```
import numpy as np
from scipy import integrate, special
import argparse
import warnings
import time
warnings.filterwarnings('ignore')

# ============================================================================
# CHRONOMÉTRAGE
# ============================================================================
tic = time.time()

def create_second_derivative_matrix(n: int) -> np.ndarray:
    """Crée la matrice de dérivée seconde avec conditions de Dirichlet."""
    h = 2.0 / (n - 1)
    D2 = np.zeros((n, n))
    for i in range(1, n-1):
        D2[i, i-1] = 1.0 / h**2
        D2[i, i] = -2.0 / h**2
        D2[i, i+1] = 1.0 / h**2
    D2[0, 0] = 1.0
    D2[-1, -1] = 1.0
    return D2

def omega_f64(t):
    """ω(t) = t·e^t / (1 + e^t)² - le noyau de la fonction eta complétée."""
    if isinstance(t, np.ndarray):
        result = np.zeros_like(t)
        mask = t > 500.0
        result[mask] = t[mask] * np.exp(-t[mask])
        mask2 = ~mask
        et = np.exp(t[mask2])
        result[mask2] = t[mask2] * et / (1.0 + et)**2
        return result
    else:
        if t > 500.0:
            return t * np.exp(-t)
        et = np.exp(t)
        return t * et / (1.0 + et)**2

def truncated_lambda(s_re: float, s_im: float, lambda_val: float, n_quad: int)
-> complex:
    """Λ_λ(s) = ∫_{λ⁻¹}^{λ} u^{s-1} · ω(u) du"""
    a = 1.0 / lambda_val
    b = lambda_val

    nodes, weights = np.polynomial.legendre.leggauss(n_quad)
    mid = 0.5 * (a + b)
    half = 0.5 * (b - a)

    u = mid + half * nodes
    w = weights * half

    ln_u = np.log(u)
    u_pow = u**(s_re - 1.0)
    phase = s_im * ln_u
    omega_u = omega_f64(u)

    integrand = u_pow * np.exp(1j * phase) * omega_u
    return complex(np.sum(w * integrand))

def xi_weighted_mellin(s_re: float, s_im: float, lambda_val: float, xi:
np.ndarray, n_modes: int, n_quad: int) -> complex:
    """G(s) = ∫ f_λ(u)·ω(u)·u^{s-1} du"""
```

```
    a = 1.0 / lambda_val
    b = lambda_val
    L = 2.0 * np.log(lambda_val)
    inv_sqrt_L = 1.0 / np.sqrt(L)

    nodes, weights = np.polynomial.legendre.leggauss(n_quad)
    mid = 0.5 * (a + b)
    half = 0.5 * (b - a)

    u = mid + half * nodes
    w = weights * half

    # xi est de taille n_modes : xi[0] = ξ₀
    xi_0 = xi[0]

    phase_base = 2.0 * np.pi * np.log(lambda_val * u) / L
    f_val = xi_0 * np.ones_like(u)

    for n in range(1, n_modes):
        xi_n = xi[n]
        f_val += 2.0 * xi_n * np.cos(n * phase_base)

    f_val *= inv_sqrt_L

    ln_u = np.log(u)
    u_pow = u**(s_re - 1.0)
    phase = s_im * ln_u
    omega_u = omega_f64(u)

    integrand = f_val * u_pow * np.exp(1j * phase) * omega_u
    return complex(np.sum(w * integrand))

def find_zeros(func, t_min: float, t_max: float, n_scan: int, bisect_iter: int =
30) -> np.ndarray:
    """Trouve les zéros par détection de changement de signe de la partie
réelle."""
    t_values = np.linspace(t_min, t_max, n_scan)
    zeros = []

    print("  Calcul de Re(F(1/2 + it))...")
    f_real = []
    for i, t in enumerate(t_values):
        if i % 500 == 0:
            print(f"    Progression: {i}/{len(t_values)}")
        try:
            val = func(0.5, t)
            f_real.append(val.real)
        except:
            f_real.append(0.0)

    f_real = np.array(f_real)

    threshold = 1e-8 * np.max(np.abs(f_real))
    if threshold < 1e-12:
        threshold = 1e-12

    print("  Détection des changements de signe...")

    for i in range(len(f_real) - 1):
        if abs(f_real[i]) < threshold and abs(f_real[i+1]) < threshold:
            continue

        if f_real[i] * f_real[i+1] < 0:
            denom = f_real[i+1] - f_real[i]
```

```
            if abs(denom) > 1e-15:
                t_zero = t_values[i] - f_real[i] * (t_values[i+1] - t_values[i])
/ denom

                a = t_values[i]
                b = t_values[i+1]
                fa = f_real[i]

                for _ in range(bisect_iter):
                    mid = 0.5 * (a + b)
                    try:
                        fm = func(0.5, mid).real
                    except:
                        fm = 0.0
                    if fa * fm < 0:
                        b = mid
                    else:
                        a = mid
                        fa = fm

                t_zero = 0.5 * (a + b)

                if not np.isnan(t_zero) and not np.isinf(t_zero) and t_min <
t_zero < t_max:
                    zeros.append(t_zero)

    if len(zeros) > 1:
        zeros.sort()
        unique = [zeros[0]]
        for z in zeros[1:]:
            if z - unique[-1] > 0.02:
                unique.append(z)
        zeros = unique

    print(f"  {len(zeros)} zéro(s) trouvé(s)")
    return np.array(zeros)

def diagnose_xi_weighted_mellin(lambda_val: float, xi: np.ndarray, n_modes: int,
n_quad: int = 100):
    """
    Diagnostic : vérifie que f_λ est correctement reconstruite.
    xi est de taille n_modes : xi[0] = ξ₀
    """
    print("\n" + "="*70)
    print("🔍 DIAGNOSTIC DE f_λ (reconstruction de ξ_λ)")
    print("="*70)

    print(f"  xi.shape = {xi.shape}, n_modes = {n_modes}")

    a = 1.0 / lambda_val
    b = lambda_val
    L = 2.0 * np.log(lambda_val)
    inv_sqrt_L = 1.0 / np.sqrt(L)

    u_test = np.linspace(a, b, n_quad)

    # CORRECTION ICI : xi[0] au lieu de xi[n_modes]
    xi_0 = xi[0]

    phase_base = 2.0 * np.pi * np.log(lambda_val * u_test) / L
    f_val = xi_0 * np.ones_like(u_test)

    for n in range(1, n_modes):
        xi_n = xi[n]
```

```
        f_val += 2.0 * xi_n * np.cos(n * phase_base)

    f_val *= inv_sqrt_L

    print(f"\n  📊 Statistiques de f_λ sur [{a:.4f}, {b:.4f}]:")
    print(f"    min  = {np.min(f_val):.6e}")
    print(f"    max  = {np.max(f_val):.6e}")
    print(f"    mean = {np.mean(f_val):.6e}")
    print(f"    std  = {np.std(f_val):.6e}")

    max_imag = np.max(np.abs(np.imag(f_val)))
    print(f"\n  🔬 Partie imaginaire max = {max_imag:.2e}")
    if max_imag > 1e-10:
        print("    ⚠ Attention : f_λ a une partie imaginaire non-
négligeable !")
    else:
        print("    ✅ f_λ est bien réelle")

    if np.max(np.abs(f_val)) < 1e-12:
        print("\n  ❌ ERREUR CRITIQUE : f_λ est NULLE !")
    else:
        print("\n  ✅ f_λ est non-nulle")
        if np.std(f_val) < 1e-12:
            print("    ⚠ f_λ est presque constante (faible variation)")
        else:
            print("    ✅ f_λ a des variations significatives")

    print("="*70 + "\n")
    return f_val

def load_riemann_zeros(n: int) -> np.ndarray:
    """Charge les premiers zéros de Riemann."""
    ref_zeros = np.array([
        14.134725141734693790457251983562470270784257,
        21.022039638771554992628479593896902777334340,
        25.010857580145688763213790992562821818659549,
        30.424876125859513210311897530584091320181560,
        32.935061587739189690662368964074903488812715,
        37.586178158825671257217763480705332821405597,
        40.918719012147495789032262362803715070735160,
        43.327073280914999519786273681687136215817434,
        48.005150881167159727942472280669109472060045,
        49.773832477672302181916784600564261058637318,
        52.970321477714460644147358607231115108585521,
        56.446247697063394804367759673400870257641919,
        59.347044002602353079653198169136871147090779,
        60.831778524609809844259901824511607510003568,
        65.112544048081600660935882106460848345885337,
        67.079810529494173714478828975345835174579626,
        69.546401711173979252758068805644252038509189,
        72.067157674481907582522179711673245125978889,
        75.704690699083933168587246793440864135581257,
        77.144840068874805372682664856394295866260905
    ])
    return ref_zeros[:n]

def cmd_mellin_compare(lambda_sq: int, n_modes: int, t_min: float, t_max: float,
n_scan: int, n_quad: int):
    """Test 3: Compare CCM vs troncature naive de Mellin."""
    print(f"\n{'='*70}\nTEST 3: Mellin Compare - CCM vs Troncature
Naive\n{'='*70}\n")
    lambda_val = np.sqrt(lambda_sq)

    print("Calcul de ξ_λ...")
```

```
    D2_xi = create_second_derivative_matrix(n_modes)
    x = np.linspace(-1, 1, n_modes)
    V_xi = np.diag(1.0 / (1 + x**2)) / (lambda_val**2)
    A = D2_xi + V_xi
    eigenvalues, eigenvectors = np.linalg.eigh(A)
    xi = eigenvectors[:, np.argmin(eigenvalues)]
    xi /= np.linalg.norm(xi, np.inf)

    print(f"ξ_λ calculé: λ²={lambda_sq}, λ={lambda_val:.6f}, N={n_modes}")
    print(f"xi.shape = {xi.shape}")
    print(f"Scan: Re(s)=1/2, Im(s) ∈ [{t_min:.1f}, {t_max:.1f}]")
    print(f"Points: quadrature={n_quad}, scan={n_scan}\n")

    # Diagnostic
    diagnose_xi_weighted_mellin(lambda_val, xi, n_modes, n_quad)

    ref_zeros = load_riemann_zeros(50)

    print("Recherche des zéros de Λ_λ (tronqué)...")
    l_zeros = find_zeros(
        lambda s_re, s_im: truncated_lambda(s_re, s_im, lambda_val, n_quad),
        t_min, t_max, n_scan
    )

    print("\nRecherche des zéros de G (pondéré)...")
    g_zeros = find_zeros(
        lambda s_re, s_im: xi_weighted_mellin(s_re, s_im, lambda_val, xi,
n_modes, n_quad),
        t_min, t_max, n_scan
    )

    print("\n" + "="*70)
    print("ZÉROS CALCULÉS PAR LE PROGRAMME")
    print("="*70)

    print(f"\nΛ_λ (tronqué) - {len(l_zeros)} zéro(s) trouvé(s):")
    if len(l_zeros) > 0:
        show = min(20, len(l_zeros))
        for i in range(show):
            print(f"  z_{i+1:>2} = {l_zeros[i]:>15.10f}")
        if len(l_zeros) > 20:
            print(f"  ... et {len(l_zeros) - 20} autre(s)")
    else:
        print("  ⚠ AUCUN ZÉRO TROUVÉ !")

    print(f"\nG (pondéré) - {len(g_zeros)} zéro(s) trouvé(s):")
    if len(g_zeros) > 0:
        show = min(20, len(g_zeros))
        for i in range(show):
            print(f"  z_{i+1:>2} = {g_zeros[i]:>15.10f}")
        if len(g_zeros) > 20:
            print(f"  ... et {len(g_zeros) - 20} autre(s)")
    else:
        print("  ⚠ AUCUN ZÉRO TROUVÉ !")

    print("\nZéros de Riemann (référence) - premiers 10:")
    for i, z in enumerate(ref_zeros[:10]):
        print(f"  γ_{i+1:>2} = {z:>15.10f}")

    print("\n" + "="*70 + "\n")

    if len(l_zeros) > 0 or len(g_zeros) > 0:
        print("COMPARAISON AVEC LES ZÉROS DE RIEMANN:")
        print(f"{'k':>5} {'Λ_λ zéro':>22} {'G zéro':>22} {'Riemann':>22} {'Err
```

```
Λ_λ':>16} {'Err G':>16}")
        print("-" * 108)

        matched_l = 0
        matched_g = 0

        for i, rz in enumerate(ref_zeros[:20]):
            if rz < t_min or rz > t_max:
                continue

            l_idx = np.argmin(np.abs(l_zeros - rz)) if len(l_zeros) > 0 else -1
            g_idx = np.argmin(np.abs(g_zeros - rz)) if len(g_zeros) > 0 else -1

            l_zero = l_zeros[l_idx] if l_idx >= 0 and len(l_zeros) > 0 else
float('nan')
            g_zero = g_zeros[g_idx] if g_idx >= 0 and len(g_zeros) > 0 else
float('nan')

            l_error = abs(l_zero - rz) if not np.isnan(l_zero) else float('nan')
            g_error = abs(g_zero - rz) if not np.isnan(g_zero) else float('nan')

            if not np.isnan(l_error) and l_error < 1.0:
                matched_l += 1
            if not np.isnan(g_error) and g_error < 1.0:
                matched_g += 1

            if not np.isnan(l_error) or not np.isnan(g_error):
                print(f"{i+1:>5} {l_zero:>22.15f} {g_zero:>22.15f} {rz:>22.15f}
{l_error:>16.6e} {g_error:>16.6e}")

        print(f"\nZéros bien identifiés (erreur < 1): Λ_λ={matched_l},
G={matched_g}")
    else:
        print("❌ AUCUN ZÉRO TROUVÉ")

# ============================================================================
# MAIN
# ============================================================================
def main():
    parser = argparse.ArgumentParser(
        description='CCM Mellin Compare - Version CORRIGÉE',
        formatter_class=argparse.RawDescriptionHelpFormatter
    )
    subparsers = parser.add_subparsers(dest='command', help='Commande à
exécuter')

    m = subparsers.add_parser('mellin-compare', help='Comparaison Mellin')
    m.add_argument('--lambda-sq', type=int, default=50)
    m.add_argument('--n-modes', type=int, default=100)
    m.add_argument('--t-min', type=float, default=5.0)
    m.add_argument('--t-max', type=float, default=55.0)
    m.add_argument('--n-scan', type=int, default=2000)
    m.add_argument('--n-quad', type=int, default=500)

    args = parser.parse_args()
    if args.command is None:
        parser.print_help()
        return

    if args.command == 'mellin-compare':
        cmd_mellin_compare(args.lambda_sq, args.n_modes, args.t_min, args.t_max,
args.n_scan, args.n_quad)

    tac = time.time()
```

```
    print(f"\n{'='*70}")
    print(f"⏱️  TEMPS D'EXÉCUTION TOTAL : {tac - tic:.2f} secondes ({ (tac -
tic)/60:.2f} minutes)")
    print(f"{'='*70}")

if __name__ == '__main__':
    main()
```