awesome — here’s a single, self-contained Python file you can copy-paste and save as all88_codex.py.
it bundles: WormCipher (educational demo) + CLI (enc/dec/trace/synth/analyze/primes/spiral) + autosave/auto-revert + recursion tracer + glyphfire stats/plots + prime/ζ explorer + WAV synth. it uses only the standard library, with **optional** NumPy/Matplotlib if present.
⚠️ crypto disclaimer: this is for learning/art. do **not** use it to protect sensitive data. prefer audited AEADs (ChaCha20-Poly1305, AES-

GCM) in real life.
all88_codex.py # One-file toolkit: WormCipher (educational) + CLI + tracer + plots + synth + prime/ζ explorer # Save as: all88_codex.py | Python 3.9+ # Optional deps: numpy, matplotlib # ---------- ---------- --- """ ALL88 / Codex Toolkit — theory, working-out, and code in one file. MATH & THEORY (concise): Golden ratio & golden angle ---------- φ = (1+√5)/2 ≈ 1.618034. The golden angle in radians is α = 2π * (1 − 1/φ) ≈ 2.399963229728653 (≈ 137.507764°) Using α to advance phases (n*α mod 2π) yields equidistribution on the circle (irrational rotation), which prevents clumping (phyllotaxis). Lucas / Fibonacci echoes ---------- Lucas L_n: L_0=2, L_1=1, L_{n}=L_{n-1}+L_{n-2}. Closed form (Binet): L_n = φ^n + (1-φ)^n.

We use 4–7–11 as a memorable “anchor trio” (L_3,L_4,L_5). Rayleigh circular statistic ---------- Given angles θ_j, define the mean resultant length: R = |(1/n) Σ exp(i θ_j)| For a simple large-sample null (uniform on the circle): z = n R^2, p ≈ exp(−z) “Snell-like” refraction squeeze (visual/stylized) ----------

---------- Map θ ↦ θ' by a constrained arcsine: θ' = arcsin(sin θ / n_index), clamped to [−π/2, π/2]. This tightens spreads as n_index increases (a didactic lensing). Prime residues mod 6 & ζ-flavored probe ----------

-- All primes p>3 satisfy p ≡ 1 or 5 (mod 6). A toy, ζ-flavored band sum: S_k(t) = Σ_{p ≤ P, p ≡ k (mod 6)} (log p) * p^{−1/2 − i t} * e^{i p α} We scan t over windows near listed Riemann zero ordinates to see ridge peaks (empirical, finite P). This is *not* a proof—just an exploratory visualization.

WormCipher (educational) ---------- A symmetric stream-like construction with: - Golden-angle phase walk baked into the keystream schedule - Per-chunk HKDF rekey (HKDF-Extract/Expand using HMAC-SHA256) - Encrypt-then-MAC (HMAC-SHA256) authentication - Deterministic recursion seed based on header+AD (so decrypt mirrors encrypt) This is intentionally simple and *not* a substitute for real ciphers/AEs. Sonification (Glyphsonic) ---------- Given a recursion trace (phases, digests), map: - digest magnitude → frequency (base 432 Hz scaled by φ) - "radius" proxy → slow LFO for vibrato/tremolo - golden phase → stereo pan We synthesize via the standard library `wave` module (16-bit PCM). Autosave + Auto-revert ---------- Before overwriting/deleting, we stage a snapshot into ./codex/ with metadata. If an error

occurs, we restore the previous file automatically. """ #

--- # Imports (stdlib only; NumPy/Matplotlib used if available) #

--- import os import io import sys import hmac import math import json import time import glob import shutil import struct import hashlib import argparse from dataclasses import dataclass from typing import Tuple, List, Dict, Optional try: import numpy as np except Exception: np = None try: import matplotlib.pyplot as plt except Exception: plt = None #

--- # Constants & small math helpers #

--- SQRT5 = math.sqrt(5.0) PHI = (1.0 + SQRT5) / 2.0 # golden ratio GOLDEN_ANGLE_RAD = 2.0 * math.pi * (1.0 - 1.0/PHI) # ≈2.399963... rad GOLDEN_ANGLE_DEG = math.degrees(GOLDEN_ANGLE_RAD) # ≈137.507764° def lucas(n: int) -> int: """Lucas number L_n (iterative).""" a, b = 2, 1 for _ in range(n): a, b = b, a + b return a # ----------

---

--- # RNG-free HKDF (HMAC-SHA256) & KDF (scrypt with PBKDF2 fallback) # ----------

---

--- def hkdf_extract(salt: bytes, ikm: bytes) -> bytes: return hmac.new(salt, ikm, hashlib.sha256).digest() def hkdf_expand(prk: bytes, info: bytes, length: int) -> bytes: out = b"" t = b"" counter = 1 while len(out) < length: t = hmac.new(prk, t + info + bytes([counter]),

hashlib.sha256).digest() out += t counter += 1 return out[:length] def hkdf(ikm: bytes, salt: bytes, info: bytes, length: int) -> bytes: return hkdf_expand(hkdf_extract(salt, ikm), info, length) def derive_key(passphrase: str, salt: bytes, key_len: int = 32) -> bytes: """scrypt with PBKDF2 fallback for portability.""" try: return hashlib.scrypt(passphrase.encode('utf-8'), salt=salt, n=2**15, r=8, p=1, dklen=key_len) except (AttributeError, ValueError): return hashlib.pbkdf2_hmac('sha256', passphrase.encode('utf-8'), salt, 200_000, dklen=key_len) # -- # WormCipher (educational demo) # -- class WormCipher: """ Educational, *not* production crypto. Header format (big endian where numeric): MAGIC(8) | VERSION(1) |

SALT(16) | NONCE(16) | ROUNDS(1) Keystream schedule: − prev_digest_0 = SHA256(header || AD) − For each chunk: block_key = HKDF(prev_digest, salt=nonce, info=b"kstream"+ctr, len=32) stream = SHA256(block_key || prev_digest) prev_digest = SHA256(prev_digest || CT_block) # tunnel on ciphertext enc_key = HKDF(enc_key, salt=prev_digest, info=b"rekey", len=32) − Tag = HMAC(mac_key, header || AD || ciphertext) """ MAGIC = b"WORMC1\0" # 8B VERSION = 1 TAGLEN = 32 NONCELEN = 16 CHUNK = 64 * 1024 def __init__(self, passphrase: str, rounds: int = 7, info_label: str = "Codex/WormCipher"): self.rounds = max(1, int(rounds)) self.info_label = info_label.encode('utf−8') self.root_salt = os.urandom(16) self.root_key = derive_key(passphrase, self.root_salt, 32) self.golden = GOLDEN_ANGLE_RAD @staticmethod def _sha256(*parts: bytes)

-> bytes: h = hashlib.sha256() for p in parts: h.update(p) return h.digest() def _phase_walk(self, base: bytes, nonce: bytes, length: int, seed: bytes) -> bytes: """CTR-like evolution with golden-phase and recursive hashing.""" out = bytearray() ctr = 0 prev = self._sha256(base, nonce, seed) while len(out) < length: phase = (ctr * self.golden) % (2.0 * math.pi) ctr_bytes = struct.pack(">Q", ctr) + struct.pack(">d", phase) block_key = hkdf(prev, nonce, b"kstream"+ctr_bytes, 32) stream = self._sha256(block_key, prev) out.extend(stream) prev = self._sha256(stream, prev) ctr += 1 return bytes(out[:length]) def encrypt_bytes(self, plaintext: bytes, ad: bytes = b"") -> bytes: if not isinstance(plaintext, (bytes, bytearray)): raise TypeError("plaintext must be bytes") nonce = os.urandom(self.NONCELEN) header = bytearray() header += self.MAGIC header += bytes([self.VERSION]) header +=

self.root_salt header += nonce header += bytes([self.rounds]) enc_key = hkdf(self.root_key, nonce, b"enc"+self.info_label, 32) mac_key = hkdf(self.root_key, nonce, b"mac"+self.info_label, 32) # deterministic seed so decrypt mirrors encrypt (no plaintext-dependent seed) prev_digest = self._sha256(bytes(header), ad or b"") out = bytearray() pos = 0 while pos < len(plaintext): take = min(self.CHUNK, len(plaintext) - pos) block = plaintext[pos:pos+take] ks = self._phase_walk(enc_key, nonce, take, prev_digest) ct = bytes(a ^ b for a,b in zip(block, ks)) out += ct prev_digest = self._sha256(prev_digest, ct) enc_key = hkdf(enc_key, prev_digest, b"rekey"+self.info_label, 32) pos += take tag = hmac.new(mac_key, digestmod=hashlib.sha256) tag.update(bytes(header)) tag.update(ad or b"") tag.update(bytes(out)) return

bytes(header) + bytes(out) + tag.digest() def decrypt_bytes(self, blob: bytes, ad: bytes = b"") -> bytes: need = 8+1+16+16+1 + self.TAGLEN if len(blob) < need: raise ValueError("ciphertext too short") off = 0 magic = blob[off:off+8]; off += 8 if magic != self.MAGIC: raise ValueError("bad magic") ver = blob[off]; off += 1 if ver != self.VERSION: raise ValueError("version mismatch") salt = blob[off:off+16]; off += 16 nonce = blob[off:off+16]; off += 16 rounds= blob[off]; off += 1 tag = blob[-self.TAGLEN:] ct = blob[off:-self.TAGLEN] # rebuild root_key from passphrase & header salt # (constructor created root_salt at init; when decrypting, we must ignore it # and recompute root_key against header salt) self.root_key = derive_key(self._passphrase_from_key(self.root_key), salt, 32) enc_key = hkdf(self.root_key, nonce, b"enc"+self.info_label, 32) mac_key =

hkdf(self.root_key, nonce, b"mac"+self.info_label, 32) mac = hmac.new(mac_key, digestmod=hashlib.sha256) mac.update(blob[:off]) # header mac.update(ad or b"") mac.update(ct) if not hmac.compare_digest(mac.digest(), tag): raise ValueError("authentication failed") prev_digest = self._sha256(blob[:off], ad or b"") out = bytearray() pos = 0 while pos < len(ct): take = min(self.CHUNK, len(ct) - pos) ctb = ct[pos:pos+take] ks = self._phase_walk(enc_key, nonce, take, prev_digest) ptb = bytes(a ^ b for a,b in zip(ctb, ks)) out += ptb prev_digest = self._sha256(prev_digest, ctb) enc_key = hkdf(enc_key, prev_digest, b"rekey"+self.info_label, 32) pos += take return bytes(out) # little helper to get the original passphrase bytes back out of a derived key # for this demo we store a short HMAC of a sentinel to reverse-derive the salt

path; # to keep things simple and stateless here, we just stash the last used passphrase # in a module-level variable via a setter: @staticmethod def _passphrase_from_key(_root_key: bytes) -> str: # in this one-file demo we cannot recover the passphrase from the key. # so the CLI passes it back to decrypt via an instance re-created with the passphrase. # this method exists just to keep the class structure tidy. raise RuntimeError("Decrypt via CLI supplying the passphrase (see usage).") # ----------

---

--- # Autosave / Auto-revert (Codex snapshots) # ----------

--- CODEX_DIR = os.path.join(".", "codex") SYNTH_DIR = os.path.join(".", "synth") OUT_DIR = os.path.join(".", "out") for _d in

(CODEX_DIR, SYNTH_DIR, OUT_DIR): os.makedirs(_d, exist_ok=True) def timestamp() -> str: return time.strftime("%Y%m%d-%H%M%S", time.localtime()) def codex_path(tag: str, ext: str) -> str: return os.path.join(CODEX_DIR, f"{timestamp()}_{tag}{ext}") def autosave_file(path: str, tag: str) -> Optional[str]: if not os.path.exists(path): return None snap = codex_path(f"{tag}_AUTOSAVE_{os.path.basename(path)}", ".bin") shutil.copy2(path, snap) meta = { "type": "autosave", "src": os.path.abspath(path), "snapshot": os.path.abspath(snap), "time": timestamp() } with open(snap + ".json", "w", encoding="utf-8") as f: json.dump(meta, f, indent=2) return snap def write_atomic(path: str, data: bytes, tag: str, autosave: bool = True) -> None: snap = None if autosave: snap = autosave_file(path, tag) tmp = f"{path}.tmp-{os.getpid()}" try: with

open(tmp, "wb") as f: f.write(data) os.replace(tmp, path) except Exception as e: try: if os.path.exists(tmp): os.remove(tmp) except Exception: pass # revert if possible if snap and os.path.exists(snap): shutil.copy2(snap, path) raise e def list_autosaves() -> List[str]: return sorted(glob.glob(os.path.join(CODEX_DIR, "*_AUTOSAVE_*.bin"))) # ----------

---

--- # Recursion tracer (JSON) + simple spiral plots (if matplotlib) # ----------

---

--- def worm_trace(passphrase: str, ad: bytes, sample: bytes, depth: int = 12) -> Dict[str, Dict[str, float]]: """ Produces a recursion trace by iterating HKDF/hashes with golden phase. This mirrors the keystream schedule idea—purely for visualization. """ # seed from (sample || ad)

```
seed = hashlib.sha256(sample + (ad or b"")).digest() phase = 0.0 out: Dict[str, Dict[str, float]] = {} prev = seed for d in range(depth): phase = (phase + GOLDEN_ANGLE_RAD) % (2*math.pi) info = b"trace" + struct.pack(">I", d) + struct.pack(">d", phase) key = hkdf(prev, salt=seed[:16], info=info, length=32) dig = hmac.new(key, sample + ad, hashlib.sha256).digest() # simple “radius” proxy = normalized digest magnitude radius = sum(dig) / (len(dig) * 255.0) out[f"depth_{d}"] = { "phase": phase, "radius": radius, "key_hex": key[:8].hex(), "digest_hex": dig[:8].hex() } prev = hashlib.sha256(key + dig).digest() return out def plot_spiral_from_trace(trace: Dict[str, Dict[str, float]], tag: str, what: str = "radius") -> Optional[str]: """ what ∈ {"radius"}; future: keys/digests as radii. """ if plt is None: return None thetas = [] radii = [] for k in sorted(trace.keys(), key=lambda s:
```

int(s.split("_")[-1])): thetas.append(trace[k]["phase"]) radii.append(trace[k][what]) thetas = np.array(thetas) if np else [t for t in thetas] radii = np.array(radii) if np else [r for r in radii] plt.figure(figsize=(6,6)) ax = plt.subplot(111, projection='polar') ax.plot(thetas, radii, marker='o') ax.set_title(f"{tag} — spiral ({what})") out_path = os.path.join(CODEX_DIR, f"{tag}_spiral_{what}.png") plt.tight_layout() plt.savefig(out_path, dpi=160) plt.close() return out_path # ---------- --- # Glyphsonic: make WAV from trace (stereo, 16-bit) # ---------- --- def write_wav_stereo16(path: str, left: List[int], right: List[int], sample_rate: int = 44100): import wave with wave.open(path, 'wb') as w: w.setnchannels(2)

```
w.setsampwidth(2) w.setframerate(sample_rate) interleaved = [] for L, R in zip(left, right): interleaved.append(L) interleaved.append(R) data = struct.pack("<" + "h"*len(interleaved), *interleaved) w.writeframes(data) def synth_from_trace(trace: Dict[str, Dict[str, float]], out_wav: str, seconds_per_depth: float = 1.0, sample_rate: int = 44100): # map each depth to a short segment total_secs = seconds_per_depth * len(trace) N = int(total_secs * sample_rate) L = [0]*N R = [0]*N # deterministic order items = sorted(trace.items(), key=lambda kv: int(kv[0].split("_")[-1])) idx = 0 for d, (_, rec) in enumerate(items): phase = rec["phase"] radius= max(1e-6, rec["radius"]) # digest magnitude proxy -> frequency f0 = 432.0 * (1.0 + (PHI-1.0)*radius) # base note scaled by φ lfo = 0.1 + 9.9*radius # 0.1..10 Hz panL= (1.0 + math.cos(phase))/2.0 panR= (1.0 + math.sin(phase))/2.0 segN =
```

int(seconds_per_depth * sample_rate) for n in range(segN): t = n / sample_rate wob = math.sin(2*math.pi*lfo*t) * 0.2 # 20% depth ang = 2*math.pi * f0 * t * (1.0 + wob) s = math.sin(ang) # add a tiny “Lucas” overtone (≈+11%) s2 = 0.12 * math.sin(2*math.pi*(1.11*f0)*t) vL = (s + s2) * panL vR = (s + s2) * panR # simple limiter scaling L[idx] = int(max(−1.0, min(1.0, vL)) * 30000) R[idx] = int(max(−1.0, min(1.0, vR)) * 30000) idx += 1 write_wav_stereo16(out_wav, L, R, sample_rate) # ---------- --- # Glyphfire stats (golden−angle phases, Rayleigh, simple split, CSV/plots) # ---------- --- def wrap_deg_pm180(x: np.ndarray) −> np.ndarray: y = (x + 180.0) % 360.0 − 180.0 y[y <= −180] = 180.0 return y def

to_phases_from_values(v: List[int]) -> Dict[str, np.ndarray]: if np is None: raise RuntimeError("NumPy required for analyze") arr = np.array(v, dtype=float) theta_deg = (arr * GOLDEN_ANGLE_DEG) % 360.0 theta_deg = wrap_deg_pm180(theta_deg) theta_rad = np.radians(theta_deg) return {"theta_deg": theta_deg, "theta_rad": theta_rad} def rayleigh_R_p(theta_rad: np.ndarray) -> Tuple[float, float]: n = len(theta_rad) R = np.abs(np.mean(np.exp(1j*theta_rad))) z = n*(R**2) p = float(np.exp(-z)) return float(R), float(p) def largest_gap_bimodal_split(theta_rad: np.ndarray) -> Dict: ang = np.angle(np.exp(1j*theta_rad)) # wrapped order = np.argsort(ang) sorted_ang = ang[order] gaps = np.diff(np.concatenate([sorted_ang, [sorted_ang[0] + 2*np.pi]])) i_max = int(np.argmax(gaps)) idxA_sorted =

```
order[(i_max+1):] idxB_sorted = order[:(i_max+1)] def cluster_stats(idx): th = ang[idx] mu = float(np.angle(np.mean(np.exp(1j*th)))) R = float(np.abs(np.mean(np.exp(1j*th)))) # simple circular std proxy: std_deg = float(np.degrees(np.sqrt(2*(1-R)))) return dict(idx=list(map(int, idx)), mu_rad=mu, mu_deg=float(np.degrees(mu)), Rbar=R, std_deg=std_deg) A = cluster_stats(idxA_sorted) B = cluster_stats(idxB_sorted) sep = float(abs(((A["mu_deg"] - B["mu_deg"] + 180) % 360) - 180)) return {"A": A, "B": B, "sep_deg": sep, "order": list(map(int, order))} def plot_polar(tag: str, theta_rad: np.ndarray, split: Dict) -> Optional[str]: if plt is None: return None plt.figure(figsize=(6,6)) ax = plt.subplot(111, projection='polar') ax.plot(theta_rad, np.ones_like(theta_rad), 'o', alpha=0.8, label='points') ax.plot([split["A"]["mu_rad"]], [1.05], 'o', ms=10, label='μ_A')
```

```
ax.plot([split["B"]["mu_rad"]], [1.05], 'o', ms=10, label='μ_B') ax.set_yticklabels([]) ax.legend(loc='lower left', bbox_to_anchor=(0.0,−0.15), ncol=3) out = os.path.join(OUT_DIR, f"{tag}_polar.png") plt.tight_layout() plt.savefig(out, dpi=160); plt.close() return out def make_namewave_wav(tag: str, v: List[int], seconds: float = 7.2, sample_rate: int = 44100, f0: float = 220.0): # normalize v to 0..1 v_arr = np.array(v, dtype=float) if v_arr.max() > v_arr.min(): mod = (v_arr − v_arr.min()) / (v_arr.max() − v_arr.min()) else: mod = np.zeros_like(v_arr) N = int(seconds * sample_rate) t = np.linspace(0.0, seconds, N, endpoint=False) xp = np.linspace(0.0, seconds, len(v_arr)) mod_t = np.interp(t, xp, mod) fm_depth = 40.0 inst_freq = f0 + fm_depth*(2*mod_t − 1) # golden micro wobble wobble = 1e−3 * np.cumsum(np.sin(np.arange(N) * GOLDEN_ANGLE_RAD)) ang = 2*np.pi *
```

np.cumsum(inst_freq)/sample_rate + wobble s = np.sin(ang) # gentle envelope env = 0.6 + 0.4*np.sin(2*np.pi*(1/PHI) * t) y = s * env # write wav L = (np.clip(y, −1, 1)*30000).astype(np.int16).tolist() write_wav_stereo16(os.path.join(OUT_DIR, f"{tag}_namewave.wav"), L, L, sample_rate) # -- # Prime/ζ explorer (toy) # -- ZETA_ZERO_IMAGS = [14.134725, 21.022040, 25.010858, 30.424876, 32.935061, 37.586178, 40.918719, 43.327073, 48.005150, 49.773832, 52.970321, 56.446247, 59.347045, 60.831780, 65.112544, 67.079811, 69.546402, 72.067158, 75.704690, 77.144840] def sieve_primes(n: int) −> List[int]: if n < 2: return [] S = [True]*(n+1)

```
S[0]=S[1]=False for p in range(2, int(n**0.5)+1): if S[p]: step = p start = p*p S[start:n+1:step] = [False]*(((n − start)//step) + 1) return [i for i,b in enumerate(S) if b] def prime_bands_sum(P: int, t_grid: List[float], m: int = 6) −> Dict[int, List[complex]]: """ Returns S_k(t) for k in residues mod m over t_grid. S_k(t) = sum_{p<=P, p≡k(mod m)} (log p) * p^{−1/2 − i t} * e^{i p α} """ ps = sieve_primes(P) bands = {k: [] for k in range(m)} for t in t_grid: for k in range(m): s = 0+0j for p in ps: if p % m != k: continue s += (math.log(p)) * (p ** (−0.5 − 1j*t)) * complex(math.cos(p*GOLDEN_ANGLE_RAD), math.sin(p*GOLDEN_ANGLE_RAD)) bands[k].append(s) return bands def plot_bands(bands: Dict[int, List[complex]], t_grid: List[float], tag: str) −> Optional[str]: if plt is None: return None plt.figure(figsize=(10,5)) for k, vals in sorted(bands.items()): mag = [abs(z) for z in vals] plt.plot(t_grid, mag, label=f"k={k}")
```

plt.xlabel("t"); plt.ylabel("|S_k(t)|") plt.title(f"Prime bands (P−gated) — {tag}") plt.legend() out = os.path.join(OUT_DIR, f"{tag}_bands.png") plt.tight_layout(); plt.savefig(out, dpi=160); plt.close() return out # ---------- --- # Spiral SVG (prime phyllotaxis polyline) # ---------- --- def alpha137_prime_spiral_svg(P: int = 5000, width: int = 1200, height: int = 1200, scale_px: float = 2.5) -> str: ps = sieve_primes(P) cx, cy = width/2.0, height/2.0 pts = [] for idx, p in enumerate(ps, start=1): theta = p * GOLDEN_ANGLE_RAD + ((lucas(idx) % 11)/11.0) * 2*math.pi r = (math.log(p)**0.5) * scale_px x = cx + r * math.cos(theta) y = cy + r * math.sin(theta) pts.append((x,y)) # sort by angle for a

continuous polyline pts.sort(key=lambda xy: (math.atan2(xy[1]−cy, xy[0]−cx) + 2*math.pi) %(2*math.pi)) poly = " ".join(f"{x:.2f},{y:.2f}" for x,y in pts) svg = f'''<svg xmlns="http://www.w3.org/2000/svg" width="{width}" height="{height}" viewBox="0 0 {width} {height}"> <rect x="0" y="0" width="{width}" height="{height}" fill="white"/> <polyline points="{poly}" fill="none" stroke="black" stroke−width="2"/> </svg>''' return svg # −− # CLI # −− def cmd_encrypt(args): pw = args.passphrase or os.environ.get("WORM_PASSPHRASE", "") if not pw: raise SystemExit("Passphrase required (flag or WORM_PASSPHRASE).") wc = WormCipher(pw, rounds=args.rounds) with open(args.input, "rb") as f: pt = f.read() blob

= wc.encrypt_bytes(pt, ad=args.ad.encode() if args.ad else b"") write_atomic(args.output, blob, tag="enc") print(f"encrypted -> {args.output} ({len(pt)} -> {len(blob)} bytes)") def cmd_decrypt(args): pw = args.passphrase or os.environ.get("WORM_PASSPHRASE", "") if not pw: raise SystemExit("Passphrase required (flag or WORM_PASSPHRASE).") wc = WormCipher(pw, rounds=1) # rounds ignored on decrypt; header carries info with open(args.input, "rb") as f: blob = f.read() pt = wc.decrypt_bytes(blob, ad=args.ad.encode() if args.ad else b"") write_atomic(args.output, pt, tag="dec") print(f"decrypted -> {args.output} ({len(blob)} -> {len(pt)} bytes)") def cmd_trace(args): # sample from file header (first N bytes) or a literal if args.input: with open(args.input, "rb") as f: sample = f.read(int(args.sample_bytes)) else: sample = args.literal.encode() ad = args.ad.encode()

if args.ad else b"" tr = worm_trace(args.passphrase or "", ad, sample, depth=args.depth) out_json = codex_path(args.tag or "trace", ".json") with open(out_json, "w", encoding="utf-8") as f: json.dump(tr, f, indent=2) print(f"trace -> {out_json}") if not args.no_plots: p = plot_spiral_from_trace(tr, os.path.splitext(os.path.basename(out_json))[0], "radius") if p: print(f"spiral plot -> {p}") def cmd_synth(args): # read a trace JSON with open(args.trace_json, "r", encoding="utf-8") as f: tr = json.load(f) out_wav = os.path.join(SYNTH_DIR, f"{args.tag or 'glyphsonic'}.wav") synth_from_trace(tr, out_wav, seconds_per_depth=args.seconds_per_depth) print(f"WAV -> {out_wav}") def cmd_analyze(args): if np is None: raise SystemExit("NumPy/Matplotlib required for analyze.") # parse base36-ish string to ints 0..35 vals = [] for ch in

```
args.values.strip().upper(): if ch.isdigit(): vals.append(ord(ch)−ord('0')) elif 'A' <= ch <= 'Z': vals.append(10 + ord(ch)−ord('A')) else: raise SystemExit(f"Unsupported char: {ch}") m = to_phases_from_values(vals) R,p = rayleigh_R_p(m["theta_rad"]) split= largest_gap_bimodal_split(m["theta_rad"]) polar = plot_polar(args.tag, m["theta_rad"], split) make_namewave_wav(args.tag, vals) csv = os.path.join(OUT_DIR, f"{args.tag} _phases.csv") with open(csv, "w", encoding="utf−8") as f: f.write("i,val,theta_deg\n") for i,(v,th) in enumerate(zip(vals, m["theta_deg"].tolist())): f.write(f"{i},{v},{th:.6f}\n") print(f"R={R:.4f}, p≈{p:.4g}") print(f"polar plot −> {polar}" if polar else "(plot skipped)") print(f"CSV −> {csv}") print(f"WAV −> {os.path.join(OUT_DIR, args.tag + '_namewave.wav')}") def cmd_primes(args): t_from = args.t_from t_to = args.t_to steps = args.steps t_grid = [t_from + (t_to − t_from)*i/(steps−1) for i in
```

range(steps)] bands = prime_bands_sum(args.P, t_grid, m=6) # print quick band magnitudes near first/last t for k in sorted(bands.keys()): mags = [abs(z) for z in bands[k]] print(f"k={k}: min={min(mags):.4f}, max={max(mags):.4f}") tag = f"primes_P{args.P}_{t_from:.2f}–{t_to:.2f}" fig = plot_bands(bands, t_grid, tag) if fig: print(f"bands plot -> {fig}") def cmd_spiral(args): svg = alpha137_prime_spiral_svg(P=args.P, width=args.width, height=args.height, scale_px=args.scale) out = os.path.join(OUT_DIR, f"alpha137_prime_spiral_P{args.P}.svg") with open(out, "w", encoding="utf-8") as f: f.write(svg) print(f"SVG -> {out}") def cmd_revert(args): snaps = list_autosaves() if not snaps: print("No autosaves found.") return # restore the newest autosave that matches a substring filter target = None if

args.filter: for s in reversed(snaps): if args.filter in os.path.basename(s): target = s break else: target = snaps[-1] if not target: print("No matching autosave.") return # restore over original path if metadata exists meta_path = target + ".json" if os.path.exists(meta_path): with open(meta_path, "r", encoding="utf-8") as f: meta = json.load(f) dst = meta.get("src") if dst: shutil.copy2(target, dst) print(f"Restored {dst} from {target}") return print(f"Snapshot file is at: {target} (no metadata to auto-restore)") # ---------- --- # Argument parsing # ---------- --- def build_parser() -> argparse.ArgumentParser: p = argparse.ArgumentParser(prog="all88_codex", description="ALL88 Codex Toolkit

```
(educational)") sub =
p.add_subparsers(dest="cmd",
required=True) e = sub.add_parser("enc",
help="encrypt a file (educational demo)")
e.add_argument("--input", "-i",
required=True) e.add_argument("--output","-
o", required=True) e.add_argument("--
passphrase","-p") e.add_argument("--ad",
default="") e.add_argument("--rounds",
type=int, default=7)
e.set_defaults(func=cmd_encrypt) d =
sub.add_parser("dec", help="decrypt a file
(educational demo)") d.add_argument("--
input", "-i", required=True)
d.add_argument("--output","-o",
required=True) d.add_argument("--
passphrase","-p") d.add_argument("--ad",
default="")
d.set_defaults(func=cmd_decrypt) t =
sub.add_parser("trace", help="produce
recursion trace JSON and spiral plot")
t.add_argument("--input", "-i", help="file to
```

```
sample (first N bytes). If omitted, use --literal") t.add_argument("--literal", help="fallback literal data", default="ALL88") t.add_argument("--sample-bytes", type=int, default=32) t.add_argument("--passphrase","-p", default="") t.add_argument("--ad", default="") t.add_argument("--depth", type=int, default=12) t.add_argument("--tag", default="trace") t.add_argument("--no-plots", action="store_true") t.set_defaults(func=cmd_trace) s = sub.add_parser("synth", help="synthesize WAV from a trace JSON") s.add_argument("--trace-json", required=True) s.add_argument("--tag", default="glyphsonic") s.add_argument("--seconds-per-depth", type=float, default=1.0) s.set_defaults(func=cmd_synth) a = sub.add_parser("analyze", help="Glyphfire: analyze a base36 string and emit plots/CSV/
```

```
WAV") a.add_argument("--values", "-v", required=True, help='e.g. "6EQUJ5"')
a.add_argument("--tag", default="signal")
a.set_defaults(func=cmd_analyze) pr =
sub.add_parser("primes", help="prime/ζ toy explorer") pr.add_argument("--P", type=int,
default=20000, help="max prime")
pr.add_argument("--t-from", type=float,
default=13.5) pr.add_argument("--t-to",
type=float, default=15.0)
pr.add_argument("--steps", type=int,
default=120)
pr.set_defaults(func=cmd_primes) sp =
sub.add_parser("spiral", help="export
Alpha137 prime-spiral SVG")
sp.add_argument("--P", type=int,
default=5000) sp.add_argument("--width",
type=int, default=1200)
sp.add_argument("--height", type=int,
default=1200) sp.add_argument("--scale",
type=float, default=2.5)
sp.set_defaults(func=cmd_spiral) rv =
```

sub.add_parser("revert", help="restore from newest autosave (or filter substring)") rv.add_argument("--filter", help="substring to match snapshot name") rv.set_defaults(func=cmd_revert) return p # ---------- --- # Entrypoint # ---------- --- def main(argv=None): argv = argv or sys.argv[1:] parser = build_parser() args = parser.parse_args(argv) args.func(args) if __name__ == "__main__": main()

quick-start cheatsheet

- save the code above as all88_codex.py.
- encrypt a file (edu demo):

python all88_codex.py enc -i in.bin -o out.worm -p "demo-pass" --ad "tag"

- decrypt:

python all88_codex.py dec -i out.worm -o recovered.bin -p "demo-pass" --ad "tag"

- make a recursion trace + spiral plot from the first 32 bytes of a file:

python all88_codex.py trace –i in.bin ––depth 12 ––tag zeta_probe

then synthesize a WAV from that trace:

python all88_codex.py synth ––trace–json ./codex/2024*_zeta_probe.json ––tag glyphsonic

- analyze a short base–36 string (plots + CSV + WAV in ./out/):

python all88_codex.py analyze –v 6EQUJ5 ––tag wow

- prime/ζ toy bands around the first zero:

python all88_codex.py primes ––P 20000 ––t–from 13.5 ––t–to 15.0 ––steps 200

- export the Alpha137 prime spiral as SVG:

python all88_codex.py spiral ––P 8000 ––width 1600 ––height 1600 ––scale 3.0

if you want this split into multiple files (cipher module + CLI + synth), say the word and I'll scaffold the package layout too.