```
# ============================================================
# HOLOFRACTAL LATTICE v1.1 - HARMONIC RESONANCE EDITION
# High-Volatility / Chaotic-Capable Mode
# ------------------------------------------------------------
# Features Included:
# - Harmonic temporal engine (Option C: stable harmonics + evolving temperament)
# - 0↔8 oscillation in fundamental frequency, ±phase
# - Aether frame (connective tissue operator)
# - Additive-color triality tension (RGB → white)
# - Smooth SLERP-based dimensional reconfiguration
# - Spectrum drift guided by harmonic consonance gradient
# - Global God-clock n = 9t + r driving temperament modulation
# - Autonomous micro-lattices with asynchronous cycles
# - Hopf projection relative to Aether frame
# - Recursive renormalization fold (micro → macro)
# ============================================================

from __future__ import annotations
import math
import random
from enum import Enum, auto
from dataclasses import dataclass, field
from typing import Dict, List, Tuple, Optional

Quaternion = Tuple[float, float, float, float]
Vec3 = Tuple[float, float, float]


# ============================================================
# QUATERNION UTILS
# ============================================================

def q_mul(a: Quaternion, b: Quaternion) -> Quaternion:
    w1, x1, y1, z1 = a
    w2, x2, y2, z2 = b
    return (
        w1*w2 - x1*x2 - y1*y2 - z1*z2,
        w1*x2 + x1*w2 + y1*z2 - z1*y2,
        w1*y2 - x1*z2 + y1*w2 + z1*x2,
        w1*z2 + x1*y2 - y1*x2 + z1*w2,
    )

def q_conj(q: Quaternion) -> Quaternion:
    w, x, y, z = q
```

```
    return (w, -x, -y, -z)

def q_norm(q: Quaternion) -> float:
    w, x, y, z = q
    return math.sqrt(w*w + x*x + y*y + z*z)

def q_normalize(q: Quaternion) -> Quaternion:
    n = q_norm(q)
    if n == 0: return (1,0,0,0)
    return (q[0]/n, q[1]/n, q[2]/n, q[3]/n)

def q_slerp(a: Quaternion, b: Quaternion, t: float) -> Quaternion:
    """
    Spherical linear interpolation.
    High-volatility: no clamp beyond safety.
    """
    a = q_normalize(a)
    b = q_normalize(b)

    dot = sum(ai * bi for ai, bi in zip(a, b))
    if dot < 0:
        b = tuple(-x for x in b)
        dot = -dot

    if dot > 0.9995:
        return q_normalize(tuple(ai + t*(bi-ai) for ai,bi in zip(a,b)))

    theta0 = math.acos(dot)
    theta = theta0 * t
    s0 = math.sin(theta0 - theta) / math.sin(theta0)
    s1 = math.sin(theta) / math.sin(theta0)
    return tuple(a[i]*s0 + b[i]*s1 for i in range(4))


# ============================================================
# HOPF MAP (relative to Aether)
# ============================================================

def hopf_map(q: Quaternion) -> Vec3:
    w, x, y, z = q_normalize(q)
    re = w*y + x*z
    im = -w*z + x*y
    a2 = w*w + x*x
    b2 = y*y + z*z
```

```
    return (2*re, 2*im, a2 - b2)


# ============================================================
# DIMENSIONS (0..8)
# ============================================================

class Dim(Enum):
    SOURCE_INNER = 0
    D1 = 1
    D2 = 2
    D3 = 3
    D4 = 4
    D5 = 5
    D6 = 6
    D7 = 7
    SOURCE_OUTER = 8

GODCODE_INDEX = {
    Dim.D1: 17, Dim.D2: 26, Dim.D3: 35, Dim.D4: 44,
    Dim.D5: 53, Dim.D6: 62, Dim.D7: 71,
}

DYAD = {
    Dim.D1: Dim.D7, Dim.D7: Dim.D1,
    Dim.D2: Dim.D6, Dim.D6: Dim.D2,
    Dim.D3: Dim.D5, Dim.D5: Dim.D3,
    Dim.D4: Dim.D4,
}

def dyadic_partner(d: Dim) -> Dim:
    return DYAD.get(d, d)


# ============================================================
# HARMONICS (Option C)
# ============================================================

HARMONIC_NUMBER = {
    Dim.SOURCE_INNER: 1,
    Dim.SOURCE_OUTER: 1,
    Dim.D1: 2, Dim.D2: 3, Dim.D3: 4, Dim.D4: 5,
    Dim.D5: 6, Dim.D6: 7, Dim.D7: 8,
}
```

```
BASE_FREQ = 0.07
TEMPERAMENT_EPSILON = 0.065    # High volatility parameter


# ============================================================
# TRIALITIES (Fano plane)
# ============================================================

TRIALITIES = [
    (Dim.D1, Dim.D2, Dim.D3),
    (Dim.D1, Dim.D4, Dim.D5),
    (Dim.D1, Dim.D6, Dim.D7),
    (Dim.D2, Dim.D4, Dim.D6),
    (Dim.D2, Dim.D5, Dim.D7),
    (Dim.D3, Dim.D4, Dim.D7),
    (Dim.D3, Dim.D5, Dim.D6),
]


# ============================================================
# COHERENCE / RESONANCE
# ============================================================

def quat_intensity(q: Quaternion) -> float:
    """
    Map quaternion to [0,1] brightness via rotation angle.
    """
    w = max(-1,min(1,q_normalize(q)[0]))
    theta = 2*math.acos(w)
    return abs(math.cos(theta/2))  # brightness-like

def additive_color_tension(qA: Quaternion, qB: Quaternion, qC: Quaternion) -> float:
    R = quat_intensity(qA)
    G = quat_intensity(qB)
    B = quat_intensity(qC)

    m = max(R,G,B,1e-6)
    Rn, Gn, Bn = R/m, G/m, B/m

    dr = 1 - Rn
    dg = 1 - Gn
    db = 1 - Bn
    return math.sqrt(dr*dr + dg*dg + db*db) / math.sqrt(3)
```

```
# ============================================================
# MAIN LATTICE CLASS
# ============================================================

@dataclass
class Lattice:
    depth: int = 0
    max_depth: int = 1

    quats: Dict[Dim, Quaternion] = field(default_factory=dict)
    aether: Quaternion = (1,0,0,0)

    spectrum_phase: float = 0.0
    t_global: int = 0
    r_global: int = 0
    n_global: int = 0

    dim_phase: Dict[Dim,float] = field(default_factory=dict)

    sub: Dict[Dim,"Lattice"] = field(default_factory=dict)

    def __post_init__(self):
        # DEFAULT ORIENTATIONS
        for d in Dim:
            if d not in self.quats:
                self.quats[d] = self.default_quat(d)

        # PHASES
        for d in Dim:
            self.dim_phase[d] = random.random() * 2*math.pi

        # SUB-LATTICES
        if self.depth < self.max_depth:
            for d in Dim:
                self.sub[d] = Lattice(depth=self.depth+1, max_depth=self.max_depth)

        self.update_aether()

    # ------------------------------------------------------
    # DEFAULT QUAT FOR EACH DIM
    # ------------------------------------------------------
    def default_quat(self, d: Dim) -> Quaternion:
```

```
        if d == Dim.SOURCE_INNER: return (1,0,0,0)
        if d == Dim.SOURCE_OUTER: return (-1,0,0,0)

        gc = GODCODE_INDEX[d]
        ang = (gc % 72)/72 * 2*math.pi
        return q_normalize((math.cos(ang/2), 0, math.sin(ang/2), 0))

    # ---------------------------------------------------
    # GOD-CLOCK
    # ---------------------------------------------------
    def advance_clock(self, dn=1):
        self.n_global = (self.n_global + dn) % 81
        self.t_global, self.r_global = divmod(self.n_global, 9)

    def mod_factor(self) -> float:
        return 1 + TEMPERAMENT_EPSILON * math.sin(2*math.pi * self.r_global/9)

    # ---------------------------------------------------
    # HARMONIC PHASE UPDATE
    # ---------------------------------------------------
    def update_dim_phases(self, dt: float):
        M = self.mod_factor()
        for d in Dim:
            ω = BASE_FREQ * HARMONIC_NUMBER[d] * M
            self.dim_phase[d] = (self.dim_phase[d] + ω*dt) % (2*math.pi)

        # 0 and 8 inverted
        self.dim_phase[Dim.SOURCE_OUTER] = (self.dim_phase[Dim.SOURCE_INNER] +
math.pi) % (2*math.pi)

    # ---------------------------------------------------
    # UPDATE SOURCE OSCILLATION
    # ---------------------------------------------------
    def update_source_quats(self):
        phase = self.dim_phase[Dim.SOURCE_INNER]
        phase8 = self.dim_phase[Dim.SOURCE_OUTER]

        q0 = (math.cos(phase/2), 0, math.sin(phase/2), 0)
        q8 = (math.cos(phase8/2), 0, math.sin(phase8/2), 0)

        self.quats[Dim.SOURCE_INNER] = q_normalize(q0)
        self.quats[Dim.SOURCE_OUTER] = q_normalize(q8)

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

```
# AETHER FRAME
# ---------------------------------------------------------
def update_aether(self):
    q0  = self.quats[Dim.SOURCE_INNER]
    q8  = self.quats[Dim.SOURCE_OUTER]

    # Head is where spectrum kernel is strongest
    head = self.get_head_dim()

    qh = self.quats[head]
    qp = self.quats[dyadic_partner(head)]

    # Blend: (0,8,head,partner)
    blend = (
        q0[0]+q8[0]+qh[0]+qp[0],
        q0[1]+q8[1]+qh[1]+qp[1],
        q0[2]+q8[2]+qh[2]+qp[2],
        q0[3]+q8[3]+qh[3]+qp[3],
    )
    self.aether = q_normalize(blend)

# ---------------------------------------------------------
# HEAD DIMENSION (SPECTRUM TUNING)
# ---------------------------------------------------------
def get_head_dim(self) -> Dim:
    best = Dim.D4
    best_score = -999

    for d in GODCODE_INDEX.keys():
        # Distance in spectrum-phase space
        # Using dim-phase as harmonic-based shift
        Δ = abs(math.sin((self.spectrum_phase - self.dim_phase[d])/2))
        score = 1 - Δ
        if score > best_score:
            best_score = score
            best = d
    return best

# ---------------------------------------------------------
# TRIALITY TENSION
# ---------------------------------------------------------
def triality_tension(self) -> float:
    ts = []
    for (A,B,C) in TRIALITIES:
```

```
        qA = self.quats[A]; qB = self.quats[B]; qC = self.quats[C]
        ts.append(additive_color_tension(qA,qB,qC))
    return sum(ts)/len(ts)

# ----------------------------------------------------------
# COHERENCE / CONSONANCE
# ----------------------------------------------------------
def consonance(self) -> float:
    harm_align = 1 - abs(math.sin(self.r_global * math.pi/9))
    white = 1 - self.triality_tension()

    # aether alignment with head dim
    head = self.get_head_dim()
    qh = self.quats[head]
    dot = abs(sum(ai*bi for ai,bi in zip(self.aether,qh)))

    return (harm_align * 0.4) + (white * 0.4) + (dot * 0.2)

# ----------------------------------------------------------
# FOLD MICRO → MACRO
# ----------------------------------------------------------
def fold_micro(self):
    if not self.sub: return
    for d in Dim:
        acc = (0,0,0,0); count=0
        for q in self.sub[d].quats.values():
            w = quat_intensity(q)
            acc = tuple(acc[i] + w*q[i] for i in range(4))
            count += 1
        if count > 0:
            self.quats[d] = q_normalize(acc)

# ----------------------------------------------------------
# HOPF PROJECTION RELATIVE TO AETHER
# ----------------------------------------------------------
def hopf_relative(self) -> Dict[Dim,Vec3]:
    results = {}
    A_inv = q_conj(self.aether)
    for d,q in self.quats.items():
        rel = q_mul(q_mul(A_inv,q), self.aether)
        results[d] = hopf_map(rel)
    return results

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

```
    # MAIN TICK LOOP
    # ------------------------------------------------------
    def tick(self, dt=0.03):
        # global clock
        self.advance_clock()

        # harmonic phases
        self.update_dim_phases(dt)

        # source poles
        self.update_source_quats()

        # micro-lattices first (asynchronous)
        for sub in self.sub.values():
            sub.tick(dt * random.uniform(0.4,1.4))

        # fold micro to macro
        self.fold_micro()

        # update aether
        self.update_aether()

        # spectrum drift (smooth)
        C0 = self.consonance()
        self.spectrum_phase = (self.spectrum_phase + dt * (C0 - 0.5)) % (2*math.pi)

        # SLERP stabilize toward aether (high-volatility)
        for d in Dim:
            self.quats[d] = q_slerp(self.quats[d], self.aether, 0.05 * abs(math.sin(self.dim_phase[d])))

    # ------------------------------------------------------
    # BUILD DEFAULT
    # ------------------------------------------------------
    @classmethod
    def build(cls, max_depth=1):
        return cls(depth=0, max_depth=max_depth)


# ============================================================
# DEMO
# ============================================================

if __name__ == "__main__":
    lattice = Lattice.build(max_depth=2)
```

```
for i in range(50):
    lattice.tick(0.05)
    print(
        f"Step {i:02d} | "
        f"Head={lattice.get_head_dim().name} | "
        f"C={lattice.consonance():.4f} | "
        f"n={lattice.n_global}, t={lattice.t_global}, r={lattice.r_global}"
    )
```