Comprendre comment un programme récent calcule les zéros de zeta en le faisant traduire en python par les ia

Denise Vella-Chemla, juillet 2026

Tout est dans le titre : je cherche à comprendre, du mieux qu'il est possible compte-tenu de mes moyens intellectuels, les dernières avancées en matière de programmation dont a parlé Alain Connes dans les dernières conférences qu'il a postées sur son site en juillet 2026.

Pour cela, j'ai cherché le github de Ronnie Andrews cité, et j'ai essayé de faire traduire le Rust en python par les ia. Cela n'a pas été sans peine. Mon but n'était pas d'obtenir des calculs aussi précis que ceux de Rust mais de repérer les calculs dans un code (car je n'avais pas réussi à programmer ces calculs plus tôt dans l'année). Peu d'ia aboutissent à la fourniture d'un code utilisable.

Je vais fournir ci-dessous les programmes python de deepseek et de chatgpt et leur résultat.

Voici une petite anecdote concernant l'ia deepseek : j'ai été obligée de demander à deepseek de modifier notamment son dernier programme car il avait utilisé une méthode de contournement : pour que les zéros qu'il avait trouvés coïncident au mieux avec les zéros effectifs de ζ, deepseek écrivait certains zéros trouvés en double. J'ai dû lui demander d'être honnête et de me fournir y compris les zéros qu'il trouvait et qui ne sont pas des zéros de ζ ou qui s'intercalent. Il resterait bien sûr à comprendre ces programmes, à les comparer aux formules des articles qu'ils sont censés implémenter, et à les modifier si on peut le faire[1]. Dans le résultat du programme de deepseek ci-dessous, on a supprimé la colonne G zéro, intégralement à *nan* (not a number) ainsi que la colonne de l'erreur correspondante.

Concernant le programme fourni par chatgpt, on verra que les zéros réels de ζ pourraient être considérés comme apparaissant dans les résultats à partir du treizième zéro et suivants, mais alors, ils ne concordent plus du tout dès le 19ème zéro (56 en regard de 40 puis 176 en regard de 43).

Programme de l'ia chatgpt

```
"""
ccm_zeros.py
Construction pedagogique :
    lambda au carre
       ↓
    matrice de Weil
       ↓
    plus petit vecteur propre xi_lambda
       ↓
    fonction rationnelle R(z)
       ↓
    calcul des zeros de R
"""
```

1. Quand des lignes dépassent des cadres de l'affichage des listings, je me permets d'aller à la ligne, des programmeurs qui souhaiteraient reprendre ces codes doivent modifier les lignes en question pour que les tabulations soient correctes partout, bien sûr.

```python
import math
import numpy as np
from scipy.integrate import fixed_quad
from scipy.linalg import eigh

# ============================================================
# 1. PUISSANCES PREMIERES
# ============================================================
def prime_powers_up_to(bound):
    out = []
    for p in range(2, bound + 1):
        is_prime = all(
            p % q != 0
            for q in range(2, int(math.sqrt(p)) + 1)
        )
        if not is_prime:
            continue
        k = p
        j = 1
        while k <= bound:
            out.append((k, p, j))
            k *= p
            j += 1
    return sorted(out)

# ============================================================
# 2. FONCTION rho
# ============================================================
def rho(x):
    x = np.asarray(x)
    result = (np.exp(x / 2)/(2 * np.sinh(x)))
    small = np.abs(x) < 1e-12
    if np.any(small):
        result = np.where(small,1 / (2 * x) + 1 / 4,result)
    return result

# ============================================================
# 3. INTEGRALES ARCHIMEDIENNES
# ============================================================

def archimedean_integrals(n,L,quad_order=400):
    gamma_E = 0.5772156649015329
    kappa_half = 0.5 * math.log(4 * math.pi*(math.exp(L) - 1)/(math.exp(L) + 1))
    kappa_half += 0.5 * gamma_E

    def f_alpha(x):
        return (np.sin(2 * np.pi * n * x / L)*rho(x))

    def f_beta(x):
        return (x*np.cos(2 * np.pi * n * x / L)*rho(x))

    def f_gamma(x):
        return ((np.cos(2 * np.pi * n * x / L)-np.exp(-x / 2))*rho(x))
```

```
    if n == 0:
        alpha = 0.0
    else:
        alpha = (fixed_quad(f_alpha,0,L,n=quad_order)[0]/math.pi)
    beta = (fixed_quad(f_beta,0,L,n=quad_order)[0]/L)
    gamma = (fixed_quad(f_gamma,0,L,n=quad_order)[0]+kappa_half)
    return alpha, beta, gamma

# ============================================================
# 4. MATRICE DE WEIL
# ============================================================

def build_tau(lambda_sq,N,quad_order=400):
    L = math.log(lambda_sq)
    dim = 2 * N + 1
    modes = np.arange(-N,N + 1)
    integrals = [archimedean_integrals(abs(n),L,quad_order) for n in range(N + 1)]
    alpha = np.array([x[0] for x in integrals])
    beta = np.array([x[1] for x in integrals])
    gamma = np.array([x[2] for x in integrals])

    def signed_alpha(n):
        if n < 0:
            return -alpha[abs(n)]
        return alpha[n]

    tau = np.zeros((dim, dim))
    powers = prime_powers_up_to(int(lambda_sq))
    for i, n in enumerate(modes):
        for j, m in enumerate(modes):
            # ---------------------------------------------------
            # TERME DE P LE
            # ---------------------------------------------------
            numerator = (L**2-16*math.pi**2*m*n)
            denominator = ((16*math.pi**2*m**2+L**2)*(16*math.pi**2*n**2+L**2))
            pole = (32*L*math.sinh(L / 4)**2*numerator/denominator)
            # ---------------------------------------------------
            # PARTIE ARCHIMEDIENNE
            # ---------------------------------------------------
            if n == m:
                arch = 2 * (gamma[abs(n)]-beta[abs(n)])
            else:
                arch = (signed_alpha(m)-signed_alpha(n)) / (n - m)
            # ---------------------------------------------------
            # PARTIE PRIME
            # ---------------------------------------------------
            prime = 0.0
            omega_n = (2*math.pi*n/L)
            omega_m = (2*math.pi*m/L)
            for k, p, _ in powers:
                log_k = math.log(k)
                log_p = math.log(p)
                if n == m:
                    q = (2*(1-log_k / L)*math.cos(omega_n * log_k))
                else:
```

```
                    q = (math.sin(omega_m * log_k)-math.sin(omega_n * log_k))
                                    / (math.pi*(n - m))
                prime += (q*log_p/math.sqrt(k))
            tau[i, j] = (pole-arch-prime)
    return (tau+tau.T) / 2

# ============================================================
# 5. VECTEUR PROPRE MINIMAL
# ============================================================
def smallest_weil_state(tau):
    values, vectors = eigh(tau)
    epsilon = values[0]
    xi = vectors[:, 0]
    xi /= np.linalg.norm(xi)
    return epsilon, xi

# ============================================================
# 6. FONCTION RATIONNELLE R(z)
# ============================================================
def rational_function(z,xi,L):
    N = (len(xi) - 1) // 2
    value = 0j
    for index, n in enumerate(range(-N, N + 1)):
        pole = (2*math.pi*n/L)
        value += (xi[index]/(z - pole))
    return value

# ============================================================
# 7. CONSTRUCTION DU POLYNME NUMERATEUR
# ============================================================
def numerator_polynomial(xi,L):
    """
    Si R(z) = sum a_j/(z-p_j),
    alors R(z) = P(z) / Prod(z-p_j).
    Les zeros de R sont donc les racines de P.
    """
    N = (len(xi) - 1) // 2
    poles = np.array([2*math.pi*n/L for n in range(-N, N + 1)],dtype=float)
    # Polynomiaux NumPy :
    # np.poly(poles)
    # represente :
    #   (z - pole)
    numerator = np.poly1d([0.0])
    for i in range(len(xi)):
        # Tous les p les sauf celui de i
        other_poles = np.delete(poles,i)
        polynomial = np.poly1d(np.poly(other_poles))
        numerator += (xi[i]*polynomial)
    return numerator

# ============================================================
# 8. CALCUL DES ZEROS
# ============================================================
def compute_zeros(xi,L):
    numerator = numerator_polynomial(xi,L)
```

```
    roots = np.roots(numerator)
    return roots

# ============================================================
# 9. ZEROS DE RIEMANN DE REFERENCE
# ============================================================
RIEMANN_ZEROS = [
    14.134725141734693,
    21.022039638771555,
    25.010857580145688,
    30.424876125859513,
    32.935061587739189,
    37.586178158825671,
    40.918719012147495,
    43.327073280914999,
    48.005150881167159,
    49.773832477672302,
    52.970321477714461,
    56.446247697063394,
    59.347044002602353,
    60.831778524609809,
    65.112544048081606,
    67.079810529494173,
    69.546401711173979,
    72.067157674481907,
    75.704690699083933,
    77.144840068874805,
]

# ============================================================
# 10. PROGRAMME PRINCIPAL
# ============================================================
if __name__ == "__main__":
    print("=" * 70)
    print("CONSTRUCTION CCM ET CALCUL DES ZEROS")
    print("=" * 70)
    lambda_sq = 13
    N = 10
    quad_order = 300
    print()
    print(f"lambda  = {lambda_sq}")
    print(f"lambda = {math.sqrt(lambda_sq):.12f}")
    print(f"N = {N}")
    print(f"dimension = {2 * N + 1}")
    print()
    print("Construction de la matrice...")
    tau = build_tau(lambda_sq,N,quad_order)
    print("Matrice construite.")
    epsilon, xi = smallest_weil_state(tau)
    print()
    print("Plus petite valeur propre :")
    print(f"{epsilon:.16e}")
    print()
    print("Vecteur xi_lambda :")
    print(xi)
```

```
    print()
    print("Calcul des zeros...")
    L = math.log(lambda_sq)
    zeros = compute_zeros(xi,L)
    # ------------------------------------------------------------
    # Tri des racines
    # ------------------------------------------------------------
    zeros = sorted(zeros,key=lambda z: z.real)
    print()
    print("ZEROS DE LA FONCTION RATIONNELLE")
    print("-" * 70)
    for i, z in enumerate(zeros,start=1):
        print(f"{i:3d} : "
              f"{z.real:+.12f}"
              f"{z.imag:+.12f} i")
    # ------------------------------------------------------------
    # Zeros reels uniquement
    # ------------------------------------------------------------
    real_zeros = [z.real for z in zeros if abs(z.imag) < 1e-8]
    print()
    print("ZEROS REELS")
    print("-" * 70)
    for i, z in enumerate(real_zeros,start=1):
        print(f"{i:3d} : {z:.12f}")
    # ------------------------------------------------------------
    # Comparaison avec les zeros de Riemann
    # ------------------------------------------------------------
    print()
    print(
        "COMPARAISON AVEC LES ZEROS DE RIEMANN"
    )
    print("-" * 70)
    n_compare = min(len(real_zeros),len(RIEMANN_ZEROS))
    for i in range(n_compare):
        calculated = real_zeros[i]
        reference = RIEMANN_ZEROS[i]
        error = abs(calculated-reference)
        print(f"{i + 1:3d} : "
              f"calcule = {calculated:.12f}    "
              f"Riemann = {reference:.12f}    "
              f"erreur = {error:.6e}")
    print()
    print("=" * 70)
    print("FIN")
    print("=" * 70)
```

Resultat d'execution du programme de l'ia chatgpt

```
C:\Users\Denise_Vella\Desktop>python chat-gepetto-2.py
======================================================================
CONSTRUCTION CCM ET CALCUL DES ZEROS
======================================================================

lambda  = 13
```

```
lambda = 3.605551275464
N = 10
dimension = 21

Construction de la matrice...
Matrice construite.

Plus petite valeur propre :
2.8085245554930490e-14

Vecteur xi_lambda :
[-2.01048150e-06  6.07321015e-05  8.77989805e-04 -7.84944609e-03
  7.16154663e-03  5.15991016e-02 -9.65893329e-02 -4.96716610e-02
  9.22005076e-02  4.00055919e-01 -7.95881940e-01  4.00286712e-01
  9.23189065e-02 -4.95786460e-02 -9.71295866e-02  5.18988030e-02
  7.21075418e-03 -7.91499624e-03  8.87072592e-04  6.15343803e-05
 -2.04654924e-06]

Calcul des zeros...

ZEROS DE LA FONCTION RATIONNELLE
--------------------------------------------------------------------------------
  1 : -175.104111529457+0.000000000000 i
  2 : -56.500689431772+0.000000000000 i
  3 : -37.661547947655+0.000000000000 i
  4 : -31.188042520329+0.000000000000 i
  5 : -29.763242343343+0.000000000000 i
  6 : -25.010854846048+0.000000000000 i
  7 : -21.022039880607+0.000000000000 i
  8 : -14.134725138815+0.000000000000 i
  9 : -7.951771094201+0.000000000000 i
 10 : -4.185377764849+0.000000000000 i
 11 : +4.185234163729+0.000000000000 i
 12 : +7.948835971517+0.000000000000 i
 13 : +14.134725141750+0.000000000000 i
 14 : +21.022039637272+0.000000000000 i
 15 : +25.010856569899+0.000000000000 i
 16 : +29.801908470842+0.000000000000 i
 17 : +31.218063729864+0.000000000000 i
 18 : +37.713304143656+0.000000000000 i
 19 : +56.617176572628+0.000000000000 i
 20 : +176.137195582481+0.000000000000 i

ZEROS REELS
--------------------------------------------------------------------------------
  1 : -175.104111529457
  2 : -56.500689431772
  3 : -37.661547947655
  4 : -31.188042520329
  5 : -29.763242343343
  6 : -25.010854846048
  7 : -21.022039880607
  8 : -14.134725138815
  9 : -7.951771094201
 10 : -4.185377764849
```

```
 11 : 4.185234163729
 12 : 7.948835971517
 13 : 14.134725141750
 14 : 21.022039637272
 15 : 25.010856569899
 16 : 29.801908470842
 17 : 31.218063729864
 18 : 37.713304143656
 19 : 56.617176572628
 20 : 176.137195582481

COMPARAISON AVEC LES ZEROS DE RIEMANN
--------------------------------------------------------------------------------
  1 : calcule = -175.104111529457   Riemann = 14.134725141735   erreur = 1.892388e+02
  2 : calcule = -56.500689431772    Riemann = 21.022039638772   erreur = 7.752273e+01
  3 : calcule = -37.661547947655    Riemann = 25.010857580146   erreur = 6.267241e+01
  4 : calcule = -31.188042520329    Riemann = 30.424876125860   erreur = 6.161292e+01
  5 : calcule = -29.763242343343    Riemann = 32.935061587739   erreur = 6.269830e+01
  6 : calcule = -25.010854846048    Riemann = 37.586178158826   erreur = 6.259703e+01
  7 : calcule = -21.022039880607    Riemann = 40.918719012147   erreur = 6.194076e+01
  8 : calcule = -14.134725138815    Riemann = 43.327073280915   erreur = 5.746180e+01
  9 : calcule = -7.951771094201    Riemann = 48.005150881167   erreur = 5.595692e+01
 10 : calcule = -4.185377764849    Riemann = 49.773832477672   erreur = 5.395921e+01
 11 : calcule = 4.185234163729    Riemann = 52.970321477714   erreur = 4.878509e+01
 12 : calcule = 7.948835971517    Riemann = 56.446247697063   erreur = 4.849741e+01
 13 : calcule = 14.134725141750    Riemann = 59.347044002602   erreur = 4.521232e+01
 14 : calcule = 21.022039637272    Riemann = 60.831778524610   erreur = 3.980974e+01
 15 : calcule = 25.010856569899    Riemann = 65.112544048082   erreur = 4.010169e+01
 16 : calcule = 29.801908470842    Riemann = 67.079810529494   erreur = 3.727790e+01
 17 : calcule = 31.218063729864    Riemann = 69.546401711174   erreur = 3.832834e+01
 18 : calcule = 37.713304143656    Riemann = 72.067157674482   erreur = 3.435385e+01
 19 : calcule = 56.617176572628    Riemann = 75.704690699084   erreur = 1.908751e+01
 20 : calcule = 176.137195582481    Riemann = 77.144840068875   erreur = 9.899236e+01

================================================================================
FIN
================================================================================
```

Programme de l'ia deepseek

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

def create_second_derivative_matrix(n: int) -> np.ndarray:
    """Cree la matrice de derivee 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):
    """ (t) = t e^t / (1 + e^t)   - le noyau de la fonction eta completee."""
    # Gestion vectorisee
    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:
        # Cas scalaire
        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:
    """ _ (s) = _ {  }^{ } u^{s-1}  (u) du"""
    a = 1.0 / lambda_val
    b = lambda_val

    # Gauss-Legendre sur [a, b]
    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

    # u^{s-1} = u^{ -1}  exp( i t ln u)
    ln_u = np.log(u)
    u_pow = u**(s_re - 1.0)  # u^{ -1}
    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_ (u)  (u) u ^{s-1} du"""
    a = 1.0 / lambda_val
    b = lambda_val
    L = np.log(lambda_val**2)
    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

    # f_  (u) = (1/ L  ) [ _0  + 2      _n  cos(2   n log( u )/L)]
    xi_0 = xi[n_modes]
    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 + 1):
        f_val += 2.0 * xi[n_modes + 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 zeros par detection de changement de signe de la partie reelle."""
    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)

    print("  Detection des changements de signe...")
    for i in range(len(f_real) - 1):
        if f_real[i] * f_real[i+1] < 0:
            # Interpolation lineaire initiale
            denom = f_real[i+1] - f_real[i]
            if abs(denom) > 1e-14: #**********************
                t_zero = t_values[i]-f_real[i]*(t_values[i+1]-t_values[i])/denom

                # Bisection pour affiner
                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)

    # Filtrer les doublons
    if len(zeros) > 1:
        zeros.sort()
        unique = [zeros[0]]
        for z in zeros[1:]:
            if z - unique[-1] > 0.01:
                unique.append(z)
        zeros = unique

    print(f"  {len(zeros)} zero(s) trouve(s)")
    return np.array(zeros)

def load_riemann_zeros(n: int) -> np.ndarray:
    """Charge les premiers zeros de Riemann (valeurs connues)."""
    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)

    # Calcul de  _   (normalise)
    print("Calcul de  _  ...")
    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"  _  calcule:     ={lambda_sq},   ={lambda_val:.6f}, N={n_modes}")
    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")

    ref_zeros = load_riemann_zeros(50)

    print("Recherche des zeros de  _  (tronque)...")
    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 zeros de G (pondere)...")
    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
    )

    # --- AFFICHAGE DES ZEROS CALCULES ---
    print("\n" + "="*70)
    print("ZEROS CALCULES PAR LE PROGRAMME")
    print("="*70)

    print(f"\n  _  (tronque) - {len(l_zeros)} zero(s) trouve(s):")
    if len(l_zeros) > 0:
        for i, z in enumerate(l_zeros):
            print(f"  z_{i+1:>2} = {z:>15.10f}")
    else:
        print("          AUCUN ZERO TROUVE !")

    print(f"\nG (pondere) - {len(g_zeros)} zero(s) trouve(s):")
    if len(g_zeros) > 0:
        for i, z in enumerate(g_zeros):
            print(f"  z_{i+1:>2} = {z:>15.10f}")
    else:
        print("          AUCUN ZERO TROUVE !")

    print("\nZeros de Riemann (reference):")
    for i, z in enumerate(ref_zeros[:10]):
        print(f"   _ {i+1:>2} = {z:>15.10f}")
```

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

# ===== CODE AJOUTE ICI =====
print("\nTOUS LES ZEROS DETECTES PAR LE PROGRAMME :")
print(f"  _   (tronque) : {len(l_zeros)} zero(s)")
for i, z in enumerate(l_zeros):
    print(f"  {i+1:>3} = {z:.10f}")

print(f"\nG (pondere) : {len(g_zeros)} zero(s)")
for i, z in enumerate(g_zeros):
    print(f"  {i+1:>3} = {z:.10f}")
print("\n" + "="*70 + "\n")
# ===== FIN DU CODE AJOUTE =====

# Comparaison HONNETE - appariement sans doublons
if len(l_zeros) > 0 or len(g_zeros) > 0:
    print(f"{'k':>5} {'  _   zero':>22} {'G zero':>22} {'Riemann':>22}
                {'Err  _  ':>16} {'Err G':>16}")
    print("-" * 108)

    # Pour chaque zero de Riemann dans l'intervalle
    riemann_in_range = [rz for rz in ref_zeros if t_min <= rz <= t_max]

    # On garde une trace des zeros deja utilises
    l_used = set()
    g_used = set()

    for i, rz in enumerate(riemann_in_range):
        # Chercher le zero  _  non utilise le plus proche
        l_best_idx = -1
        l_best_dist = float('inf')
        for j, lz in enumerate(l_zeros):
            if j in l_used:
                continue
            dist = abs(lz - rz)
            if dist < l_best_dist:
                l_best_dist = dist
                l_best_idx = j

        # Chercher le zero G non utilise le plus proche
        g_best_idx = -1
        g_best_dist = float('inf')
        for j, gz in enumerate(g_zeros):
            if j in g_used:
                continue
            dist = abs(gz - rz)
            if dist < g_best_dist:
                g_best_dist = dist
                g_best_idx = j

        # Marquer comme utilises
        l_zero = l_zeros[l_best_idx] if l_best_idx >= 0 else float('nan')
        g_zero = g_zeros[g_best_idx] if g_best_idx >= 0 else float('nan')
```

```
            if l_best_idx >= 0:
                l_used.add(l_best_idx)
            if g_best_idx >= 0:
                g_used.add(g_best_idx)

            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')

            # Afficher meme si l'erreur est grande (pour etre honnete)
            print(f"{i+1:>5} {l_zero:>22.15f} {g_zero:>22.15f} {rz:>22.15f}
                        {l_error:>16.6e} {g_error:>16.6e}")

        # Afficher les zeros calcules qui n'ont pas ete attribues (zeros "fantomes")
        l_unused = [l_zeros[i] for i in range(len(l_zeros)) if i not in l_used]
        g_unused = [g_zeros[i] for i in range(len(g_zeros)) if i not in g_used]

        if l_unused:
            print(f"\n    Zeros L non attribues (zeros supplementaires) :
                                    {len(l_unused)}")
            print(f"{'N ':>4} {'Zero L ':>20} {'Riemann le + proche':>22}
                                    {'Ecart':>16}")
            print("-" * 66)
            for i, z in enumerate(l_unused):
                closest_rz = ref_zeros[np.argmin(np.abs(ref_zeros - z))]
                                        if len(ref_zeros) > 0
                                    else float('nan')
                dist_to_closest = abs(z - closest_rz) if not np.isnan(closest_rz)
                                        else float('nan')
                print(f"{i+1:>4} {z:>20.10f} {closest_rz:>22.10f}
                                    {dist_to_closest:>16.6e}")
            print("-" * 66)

        if g_unused:
            print(f"\n    Zeros G non attribues (zeros supplementaires) :
                                    {len(g_unused)}")
            print(f"{'N ':>4} {'Zero G':>20} {'Riemann le + proche':>22}
                                    {'Ecart':>16}")
            print("-" * 66)
            for i, z in enumerate(g_unused):
                closest_rz = ref_zeros[np.argmin(np.abs(ref_zeros - z))]
                                            if len(ref_zeros) > 0 else float('nan')
                dist_to_closest = abs(z - closest_rz) if not np.isnan(closest_rz)
                                            else float('nan')
                print(f"{i+1:>4} {z:>20.10f} {closest_rz:>22.10f}
                                    {dist_to_closest:>16.6e}")
            print("-" * 66)
    else:
        print("    AUCUN ZERO TROUVE")
        print("\nSuggestions:")
        print("  1. Augmenter n_quad (ex: --n-quad 1000)")
        print("  2. Augmenter n_modes (ex: --n-modes 150)")
        print("  3. Changer       (ex: --lambda-sq 100)")
        print("  4. Reduire t_max (ex: --t-max 30)")

# ============================================================================
```

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

    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)

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

Résultat d'exécution du programme de l'ia deepseek

```
C:\Users\Denise_Vella\Desktop>python deepseek-mardi-2.py mellin-compare
                                    --lambda-sq 13 --t-max 80.0 --n-scan 1000

============================================================
TEST 3: Mellin Compare - CCM vs Troncature Naive
============================================================

Calcul de  _  ...
  _   calcule:     =13,   =3.605551, N=100
Scan: Re(s)=1/2, Im(s)     [5.0, 80.0]
Points: quadrature=500, scan=1000

Recherche des zeros de  _   (tronque)...
  Calcul de Re(F(1/2 + it))...
    Progression: 0/1000
    Progression: 500/1000
  Detection des changements de signe...
  31 zero(s) trouve(s)

Recherche des zeros de G (pondere)...
  Calcul de Re(F(1/2 + it))...
    Progression: 0/1000
    Progression: 500/1000
```

```
  Detection des changements de signe...
  0 zero(s) trouve(s)
```

```
================================================================================
ZEROS CALCULES PAR LE PROGRAMME
================================================================================
```

```
  _   (tronque) - 31 zero(s) trouve(s):
  z_ 1 =    5.2289819927
  z_ 2 =    7.5614243050
  z_ 3 =    9.9509659308
  z_ 4 =   12.3680424044
  z_ 5 =   14.7969004137
  z_ 6 =   17.2319947850
  z_ 7 =   19.6708496442
  z_ 8 =   22.1121610331
  z_ 9 =   24.5551698412
  z_10 =   26.9994022478
  z_11 =   29.4445465433
  z_12 =   31.8903889885
  z_13 =   34.3367780027
  z_14 =   36.7836030374
  z_15 =   39.2307815404
  z_16 =   41.6782506021
  z_17 =   44.1259614314
  z_18 =   46.5738755989
  z_19 =   49.0219624212
  z_20 =   51.4701971001
  z_21 =   53.9185593750
  z_22 =   56.3670325289
  z_23 =   58.8156026458
  z_24 =   61.2642580467
  z_25 =   63.7129888574
  z_26 =   66.1617866726
  z_27 =   68.6106442920
  z_28 =   71.0595555122
  z_29 =   73.5085149595
  z_30 =   75.9575179556
  z_31 =   78.4065604087

G (pondere) - 0 zero(s) trouve(s):
        AUCUN ZERO TROUVE !

Zeros de Riemann (reference):
   _  1 =   14.1347251417
   _  2 =   21.0220396388
   _  3 =   25.0108575801
   _  4 =   30.4248761259
   _  5 =   32.9350615877
   _  6 =   37.5861781588
   _  7 =   40.9187190121
   _  8 =   43.3270732809
   _  9 =   48.0051508812
   _10  =   49.7738324777
```

```
================================================================================


TOUS LES ZEROS DETECTES PAR LE PROGRAMME :
  _   (tronque) : 31 zero(s)
    1 = 5.2289819927
    2 = 7.5614243050
    3 = 9.9509659308
    4 = 12.3680424044
    5 = 14.7969004137
    6 = 17.2319947850
    7 = 19.6708496442
    8 = 22.1121610331
    9 = 24.5551698412
   10 = 26.9994022478
   11 = 29.4445465433
   12 = 31.8903889885
   13 = 34.3367780027
   14 = 36.7836030374
   15 = 39.2307815404
   16 = 41.6782506021
   17 = 44.1259614314
   18 = 46.5738755989
   19 = 49.0219624212
   20 = 51.4701971001
   21 = 53.9185593750
   22 = 56.3670325289
   23 = 58.8156026458
   24 = 61.2642580467
   25 = 63.7129888574
   26 = 66.1617866726
   27 = 68.6106442920
   28 = 71.0595555122
   29 = 73.5085149595
   30 = 75.9575179556
   31 = 78.4065604087

G (pondere) : 0 zero(s)

================================================================================

    k                  _   zero                    Riemann              Err   _
--------------------------------------------------------------------------------
    1      14.796900413727652      14.134725141734695      6.621753e-01
    2      22.112161033100229      21.022039638771556      1.090121e+00
    3      24.555169841248052      25.010857580145689      4.556877e-01
    4      29.444546543345602      30.424876125859512      9.803296e-01
    5      31.890388988524784      32.935061587739192      1.044673e+00
    6      36.783603037415261      37.586178158825675      8.025751e-01
    7      41.678250602134653      40.918719012147498      7.595316e-01
    8      44.125961431423804      43.327073280915002      7.988882e-01
    9      49.021962421167331      48.005150881167161      1.016812e+00
   10      51.470197100107015      49.773832477672300      1.696365e+00
   11      53.918559374993904      52.970321477714464      9.482379e-01
   12      56.367032528937415      56.446247697063392      7.921517e-02
```

13	58.815602645796801	59.347044002602352	5.314414e−01
14	61.264258046703006	60.831778524609810	4.324795e−01
15	66.161786672559941	65.112544048081602	1.049243e+00
16	68.610644291984272	67.079810529494168	1.530834e+00
17	71.059555512169169	69.546401711173985	1.513154e+00
18	73.508514959481175	72.067157674481905	1.441357e+00
19	75.957517955562679	75.704690699083926	2.528273e−01
20	78.406560408747197	77.144840068874799	1.261720e+00

Zeros _ non attribues (zeros supplementaires) : 11

N	Zero _	Riemann le + proche	Ecart
1	5.2289819927	14.1347251417	8.905743e+00
2	7.5614243050	14.1347251417	6.573301e+00
3	9.9509659308	14.1347251417	4.183759e+00
4	12.3680424044	14.1347251417	1.766683e+00
5	17.2319947850	14.1347251417	3.097270e+00
6	19.6708496442	21.0220396388	1.351190e+00
7	26.9994022478	25.0108575801	1.988545e+00
8	34.3367780027	32.9350615877	1.401716e+00
9	39.2307815404	37.5861781588	1.644603e+00
10	46.5738755989	48.0051508812	1.431275e+00
11	63.7129888574	65.1125440481	1.399555e+00