```python
import numpy as np
import math


def _next_pow2(x: int) -> int:
    n = 1
    while n < x:
        n <<= 1
    return n


def _gray(i: int) -> int:
    return i ^ (i >> 1)


def _walsh_row(N: int, k: int) -> np.ndarray:
    gk = _gray(k)
    #S
    n = np.arange(N, dtype=np.uint64)
    bits = np.bitwise_and(n, gk)
    pc = np.bit_count(bits)  # Requires numpy 1.24+
    return np.where((pc & 1) == 0, 1, -1).astype(np.int8)


def _bitcount_vec_uint64(arr: np.ndarray) -> np.ndarray:
    """
    Fast bitcount for uint64 vector. Prefer np.bit_count (NumPy>=1.24),
    fallback to unpackbits. Input is uint64 array.
    """
    if hasattr(np, "bit_count"):  # modern NumPy ufunc
        return np.bit_count(arr)
    # Fallback: view as bytes and use unpackbits
    # Each uint64 -> 8 bytes -> 64 bits. Sum bits per element.
    u8 = arr.view(np.uint8).reshape(arr.size, 8)
    # unpack bits per byte then sum
    bits = np.unpackbits(u8, axis=1)  # shape: (N, 8*8=64)
    return bits.sum(axis=1).astype(np.int64)

def _walsh_row(N: int, k: int) -> np.ndarray:
    """
    Deterministic Walsh row via Gray code and parity of bitwise intersections.
    Returns ±1 int8 vector length N.
    """
    gk = np.uint64(_gray(k))
    n = np.arange(N, dtype=np.uint64)
    bits = n & gk
    pc = _bitcount_vec_uint64(bits)  # fast bit count
    return np.where((pc & 1) == 0, 1, -1).astype(np.int8)

def truncated_hadamard(m: int, idx: int = 1) -> np.ndarray:
    if m <= 0:
        return np.zeros(1, dtype=np.int8)
    N = _next_pow2(m)
    k = idx % N
    if k == 0:
        k = 1
    row = _walsh_row(N, k)
    return row[:m]


def stride_near(T: int, frac: float, forbid=(1, 2), search_radius=None):
    if T <= 4:
        return max(1, T - 2)
    target = int(round((frac % 1.0) * T)) % T
    target = min(max(target, 2), T - 2)
    w_alias = 0.40
    w_triv = 2.50
    w_hr = 0.15
    divs = [d for d in range(2, min(64, T // 2) + 1) if T % d == 0]

    def alias_penalty(s: int) -> float:
        pen = 0.0
        for d in divs:
            step = T // d
            k = round(s / step)
            delta = abs(s - k * step) / step
            if delta < 0.5:
                pen += (0.5 - delta)
        return pen

    def harmonic_ripple(s: int, H: int = 8) -> float:
        acc = 0.0
        for r in range(2, H + 1):
            x = math.sin(math.pi * r * s / T)
            acc += 1.0 / (1e-9 + abs(x))
        return acc

    triv = {1, 2, 3, T - 1, T - 2, T - 3}
```

```python
    golden = (math.sqrt(5) - 1.0) * 0.5
    prefer = int(round(golden * T)) % T
    candidates = [s for s in range(2, T - 1) if math.gcd(s, T) == 1]
    best_s = 2
    best_score = float("inf")
    for s in candidates:
        base = abs(s - target)
        pen_alias = alias_penalty(s)
        pen_triv = w_triv if (s in triv or s in forbid) else 0.0
        pen_hr = harmonic_ripple(s)
        reward = 0.05 * abs(s - prefer)
        score = base + w_alias * pen_alias + pen_triv + w_hr * pen_hr + reward
        if score < best_score:
            best_score = score
            best_s = s
    return best_s


# Simulation
#clauses = [
#    [1, 2, 3],  # x or y or z
#    [-1, -2, 3],  # ~x or ~y or z
#    [-1, 2, -3],  # ~x or y or ~z
#    [1, -2, -3]  # x or ~y or ~z
#]  # Classic UNSAT example

#n = 3  # variables

clauses = [
    [1, 2, 3, 4], [1, 2, 3, -4], [1, 2, -3, 4], [1, 2, -3, -4],
    [1, -2, 3, 4], [1, -2, 3, -4], [1, -2, -3, 4], [1, -2, -3, -4],
    [-1, 2, 3, 4], [-1, 2, 3, -4], [-1, 2, -3, 4], [-1, 2, -3, -4],
    [-1, -2, 3, 4], [-1, -2, 3, -4], [-1, -2, -3, 4], [-1, -2, -3, -4]
]
n = 4

T = 1000  # steps for annealing
frac = (math.sqrt(5) - 1) / 2  # golden ratio conjugate for irrational flavor
stride = stride_near(T, frac)
print(f"Anti-aliasing stride: {stride}")

# Initial spins (variables as -1/1)
spins = np.random.choice([-1, 1], n)


# Energy: number of unsatisfied clauses (lower = more resonant harmony)
def energy(spins):
    unsat = 0
    for clause in clauses:
        sat = any(spins[abs(lit) - 1] == (1 if lit > 0 else -1) for lit in clause)
        if not sat:
            unsat += 1
    return unsat


current_energy = energy(spins)
temp = 2.0  # initial annealing temp

for t in range(T):
    # Stride-guided var selection to avoid aliasing cycles
    var_idx = (t * stride) % n

    # Hadamard-probed update direction (deterministic Walsh for resonance probe)
    h_probe = truncated_hadamard(n, t % 32)  # Cycle through indices

    # Proposed flip with Hadamard influence (emulate feedback alignment)
    new_spins = spins.copy()
    flip_prob = (h_probe[var_idx] + 1) / 2  # Bias flip based on probe
    if np.random.rand() < flip_prob:
        new_spins[var_idx] = -new_spins[var_idx]

    new_energy = energy(new_spins)
    delta = new_energy - current_energy

    # Accept with annealing (noise-gate implicit in temp decay)
    if delta < 0 or np.random.rand() < math.exp(-delta / temp):
        spins = new_spins
        current_energy = new_energy

    temp *= 0.999  # Gradual annealing for resonance buildup

    if t % 100 == 0:
        print(f"Step {t}: Energy {current_energy} (resonance building...)")

# Final check
print(f"\nFinal energy: {current_energy}")
if current_energy == 0:
    print("Full resonance: SAT instance!")
```

```
else:
    print("Resonance decay barrier hit: UNSAT confirmed. AO pure—no SR fluff needed.")

# Bonus: Coherence metric (von Mises kappa proxy for phase harmony)
phases = np.arccos(spins)  # Map spins to phases for resonance viz
mean_vec = np.mean(np.exp(1j * phases))
kappa_proxy = abs(mean_vec)  # 1 = perfect alignment, 0 = chaos
print(f"Final coherence (kappa proxy): {kappa_proxy:.4f}")
```