```
import numpy as np
from scipy.optimize import minimize_scalar
import time
import gc
import sys

# ============================================================
# Fonctions de calcul
# ============================================================

def zeta_approx_batch(t_values, K, batch_size=5000):
    """
    Calcule zeta(1/2 + i*t) par lots sur k.
    Retourne les valeurs complexes.

    Optimisation vs version initiale: pour s = 0.5+it on a toujours
    B = A - 1 (avec A = 1-s = 0.5-it). Donc x^B = x^(A-1) = x^A / x,
    ce qui permet de ne calculer que 2 exponentielles complexes
    (k^A et (k+1)^A) au lieu de 4 (k^A, (k+1)^A, k^B, (k+1)^B).
    Gain mesuré: ~1.6-1.7x, sans perte de précision (les exp() complexes
    sont le poste de calcul dominant ici).
    """
    t_values = np.asarray(t_values, dtype=np.float64)
    scalar_input = (t_values.ndim == 0)
    t_values = np.atleast_1d(t_values)
    n_t = len(t_values)

    result = np.zeros(n_t, dtype=np.complex128)

    # Traitement par blocs sur k
    n_batches = (K + batch_size - 1) // batch_size

    for batch_idx in range(n_batches):
        k_start = batch_idx * batch_size + 1
        k_end = min((batch_idx + 1) * batch_size, K)
        k = np.arange(k_start, k_end + 1, dtype=np.float64)
        kpl = k + 1.0
        k_col = k.reshape(-1, 1)
        kpl_col = kpl.reshape(-1, 1)

        logk_col = np.log(k_col)
        logkpl_col = np.log(kpl_col)

        for i, ti in enumerate(t_values):
            A = 0.5 - 1j * ti
            B = A - 1.0  # toujours vrai ici (B = -0.5 - it)

            p_k = np.exp(logk_col * A)      # k^A
            p_kpl = np.exp(logkpl_col * A)  # (k+1)^A

            # term1 = (1/A)*((k+1)^A - k^A)
            # term2 = (k/B)*((k+1)^B - k^B) = (k/B)*(p_kpl/(k+1) - p_k/k)
            #       = (k_col/kpl_col)*p_kpl/B - p_k/B
            term = (p_kpl - p_k) / A - (k_col / kpl_col) * p_kpl / B + p_k / B

            result[i] += np.sum(term)

        # NB: gc.collect() a été retiré ici. Il était appelé à CHAQUE lot
        # (des centaines de fois par évaluation de zeta), ce qui coûtait
        # ~10x le temps de calcul réel — critique car minimize_scalar()
        # rappelle zeta_approx_single des dizaines de fois par zéro affiné.
        # numpy libère ces tableaux tout seul dès qu'ils sont réassignés
        # au tour de boucle suivant; gc.collect() manuel n'apporte rien ici.
```

```
    # Partie x < 0
    s_conj = 0.5 - 1j * t_values
    J = result + 1.0 / s_conj
    s = 0.5 + 1j * t_values
    result_final = -s * J

    return result_final[0] if scalar_input else result_final


def zeta_approx_single(t, K, batch_size=5000):
    """Calcule zeta pour un seul t."""
    return zeta_approx_batch(np.array([t]), K, batch_size)[0]


# ============================================================
# Recherche des zéros avec affichage en temps réel
# ============================================================

# Zéros connus pour vérification
ZEROS_CONNUS = [
    14.134725142, 21.022039639, 25.010857580, 30.424876126,
    32.935061588, 37.586178159, 40.918719012, 43.327073281,
    48.005150881, 49.773832478, 52.970321478, 56.446247697,
    59.347044003, 60.831778525
]


def find_zeros_with_display(t_min, t_max, K, step=0.05, threshold=0.01,
batch_size=5000,
                            coarse_factor=6):
    """
    Recherche les zéros et affiche les résultats au fur et à mesure.

    IMPORTANT: `threshold` est le seuil final (appliqué APRÈS affinement local).
    Le balayage grossier (pas = `step`) utilise un seuil plus large,
    `coarse_threshold = coarse_factor * threshold`, pour repérer les *candidats*
    à affiner. Sans cette marge, un vrai zéro peut tomber entre deux points de
    la grille et n'être jamais détecté simplement parce qu'aucun point
    échantillonné n'est assez proche de lui (bug initial: le seuil strict était
    appliqué avant l'affinement, ce qui rejetait des zéros réels).
    En pratique, il faut coarse_threshold >~ (pente locale de |zeta|) * step / 2
;
    coarse_factor=6 laisse une marge confortable pour step=0.05.
    """
    coarse_threshold = coarse_factor * threshold

    print(f"\n{'='*70}")
    print(f"RECHERCHE DES ZÉROS DE ζ(1/2+it)")
    print(f"{'='*70}")
    print(f"Plage: [{t_min}, {t_max}]")
    print(f"Pas du balayage: {step}")
    print(f"K = {K:,} paliers")
    print(f"Seuil de détection (grille grossière): {coarse_threshold}")
    print(f"Seuil de validation (après affinement): {threshold}")
    print(f"{'='*70}\n")

    # Création du balayage
    t_scan = np.arange(t_min, t_max, step)
    n_points = len(t_scan)
    print(f"Nombre de points à évaluer: {n_points}")
    print(f"Estimation du temps: {n_points * K / 1e8:.1f} secondes\n")

    # Préparation pour le stockage des résultats
    found_zeros = []
```

```
    vals_all = np.empty(n_points, dtype=np.float64)

    # -------- Phase 1: calcul de |zeta| sur toute la grille, par lots de t
    # (le lotissement ici sert seulement à limiter la mémoire/le temps par
    # appel et à afficher une progression ; la détection des minima se fait
    # APRES, sur le tableau complet, pour éviter tout bug de voisinage aux
    # frontières de lot).
    batch_t_size = 20  # Nombre de t simultanés
    start_total = time.time()

    for start_idx in range(0, n_points, batch_t_size):
        end_idx = min(start_idx + batch_t_size, n_points)
        t_batch = t_scan[start_idx:end_idx]

        start_calc = time.time()
        z_batch = zeta_approx_batch(t_batch, K, batch_size)
        vals_all[start_idx:end_idx] = np.abs(z_batch)
        calc_time = time.time() - start_calc

        progress = (end_idx / n_points) * 100
        elapsed = time.time() - start_total
        print(f"  Progression (calcul): {progress:.1f}%  (lot de {len(t_batch)}
points en {calc_time:.2f}s, total: {elapsed:.1f}s)")
        sys.stdout.flush()

    # -------- Phase 2: détection des minima locaux + affinement, sur le
    # tableau complet vals_all (indices de voisinage toujours valides ici,
    # contrairement à la version d'origine qui comparait des indices
    # globaux dans un tableau local de taille <= batch_t_size).
    print(f"\n  Détection des minima et affinement...")
    sys.stdout.flush()
    for global_idx in range(n_points):
        if global_idx == 0:
            is_min = n_points > 1 and vals_all[0] < vals_all[1]
        elif global_idx == n_points - 1:
            is_min = vals_all[global_idx] < vals_all[global_idx - 1]
        else:
            is_min = (vals_all[global_idx] < vals_all[global_idx - 1] and
                      vals_all[global_idx] < vals_all[global_idx + 1])

        if is_min and vals_all[global_idx] < coarse_threshold:
            t_candidate = t_scan[global_idx]
            try:
                res = minimize_scalar(
                    lambda t: abs(zeta_approx_single(t, K, batch_size)),
                    bounds=(t_candidate - step, t_candidate + step),
                    method='bounded',
                    options={'xatol': 1e-8, 'maxiter': 100}
                )
                t_found = res.x
                val_found = res.fun
            except Exception:
                t_found = t_candidate
                val_found = vals_all[global_idx]

            # Seuil strict appliqué APRES affinement (comparable à la
            # précision réelle de zeta_approx pour ce K)
            if val_found < threshold:
                found_zeros.append((t_found, val_found))
                nearest = min(ZEROS_CONNUS, key=lambda z: abs(z - t_found))
                ecart = abs(t_found - nearest)
                print(f"✓ Zéro trouvé: t = {t_found:.9f}  |ζ| = {val_found:.2e}
"
                      f"(connu: {nearest:.9f}, écart: {ecart:.2e})")
```

```
                sys.stdout.flush()

    total_time = time.time() - start_total
    print(f"\n{'='*70}")
    print(f"RECHERCHE TERMINÉE")
    print(f"{'='*70}")
    print(f"Temps total: {total_time:.1f} secondes")
    print(f"Nombre de zéros trouvés: {len(found_zeros)}")

    # Tri et affichage final
    found_zeros.sort(key=lambda x: x[0])

    print(f"\n{'N°':>4} {'t trouvé':>14} {'|ζ|':>12} {'connu':>18}
{'écart':>12}")
    print("-" * 70)

    # Vérification des zéros manqués
    manques = []
    for z in ZEROS_CONNUS:
        if t_min <= z <= t_max:
            if not any(abs(z - t) < 0.02 for t, _ in found_zeros):
                manques.append(z)

    for i, (t_found, val) in enumerate(found_zeros, 1):
        nearest = min(ZEROS_CONNUS, key=lambda z: abs(z - t_found))
        ecart = abs(t_found - nearest)
        print(f"{i:4d} {t_found:14.9f} {val:12.2e} {nearest:18.9f}
{ecart:12.2e}")

    if manques:
        print("\n⚠ Zéros non détectés:")
        for z in manques:
            print(f"  t = {z:.9f}")

    return found_zeros


# ============================================================
# MAIN
# ============================================================

if __name__ == '__main__':
    import mpmath as mp
    mp.mp.dps = 30

    print("=" * 70)
    print("TEST DE PRÉCISION AVEC mpmath.zeta")
    print("=" * 70)

    # Test avec K petit pour vérifier
    K_test = 10**6
    t_test = 14.134725

    print(f"\nK = {K_test:,}, t = {t_test}")
    start = time.time()
    za = zeta_approx_single(t_test, K_test)
    elapsed = time.time() - start
    zt = complex(mp.zeta(0.5 + 1j * t_test))
    erreur = abs(za - zt)
    print(f"  zeta_approx = {za.real:.12f}{za.imag:+.12f}j")
    print(f"  mpmath.zeta  = {zt.real:.12f}{zt.imag:+.12f}j")
    print(f"  Erreur       = {erreur:.2e}")
    print(f"  Temps        = {elapsed:.3f} s")
```

```
    # Recherche des zéros
    print("\n" + "=" * 70)
    print("LANCEMENT DE LA RECHERCHE DES ZÉROS")
    print("=" * 70)

    # Paramètres
    K = 2_000_000  # calibré pour rester sous 3 minutes sur t in [10,60],
pas=0.05,
                   # avec la version optimisée de zeta_approx_batch (2 exp() au
lieu
                   # de 4 par terme). Mesuré: ~172.7s sur cette machine.
    print(f"\nK = {K:,}")
    print("Attention: les résultats s'afficheront en temps réel pendant le
calcul.")
    print("Ne fermez pas la fenêtre.\n")

    # Lancement
    found = find_zeros_with_display(
        t_min=10.0,
        t_max=60.0,
        K=K,
        step=0.05,
        threshold=0.01,
        batch_size=5000
    )
```