Opérateur d'Alain Connes pour les zéros de zeta, version de l'IA gemini

On fournit simplement ici la version simplifiée par gemini du programme python `operator.py` fourni par Akiva Groskin, dans le github ici lien github, qui fait partie d'un ensemble de programmes en appui de son article posté récemment sur arxiv 2605.20224v1.

Le travail de Groskin s'appuie sur plusieurs articles de Connes, van Suijlekom, Consani, Moscovici (notamment les articles d'Alain Connes en 1997 : Trace formula in noncommutative geometry and the zeros of the Riemann zeta function, Journées EDP Saint-Jean-de-Monts, et 1999 article Selecta Math, ainsi que les articles [1], [2], [3], [4], [5], [6]) au sujet, parmi de multiples autres sujets, de la recherche d'un opérateur dont les valeurs propres seraient les parties imaginaires des zéros non triviaux de la fonction ζ de Riemann.

Ces programmes python utilisent les outils de la librairie `GMPy2` pour les calculs sur des flottants en très très haute précision, ainsi que les outils de la librairie `mpmath` pour les calculs matriciels, ainsi enfin que le calcul de la fonction digamma effectué par la librairie `python-flint`.

L'exécution de ce programme prenant beaucoup de temps sur notre machine, on demande à l'IA gemini de réécrire le code en effectuant des calculs de moindre précision mais plus rapides, en utilisant des matrices `numpy` plutôt que des matrices `mpmath` (car le traitement des matrices par `numpy` est très efficace), de se débrouiller pour calculer digamma en évitant la librairie `python-flint`. Gemini rend également plus efficace le calcul des intégrales.

On le fait tourner sur 2 PC différents :

— *pc ubuntu* :
OS : Linux 7.0.0-22-generic (v22-ubuntu Smp preempt-dynamic 25.5.2026),
python 3.14.4,
8 cœurs physiques, 16 cœurs logiques,
mémoire RAM totale 12.99 Go.

— *pc windows* :
OS : Windows 11 (v10.0.26200),
python 3.12.10,
Processeur : AMD64 Family 23 Model 104 Stepping 1, AuthenticAMD,
8 cœurs physiques, 16 cœurs logiques,
mémoire RAM totale 15.33 Go.

Denise Vella-Chemla, mai 2026.

Résultat de l'exécution :

```
Construction de la matrice de Galerkin...
Taille de la matrice : 201 x 201
Calcul de l'etat fondamental...
  lambda_min(c=7) = -0.024684
Extraction des zeros de Riemann...
{'k': 1, 'gamma_true': 14.134725141734695,
          'gamma_detected': 14.133320585088521,
          'error': 0.0014045566461735604}
{'k': 2, 'gamma_true': 21.022039638771556,
          'gamma_detected': 21.03333984144365,
          'error': 0.011300202672092752}
{'k': 3, 'gamma_true': 25.01085758014569,
          'gamma_detected': 25.022990152472627,
          'error': 0.012132572326937208}
Temps total d'execution : 157.82 s.
```

Sur la machine windows, plus ancienne, le temps d'exécution fournit exactement le même résultat en 250.01 s.

Le programme en python

```python
import time
import numpy as np
import mpmath as mp

def prime_powers_up_to(c):
    c = int(c)
    is_prime = [True] * (c + 1)
    is_prime[0] = is_prime[1] = False
    for i in range(2, int(c**0.5) + 1):
        if is_prime[i]:
            for j in range(i*i, c + 1, i):
                is_prime[j] = False
    primes_list = [i for i, prime in enumerate(is_prime) if prime]
    result = []
    for p in primes_list:
        pk = p
        while pk <= c:
            result.append((pk, np.log(pk), np.log(p) / np.sqrt(pk)))
            pk *= p
    return sorted(result, key=lambda x: x[0])

def psi_prime(x, L, prime_data):
    s = sum(w * np.sin(2 * np.pi * x * (1 - logn / L))
            for (_, logn, w) in prime_data)
    return -s / np.pi
```

```
def psi_prime_deriv(x, L, prime_data):
    s = sum(w * 2 * (1 - logn / L) * np.cos(2 * np.pi * x * (1 - logn / L))
            for (_, logn, w) in prime_data)
    return -s

# --- Correction majeure ici : Utilisation stricte des fonctions de mpmath
# pour l'integrande ---
def psi_pole(x, L):
    # x et L sont convertis en float pour NumPy, mais l'integrande doit
    # accepter le format de mpmath
    def integrand(y):
        return mp.sin(2 * mp.pi * x * (1 - y / L)) * 2 * mp.cosh(y / 2)
    return float(mp.quad(integrand, [0, L])) / np.pi

def psi_pole_deriv(x, L):
    def integrand(y):
        return (1 - y / L) * mp.cos(2 * mp.pi * x * (1 - y / L))
                                * 2 * mp.cosh(y / 2)
    return 2 * float(mp.quad(integrand, [0, L]))

def h_plus(tau):
    # Force tau en float puis utilise le digamma de mpmath
    z = mp.mpc(0.25, float(tau) / 2)
    return float(mp.re(mp.digamma(z)) - mp.log(mp.pi))

def _re_S_and_dS_fused(tau, x, L):
    """Calculs geometriques en floats purs."""
    alpha = 2 * np.pi * x / L
    s2pi, c2pi = np.sin(2 * np.pi * x), np.cos(2 * np.pi * x)
    beta1 = alpha - tau
    if abs(beta1) < 1e-10:
        A1r, A1i = L, 0.0
        B1r, B1i = L / 2, 0.0
    else:
        bL1 = beta1 * L
        sh1 = np.sin(bL1 / 2)
        sf1 = np.sin(bL1)
        A1r = sf1 / beta1
        sh1_sq2 = 2 * sh1 * sh1
        A1i = sh1_sq2 / beta1
        Lb1b1 = L * beta1 * beta1
        B1r = sh1_sq2 / Lb1b1
        if abs(bL1) < 1e-5:
            B1i = beta1 * L * L / 6 * (1 - (bL1**2) / 20)
        else:
            B1i = (bL1 - sf1) / Lb1b1
    beta2 = -(alpha + tau)
    if abs(beta2) < 1e-10:
        A2r, A2i = L, 0.0
        B2r, B2i = L / 2, 0.0
```

```
    else:
        bL2 = beta2 * L
        sh2 = np.sin(bL2 / 2)
        sf2 = np.sin(bL2)
        A2r = sf2 / beta2
        sh2_sq2 = 2 * sh2 * sh2
        A2i = sh2_sq2 / beta2
        Lb2b2 = L * beta2 * beta2
        B2r = sh2_sq2 / Lb2b2
        if abs(bL2) < 1e-5:
            B2i = beta2 * L * L / 6 * (1 - (bL2**2) / 20)
        else:
            B2i = (bL2 - sf2) / Lb2b2
    re_Ic = (A1r + A2r) / 2
    re_Is = (A1i - A2i) / 2
    re_S = s2pi * re_Ic - c2pi * re_Is
    re_Bc = (B1r + B2r) / 2
    re_Bs = (B1i - B2i) / 2
    re_dS = 2 * np.pi * (c2pi * re_Bc + s2pi * re_Bs)
    return re_S, re_dS

def psi_arch_and_deriv(x, L, T):
    if x == 0:
        return 0.0, 0.0
    alpha_x = 2 * np.pi * x / L
    sings = sorted([s for s in [0.0, alpha_x, -alpha_x] if -T < s < T])
    pts = [-T] + sings + [T]
    # Ici, tau est fourni par mpmath.quad, on le force en float pour nos calculs
    def integrand_S(tau):
        t_fl = float(tau)
        hp = h_plus(t_fl)
        S, _ = _re_S_and_dS_fused(t_fl, x, L)
        return hp * S
    def integrand_dS(tau):
        t_fl = float(tau)
        hp = h_plus(t_fl)
        _, dS = _re_S_and_dS_fused(t_fl, x, L)
        return hp * dS
    total_S = sum(float(mp.quad(integrand_S, [pts[i], pts[i+1]]))
                  for i in range(len(pts) - 1))
    total_dS = sum(float(mp.quad(integrand_dS, [pts[i], pts[i+1]]))
                   for i in range(len(pts) - 1))
    norm = 2 * np.pi * np.pi
    return total_S / norm, total_dS / norm

def build_galerkin_matrix(c, N=100, T=400):
    L = np.log(c)
    prime_data = prime_powers_up_to(c)
    psi_vals = np.zeros(2 * N + 1)
    psi_deriv_vals = np.zeros(2 * N + 1)
```

```
    for idx, n_idx in enumerate(range(-N, N + 1)):
        p_prime = psi_prime(n_idx, L, prime_data) + psi_pole(n_idx, L)
        p_deriv = psi_prime_deriv(n_idx, L, prime_data)
                        + psi_pole_deriv(n_idx, L)
        p_arch, p_arch_d = psi_arch_and_deriv(n_idx, L, T)
        psi_vals[idx] = p_prime + p_arch
        psi_deriv_vals[idx] = p_deriv + p_arch_d
    DIM = 2 * N + 1
    Q = np.zeros((DIM, DIM))
    for i in range(DIM):
        m_idx = i - N
        for j in range(DIM):
            n_idx = j - N
            if m_idx == n_idx:
                Q[i, j] = psi_deriv_vals[j]
            else:
                Q[i, j] = (psi_vals[i] - psi_vals[j]) / (m_idx - n_idx)
    Q = (Q + Q.T) / 2
    return Q

def compute_ground_state(Q):
    DIM = Q.shape[0]
    N = (DIM - 1) // 2
    V_even = np.zeros((DIM, N + 1))
    V_even[N, 0] = 1.0
    inv_sqrt2 = 1.0 / np.sqrt(2)
    for k in range(1, N + 1):
        V_even[N + k, k] = inv_sqrt2
        V_even[N - k, k] = inv_sqrt2
    Q_even = V_even.T @ Q @ V_even
    eigs, vecs = np.linalg.eigh(Q_even)
    min_idx = np.argmin(eigs)
    lambda_even = eigs[min_idx]
    v_even_proj = vecs[:, min_idx]
    v_even_proj /= np.linalg.norm(v_even_proj)
    v_full = V_even @ v_even_proj
    v_full /= np.linalg.norm(v_full)
    return lambda_even, v_full

def extract_zeros(eigvec, L, n_zeros=3):
    DIM = len(eigvec)
    N = (DIM - 1) // 2
    def F_even(tau):
        tau = float(tau)
        total = 0.0j
        exp_tL = np.exp(-1j * tau * L)
        for k in range(-N, N + 1):
            c_coef = eigvec[k + N]
            if c_coef == 0:
                continue
```

```
            denom = 2 * np.pi * k / L - tau
            if abs(denom) < 1e-10:
                term = L + 0j
            else:
                term = (exp_tL - 1.0) / (1j * denom)
            total += c_coef * term
        total /= np.sqrt(L)
        return np.real(np.exp(1j * tau * L / 2) * total)
    gamma_true = [float(mp.im(mp.zetazero(k))) for k in range(1, n_zeros + 1)]
    results = []
    for k, g in enumerate(gamma_true, 1):
        entry = {'k': k, 'gamma_true': g, 'gamma_detected': None, 'error': None}
        try:
            # Recherche de racine robuste avec tolerance float standard
            root = float(mp.findroot(F_even, g, tol=1e-10))
            entry['gamma_detected'] = root
            entry['error'] = abs(root - g)
        except Exception:
            pass
        results.append(entry)
    return results

tic = time.time()
c = 7
L = np.log(c)
print("Construction de la matrice de Galerkin...")
Q = build_galerkin_matrix(c, N=100, T=400)
print(f"  Taille de la matrice : {Q.shape[0]} x {Q.shape[1]}")
print("Calcul de l'etat fondamental...")
lam_min, eigvec = compute_ground_state(Q)
print(f"  lambdamin(c={c}) = {lam_min:.6f}")
print("Extraction des zeros de Riemann...")
zeros = extract_zeros(eigvec, L=L, n_zeros=3)
for z in zeros:
    print(z)
print(f"Temps total d'execution : {time.time() - tic:.2f} s.")
```