuencias_armonicas = []

for h in range(1, n_armonicos + 1):
    frecuencia_objetivo = frecuencia_fundamental * h

    # Verificar si armónico excede Nyquist
    if frecuencia_objetivo > self.frecuencia_nyquist:
        armonicos.append(0.0)
        frecuencias_armonicas.append(frecuencia_objetivo)
        continue

    # Búsqueda de pico en ventana espectral
    ventana_frec = self.VENTANA_BUSQUEDA_HZ
    mascara = (frecuencias >= frecuencia_objetivo - ventana_frec) & \
              (frecuencias <= frecuencia_objetivo + ventana_frec)

    if np.any(mascara):
        magnitud_filtrada = magnitud[mascara]
        frecuencias_filtradas = frecuencias[mascara]

        # Detectar pico significativo
        ruido_fondo = np.median(magnitud)
        if np.max(magnitud_filtrada) > ruido_fondo * self.UMBRAL_SNR:
            idx_local = np.argmax(magnitud_filtrada)
            magnitud_pico = magnitud_filtrada[idx_local]
```

```
            frecuencia_pico = frecuencias_filtradas[idx_local]

            armonicos.append(magnitud_pico)
            frecuencias_armonicas.append(frecuencia_pico)
        else:
            armonicos.append(0.0)
            frecuencias_armonicas.append(frecuencia_objetivo)
    else:
        armonicos.append(0.0)
        frecuencias_armonicas.append(frecuencia_objetivo)

# Cálculo de THD
H1 = armonicos[0]  # Fundamental

if H1 > self.UMBRAL_SILENCIO:
    suma_cuadrados = np.sum(np.array(armonicos[1:]) ** 2)
    thd_ratio = np.sqrt(suma_cuadrados) / H1
    thd_porcentaje = thd_ratio * 100

    if thd_ratio > 0:
        thd_db = 20 * np.log10(thd_ratio)
    else:
        thd_db = -np.inf
else:
    thd_ratio = 0.0
    thd_porcentaje = 0.0
```

```
        thd_db = -np.inf

    return {
        'thd_porcentaje': thd_porcentaje,
        'thd_db': thd_db,
        'thd_ratio': thd_ratio,
        'armonicos': armonicos,
        'frecuencias_armonicas': frecuencias_armonicas,
        'H1': H1,
        'n_armonicos_analizados': n_armonicos
    }

def calcular_correlacion_temporal(
    self,
    audio: np.ndarray,
    retardo_ms: float = 10.0
) -> float:
    """
    Calcula correlación temporal con retardo específico.

    Fórmula: r = Σ(xᵢ·yᵢ) / √(Σxᵢ² · Σyᵢ²)

    Interpretación:
    - r = 1.0: Correlación perfecta
    - r = 0.0: No hay correlación lineal
    - r < 0.70: Pérdida significativa de correlación
```

```
Args:
    audio: Señal de audio
    retardo_ms: Retardo en milisegundos

Returns:
    Coeficiente de correlación de Pearson [-1, 1]
"""
if len(audio) < 2:
    return 0.0

if retardo_ms < 0:
    raise ValueError("retardo_ms debe ser positivo")

retardo_muestras = int((retardo_ms / 1000.0) * self.frecuencia_muestreo)

if retardo_muestras >= len(audio) or retardo_muestras < 1:
    return 0.0

# Crear segmentos con offset temporal
x = audio[:-retardo_muestras]
y = audio[retardo_muestras:]

# Verificar varianza no nula
if len(x) < 2 or len(y) < 2:
    return 0.0
```

```
    std_x = np.std(x)
    std_y = np.std(y)

    if std_x < self.UMBRAL_SILENCIO or std_y < self.UMBRAL_SILENCIO:
        return 0.0

    # Calcular correlación de Pearson
    try:
        matriz_correlacion = np.corrcoef(x, y)
        r = matriz_correlacion[0, 1]

        if np.isnan(r) or np.isinf(r):
            r = 0.0
    except Exception:
        r = 0.0

    return float(r)

def protocolo_fse(
    self,
    frecuencia: float,
    n_replicas: int = 10,
    verbose: bool = True
) -> Dict:
    """
```

PROTOCOLO FSE: Fisión por Saturación de Energía

Procedimiento:

1. Generar tono puro a frecuencia especificada

2. Incrementar ganancia sistemáticamente (Δ = 0.1 dB)

3. Monitorear criterios duales:

  - Criterio A: THD > 1.0%

  - Criterio B: r < 0.70

4. Registrar E_fission cuando ambos criterios se cumplan simultáneamente

5. Repetir n_replicas veces

Args:

  frecuencia: Frecuencia a evaluar (Hz)

  n_replicas: Número de réplicas independientes

  verbose: Mostrar progreso detallado

Returns:

  dict: Resultados completos con estadísticas

"""

if verbose:

  print("\n" + "=" * 80)

  print(f" PROTOCOLO FSE - FISIÓN POR SATURACIÓN DE ENERGÍA")

  print(f" Frecuencia: {frecuencia} Hz | Réplicas: {n_replicas}")

  print("=" * 80 + "\n")

Contenedores de resultados

```
    valores_efission = []
    thd_en_fision = []
    correlacion_en_fision = []

    for replica in range(n_replicas):
      try:
        # Generar tono puro
        audio, _ = self.generar_tono_puro(frecuencia, duracion=5.0)

        # Variables de iteración
        ganancia_actual = 0.0
        fision_encontrada = False

        # Incremento sistemático de ganancia
        while ganancia_actual <= self.GANANCIA_MAXIMA_DB and not
fision_encontrada:
          # Aplicar ganancia
          ganancia_lineal = 10 ** (ganancia_actual / 20.0)
          audio_amplificado = audio * ganancia_lineal

          # Limitar para prevenir clipping
          audio_amplificado = np.clip(audio_amplificado, -0.999, 0.999)

          # CRITERIO A: Total Harmonic Distortion
          resultado_thd = self.calcular_thd(audio_amplificado, frecuencia)
          thd = resultado_thd['thd_porcentaje']
```

```
            # CRITERIO B: Correlación temporal
            correlacion = self.calcular_correlacion_temporal(audio_amplificado,
retardo_ms=10.0)

            # Lógica de decisión: AMBOS criterios deben cumplirse
            if thd > self.UMBRAL_THD_PORCENTAJE and correlacion <
self.UMBRAL_CORRELACION:
                valores_efission.append(ganancia_actual)
                thd_en_fision.append(thd)
                correlacion_en_fision.append(correlacion)
                fision_encontrada = True

                if verbose:
                    print(f" Réplica {replica + 1:2d}: E_fission = {ganancia_actual:5.2f} dB "
                        f"| THD = {thd:6.3f}% | r = {correlacion:5.3f} ✓")

            ganancia_actual += self.PASO_GANANCIA_DB

        if not fision_encontrada and verbose:
            print(f" Réplica {replica + 1:2d}: NO alcanzó umbral "
                f"(límite {self.GANANCIA_MAXIMA_DB} dB) ✗")

    except Exception as e:
        if verbose:
            print(f" Réplica {replica + 1:2d}: Error - {e} ✗")
```

```
            continue

        # Cálculo de estadísticas descriptivas
        if len(valores_efission) >= self.MINIMO_REPLICAS:
            estadisticas = self._calcular_estadisticas_descriptivas(valores_efission)

            resultados = {
                'frecuencia': frecuencia,
                'protocolo': 'FSE',
                'n_replicas_exitosas': len(valores_efission),
                'n_replicas_solicitadas': n_replicas,
                'tasa_exito': len(valores_efission) / n_replicas * 100,
                'valores_efission': valores_efission,
                'thd_en_fision': thd_en_fision,
                'correlacion_en_fision': correlacion_en_fision,
                **estadisticas,
                'unidad': 'dB'
            }

            # Test de normalidad (Shapiro-Wilk)
            if len(valores_efission) >= 3:
                try:
                    _, p_shapiro = shapiro(valores_efission)
                    resultados['test_normalidad'] = {
                        'metodo': 'Shapiro-Wilk',
                        'valor_p': float(p_shapiro),
```

```
                'es_normal': p_shapiro >= self.ALFA_SIGNIFICANCIA
            }
        except Exception:
            resultados['test_normalidad'] = None
    else:
        resultados = {
            'frecuencia': frecuencia,
            'protocolo': 'FSE',
            'n_replicas_exitosas': len(valores_efission),
            'n_replicas_solicitadas': n_replicas,
            'tasa_exito': len(valores_efission) / n_replicas * 100 if n_replicas > 0 else 0.0,
            'error': 'Datos insuficientes para estadísticas',
            'unidad': 'dB'
        }

    # Reporte de estadísticas
    if verbose:
        self._reportar_estadisticas(resultados, 'FSE')

    return resultados

def calcular_coherencia(
    self,
    pista_a: np.ndarray,
    pista_b: np.ndarray,
    frecuencia_objetivo: float,
```

```
    nsegmentos: int = 4096,
    solapamiento: Optional[int] = None
) -> Tuple[float, np.ndarray, np.ndarray]:
    """
    Calcula Magnitude-Squared Coherence en frecuencia específica.

    Fórmula: MSC(f) = |Pxy(f)|² / (Pxx(f) · Pyy(f))

    Propiedades:
    - Rango: 0 ≤ MSC(f) ≤ 1
    - MSC = 1: Coherencia perfecta
    - MSC = 0: Incoherencia total
    - MSC < 0.5: Pérdida significativa

    Args:
        pista_a, pista_b: Señales de audio a comparar
        frecuencia_objetivo: Frecuencia objetivo para extraer MSC
        nsegmentos: Longitud de segmento FFT
        solapamiento: Superposición de segmentos

    Returns:
        tuple: (valor_msc, frecuencias, coherencia_completa)
    """
    # Validación de entradas
    if len(pista_a) != len(pista_b):
        raise ValueError("Las pistas deben tener la misma longitud")
```

```
if len(pista_a) < nsegmentos:
    raise ValueError("Señal muy corta para análisis")

# Overlap por defecto: 50%
if solapamiento is None:
    solapamiento = nsegmentos // 2

if solapamiento >= nsegmentos:
    raise ValueError("Solapamiento debe ser menor que nsegmentos")

try:
    # Método de Welch para estimación espectral
    frecuencias, coherencia = signal.coherence(
        pista_a,
        pista_b,
        fs=self.frecuencia_muestreo,
        window='hann',
        nperseg=nsegmentos,
        noverlap=solapamiento,
        nfft=nsegmentos,
        detrend='constant'
    )
except Exception as e:
    raise RuntimeError(f"Error calculando coherencia: {e}")
```

```
    # Buscar índice más cercano a frecuencia objetivo
    idx = np.argmin(np.abs(frecuencias - frecuencia_objetivo))
    valor_msc = coherencia[idx]

    return float(valor_msc), frecuencias, coherencia

def protocolo_fdt(
    self,
    frecuencia: float,
    n_replicas: int = 10,
    verbose: bool = True
) -> Dict:
    """
    PROTOCOLO FDT: Fisión por Desincronización Temporal

    Procedimiento:
    1. Crear dos pistas idénticas del tono puro
    2. Aplicar offset temporal variable (Δt) a Pista B
    3. Calcular MSC en frecuencia fundamental
    4. Usar búsqueda binaria para encontrar Δt donde MSC ≈ 0.50
    5. Repetir n_replicas veces

    Args:
        frecuencia: Frecuencia a evaluar (Hz)
        n_replicas: Número de réplicas independientes
        verbose: Mostrar progreso detallado
```

```
Returns:
    dict: Resultados completos con estadísticas
"""
if verbose:
    print("\n" + "=" * 80)
    print(f" PROTOCOLO FDT - FISIÓN POR DESINCRONIZACIÓN TEMPORAL")
    print(f" Frecuencia: {frecuencia} Hz | Réplicas: {n_replicas}")
    print("=" * 80 + "\n")

# Contenedores de resultados
valores_tdelta = []
valores_msc = []
iteraciones_por_replica = []

# Pre-generar tono base
try:
    pista_base, _ = self.generar_tono_puro(frecuencia, duracion=5.0)

    # Ajuste de nivel para suma de dos pistas
    factor_ajuste = 10 ** (-3.0 / 20.0)
    pista_base = pista_base * factor_ajuste

except Exception as e:
    if verbose:
        print(f" ✗ Error generando tono base: {e}")
```

```
    return {
        'frecuencia': frecuencia,
        'protocolo': 'FDT',
        'n_replicas_exitosas': 0,
        'n_replicas_solicitadas': n_replicas,
        'tasa_exito': 0.0,
        'error': f'Error en generación de tono: {e}',
        'unidad': 'ms'
    }

# Ejecutar réplicas
for replica in range(n_replicas):
    try:
        # Crear copias independientes
        pista_a = pista_base.copy()
        pista_b = pista_base.copy()

        # Búsqueda binaria optimizada
        tmin = 0.5   # ms (límite inferior)
        tmax = 30.0  # ms (límite superior)
        tactual = (tmin + tmax) / 2.0

        max_iteraciones = 30
        convergencia_encontrada = False

        for iteracion in range(max_iteraciones):
```

```
            # Calcular offset en muestras
            offset_muestras = int((tactual / 1000.0) * self.frecuencia_muestreo)

            # Validación de límites
            if offset_muestras >= len(pista_b) or offset_muestras < 1:
                break

            # Crear Pista B con offset temporal
            pista_b_offset = np.zeros_like(pista_a)
            pista_b_offset[offset_muestras:] = pista_b[:-offset_muestras]

            # Calcular MSC en frecuencia fundamental
            try:
                msc, _, _ = self.calcular_coherencia(pista_a, pista_b_offset, frecuencia)
            except Exception as e:
                break

            # Criterio de convergencia
            if abs(msc - self.OBJETIVO_COHERENCIA) <
self.TOLERANCIA_CONVERGENCIA:
                valores_tdelta.append(tactual)
                valores_msc.append(msc)
                iteraciones_por_replica.append(iteracion + 1)
                convergencia_encontrada = True

                if verbose:
```

```
                print(f" Réplica {replica + 1:2d}: T_Δ = {tactual:6.2f} ms "
                      f"| MSC = {msc:.4f} | iter = {iteracion + 1:2d} ✓")
                break

            # Lógica de búsqueda binaria
            if msc > self.OBJETIVO_COHERENCIA:
                tmin = tactual  # Coherencia alta → aumentar delay
            else:
                tmax = tactual  # Coherencia baja → reducir delay

            tactual = (tmin + tmax) / 2.0

            # Criterio de convergencia secundario
            if (tmax - tmin) < self.TOLERANCIA_TIEMPO_MS:
                if abs(msc - self.OBJETIVO_COHERENCIA) <
self.TOLERANCIA_CONVERGENCIA * 2:
                    valores_tdelta.append(tactual)
                    valores_msc.append(msc)
                    iteraciones_por_replica.append(iteracion + 1)
                    convergencia_encontrada = True

                    if verbose:
                        print(f" Réplica {replica + 1:2d}: T_Δ = {tactual:6.2f} ms "
                              f"| MSC = {msc:.4f} | iter = {iteracion + 1:2d} (límite) ✓")
                    break
```

```
            if not convergencia_encontrada and verbose:
                print(f"  Réplica {replica + 1:2d}: NO convergió ✗")

        except Exception as e:
            if verbose:
                print(f"  Réplica {replica + 1:2d}: Error - {e} ✗")
            continue

    # Cálculo de estadísticas descriptivas
    if len(valores_tdelta) >= self.MINIMO_REPLICAS:
        estadisticas = self._calcular_estadisticas_descriptivas(valores_tdelta)

        resultados = {
            'frecuencia': frecuencia,
            'protocolo': 'FDT',
            'n_replicas_exitosas': len(valores_tdelta),
            'n_replicas_solicitadas': n_replicas,
            'tasa_exito': len(valores_tdelta) / n_replicas * 100,
            'valores_tdelta': valores_tdelta,
            'valores_msc': valores_msc,
            'iteraciones_por_replica': iteraciones_por_replica,
            **estadisticas,
            'iteraciones_promedio': float(np.mean(iteraciones_por_replica)),
            'msc_promedio': float(np.mean(valores_msc)),
            'msc_desviacion': float(np.std(valores_msc)),
            'unidad': 'ms'
```

```
    }

    # Test de normalidad
    if len(valores_tdelta) >= 3:
        try:
            _, p_shapiro = shapiro(valores_tdelta)
            resultados['test_normalidad'] = {
                'metodo': 'Shapiro-Wilk',
                'valor_p': float(p_shapiro),
                'es_normal': p_shapiro >= self.ALFA_SIGNIFICANCIA
            }
        except Exception:
            resultados['test_normalidad'] = None
else:
    resultados = {
        'frecuencia': frecuencia,
        'protocolo': 'FDT',
        'n_replicas_exitosas': len(valores_tdelta),
        'n_replicas_solicitadas': n_replicas,
        'tasa_exito': len(valores_tdelta) / n_replicas * 100 if n_replicas > 0 else 0.0,
        'error': 'Datos insuficientes para estadísticas',
        'unidad': 'ms'
    }

# Reporte de estadísticas
if verbose:
```

```
        self._reportar_estadisticas(resultados, 'FDT')

    return resultados

def _calcular_estadisticas_descriptivas(self, datos: List[float]) -> Dict:
    """
    Calcula estadísticas descriptivas básicas.

    Args:
        datos: Lista de valores numéricos

    Returns:
        dict: Estadísticas descriptivas
    """
    if len(datos) == 0:
        return {
            'media': 0.0,
            'desviacion': 0.0,
            'error_estandar': 0.0,
            'mediana': 0.0,
            'minimo': 0.0,
            'maximo': 0.0,
            'rango': 0.0,
            'q25': 0.0,
            'q75': 0.0,
            'rango_intercuartil': 0.0,
```

```
            'coeficiente_variacion': np.nan
        }

    datos_array = np.array(datos)

    return {
        'media': float(np.mean(datos_array)),
        'desviacion': float(np.std(datos_array, ddof=1)),
        'error_estandar': float(np.std(datos_array, ddof=1) / np.sqrt(len(datos_array))),
        'mediana': float(np.median(datos_array)),
        'minimo': float(np.min(datos_array)),
        'maximo': float(np.max(datos_array)),
        'rango': float(np.max(datos_array) - np.min(datos_array)),
        'q25': float(np.percentile(datos_array, 25)),
        'q75': float(np.percentile(datos_array, 75)),
        'rango_intercuartil': float(np.percentile(datos_array, 75) -
np.percentile(datos_array, 25)),
        'coeficiente_variacion': float((np.std(datos_array, ddof=1) /
np.mean(datos_array)) * 100)
                        if np.mean(datos_array) > self.UMBRAL_SILENCIO else np.nan
    }

  def _reportar_estadisticas(self, resultados: Dict, protocolo: str) -> None:
    """
    Imprime reporte de estadísticas en formato legible.

    Args:
```

```
        resultados: Diccionario con resultados del protocolo
        protocolo: Nombre del protocolo ('FSE' o 'FDT')
    """
    print("\n" + "–" * 80)
    print(f" ESTADÍSTICAS {protocolo} - {resultados['frecuencia']} Hz")
    print("–" * 80)

    if 'media' in resultados and resultados['n_replicas_exitosas'] >=
self.MINIMO_REPLICAS:
        print(f" Réplicas exitosas:
{resultados['n_replicas_exitosas']}/{resultados['n_replicas_solicitadas']} "
            f"({resultados['tasa_exito']:.1f}%)")
        print(f" Media (M):        {resultados['media']:.3f} {resultados['unidad']}")
        print(f" Desviación Est. (SD): {resultados['desviacion']:.3f}
{resultados['unidad']}")
        print(f" Error Est. (SEM):   {resultados['error_estandar']:.3f}
{resultados['unidad']}")
        print(f" Mediana:          {resultados['mediana']:.3f} {resultados['unidad']}")
        print(f" Rango:            [{resultados['minimo']:.2f}, {resultados['maximo']:.2f}]
{resultados['unidad']}")
        print(f" Rango Intercuartil:  {resultados['rango_intercuartil']:.3f}
{resultados['unidad']}")
        print(f" Coef. Variación:    {resultados['coeficiente_variacion']:.2f}%")

        if resultados.get('test_normalidad'):
            nt = resultados['test_normalidad']
            estado = "✓ Normal" if nt['es_normal'] else "✗ No normal"
            print(f" Test de normalidad:  {estado} (p={nt['valor_p']:.4f})")
```

```
        else:
            print(f" Datos insuficientes: {resultados['n_replicas_exitosas']}/{resultados['n_replicas_solicitadas']} réplicas")
            print(f" Se requieren mínimo {self.MINIMO_REPLICAS} réplicas para estadísticas")
        print("–" * 80 + "\n")

    def ejecutar_analisis_completo(
        self,
        frecuencia: float = 432,
        n_replicas: int = 10
    ) -> Dict:
        """
        Ejecuta análisis completo: FSE + FDT para una frecuencia.

        Args:
            frecuencia: Frecuencia a analizar
            n_replicas: Número de réplicas por protocolo

        Returns:
            dict: Resultados consolidados de ambos protocolos
        """
        print("\n" + "╔" + "=" * 78 + "╗")
        titulo = f" ANÁLISIS COMPLETO DE ESTABILIDAD ACÚSTICA - {frecuencia} Hz"
        padding = " " * (78 - len(titulo))
        print(f"║{titulo}{padding}║")
        print("╚" + "=" * 78 + "╝")
```

```
# Ejecutar Protocolo FSE
print("\n[1/2] Ejecutando Protocolo FSE...")
resultados_fse = self.protocolo_fse(frecuencia, n_replicas, verbose=True)

# Ejecutar Protocolo FDT
print("\n[2/2] Ejecutando Protocolo FDT...")
resultados_fdt = self.protocolo_fdt(frecuencia, n_replicas, verbose=True)

# Consolidar resultados
resultados_completos = {
    'frecuencia': frecuencia,
    'timestamp': datetime.now().isoformat(),
    'frecuencia_muestreo': self.frecuencia_muestreo,
    'nivel_rms_db': self.nivel_rms_db,
    'FSE': resultados_fse,
    'FDT': resultados_fdt,
    'resumen': {
        'E_fission_media': resultados_fse.get('media', np.nan),
        'E_fission_desviacion': resultados_fse.get('desviacion', np.nan),
        'E_fission_error_estandar': resultados_fse.get('error_estandar', np.nan),
        'T_delta_media': resultados_fdt.get('media', np.nan),
        'T_delta_desviacion': resultados_fdt.get('desviacion', np.nan),
        'T_delta_error_estandar': resultados_fdt.get('error_estandar', np.nan),
        'FSE_tasa_exito': resultados_fse.get('tasa_exito', 0.0),
        'FDT_tasa_exito': resultados_fdt.get('tasa_exito', 0.0)
```

```
        }
    }

    # Almacenar en caché
    self.resultados[frecuencia] = resultados_completos

    print("\n" + "╔" + "=" * 78 + "╗")
    print("║" + " " * 20 + "✓ ANÁLISIS COMPLETADO EXITOSAMENTE" + " " * 24 + "║")
    print("╚" + "=" * 78 + "╝\n")

    return resultados_completos

def comparar_frecuencias(
    self,
    frecuencia1: float = 432,
    frecuencia2: float = 440,
    n_replicas: int = 10,
    test_estadistico: bool = True,
    verbose: bool = True
) -> Dict:
    """
    Compara estabilidad acústica entre dos frecuencias con análisis estadístico.

    Args:
        frecuencia1: Primera frecuencia
        frecuencia2: Segunda frecuencia
```

```
    n_replicas: Número de réplicas por frecuencia
    test_estadistico: Realizar pruebas estadísticas
    verbose: Mostrar resultados detallados

Returns:
    dict: Resultados comparativos con tests estadísticos
"""
print("\n" + "╔" + "=" * 78 + "╗")
titulo = f" ANÁLISIS COMPARATIVO: {frecuencia1} Hz vs {frecuencia2} Hz"
padding = " " * (78 - len(titulo))
print(f"║{titulo}{padding}║")
print("╚" + "=" * 78 + "╝\n")

# Ejecutar análisis para ambas frecuencias si no existen
if frecuencia1 not in self.resultados:
    print(f"\n[1/2] Analizando {frecuencia1} Hz...")
    self.ejecutar_analisis_completo(frecuencia1, n_replicas)

if frecuencia2 not in self.resultados:
    print(f"\n[2/2] Analizando {frecuencia2} Hz...")
    self.ejecutar_analisis_completo(frecuencia2, n_replicas)

resultados1 = self.resultados[frecuencia1]
resultados2 = self.resultados[frecuencia2]

comparacion = {
```

```
        'frecuencia1': frecuencia1,
        'frecuencia2': frecuencia2,
        'timestamp': datetime.now().isoformat(),
        'n_replicas': n_replicas
    }

    # ===== COMPARACIÓN PROTOCOLO FSE =====
    if verbose:
        print("\n" + "–" * 80)
        print(" COMPARACIÓN PROTOCOLO FSE (Energía de Fisión)")
        print("–" * 80)

    fse1_media = resultados1['FSE'].get('media', np.nan)
    fse2_media = resultados2['FSE'].get('media', np.nan)
    fse1_desviacion = resultados1['FSE'].get('desviacion', np.nan)
    fse2_desviacion = resultados2['FSE'].get('desviacion', np.nan)

    if not (np.isnan(fse1_media) or np.isnan(fse2_media)):
        diferencia = fse1_media - fse2_media
        diferencia_porcentual = (diferencia / fse2_media * 100) if fse2_media != 0 else
np.nan
        cohens_d = (fse1_media - fse2_media) / np.sqrt((fse1_desviacion**2 +
fse2_desviacion**2) / 2)
    else:
        diferencia = np.nan
        diferencia_porcentual = np.nan
        cohens_d = np.nan
```

```
        comparacion['FSE'] = {
            f'{frecuencia1}Hz': {
                'media_db': float(fse1_media),
                'desviacion_db': float(fse1_desviacion),
                'n': resultados1['FSE'].get('n_replicas_exitosas', 0)
            },
            f'{frecuencia2}Hz': {
                'media_db': float(fse2_media),
                'desviacion_db': float(fse2_desviacion),
                'n': resultados2['FSE'].get('n_replicas_exitosas', 0)
            },
            'diferencia_db': float(diferencia),
            'diferencia_porcentual': float(diferencia_porcentual),
            'cohens_d': float(cohens_d)
        }

        if verbose:
            print(f" {frecuencia1} Hz: M = {fse1_media:.3f} ± {resultados1['FSE'].get('error_estandar', np.nan):.3f} dB")
            print(f" {frecuencia2} Hz: M = {fse2_media:.3f} ± {resultados2['FSE'].get('error_estandar', np.nan):.3f} dB")
            print(f" Diferencia: {diferencia:.3f} dB ({diferencia_porcentual:.2f}%)")
            print(f" Cohen's d:  {cohens_d:.3f}")

        # ===== COMPARACIÓN PROTOCOLO FDT =====
        if verbose:
```

```
    print("\n" + "–" * 80)
    print(" COMPARACIÓN PROTOCOLO FDT (Umbral Temporal)")
    print("–" * 80)

  fdt1_media = resultados1['FDT'].get('media', np.nan)
  fdt2_media = resultados2['FDT'].get('media', np.nan)
  fdt1_desviacion = resultados1['FDT'].get('desviacion', np.nan)
  fdt2_desviacion = resultados2['FDT'].get('desviacion', np.nan)

  if not (np.isnan(fdt1_media) or np.isnan(fdt2_media)):
    diferencia = fdt1_media - fdt2_media
    diferencia_porcentual = (diferencia / fdt2_media * 100) if fdt2_media != 0 else
np.nan
    cohens_d = (fdt1_media - fdt2_media) / np.sqrt((fdt1_desviacion**2 +
fdt2_desviacion**2) / 2)
  else:
    diferencia = np.nan
    diferencia_porcentual = np.nan
    cohens_d = np.nan

  comparacion['FDT'] = {
    f'{frecuencia1}Hz': {
      'media_ms': float(fdt1_media),
      'desviacion_ms': float(fdt1_desviacion),
      'n': resultados1['FDT'].get('n_replicas_exitosas', 0)
    },
    f'{frecuencia2}Hz': {
```

```
            'media_ms': float(fdt2_media),
            'desviacion_ms': float(fdt2_desviacion),
            'n': resultados2['FDT'].get('n_replicas_exitosas', 0)
        },
        'diferencia_ms': float(diferencia),
        'diferencia_porcentual': float(diferencia_porcentual),
        'cohens_d': float(cohens_d)
    }

    if verbose:
        print(f" {frecuencia1} Hz: M = {fdt1_media:.3f} ± {resultados1['FDT'].get('error_estandar', np.nan):.3f} ms")
        print(f" {frecuencia2} Hz: M = {fdt2_media:.3f} ± {resultados2['FDT'].get('error_estandar', np.nan):.3f} ms")
        print(f" Diferencia: {diferencia:.3f} ms ({diferencia_porcentual:.2f}%)")
        print(f" Cohen's d:  {cohens_d:.3f}")

    # ===== PRUEBAS ESTADÍSTICAS =====
    if test_estadistico:
        comparacion['tests_estadisticos'] = {}

        # Test para FSE
        if ('valores_efission' in resultados1['FSE'] and
            'valores_efission' in resultados2['FSE'] and
            len(resultados1['FSE']['valores_efission']) >= 3 and
            len(resultados2['FSE']['valores_efission']) >= 3):
```

```
        valores1 = np.array(resultados1['FSE']['valores_efission'])
        valores2 = np.array(resultados2['FSE']['valores_efission'])

        try:
            # Verificar normalidad
            _, p_normalidad1 = shapiro(valores1)
            _, p_normalidad2 = shapiro(valores2)

            es_normal1 = p_normalidad1 >= self.ALFA_SIGNIFICANCIA
            es_normal2 = p_normalidad2 >= self.ALFA_SIGNIFICANCIA
            ambas_normales = es_normal1 and es_normal2

            if verbose:
                print(f"\n FSE - Tests de normalidad:")
                print(f"   {frecuencia1} Hz: p = {p_normalidad1:.4f} {'(Normal ✓)' if es_normal1 else '(No normal ✗)'}")
                print(f"   {frecuencia2} Hz: p = {p_normalidad2:.4f} {'(Normal ✓)' if es_normal2 else '(No normal ✗)'}")

            if ambas_normales:
                # t-test paramétrico
                t_stat, p_valor = ttest_ind(valores1, valores2)
                nombre_test = 't-test independiente'
            else:
                # Mann-Whitney U test (no paramétrico)
                u_stat, p_valor = mannwhitneyu(valores1, valores2, alternative='two-sided')
```

```
                nombre_test = 'Mann-Whitney U'

            comparacion['tests_estadisticos']['FSE'] = {
                'test': nombre_test,
                'estadistico': float(t_stat if ambas_normales else u_stat),
                'valor_p': float(p_valor),
                'significativo': p_valor < self.ALFA_SIGNIFICANCIA
            }

            if verbose:
                print(f"\n FSE - {nombre_test}:")
                print(f"   p = {p_valor:.4f}")
                if p_valor < self.ALFA_SIGNIFICANCIA:
                    print(f"   ✓ Diferencia estadísticamente SIGNIFICATIVA (p < {self.ALFA_SIGNIFICANCIA})")
                else:
                    print(f"   ✗ No hay diferencia significativa (p ≥ {self.ALFA_SIGNIFICANCIA})")

        except Exception as e:
            comparacion['tests_estadisticos']['FSE'] = {'error': str(e)}

    # Test para FDT
    if ('valores_tdelta' in resultados1['FDT'] and
        'valores_tdelta' in resultados2['FDT'] and
        len(resultados1['FDT']['valores_tdelta']) >= 3 and
        len(resultados2['FDT']['valores_tdelta']) >= 3):
```

```
        valores1 = np.array(resultados1['FDT']['valores_tdelta'])
        valores2 = np.array(resultados2['FDT']['valores_tdelta'])

        try:
            # Verificar normalidad
            _, p_normalidad1 = shapiro(valores1)
            _, p_normalidad2 = shapiro(valores2)

            es_normal1 = p_normalidad1 >= self.ALFA_SIGNIFICANCIA
            es_normal2 = p_normalidad2 >= self.ALFA_SIGNIFICANCIA
            ambas_normales = es_normal1 and es_normal2

            if verbose:
                print(f"\n FDT - Tests de normalidad:")
                print(f"   {frecuencia1} Hz: p = {p_normalidad1:.4f} {'(Normal ✓)' if es_normal1 else '(No normal ✗)'}")
                print(f"   {frecuencia2} Hz: p = {p_normalidad2:.4f} {'(Normal ✓)' if es_normal2 else '(No normal ✗)'}")

            if ambas_normales:
                # t-test paramétrico
                t_stat, p_valor = ttest_ind(valores1, valores2)
                nombre_test = 't-test independiente'
            else:
                # Mann-Whitney U test
```

```
                u_stat, p_valor = mannwhitneyu(valores1, valores2, alternative='two-
sided')
                nombre_test = 'Mann-Whitney U'

            comparacion['tests_estadisticos']['FDT'] = {
                'test': nombre_test,
                'estadistico': float(t_stat if ambas_normales else u_stat),
                'valor_p': float(p_valor),
                'significativo': p_valor < self.ALFA_SIGNIFICANCIA
            }

            if verbose:
                print(f"\n FDT - {nombre_test}:")
                print(f"   p = {p_valor:.4f}")
                if p_valor < self.ALFA_SIGNIFICANCIA:
                    print(f"   ✓ Diferencia estadísticamente SIGNIFICATIVA (p <
{self.ALFA_SIGNIFICANCIA})")
                else:
                    print(f"   ✗ No hay diferencia significativa (p ≥
{self.ALFA_SIGNIFICANCIA})")

        except Exception as e:
            comparacion['tests_estadisticos']['FDT'] = {'error': str(e)}

    if verbose:
        print("–" * 80)
```

```
    return comparacion


#
================================================================================
# FUNCIONES DE EJEMPLO
#
================================================================================

def ejemplo_completo():
    """
    Ejemplo completo de uso del analizador para investigación académica.
    """
    try:
        print("\n" + "╔" + "═" * 78 + "╗")
        print("║" + " " * 20 + "EJEMPLO DE USO COMPLETO" + " " * 34 + "║")
        print("╚" + "═" * 78 + "╝\n")

        # Crear instancia del analizador
        analizador = AnalizadorEstabilidadAcustica(frecuencia_muestreo=48000,
nivel_rms_db=-18.0)

        # Análisis individual: 432 Hz
        print("\n" + "═" * 80)
        print(" PASO 1: Análisis de 432 Hz")
        print("═" * 80)
```

```
    resultados_432 = analizador.ejecutar_analisis_completo(frecuencia=432,
n_replicas=5)


    # Comparación: 432 Hz vs 440 Hz
    print("\n" + "=" * 80)
    print(" PASO 2: Comparación 432 Hz vs 440 Hz")
    print("=" * 80)
    comparacion = analizador.comparar_frecuencias(
      frecuencia1=432,
      frecuencia2=440,
      n_replicas=5,
      test_estadistico=True,
      verbose=True
    )


    # Guardar resultados
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    nombre_archivo = f'resultados_estabilidad_acustica_{timestamp}.json'


    datos_exportar = {
      'metadata': {
        'estudio': 'Quantum Sonorology',
        'autor': 'Julio Alberto Solis Trejo (Liah Steer)',
        'version_software': '2.1',
        'fecha_analisis': datetime.now().isoformat()
      },
```

```
        'resultados_432hz': resultados_432,
        'comparacion_432_440': comparacion
    }

    with open(nombre_archivo, 'w', encoding='utf-8') as f:
        json.dump(datos_exportar, f, indent=2, ensure_ascii=False)

    print(f"\n✓ Resultados exportados: {nombre_archivo}")

    # Resumen
    print("\n" + "╔" + "═" * 78 + "╗")
    print("║" + " " * 20 + "RESUMEN DE RESULTADOS" + " " * 35 + "║")
    print("╠" + "═" * 78 + "╣")
    print("║  Frecuencia 432 Hz:                                                          ║")
    print(f"║    • FSE - E_fission: {resultados_432['resumen']['E_fission_media']:.2f} ± {resultados_432['resumen']['E_fission_error_estandar']:.2f} dB  ║")
    print(f"║    • FDT - T_Δ:     {resultados_432['resumen']['T_delta_media']:.2f} ± {resultados_432['resumen']['T_delta_error_estandar']:.2f} ms  ║")

    if 'tests_estadisticos' in comparacion:
        print("║                                                                              ║")
        print("║  Comparación 432 Hz vs 440 Hz:                                               ║")

        if 'FSE' in comparacion['tests_estadisticos']:
            test_fse = comparacion['tests_estadisticos']['FSE']
            if test_fse.get('significativo'):
                print("║    • FSE: Diferencia SIGNIFICATIVA (p < 0.05)                        ║")
```

```
            else:
                print("║  • FSE: No hay diferencia significativa                    ║")

        if 'FDT' in comparacion['tests_estadisticos']:
            test_fdt = comparacion['tests_estadisticos']['FDT']
            if test_fdt.get('significativo'):
                print("║  • FDT: Diferencia SIGNIFICATIVA (p < 0.05)                ║")
            else:
                print("║  • FDT: No hay diferencia significativa                    ║")

        print("╚" + "═" * 78 + "╝\n")

        return analizador, resultados_432, comparacion

    except Exception as e:
        print(f"\n✗ Error durante el análisis: {e}")
        import traceback
        traceback.print_exc()
        return None, None, None


def ejemplo_rapido(frecuencia=432, replicas=3):
    """
    Ejemplo rápido para demostración básica.

    Args:
```

```
    frecuencia: Frecuencia a analizar
    replicas: Número de réplicas (reducido para demo rápida)
    """
    print("\n" + "╔" + "═" * 78 + "╗")
    print("║" + " " * 25 + "ANÁLISIS RÁPIDO" + " " * 38 + "║")
    print("╚" + "═" * 78 + "╝\n")

    analizador = AnalizadorEstabilidadAcustica()
    resultados = analizador.ejecutar_analisis_completo(frecuencia=frecuencia,
n_replicas=replicas)

    print(f"\n✓ Análisis rápido completado para {frecuencia} Hz")
    print(f"  FSE - E_fission: {resultados['resumen']['E_fission_media']:.2f} dB")
    print(f"  FDT - T_Δ: {resultados['resumen']['T_delta_media']:.2f} ms\n")

    return analizador, resultados


#
================================================================
========
# EJECUCIÓN PRINCIPAL
#
================================================================
========

if __name__ == "__main__":
    print("""
```

```
╔═══════════════════════════════════════════════════════════════════════════════╗
║                                                                               ║
║   ANALIZADOR DE ESTABILIDAD ACÚSTICA - PROTOCOLO COMPLETO FSE/FDT v2.1        ║
║                                                                               ║
║   Basado en: "Quantum Sonorology: Sound Fission and the Test of               ║
║               Biophysical Stability of Vibrating Matter at 432/440 Hz"        ║
║                                                                               ║
║   Autor del estudio: Julio Alberto Solis Trejo (Liah Steer)                   ║
║   Implementación: Código optimizado para publicación académica                ║
║   Licencia: CC BY 4.0                                                         ║
║   Versión: 2.1 (Corregida y optimizada)                                       ║
║                                                                               ║
║   CARACTERÍSTICAS:                                                            ║
║   ✓ Protocolos FSE y FDT implementados                                        ║
║   ✓ Análisis estadístico completo                                             ║
║   ✓ Exportación de resultados en múltiples formatos                           ║
║   ✓ Documentación exhaustiva en español                                       ║
║   ✓ Código validado y libre de errores                                        ║
║                                                                               ║
╚═══════════════════════════════════════════════════════════════════════════════╝
    """)

    # Ejecutar ejemplo completo
    print("\nEjecutando análisis completo...\n")
```

```
    analizador, resultados, comparacion = ejemplo_completo()


    print("\n" + "╔" + "═" * 78 + "╗")
    print("║" + " " * 20 + "PROGRAMA FINALIZADO CORRECTAMENTE" + " " * 26 + "║")
    print("╚" + "═" * 78 + "╝\n")
```