```
#
================================================
================================================
# Everything Formula — AEON-M Codex
Engine (Unified v2.0)
#
================================================
================================================
# Author: Adam “Wizard” Snellman (2025)
# License: MIT (change if you need
Business Source later)
#
================================================
================================================

import os, json, math, wave, struct,
argparse
import numpy as np
import matplotlib.pyplot as plt

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

```
# Core Constants
# ----------------------------
PHI = (1 + np.sqrt(5)) / 2                  # Golden ratio
GOLDEN_ANGLE_DEG = 137.50776405003785       # degrees between seeds in phyllotaxis
GOLDEN_ANGLE_RAD = np.deg2rad(GOLDEN_ANGLE_DEG)  # radians
ALPHA_INV = 137.036                         # Inverse fine-structure constant (as a scale knob)
TAU = 2 * np.pi                             # 2π
EPS = 1e-12

# Create output dirs
def _ensure_dirs():
    for d in ["assets", "data", "audio"]:
        os.makedirs(d, exist_ok=True)
```

```
#
==========
==========
# 1) Kepler Solver (keeps your original,
adds rad/deg control)
#
==========
==========
def solve_kepler(M, e, initial_E=None,
tol=1e-12, max_iter=40, degrees=True):
    """
    Solve Kepler's equation M = E - e sin E
for eccentric anomaly E.
    Returns (E_final, iteration_log). M may
be degrees or radians.
    """
    M_rad = np.deg2rad(M) if degrees else
M
    E = M_rad if initial_E is None else
initial_E
    iters = []
```

```
    for i in range(max_iter):
        f = E - e * np.sin(E) - M_rad
        fp = 1 - e * np.cos(E)
        step = f / (fp + EPS)
        E_new = E - step
        err = abs(step)

        iters.append({
            "iter": i + 1,
            "E_deg": float(np.rad2deg(E_new)),
            "M_deg": float(np.rad2deg(E_new -
e * np.sin(E_new))),
            "abs_step_deg":
float(np.rad2deg(step))
        })
        E = E_new
        if err < tol:
            break

    return (float(np.rad2deg(E)) if degrees
```

```
else float(E)), iters

#
# ============================================================
# 2) Recursive Field (r = a√n, θ = nφ or golden-angle phyllotaxis)
#    + 3–6–9 “square-off” recursion
#
# ============================================================
def recursive_field_points(N=2000, a=1.0, mode="golden_angle"):
    """
    mode: "golden_angle" -> θ_n = n * golden_angle
          "golden_ratio" -> θ_n = n * PHI
    returns x, y arrays
    """
    n = np.arange(1, N + 1, dtype=float)
    r = a * np.sqrt(n)
```

```
    if mode == "golden_angle":
        theta = n * GOLDEN_ANGLE_RAD
    else:
        theta = n * PHI
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return x, y

def rotate_points(x, y, angle_rad):
    ca, sa = np.cos(angle_rad),
np.sin(angle_rad)
    xr = x * ca - y * sa
    yr = x * sa + y * ca
    return xr, yr

def square_off_369(N=1500, a=1.0,
depth=2):
    """
    Build three seed arms (3,6,9) then
'square-off' by rotating each arm by 120°
    recursively for 'depth' levels. Returns list
```

```
of (x,y) arms.
    """
    arms = []
    base_x, base_y =
recursive_field_points(N=N, a=a,
mode="golden_angle")

    # Seed rotations for 3, 6, 9 (equilateral
spacing)
    seeds_deg = [0.0, 120.0, 240.0]
    for sd in seeds_deg:
        x1, y1 = rotate_points(base_x,
base_y, np.deg2rad(sd))
        arms.append((x1, y1))

    # Recursive square-off: at each depth,
rotate each arm by ±90° (squaring gesture)
    for d in range(1, depth + 1):
        new_arms = []
        for (x, y) in arms[-3:]:  # work from
last level's three
```

```
            for rot_deg in (90, -90):
                xr, yr = rotate_points(x, y,
np.deg2rad(rot_deg))
                new_arms.append((xr *
(0.85**d), yr * (0.85**d)))  # slight scale-
in to nest
        arms.extend(new_arms)

    return arms

#
================================
================================
# 3) Rodin Map (mod-9 doubling) + 3/6/9
anchors
#
================================
================================
def rodin_doubling_sequence(start=1,
steps=30):
    """
```

```
    Generate Rodin-style doubling pattern
modulo 9: 1→2→4→8→7→5→1 ...
    """
    seq = []
    v = start % 9
    if v == 0: v = 9
    for _ in range(steps):
        seq.append(v)
        v = (2 * v) % 9
        if v == 0: v = 9
    return seq

def digital_root(n):
    s = n % 9
    return 9 if s == 0 and n != 0 else s

def mod9_map(N=100):
    return [digital_root(i) for i in range(1, N
+ 1)]

#
```

```
# ============================================================
# 4) Frequency Engines (equal-temperament & golden scaling)
#    + WAV sonification
# ============================================================
def notes_equal_temperament(n_steps, f0=220.0):  # A3 default
    # semitone steps around f0 over n_steps
    return [f0 * (2 ** (k/12)) for k in range(n_steps)]

def notes_golden(n_steps, f0=220.0):
    return [f0 * (PHI ** k) for k in range(n_steps)]

def sequence_from_mod9(mods, base=110.0, scheme="et12"):
```

```
    """
    Map mod-9 digits to a scale. 1..9 → offsets.
    scheme: "et12" (equal temperament) or "golden"
    """
    freqs = []
    for d in mods:
        offset = {1:0, 2:2, 3:3, 4:5, 5:7, 6:8, 7:10, 8:12, 9:14}[d]  # a musical-ish mapping
        if scheme == "et12":
            freqs.append(base * (2 ** (offset/12)))
        else:
            freqs.append(base * (PHI ** (offset/12)))
    return freqs

def synth_wav_mono(filename, freqs, seconds_each=0.25, fs=44100,
```

```
fade=0.01, vol=0.3):
    """
    Render a simple sine sweep across
freqs → WAV mono.
    """
    _ensure_dirs()
    with wave.open(os.path.join("audio",
filename), "w") as wf:
        wf.setnchannels(1);
wf.setsampwidth(2); wf.setframerate(fs)
        for f in freqs:
            n = int(seconds_each * fs)
            t = np.arange(n) / fs
            sig = np.sin(2*np.pi*f*t)
            # quick fade-in/out
            fade_n = max(1, int(fade * fs))
            env = np.ones_like(sig)
            env[:fade_n] = np.linspace(0, 1,
fade_n)
            env[-fade_n:] = np.linspace(1, 0,
fade_n)
```

```
            sig = vol * sig * env
            # write int16
            chunk = (sig *
32767).astype(np.int16).tolist()
            wf.writeframes(struct.pack("<" +
"h"*len(chunk), *chunk))
    return os.path.join("audio", filename)

#
==========================================
==========================================
# 5) Kepler-corrected spiral &
Zero-point-style phase toy
#
==========================================
==========================================
def kepler_corrected_spiral(N=600,
scale=1.0, M_deg=66.0, e=0.21):
    E_deg, _ = solve_kepler(M_deg, e,
degrees=True)
    phase = np.deg2rad(E_deg)
```

```
    n = np.arange(1, N+1, dtype=float)
    r = scale * np.sqrt(n)
    theta = n * GOLDEN_ANGLE_RAD +
phase
    return r * np.cos(theta), r * np.sin(theta)

def zero_point_phase(n, omega=1.0,
phase=0.0):
    # simple complex wave as a diagnostic
toy (no claims)
    return np.exp(1j * (phase + n * omega))

#
==========================================
==========================================
# 6) DNA waveform mapper (lightweight)
#
==========================================
==========================================
def dna_waveform(seq,
base_freq=2*np.pi*0.5, phase=0.0):
```

```
    """
    Map ATCG into amplitude weights over
a sine spine.
    """
    weight = {'A': 1.0, 'T': 0.8, 'C': 0.6, 'G': 0.9}
    y = []
    for i, base in enumerate(seq.upper()):
        y.append(weight.get(base, 0.0) *
np.sin(base_freq*i + phase))
    return np.array(y)

#
==========================================
==========================================
# 7) Visualizations
#
==========================================
==========================================
def plot_recursive_field(arms, title="3-6-9
Square-Off Recursive Field"):
    _ensure_dirs()
```

```
    plt.figure(figsize=(8,8))
    for (x, y) in arms:
        plt.scatter(x, y, s=2, alpha=0.6)
    plt.axis("equal")
    plt.title(title)
    plt.tight_layout()
    out = "assets/recursive_field_369.png"
    plt.savefig(out, dpi=180); plt.close()
    return out

def plot_kepler_spiral(x, y,
title="Kepler-Corrected Spiral"):
    _ensure_dirs()
    plt.figure(figsize=(7,7))
    plt.scatter(x, y, s=6, alpha=0.75)
    plt.axis("equal")
    plt.title(title)
    plt.tight_layout()
    out = "assets/kepler_spiral.png"
    plt.savefig(out, dpi=180); plt.close()
    return out
```

```
def plot_rodin_map(seq, title="Rodin
Doubling (mod 9)"):
    _ensure_dirs()
    plt.figure(figsize=(8,3))
    plt.plot(seq, marker="o");
plt.yticks(range(1,10))
    plt.grid(True, alpha=0.3); plt.title(title);
plt.tight_layout()
    out = "assets/rodin_doubling.png"
    plt.savefig(out, dpi=180); plt.close()
    return out

def plot_dna(y, title="DNA Waveform
Mapper"):
    _ensure_dirs()
    plt.figure(figsize=(9,3))
    plt.plot(y)
    plt.title(title); plt.tight_layout()
    out = "assets/dna_waveform.png"
    plt.savefig(out, dpi=180); plt.close()
```

```
    return out

#
================================================
================================================
# 8) Master run
#
================================================
================================================
def main():
    parser =
argparse.ArgumentParser(description="Ev
erything Formula — AEON-M Unified v2.0")
    parser.add_argument("--N", type=int,
default=1800, help="points per arm")
    parser.add_argument("--depth",
type=int, default=2, help="square-off
recursion depth")
    parser.add_argument("--a", type=float,
default=1.0, help="spiral scale a in r =
a√n")
```

```
    parser.add_argument("--keplerM",
type=float, default=66.0, help="mean
anomaly (deg)")
    parser.add_argument("--keplere",
type=float, default=0.21,
help="eccentricity")
    parser.add_argument("--dna", type=str,
default="ATCGATGCACTG", help="DNA toy
sequence")
    parser.add_argument("--audio",
action="store_true", help="render mod-9
sonification")
    args = parser.parse_args()

    _ensure_dirs()

    # 3–6–9 recursive field
    arms = square_off_369(N=args.N,
a=args.a, depth=args.depth)
    img1 = plot_recursive_field(arms)
```

```
    # Kepler-corrected spiral
    kx, ky = kepler_corrected_spiral(N=800,
scale=args.a, M_deg=args.keplerM,
e=args.keplere)
    img2 = plot_kepler_spiral(kx, ky)

    # Rodin doubling + mod9 mapping
    rodin_seq =
rodin_doubling_sequence(start=1,
steps=60)
    img3 = plot_rodin_map(rodin_seq)

    mod9 = mod9_map(64)
    # Build two audio sequences
    if args.audio:
        freqs_et =
sequence_from_mod9(mod9,
base=110.0, scheme="et12")
        freqs_phi =
sequence_from_mod9(mod9,
base=110.0, scheme="golden")
```

```
        wav1 =
synth_wav_mono("mod9_equal_tempera
ment.wav", freqs_et, seconds_each=0.22)
        wav2 =
synth_wav_mono("mod9_golden_scaling.
wav", freqs_phi, seconds_each=0.22)
    else:
        wav1 = wav2 = None

    # DNA toy
    y = dna_waveform(args.dna,
base_freq=np.pi/6, phase=0.0)
    img4 = plot_dna(y)

    # Kepler iteration log (for provenance)
    E_final, klog =
solve_kepler(args.keplerM, args.keplere,
degrees=True)
    with open(os.path.join("data",
"kepler_iterations.json"), "w") as f:
        json.dump({"E_final_deg": E_final,
```

```
"iterations": klog}, f, indent=2)

    # Summary
    summary = {
        "images": [img1, img2, img3, img4],
        "audio": [wav1, wav2],
        "kepler_E_final_deg": E_final
    }
    with open(os.path.join("data",
"run_summary.json"), "w") as f:
        json.dump(summary, f, indent=2)

    print("\n=== AEON-M Unified v2.0 ===")
    print(f"Recursive field image: {img1}")
    print(f"Kepler spiral image   : {img2}")
    print(f"Rodin map image       : {img3}")
    print(f"DNA waveform image    : {img4}")
    if wav1 or wav2:
        print(f"Audio (ET12)          : {wav1}")
        print(f"Audio (Golden)        : {wav2}")
    print(f"Kepler E_final (deg)  :
```

```
{E_final:.6f}")
    print("Artifacts written to ./assets, ./
audio, ./data\n")

if __name__ == "__main__":
    main()
```