```
"""
ccm_zeros.py

Construction pédagogique :

    lambda²
       ↓
    matrice de Weil
       ↓
    plus petit vecteur propre xi_lambda
       ↓
    fonction rationnelle R(z)
       ↓
    calcul des zéros de R
"""

import math
import numpy as np

from scipy.integrate import fixed_quad
from scipy.linalg import eigh


# ============================================================
# 1. PUISSANCES PREMIÈRES
# ============================================================

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. INTÉGRALES ARCHIMÉDIENNES
# ============================================================

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 PÔLE
        # ------------------------------------------------

        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 ARCHIMÉDIENNE
# ------------------------------------------------

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 POLYNÔME NUMÉRATEUR
# ============================================================

def numerator_polynomial(
    xi,
    L
):
    """
    Si

        R(z) = Σ a_j/(z-p_j),

    alors

        R(z) = P(z) / Π(z-p_j).

    Les zéros 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)
    #
    # représente :
    #
    # Π(z - pole)

    numerator = np.poly1d(
        [0.0]
    )

    for i in range(
        len(xi)
    ):

        # Tous les pôles 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 ZÉROS
# ============================================================
```

```
def compute_zeros(
    xi,
    L
):

    numerator = numerator_polynomial(
        xi,
        L
    )

    roots = np.roots(
        numerator
    )

    return roots


# ============================================================
# 9. ZÉROS DE RIEMANN DE RÉFÉRENCE
# ============================================================

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 ZÉROS"
    )

    print("=" * 70)

    lambda_sq = 13

    N = 10

    quad_order = 300

    print()

    print(
        f"lambda² = {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 zéros..."
)

L = math.log(
    lambda_sq
)

zeros = compute_zeros(
    xi,
    L
)

# ----------------------------------------------------------
# Tri des racines
# ----------------------------------------------------------

zeros = sorted(
    zeros,
    key=lambda z: z.real
)

print()

print(
    "ZÉROS 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"

    )

# ----------------------------------------------------------
# Zéros réels uniquement
```

```
# ----------------------------------------------------------

real_zeros = [

    z.real

    for z in zeros

    if abs(z.imag) < 1e-8

]

print()

print(
    "ZÉROS RÉELS"
)

print("-" * 70)

for i, z in enumerate(
    real_zeros,
    start=1
):

    print(
        f"{i:3d} : {z:.12f}"
    )

# ----------------------------------------------------------
# Comparaison avec les zéros de Riemann
# ----------------------------------------------------------

print()

print(
    "COMPARAISON AVEC LES ZÉROS 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"calculé = {calculated:.12f}   "
        f"Riemann = {reference:.12f}   "
        f"erreur = {error:.6e}"

    )

print()

print("=" * 70)

print(
    "FIN"
)

print("=" * 70)
```